Decentralised Realtime Peer-to-peer

Communication Protocol

Protokollbeschreibung Cy4NET-System

2019 / 2020 / 2021

Dokument Version 1.3

Copyright

Author:

Dipl. Inf. Reimar Grasbon

Hinweise, Vorschläge und Fehlerbenachrichtigungen:

- info@cy4net.org

Website des Projektes mit Dateien zu Cy4NET:

- www.cy4net.org

Inhalt

1 Einführung....5
2 Physikalische Ebene....7
3 Cy4NET Geräte....9
4 Paketzustellungen....11
5 Adressierung....12
6 Dienste....14
6.1 Aufbau....14
6.2 Adjazente Dienste....15
6.3 Pflichtdienste....16
6.4 Private Typennummern....20
7 Datagramme....21
7.1 Rundrufe....24
8 Zeitbetrachtungen....26
8.1 PDU-Umlaufdauer....26
8.2 Backoff....27
8.3 Queuing Delay....28
8.4 Round-Trip-Time....28
8.5 Interbyte Timeout....29
9 Interner Ablauf der Kommunikation....30
9.1 Zustandsautomaten....30
9.2 Die Kommunikationssequenzen....31
9.2.1 Die ausgehende Kommunikationssequenz....31
9.2.2 Die eingehende Kommunikationssequenz....34
9.2.3 Zustände und Ereignisse....36
9.2.4 Beispiel einer Kommunikationssequenz....38
10 Referenzimplementierung....42
10.1 Einleitung....42
10.2 Schichten....43
10.3 Implementierung....43
10.3.1 Die Hauptschleife....43
10.3.2 Das BOARD-Modul....44
10.3.3 Das Cy4NET-Modul....49

10.3.3.1 Der Kern....50
10.3.3.2 Die Schale....53
10.3.4 Die API....54
10.3.4.1 Die Handhabung von Diensten....54
10.3.4.2 Anfragen und Antworten....58
11 Zugang zum Netz....65
11.1 Das Gateway....65
11.2 Das Gateway Datagramm....67
11.3 ASCII-PDU....68
11.4 Busmonitor....70
11.5 cy4cmd....71
12 Firmware....76
12.1 Das Programmladermodul A00001....76
12.1.1 Der Typcode....78
12.1.2 Signalisierung....81
12.1.3 Rettung in den Programmlader....81
13 Inbetriebnahme eines Cy4NET-Gerätes....83
13.1 Programmierung am Beispiel des Gateways....83
13.1.1 Microchip-Studio-IDE....84
13.1.2 AVR-Dude....88
14 Anwendungsbeispiel....89
14.1 Ausgangssituation....89
14.2 Vorbereitung....90
14.3 Konfiguration....91
Anhang A: Einteilung des Dienst-Speicherraumes....94
Anhang B: Nutzung der Rundrufzone Exxxxxh....95
Anhang C: CRC Ermittlung....98
Anhang D: B00120 Cy4Nut....100
Anhang E: B00023 Cy4-To-Serial Gateway....102
Anhang F: Dienste der Gateway-Anwendung A00111....106
Anhang G: B00101 Evaluierungsboard ATmega32 3M....112
Konventionen im Cy4NET....116
Literatur und Bezugsquellen....117

1 Einführung

Die Grundlage jeder Interaktion bildet der Austausch von Informationen. Das ist im gesellschaftlichen Umfeld nicht anders als im technischen. Voraussetzung dafür ist das Vorhandensein eines Mediums, das sich alle Kommunikationspartner teilen und das sie verbindet. Das Medium allein reicht allerdings nicht aus. Um Informationen austauschen zu können, um sie zu verstehen und gleich zu interpretieren, bedarf es darüber hinaus einer methodischen Infrastruktur, einer Sprache. Erst durch sie wird ein Verbund zu einem Kommunikationsverbund.

Diese Situation lässt sich unmittelbar auf informationstechnische Bereiche übertragen. Das reine Vorhandensein eines Mediums, ob Kupfer, Licht, Funk oder auch eines anderen, bildet zwar die Grundlage für den Informationsaustausch. Jedoch bedarf es einer übergeordneten Datenstruktur, die den Austausch von Informationen und ihre Verarbeitung erst möglich werden lässt. Bei vernetzten Computersystemen übernimmt diese Aufgabe das Protokoll.

Das im Folgenden vorgestellte Cy4NET-Projekt (gesprochen *ßei-vier-Net*) ist im Wesentlichen ein solches Protokoll. Es wurde geschaffen, um unterschiedliche Geräte, Aktoren, Sensoren oder allgemein Komponenten verschiedenster Gattung in einem Kommunikationsverbund zu organisieren.

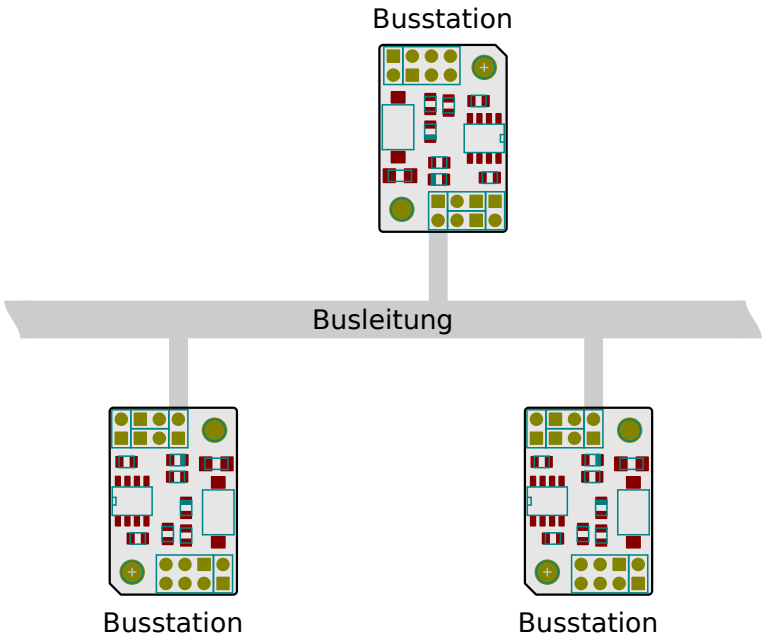


Abbildung 1: Topologie des Bussystems

Die Topologie des Kommunikationsverbundes ist die Busstruktur, d.h. alle Teilnehmer des Verbundes, die sogenannten Busknoten oder Busstationen, teilen sich als gemeinsames Kommunikationsmedium die Busleitung. Abbildung 1 zeigt den strukturellen Aufbau des Bussystems. Das Cy4NET-System fällt in die Kategorie der Feldbusse.

Protokolle für Feldbussysteme gibt es bereits einige. Die Motivation, ein weiteres zu schaffen, bestand jedoch in der Zielsetzung. Schwerpunkt bei der Entwicklung des Cy4NET-Systems wurde maßgeblich durch drei Faktoren bestimmt:

1. **Ressourcenschonend:** Um die Anforderungen an die Kommunikationspartner möglichst gering zu halten, sollte das Protokoll auf einfacher Hardware lauffähig sein. Basis für die Entwicklung waren Mikrocontroller aus der AVM-Reihe. Auf MCUs dieser Familie kommt das Cy4NET-Kernprotokoll mit gut 5 KB Programmspeicher (Flash) und mit knapp 600 Bytes statischem Datenspeicher (SRAM) aus. Mit diesen Anforderungen wäre es möglich, das Cy4NET-Protokoll sogar auf einem ATmega 8 zu betreiben. Berücksichtigt man jedoch das Vorhandensein eines Programmladermoduls, das häufig mit weiteren 2K zu Buche schlägt, sind Controller ab dem ATmega168 die bessere Wahl.
2. **Multimaster:** Alle Knoten des Bussystems sollen ohne zentrale Steuerungsinstanz miteinander kommunizieren können, also Peer-To-Peer. Jeder Teilnehmer des Netzes kann direkt mit jedem anderen Teilnehmer in Kommunikation treten, die Funktionalität des Netzes ist damit dezentral.
3. **Echtzeitfähigkeit** Die Kommunikation im Cy4NET ist paketorientiert. Die Umlaufzeit eines Paketes, die so genannte Round-Trip-Time, beträgt dabei maximal 500 ms. Spätestens nach Ablauf dieser Zeitgrenze weiß ein Sender, ob seine Anfrage erfolgreich war oder nicht, d.h. das Protokoll ist echtzeitfähig.

2 Physikalische Ebene

Ressourcenschonung ist einer der zentralen Punkte des Cy4NET-Systems. Diese Tatsache bedingt, dass der Bus kein Hochgeschwindigkeitsbus sein kann. Die Datenbytes werden seriell mit einer Symbolrate von 19200 Bd übertragen. Die Übertragung findet asynchron statt, d.h. es gibt keine Taktleitung. Wie bei RS232 [1] wird die Synchronisation mittels Start- und Stoppbits durchgeführt. Das Startbit ist logisch 0, das Stoppbit logisch 1.

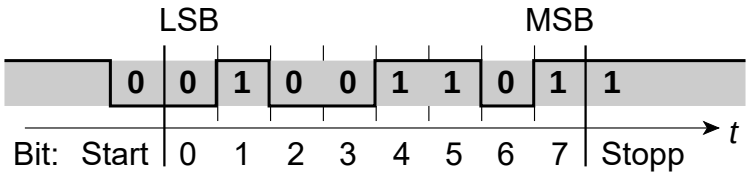


Abbildung 2: Serielle Bitübertragung des Wertes 178 (10110010) mit 8N1

Zwischen Start- und Stoppbit werden 8 Datenbit, beginnend mit dem niederwertigsten Bit (LSB first), übertragen. Ein Paritätsbit oder eine Zustandswechselschrift (RZ-Code) ist nicht vorgesehen. Bei RS232-Verbindungen heißt dieses Framing-Format 8N1, NRZ. Die Nettoübertragungsrate liegt bei ca. 19200÷10 = 1920 Bytes pro Sekunde oder gut 520 µs pro Byte. Es ist kein Zufall, dass die Buskommunikation auf Basis der UARTs in den Controllern stattfindet. Im Gegensatz zur Vollduplex-Übertragung der RS232 findet die Kommunikation auf dem Bus im Halbduplexverfahren statt, also im Wechselbetrieb. Das dezentrale, asynchrone Verfahren zur Buskommunikation führt im Multimasterbetrieb unweigerlich zu Kollisionen. Senden mehrere Knoten zur gleichen Zeit, kommt bei Mehrfachzugriff das Carrier Sense Multiple Access Verfahren, CSMA zum Einsatz [2]. Jeder sendende Knoten muss prüfen, ob seine Daten mit denen auf dem Bus übereinstimmen (Echo-Prüfung). Senden zeitgleich mehrere Knoten, erscheint je nach Dominanz der Zustände, eine Mixtur der Daten und Synchronisationsmarken auf dem Bus. Ist die Information auf dem Bus zerstört, müssen alle beteiligten Knoten ihre Übertragung unmittelbar beenden. Die physikalische Ebene (OSI-Schicht 1 [3]) muss daher in der Lage sein, Kollisionszustände handhaben zu können. Welches Verfahren auf dieser Ebene letztendlich zum Einsatz kommt ist unerheblich, insofern es kollisionstauglich ist und alle Teilnehmer im Cy4NET die Signalpegel gleich interpretieren. EIA-485 (auch bekannt als RS485) eignet sich beispielsweise für diese Aufgabe nicht, da in der Regel die Schnittstellentreiber bei widersprüchlichen Buszuständen gegeneinander arbeiten.

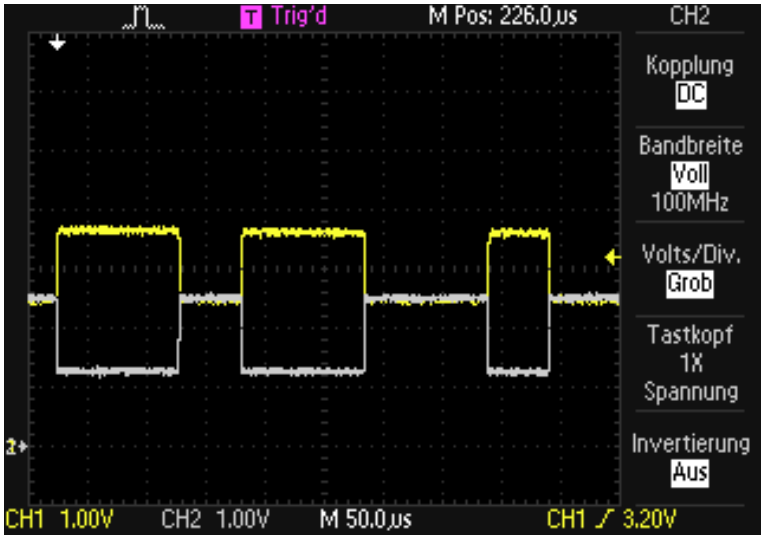


Abbildung 3: Wert 178 (10110010) als differentielles CAN-Signalpegelpaar

Bei bereits im Einsatz befindlichen Cy4NET-Bussen kommt ein differentielles Übertragungsverfahren zum Einsatz, dass mit Spannungspegeln wie denen des CAN-Bussystems [4] arbeitet, d.h. 1,5 V – 3,5 V. Abbildung 3 zeigt das Oszillogramm des Wertes 178 als differentielles Signalpaar.

3 Cy4NET Geräte

Cy4NET-Geräte sind als eine Einheit aus Hardware- und Softwareplattform zu betrachten. Die Hardware besteht üblicherweise aus einer Leiterplatte, die als Mindestausstattung die Elektronik für die Buskommunikation sowie eine Rechnereinheit bereitstellt. Welche zusätzlichen Eigenschaften, welche Aktorik oder Sensorik die Hardwareplattfom zusätzlich noch bietet, hängt gänzlich von der Art des Gerätes ab. Im Cy4NET-System wird die Hardwareplattform als das Board bezeichnet. Das Board stellt, ganz allgemein, bestimmte Funktionen zur Verfügung. Ob es dabei auf eine bestimmte Aufgabe spezialisiert ist oder eher allgemeine Funktionen erfüllt, ist unerheblich. Ein Barometer misst üblicherweise nur den Luftdruck, wohingegen ein Schaltaktor einen Beleuchtungskörper ebenso wie eine Aquariumpumpe schalten kann.

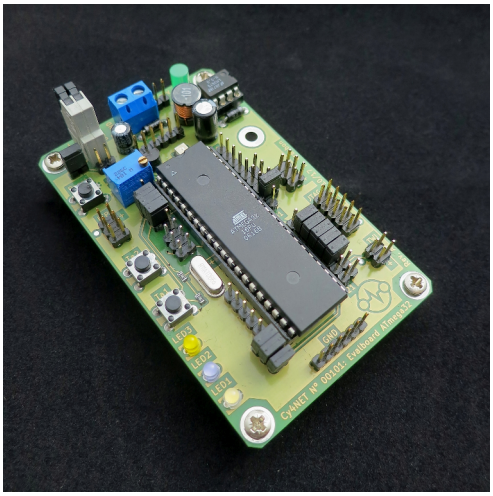


Abbildung 4: Evalboard B00101, Rev A

Jedes Board, genauer gesagt jeder Boardtyp hat im Cy4NET eine eindeutige Nummer. Durch diese Nummer ist die exakte Funktionalität der Hardware spezifiziert. Wird durch eine Modifikation die Funktionalität des Boards verändert, ändert das den Typ und damit auch die Boardtypennummer.

Was die Nummer hingegen nicht zwingend festlegt, ist die Zusammenstellung und das Zusammenwirken der elektronischen Komponenten. Werden Anpassungen durchgeführt, die an der bereits vorhandenen Funktionalität nichts ändern, werden Leiterbahnen anders verlegt oder Bauteile gegen Ersatztypen getauscht, zieht das nicht unbedingt eine Änderung der Nummer nach sich. Daher werden Modifikationen an den Boards, im Gegenzug zu Software-Versionen, hier als Revisionen bezeichnet und mit einem Buchstaben von A bis Z als „Zählweise“ versehen.

Die Boardtypennummer ist eine maximal fünfstellige Zahl. Sie kann Werte von 1 bis 99999 annehmen. 0 ist nicht erlaubt. Als Kennzeichnung für eine Board-Nummer wird ihr häufig der Buchstabe ‚B‘ vorangestellt. Abbildung 4 zeigt beispielsweise Revision A des Boards B00101. Das Board ist in Anhang G beschrieben.

Das Pendant zur Boardtypennummer der Hardwareplattform ist die Applikationstypennummer der Softwareplattform. Die Software (bei Controllersystemen häufig auch als Firmware von *firm* = engl. *fest* bezeichnet) beinhaltet neben den betriebsystemtechnischen Aufgaben wie z.B. der Schnittstelle zur Hardware oder dem Speichermanagement auch die Applikation, also die Anwendung. Erst durch sie werden Funktionalitäten des Boards nach außen getragen und können dort genutzt werden, d.h. die Applikation exportiert Boardfunktionen. Ob z.B. der Schaltaktor im Beispiel oben nur einen Lichtschalter darstellt oder eine aufwendige Zeitschaltuhr realisiert, hängt gänzlich von der Art der Applikation ab. Möchte man einem Gerät eine andere oder neue Funktionalität verleihen, muss eine Applikation anderen Typs oder neuerer Version auf das Board gebracht werden.

Wie bei den Boards kann die Applikationstypennummer ebenfalls Werte von 1 bis 99999 annehmen. Um den Unterschied zur Boardtypennummer zu kennzeichnen wird sie mit einem vorangestelltem ‚A' gekennzeichnet, Beispiel A04321.

Zu einer Applikation gehört immer einer Versionsangabe. Die Applikationsversion besteht aus zwei dezimalen Ziffern, bei denen die erste die Hauptversionsnummer (Majornumber), die zweite die Unternummer (Minornumber) der Anwendung beschreibt. Dargestellt wird die Versionsnummer in der üblichen Form Major.Minor, gegebenenfalls mit führendem ‚V' . Auch hier ist 0.0 nicht erlaubt, so dass sich die Anwendungsversionen über 99 Stufen von V0.1 bis V9.9 erstrecken können.

Cy4NET-Geräte sind also stets eine Kombination aus Board und Applikation. Ein Boardtyp kann zwar Arbeitsgrundlage verschiedener Applikationstypen sein und ein Applikationstyp kann in verschiedenen Ausprägungen für unterschiedliche Boards existieren, doch letztlich spezifiziert immer die konkrete Kombination aus beiden Einheiten das Gerät. Dazu siehe auch: Der Typcode, Kapitel 12.1.1.

4 Paketzustellungen

Wie eingangs erwähnt, ist das Verfahren zur Datenübertragung im Cy4NET-System die Paketvermittlung. Details zum Aufbau der Datenpakete finden sich weiter unten. Diese Datagramme genannten Pakete erfolgen stets in der Abfolge Anfrage – Antwort. Der Sender verpackt seine Anfrage in ein Anfrage-Datagramm und schickt es dem Empfänger. Dieser nimmt die Anfrage entgegen, bearbeitet sie und schickt seine Antwort wieder in Form eines Antwort-Datagramms an den Sender zurück. Existiert der adressierte Empfänger nicht oder ist die Anfrage, aus welchem Grund auch immer, für den Empfänger nicht beantwortbar, wird keine Antwort gesendet. Das bedeutet: Es gibt keine Fehler-Pakete oder anderweitige protokollspezifischen Fehlerrückmeldungen. Tritt während der Kommunikation ein Fehler auf, bleibt die Anfrage unbeantwortet. Folge davon ist, dass der Sender nach der oben erwähnten *Round-Trip-Time* weiß, dass seine Anfrage negativ beantwortet ist. Aus was genau sich Datagramme im Cy4NET zusammensetzen und wie sie aufgebaut sind, erschließt sich aus den folgenden Kapiteln.

5 Adressierung

Im Cy4NET können die Gerät direkt miteinander kommunizieren. Damit das funktioniert, muss jedes Gerät durch eine eindeutige Adresse identifizierbar sein. Eine Cy4NET-Adresse besteht aus zwei Komponenten, mit jeweils einem Byte. Das erste Byte ist die Netzadresse (`NET`), das zweite die Geräteadresse (`DEV`).

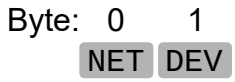


Abbildung 5: Die Komponenten der Cy4NET-Addresse

Nominell würde damit der Adressraum des Netzwerks 2·8 = 16 Bit umfassen. Doch ein Netzwerk mit mehreren tausend Kommunikationsstationen würde neben der Bandbreite auch die elektrischen Eigenschaften des Verbundes überfordern. Sinn der Unterteilung in Netz- und Geräteadresse besteht darin, Busknoten gruppieren zu können. Das können räumliche Verteilungen sein wie auch Funktionsverbände. Gegebenenfalls können ganze Teilnetze durch Router separiert werden um so die Buslast bezogen auf die Signalpegel der Mitglieder (vergleichsweise dem Fan-Out) wie auch den Datenverkehr (Traffic), im Zaum zu halten. Die Separation des Adressraumes ermöglicht es, die Knoten des Netzes in einer zweistufigen Baumstruktur zu organisieren. Sie kann real oder auch virtuell sein.

Angelehnt an die Schreibweise einer Hostadresse in IP-Netzwerken, wird bei einer Cy4NET-Adresse die Netz- und die Geräteadresse mit einem Punkt getrennt dargestellt, Beispiel `165.12`.

Bei der Wahl der Adressen gibt es Einschränkungen. So ist Adresse `0` generell für Rundrufe (Broadcasts) reserviert. Das gilt sowohl für Netz- als auch für Geräteadressen. Es ist also möglich Netz- wie auch Gerätegruppen gemeinschaftlich eine Nachricht zukommen zu lassen. Folglich ist `0.0` die Adresse, die die Gesamtheit aller Busstationen im Netz adressiert.

Des Weiteren gibt es zwei Konventionen. Konvention 1 betrifft Adresse `255.255`. Die letzte Adresse im Adressraum ist allgemein die Ausprägung für eine ungültige Adresse und sollte daher an Busstationen nicht vergeben werden.

Konvention 2 und 3 betreffen die letzte Netzadresse: `255.` Man beachte, hier geht es um den Adressbereich den diese Netzadresse aufspannt, also um alle 254 Cy4NET-Adressen von `255.1` bis `255.254`.

Da es im Cy4NET keinen übergeordneten Mechanismus zur automatischen Vergabe von Adressen gibt, haben die Busstationen zu Anfang ihre werksvoreingestellte Adresse. Bei Inbetriebnahme muss dann sichergestellt werden, dass Adressen nicht mehrfach vergeben sind. Insofern kein explizites Adressmanagement, beispielsweise mit Hilfe von DIP-Schaltern, die Eindeutigkeit der Adressen sicherstellen, hören Geräte gleichartiger Bauart und gleicher Firmware anfangs auf die gleiche Adresse. Ein solches Szenario kann nur verhindert werden, indem Busstationen sukzessi-

ve ans Netz gelassen und dabei ihre Adressen eindeutig verteilt werden. Generell ist es gute Praxis, Adressen einer Struktur folgend auf die Geräte zu verteilen. Um einen Wildwuchs bei der Erstinbetriebnahme zu vermeiden, regelt Konvention 2 die Vergabe der werksvoreingestellten Cy4NET-Addresse, so dass sie stets im Bereich von `255.100` bis `255.199` liegen. Das erleichtert das Auffinden neuer Gerät um sie danach auf ihren finalen Ort versetzen zu können. Gemäß Konvention 2 hat das Entwicklungsboard in Abbildung 4 z.B. die werksvoreingestellte Adresse `255.101`. Dokumentiert ist das Hardwareprojekt in Anhang G.

6 Dienste

In jedem Verbundsystem müssen Komponenten miteinander in Interaktion treten. Die Teilnehmer müssen untereinander wissen, was der andere kann oder leistet. Das ist auch im Cy4NET so. Aber Funktionalitäten, die von Geräten zu Verfügung gestellt oder in Anspruch genommen werden, sind strukturell nur schwer zu erfassen. Der Versuch, häufig vorkommende Aktionen unter den Busstationen mittels vorgefertigter Funktionalitäten abzubilden führt letztlich zu einer Ansammlung unspezifischer Anweisungen, deren Vielzahl durch ihre Ausnahmen bestimmt wird. Jedes Gerät hat Spezialitäten, deren Nutzung eine gerätespezifische Erweiterung des existierenden Funktionssatzes nötig werden lässt. Die Folge ist, dass die Liste möglicher Aktivitäten oder Aufgaben mit der Zeit anwächst und sich ständig erweitert. Der so entstehende Wildwuchs an diversen Funktionen macht deren Abgrenzung von oder Zuordnung zu bestimmten Aufgaben immer unübersichtlicher und erschwert eine einheitliche Betrachtung des Kommunikationsverbundes. Grundsätzlich lässt sich das Problem nicht vollständig lösen, es ist eine Frage der Abstraktion.

Beim Cy4NET wird aus diesem Grund eine Zwischenebene definiert. Sie besteht aus einer Low-Level-Abbildung von Funktionalitäten und versucht damit der oben genannten Diversifizierung entgegenwirkt. Umgesetzt ist diese Ebene durch das Prinzip der Dienste (Services).

6.1 Aufbau

Ein Dienst besteht aus drei Teilen: Dem Dienstort (Location), der Dienstbreite (Size) und dem Zugriffsrecht (Access). Der Dienstort ist nichts anderes als eine Speicheradresse. Sie gibt an, wo in einem virtuellen Adressraum die Bytes des Dienstes zu finden sind. Die Dienstbreite beschreibt die Anzahl der Bytes und das Zugriffsrecht regelt, wie auf den Speicherbereich zugegriffen werden darf: Lesend, schreibend oder beides.

Geräte stellen Dienste bereit und nutzen Dienste anderer. Alle Aktivitäten im Cy4NET bestehen aus der Nutzung dieser Dienste, d.h. alle Aktionen und Reaktionen müssen in Diensten darstellbar sein. Konkret bedeutet es, dass alles was ein Gerät kann oder anbietet, auf den Speicherbereich eines oder mehrerer Dienste abgebildet werden muss. Die Funktionalitäten sind quasi *memory mapped*. Dieses Prinzip ist angelehnt an das Modbus-Protokoll [5], wo Speicherregister gelesen oder beschrieben werden und die Funktionen des Gerätes dort abgebildet werden. Beim Modbus-Protokoll sind die Register in der Regel 16-Bit breite Datenworte auf die mit einer Vielzahl von Funktionen zugegriffen werden kann. Ähnlich ist das auch beim Cy4NET, allerdings bestehen die Speicherbereiche hier aus 8-Bit-Werten, also Bytes, und es gibt genau zwei Funktionen auf den Dienst: Speicherzellen lesen oder Speicherzellen beschreiben.

Wie bei der Cy4NET-Addresse gibt es auch hier eine Notation, die die Signatur eines Dienstes beschreibt. Alle drei Teile werden dabei von einen Doppelpunkt getrennt, Beispiel: `100A00h:12:RW`. Da der Dienstort eine Speicheradresse darstellt, wird er üblicherweise hexadezimal angegeben (zu erkennen am abschließenden `h`). Er kann Werte von `0h` bis `FFFFFEh` annehmen und belegt damit

exakt 24 Bit oder 3 Bytes. Dienstort FFFFFFh ist reserviert. Ähnlich wie bei den Adressen, ist FFFFFFh die Wertausprägung für einen ungültigen Dienstort. Diese Festlegung ist Konvention 4.

Die Breite des Dienstes, also die Größe des Speicherstücks, kann theoretisch den kompletten Adressraum umfassen - zumindest fast. Das wird aber in der Realität nur in Ausnahmefällen eingesetzt. Die breitesten Dienste finden sich aktuell in Programmlader-Anwendungen, bei denen der komplette Programmspeicher eines Gerätes als ein großer Dienst zur Verfügung gestellt wird. Normale Dienste erstrecken sich üblicherweise nur über wenige Bytes. Eine Dienstbreite von 0 ist nicht erlaubt, Minimum ein Byte muss ein Dienst belegen. Das Zugriffsrecht kann RO, nur lesbar, WO nur beschreibbar oder, wie oben RW also schreib und lesbar sein.

Beispiel: Ein Cy4NET-Gerät bietet folgende zwei Dienste:

Luftdruck:	100210h:2:RO
Anwendung (neu) starten:	FFFFF8h:1:WO

Ein Gerät ist mit einem Barometer-Sensor ausgestattet und stellt anderen Geräten den gemessenen Luftdruck als Dienst ab 100210h zur Verfügung. Der Druck liegt im Bereich von 300 hPa bis 1100 hPa. Um ihn darstellen zu können sind mindestens 2 Byte nötig, d.h. die Breite des Dienstes beträgt 2. Der Luftdruck ist verständlicherweise nur lesbar. Damit lautet die Signatur dieses Dienstes: 100210h:2:RO.

Der zweite Dienst ist ein Auslöser. Er ist nur beschreibbar. Die Funktion ist vergleichbar mit dem eines Strobe-Registers. Wird auf sein Byte am Dienstort FFFFF8h der Wert 94 (5Eh) geschrieben, löst das im Gerät einen internen Reset aus und die Anwendung startet neu. Der Wert 94 ist kein Zufall. Er entspricht dem ASCII-Code des Zeichens ‚^' dessen zugehörige Taste ganz oben links auf gängigen Tastatur zu finden ist. Gemäß Konvention 5 ist 94 im Cy4NET der allgemeine Wert zur Auslösung einfacher Ereignisse.

6.2 Adjazente Dienste

Dienste dürfen sich, bezogen auf ein Gerät, nicht überlappen, d.h. mengentheoretisch betrachtet müssen ihre Speicherbereiche disjunkt sein. Aber Dienste können angrenzend liegen. Der Luftdruckwert oben wird im Format Highbyte/Lowbyte in den beiden Bytes bereitgestellt. Nebenbei bemerkt, ist das Konvention 6 im Cy4NET: Die Byte-Reihenfolge bei Mehrbytewerten ist Big-Endian. Die zwei Bytes müssen aber nicht zwingend in einem Zugriff, also auf einmal, gelesen werden. Generell ist es erlaubt, Dienste auch stückweise zu nutzen. Partielle Zugriffe, lesend wie schreibend sind möglich. Die Bytes des Dienstespeichers können in Gruppen oder auch einzeln angesprochen werden. Dabei dient der Dienstort als Speicheradresse. Im Beispiel oben könnte man daher am Dienstort 100210h das Highbyte und an 100211h das Lowbyte des Luftdruckwertes einzeln erfahren. Umgekehrt gilt das ebenso. Liegen zwei Dienste direkt adjazent, können ihre Werte in einem einzigen Zugriff, also dienstübergreifend, gelesen oder geschrieben werden. Möchte man das verhindern, muss zwischen den betroffenen Speicherbereichen eine Lücke gelassen werden. Ein

einzelnes Byte reicht bereits, damit Dienste nicht mehr adjazent liegen und damit der gemeinschaftliche Zugriff verwehrt ist. Mitunter ist das sogar gefordert, denn konzeptionell sind adjazente Dienste nur erlaubt, wenn sie über gleiche Zugriffsrechte verfügen. Würden Dienste mit unterschiedlichen Zugriffsrechten adjazent liegen, wäre bei gemeinschaftlicher Nutzung eine Zugriffsverletzung nicht ausgeschlossen. Lücken zu lassen ist problemlos möglich, da Dienstbereiche einen virtuellen Speicherbereich darstellen, dessen 16 MiB Adressraum üblicherweise weit über den Bedarf normaler Busstationen hinausgeht.

Die Einteilung in Dienste ist also eher als semantische Struktur zu verstehen. Ob es besser ist, Dienste zu partitionieren oder zu Größeren zu vereinigen entscheidet sich je nach Geschmack und Gegebenheit. Ob partielle Zugriffe auf einen Dienst sinnvoll sind oder nicht, liegt nicht im Ermessen des Protokolls, die Firmware der Geräte muss es in vernünftiger Weise handhaben.

6.3 Pflichtdienste

Busstationen sind in der Festlegung der von ihnen angebotenen Dienste frei. Doch auf falsch angefragte Dienste gibt es von den Busstationen keine Reaktion. Ohne also konkretes Wissen über das Gerät bzw. ohne Kenntnisse über seine Dienste, ist es aussichtslos festzustellen, ob ein Gerät an einer bestimmten Adresse online ist. Was fehlt ist eine allgemeine Methode um mit Geräten in Kontakt treten zu können.

Da Geräte ja nur auf bekannte Dienste antworten ist es nötig Dienste zu definieren, die unabhängig von der Art des jeweiligen Gerätes, übergeordnet zur Verfügung stehen. Daher verlangt das Cy4NET-Protokoll, dass jedes Gerät mindestens vier Dienste anbietet. Sie sind für alle Geräte verpflichtend und lauten:

Typenschild abrufen (CallTypePlate):	`FFFFF0h:7:RO`
Anwendung starten (EnterApplication):	`FFFFF8h:1:WO`
Programmlader starten (EnterProgramLoader):	`FFFFFAh:1:WO`
Cy4-Adresse wechseln (ChangeCy4Address):	`FFFFFCh:2:WO`

Die Dienstbereiche sind am Ende des Dienstspeicher-Adressraumes ab `FFFFF0h` angesiedelt.

Der erste Pflichtdienst dient dazu ein virtuelles Typenschild zu erfragen. Wie oben bereits erwähnt, identifiziert sich ein Gerät durch seinen Board- und seinen Applikationstyp. Die Typnummern beider Komponenten lassen sich mit diesem Dienst in Erfahrung bringen. Ebenfalls darin enthalten: Die Versionsnummer der Applikation. Im Gegensatz zur Entwicklungsstufe der Software, wird die der Hardwareplattform nicht mitgegeben. Der Grund dafür ist, dass für die Firmware aus sich heraus keine Möglichkeit besteht, die zugrundeliegende Boardrevision zu erfahren. Folglich ist die Revisionierung der Hardware auf externe Maßnahmen angewiesen, z.B. als Beschriftung auf dem PCB.

Der Dienst *CallTypePlate* ist nur lesbar, umfasst sieben Bytes und kodiert die Informationen des Typenschildes in einem genau festgelegten Schema, siehe Abbildung 6.

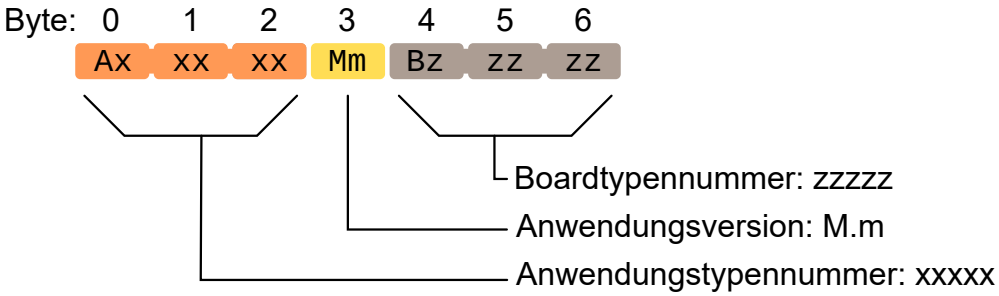


Abbildung 6: Typenschild, Belegung der 7 Bytes

Die ersten 3 Bytes repräsentieren die Applikationstypennummer. Im 4. Byte liegt die Anwendungsversion und in den restlichen 3 Bytes ist die Boardtypennummer untergebracht. Die Byte-Reihenfolge aller Einträge ist gemäß Konvention 6 *Big-Endian*. Die Einträge der Typennummern führen im obersten Nibble ihre Typ-Präfixe, also Ah bzw. Bh. Die restlichen Ziffern sind BCD kodiert, d.h. die ‚x' und ‚z'-Werte sind als hexadezimal geschriebene Dezimalzahlen zu interpretieren. Die Versionsnummer hat keinen Präfix und ist direkt als zweistellige BCD-Zahl abgelegt. Beispiel:

```
A0 01 23 45 B0 67 89
```

Applikationstypennummer:123	(oder A00123)
Anwendungsversion: 45	(oder V4.5)
Boardtypennummer: 6789	(oder B06789)

Auf diese Weise bietet *CallTypePlate* nicht nur Möglichkeit festzustellen, ob an einer bestimmten Zieladresse ein Gerät wartet. Ebenso kann in Erfahrung gebracht werden, welches Board mit welcher Applikation dort existiert. Es liegt auf der Hand, dass abstrakte Nummern allein die Identifikation eines Gerätes nicht unbedingt erleichtern. Das zu ändern liegt jedoch nicht im Aufgabenbereich der Pflichtdienste. Das Thema wird weiter unten noch einmal aufgegriffen.

Die nächsten beiden Pflichtdienste haben mit der Fähigkeit zu tun Geräte über das Cy4NET-System mit einer neuen Firmware zu versorgen. Dazu muss man wissen, dass auf Cy4-Geräten im Grunde zwei Cy4-Anwendungen schlummern. Die Erste ist die Hauptapplikation, also die offizielle Anwendung die das eigentliche Gerät ausmacht. Die Zweite liegt etwas im Verborgenen. Es ist die Programmlader-Applikation. Sie verhält sich wie eine ganz normale Cy4NET-Applikation, ist aber minimalistisch. Neben den vier obligatorischen Pflichtdiensten stellt der Programmlader einen spezifischen Satz an Diensten bereit, mittels derer die Firmware des Gerätes, genauer gesagt, die Hauptapplikation, ausgetauscht werden kann. Wie und auf welche Weise das geschieht, ist nicht durch das Protokoll festgelegt sondern programmladerspezifisch.

Um nun im laufenden Betrieb zwischen diesen beiden Applikationsarten wechseln zu können, dienen die beiden Dienste *EnterProgramLoader* und *EnterApplication*. Beide Dienste sind nur beschreibbar und Auslöser gemäß Konvention 5. Das Auslösen von *EnterProgramLoader* hat zur Fol-

ge, dass die aktuell laufende Applikation beendet wird und die Programmlader-Anwendung startet. Programmlader haben eine eigene Applikationstypennummer, die laut Konvention 7 im Nummernbereich von 1 bis 99 liegt. Durch Abfrage des Typenschildes kann damit klar festgestellt werden, ob es sich bei der aktuell laufenden Anwendung um ein Programmlader handelt oder um die eigentliche Applikation. 99 verschiedene Applikationstypen sind zwar verhältnismäßig wenig, jedoch wird in der Realität die Variantenvielfalt der Programmlader nicht zu vielseitig ausfallen. Der bereits existierende Programmlader hat die Anwendungstypennummer A00001 und ist Thema von Kapitel 12.1. Programmlader sind üblicherweise sehr eng mit dem Board verbunden und benötigen, um ausgetauscht oder geändert zu werden, meistens weitergehende externe Maßnahmen wie z.B. einen Programmieradapter. Wie eine Inbetriebnahme stattfinden kann, zeigt Kapitel 13.

Durch Nutzung der Programmlader-Dienste kann Stückweise der Programmspeicher der Recheneinheit beschrieben und damit eine neue Applikation auf das Board übertragen werden. Ist der Vorgang abgeschlossen, kommt der zweite Dienst, *EnterApplication*, zum Einsatz. Er fand im Beispiel oben bereits Erwähnung und startet die eigentliche Anwendung. Es ist zwar nicht zwingend der Fall, doch häufig geht der Applikationswechsel mit einem Reset der Rechensystems einher. Natürlich ist es erlaubt, auch die aktuell laufende Anwendung mittels des entsprechenden Dienstes selbst neu zu starten, d.h. der Wechsel zwischen den Betriebsmodi ist möglich, aber keineswegs ein Muss. Bei der Bestückung des Controllers mit einer neuen Applikation ist darauf zu achten, dass die neue Anwendung komplett im Speicher des Controllers vorliegt, bevor *EnterApplication* aufgerufen wird. Der verfrühte Start einer lückenhaft vorhandenen Anwendung führt in den meisten Fallen zum Absturz. Kapitel 12 greift dieses Thema auf.

Letztlich gibt es auch Geräte bzw. Boards, die technisch nicht in der Lage sind, ihren Speicher selbst neu zu beschreiben. Diese Geräte verfügen damit auch über keinen Programmlader. Trotzdem muss das Gerät alle vier Pflichtdienste anbieten. Hat das Gerät keinen Programmlader, führt die Auslösung von *EnterProgramLoader* dann dazu, dass die eigentliche Anwendung neu startet, d.h. *EnterProgramLoader* verhält sich in dem Fall wie *EnterApplication*.

Man beachte: Eine positive Antwort auf *EnterProgramLoader* bzw. *EnterApplication* sagt nichts darüber aus, ob der Wechsel vollzogen wurde, geschweige denn ob ein Programmlader bzw. eine Applikation überhaupt existiert. Eine positive Antwort bedeutet lediglich, dass das Gerät die Anforderung zum Wechsel erhalten hat.

Im Zusammenhang mit den Dienstorten der Applikationen gibt noch eine wichtige Einschränkung: Da Programmlader den Programmspeicher der Rechnereinheit üblicherweise in direkter Weise über ihre Dienste ansprechen, ist ihr Dienstort auf den Bereich von `000000h` bis `0FFFFFh` eingeschränkt. Die Pflichtdienste sind davon ausgenommen. Damit steht für den Programmierprozess theoretisch eine Speichergröße von 1 MiB zur Verfügung. Sollte dennoch mehr übertragen werden, ist abzuwägen, ob die Datenübertragung wirklich über das Cy4NET-System stattfinden sollte oder ob vielleicht ein anderer Weg die bessere Alternative ist.

Im Gegenzug dazu dürfen Dienstorte der eigentlichen Applikation diesen Bereich nicht nutzen, er ist für sie verboten. Für sie beginnt der erlaubte Bereich ab `100000h`. Hintergrund dieser Separation ist, sicherzustellen dass Dienstortbereiche des Programmladers und der Applikation nicht mehrfach belegt werden. Andernfalls würden Dienste zur Programmierung und normale Betriebsdienste

sich in ihren Schreibzugriffen gegenseitig stören bzw. zerstören. Diese Aufteilung ist Teil des Protokolls, es ist keine Konvention.

Der letzte Pflichtdienst, *ChangeCy4Address*, behandelt die Cy4NET-Adresse des betrachteten Gerätes. Wie in Kapitel 5 erwähnt, sollten Geräte bei Inbetriebnahme von ihren werksvoreingestellten Geräteadressen auf andere, freie Bereiche verlegt werden.

Die Dienstbreite hier beträgt zwei Bytes. Das erste Byte nimmt die Netz-, das zweite die Geräteadresse der neuen Zieladresse auf. *ChangeCy4Address* ist nur beschreibbar. Das Auslesen der Cy4NET-Adresse sinnlos, da bei Zugriff auf den Dienst die entsprechende Adresse ja ohnehin bekannt sein muss. Gemäß Konvention 1 ist Adresse `255.255` unerwünscht. Verboten hingegen sind Rundruf-Adressen, also `0` als Netz- oder Geräteadresse. Die Bestätigung der Änderungsanforderung sendet das Gerät noch unter seiner alten Adresse zurück. Von da ab ist es unter seiner neuen Adresse zu erreichen.

> **Hinweis:** Vor einer Adressänderung sollte auf jeden Fall sondiert werden, ob die gewünschte Zieladresse nicht bereits durch ein anderes Gerät belegt ist. Erhalten zwei oder mehr Geräte die gleiche Adresse, sind sie im Nachgang nicht mehr individuell ansprechbar. Es gibt keine automatisierte Methode, diese Mehrfachbelegung wieder zu entflechten. Ein Adresskonflikt lässt sich nur aufzulösen, indem einer der Konfliktpartner temporär von der Kommunikation ausgeschlossen wird, um dann die Adresse des anderen neu vergeben zu können. Eine Maßnahme, die häufig manuellen Eingriff erfordert.

Neben den Pflichtdiensten gibt es noch zwei weitere Dienste, die üblicherweise von jedem Cy4-Gerät angeboten werden. Es sind:

Gerätebezeichnung (DeviceLabel):	`FFFF00h:61:RW`
Werkseinstellungen (FactorySettings):	`FFFF40h:01:WO`

DeviceLabel und *FactorySettings* sind keine Pflichtdienste. Sie müssen also nicht zwingend in der Applikation implementiert sein und in Programmladern sind sie es üblicherweise auch nicht. Die beiden Dienste haben den Zweck, den Umgang mit Geräten zu erleichtern.

Hilfreich dafür ist insbesondere der erste Dienst, die *Gerätebezeichnung*. Mittels dieses Dienstes kann man einem Gerät eine individuelle Bezeichnung, eine Beschriftung oder einen Namen geben. Das ermöglicht, ein ansonsten abstraktes Gerät zu individualisieren. Ein „Außenthermometer Veranda“ ist viel eingängiger als A00124/B00028. Verwaltungstool wie das `cy4cmd` (siehe Kapitel 11.5), nutzen diese Möglichkeit, um dem Anwender eine eingängige Bezeichnung für gefundene Geräte zu präsentieren.

Der Dienst hat eine Breite von 61 Bytes. Erwartet werden ASCII oder UTF-Zeichenketten, doch vorgeschrieben ist das nicht. Vielmehr stellt der Dienst 60 frei beschreibbare Bytes zur Verfügung. Da Zeichenketten üblicherweise mit einem Nullbyte terminiert sind, hat das letzte, also 61. Byte immer den Wert 0.

Je nach Aufgabe und Komplexität der Applikation fallen im Betrieb Daten an, die persistent gespeichert werden müssen. In der Regel handelt es sich dabei um Betriebsparameter oder Einstellungen, die für den Betrieb der Anwendung wichtig sind und dafür im Gerät vorgehalten werden müssen. Ebenso müssen für alle Einstellungen initial gültige Werte vorgesehen werden, die die grundlegende Funktionalität des Gerätes sicherstellen. Gemeint sind: Die Werksvoreinstellungen. Damit ist eigentlich schon klar, welchen Zweck der zweite Dienst hat. Wird *Werkseinstellungen* (*FactorySettings*) ausgelöst, erhalten alle Einstellungen ihre ursprünglichen, nach Inbetriebnahme voriniitialisierten Werte zurück. Das Gerät wird damit in seinen Grundzustand versetzt.

Dabei gibt es eine Ausnahme: Die Cy4NET-Adresse des Gerätes. Sie ist von der Rückstellung der Werte nicht betroffen, sondern behält die mittels *ChangeCy4Address* zugeordneten Adresse im Netz. Andernfalls würde mit Rückstellung der Werte auch das Gerät verschwinden und an seine Werksadresse wieder auftauchen. Das ist bei Zurücksetzen auf *Defaults* in der Regel nicht erwünscht.

> **Hinweis:** Bei allen Diensten, deren Nutzung einen Neustart zur Folge hat, sollte man die Zeit für den Bootvorgang berücksichtigen. Für die Zeit des Neustarts ist der Controller nicht betriebsbereit und kann auf neue Anfragen nicht reagieren. Daher sollte man nach Initiierung eines Neustarts der Busstation eine bestimmte Karenzzeit einräumen, bevor neue Anfragen gesendet werden. Betroffen davon sind die Pflichtdienste *EnterApplication* und *EnterProgramLoader.* Aber auch die Rückstellung auf Werkseinstellungen geht häufig mit einem Neustart einher. Bei der überwiegenden Mehrzahl an Busstationen sollte eine Karenzzeit von zwei Sekunde ausreichend sein.

6.4 Private Typennummern

Das Cy4NET ist als offenes Projekt konzipiert. Anwendungen und Boards sollten einheitlich und öffentlich zugänglich sein. Jeder kann seinen Beitrag dazu leisten und eine neue Anwendung oder ein neue Hardwareplattform beisteuern. Die Typennummern helfen bei der Zuordnung und ermöglichen es Übersicht zu behalten. Doch es gibt es Geräte, die eine interne oder proprietäre Entwicklung darstellen und deren Veröffentlichung nicht gewünscht oder nicht möglich ist. Zu diesem Zweck gibt es einen speziellen Typennummernbereich, der durch Konvention 8 festgelegt ist: Applikationen und Boards dieser Kategorie sollten eine Typennummer im Bereich von 90000 bis 99999 tragen, also am Ende des Typennummernbereiches. Dieser Bereich wird als privat angesehen und öffentlich nicht verwendet. Vergleichbar den *Private Use Areas* (PUAs) des Unicode-System sind Typennummern aus diesem Bereich für den eigenen Gebrauch vorgesehen und können frei genutzt werden.

7 Datagramme

Kommunikation im Cy4NET findet durch Austausch von Datagrammen statt. Ein Datagramm ist eine Bytefolge, eine Einheit von Werten. Daher wird ein Datagramm in der Netzwerktechnik auch als Protokoll-Daten-Einheit, PDU (Protocol Data Unit) [6], bezeichnet. Beide Begriffe werden hier synonym eingesetzt. Eine PDU führt alle, für die Kommunikation notwendigen Informationen mit sich. Dazu gehören neben Struktur- und Verwaltungsinformationen auch Sicherungsmaßnahmen und letztlich die eigentlichen Passagiere des Datagramms, die Daten. Die Cy4NET-PDU beinhaltet unter anderem Absender- und Zieladresse, sowie Dienstort (Service-Location), Dienstbreite (Service-Size) und Nutzlast (Payload). Streng genommen müsste man aus Sicht der Netzwerktechnik die drei letztgenannten Einträge als eine eigenständige, eingebettete Datenstruktur betrachten. Doch in Anbetracht dessen, dass sich die Cy4NET-Kommunikation nur über zwei Schichten erstreckt, können wir auf die Betrachtung einer so genannten SDU (Service Data Unit) verzichten und das Datagramm als Einheit direkt verarbeiten. In Abbildung 7 ist die Zusammensetzung der Cy4NET-PDU zu sehen.

Abbildung 7: Die Cy4NET PDU

Ein Datagramm kann in der Sektion `DAT` bis zu 64 Bytes als Nutzlast mit sich führen. Je nach Ausstattung umfasst das komplette Datagramm damit zwischen 11 bis 76 Bytes. Die PDU setzt sich aus folgenden Sektionen zusammen:

- DPS (Datagram Preamble Sequence): Die Datagramm Einleitungs-Bitsequenz. Dieses immer gleiche 8-Bit Startmuster leitet die Übertragung einer Cy4NET-PDU ein. Gesendet wird binär `10101010` (AAh).
- DTF (Datagram Type Flags): Die Typflags des Datagramms: Die oberen 6 Bits enthalten die Datagramm-Identifikationsbits DIB (Datagram Identification Bits). Cy4NET-PDUs identifizieren sich hier stets mit der Bitfolge `110000` (entspricht der ASCII-Nummer für ‚`0`'). Übrig bleiben Bit 1 und 0, die die Charakteristik des Datagramms bestimmen. Den Aufbau des DTF zeigt Abbildung 8.

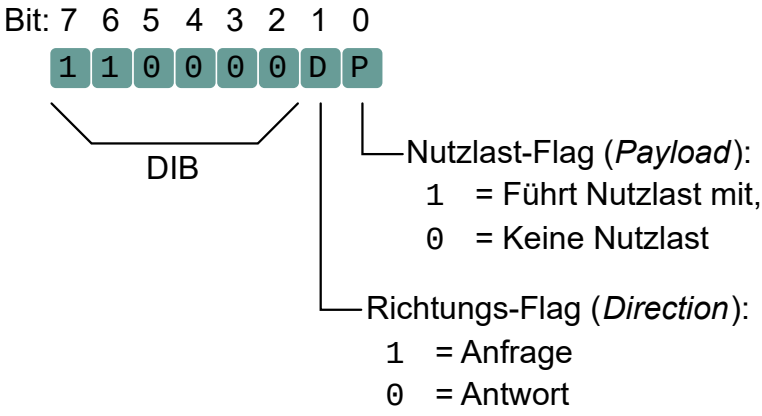


Abbildung 8: Datagramm Typflags (DTF)

Ein Sender schickt eine Anfrage-PDU zum Empfänger (D=1), der beantwortet die Anfrage mit einer entsprechenden Antwort-PDU (D=0). Zusammen mit der Eigenschaft, dass Datenbytes an Dienste gegeben oder von ihnen abgerufen werden, zeigt das P-Bit an, ob das Datagramm Nutzlast mitgeführt (P=1) oder nicht (P=0). Daraus ergeben sich vier mögliche Datagramm-Typen. Welche das sind, zeigt Abbildung 9.
Es ist offensichtlich, dass auf eine Leseanforderung nur eine Leseantwort erfolgen kann. Hier bewegen sich Nutzlast vom Empfänger zum Sender d.h. die Leseanforderung hat keine Nutzlast, die Leseantwort hingegen schon. Bei Schreibvorgängen ist die Sache umgekehrt. Dort werden mittels der Schreibaufforderung übertragenen Nutzdaten durch den Empfänger bestätigt. Folglich führt die Schreibaufforderung Nutzlast mit sich, die Schreibbestätigung hingegen nicht.
Durch invertieren der Bits D und P im DTF wird eine Anfrage zur Antwort und umgekehrt.

DIB	D P	DTF	Typ
110000	1 0	0xC2	Leseanforderung
110000	0 1	0xC1	Leseantwort
110000	1 1	0xC3	Schreibaufforderung
110000	0 0	0xC0	Schreibbestätigung

Abbildung 9: Die vier Datagrammtypen

- DST (Destination): Die Cy4NET-Adresse des Empfängergerätes. Entsprechend der Darstellung oben, ist das erste Byte die Netz-, das zweite Byte die Geräteadresse. Die Zieladresse einer PDU kann ein Rundruf sein.
- SRC (Source): Die Cy4NET-Adresse des Sende-Gerätes. Das Format ist wie bei DST. Da die Quelladresse ja die des sendenden Knoten ist, kann sie nie eine Rundrufadresse sein.

- LOC (Location): Diese vorzeichenlose 24-Bit-Zahl bestimmt den Dienstort oder die Dienstadresse. Der Konvention 6 entsprechend ist die Byte-Reihenfolge Big-Endian, hier also High/Mid/Low. Die Zuordnung der Dienstadressen unterliegt einigen Einschränkungen, doch dazu später mehr.
- CNT (Count): Je nach Datagrammtyp, die Anzahl der
    - mitgeführten (bei Schreibaufforderungen)
    - geschriebenen (bei Schreibbestätigungen)
    - angeforderten (bei Leseanforderungen)
    - gelesenen (bei Leseantworten)

  Datenbytes. Mögliche Werte für CNT sind 1 bis 64, 0 ist nicht erlaubt.
- DAT (Data): Nutzlast-Datenbytes. Datagramme vom Typ Schreibaufforderung und Leseantwort führen in dieser Sektion `<CNT>` viele Datenbytes mit sich (siehe P-Flag im DTF). Bei Schreibbestätigungen und Leseanforderungen ist die Sektion DAT leer.
- CRC (Cyclic Redundancy Check): Prüfsummenwert aus allen Bytes des Datagramms, ausgenommen natürlich das CRC-Byte selbst. Zum Einsatz kommt der DOW-CRC, eine 8-bittige zyklische Redundanzprüfung. In Anhang C befindet sich ein Berechnungsbeispiel mit Code.

Ein Beispiel zum Ablauf der Kommunikation:

Gegeben seien zwei Geräte A und B. Gerät A hat Adresse `155.101` (`9Bh.65h`), B hat `24.17` (`18h.11h`). Gerät B stellt die im vorangegangenen Kapitel vorgestellten Dienste „Luftdruck“ und „Anwendung starten“ zur Verfügung. Um nun von B den Luftdruck zu erfahren, muss A folgende Leseanforderung an B schicken:

```
AA C2 18 11 9B 65 10 02 10 02 8B
```

Gerät B erhält die Anfrage und beantwortet sie mit einer entsprechenden Leseantwort-PDU:

```
AA C1 9B 65 18 11 10 02 10 02 03 F3 6F
```

Nach Empfang der Leseantwort kann Gerät A den Luftdruck aus den mitgeführten Datenbytes auslesen. Er beträgt in diesem Fall `03F3h` = 1011 hPa.

Anfragen und Antworten sind von der Struktur ihrer Datagramme her gleich. Quell- und Zieladresse werden vertauscht, die Bits im DTF intertiert und Nutzlast hinzugefügt bzw., bei Schreibbestätigungen, entfernt.

Erreicht die Anfrage Gerät B nicht oder unvollständig wird B keine Antwort zurückschicken. Das Gleiche gilt, wenn A eine Anfrage an einen Dienst stellt, den B nicht bereitstellt. In beiden Fällen weiß der Sender nach Ablauf der maximalen Round-Trip-Time, dass seine Anfrage erfolglos war.

Der Vollständigkeit halber hier noch die Auslösung eines Neustart der Anwendung von B durch A:

Schreibaufforderung von A an B:

```
AA C3 18 11 9B 65 FF FF F8 01 5E 48
```

Schreibbestätigung von B an A:

```
AA C0 9B 65 18 11 FF FF F8 01 C4
```

Man beachte, dass B nicht unmittelbar neu startet, sondern erst noch die Schreibbestätigung absetzt.

7.1 Rundrufe

Es besteht die Möglichkeit, mehrere Geräte eines Netzes gemeinschaftlich durch einen Rundruf (Broadcast) anzusprechen. Dazu ist eine DST-Adresse anzugeben, bei der die Netz- und/oder die Gerätedresse null ist. Null kann daher als Jokerwert (Wildcard) betrachten, bei dem alle Geräte angesprochen werden, deren eigenen Adressen dem Muster der Zieladresse entsprechen. Beispielsweise würde die Anfrage `123.0` alle Geräte ansprechen, deren Netzadresse `123` lautet. Entsprechend würde die Anfrage `0.123` alle Busstationen adressieren, die als Geräteadresse `123` haben.

Rundrufe stellen im Konzept der Anfragen und Antworten eine Besonderheit dar: Werden mit einer einzelnen Anfrage mehrere Geräte angesprochen, können die Geräte keine Antwort senden ohne sich gegenseitig bei der Übertragung zu behindern. Je nach Umfang der betroffenen Busstationen würden Antworten den Sender erst sehr verspätet bis gar nicht erreichen. Daher gilt generell: Rundrufe werden nie beantwortet, m. a. W. *fire-and-forget*.

Aus dieser Tatsache resultiert eine weitere Einschränkung: Da Broadcasts nie beantwortet werden, gibt es auch keine Möglichkeit, Daten zum Aufrufer zurück zu übertragen. Das heißt: Rundrufe sind stets Schreibaufforderungen die nicht bestätigt werden.

Die gemeinschaftliche Adressierung mehrerer Busknoten hat noch einen weiteren Aspekt: Rundrufe ergeben nur Sinn, wenn alle betroffenen Bustationen am angefragten Ort den gleichen Dienst anbieten. Folge dieser einheitlichen Ansprache ist, dass Knoten einen Teil ihrer Dienste konsolidieren müssen. Andererseits ist jedoch sicherzustellen, dass Dienste, die individuell für ein bestimmtes Gerät existieren, nicht „Opfer“ eines vermeintlichen Broadcasts werden. Man stelle sich nur vor, dass ein globaler Rundruf an ChangeCy4Address alle Busstationen auf eine einheitliche Adresse versetzt.

Um diesem Problem zu entgehen ist es nötig, Dienstbereiche zu separieren. Vergleichbar zu den ersten 1024 KiB des Dienstbereiches, der exklusiv der Nutzung von Programmladern vorbehalten ist, sind Rundrufe nur im Bereich von `E00000h` bis `EFFFFFh` erlaubt. Rundrufe an Dienste außerhalb dieses Bereiches werden nicht wahrgenommen. Dieser Zielbereich ist nicht exklusiv für Rundrufe reserviert, auch normale, also individuelle Dienste können dort angesiedelt sein. Um jedoch ungewollte Nebeneffekte zu vermeiden, sollte dieser Bereich für individuelle Dienste nicht verwendet werden.

Gruppen von Busstationen, die gemeinschaftlich mittels Rundrufe angesprochen werden, müssen über einen gemeinschaftlichen Satz an Diensten verfügen. Bei den bestehenden Busstationen haben sich bereits einige allgemein nutzbare Rundruf-Dienste etabliert. In Anhang B sind die aktuell existierenden Rundrufe zusammengestellt. Zukünftig werden weitere hinzukommen. Auch wenn jede Gruppe von gemeinschaftlich angesprochenen Busstationen ihren eigenen Satz an Diensten in der Rundruf-Zone zusammenstellen kann, hat es Vorteile, die bereits im Einsatz befindlichen Dienstbereiche dabei zu umgehen.

In Anhang A befindet sich eine Zusammenfassung der aktuellen Dienstort-Segmentierung.

8 Zeitbetrachtungen

Cy4NET ist ein Multimaster-Echtzeitkommunikationssystem. Es ist so ausgelegt, dass es auch auf sehr kleinen Controllertypen zum Einsatz kommen kann. Die Symbolrate beträgt 19,2 kBd und ist damit eher als moderat anzusehen. Auch wenn Controller die Datenkommunikation üblicherweise per Interrupt abhandeln, stehen ihnen für Entgegennahme und Verarbeitung eines ankommenden Datenbytes ca. 500 µs zur Verfügung. Eine Erhöhung der Symbolrate würde neben elektrischen Effekten, wie die Erhöhung der Störanfälligkeit, auch eine höhere Arbeitsgeschwindigkeit der Controller voraussetzen bzw. andere Verarbeitungsweisen nötig werden lassen, wie z.B. DMA.

8.1 PDU-Umlaufdauer

Eine PDU kann eine maximale Länge von 76 Bytes haben. Die Übertragung eines Datagramms dieser Länge in der o.g. Symbolrate und im Interruptbetrieb dauert in etwa 45 ms. Auf eine Schreibaufforderung kommt die Schreibbestätigung, auf eine Leseanforderung die Leseantwort. Das bedeutet, dass bei der Anfrage-Antwort-Sequenz entweder die Anfrage oder die Antwort diese Länge erreichen kann. Das entsprechende Gegenstück hat stets die minimale Länge von 11 Bytes. Damit benötigt die Übertragung ca. 7ms. Die maximale Netto-Paketumlaufzeit beträgt damit 45+7 = 52 ms. Wenn man für die Verarbeitung der Anfrage und die Bereitstellung der Antwort noch einmal grob 8 ms veranschlagt, erhöht sich die Zeit auf ungefähre 60 ms.

Abbildung 10 zeigt eine Leseanforderung für 64 Bytes mitsamt der nachfolgenden Leseantwort. Da die Antwort die 64 angefragten Bytes als Nutzlast mitführt, ist das Antwort-Datagramm mit einer Dauer von ca. 44 ms entsprechend länger als die Anfrage.

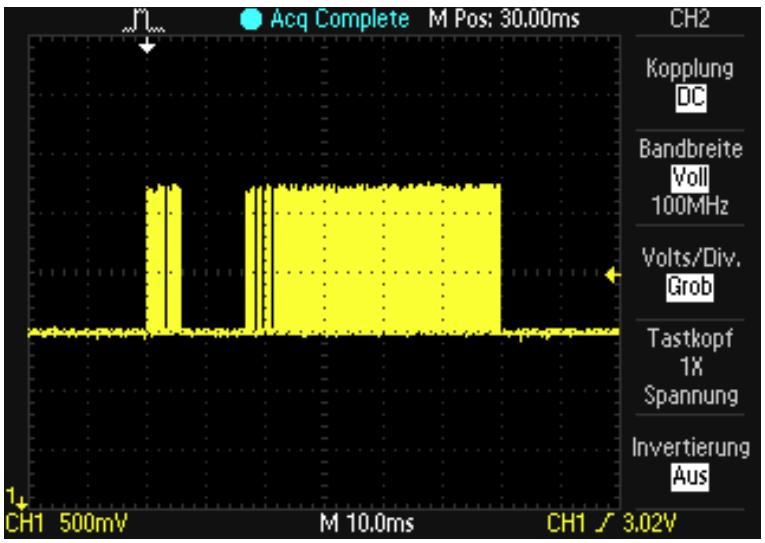


Abbildung 10: Leseanforderung für 64 Bytes und Leseantwort.

8.2 Backoff

Cy4NET ist ein Multimastersystem, d.h. mehrere Busstationen senden gegebenenfalls gleichzeitig. Der Sender kann also nicht davon ausgehen, dass ihm für seinen PDU-Austausch der Bus exklusiv zur Verfügung steht. Deshalb muss er zur Erlangung des Zugriffsrechts den Bus beobachten und darf einen Sendevorgang nur starten, wenn der Bus frei ist. Damit ist zwar sichergestellt, dass Busstationen keine laufenden Übertragungen stören, jedoch kann es passieren, dass zwei Busstationen zeitgleich einen freien Bus detektieren und einen Sendevorgang in Gang setzen. Die Folge sind Kollisionen. Das im Cy4NET eingesetzte Verfahren ist CSMA/CD (Carrier Sense Multiple Access). Das CD steht für Collision Detection und bedeutet, dass Kollisionen zwar erkannt, aber nicht aufgelöst werden. Kommt es auf dem Bus also zu einer Kollision, sind in der Regel die involvierten Datenbytes verändert und die Kommunikationssequenz damit zerstört. In einem solchen Fall müssen alle beteiligten Geräte unmittelbar die Übertragung beenden, sich also zurücknehmen. Nach einer bestimmten Wartezeit dürfen sie einen erneuten Sendeversuch starten. Die Verzögerung des Neusendeversuchs nennt sich Backoff. Im Cy4NET beträgt der Backoff zwischen 3 und 15 ms. Damit nach Ablauf des Backoff nicht alle zeitgleich ihren Neusendeversuch starten und sich die Situation wiederholt, wird die Wartezeit in jedem Gerät individuell per Zufall ausgewählt.

Ungeachtet des Backoffs ist im Cy4NET auf physikalischen Ebene nicht nur CSMA/CD (also Kollisionserkennung), sondern auch CSMA/CR (Auflösung von Kollisionen) möglich. Die Anordnung der Addressbytes im Cy4NET-Datagramm lässt dies ohne Probleme zu. Das leistungsstärkere Collision Resolution Verfahren, das beispielsweise beim I²C [7] Protokoll zum Einsatz kommt, setzt allerdings hardwareunterstützt spezifische Übertragungseinheiten voraus. Um den Aufbau der Busstationen jedoch möglichst einfach zu gestalten (Schwerpunkt 1, Ressourcenschonung), findet die Kommunikation hier auf Basis von UARTs statt. Diese in fast allen Controllertypen vorzufinden asynchronen seriellen Übertragungseinheiten sind in der Regel auf die Erkennung von Kollisionen nicht ausgelegt. Ungeachtet der Situation auf dem Bus, übertragen sie stets den gesamten Bitrahmen, im diesem Fall 8N1, also 10 Bit. Erst nach Abschluss der Übertragung, mit Erhalt des Echos, wird das Auftreten einer Kollision erkannt. Für die Fortsetzung der Kommunikation durch den dominanten Teilnehmer ist das in der Regel aber zu spät. Grundsätzlich steht dem Einsatz von CSMA/CR im Cy4NET-System aber nichts im Wege.

8.3 Queuing Delay

Hat ein Gerät den Bus für seine Kommunikationssequenz gesichert, müssen alle anderen mit ihren Sendewünschen warten, bis der Bus wieder frei ist. Dieses „Ausreden lassen“ kann dazu führen, dass Sendewünsche über einen längeren Zeitpunkt nicht erfüllt werden können. Um zu vermeiden, dass eine einzelne Busstation durch sukzessive Kommunikationsaktivitäten das Medium Bus exklusiv für sich beansprucht, also „nicht aufhört zu reden“, muss jede sendende Einheit nach Abschluss einer erfolgreichen Kommunikationssequenz eine Pause einlegen, bevor sie neue Anfrage auf die Reise schicken darf. Diese Verzögerung nennt sich Queuing Delay und beträgt beim

Cy4NET-Protokoll 50ms. Abbildung 11 zeigt das Oszillogramm der Queuing Delay bei kurz aufeinanderfolgenden Kommunikationssequenzen.

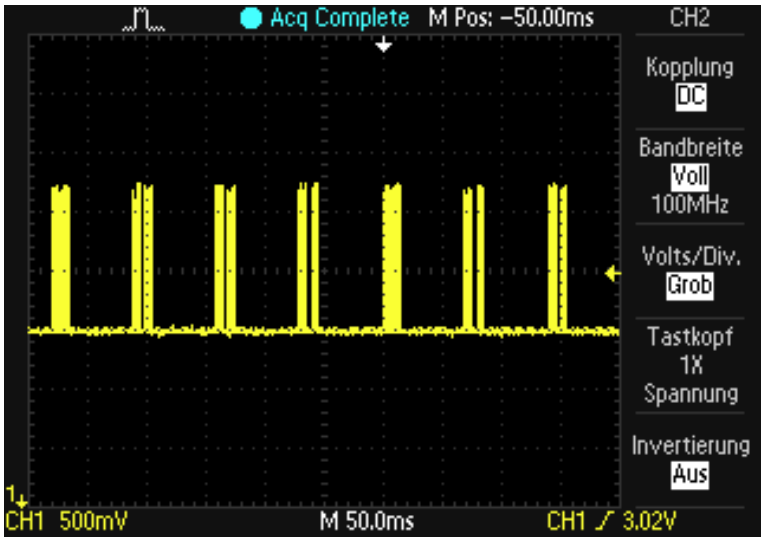


Abbildung 11: 50 ms Queuing Delay

Natürlich kann es weiterhin passieren, dass Busstationen länger auf eine Sendemöglichkeit warten, jedoch werden sie dabei nicht durch ein einzelnes Gerät blockiert.

8.4 Round-Trip-Time

Das Warten auf einen freien Bus, sowie Queuing-Delay und Backoff, können die Dauer einer Kommunikationssequenz deutlich in Höhe treiben. Da eine Nachricht stets aus einer Anfrage und einer Antwort besteht müssen dazu immer zwei Datagramme auf dem Bus übertragen werden. Um mehrere Neuversuche und Wartezeiten überbrücken zu können, steht den Busstationen für Anfrage und Antwort jeweils ein Zeitfenster von 250 ms zur Verfügung. Darin bereits enthalten ist die Bearbeitungszeit, die das antwortende Gerät für die Bereitstellung der Antwort benötigt. Insgesamt ist mit Initiierung einer Nachrichtenübertragung nach spätestens 2·250 ms = 500 ms also klar, ob die Übertragung erfolgreich war oder nicht. Schließt sich das Zeitfenster dieser so genannten Round-Trip-Time, gilt die Übertragung als fehlgeschlagen. Man beachte, dass die Round-Trip-Time den w*orst case* und damit eine Zeitobergrenze darstellt. Üblicherweise wird die Kommunikation schneller abgewickelt. Kann beispielsweise die sendende Busstation ihre Anfrage direkt auf dem Bus platzieren, ist bereits nach der Hälfte der Zeit klar, dass die Übertragung fehlgeschlagen ist. Die theoretisch kürzeste Antwortzeit liegt bei 12 ms (Anfrage plus Antwort je 11 bzw. 12 Bytes mit 520 µs pro Byte).

8.5 Interbyte Timeout

Um ein Datagramm zusammensetzen zu können, müssen mehrere aufeinanderfolgender Datenbytes beim Empfänger eingegangen sein. Bei der asynchronen Datenübertragung, mit einer Symbolrate von 19,2 kBd, dauert der Empfang eines Datenbytes ungefähr eine halbe Millisekunde. Damit ist jedoch noch nichts darüber gesagt, wie viel Zeit zwischen dem Empfang zweier aufeinanderfolgender Datenbytes vergehen darf. Eine Betrachtung der PDU-Umlaufzeiten wie oben ergibt nur Sinn, wenn die Zeitlücke zwischen dem Empfang zweier Datenbytes mit einbezogen wird.

Für Empfangsstationen ist es gar nicht einfach, den Start eines neuen Datagramms festzustellen. Zwar gibt es das DPS und das DTF, die zu Beginn eines Datagramms versendet werden, jedoch sind diese Bytewerte kein exklusives Startmuster. Ebenso können sie in Mitten einer Nutzlast auftreten, wo sie mitnichten den Beginn eines neues Datagramms einläuten. Wichtig ist vielmehr, das Datagramme immer in einer Sequenz von Datenbytes (data bursts) übertragen werden. Bei der Abfrage einer Temperatur beispielsweise, wird die Thermometerstation ihre Antwort in einer zusammenhängenden Datensequenz liefern. Doch was bedeutet hier zusammenhängend? Die Frage, die sich stellt ist also: Ab wann gilt die Sequenz eingehender Datenbytes als unterbrochen? Im Grunde stellt sich diese Frage bereits auf OSI-Ebene 1, also der Bitübertragungsschicht (*Physical Layer*), denn jegliche Funktionalität des Datenempfang seitens der Hardware muss dieses Problem behandeln. Im Cy4NET übernimmt das, wie in Kapitel 10 erläutert, das Board-Modul. Darin ist festgelegt, dass zwischen dem Empfang zweiter sukzessiver Datenbytes nicht mehr als 3 ms verstreichen dürfen. Andernfalls gilt die Empfangssequenz als unterbrochen und ein gerade in Aufbau befindliches Datagramm wird verworfen. Diese Interbyte-Zeitgrenze ist in etwa vergleichbar mit dem `VTIME` Parameter der POSIX-`termios` Datenstruktur bei einer seriellen Übertragung.

9 Interner Ablauf der Kommunikation

Pakete werden gesendet, Pakete werden empfangen. Wenn ein Gerät eine Anfrage versendet, schickt es ein entsprechendes Datagramm über den Bus und wartet auf die zurückkommende Antwort. Für die empfangende Busstation stellt sich die Sache umgekehrt dar: Eine Anfrage wird entgegengenommen, bearbeitet und eine entsprechende Antwort zurückgeschickt. Folglich gibt es zwei Arten von Sequenzen, die unabhängig voneinander betrachtet werden wollen: Die Eingehende und die Ausgehende. Auch wenn der Initiator einer Übertragung später die Antwort wieder entgegennimmt, beginnt er mit dem Aussenden einer Anfrage. Daher wird seine Sequenz als *ausgehende Übertragung* bezeichnet. Für den Empfänger ist es umgekehrt. Seine erste Aktivität beginnt mit der Entgegennahme einer Anfrage. Seine Sequenz die *eingehende Übertragung*.

Eine ausgehende Sequenz im Sender verursacht also stets eine eingehende Sequenz im Empfänger. Nun können die Geräte eines Multimastersystems aber gleichzeitig Sender und Empfänger sein. Das bedeutet: Jedes Gerät, das selbständig Anfragen senden möchte, muss beide Teile der Kommunikationssequenz beherrschen - und das asynchron.

Aufgaben solcher Art übernehmen im Bereich der Mikrocontroller meist Zustandsautomaten [8]. Diese Kalküle aus dem Bereich der theoretischen Informatik arbeiten nicht wie klassische Kontrollstrukturen, funktional oder imperativ. Automaten basieren auf Zuständen, Zustandsübergängen und Aktionen. Automaten, die bei der Verarbeitung der oben erwähnten Übertragungssequenzen eingesetzt werden, sind vom Typ her den Mealy-Automaten [9] am nächsten. Da bei der Einführung der Kommunikationssequenzen in Kapitel 9.2 einige Begrifflichkeiten aus dem Bereich der Automatentheorie Verwendung finden, folgt zunächst eine kurze, formale Vorstellung.

9.1 Zustandsautomaten

Ein Zustandsautomat, auch Zustandsmaschine genannt, ist ein 5-Tupel das wie folgt definiert ist:

$$M = (S , \Sigma , \Omega , \zeta , s_0)$$

mit

S	Menge der Zustände
Σ	Menge der Eingaben
Ω	Menge der Ausgaben
ζ	Zustandsübergangsfunktion
s_0	Anfangszustand

Der Automat besteht aus einer Reihe von Zuständen, die in der Menge S zusammengefasst sind. Einer der Zustände ist als Startzustand markiert, d.h. zu Beginn befindet sich der Automat in dem Zustand s_0. Nun können Ereignisse eintreten, die im Automaten einen Zustandswechsel hervorrufen. Im Terminus der Automatentheorie heißen diese Ereignisse Eingaben. Die Gesamtheit mögli-

cher Eingaben sind im Eingabealphabet Σ (Sigma) untergebracht. Welche Ereignisse dabei welche Wechsel verursachen, bestimmt die sogenannte Zustandsübergangsfunktion ζ (Zeta). In ihr ist das komplette Verhalten des Automaten abgebildet. ζ hat folgende Signatur:

$$\zeta : S \times \Sigma \rightarrow S \times \Omega$$

Die Funktion legt fest, von welchem Zustand $s \in S$ aus bei Eintritt welcher Eingabe $e \in \Sigma$ der Automat in welchen Folgezustand wechselt. Tritt ein Zustandswechsel ein, wird dabei die Ausgabe $o \in \Omega$ getätigt. Auch sie ist in der Zustandsübergangsfunktion spezifiziert.

Zeta wird üblicherweise als Wertetabelle notiert. Die Definition ist vollständig, was bedeutet, dass zu jeder Kombination aus Zustand und Eingabe ($S \times \Sigma$) eindeutig festgelegt ist, welcher Folgezustand angenommen und welche Ausgabe getätigt wird ($S \times \Omega$). Abbildung 12 zeigt den konzeptionellen Aufbau der Wertetabelle.

S \ Σ	e_1	e_2	$\cdots$	e_m
s_1	s_{a1}, o_{b1}	s_{a2}, o_{b2}	$\cdots$	s_{a3}, o_{b3}
s_2	s_{a4}, o_{b4}	s_{a5}, o_{b5}		s_{a6}, o_{b6}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
s_n	s_{a7}, o_{b7}	s_{a8}, o_{b8}	$\cdots$	s_{a9}, o_{b9}

$a_i \in \{ 1 .. n \}$
$b_j \in \{ 1 .. |\Omega| \}$

Abbildung 12: ζ-Wertetabelle

Für die menschliche Auffassung deutlich eingänger sind jedoch Zustandsdiagramme. Sie verdeutlichen die Arbeitsweise des Automaten auf grafische Weise. Jeder Zustand ist als Kreis dargestellt, Zustandswechsel werden als Pfeile eingezeichnet, die zwischen den Zustandskreisen verlaufen. Ein- und Ausgaben tauchen als Kantenmarkierungen an den Pfeilen auf. Auf ein Beispiel wird hier verzichtet, da im folgenden Kapitel Zustandsdiagramme für die Darstellung der Kommunikationssequenzen vorkommen werden.

In der Definition des Automaten muten einige Bezeichnungen vielleicht etwas fremdartig an. Der Grund dafür ist, dass Automaten in der theoretischen Informatik mitunter eingesetzt werden, formale Sprachen und Grammatiken zu erkennen, algorithmische Eigenschaften zu beschreiben und Theoreme zu beweisen. Der oben vorgestellte Automatentyp ist dabei nur einer unter vielen.

Wichtig für den Sachverhalt hier ist: Eingaben sind Ereignisse, Ausgaben sind Aktionen.

9.2 Die Kommunikationssequenzen

9.2.1 Die ausgehende Kommunikationssequenz

Als Erstes wird die ausgehende Kommunikationssequenz betrachtet. Der zuständige Automat verfügt über fünf Zustände: `IDLE`, `OUTBOX`, `TRANSMIT`, `RECEIVE` und `INBOX`. Aufgrund der Entsprechung zur Referenzimplementierung sind die Bezeichnungen in englisch gehalten. Abbildung 13 zeigt das Zustandsdiagramm des Automaten.

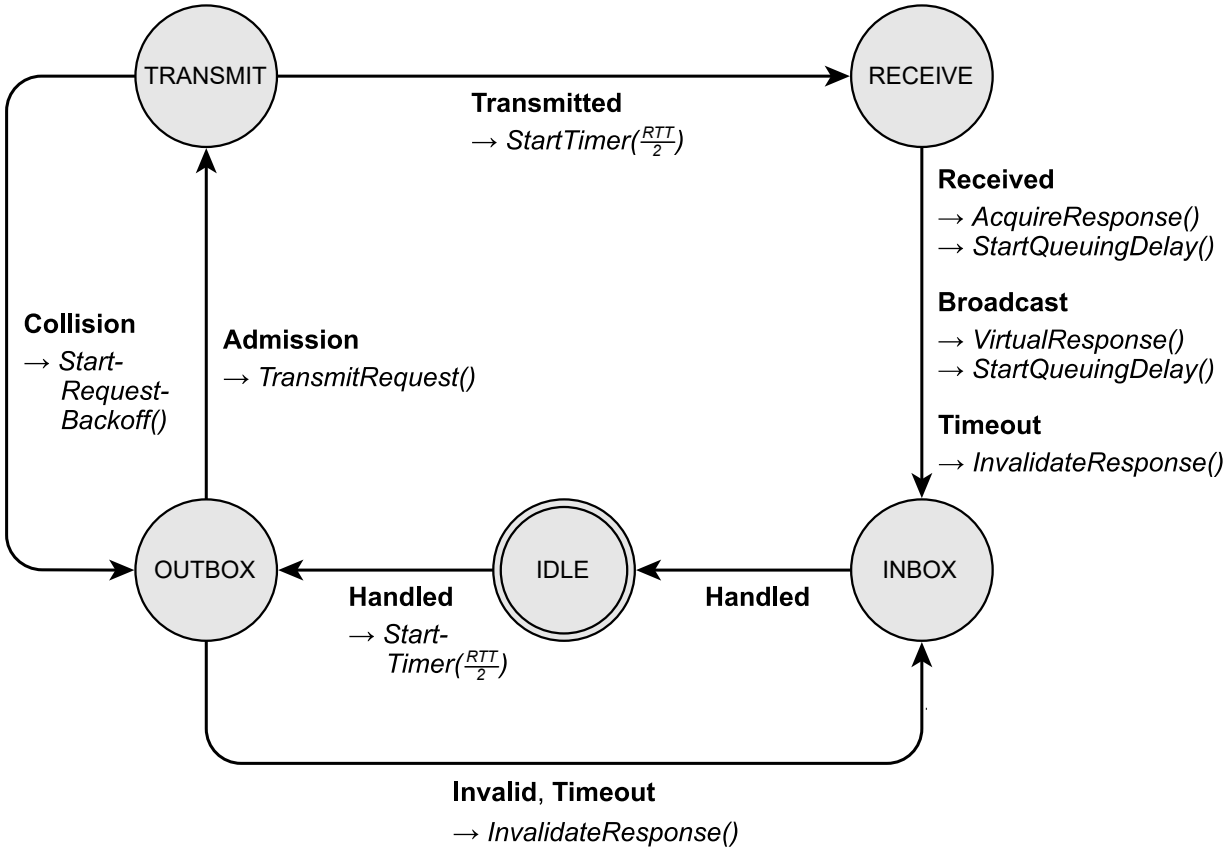


Abbildung 13: Zustandsdiagramm der ausgehenden Kommunikationssequenz

Zu Beginn befindet sich der Automat im Zustand IDLE. Dieser Zustand ist der Ausgangszustand, in dem der Automat verweilt, wenn aktuelle keine ausgehende Übertragung im Gange ist. Initiiert das umgebende System nun das Senden einer Anfrage, löst das im Automaten das Ereignis Handled aus. Der Automat wechselt daraufhin in den Zustand OUTBOX und führt dabei die Aktion `StartTimer()` aus. Gemeint ist die Zeitgrenze für die Paketumlaufzeit. Da der Paketumlauf aus dem Senden und dem Empfang einer PDU besteht, teilt sich die Round-Tip-Time (RTT) in zwei gleiche Hälften auf. `StartTimer()` stellt den Zeitgeber also auf RRT÷2 = 250 ms. Jetzt wartet der Automat darauf, die Anfrage-PDU auf dem Bus übertragen zu dürfen. Das zugehörige Ereignis ist Admission. Unter welchen Bedingungen die Freigabe zum Senden erteilt wird, wird später erläutert. Läuft die oben eingestellte Zeit ab, bevor das Ereignis Admission eingetreten ist, löst das das Ereignis Timeout aus. Die Übertragung gilt als gescheitert. Der Automat wechselt in den Zustand INBOX und erzeugt mittels `InvalidateResponse()` eine entsprechende Mitteilung, die in der Inbox für das System zur Abholung bereitgestellt wird. Das Gleiche passiert, wenn die vom System übergebene Anfrage fehlerhaft oder ungültig war. Das dazugehörige Ereignis ist Invalid. Es wird immer dann ausgelöst wird, wenn die gerade betrachtete PDU nicht dem in Kapitel 7 dargelegten Format entspricht.

Tritt im Zustand OUTBOX aus Ereignis Admission ein, beginnt die Ausgabe der PDU-Bytes auf dem Bus. Der Automat wechselt dazu in den Zustand TRANSMIT und verbleibt dort, bis die Übertragung

aller Bytes abgeschlossen ist. Während der Ausgabe wird laufend geprüft, ob die Daten korrekt auf dem Bus erscheinen oder ob konkurrierende Schreibvorgänge die Übertragung stören. Sollte das der Fall sein, wird das Ereignis Collision ausgelöst. Die Übertragung wird unmittelbar abgebrochen und der Automat kehrt in den Zustand OUTBOX zurück. Auf dem Weg dorthin wird mittels `StartRequestBackoff()` die Pause zur Neusendung gestartet. Solange diese Pause anhält, wird die Übertragungseinheit kein Admission-Ereignis auslösen. Man beachte, dass der Zeitgeber der Paketumlaufzeit davon nicht beeinflusst wird und weiterläuft. Die Übertragungseinheit hat für das Senden der Anfrage-PDU also insgesamt 250 ms Zeit. Darin enthalten sind Neusendeversuche nach Kollisionen sowie deren Verzögerungen.

Schafft es die Übertragungseinheit, die komplette PDU abzusetzen, hat der Sender seine Aufgabe erledigt. Die Übertragung der Anfrage ist abgeschlossen und das Ereignis Transmitted wird auslöst. Der Automat wechselt in den Zustand RECEIVE. Dabei startet er mittels `StartTimer()` den zweiten Teil der Round-Tip-Time. Diese Laufzeit gilt dem Empfänger. Ihm obliegt es nun, die Anfrage entgegenzunehmen, zu bearbeiten und zurückzusenden. Der Sender wartet lediglich auf den Eingang der Antwort. Schafft es der Empfänger nicht die Antwort in gebührender Zeit zu liefern, passiert das Gleiche wie im Zustand OUTBOX: Das Ereignis Timeout wird ausgelöst und die Übertragungseinheit stellt wieder eine entsprechende Mitteilung in der Inbox zur Abholung bereit. Diese Mitteilung ist übrigens eine aus der Anfrage abgeleitete Antwort, die als ungültig gekennzeichnet ist. Daher auch der Name der Aktion: `InvalidateResponse()`.

Die beiden übrigen Ereignisse verlassen den Automat ebenfalls Richtung INBOX. Erhält der Sender eine korrekte Antwort-PDU löst das das Ereignis Received aus. Die Übertragungseinheit generiert daraus die Antwort-Mitteilung und stellt sie dem System in der Inbox zur Abholung bereit. Die Übertragung ist erfolgreich abgeschlossen.

Wäre da noch das Ereignis: Broadcast. Wie in Kapitel 7.1 erwähnt, werden Rundrufe nie beantwortet. Bei dem Automaten für ausgehende Übertragungen hat das zur Folge, dass im Zustand RECEIVE niemals eine entsprechende Antwort eintrifft. Wie das Ereignis Invalid bei ungültigen PDUs, gibt es dort das Ereignis Broadcast. Es wird ausgelöst, wenn es sich bei der betrachteten PDU um einen Rundruf handelt. Erreicht der Automat den Zustand RECEIVE und die Anfrage-PDU ist ein Rundruf, wechselt der Automat unmittelbar in den Zustand INBOX. Damit nach extern die Antwort-Mittelungen einheitlich behandelt werden können, wird analog zur Zeitüberschreitung eine künstliche Antwort aus der Anfrage abgeleitet. Im Unterschied zur Zeitüberschreitung ist die durch `VirtualResponse()` erzeugte Antwort aber gültig.

Received und Broadcast sind Ereignisse, die bei einer erfolgreichen Übertragung ausgelöst werden. Wie in Kapitel 8.3 erwähnt, müssen Busstationen nach einer erfolgreichen Übertragung eine kurze Karenzzeit einhalten, bevor sie eine neue Anfrage starten dürfen. Die Aktion `StartQueuingDelay()` der Ereignisse Received und Broadcast sorgen dafür, dass eine neue Anfrage für einige Zeit warten muss. Konkret bedeutet das, dass eine neue Anfrage zwar entgegengenommen wird, jedoch das Ereignis Admission seitens der Übertragungseinheit so lange verweigert wird, wie die Queuing Delay einer vorausgegangenen Übertragung noch nicht abgelaufen ist. Der Automat verharrt solange im Zustand OUTBOX.

Dass bei dem Zustandswechsel hier zwei Aktionen ausgeführt werden widerspricht streng genommen der oben angegebenen Signatur der Übergangsfunktion. Die rechte Seite müsste dafür zu S×Ω×Ω erweitert werden. Das wäre an sich kein Problem und je nach Typ des Automaten auch abwägenswert. Doch Grundsätzlich hätte diese Erweiterung keinen Einfluss auf die Arbeit des Automaten. Schließlich ist es immer möglich, mehrere Aktionen zu einer einzelnen zusammenzufassen und damit mehrere Aktionen auf die Ausführung einer einzelnen Vertreteraktion zu reduzieren. In der Referenzimplementierung sind die Aufrufe zur Neusendeverzögerung übrigens mit in den Aktionen `Acquire`- und `VirtualResponse()` untergebracht.

Liegt eine Antwort vor, ob gültig oder nicht, muss das System sie abholen. Wie zu Beginn, bei der Initiierung der Anfrage, löst die Entgegennahme der Antwort das Ereignis Handled aus. Die Abholung bildet den Abschluss der ausgehenden Kommunikationssequenz. Der Automat befindet sich wieder im Zustand IDLE und ist bereit für die nächste Anfrage.

> Man beachte, dass stets wenn der Automat eine Anfrage vom System entgegengenommen hat, auch eine entsprechende Antwort geliefert bzw. generiert wird. Damit ist von außen seitens des Systems eine einheitliche Handhabung möglich, d.h. jede entgegengenommene Anfrage wird beantwortet. Die Gültigkeit der Antwort zeigt an, ob die Kommunikationssequenz erfolgreich war oder fehlgeschlagen ist.

9.2.2 Die eingehende Kommunikationssequenz

Zu jeder ausgehenden Sequenz des Sender muss es eine eingehende Sequenz im Empfänger geben. Mindestens eine, denn im Falle eines Rundrufes können es auch mehrere sein. Die anfragende Busstation sendet eine PDU und empfängt eine PDU. Die antwortende Station empfängt eine PDU und sendet anschließend eine zurück. In gewisser Hinsicht sind die Aufgaben, die die Kommunikationsautomaten haben, bis auf eine Phasenverschiebung des Sende- und Empfangsprozesses, symmetrisch. Und in der Tat verfügen beide Automaten über die gleichen Zustände : IDLE, OUTBOX, TRANSMIT, RECEIVE und INBOX. Da jedoch der eingehende Automat darauf wartet PDUs entgegenzunehmen, ist sein Ausgangszustand nicht IDLE, sondern RECEIVE. In Abbildung 14 ist der Zustandsdiagramm des Automaten für die eingehende Kommunikationssequenz dargestellt.

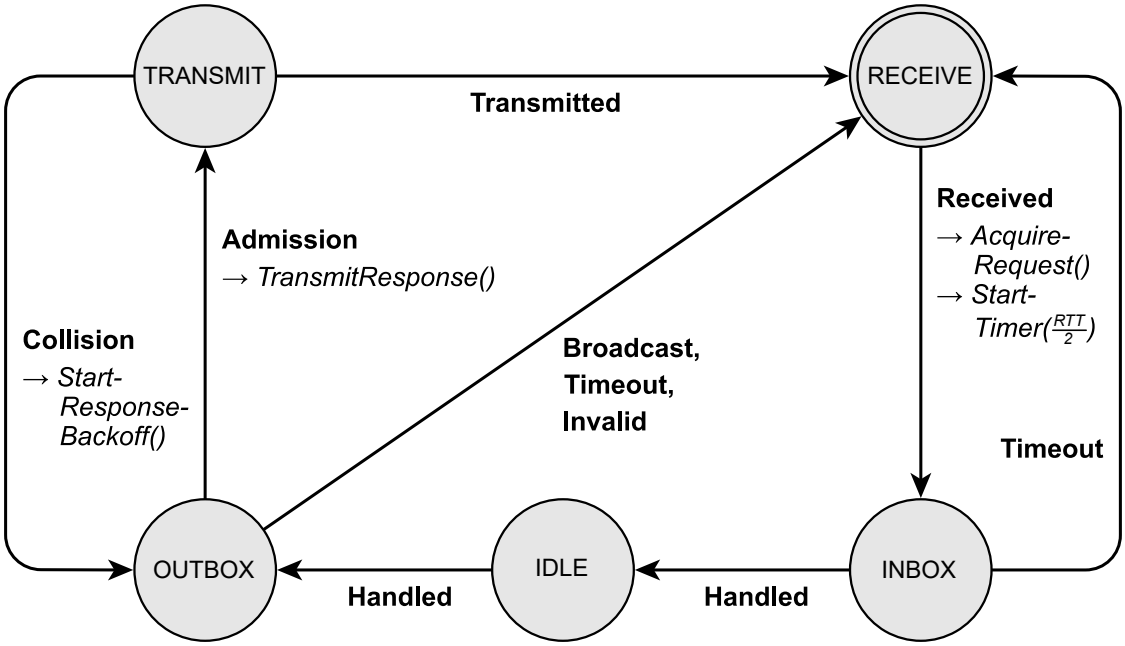


Abbildung 14: Zustandsdiagramm der eingehenden Kommunikationssequenz

Nachdem das sendende Gerät seine Anfrage erfolgreich über den Bus abgesetzt hat, beginnt für den Empfänger die Arbeit. Das Eintreffen einer Anfrage löst das Ereignis Received aus und setzt damit den Automaten für die eingehende Kommunikationssequenz in Gang. Dieser übernimmt die Anfrage mittels `AcquireRequest()` und wechselt in den Zustand INBOX. Damit hat das System nun die Aufgabe, die Anfrage entgegenzunehmen, sie zu bearbeiten und die Antwort wieder dem Sender zukommen zu lassen. Da mit Erhalt der Anfrage bereits der erste Teil des Round-Trips seitens des Senders absolviert wurde, wird der Zeitgeber wieder auf die Hälfte der RTT gesetzt. Sollte es das System versäumen, die Anfrage im geeigneten Zeitrahmen aus der Inbox abzuholen, wird der Automat durch das Ereignis Timeout wieder in den Ausgangszustand versetzt. Die Kommunikationssequenz ist damit gescheitert, der Sender würde ohnehin mit keiner Antwort mehr rechnen.

Der Weg von INBOX zu OUTBOX ist Aufgabe des Systems. Entgegennahme der Anfrage und Auslieferung der Antwort lösen dasselbe Ereignis aus: Handled. Während der Bearbeitung durch das System ist die Übertragungseinheit arbeitslos, sie verweilt im Zustand IDLE. Nach Abschluss der Arbeit übergibt das System die Antwort an die Übertragungseinheit und führt damit den Automat in den Zustand OUTBOX. Wie bei dem ausgehenden Automaten wartet der eingehende Automat in diesem Zustand darauf, die Freigabe zur Übertragung einer PDU zu erlangen. Im Gegensatz zu oben handelt es sich hier aber um die Antwort-PDU. Ist beim Wechsel in den Zustand OUTBOX bereits eine Zeitüberschreitung eingetreten, geschieht dasselbe wie bei INBOX: Der Automat verwirft die Antwort und kehrt direkt in seinen Anfangszustand zurück. Die gleiche Aktion findet statt, wenn die Anfrage ein Rundruf war oder wenn das System auf die Beantwortung der Anfrage verzichten

möchte. Im zweiten Fall liefert das System zwar eine Mitteilung an die Übertragungseinheit, jedoch hat das System diese Mitteilung zuvor für ungültig erklärt, was im Automaten das Ereignis Invalid auslöst.

Der Rest der Kommunikationssequenz funktioniert in analoger Weise zum ausgehenden Automaten. Wird die Freigabe erteilt, wechselt der Automat in den Zustand TRANSMIT und beginnt mit der Übertragung der Antwort-PDU. Tritt dabei eine Kollision ein, springt der Automat wieder zurück in den Zustand OUTBOX und kann nach der obligatorischen Backoff-Verzögerung einen neuen Versuch starten. Das Ende der Übertragung wird dem Automaten durch das Ereignis Transmitted mitgeteilt, der dadurch ausgelöst wieder seinen Anfangszustand RECEIVE zurückkehrt. Die eingehende Kommunikationssequenz ist damit abgeschlossen.

9.2.3 Zustände und Ereignisse

Bei der Betrachtung der beiden Automaten fallen einige Übereinstimmungen auf. Ihre Zustände sind gleich betitelt, die Ereignisse beider Automaten tragen gleiche Namen. Die Strukturen ihrer Zustandsdiagramme sind sehr ähnlich, was auf eine große Übereinstimmungen in ihren Übergangsfunktionen schließen lässt. Und wie aus dem nachfolgenden Kapitel hervorgeht, ist die Abfolge der Zustandswechsel in den jeweiligen Phasen der Kommunikation ebenfalls gleich.

Zustände und Ereignisse in beiden Automaten tragen aber nicht nur gleiche Namen, sie haben auch in beiden Automaten einheitliche Bedeutungen. Das heißt, jede Art von Ereignis wird durch die gleichen Ursachen oder Gegebenheiten ausgelöst. Auch wenn beide Automaten hinsichtlich Übergangsfunktionen oder Startzustände Unterschiede aufweisen, bedeutet die Möglichkeit, Ereignisse und Zustände einheitlich betrachten und behandeln zu können, eine deutliche Vereinfachung bei der Implementierung des Protokolls. Übergeordnet lassen sich die Zustände wie folgt beschreiben:

- IDLE: Die Übertragungseinheit hat nichts zu tun. Entweder wartet der eingehende Automat auf einen Sendeauftrag oder der ausgehende Automat darauf, dass das System die von ihm bereitgestellte Anfrage in eine Antwort verarbeitet.
- OUTBOX: Es steht eine PDU zur Übertagung bereit. Je nachdem ist es eine Anfrage- oder eine Antwort-PDU. In beiden Fällen besteht die Hauptaufgabe darin abzuwarten, bis die Übertragungseinheit das Aussenden über den Bus freigibt.
- TRANSMIT: Die Übertragungseinheit ist damit beschäftigt, die Datenbytes der PDU auf dem Bus auszugeben. Nach erfolgreichem Abschluss erlöst Received den Automaten aus diesem Zustand. Tritt ein Fehler auf, muss der Automat wieder zurück in den Zustand OUTBOX.
- RECEIVE: Die Übertragung ist abgeschlossen. Nun wartet der Automat entweder auf den Erhalt einer Antwort oder auf den Eingang einer Anfrage.
- INBOX: Eine PDU steht zur Abholung durch das System bereit. Handelt es sich dabei um eine Antwort-PDU ist die Übertragung mit Abnahme der PDU abgeschlossen. Im Falle einer Anfrage beginnt damit die Bearbeitung.

Das Auslösen von Ereignissen findet unabhängig vom Zustand des Automaten statt. Das bedeutet, dass Ereignisse immer dann ausgelöst werden, wenn die entsprechenden Voraussetzungen dafür gegeben sind. Beispiel: Broadcast. Das Ereignis wird immer dann ausgelöst, wenn es sich bei der

aktuell behandelten PDU um einen Rundruf handelt. Auf diese Weise betrachtet, ist das Ereignis Broadcast im Grunde ein Zustand. Jedoch kommt es im Kontext unseres Übertragungsprotokolls nicht als Zustand, sondern als Ereignis zum Einsatz. Die Einteilung in Ereignisse und Zustände ist nicht immer offensichtlich. Sie muss, je nach Gegebenheit und Aufgabe für jeden Automaten individuell neu getroffen werden.

In Cy4NET-Automaten kommen acht Ereignisse zum Einsatz. Sie und die Ursache ihrer Auslösung sind in der folgenden Liste aufgeführt:

- Transmitted: Dieses Ereignis wird ausgelöst, wenn die Übertragungseinheit das letzte Byte einer PDU erfolgreich auf den Bus geschrieben hat. Bei der ausgehenden Kommunikationssequenz ist es eine Anfrage-, bei der eingehenden eine Antwort-PDU. Das Ereignis wird bei den Automaten nur im Zustand TRANSMIT verwertet.
- Collision: Ebenfalls nur im Zustand TRANSMIT kommt dieses Ereignis zum Tragen. Bei Aussendung eines Datenbytes auf dem Bus wird das geschriebene Byte unmittelbar vom Bus wieder zurückgelesen. Unterscheidet sich das Echo vom ausgegebenen Datenbyte, hat es ein Problem auf dem Bus gegeben. Üblicherweise wird dieser Umstand durch gleichzeitige Schreibvorgänge mehrerer Busstationen hervorgerufen. Kollidierende Schreibvorgänge, führen zur Auslösung dieses Ereignisses.
- Received: Immer, wenn die Übertragungseinheit eine gültige PDU empfangen hat, wird dieses Ereignis ausgelöst. Bei der ausgehenden Kommunikationssequenz ist es eine Antwort-, bei der eingehenden Sequenz eine Anfrage-PDU. Received kommt nur im Zustand RECEIVE zum Einsatz.
- Broadcast: Handelt es sich bei der aktuell behandelten PDU um einen Rundruf wird dieses Ereignis ausgelöst. Broadcast kennzeichnet damit, wie eben bereits erwähnt, im Grunde einen Zustand.
- Invalid: Ebenfalls ein Zustand ist Ursache für die Auslösung dieses Ereignisses. Entspricht die aktuell betrachtete PDU nicht dem vorgegebenen Format, ist sie ungültig. Diese Eigenschaft löst Invalid aus.
- Timeout: Ist der Zeitgeber abgelaufen, löst das eine Zeitüberschreitung aus. Gemeint ist hier der Zeitgeber, der die Paketumlaufzeit kontrolliert. Auch das Ereignis Timeout beschreibt damit einen Zustand.
- Handled: Interaktion zwischen dem System und der Übertragungseinheit löst dieses Ereignis aus. Das betrifft das Einstellen oder die Entgegennahme von PDUs. Handled dient somit als Schnittstelle zur externen Verarbeitungseinheit. In der Referenzimplementierung sind die Aufrufe der `Post`- und `Take`-Funktionen Auslöser für dieses Ereignis.
- Admission: Das Ereignis mit den wohl meisten Auslösebedingungen. Admission ist das Freigabesignal um eine PDU absenden zu können. Dazu muss sich der Automat im Zustand OUTBOX befinden. Erste Voraussetzung für Admission ist, dass keine Pausen ausstehend sind. Davon betroffen sind die Queuing Delay sowie die Backoff-Pause. Solange beide Zeitfenster nicht geschlossen sind, wird die Freigabe verweigert. Weiterhin ist es wichtig, dass aktuell keine Datenbytes auf dem Bus ausgegeben werden, d.h. der Bus muss frei sein. Die Übertragungseinheit überwacht dazu durchgehend die Busaktivität. Erst wenn eine bestimmte Zeit lang Ruhe herrscht, also keine Kommunikation auf dem Bus stattfindet, gilt der Bus als frei. Diese Ruhezeit ist nicht einheitlich geregelt, bei den meisten Geräten

beträgt sie ca. 3-5 ms. Und dann noch Eines: Die eigene, sprich interne, Kommunikation muss ruhen. Zwar arbeiten beide Automat parallel und unabhängig voneinander, jedoch müssen sie sich beim Zugriff auf das Betriebsmittel Bus synchronisieren. Voraussetzung ist daher, dass sich keiner der beiden Automaten im Zustand TRANSMIT befindet.

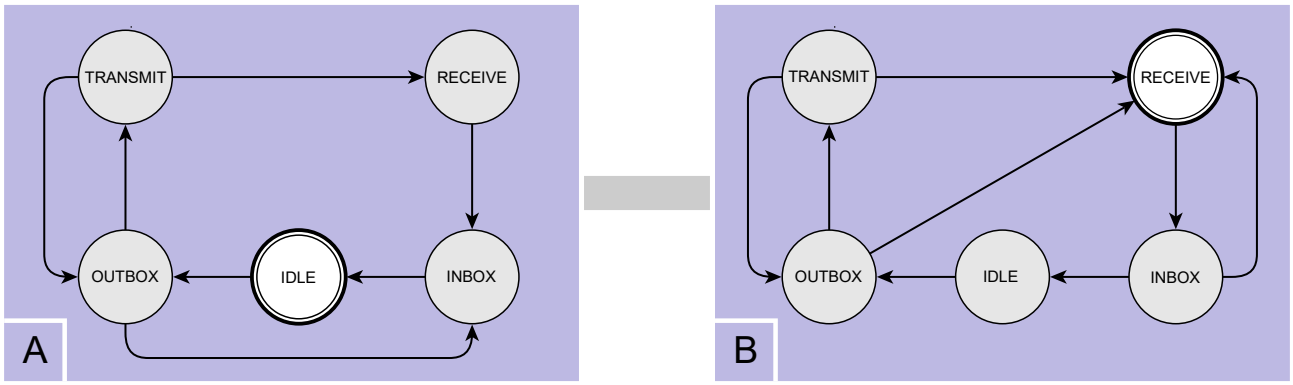
Erst mit Einhaltung all dieser Bedingungen, wird die Übertragungseinheit die Sende-Freigabe erteilen, respektive das Ereignis Admission auslösen.

Auch wenn die Bedeutung der Zustände und Ereignisse in beiden Automaten übereinstimmend ist, muss berücksichtigt werden, dass es sich bei der eingehenden und ausgehenden Kommunikationssequenz um zwei unabhängig arbeitende Automaten handelt, jeder mit seinem Satz an Ereignissen und Auslösern und jeder in seinem eigenen, individuellen Zustand.

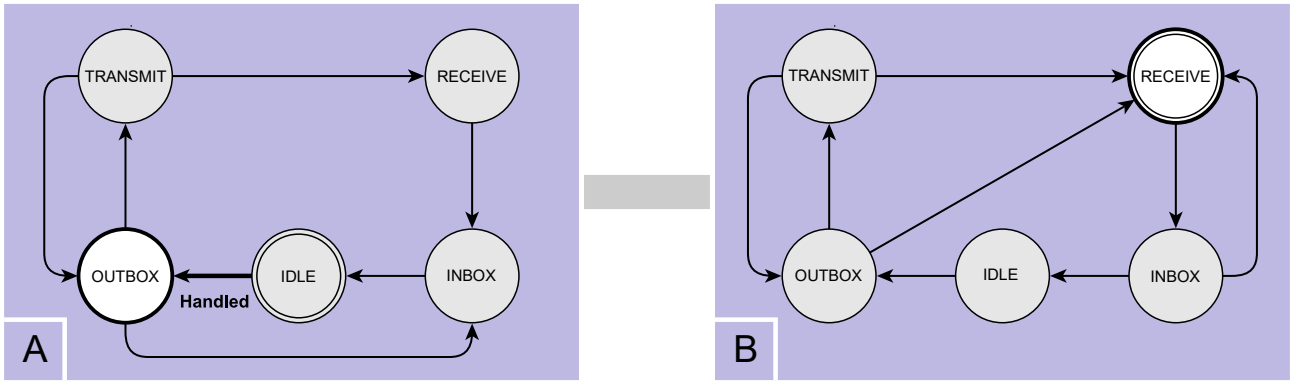
9.2.4 Beispiel einer Kommunikationssequenz

Zum Abschluss dieses Kapitels ein Beispiel für das Zusammenspiel der Automaten im Rahmen einer Kommunikationssequenz.

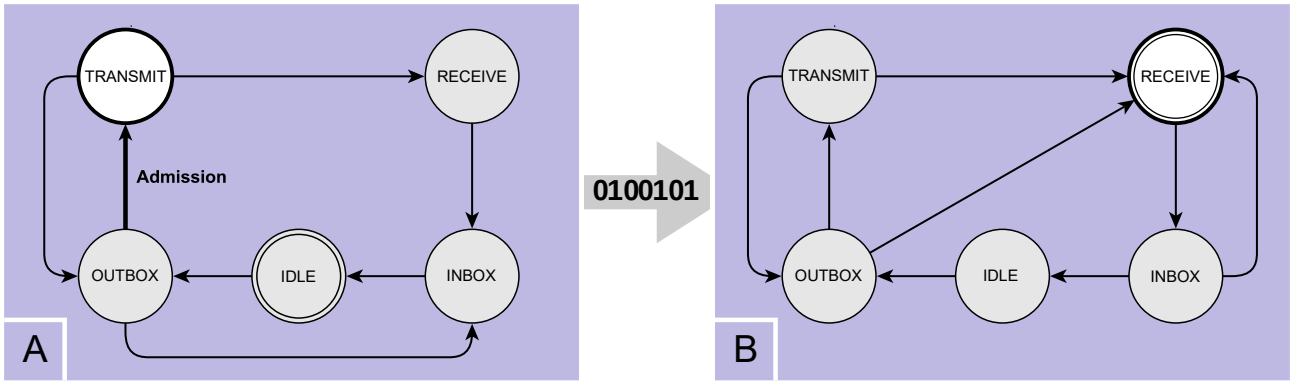
Wie im Beispiel aus Kapitel 7 sendet dazu wieder Gerät A eine Anfrage an B. In A betrachten wir dazu den Automat für die ausgehende Kommunikationssequenz, in B wird der für die eingehende Sequenz gezeigt. Ob es sich dabei um eine Leseanfrage oder Schreibaufforderung handelt, ist für den Ablauf der Kommunikationssequenz bzw. ihrer Darstellung unerheblich.



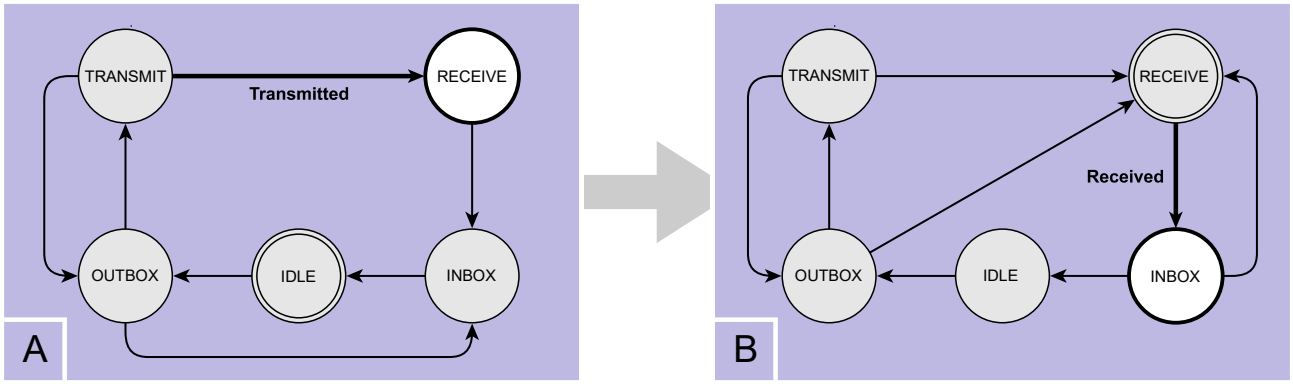
Zu Beginn befinden sich beide Automaten in ihren Ausgangszuständen: IDLE bzw. RECEIVE.



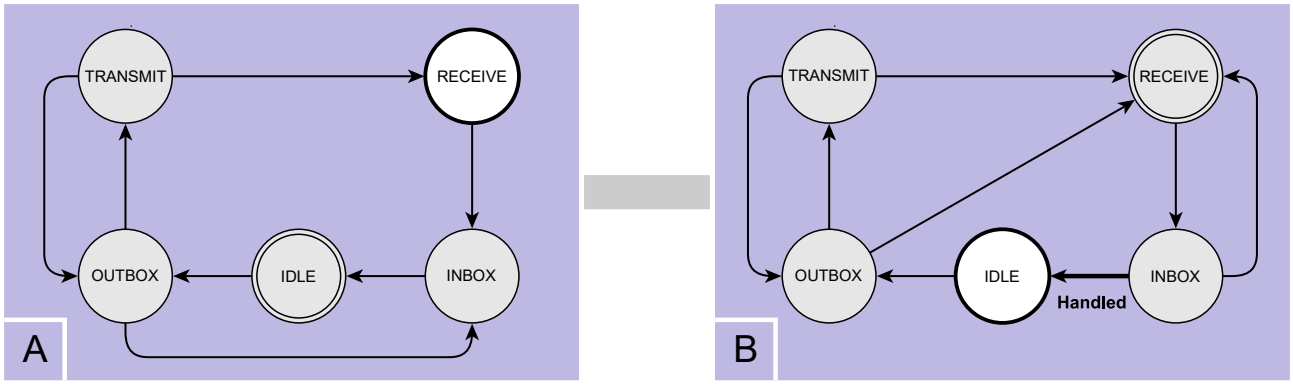
Das umgebende System erteilt A nun einen Sendeauftrag. Das löst Handled aus und A wechselt in den Zustand OUTBOX. B ist bislang noch nicht involviert.



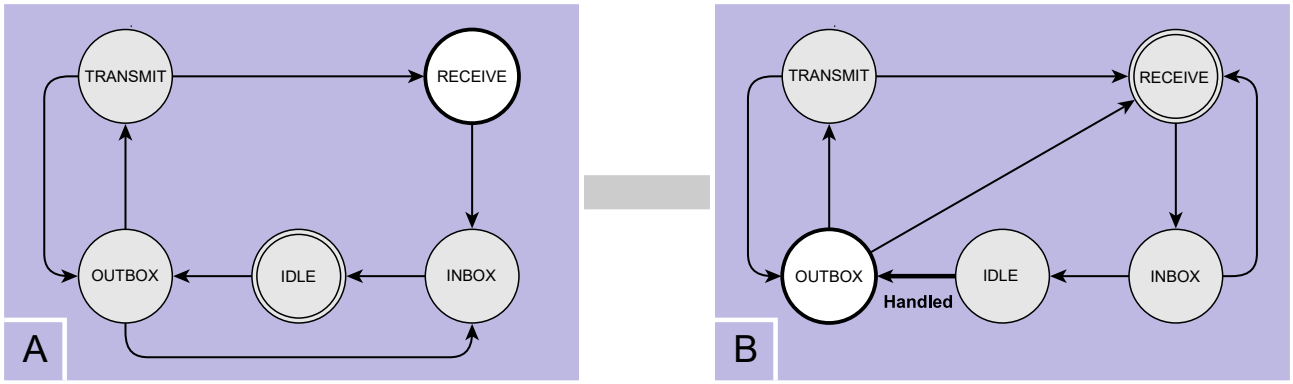
Mit der Freigabe zur Übertragung begibt sich A nun in den Zustand TRANSMIT. Damit beginnt die Übertragung der Anfrage-PDU auf dem Bus. Die Übertragungseinheit von B nimmt die Daten im Hintergrund bereits entgegen, der Automat B befindet sich weiterhin im Zustand RECEIVE.



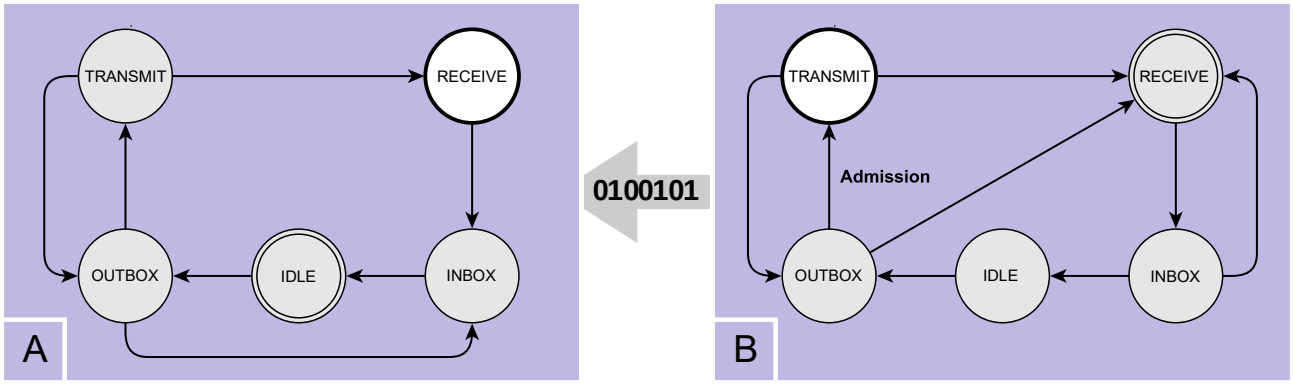
Die erfolgreiche Übertragung der Anfrage löst in A das Ereignis Transmitted aus. Zeitgleich wird in B das Ereignis Received ausgelöst. Das bedeutet: Eine Anfrage ist eingegangen und der Automat B wechselt in den Zustand INBOX.



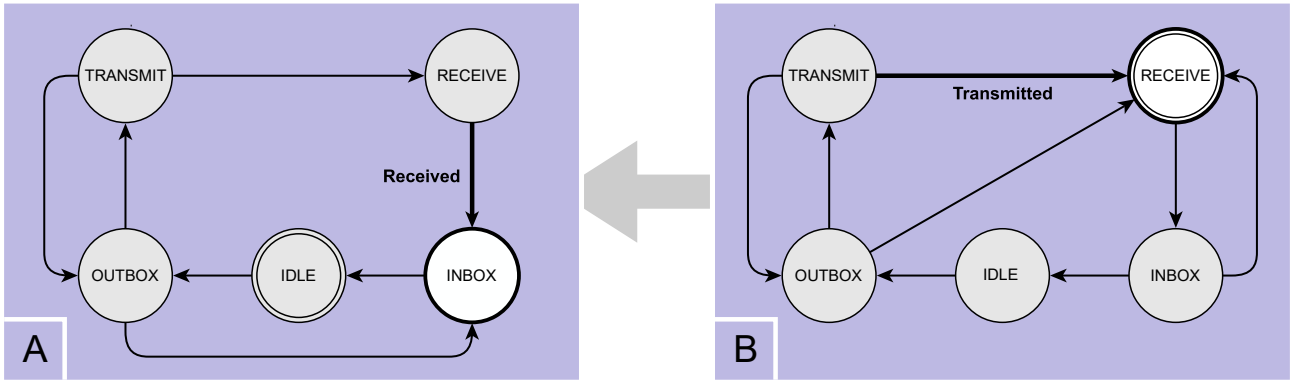
Die Übertragungseinheit in B hat die Anfrage dem System zur Abholung bereitgestellt. Das System nimmt die Anfrage entgegen, was das Ereignis Handled in B auslöst. B wechselt zu IDLE, A wartet auf die Antwort in RECEIVE.



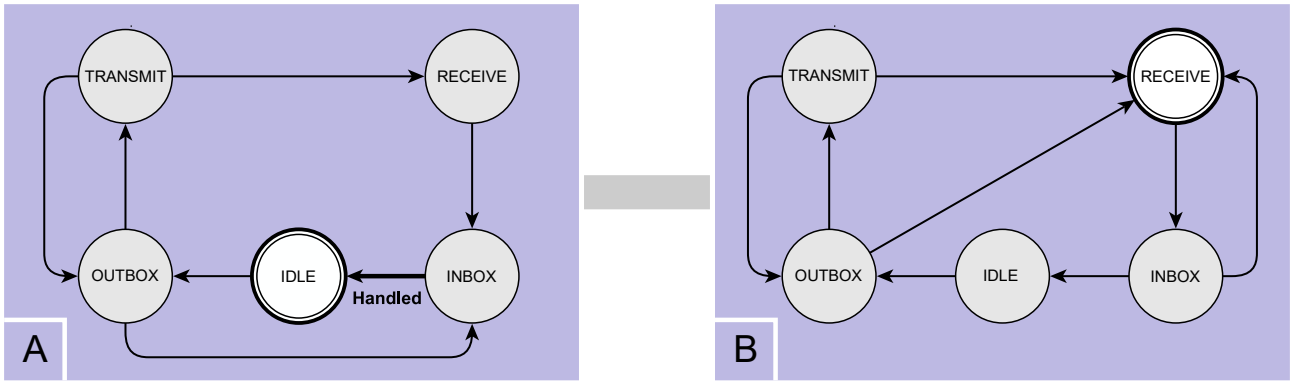
Das System hat die Anfrage bearbeitet und eine entsprechende Antwort an die Übertragungseinheit übergeben. Mit Handled wechselt B nun nach OUTBOX, um die Antwort an A zurückzusenden. A wartet weiterhin in RECEIVE.



Mit dem Ereignis Admission startet nun die Rückübertragung der Antwort von B nach A. B befindet sich im Zustand TRANSMIT. Jetzt ist es die Übertragungseinheit von A, die Daten im Hintergrund aufsammelt.



Der Abschluss der Übertragung löst in B das Ereignis Transmitted aus und führt B in seinen Ausgangszustand RECEIVE zurück. B hat damit die Anfrage beantwortet, die Sequenz ist abgeschlossen. In A löst das Eintreffen der Antwort Received aus. A nimmt die Antwort entgegen und wechselt in den Zustand INBOX.



Für den Abschluss der Kommunikationssequenz ist nun wieder das System A zuständig. Mit Entgegennahme der Antwort im zugehörigen Automaten wird das Ereignis Handled ausgelöst und A kehrt ebenfalls in seinen Ausgangszustand zurück. Damit ist auch für A die Sequenz beendet.

Was in der Darstellung des Beispiels nicht erwähnt wurde: Busstationen, die die Multimaster-Fähigkeiten nutzen, verfügen üblicherweise über beide Automaten, also den für die ein- wie auch den für die ausgehende Kommunikationssequenz. Beide Automaten laufen, bis auf die Synchronisation im Moment des Buszugriffs, asynchron.

10 Referenzimplementierung

10.1 Einleitung

Das Cy4NET-System ist darauf ausgelegt auf kleinen Controllersystemen zum Einsatz zu kommen. Die Referenzimplementierung basiert auf MCUs der Atmega-Reihe. Es sind 8-Bit-Controller des Herstellers Microchip [10]. Sie tragen ihren Namen nach dem ehemaligen Hersteller Atmel, der 2016 vom Chiphersteller Microchip übernommen wurde. Als Basis für die Referenzimplementierung dient Anwendung A00272 [11], die für das Board B00120 zur Verfügung steht. Das Boards ist in Anhang D dokumentiert. Ebenfalls von Microchip bereitgestellt wird die freie Entwicklungsumgebung Microchip-Studio [12], die bis November 2020 noch unter dem Atmel-Studio geführt wurde.

Als Entwicklungswerkzeug dient die Quelloffene GNU-Toolchain für AVM-Controller, V3.6.2 [13]. Sie ist in der Version 7 der Entwicklungsumgebung bereits integriert. Implementierungssprache ist C ab Standard C99, der eingesetzte Compiler hat die Version 5.4.0. Die Erläuterung aller Codezeilen der Referenzimplementierung würde den Rahmen dieser Dokumentation sprengen. Vorgestellt werden daher nur für das Verständnis wichtige Funktionseinheiten sowie die Signaturen des Cy4NET-APIs. Für tiefer gehende Untersuchungen steht unter der o.g. Referenz der gesamte Quellcode der Referenzimplementierung als Microchip Studio-Projekt bereit.

Controller dieser Klasse verfügen nicht über ausreichend Ressourcen um darauf ein Betriebssystem einsetzen zu können, d.h. sie werden meist *bare-metal* eingesetzt. Gemeint ist damit das Fehlen eines klassischen Betriebssystemes, wie man es von leistungsfähigeren, eingebetteten Systemen bzw. Prozessorboards her kennt. Folge davon ist, dass viele Grundfunktionen, die ein Betriebssystem mitbringt auf den Controllerboards per se nicht vorhanden sind und gegebenenfalls gesondert implementiert werden müssen. Im Gegenzug dazu erhält man für Controllerboards ein sehr kompaktes und auf die jeweilige Aufgabe hin optimiertes System. Das betrifft nicht nur das Startverhalten, Controllersysteme sind in Bruchteilen von Sekunden einsatzbereit, auch die Leistungsaufnahme ist im Vergleich zu größeren Prozessorboards deutlich geringer. Das Fehlen eines Betriebssystems spart Energie. Zum Vergleich: Das in Anhang G vorgestellte Evaluierungsboard B00101 hat im Regelbetrieb mit der Cy4NET-Standardsoftware eine Leistungsaufnahme von ca. 0,34 Watt (14 mA bei 24V). Ein Raspberry Pi 3B+ hat im Idle-Modus, also im Leerlauf ohne Grafik oder Netzwerkaktivität, bereits eine um den Faktor 6 höhere Leistungsaufnahme [14].

Selbstverständlich ist der Vergleich ungerecht. Schließlich verfügt der Raspi 3 über einen 64-Bit-SoC mit vier Kernen, das Evaluierungsboard B00101 hingegen ist nur mit einem ATmega32 ausgestattet. Andererseits geht es in unserem Zusammenhang um einfache Dienste, für deren Bereitstellung die Busstationen herangezogen werden. Prozessorboards mit wesentlich höherer Performance, können ihren Vorteil nur ausspielen, wenn ihre Leistungsfähigkeit auch abverlangt wird, wie das beispielsweise bei Protokollen wie MQTT [15] der Fall ist. Werden sie jedoch in vergleich-

barer Weise wie die Controllerboards eingesetzt, bleibt am Ende nur eine höhere Verbrauchsbilanz.

10.2 Schichten

Die Referenzimplementierung für das B00120 trägt die Applikationstypennummer A00272. Die Applikation stellt kein komplettes Betriebssystem bereit, jedoch sind einige betriebssystemtypische Softwarestrukturen vorhanden. Allen voran gibt es eine Hardwareabstraktionsschicht (HAL, Hardware Abstraction Layer). Realisiert ist sie im Softwaremodul Board. Dieses Modul wird für jedes Gerät individuell entworfen oder angepasst. Es bietet neben den Standardfunktionalitäten auch die Schnittstelle zu hardwarespezifischen Eigenarten und Mechanismen des Gerätes.

Auch das Cy4NET-Modul ist als Teil des Betriebssystems zu verstehen, ebenso wie Zeitgeber oder Funktionen für den Speicherzugriff. Abbildung 15 zeigt den prinzipiellen Schichtaufbau einer Cy4NET-Applikation. Je nach Funktionsreichtum der Busstationen kommen weitere Module hinzu.

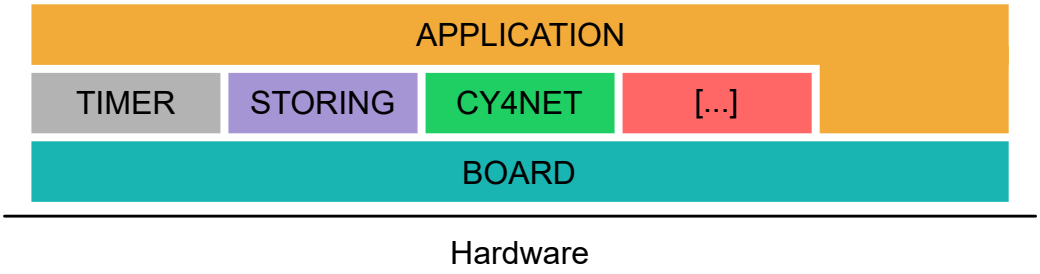


Abbildung 15: Softwareschichen einer Cy4NET-Anwendung

10.3 Implementierung

10.3.1 Die Hauptschleife

Zentrale Stelle der Betriebssoftware ist die Hauptschleife. Sie findet sich in `main()` und versorgt alle beteiligten Module mit Rechenzeit.

```
void
main(void)
{
    BOARD_Startup();
    STORING_Startup();
    CY4NET_Startup();
    APPLICATION_Startup();
    while (BOARD_DoMainLoop())
    {   BOARD_Proceed();
        STORING_Proceed();
```

```
        CY4NET_Proceed();
        APPLICATION_Proceed();
    }
    APPLICATION_Shutdown();
    CY4NET_Shutdown();
    STORING_Shutdown();
    BOARD_Shutdown();
}
```

Im Gegensatz zu präemptivem Multitasking, wo ein *Scheduler* die Rechenleistung unter den Prozessen verteilt, ist das System hier darauf angewiesen, dass die Module kooperieren. Das bedeutet, sie dürfen in ihrer jeweiligen Verfahrensfunktion `Proceed()` die Kontrolle nur für kurze Zeit beanspruchen um sie möglichst schnell wieder an die Hauptschleife abzugeben. Nacheinander erhalten so alle betriebsrelevanten Module die Möglichkeit, ihre Arbeiten durchzuführen. Flankiert wird die Hauptschleife von den Initialisierungs- bzw. Abschaltungsfunktionen der Module, `Startup()` und `Shutdown()`.

Des Weiteren sei die Konfiguration der Applikation in `Conf.h` erwähnt. Neben Cy4NET-Spezifischen Werten, wie beispielsweise die Einträge des Typenschildes, sind dort die Kapazitäten der Sende- und Empfangsschlangen sowie Zeitgeberparameter festgelegt.

10.3.2 Das BOARD-Modul

Das Board-Modul ist die *HAL* des Gerätes. Als Schnittstelle zur Hardware muss sie neben den gerätespezifischen Funktionen auch die Betriebseinheiten des Cy4NETs bereitstellen. In erster Linie betrifft das die Datenein- und ausgabe zur seriellen Busverbindung. Dabei mach die Echtzeit- und Multimastereigenschaft des Protokolls die Kommunikation mit der Hardware deutlich komplizierter. Simple Sende- und Empfangsfunktionen reichen nicht aus. Während des Sendevorgangs muss der Datenstrom auf eventuell auftretende Kollisionen hin überprüft werden und beim Empfang ist das Zeitverhalten des Datenstroms zu berücksichtigen.

Vorausschickend sei gesagt, dass die Schnittstelle zur Hardware in Einklang mit der Cy4NET-Implementierung stehen muss. Auch wenn sich die grundlegende Verfahrensweise nicht ändert, kann es doch vorkommen, dass die API an eine modifizierten Cy4NET-Version angepasst werden muss. Aktuelle Cy4NET-Version ist 19.10, benannt nach dem Erscheinungsmonat. Zu dieser Version muss das Board-Modul folgende Funktionen bereitstellen:

```
bool BOARD_BusIsIdle(void)
```

Beschreibung: Die Funktion prüft, ob der Bus momentan frei ist. Der Bus gilt als frei, wenn eine Kommunikationspause von mehr `CFG_BUS_INTERBYTE_TIMEOUT` ms eintritt. Der Wert ist in `Conf.h` definiert und beträgt aktuell 3 ms. Dazu siehe auch `BOARD_BusIsReceiving()`.

Eingabe: –

Ausgabe: *true*, wenn der Bus frei ist.

```
void BOARD_BusTransmit(UInt08 byte)
```

Beschreibung: Sendet das Datenbyte in `byte` auf den Bus. Die Sendeeinheit ist durch den Zustandsautomaten in Abbildung 16 realisiert. Ausgangszustand ist IDLE. Der Automat ist unabhängig von den Zustandsautomaten der Cy4NET-Kommunikation aus Kapitel 9.2 zu betrachten.

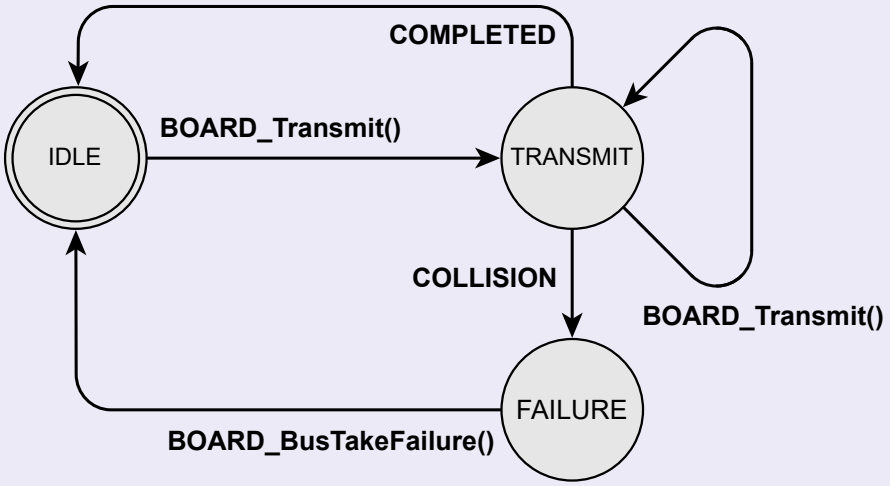


Abbildung 16: Zustandsautomat der Sendeeinheit

Mit Aufruf von `BOARD_BusTransmit()` wechselt der Automat von IDLE nach TRANSMIT und beginnt mit der Übertragung des angegebenen Datenbytes. Ist bereits eine Übertragung im Gange (Zustand TRANSMIT), wird bei Aufruf der Funktion das zu übertragende Datenbyte für die spätere Übertragung in die Sende-Warteschlange gestellt. Nach Aussendung aller zu übertragenden Datenbytes (Ereignis COMPLETED) kehrt der Automat wieder in den Zustand IDLE zurück.
Ereignet sich bei der Übertragung eine der in Kapitel 9.2.3 beschriebenen Kollisionen, wird das Ereignis COLLISION ausgelöst und die Sendeeinheit wechselt in den Zustand FAILURE. Die Übertragung wird in dem Fall nicht mehr fortgesetzt und die Sende-Warteschlange geleert. Gegebenenfalls noch ausstehende Bytes werden also nicht mehr übertragen. Im Zustand FAILURE bleiben Aufrufe von `BOARD_BusTransmit()` ohne Wirkung. Um neue Datenbytes senden zu können, muss der Automat erst durch Aufruf von `BOARD_BusTakeFailure()` wieder in den Zustand IDLE zurückversetzt werden.

Eingabe: `byte`: Das zu übertragende Datenbyte
Ausgabe: –

```
bool BOARD_BusIsTransmitting(void)
```

Beschreibung: Prüft, ob sich der Zustandsautomat der Sendeeinheit aus Abbildung 16 im Zustand TRANSMIT befindet. Das ist solange der Fall, wie die Sendeeinheit mit der Übertragung von Datenbytes beschäftigt ist.

Eingabe: –
Ausgabe: *true*, solange der Sendevorgang im Gange ist.

```
bool BOARD_BusTakeFailure(void)
```

Beschreibung: Prüft, ob sich der Zustandsautomat der Sendeeinheit aus Abbildung 16 im Zustand FAILURE befindet. Ist das der Fall, löst der Aufruf dieser Funktion den Wechsel des Automaten in den Zustand IDLE aus und ermöglicht damit die Übertragung neuer Datenbytes.

Eingabe: –
Ausgabe: t*rue*, wenn ein Übertragungsfehler vorlag.

```
bool BOARD_BusReceive(UInt08* byte)
```

Beschreibung: Steht bei Aufruf der Funktion ein empfangenes Datenbyte zur Abholung bereit, wird es aus der Empfangsschlange entfernt und in `byte` übergeben. `BOARD_BusReceive()` kehrt so lange mit *true* zurück, wie Datenbytes zur Abholung bereitstehen.
Man beachte: Bei einer Unterbrechung der Empfangssequenz kehrt die Funktion mit *false* zurück. Umgekehrt gilt das jedoch nicht: Der Rückgabewert *false*, bedeutet nicht zwingend, dass auch die Empfangssequenz bereits unterbrochen ist! Dazu siehe `BOARD_BusIsReceiving()`.

Eingabe: `byte`: Referenz, die das nächste zur Abholung bereitstehende Datenbyte aufnimmt.
Ausgabe: *true*, wenn in `byte` ein Datenbyte abgelegt wurde.

```
bool BOARD_BusIsReceiving(void)
```

Beschreibung: Die Funktion prüft, ob die aktuelle Empfangssequenz noch im Gange ist.
Eine Gruppe empfangener Datenbytes gilt als zusammenhängende Empfangssequenz, wenn die Zeitspanne zwischen dem Empfang sukzessiver Datenbytes einen bestimmten Zeitrahmen nicht überschreitet. Mit anderen Worten: Liegen die Zeitpunkte zweier aufeinanderfolgend empfangener Datenbytes länger als `CFG_BUS_INTERBYTE-`

_TIMEOUT ms auseinander, gilt die Empfangssequenz als unterbrochen. Siehe dazu auch BOARD_BusIsIdle() und Kapitel 8.5.
Die Unterbrechungen werden mit in der Empfangsschlange vermerkt, so dass beim Auslesen der empfangenen Bytes mittels BOARD_BusReceive() die Empfangssequenzen zeitunabhängig separiert werden können. Solange BOARD_BusIsReceiving() mit *true* zurückkehrt, ist die Empfangssequenz noch im Gange.

Eingabe: –
Ausgabe: *true*, solange die Empfangssequenz im Gange ist.

```
void BOARD_BusOnActivity(void)
```

Beschreibung: Rückruffunktion der Cy4NET-Übertragungseinheit. BOARD_BusOnActivity() wird vom Cy4NET-Modul aufgerufen, wenn mit dem Gerät über den Bus kommuniziert wird. Die Funktion hat im Rahmen der Kommunikation keine Wirkung, sondern wird üblicherweise dazu eingesetzt Aktivität auf dem Bus, z.B. mittels einer LED, kenntlich zu machen.

Eingabe: –
Ausgabe: –

```
UInt16 BOARD_MsecTicks(void)
```

Beschreibung: Die Funktion liefert den Stand eines monotonen 16-Bit Millisekunden-Zeitzählers. Kommt es nach 65535 ms zum Überlauf, beginnt der Zeitzähler wieder bei 0.

Eingabe: –
Ausgabe: Aktueller Stand des Millisekunden-Zeitzählers: 0 .. 65535.

Für das Senden von Datenybtes ist also die Funtionsgruppe bestehend BOARD_BusTransmit(), BOARD_BusIsTransmitting() und BOARD_BusTakeFailure() zuständig. In einer Schleife werden zuerst alle zu sendenden Datebytes mittels BOARD_BusTransmit() der Übertragungseinheit übergeben. Da die Übergabe auszusendender Datenbytes üblicherweise um ein Vielfaches schneller geht als die eigentliche Übertragung auf dem Medium, werden ausstehende Datenbytes in einer Sendeschlange aufgereiht. Ein Interrupt kümmert sich um die Übertragung. Sollte während dessen ein Fehler auftreten, wird der Sendevorgang unmittelbar eingefroren. Neue Datenbytes werden nicht mehr entgegengenommen und noch ausstehende gelöscht. Weitere Aufrufe von BOARD_BusTransmit() bleiben ohne Wirkung. Erst der Aufruf von BOARD_BusTakeFailure() entsperrt den Sendevorgang wieder. Die Funktion berichtet den Fehler nach Außen und macht den Weg frei für weitere Sendeaufträge.

Sind alle gesendenden Datebytes an die Übertragungseinheit übergeben worden, kann mittels `BOARD_BusIsTransmitting()` festgestellt werden, wann alle Datenbytes die Einheit verlassen haben und der eigentliche Transportvorgang abgeschlossen ist.

Im Gegensatz zum Sendevorgang, der seitens der Anwendung explizit in Gang gesetzt wird, kann Datenempfang zu jedem beliebigen Zeitpunkt stattfinden. Eingehende Bytes werden entgegengenommen und zur Abholung in einem FIFO-Speicher, der Empfangsschlange, bereitgestellt. Wie beim Sendevorgang wird das auch hier mittels eines Interrupts gehandhabt.

Die Schnittstelle zur Abholung empfangener Datenbytes bilden die Funktionen `BOARD_BusReceive()` und `BOARD_BusIsReceiving()`. Solange Datenbytes in der Empfangsschlange anstehen, können sie mittels der ersten Funktion Byte für Byte abgeholt werden. Dabei wird zwar die Reihenfolge des Dateneingangs aufrechterhalten nicht jedoch das Zeitverhalten abgebildet. Zwischen Empfang eines Datenbytes und dessen Abholung durch `BOARD_BusReceive()` kann eine unbestimmte Zeitspanne liegen, zumindest theoretisch. Beiden Aktionen finden asynchron statt.

Die Frage hier ist nicht, wann das eingegangene Datenbyte die Empfangseinheit erreicht hat, sondern ob zwischen dem Empfang zweier Datenbytes die in Kaptiel 8.5 erwähnte *Interbyte-Timeout* aufgetreten ist. Eine solche Unterbrechung der Empfangssequenz würde Folgen für die Verarbeitung der PDU nach sich ziehen.

Aus diesem Grund prüft und vermerkt die Empfangseinheit Zeitlücken beim Eingang der Datenbytes. Vergeht nach Empfang eines Datenbytes mehr als die in `Conf.h` angegebene Zeit von `CFG_BUS_INTERBYTE_TIMEOUT` ms, gilt die Empfangssequenz als unterbrochen. Diese Zeitüberschreitung wird von der Empfangseinheit mit in der Schlange vermerkt, so dass später, bei Abholung der Datenbytes, nachträglich festgestellt werden kann, ob eine Zeitlücke entstanden ist. Dabei ist nicht Länge der Pause zwischen zwei Bytes von Interesse, sondern lediglich die Tatsache ihres Auftretens. Diesem Zweck dient die zweite Funktion: `BOARD_BusIsReceiving()`. Solange die gerade abgefragte Sequenz nicht unterbrochen ist, liefert die Funktion *true* zurück. Man beachte, dass `BOARD_BusIsReceiving()` sich dabei auf die aktuell abgefragte Sequenz bezieht, also nicht etwa über die aktuelle Empfangssituation berichtet. Beispiel:

```
UInt08 sequence[SEQUENCE_CAPACITY];
UInt08 index = 0;

while (…)
{  UInt08 byte;
   while (BOARD_BusReceive(&byte))
   {   // Empfang einer Sequenz im Gange
       if (index < sizeof(sequence))
       {   sequence[index] = byte;
           index++;
       }
       else
       {   // Überlauf …
       }
   }
```

```
    if (BOARD_BusTakeFailure())
    {   // Empfangsfehler aufgetreten, Sequenz verwerfen
        index = 0;
        HandleFailure();
    }
    else if (BOARD_BusIsReceiving())
    {   // Sequenz läuft, aber aktuell stehen keine weiteren Datenbytes zur Abholung
        // bereit
    }
    else
    {   // Sequenz unterbrochen, empfangene Sequenz verarbeiten und Neue starten
        DoSomething(sequence);
        index = 0;
    }
}
```

Um eingegangene Sequenzen eindeutig separieren zu können, muss immer, wenn keine neuen Datenbytes zur Abholung anstehen geprüft werden, ob eine Zeitüberschreitung stattgefunden hat, ob also die Sequenz unterbrochen wurde. Das Gleiche gilt übrigens ebenso für das Auftreten eines Fehlers.

10.3.3 Das Cy4NET-Modul

Unmittelbar auf die HAL setzt das Cy4NET-Modul auf. In ihm ist das Protokoll umgesetzt. Die elf Quellcode-Dateien des Moduls finden sich im Unterverzeichnis `Cy4NET` des jeweiligen Projektes.

```
Cy4NET
 |
 +-- Cy4Address.c
 +-- Cy4Address.h
 +-- Cy4Core.c
 +-- Cy4Core.h
 +-- Cy4Message.c
 +-- Cy4Message.h
 +-- Cy4Net.c
 +-- Cy4Net.h
 +-- Cy4Service.c
 +-- Cy4Service.h
 +-- Cy4TypePlate.h
```

Das Modul selbst teilt sich in zwei Übersetzungseinheiten auf: `Cy4Net` und `Cy4Core`. Ergänzend gibt es weitere Datenstrukturen, die bei der Nutzung des Protokolls zum Einsatz kommen. Dazu gehört die Adresse `Cy4Address`, die Nachricht `Cy4Message` und der Dienst `Cy4Service`. In `Cy4TypePlate.h` ist das Makro `CY4TYPEPLATE` definiert. Es bietet Unterstützung bei der Zusammenstellung des Cy4-Typenschildes gemäß des in Kapitel 6.3 vorgegebenen Formates. `CY4TYPEPLATE` kommt üblicherweise im Bearbeiter des Pflichtdienstes *CallTypePlate* zum Einsatz.

Um mit dem Cy4NET-System in Kontakt zu treten, ist es nicht nötig PDUs händisch zusammenzustellen oder anderweitig zu verarbeiten. Der Datenaustausch basiert auf Objekten des Typs `Cy4Message`. Dieses Nachrichtenobjekt führt alle für die Kommunikation nötigen Bestandteile mit

sich: Die Adresse des Empfängers bzw. die des Senders, der Dienstort und die Nutzlast. Zusätzlich ist dort die Zugriffsart vermerkt, d.h. lesend oder schreibend, sowie ein *Flag*, mittels dessen die Nachricht für gültig bzw. ungültig erklärt werden kann. Ob es sich bei einer Nachricht um eine Anfrage oder eine Antwort handelt, ergibt sich eindeutig aus dem jeweiligen Kontext.

10.3.3.1 Der Kern

Die wesentliche Funktionseinheiten zur Kommunikation im Cy4NET sind in `Cy4Core` untergebracht, dem sogenannten Kern. Auch die in Kapitel 9.2 vorgestellten Automaten für eingehende und ausgehende Kommunikationssequenzen sind im Kern angesiedelt. Damit er diese Aufgaben bewerkstelligen kann, setzt der Kern direkt auf die HAL auf. Das bedeutet, alle Schnittstellenfunktionen des Board-Moduls kommen hier zum Einsatz.

Unter den Funktionen des Kernmoduls gibt es vier exponierte, mittels derer die Kommunikation bewerkstelligt wird. Zwei davon sind für die eingehende, zwei für die ausgehende Kommunikation zuständig. Beginnen wir mit der Ausgehenden:

```
bool CY4CORE_PostRequest(const Cy4Message* message)
```

Beschreibung: Initiiert das Senden einer Anfrage. Die Funktion kehrt unmittelbar zum Aufrufer zurück. Die Übertragungseinheit ist in der Lage stets nur eine einzelne Anfrage-Antwort-Sequenz zu handhaben. In der Folge werden weitere Anfragen so lange abgewiesen, wie eine laufende Übertragungssequenz noch nicht abgeschlossen ist. Den Abschluss einer Übertragungssequenz bildet das Abholen der Antwort mittels `CY4CORE_TakeResponse()`.

Eingabe: `message`: Die zu sendende Anfrage-Nachricht.
Ausgabe: *true*, wenn die Nachricht zur Absendung entgegengenommen wurde.

```
bool CY4CORE_TakeResponse(Cy4Message* message)
```

Beschreibung: Jede von `CY4CORE_PostRequest()` entgegengenommene Anfrage wird beantwortet. Steht bei Aufruf dieser Funktion eine Antwort zur Abholung bereit, wird sie entgegengenommen und in `message` abgelegt. War die Kommunikation erfolgreich, ist die Antwort-Nachricht gültig und führt die gewünschten Daten mit sich. Andernfalls erhält man als Antwort eine ungültige Kopie der Anfrage.
Solange eine bereitstehende Antwort nicht abgeholt ist, gilt die Übertragungssequenz als nicht abgeschlossen und `CY4CORE_PostRequest()` wird die Annahme neuer Anfragen verweigern.

Eingabe: `message`: Referenz, die eine eventuell bereitstehende Antwort aufnimmt. Steht zum Zeitpunkt des Aufrufs keine Antwort zur Abholung bereit, ist die in `message` abgelegte Nachricht ungültig.
Ausgabe: *true*, wenn in `message` eine abgeholte Antwort abgelegt wurde.

Um eine Anfrage zu senden, muss als Erstes ein `Cy4Message`-Objekt erzeugt und ausgestattet werden. Dazu dienen die `CY4MESSAGE_Setup…()`-Funktionen der gleichnamigen Datenstruktur. Angegeben werden muss die Adresse des Empfängers, der Dienstort sowie die Anzahl zu lesender bzw. die zu schreibenden Daten. Das entsprechend ausgestattete Objekt wird dann der Funktion `CY4CORE_PostRequest()` übergeben. Dabei ist auf den Rückgabewert der Funktion zu achten. Nur wenn die Funktion mit *true* wiederkehrt, wurde die Anfrage von der Übertragungseinheit auch entgegengenommen.

Beispiel: Anfrage des Luftdrucks aus Kapitel 7

```
Cy4Message message;
Cy4Address address = CY4ADDRESS(24,17);
CY4MESSAGE_SetupRead(&message, address, 0x100210, 2);
if (CY4CORE_PostRequest(&message))
{   // Anfrage wurde entgegengenommen …
}
```

Jede durch `CY4CORE_PostRequest()` entgegengenommene Anfrage wird beantwortet. Unabhängig davon, ob die Übertragung erfolgreich war oder nicht, wird nach spätestens 500 ms eine Antwort zur Abholung erscheinen. Diese muss mittels `CY4CORE_TakeResponse()` abgenommen werden. Solange das nicht geschehen ist, gilt die Kommunikationssequenz als nicht abgeschlossen. Da die Übertragungseinheit in der Lage ist, stets nur eine einzelne Sequenz zu behandeln, wird sie die Annahme weiterer Anfrage solange verweigern, wie noch eine Antwort einer vorausgehenden Anfrage auf Abholung wartet.

`CY4CORE_TakeResponse()` liefert wieder ein Objekt vom Typ `Cy4Message` zurück. Dieses Mal ist es die Antwort. Die Gültigkeit der Nachricht zeigt dabei an, ob die Kommunikation erfolgreich war oder nicht. Ist die Antwort gültig bedeutet das je nach Art der Anfrage: Der Schreibauftrag ist bestätigt oder die Leseanforderung war erfolgreich. Im zweiten Fall finden sich die gelesenen Daten im Nutzlastbereich der Nachricht.

Beispiel: Antwort auf eine Anfrage

```
if (CY4CORE_TakeResponse(&message))
{   // Antwort erhalten ...
    if (CY4MESSAGE_IsValid(&message))
    {   // Kommunikationssequenz war erfolgreich …
    }
}
```

Für die eingehende Übertragungssequenz, also die Entgegennahme einer Anfrage, sind die beiden folgenden Funktionen des Kernes zuständig. Ihr Aufbau ist in gewisser Weise symmetrisch zu den Funktionen der ausgehenden Sequenz.

bool CY4CORE_TakeRequest(Cy4Message*message)

Beschreibung: Empfängt die Übertragungseinheit eine Anfrage, dient diese Funktion dazu die Anfrage entgegenzunehmen. Eine eingegangene Anfrage muss angenommen, bearbeitet und die Antwort mittels `CY4CORE_PostResponse()` wieder zurückgeschickt werden.

Eingabe: `message`: Referenz, die eine bereitstehende Anfrage aufnimmt. Solange keine Anfrage zur Abholung bereitsteht, ist die in `message` abgelegte Nachricht ungültig.

Ausgabe: *true*, wenn eine Anfrage zur Abholung bereitstand, wenn also `message` eine gültige Anfrage erhalten hat.

bool CY4CORE_PostResponse(const Cy4Message* message)

Beschreibung: Initiiert das Senden einer Antwort auf eine zuvor mittels `CY4CORE_TakeRequest()` entgegengenommene Anfrage. Kann die Anfrage, aus welchem Grund auch immer, vom System nicht bearbeitet werden, ist die übergebende Antwort-Nachricht vor der Übergabe für ungültig zu erklären.

Eingabe: `message`: Die zu sendenden Antwort-Nachricht.

Ausgabe: *true*, wenn die Nachricht zur Absendung entgegengenommen wurde, *false*, wenn die Übertragungseinheit noch mit dem Versenden einer anderen Antwort beschäftigt ist.

Das System erhält eine Anfrage durch Aufruf von `CY4CORE_TakeRequest()`. Sobald die Funktion als Rückgabewert *true* liefert, ist eine Anfrage in Form einer `Cy4Message` eingegangen. Daraufhin muss das System die Anfrage bearbeiten. Das bedeutet, den entsprechenden Dienst zu lokalisieren und je nach Zugriffsart Daten aus der Nachricht aus- oder einzuladen oder auch Aktionen auszuführen. Den Abschluss der Behandlung bildet der Aufruf `CY4CORE_PostResponse()`. Der Funktion wird wieder ein `Cy4Message`-Objekt übergeben, dieses Mal ist es die Antwort. Gängige Praxis bei der Bearbeitung ist es, die eingegangene Nachricht *in situ* zu bearbeitet und als Antwort wieder zurückzuschicken.

Beispiel: Behandlung des Pflichtdienstes *EnterProgramLoader* aus Kapitel 6.3

```
Cy4Message message;
if (CY4CORE_TakeRequest(&message))
{   // Anfrage erhalten, passenden Dienst suchen
    if (CY4MESSAGE_Location(&message) == 0xFFFFFA)
    {   // Pflichtdienst "EnterProgramLoader"
        if (CY4MESSAGE_Access(&message) == CY4MESSAGE_WRITE &&
            CY4MESSAGE_ConstData(&message)[0] == 0x5E)
        {   // Schreibaufforderung mit korrektem Auslösewert erhalten
            BOARD_EnterPgmLdr();
```

```
        }
        else
        {   // Zugriffsart und/oder Auslösewert falsch
            CY4MESSAGE_Invalidate(&message);
        }
    }
    else if (/* weitere Dienste */)
    // …
    else
    {   // Kein passender Dienst vorhanden
        CY4MESSAGE_Invalidate(&message);
    }
    CY4CORE_PostResponse(&message);
}
```

Zu beachten ist, dass jede anstehende Anfrage abgeholt und beantwortet werden muss. Kann die abgeholte Anfrage nicht bearbeitet werden, weil kein passender Dienst zur Verfügung steht oder Parameterwerte nicht stimmen, ist das zurückgegebene `Cy4Message`-Objekt vor Übergabe für ungültig zu erklären.

Der Rückgabewert von `CY4CORE_PostResponse()` informiert darüber, ob die Übertragungseinheit noch mit dem Versenden einer anderen Antwort beschäftigt ist. Im Beispiel oben kann auf diese Prüfung verzichtet werden. Da Entgegennahme der Anfrage und Rücksendung der Antwort innerhalb des gleichen Kontrollflusses stattfinden, kann es nicht passieren, dass die Übertragungseinheit anderweitig belegt ist. Finden Abnahme und Rücksendung jedoch getrennt statt, ist die Prüfung der Bereitschaft obligatorisch.

10.3.3.2 Die Schale

Das Kernmodul allein würde ausreichen, um die Kommunikation mit dem Cy4NET-System durchführen zu können. Die reine Kommunikation. Bislang unzureichende Unterstützung findet sich hingegen für die höher angesiedelten Protokollebenen. Erwähnt seinen da beispielsweise die Dienste, insbesondere die Pflichtdienste, aber auch die Handhabung der ausgehenden Kommunikation. Diese Themen rein mittels Funktionen des `Cy4Core` abzuhandeln wäre zwar möglich, jedoch verhältnismäßig umständlich. Da sich bestimmte Abläufe stets wiederholen, liegt es nahe, die Schnittstelle um eine Ebene zu erweitern um die Kommunikation komfortabler gestalten zu können. Diese Aufgabe übernimmt die Schale. Umgesetzt ist sie in der zweiten Übersetzungseinheiten des Cy4NET-Moduls, namens `Cy4Net`.

Dass Modul und Übersetzungseinheit hier den gleichen Namen tragen, hängt damit zusammen, dass die API (*Application Programming Interface*) des Cy4NET-Protokolls später allein durch Schnittstellenfunktionen der Schale, also dem `Cy4Net`, abgedeckt werden. Nur in Ausnahmefällen wird es nötig sein, auf Funktionen des Kerns zurückgreifen.

Die Terminologie von Kern und Schale suggeriert, dass das eine den inneren Teil des anderen darstellt. Das ist nicht ganz korrekt, denn beide Teile bauen wie Ebenen aufeinander auf. Die Basis bildet `Cy4Core`, darauf aufgesetzt ist `Cy4Net`.

Wie bei allen Betriebsmodulen wird der aktive Teil des Moduls in der Hauptschleife von `main()` behandelt. Dazu finden sich neben den Initialisierungsfunktionen `CY4NET_Startup()` und `CY4NET_Shutdown()` die periodisch aufgerufene Verfahrensfunktion `CY4NET_Proceed()`. Dieser Mechanismus existiert übrigens in gleicher Weise auch für den Kern. Grund dafür, dass er oben unerwähnt blieb, ist, dass mit Einsatz des `Cy4Net` die Prozessaufrufe des `Cy4Core` mit abgehandelt werden. Der explizite Aufruf in `main()` wäre damit überflüssig bis störend.

10.3.4 Die API

Die Grundlage des Cy4NET-APIs bilden Rückrufe (*Callbacks*). Dabei werden Programmabläufe von einer aktiven, funktionalen Ebene auf eine passive, ereignisorientierte Ebene verlagert. Das bedeutet, dass die Kontrolle der Kommunikation im Cy4NET nicht durch direkte Funktionsaufrufe stattfindet, sondern indirekt. Dem Cy4NET-System werden dabei Funktionen übergeben, die bei Eintritt bestimmter Ereignisse aufgerufen, genauer gesagt, zurückgerufen werden. Basis der Rückrufe bilden drei Funktionszeiger, die in `Cy4Net.h` definiert sind:

```
// Cy4-Dienst Anfragebearbeiter
typedef void (*Cy4RequestHandler)(const Cy4Service*, Cy4Message*);

// Cy4-Nachrichten-Ausstatter
typedef void (*Cy4MessageSetup)(Cy4Message*);

// Cy4-Dienst Antwortbehandler
typedef void (*Cy4ResponseHandler)(const Cy4Message*);
```

Die Instanzen der Rückruffunktionen finden sich üblicherweise in Modulen höherer Ebenen, häufig im Application-Modul.

10.3.4.1 Die Handhabung von Diensten

Die Cy4NET-Kommunikation ist vergleichbar mit dem Prinzip der RPCs (*Remote Procedure Calls*): Anfragen werden ausgesendet, um an anderer Stelle Dienste auszuführen. In der Cy4NET-API wird ein Dienst durch einen entsprechenden Dienstbearbeiter repräsentiert. Jedem Dienst ist ein entsprechender Bearbeiter zugeordnet. Er ist für die Umsetzung aller mit dem Dienst zusammenhängenden Arbeiten zuständig. Sobald eine Anfrage das Cy4NET-System erreicht, sucht die Übertragungseinheit nach der passenden Bearbeiterfunktion und ruft sie auf. Definiert ist der *Dienst Anfragebearbeiter* in `Cy4RequestHandler`. Es ist der erste Funktionszeiger in `Cy4NET.h`. Als Parameter wird dem Bearbeiter der zugrundeliegende Dienst und die konkrete Anfrage mitgegeben.

Ein Cy4NET-Gerät bietet üblicherweise mehrere Dienste an. Neben den Pflichtdiensten sind das weitere, die die spezifischen Funktionen des Gerätes von außerhalb nutzbar machen. Damit das Cy4NET-System nun einer eintreffenden Anfrage einen passenden Dienstbearbeiter zuordnen kann, sind alle Dienste des Gerätes, mitsamt ihren Bearbeitern in einer Liste aufgeführt. Diese Liste wird zu Beginn an das Cy4NET-System übergeben. Die API-Funktion dazu heißt:

`void CY4NET_AssignServices(const Cy4ServiceItem* table)`

Beschreibung: Die Funktion übergibt dem Cy4NET-System die Dienste, die das jeweilige Gerät bereitstellt. Dazu erwartet die Funktion in `table` die Adresse eines Feldes mit Einträgen vom Typ `Cy4ServiceItem`. Jeder Eintrag beschreibt dabei einen Dienst bestehend aus: Der Dienstsignatur, also Dienstort, Größe und Zugriffsart und einer Bearbeiter-Rückruffunktion. Der Bearbeiter wird aufgerufen sobald bei der Übertragungseinheit eine zum Dienst passende Anfrage eingegangen ist.
Es ist zwingend, dass die Einträge der Tabelle nach Dienstadressen aufsteigend sortiert aufgelistet sind! Den Abschluss der Liste müssen immer die vier Pflichtdienste *CallTypePlate*, *EnterApplication*, *EnterProgramLoader* und *ChangeCy4Address* bilden. Die Tabelle wird im Programmspeicher des Controllers erwartet und darf bis zu 127 Einträge haben, Pflichtdienste eingerechnet. Es ist jederzeit möglich, die Dienste-Tabelle zu tauschen.

Eingabe: `table`: Adresse der Dienste-Tabelle (im Programmspeicher).
Ausgabe: –

Die Zuweisung der Dienstbearbeiter findet normalerweise in der Startphase, häufig in der Initialisierungsfunktion der Anwendung, also in `APPLICATION_Startup()`, statt. Erwartet wird die Liste der Dienste in Form einer Tabelle. Darin aufgeführt, die Dienste mitsamt ihren spezifischen Bearbeitern. In der Referenzimplementierung von Cy4Nut sieht das beispielsweise wie folgt aus:

```
void
APPLICATION_Startup(void)
{
    static const Cy4ServiceItem  servicesTable[] PROGMEM =
    {   //|                    service                      |        handler        |
        //+------------------+-----+-----------------------+-----------------------+
        { CY4SERVICE(0x100200,  1  , CY4SERVICE_READ_WRITE), OnInfoLedState        },
        { CY4SERVICE(0x100400,  6  , CY4SERVICE_READ_WRITE), OnPeriodicPushMessage },
        { CY4SERVICE(0xFFF020,  1  , CY4SERVICE_READ_WRITE), OnSystemLedTask       },
        { CY4SERVICE(0xFFFF00, 61  , CY4SERVICE_READ_WRITE), OnDeviceLabel         },
        { CY4SERVICE(0xFFFF40,  1  , CY4SERVICE_WRITE_ONLY), OnFactorySettings     },
        { CY4SERVICE(0xFFFFF0,  7  , CY4SERVICE_READ_ONLY ), OnCallTypePlate       },
        { CY4SERVICE(0xFFFFF8,  1  , CY4SERVICE_WRITE_ONLY), OnEnterApplication    },
        { CY4SERVICE(0xFFFFFA,  1  , CY4SERVICE_WRITE_ONLY), OnEnterProgramLoader  },
        { CY4SERVICE(0xFFFFFC,  2  , CY4SERVICE_WRITE_ONLY), OnChangeCy4Address    }
    };
    CY4NET_AssignServices(servicesTable);
    Cy4Address address = STORING_LoadCy4SelfAdr();
    CY4NET_SetAddress(address);
    SysLedTask task;
    STORING_Load(STO_SysLedTask, &task);
    BOARD_SetSysLedTask(task);
}
```

Zusätzlich dazu werden in `APPLICATION_Startup()` weitere Initialisierungsaufgaben erledigt, dazu gehört z.B. das Setzen der geräteeigenen Cy4NET-Adresse.

Mit Rücksicht auf die Harvard-Architektur des Controllers, ist die Tabelle im Programmspeicher untergebracht. Jede Zeile der Tabelle besteht aus einem Eintrag vom Typ `Cy4ServiceItem`. Mit Übergabe der Dienste-Tabelle an das Cy4NET-System ist die Arbeit für die Applikation getan. Von nun an werden eingehende Anfragen direkt durch das Cy4NET-System an die entsprechenden Dienstbearbeiter delegiert. Zuständig dafür ist die Funktion `HandleRequest()` im Inneren des Moduls Cy4Net. Da `HandleRequest()` die Tabelle per binärer Suche nach dem passenden Dienst durchforstet, ist es zwingend notwendig, dass die Einträge der Tabelle nach Dienstorten aufsteigend sortiert abgelegt sind. Die Suche hat damit eine Zeitkomplexität von $O(\log_2 n)$, d.h. bei einer Tabelle mit 50 Einträgen ist der passende Dienst nach spätestens 6 Iterationen gefunden. Die letzten Einträge bilden immer die vier Pflichtdienste mit *ChangeCy4Address* als Abschluss.

`HandleRequest()` prüft genau, ob eine Anfrage zu einem angegebenen Dienst passt. Für den Bearbeiter bedeutet dass, dass die in `Cy4Message` übergebene Anfrage gültig ist und Dienstort, Größe und Zugriffsart zum Dienst in `Cy4Service` passen. Auf eine zusätzliche Gültigkeitsprüfung innerhalb des Dienstbearbeiters kann daher verzichtet werden.

Die Aufgabe des Bearbeiters besteht nun darin, sich um die eingegangene Anfrage zu kümmern. Bei Leseanfragen muss er das übergebene `Cy4Message`-Objekt mit den gewünschten Daten ausstatten, bei Schreibanforderungen muss er die Daten aus der `Cy4Message` lesen und entsprechend verarbeiten. Dabei dient die in `Cy4Message` übergeben Datenstruktur als Ein- wie auch als Ausgabemedium. Bei Aufruf der Funktion bringt `Cy4Message` die Anfrage mit. Nach ihrer Bearbeitung erwartet die Übertragungseinheit die Antwort wieder in `Cy4Message` zurück. Das bedeutet, eine in `Cy4Message` übergebene Nachricht ist nicht auf die Rolle Anfrage oder Antwort festgelegt. Erst der Ort ihres Auftretens weist der Nachricht ihre Aufgabe zu. In diesem Fall verwandelt die Bearbeitung eine Anfrage in eine Antwort.

Es ist natürlich keineswegs sichergestellt, dass die Bearbeitung des Dienstes immer erfolgreich ist. Zwar sichert das Cy4NET-System zu, dem Bearbeiters bei Aufruf eine gültige und zum Dienst passender Anfrage zu übergeben, ob jedoch die übermittelten Daten die Bearbeitung zulassen, kann erst bei Ausführung der entsprechenden Funktion entschieden werden. Ein Beispiel dafür findet sich in dem aus Kapitel 6.3 bekannten Dienst *FactorySettings* (`FFFF40h:1:WO`). Er bewirkt die Rücksetzung aller Einstellungen auf ihren Ursprungszustand. Der zuständige Bearbeiter in `Application.c` ist folgendermaßen implementiert:

```
static void
OnFactorySettings(const Cy4Service* service, Cy4Message* message)
{
    (void)service;
    if (CY4MESSAGE_ConstData(message)[0] == 0x5E)
    {   STORING_RestoreDefaults();
    }
    else
    {   CY4MESSAGE_Invalidate(message);
```

```
    }
}
```

Umgesetzt ist der Dienst als Auslöser gemäß Konvention 4. Dazu wird geprüft, ob die Anfrage den entsprechenden Wert mitführt. Weicht der Wert von der Vorgabe ab, wird statt der Rücksetzung der Einstellungen `message` für ungültig erklärt. Eine ungültige Antwort-Nachricht symbolisiert dem Cy4NET-System, dass die Ausführung des Dienst fehlgeschlagen ist und damit unbeantwortet bleibt. Wie letztlich übergebenen Werte behandelt werden und ob sie zur Verweigerung der Dienstausübung führen, liegt im Ermessen des Behandlers.

Es gibt einen weiteren Punkt, der bei der Implementierung von Dienst-Anfragebearbeitern berücksichtigt werden muss. Es handelt sich dabei um die Kapitel 6.2 beschriebene Möglichkeit, Dienstspeicher partiell nutzen zu können. Bei einer Dienstbreite von eins, also nur einem Byte, ist das trivial, doch sobald der Dienstspeicher mehrere Bytes umfasst, muss immer die Möglichkeit bestehen, auf Teile des Dienstspeichers, bis hin zu einem einzelnen Byte, zuzugreifen. Wie so etwas aussehen kann, zeigt der Dienst *DeviceLabel*. Der in Kapitel 6.3 beschriebene Dienst zum Setzen einer benutzerdefinierten Gerätebezeichnung umfasst bis zu 61 Bytes. Auf ihn kann als Ganzes, auf Teilzeichenketten oder auch nur Zeichenweise zugegriffen werden. Grundsätzlich wird die Verarbeitung von Diensten, durch die Anforderung partiell genutzt werden zu können, umständlicher. Eine Möglichkeit besteht darin, den Dienst als Ganzes beizubehalten und nur diejenigen Datenbytes ein- bzw. ausladen, die in der Nachricht adressiert sind. So ist es auch im Bearbeiter für die Gerätebezeichnung, `OnDeviceLabel()` umgesetzt:

```
static void
OnDeviceLabel(const Cy4Service* service, Cy4Message* message)
{
    UInt08  label[61];

    STORING_Load(STO_DeviceLabel, label);
    CY4SERVICE_TransferPayload(service, message, label);
    label[sizeof(label)-1] = '\0';     // Das letztes Byte muss immer 0 sein
    STORING_Save(label, STO_DeviceLabel);
}
```

Zu Beginn wird der komplette Datenbestand des Dienstes, also die komplette Gerätebezeichnung, lokal nach `label` kopiert. Im zweiten Schritt werden nun die in der Nachricht angesprochenen Datenbytes verändert, d.h. bei einer Schreibnachricht wandern Zeichen aus der Nachricht nach `label`, bei einer Lesenachricht wandern umgekehrt die gewünschten Zeichen von `label` in die Nachricht. Abschließend wird `label` als Ganzes, mitsamt allen eventuell stattgefundenen Modifikationen wieder abgelegt. Unterstützt wird diese Verfahrensweise durch die Hilfsfunktion `CY4SERVICE_TransferPayload()` aus der Übersetzungseinheit `Cy4Service`. Ebenso wie das Makro `CY4TYPEPLATE` handelt es sich um eine reine Komfortfunktion:

```
void CY4SERVICE_TransferPayload(const Cy4Service* service,
                                Cy4Message*       message,
                                UInt08*           data)
```

Beschreibung: Je nach Zugriffsart von `message` bewegt diese Funktion Nutzlast aus der Nachricht ins Datenfeld des Dienstes oder umgekehrt. D.h. bei `CY4MESSAGE_READ` werden Datenbytes aus `data` gelesen und in `message` abgelegt, bei `CY4MESSAGE_WRITE` wird die Nutzlast aus `message` nach `data` geschrieben. An welcher Stelle der Zugriff stattfindet, bestimmt der Dienstort in `message`.
Die aufrufende Umgebung muss sicherstellen, dass `data` auf ein zum Dienst passend dimensionieres Bytefeld zeigt.

Eingabe: `service`: Der zugrunde liegende Dienst.
`message`: Die betroffene Nachricht.
`data`: Datenfeld des Dienstes.

Ausgabe: –

Habitat von `CY4SERVICE_TransferPayload()` sind die Dienst-Anfragebearbeiter. Man beachte, dass Dienste bzw. die Dienstspeicherbereiche weitaus größer sein können als die 64 Byte Nutzlastkapazitätsgrenze der Cy4-PDU bzw. der `Cy4Message`. In einem solchen Fall ist es besser, die Speicherbereiche eines Dienstes stückweise bereitzustellen.

Partielle Zugriffe können zu Inkonsistenzen führen. Beispiel: Man ändert durch einen gezielten Zugriff die Tagesnummer eines Datums. Verfügt nun der aktuell eingestellte Monat des Datums nicht über die gesetzte Anzahl Tage, hinterlässt dieser Zugriff ein ungültiges Datum. Grundsätzlich können solche Inkonsistenzen nicht seitens des Protokolls abgefangen werden. Wie in solchen Fällen verfahren wird, muss im jeweilige Behandler oder der Anwendung entschieden werden.

10.3.4.2 Anfragen und Antworten

Anfragen erreichen das Gerät zu beliebigen Zeitpunkten. Anders sieht es aus, wenn die Busstation selbst tätig wird und den Dienst einer anderen Station in Anspruch nehmen möchte. Der Kern bietet zu diesem Zweck die Funktionen `CY4CORE_PostRequest()` und `CY4CORE_TakeResponse()` an. Die Erste setzt die Anfrage ab, die Zweite nimmt die Antwort entgegen. So genügsam die beiden Funktionen ihren Aufgaben nachkommen, könnte man geneigt sein, die ausgehende Kommunikation durch Hintereinanderausführung beider Funktion stattfinden zu lassen. Doch das ist keine gute Idee, denn zwischen Initiierung einer Anfrage und Entgegennahme ihrer Antwort vergeht üblicherweise einige Zeit, bis hin zum Ablauf der Round-Tip-Time. Damit eignet sich das Funktionsduo nicht dazu, innerhalb des gleichen Kontrollflusses eingesetzt zu werden. Zumindest nicht, wenn man nicht gewillt ist, das System mittels einer Abfrageschleife (*Spin-Loop*) warten zu lassen, wovon generell abzuraten ist, wenn die Wartezeit signifikant werden kann.

Da die Übertragungseinheit immer nur in der Lage ist eine einzelne Übertragungssequenz abzuarbeiten, muss bei Funktionen wie `CY4CORE_PostRequest()`, die direkt auf den Kern aufsetzen, stets geprüft werden, ob die Übertragungseinheit aktuell überhaupt dazu bereit ist, eine Nachricht

zu versenden. Gegebenenfalls muss der Sendeversuch zu einem späteren Zeitpunkt wiederholt werden.

Moderne Kommunikationssysteme begegnen diesem Problem, indem sie Sendeaufträge nicht unmittelbar abarbeiten, sondern die Aufträge zuerst in einer Abarbeitungsschlange aufreihen. Sobald die Übertragungseinheit für einen weiteren Sendeauftrag bereit ist, holt sie sich den nächsten wartenden Auftrag aus der Abarbeitungsschlange und schickt ihn auf die Reise. Sendeauftrag und Sendevorgang sind auf diese Weise zeitlich voneinander entkoppelt. Der Vorteil dieser Methode ist, dass sich die aufrufende Umgebung nicht darum kümmern muss, ob die Übertragungseinheit aktuell überhaupt zur Entgegennahme des Auftragen bereit ist. Ist der Auftrag erteilt, kümmert sich das Übertragungssystem darum, die Anfrage bei nächster Gelegenheit abzuschicken. Der Nachteil an der Methode ist: Die Echtzeitfähigkeit leidet. Wird ein Sendeauftrag erteilt, erscheint er als neuer letzter Auftrag am hinteren Ende der Schlange. Warten bereits andere Aufträge auf Aussendung, verlängert sich die Antwortzeit der aktuell eingestellten Anfrage um jeweils die der wartenden. Man muss sich also darüber im Klaren sein, dass wenn bei Auftragserteilung beispielsweise zwei Aufträge warten, sich die Antwortzeit des neuen Auftrages um 2·500 ms auf dann insgesamt 1500 ms erhöht. Zumindest theoretisch, denn in der Praxis sind die Antwortzeiten in der Regel wesentlich kürzer. Trotzdem sollte man bei zeitkritischen Anfragen die momentane Länge der Abarbeitungsschlange mit in Betracht ziehen.

Auch das Cy4NET-System bietet die Möglichkeit, Kommunikation auf diese Weise stattfinden zu lassen. Doch dabei gibt es ein Problem: Controllersysteme, wie die der AVR-Klasse, verfügen nicht über ausreichende RAM-Kapazitäten um eine Mehrzahl von Sendeaufträgen in Form von Anfrage-Nachrichten im Speicher zu parken. Jede Anfrage würde als `Cy4Message`-Objekt mindestens 73 Bytes des RAM-Speichers in Anspruch nehmen. Hat die Abarbeitungsschlange selbst eine Kapazität von, angenommen 10 Nachrichten, würde das mit einer statischen Belegung von 10·73 = 730 Bytes zu Buche schlagen. Zuviel, für einen Controller, dessen weitere Aufgaben ebenfalls Speicher benötigen.

Wenn man jedoch berücksichtigt, dass die physische Existenz einer Anfrage-Nachricht erst zum Zeitpunkt ihrer Verschickung wichtig wird, eröffnet sich eine elegante Alternative: Mit der Anfrage wird nicht etwa die Nachricht selbst abgelegt, sondern nur eine Funktionalität, die die Herstellung einer Anfrage ermöglicht. Erst unmittelbar bevor die betroffene Anfrage an der Reihe ist gesendet zu werden, wird mit Hilfe dieser Funktionalität die konkrete Anfrage erzeugt. Diese Methode der „*Anfrage auf Abruf*“ folgt dem Funktionsprinzip der *Lazy Evaluation* und ermöglicht damit die Realisierung einer Abarbeitungsschlange mit minimalem Speicheraufwand.

Doch was ist mit der Antwort? Anfrage und Antwort liegen zeitlich auseinander. Die Abarbeitung innerhalb des gleichen Kontrollflusses ist, zumindest ohne die oben erwähnten Einschränkungen, nicht möglich.

Eine sinnvolle Lösung besteht darin, zum Zeitpunkt der Erteilung einer Anfrage bereits festzulegen was geschehen soll, wenn die Übertragungseinheit die Antwort auf die Anfrage erhält. Die Erteilung der Anfrage und die Bearbeitung ihrer Antwort finden damit unabhängig und an unterschiedlichen Stellen statt.

Das Mittel zum Zweck für diese beiden Aktionen sind wieder Rückrufe. Ins Spiel kommen nun die beiden übrigen Funktionszeiger aus `Cy4Net.h`, gemeint sind der *Nachrichten-Ausstatter* `Cy4MessageSetup` und der *Dienst Antwortbehandler* `Cy4ResponseHandler`. Die Namen lassen bereits auf ihre Funktionalitäten schließen. Eine Anfrage zu senden besteht nun darin, zwei Funktionen in Form von zwei Funktionszeigern an die Übertragungseinheit zu übergeben. Damit ist der Sendevorgang auf den Weg gebracht, um alles Weitere kümmert sich die Übertragungseinheit, sprich das Cy4NET-System. Die API-Funktion, die diese Aktion ermöglicht, heißt:

```
bool CY4NET_Post(Cy4MessageSetup    messageSetup
                 Cy4ResponseHandler responseHandler)
```

Beschreibung: Die Funktion nimmt eine Anfrage entgegen und stellt sie als neues letztes Element in die Schlange auszugehender Sendungen. Eine Anfrage ist bestimmt durch die Funktionszeiger `messageSetup` und `responseHandler`. Ist die Anfrage an der Reihe, wird vor ihrer Verschickung die durch `messageSetup` referenzierte Funktion aufgerufen. Ihr wird als Parameter ein `Cy4Message`-Objekt übergeben, das mit der gewünschten Anfrage ausstattet werden muss. Lässt die Ausstattungsfunktion das mitgegebene `Cy4Message`-Objekt unangetastet bzw. ist die erzeugte `Cy4Message` ungültig, wird die Anfrage ignoriert und aus der Schlange entfernt.
Nach Erhalt der Antwort wird der in `responseHandler` zu Verfügung gestellte Antwortbehandler aufgerufen, dem als Parameter die erhaltene Antwort-Nachricht mitgegeben wird. Ist diese Nachricht gültig, war die Kommunikation erfolgreich. Andernfalls war die Anfrage fehlerhaft oder die Übertragung ist gescheitert. Die übergebene `Cy4Message` ist in dem Fall eine als ungültig gekennzeichnete Kopie der Anfrage.
Im Gegensatz zur Ausstattungsfunktion in `messageSetup` ist die Angabe eines Antwortbehandlers optional, d.h. `responseHandler` darf *`NULL`* sein.
Hinweis: Rundrufe werden im Protokoll nie beantwortet. Trotz dessen wird aber auch für sie der Antwort-Behandler aufgerufen. Als Antwort-Nachricht erhält der Behandler in dem Fall eine Kopie der Anfrage, deren Gültigkeit darüber informiert, ob das System den Rundruf hat absetzen können oder nicht. Das ist insofern kein Problem, da Rundrufe stets Schreibaufforderungen sind, deren Antworten keinerlei Nutzlast befördern. Ob der Rundruf letztlich alle Zielknoten erreicht hat, bleibt natürlich ungewiss.
Die Aufnahmekapazität der Schlange ist durch `CFG_CY4_POSTING_QUEUE_CAPACITY` festgelegt. Man beachte, dass sich bei auf Aufreihung von Anfragen deren Antwortzeiten erhöhen können.

Eingabe: `messageSetup`: Cy4-Nachrichten-Ausstattungsfunktion.
`responseHandler`: Cy4-Dienst Antwortbehandler oder *`NULL`*.

Ausgabe: *true*, wenn die Anfrage zur Absendung entgegengenommen wurde, *false*, wenn die Schlange voll ist und daher die Annahme der Anfrage verweigert wurde.

Selbstverständlich ist auch die Abarbeitungsschlange endlich. Ihre Länge ist nicht durch das Protokoll bestimmt, sondern muss je nach Bedarf und Kapazität projektspezifisch festgelegt werden. In

der Referenzimplementierung ist ihre Länge durch `CFG_CY4_POSTING_QUEUE_CAPACITY` bestimmt. Definiert ist der Wert in `Conf.h`. Generell sollte die Länge so dimensioniert sein, dass ein Volllaufen im normalen Betrieb nicht stattfindet. Der Rückgabewert von `CY4NET_Post()` informiert darüber, ob es dennoch passiert ist. Die aktuelle Länge der Abarbeitungsschlange kann mittels der folgenden API-Funktion erfragt werden:

```
UInt08 CY4NET_QueueLength(void)
```

Beschreibung: Die Funktion liefert die Anzahl auf Aussendung wartender Sendungen der Abarbeitungsschlange. Solange die Funktion einen Wert kleiner `CFG_CY4_POSTING_QUEUE_CAPACITY` zurückliefert, ist die Schlange nicht voll, d.h. `CY4NET_Post()` wird die Annahme weiterer Anfragen akzeptieren.

Eingabe: –
Ausgabe: Anzahl wartender Sendungen: `0` … `CFG_CY4_POSTING_QUEUE_CAPACITY`.

Man beachte, dass durch den Aufruf von `CY4NET_Post()` weder Werte noch der Dienst konkretisiert werden. Es werden lediglich Funktionalitäten, also Metadaten, angegeben. Ob die Anfrage eine Schreibaufforderung oder eine Leseanfrage darstellt, entscheidet sich erst zu dem Zeitpunkt ihrer Erstellung, d.h. im Nachrichten-Ausstatter.

Beispiel: Das Cy4Nut-Gerät bietet einen Dienst an, der die Info-LED ein- und ausschaltet. Er lautet:

InfoLedState	**100200h:1:RW**

Beschreibung: Liefert oder ändert den Zustand der Info-LED.

Belegung:

0
INF

INF	Zustand der Info-LED: 0 = Aus, 1 = Ein

Die Bedeutung der Tabelleneinträge kann Anhang B entnommen werden.

Um den Dienst zu nutzen wird nun ein entsprechender Anfrage-Ausstatter benötigt. Die dort übergebene Cy4Message wird in unserem Beispiel mit einer Schreibaufforderung an Zieladresse `155.120` bestückt. Der Wert in `infoLedOn` bestimmt den Zustand der Info-LED im Zielgerät, in diesem Fall *true* für ein oder *false* für aus. `SetupInfoLedStateMessage()` ist als lokale Funktion im Application-Modul angesiedelt:

```
static void
```

```
SetupInfoLedStateMessage(Cy4Message* message)
{
    UInt08 data[1] = { infoLedOn };
    Cy4Address address = CY4ADDRESS(155,120);
    CY4MESSAGE_SetupWrite(message, address, 0x100200, data, sizeof(data));
}
```

Ob die Anwendung an einer Antwort auf die Anfrage interessiert ist, muss fallbezogen entscheiden werden. Bei einer Schreibaufforderung informiert die Antwort lediglich darüber, ob die Kommunikation erfolgreich war oder nicht. Wenn diese Information für den Aufrufer irrelevant ist, weil er sie nicht weiter verwertet, kann er auf einen entsprechenden Antwortbehandler verzichten. Das geschieht, indem er in `responseHandler` der Wert *`NULL`* übergibt.

Anders sieht es aus bei Leseanforderungen. Zwar besteht auch hier die Möglichkeit, auf den Antwortbehandler und damit auf die Antwort zu verzichten, jedoch ergäbe es wenig Sinn. Schließlich führt die Leseantwort die angeforderten Daten mit sich. Ohne Behandler würden diese verloren gehen, ebenso wie die Information über Erfolg oder Misserfolg.

In unserem Beispiel wird auf einen Antwortbehandler nicht verzichtet. Die Schreibbestätigung wird genutzt, um bei einer nachfolgendem Schreibaufforderung den Zustand der Info-LED umzukehren. Die Antwort der Anfrage wird durch folgenden Behandler entgegengenommen:

```
static void
OnResponseOfInfoLedState(const Cy4Message* message)
{
    if (CY4MESSAGE_IsValid(message))
    {   // Übertragung erfolgreich. Beim nächsten Aufruf, LED-Zustand invertieren
        infoLedOn = !infoLedOn;
        // …
    }
    else
    {   // Übertragung fehlgeschlagen. Anfrage gegebenenfalls wiederholen
        //
        // attempt++;
        // if (attempt < MAX_ATTEMPTS)
        // {   CY4NET_Post(SetupInfoLedStateMessage, OnResponseOfInfoLedState);
        // }
    }
}
```

Aufgrund der zeitlichen Divergenz der Aktionen ist es übrigens kein Problem, bei fehlgeschlagenen Kommunikationssequenzen bereits innerhalb des Antwortbehandlers einen erneuten Übertragungsversuch in die Wege zu leiten. Damit dabei der Datenverkehr auf dem Bus aufgrund einer möglichen Endlosschleife nicht zu arg strapaziert wird, sollte die Anzahl der Fehlversuche dabei aber begrenzt werden.

Um die Dienstanfrage auszulösen, genügt letztendlich an geeigneter Stelle der einfache Aufruf:

```
// …
CY4NET_Post(SetupInfoLedStateMessage, OnResponseOfInfoLedState);
// …
```

Die oben gezeigten Schreibaufforderung war an Zieladresse `155.120` gerichtet, üblicherweise ein fremdes Gerät. Jedoch ist das nicht zwingend vorgeschrieben. Ebenso ist es möglich als Ziel das eigenen Gerät zu adressieren. Die Verarbeitung findet wie jede andere Übertragung auch statt. Die Anfrage wird auf den Bus ausgegeben und von der eigenen Empfangseinheit gleich wieder aufgenommen. Nach der lokalen Bearbeitung schickt das Gerät die Antwort ordnungsgemäß wieder zurück, wo sie vom eigenen Antwort-Behandler entgegengenommen wird. Die Übertragungseinheit unterscheidet nicht zwischen *Loopback*-Nachrichten und externer Kommunikation. Im Grunde bekommt sie sogar gar nicht mit, dass ein lokaler Dienst in Anspruch genommen wurde.

Der Einsatz von *Loopback*-Nachrichten kann von Vorteil sein, wenn sich durch Aufruf eines lokalen Dienstes der Aufbau einer speziellen Schnittstelle für eine exponierte Funktion erübrigt.

Oben wurde verlangt, die Länge der Abarbeitungsschlange so zu dimensionieren, dass ein Volllaufen im normalen Betrieb nicht stattfindet. Mitunter kann die Dimensionierung der Schlange aber Probleme bereiten. Insbesondere, wenn die Anzahl an Anfragen innerhalb kurzer Zeit stark ansteigt. Die Kapazität der Abarbeitungsschlange muss konzeptionell so ausgelegt sein, dass sie auch hohes Aufkommen verkraftet. Die Schlange stellt sicher, dass jede aufgereihte Anfrage abgearbeitet wird. Jedoch das ist nicht unbedingt immer von Interesse. Eine wichtige Rolle spielt dabei die Art der zu übertragenden Informationen.

Dazu ein Beispiel: Eine Busstation ist mit einem Sensor ausgestattet, dessen aktueller Wert stets einem anderen Gerät mitgeteilt wird. Die Werte des Sensors werden vom ersten Gerät ständig überwacht. Sobald eine Änderung eintritt, wird eine entsprechende Schreibaufforderung an das zweite Gerät abgesetzt. Abhängig von der Änderungsrate des Sensorwertes kann dabei das Aufkommen an Nachrichten, mit Einschränkungen, beliebig hoch werden. Eine Möglichkeit dem entgegenzuwirken besteht natürlich darin, das Abfrageintervall des Sensors und damit die Senderate der Schreibaufforderungen zu verlangsamen. Aber es gibt noch einen anderen Aspekt: Die Frage ist: Sind Wertänderungen, die bereits veraltet sind, für die Gegenstelle überhaupt noch wichtig? Möglicherweise kann auf all diejenigen Anfragen verzichtet werden, die durch eine ohnehin veraltete Wertänderung initiiert wurden.

Der Mechanismus der *Anfrage auf Anforderung* bedingt, dass erst unmittelbar vor Absendung die konkrete Anfrage erzeugt wird. In unserem Beispiel bedeutet es, dass erst zu diesem Zeitpunkt der aktuelle Sensorwert erfasst und damit die Anfrage bestückt wird. Dem Empfänger wird also stets der neueste und aktuellste Sensorwert übermittelt. Da Auslöser und Sensorwert zum Zeitpunkt der Anfrage nicht korrelieren, ergibt sich daraus auch keine Notwendigkeit, mehrere Anfragen gleichen Typs in der Abarbeitungsschlange warten zu lassen. Welchen Sinn ergäbe es, eine weitere Schreibnachricht auf die Reise zu schicken, wenn eine bereits Wartende ohnehin schon den aktuellsten Wert übermitteln wird?

Was man in diesem Zusammenhang benötigt ist eine Möglichkeit Duplikate zu vermeiden. Das geht, indem geprüft wird, ob eine bestimmten Anfrage bereits in der Schlange vorkommt. Die folgenden API-Funktion übernimmt genau diese Prüfung:

```
bool CY4NET_IsPending(Cy4MessageSetup    messageSetup
                      Cy4ResponseHandler responseHandler)
```

Beschreibung: Die Funktion prüft, ob eine durch `messageSetup` und `responseHandler` bestimmte Anfrage bereits in der Schlange auf Absendung wartet. Die Information darüber, ob eine solche Anfrage bereits auf Versendung wartet ermöglicht es, redundante Anfragen zu vermeiden, Buskommunikation zu verringern und einem Volllaufen der Sendeschlange entgegenzuwirken.

Eingabe: `messageSetup`: Cy4-Nachrichten-Ausstattungsfunktion.
`responseHandler`: Cy4-Dienst Antwortbehandler oder *`NULL`*.

Ausgabe: *true*, wenn bereits eine entsprechende Anfrage in der Sendeschlange wartet.

Vor Absendung einer Schreibaufforderung wäre im Beispiel oben also zu prüfen, ob bereits eine entsprechende Nachricht auf Absendung wartet. Ist das der Fall, kann auf das Einstellen einer weiteren Nachricht gleichen Typs verzichtet werden. Der folgende Beispielcode setzt genau das um:

```
// …
static UInt08 oldValue = 0xFF;
UInt08 currentValue = SensorValue();    // 0 .. 100 [#], 80h im Fehlerfall
if (oldValue != currentValue)
{   oldValue  = currentValue;
    if (!CY4NET_IsPending(SetupSensorMessage, NULL))
    {   CY4NET_Post(SetupSensorMessage, NULL);
    }
}
// …
```

Diese Vorgehensweise erleichtert die Kapazitätsabschätzung für die Abarbeitungsschlange erheblich. Da unabhängig von der Änderungsrate des Sensorwertes, stets höchstens eine Nachricht in der Schlange wartet, muss die Kapazität der Schlange in diesem Fall auch nur um eins erhöht werden. Ein Überlaufen ist ausgeschlossen.

Wann und in welchem Umfang von dieser Reduktion der Anfragemenge Gebrauch gemacht werden kann, hängt stark von der Art der zu übermittelnden Informationen ab. Es gibt Fälle in denen es dringend notwendig ist, mehrere Anfragen gleichen Typs in der Abarbeitungsschlange beizubehalten.

11 Zugang zum Netz

Wie alle Feldbussysteme ist auch das Cy4NET vorerst einmal ein in sich geschlossenes System. Jegliche Kommunikation zwischen den Knoten findet im inneren Kreis ohne externe Einflussnahme statt. Für das Gefüge an Busstationen stellt das im Grunde kein Problem dar. Sollen jedoch Busstationen konfiguriert, Geräteadressen geändert, Parameter geschrieben oder Daten ausgetauscht werden, steht man vor dem Problem von außen keinen Kontakt zu den Busstationen aufbauen zu können. Man ist ausgeschlossen. Daher besteht eine der ersten Aufgaben beim Aufbau eines Bussystemes darin, einen externen Zugang zu schaffen. Also ein Gateway.

Wie das Gateway die Kommunikation nach extern gestaltet, ist dabei nicht Teil des Protokolls. Folglich ist auch die hier vorgestellte Realisierung eines Gateways als Fallbeispiel zu verstehen und nicht als Teil der Cy4NET-Spezifikation. Da jedoch das Vorhandensein eines externen Zuganges von fundamentaler Bedeutung ist, rechtfertigt das die Platzierung des Themas in diesem Dokument.

11.1 Das Gateway

Das Gateway ist im Grunde ein Januskopf, ein Gerät mit zwei Gesichtern. Auf Seiten des Cy4NET ist das Gateway ein Gerät wie jedes andere. Es stellt spezifische Dienste zur Verfügung und verfügt über alle Pflichtdienste. Als solches unterscheidet es sich in keiner Weise von normalen Busstationen und in der Tat ist von sich aus gar nicht erkennbar, ob eine Busstation auch die Funktionalität eines Gateways anbietet oder nicht. Abbildung 17 zeigt das hier vorgestellte Gateway. Es trägt die Boardtypennummer B00023 und wird in Anhang E vorgestellt. Die zugehörige Gateway-Applikation ist A00111 [16].

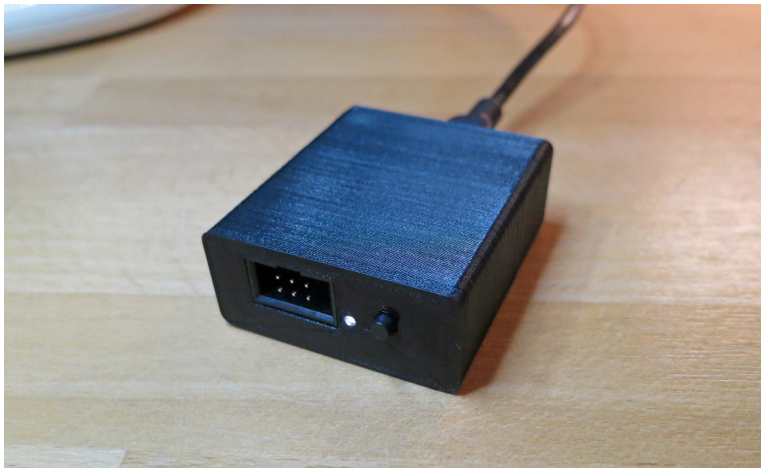


Abbildung 17: B00023 Cy4-To-Serial Gateway im Gehäuse

Auch wenn das hier vorgestellte Gerät in erster Linie die Funktion eines Gateways bereitstellt, ist es nicht notwendig, eine Busstation exklusiv für diese Aufgabe zu reservieren. So realisiert Appli-

kation A00111 auf Board B00023 neben dem Gateway zusätzlich einen Busmonitor, mittels dessen sich die Kommunikation auf dem Bus verfolgen lässt, in Kapitel 11.4 wird darauf eingegangen. So nehmen Gateways neben ihrer Tätigkeit als Pförtner, oft auch weitere Aufgaben wahr. Häufig betreiben aktuell bereits im Einsatz befindliche Gateways diese Aufgabe sogar eher als Nebentätigkeit. Als Beispiel sei B00024 erwähnt, die sogenannte *Powerstation* [17]. Sie dient als Versorgungs- und Kontrollstation eines Cy4NET-Systems. Zusätzlich zu diesen Aufgaben bietet die Powerstation einen Cy4NET-Zugang mittels Ethernet-LAN und ermöglicht damit den Zugriff via TCP/IP.

Je nach Aufgabe und Zielsetzung gibt es diverse Möglichkeiten, mit einem Gateway in Interaktion zu treten. Das hier vorgestellte Gateway kommuniziert extern mittels einer seriellen Verbindung. Ob dabei die klassische RS232, ein USB-zu-seriell Umsetzer oder, wie bei der Powerstation oben, eine Serial-zu-TCP-Schnittstelle zum Einsatz kommt, ist für die grundsätzliche Kommunikation unerheblich. Auf welcher Basis die serielle Verbindung zustande kommt, muss je nach Einsatzumfeld entschieden werden.

Bei den Einheiten, die von außen mit dem Gateway in Kontakt treten, handelt es sich üblicherweise um performante Rechner, beispielsweise einem PCs. Da gängige Desktop-Betriebssysteme jedoch nicht darauf ausgelegt sind, exakte Timingvorgaben einzuhalten, reduziert sich die Kommunikation mit dem Gateway auf einen reinen *Master-Slave*-Betrieb. D.h. der eine stellt die Anfragen, der andere liefert die Antworten. Nach außen hin wird auf die Multimaster-Kommunikation aufgrund der häufig fehlenden Echtzeitfähigkeit der externen Systeme verzichtet. Das ist insofern keine große Einschränkung, als dass externe Geräte im Verbund der Cy4NET-Stationen ohnehin eine Sonderstellung einnehmen. Durch die Wahrnehmung der oben genannten Aufgaben findet ihre Beteiligung an der Cy4NET-Kommunikation eher punktuell statt. Der Verzicht auf den Multimasterbetrieb bedeutet im Wesentlichen, dass Busstationen selbständig keine Anfragen nach außen schicken können, sondern darauf angewiesen sind, gefragt zu werden.

Die externe Kommunikation mit A00111 basiert auf dem Austausch von ASCII-Werten. Genauer gesagt: Es werden Textzeilen versendet. Jede Anfrage von außen wird in einer Textzeile formuliert an das Gateway geschickt. Das Gateway wandelt die Textzeile in eine Cy4NET-komforme Anfrage um und leitet sie, in seinem Namen, weiter an das adressierte Zielgerät. Dabei kann das Zielgerät auch das Gateway selbst sein. Nach Entgegennahme der Antwort baut das Gateway daraus wieder eine entsprechende Antwort-Textzeile zusammen und reicht sie an die externe Einheit zurück. Das Gateway übernimmt dabei also die protokollkonforme Umsetzung der Kommunikation. Es vertritt auf diese Weise das externe Gerät gegenüber dem Cy4NET. Für die Cy4NET-Stationen stellt sich die Anfrage von außen als ganz normale Anfrage durch ein anderes Cy4NET-Gerät dar. Für das externe Gerät wird die Anfrage via Gateway direkt an das adressierte Zielgerät geleitet und es erhält dessen Antwort auf gleicher Weise wieder zurück.

Auf beiden Seiten ist die Kommunikation also völlig transparent. Das geht sogar so weit, dass sich dem externen Gerät ohne Weiteres keine Möglichkeit bietet, die Adresse seines Gateways festzustellen. Es kann mit allen erreichbaren Busstationen kommunizieren. Eines davon ist das Gateway. Doch welches es ist, lässt sich ohne zusätzliche Mittel nicht feststellen. Wie es trotzdem geht, wird im folgenden Kapitel gezeigt.

11.2 Das Gateway Datagramm

Die Gateway-PDU ist abgeleitet von der Cy4NET-PDU aus Kapitel 7. Auf einige Felder wurde verzichtet, andere wurden erweitert. Abbildung 18 zeigt ihren Aufbau.

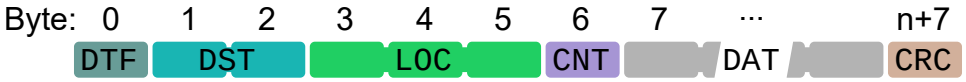


Abbildung 18: Die Gateway-PDU

Zu den Feldern, auf die verzichtet wurde, gehört die DPS und, da die externe Einheit nicht adressierbar ist, die Quelladresse SRC. Dagegen wurde das DTF um ein zusätzliches Bit erweitert. Abbildung 19 zeigt dessen modifizierten Aufbau mit dem zusätzlichen Fehlerbit F.

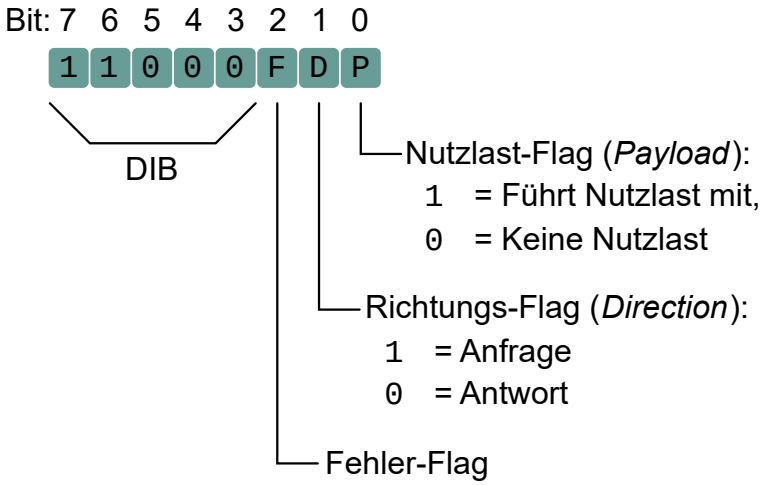


Abbildung 19: Datagramm Typflags (DTF) der Gateway-PDU

Der Cy4NET-spezifische Teil der Kommunikation wird durch das Gateway vertreten. Das betrifft insbesondere die Timings. In der Cy4NET-Kommukation kennzeichnet das Ausbleiben einer Antwort den Misserfolg einer Anfrage. Um die externe Einheit von der Einhaltung dieses Zeitfensters zu entlasten, informiert das Gateway die Außenstelle nicht nur bei Erhalt einer Antwort, sondern ebenso bei Misserfolg einer Anfrage. Auf diese Weise ist die externe Kommunikation mit dem Cy4NET-System zeitlich entkoppelt. Das externe Gerät kann sich darauf verlassen, stets eine Antwort auf eine Anfrage zu erhalten, gegebenenfalls eine abschlägig bescheidene. Um diese zusätzlichen PDU-Typen darstellen zu können, wurde das DTF, wie in Abbildung 20 zu sehen, um zwei Fehlerausprägungen erweitert.

DIB	F D P	DTF	Typ
11000	0 1 0	0xC2	Leseanforderung
11000	0 0 1	0xC1	Leseantwort
11000	1 0 1	0xC5	Lesefehler
11000	0 1 1	0xC3	Schreibaufforderung
11000	0 0 0	0xC0	Schreibbestätigung
11000	1 0 0	0xC4	Schreibfehler

Abbildung 20: Die sechs Typen der Gateway-PDU

11.3 ASCII-PDU

Jedes Byte wird durch eine zweistellige Hexadezimalzahl dargestellt. Zwischen den Zahlen gibt es kein Trennzeichen. Das Ende der Textzeile kennzeichnet ein Zeilenwechsel-Zeichen. Das kann CR, LF oder eine Kombination aus beiden Zeichen sein. Das folgende Beispiel zeigt, welche Textzeilen bei erfolgreichem Abruf eines Typenschildes von `255.3` ausgetauscht werden:

Leseanforderung: `"C2FF03FFFFF00717"`

Leseantwort: `"C1FF03FFFFF007A0012008B00023EA"`

Ein weiteres Beispiel ist der Schreibaufruf *EnterApplication*. Existiert ein Gerät an Adresse `1.6`, wird die Antwort wie folgt ausfallen:

Schreibaufforderung: `"C30106FFFFF8015EC6"`

Schreibbestätigung: `"C00106FFFFF80175"`

Schlägt hingegen eine Anfrage fehl, reagiert das Gateway ebenfalls mit einer Antwort. In einem solchen Fall reagiert das Gateway mit einer Kopie der Anfrage, allerdings je nach Zugriffsart als Typ `C4` oder `C5`. Im folgenden Beispiel wird das Typenschildes eines nichtexistierenden Gerätes an Adresse `20.3`: abgerufen:

Leseanforderung: `"C21403FFFFF007C6"`

Leseantwort: `"C51403FFFFF00775"`

Die Anfrage ist fehlgeschlagen. Das Gateway schickt die Anfrage als `C5`-Typ zurück: Lesefehler.

Das Gateway sorgt dafür, dass das externe Gerät mit einer entsprechenden Antwort versorgt wird. Um abzuschätzen, wann in etwa mit der Antwort zu rechnen ist, kann die Round-Trip-Time herangezogen werden. Hat das Gateway auf seine Anfrage nach 500 ms keine Antwort erhalten, schickt es eine entsprechende Fehlernachricht an das externe Gerät zurück. Die Zeit verlängert sich um die Umlaufdauer der PDUs zwischen externem Gerät und Gateway.

Damit die Kommunikation auf dem Cy4NET die externe Kommunikation nicht überholt, ist auf eine ausreichend hohe Symbolrate mit Extern zu achten. Unter Berücksichtigung, dass das Gateway seine PDU in Form einer ASCII-Zeichenkette verschickt, bei der ein Byte durch genau zwei Zeichen dargestellt wird, ergibt sich eine Symbolrate, die mindestens dem Doppelten der Cy4NET-Rate entsprechen muss. Bei dem hier vorgestelltem Gateway beträgt die Symbolrate der seriellen Schnittstelle 57600 Bd und damit dem dreifachen der Cy4NET-Rate.

Um die Kommunikation durchzuführen, reicht es im Grunde aus, mit einem Terminal eine serielle Verbindung mit den gerade genannten Parametern zu einem Gateway herzustellen und die Anfrage-Textzeile samt Zeilenwechsel zu senden. Je nach adressiertem Zielgerätes wird die Antwort-Textzeile ähnlich den obigen ausfallen. Dabei ist allerdings Folgendes zu beachten.

> Das Gateway erwartet, dass eine Anfrage-Textzeile innerhalb von 700 ms komplett gesendet ist. Andernfalls wird die Anfrage verworfen und ignoriert. Sollte die Übertragung also länger dauern, weil beispielsweise der Anfragetext per Tastatur eingegeben wird, wird das Gateway darauf nicht reagieren. Auch wenn die Übertragung textueller Natur ist, ist die Kommunikation mit dem Gateway nicht auf manuelle Eingaben ausgelegt. Ohnehin wäre die manuelle Berechnung der jeweiligen Prüfsumme eine mühselige Arbeit.

Für die Gateway-Kommunikaiton kommen spezielle Software-Werkzeuge zum Einsatz, die auf den jeweiligen Clienten laufen. Das Einfachste ist das Kommandozeilen-Werkzeug `cy4cmd`, das in Kapitel 11.5 beschrieben ist.

Kommen wir zurück zu dem Problem, dass sich der externen Einheit keine Möglichkeit bietet, die Adresse seines Gateways zu erfahren. Da alle Geräte auf dem Bus erreichbar sind, ist diese Information in der Praxis auch nicht unbedingt von Interesse. Um sie dennoch zu erfahren, gibt es einen Sonderweg außerhalb des Gateway-Kommunikationsprotokolls. Erhält das Gateway als Anfrage-Textzeile ein einzelnes Asterisk-Zeichen (*, ASCII-Code 2Ah), verhält es sich so, als wäre sein *CallTypePlate*-Dienst aufgerufen worden. In der Antwort-PDU findet sich dann neben dem Typenschild auch die Adresse des Gateways. Und da das Gateway für die Übertragung einer PDU-Textzeile einen Zeitrahmen von 700 ms einräumt, ist ausreichend Zeit, diese Anfrage, wie in Abbildung 21 zu sehen, gegebenenfalls über ein Terminal per Tastatur einzugeben. Die Parameter der Verbindung lauten: 57600 Bd, 8N1.

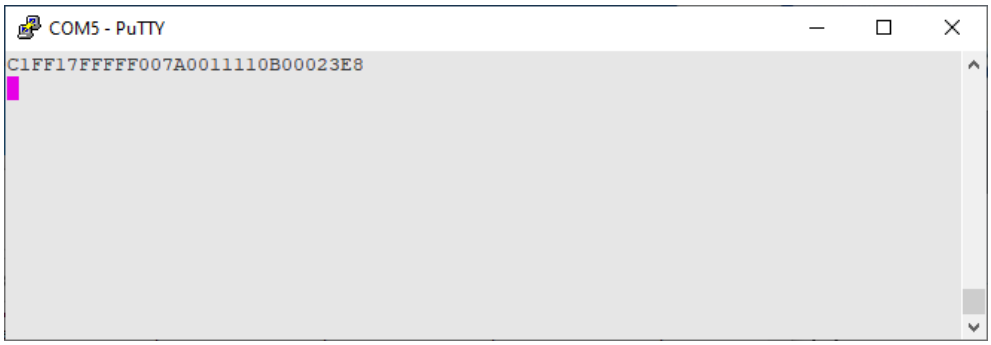


Abbildung 21: Gateway-Antwort auf eine ''-Anfrage im Terminal.*

Mittels der Asterisk-Anfrage lässt sich darüber hinaus auf leichte Weise feststellen, ob an einem geöffneten Kommunikationskanals ein Gateway auf Anfragen wartet. Das ist eleganter, als mittels einer Sondierungsanfrage an eine fiktive Busstation das Gateway zu stimulieren, eine Antwort zu senden.

11.4 Busmonitor

Die Standardanwendung für B00023 ist die Gateway-Applikation A00111. Sie vereinigt im Grunde zwei Geräte in einem: Einmal das Gateway und zusätzlich dazu einen Busmonitor. Der Monitor ermöglicht es, die Kommunikationssequenzen auf dem Bus sichtbar zu machen. Jede Anfrage und jede Antwort wird textuell über die serielle Verbindung ausgegeben. Wie oben dient ein einfaches Terminal zur Ausgabe. Auch fehlerhafte PDUs und Kollisionen können mitverfolgt werden. Abbildung 22 zeigt einen Auszug aus der Ausgabe des Busmonitors auf dem Terminal. Mittels eines Dienstes von A00111 kann festgelegt werden, ob direkt PDU-Bytes ausgegeben werden (Rohwertausgabe) oder ob die Ausgabe, wie in Abbildung 22 zu sehen, in einer aufbereiteten Form stattfindet. Die aufbereitete Form erleichtert die Nachvollziehbarkeit der Anfrage und Antwortsequenzen. So werden Anfragen in dieser Darstellung beispielsweise als Pfeil nach rechts (`--R?-->` bzw. `--W?-->`) und Antworten als Rückpfeil (`<-R!--` bzw. `<-W!--`) dargestellt.

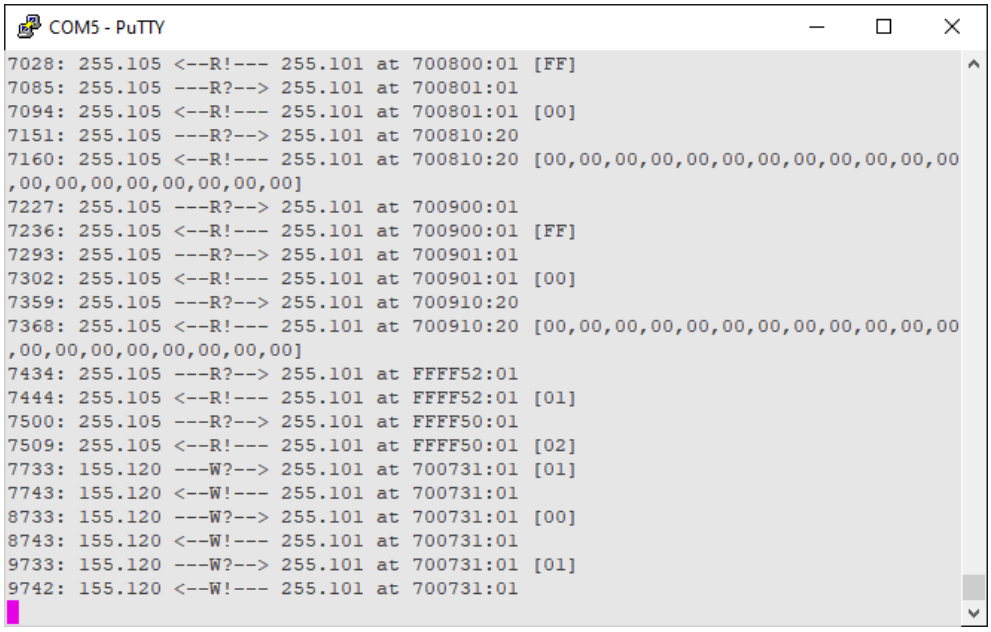


Abbildung 22: Monitorausgabe im Terminal

In der voreingestellten Konfiguration dient der herausgeführte Taster am Gerät dazu, zwischen den Betriebsarten Gateway und Monitor zu wechseln. Ist die Betriebsart Monitor aktiv, kann das Gerät natürlich nicht zeitgleich als Gateway dienen. Erreichbar ist es trotzdem, allerdings nur über andere Busknoten bzw. ein anderes Gateway. Die Dienste von A00111 sind in Anhang F aufgeführt.

11.5 cy4cmd

Die Kommunikation mit den Busstationen eines Cy4NET-Verbundes mittels Austauschs von Rohdatagrammen ist natürlich nicht dienlich. Eine Möglichkeit, mit dem Cy4NET in Verbindung zu treten besteht in der Nutzung des Kommandozeilenwerkzeugs `cy4cmd`. Wie das komplette Projekt steht dieses Softwarewerkzeug unter der freien FreeBSD-Lizenz und kann unter [18] heruntergeladen werden. Vorübersetzte Binaries gibt es bereits für Windows und x64er Linux-Systeme:

```
>.\cy4cmd version

Cy4NET command line tool version 1.0 (URL: www.cy4net.org)
Copyright (c) 2020 Reimar Grasbon
```

Um festzustellen, ob an einer bestimmten Schnittstelle, in diesem Beispiel COM3, ein Gateway wartet, dient der Kommandoparameter `gateway`:

```
>.\cy4cmd gateway COM3
```

Existiert an COM3 ein Gateway vom Typ A00111, erhält man als Antwort dessen Adresse, das Typenschild und, insofern vorhanden, die Gerätebezeichnung:

```
155.123: A00111 V1.1 B00023 "[Cy4NET to serial gateway / monitor]"
```

Gibt es kein Gateway an COM3, bleibt eine Antwort aus:

```
cy4cmd: No response from gateway at COM3.
```

Die gleiche Ausgabe erhält man, wenn man direkt das Typenschild erfragt. Der entsprechende Kommandoparameter dazu lautet `typeplate`. Voraussetzung für die Nutzung von `typeplate` ist jedoch die Kenntnis der entsprechenden Zieladresse des Gerätes:

```
>.\cy4cmd typeplate COM3 155.123
155.123: A00111 V1.1 B00023 "[Cy4NET to serial gateway / monitor]"
```

Üblicherweise ist nicht immer bekannt, an welchen Adressen sich Cy4NET-Teilnehmer befinden. Daher ist es sinnvoll, einen Adressbereich auf mögliche Teilnehmer hin sondieren zu können. Den kompletten Adressraum abzulaufen wäre aber ebenso zeitaufwändig wie sinnlos. Letztendlich muss ein Kommunikationsverbund so strukturiert sein, dass entsprechende Geräte sich gruppieren oder in Untergruppen zusammenfinden. Mittel zum Zweck dazu ist die in Kapitel 5 erwähnt Teilung der Cy4NET-Adresse in den netz- und den gerätespezifischen Teil.

Um einen Adressbereich auf Cy4NET-Teilnehmer hin zu sondieren, dient der Kommandoparameter `scan`. Als Parameter wird, zusätzlich zur der Schnittstelle, der zu untersuchende Netzadressbereich angegeben. Soll dabei nur ein Teilbereich der möglichen 255 Geräteadressen geprüft werden, lässt sich der Suchbereich optional weiter eingrenzen. Gemäß Konvention 2 erhalten neu in Betrieb genommene Geräte eine Adresse aus dem Bereich von `255.100` bis `255.199`. Folgende Anweisung listet sie auf:

```
>.\cy4cmd scan COM3 255 100 199
```

Die Sondierung der 100 Geräteadressen nimmt in etwa 25 Sekunden in Anspruch. Der Vorgang kann in der Ausgabe mitverfolgt werden (hier nur dargestellt bis `255.111`):

```
255.100: ---
255.101: A00001 V2.1 B00101 (Program loader)
255.102: A00351 V1.1 B00102 "[Evalboard 32 OLED]"
255.103: ---
255.104: ---
255.105: A00317 V1.5 B00105 "[Evalboard 644p Kit]"
255.106: ---
255.107: ---
```

```
255.108: ---
255.109: ---
255.110: A00001 V2.1 B00110 (Program loader)
255.111: ---
…
```

Das Kommando `typeplate` nutzt im Hintergrund den Pflichtdienst *CallTypePlate*. Naheliegenderweise verfügen auch die restlichen drei Pflichtdienste über ihre eigenen Kommandos. Die Betriebsart lässt sich mit `application` und `programloader` ändern. Sie sind die Entsprechung zu *EnterApplication* und *EnterProgramLoader*. *ChangeCy4Address* wird durch `address` umgesetzt:

```
>.\cy4cmd scan COM3 255 101 104
255.101: A00001 V2.1 B00101 (Program loader)
255.102: A00351 V1.1 B00102 "[Evalboard 32 OLED]"
255.103: ---
255.104: ---
Scanned: 4, found: 2.

>.\cy4cmd scan COM3 155 101 104
155.101: ---
155.102: ---
155.103: ---
155.104: ---
Scanned: 4, found: 0.

>.\cy4cmd address COM3 255.102 155.103

>.\cy4cmd scan COM3 255 101 104
255.101: A00001 V2.1 B00101 (Program loader)
255.102: ---
255.103: ---
255.104: ---
Scanned: 4, found: 1.

>.\cy4cmd scan COM3 155 101 104
155.101: ---
155.102: ---
155.103: A00351 V1.1 B00102 "[Evalboard 32 OLED]"
155.104: ---
Scanned: 4, found: 1.
```

Die wohl wichtigsten Kommandos im Zusammenhang mit `cy4cmd` sind jedoch die Lese- und Schreibkommandos `read` und `write`. Sie ermöglichen einen direkten Zugriff auf die angebotenen Dienste einer Applikation. So ist das Typenschild von oben auch mittels des normalen Lesekommandoparameters auslesbar:

```
>.\cy4cmd read COM3 155.123 fffff0h 7
160 1 17 17 176 0 35
```

Die Ausgabe des Gelesenen findet als vorzeichenlose 8-Bit Dezimalzahlen statt. Das lässt sich durch den Parameters `<format>` ändern. Er steht als Option für das Kommando `read` wie auch für `write` zur Verfügung und legt fest, wie die aus- bzw. einzugebenden Daten interpretiert werden. So findet die oben genannte Ausgabe beispielsweise hexadezimal statt, indem als Ausgabeformat `hex[1]` angegeben wird:

```
>.\cy4cmd read COM3 155.123 fffff0h 7 hex[1]
A0h 01h 11h 11h B0h 00h 23h
```

Die Zahl in eckigen Klammern bestimmt dabei, wie viele Bytes der Daten für die Erzeugung eines Zahlenwert zusammengefasst, mögliche Breiten sind 1 bis 4 Bytes. Beispiel:

```
>.\cy4cmd read COM3 155.123 fffff0h 7 hex[3]
A00111h 11B000h  Rest: [23h, ?, ?]
```

Die Auswahl möglicher Formate zeigt der Kommandoparameter `help`. Darunter finden sich einige Fixpunktformate, eine 32 Bit Gleitkommazahl gemäß IEEE-754 [19] oder auch UTF-8 Text.

Die Gegenrichtung ermöglicht der Kommandoparameter `write`. Die zu übertragenden Daten werden direkt hinter die obligatorischen Parameter angehängt. Möglich sind bis zu 64 Bytes. Wie bei `read` können sich aber auch hier Daten über mehrere Bytes erstrecken. Wird auf die Angabe eines Formates verzichtet interpretiert `write` die Daten als vorzeichenlose 8-Bit Dezimalzahlen. Die Gerätebezeichnung des Gateways lässt sich beispielsweise wie folgt ändern:

```
>.\cy4cmd write COM3 155.123 ffff00h 65 98 99 0

>.\cy4cmd read COM3 155.123 ffff00h 4
65 98 99 0
```

Eleganter geht es hier jedoch mit der Formatangabe `text`:

```
>.\cy4cmd write COM3 155.123 ffff00h text "Gateway schwarz"

>.\cy4cmd read COM3 155.123 ffff00h 61 text
"Gateway schwarz"
```

Abschließendes Beispiel: Anhang F ist zu entnehmen, dass A00111 unter anderem auch den Dienst *Factory Settings* anbietet. Seine Signatur lautet `FFFF40h:1:WO`. Um zu den Werkseinstellungen zurückzukehren, reicht folgender Aufruf:

```
>.\cy4cmd write COM3 155.123 ffff40h 94

>.\cy4cmd read COM3 155.123 ffff00h 61 text
"[Cy4NET to serial gateway / monitor]"
```

Die aktuelle Programmversion von `cy4Cmd` ist V1.0. In Zukunft wird das Werkzeug sicher verändert und erweitert werden. Den jeweils aktuellen Stand erfährt man mit den Kommandos `help` und `version`.

12 Firmware

Das Ersetzen einer Firmware ist immer ein kritischer Vorgang. Um auf einem System eine neue Firmware zu installieren bedarf es seinerseits einer Firmware, die diese Aufgabe übernimmt. Da es für eine laufende Software jedoch unmöglich ist sich selbst zu ersetzen, steht man vor dem Dilemma: Wie soll ein System eine neue Firmware erhalten?

Eine Möglichkeit diesem, auf den ersten Blick widersprüchlichen, Problem zu begegnen ist, das monolithische Gesamtsystem in Teilsysteme zu zerlegen. Ein Teilsystem als unabhängig arbeitende Sektion ist dann in der Lage die Firmware einer jeweils anderen Sektionen zu ersetzen. Größere Systeme können so über eine Kaskade von Urlader-Programmen stückweise mit ihrer finalen Betriebssoftware ausgestattet werden. Bei Controllersystemen genügt üblicherweise eine Zweiteilung. Dabei ist diese Aufteilung nicht gleich. Es gibt einen kleinen Teil, dessen Aufgabe hauptsächlich darin besteht den größeren mit einer passenden Firmware zu versorgen. Daher bezeichnet man diesen Teil auch als Programmlader oder Programmladermodul. Im üblicherweise größeren Teil ist die eigentliche Systemanwendung angesiedelt. Soll nun die Software der Anwendung ausgetauscht werden, übernimmt diese Aufgabe das Programmladermodul.

Dabei ist eine wichtige Sache zu berücksichtigen: Sollte die neu aufgespielte Firmware fehlerhaft oder unvollständig sein, führt das in der überwiegenden Zahl der Fälle zum Absturz der Anwendung. Die Folge ist ein nicht mehr reagierendes System. Zwar könnte der Programmlader diesen Fehler beheben, indem er die Firmware gegen eine korrigierte Version ersetzt, doch leider gibt es keine Möglichkeit aus einer eingefrorenen Hauptanwendung den Programmlader zu aktivieren. Ein auf diese Weise handlungsunfähiges Gerät wird als „Brick“ („Ziegel“) bezeichnet.

Daher sind Vorkehrungen zu treffen, die die explizite Aktivierung des Programmladers ermöglichen und damit das System aus dieser misslichen Situation befreien können. Wie der Programmlader aktiviert werden kann, ist systemspezifisch. Es gibt Geräte, die eine spezielle Taste oder Tastenkombination dafür bereitstellen. Andere verzweigen in den Programmlader, wenn ihnen kurz nach Systemstart eine speziellen Kommunikationssequenz gesendet wird. Daher bezeichnet man eine solche Art von Programmlader auch als „Bootloader“.

12.1 Das Programmladermodul A00001

Der auf bislang existierenden Geräten zum Einsatz kommende Programmlader war der erste seiner Art und trägt daher die Anwendungstypennummer 1. Er wird durch den bereits in Kapitel 6.3 erwähnten Pflichtdienste *EnterProgramLoader* aktiviert. A00001 ist damit kein klassischer Bootloader, da er jederzeit und nicht nur zu Systemstart zum Einsatz kommen kann. Der entsprechende Gegendienst lautet *EnterApplication*.

Die folgende `cy4cmd` Befehlszeile startet z.B. den Programmlader des Gerätes an Adresse `155.120`, hier ist es das Cy4Nut-Gerät:

```
>.\cy4cmd programloader COM3 155.120
155.120: A00001 V2.1 B00120 (Program loader)
```

Der Wechseln zurück zur Anwendung geschieht durch:

```
>.\cy4cmd application COM3 155.120
155.120: A00272 V1.1 B00120 "[Cy4Nut Reference Implementation]"
```

Wie in Kapitel 6.3 erwähnt, ist der Programmlader eine Applikation wie jede andere auch. Zusätzlich zu den vorgeschriebenen Pflichtdiensten verfügt A00001 aber nur über einen einzigen weiteren Dienst, dieser jedoch mit beträchtlicher Breite:

ProgramMemory: `000000h:<n>:W0`

Der Dienst *ProgramMemory* ist nur beschreibbar. Sein Dienstbereich wird in den Addressraum der jeweiligen Rechnereinheit eingeblendet (memory mapped). Dabei bedeutet Dienstort `000000h` nicht zwingend Adresse 0 im Addressraum. Dienstort `000000h` verweist vielmehr auf die erste Speicheradresse des Programmsspeichers im betroffenen Controller. Gängige STM32 Controller blenden ihren Flashspeicher beispielsweise ab Adresse 08000000h in den Adressraum ein so dass bei diesen Controllern Dienstort `000000h` diese Adresse referenziert. Bei ATmega-Typen hingegen beginnt der Programmspeicher bei 0, hier stimmen also Dienstort und Speicheradresse überein. Die Dienstbreite n überspannt den kompletten Programmspeicher-Bereich. Folglich zeigt `000000h+n-1` auf die letzte beschreibbare Programmspeicherzelle. Der Zugriff ist *random access* und damit komplett frei. Es ist weder nötig, stets den kompletten Programmspeicher zu beschreiben noch muss der Speicher linearer beschrieben werden. Es reicht, gegebenenfalls nur die zu ändernden Bereiche mit neuen Daten zu befüllen. Dabei liegt es in der Verantwortung des Programmladers, Schreibzugriffe zu prüfen und zu verhindern, dass geschützte Speicherbereiche beschrieben werden. Das betrifft insbesondere den Speicherbereich, in dem das Lademodul selbst untergebracht ist.

Da die Nutzlastkapazität eines Cy4NET-Datagramms auf 64 Bytes begrenzt ist, kann der komplette Dienst nicht mittels einer einzigen Cy4NET-Nachricht bedient werden. In folgedessen muss der Speicher fragmentiert, in Datenhäppchen mit bis zu 64 Byte, beladen werden. Dabei achtet der Programmlader darauf dass die übertragenen Datenbytes den Programmlader korrekt erreichen. Andernfalls bleibt der Schreibvorgang protokollkonform unbestätigt. Sind alle Programmspeicherbereiche mit den gewünschten Daten befüllt, beendet der Pflichtdienst *EnterApplication* den Programmlader und startet die dann neue Anwendung.

Im Falle des Cy4Nut-Gerätes ist der Controller vom Typ ATmega328, der mit einen Festwertspeicher (Flashspeicher) von 32 KiB ausgestattet ist. Damit nun möglichst viel Speicher für die Anwendung zur Verfügung steht, sollte das Programmladermodul möglichst klein gehalten sein. Das A00001 des Cy4Nut-Gerätes hat eine Größe von nur 2 KiB, so dass für die Hauptapplikation noch 32-2 = 30 KiB zur Verfügung stehen. Der beschreibbare Speicher beginnt bei Adresse 0000h und endet mit 77FFh, die Dienstbreite n beträgt hier also 30720 Bytes.

Das `cy4cmd`-Werkzeug ist bereits darauf vorbereitet, Programmlader vom Typ A00001 für den Hochladeprozess einer Firmware zu nutzen. Das Programm führt dazu selbstständig alle nötigen Schritte durch, von der Aktivierung des Programmladers über das Hochladen der Firmware bis hin zum abschließenden Start der Anwendung. Aktuell erwartet `cy4cmd` die Firmware im schon betagtem Intel-Hex-Format [20]. Dieses Ausgabeformat kann von den meisten Compilern direkt erzeugt werden und beschreibt die binären Daten in textueller Form. Der folgende Aufruf bestückt das Cy4-Nut-Gerät mit der Firmware A00272 (Ausgabe gekürzt):

```
>.\cy4cmd upload .\A00272_V11_B00120.hex COM3 155.120
155.120: Enter program loader.
155.120: Start upload.
Write 000000h:64 ... done   1%.
Write 000040h:64 ... done   2%.
Write 000080h:64 ... done   3%.
Write 0000C0h:64 ... done   4%.
Write 000100h:64 ... done   5%.
Write 000140h:64 ... done   5%.
Write 000180h:64 ... done   6%.
Write 0001C0h:64 ... done   7%.
…
Write 001C80h:64 ... done  96%.
Write 001CC0h:64 ... done  97%.
Write 001D00h:64 ... done  98%.
Write 001D40h:64 ... done  99%.
Write 001D80h:64 ... done 100%.
Write 001DC0h:62 ... done 100%.
155.120: Upload completed.
155.120: Start application.
155.120: Programming device successful.
```

12.1.1 Der Typcode

Erwähnenswert dabei ist die Namensgebung der Hex-Datei im Beispiel oben: `A00272_V11_B00120.hex`. Der Name ergibt sich aus dem Typcode der Firmware, der sich seinerseits aus den Einträgen des Typenschildes zusammensetzt. Der Typcode ermöglicht eine eindeutige Zuordnung der Anwendung zum Board. Im Beispiel oben ist das Anwendung A00272, in der Version 1.1 für das Board B00120. Die Benennung der Firmware nach diesem Schema ist nicht zwingend, aber hilfreich. Letztlich ist diese Kennung durch das Vorhandensein des Pflichtdienstes *Cy4TypePlate* ja ohnehin integraler Bestandteil jeder Firmware, man muss sie nur auslesen. Doch da liegt das Problem: Der Ort des Typenschildes ist nach dem Übersetzungsvorgang in den reinen Binärdaten der Firmware nicht unmittelbar mehr bekannt. Zwar haben die Daten des Typenschilds ein markantes Format, allerdings gewährleistet das allein keine eindeutige Identifikation im Code.

Um trotzdem den Typcode der Firmware ermitteln zu können, wird die Stelle, an der das Typenschild im Code vorkommt, mit einer speziellen Kennung markiert. Im Quellcodeauszug unten ist diese Kennung zu sehen. Es handelt sich um die Zeichenkette „Cy4Code“ unmittelbar vor Instanzierung des Typenschildes.

```
static void
OnCallTypePlate(const Cy4Service* service, Cy4Message* message)
{
    static const UInt08  typePlate[][7] PROGMEM = {  {'C','y','4','C','o','d','e'},
                                                      CY4TYPEPLATE(DEV_APPL_TYPE,
                                                                   DEV_APPL_VERS,
                                                                   DEV_BOARD_TYPE)
                                                   };

    UInt08 j = (UInt08)(CY4MESSAGE_Location(message) - CY4SERVICE_Base(service));
    for (UInt08 i=0; i<CY4MESSAGE_Count(message); i++)
    {   CY4MESSAGE_Data(message)[i] = pgm_read_byte(&typePlate[1][j]);
        j++;
    }
}
```

Untergebracht ist die statische Datenstruktur innerhalb des Antwortbehandlers des *Cy4TypePlate*-Dienstes. Die Kennung ist so platziert, dass sie direkt vor dem Typenschild im Speicher auftaucht. Durch die Compilierung und dem anschließendem Link-Vorgang landet diese Kennung später unmittelbar vor dem Typenschild im Binärcode. Abbildung 23 zeigt einen Speicherauszug der resultierenden Hex-Datei. Die entsprechenden Zeilen sind markiert.

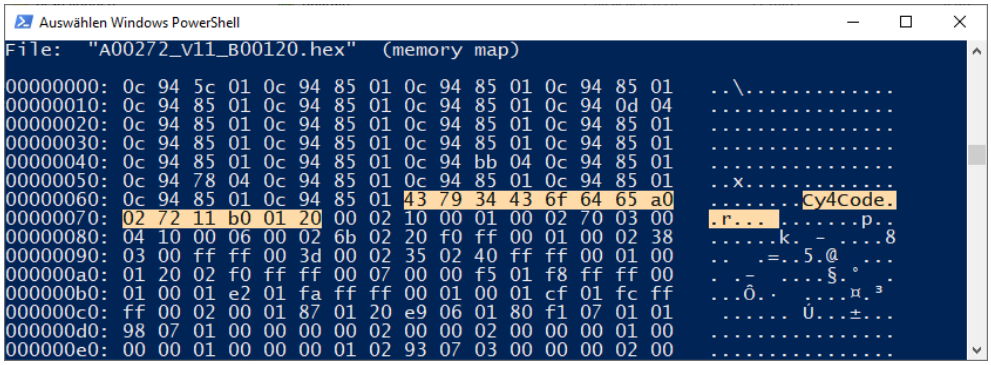


Abbildung 23: Speicherauszug der Firmware-Binärdatei.

Ist durch die eingesetzte Prozessorarchitektur oder dem Compilertyp nicht gewährleistet, dass die Einträge in aufsteigenden Byteadressen liegen, muss gegebenenfalls die Darstellung im Code angepasst werden.

Durch den Zusatz dieser Kennung ist letztlich noch immer nicht ausgeschlossen, dass Mehrdeutigkeiten in den übersetzten Binärdaten vorkommen. Jedoch ist die Wahrscheinlichkeit sehr viel geringer.

Das cy4cmd kann Informationen aus einer Firmware-Hexdatei extrahieren und anzeigen. Neben Größe, Speicherverbrauch oder Adressbereich wird auch der ermittelte Typcode angezeigt:

```
>.\cy4cmd info .\Firmware.hex

   File name ............: Firmware.hex
   Binary size ..........: 7678 bytes
   Coverage .............: 7678 bytes (100%)
   Address range ........: 0000h .. 1DFDh
   Cy4 application ......: A00272, V1.1
   Cy4 board target .....: B00120
   Firmware type code ...: A00272_V11_B00120
```

Darüber hinaus bietet das cy4cmd die Möglichkeit, eine bestehende Firmware-Hexdatei entsprechend ihres enthaltenen Typcodes automatisch umzubenennen. Der zugehörige Kommandoparameters heißt rename. Der Dateiname der oben gezeigten Datei Firmware.hex würde durch rename umgeändert in A00272_V11_B00120.hex.

Unabhängig von der Benennung der Datei prüft das cy4Cmd Werkzeug vor einem Hochladevorgang, ob die angegebene Firmware überhaupt für das Board vorgesehen ist. Das ist dann der Fall, wenn die Boardtypennummern übereinstimmen. Andernfalls ist die Firmware für ein anderes Board, sprich eine andere Hardware konzipiert. Das Aufspielen einer solchen Firmware wird üblicherweise in einer nicht funktionierenden Anwendung münden. Um einen solchen Fall zu vermeiden, prüft cy4cmd im Vorfeld die Nummern und verweigert gegebenenfalls den Hochladevorgang.

```
 .\cy4cmd upload .\A00310_V16_B00105.hex COM3 155.120
cy4cmd: Firmware in ".\A00310_V16_B00105.hex" is destined for different board.
cy4cmd: Upload denied. (Use option force).
```

Möchte man trotzdem die angegebene Firmware hochladen, den Hochladevorgang also erzwingen, ist das mittels der Option force möglich. Es gibt Fälle, in denen das Aufspielen einer artfremden Firmware sinnvoll sein kann.

Hinweis: Der aktuell im Einsatz befindliche Programmlader trägt die Applikationstypennummer A00001 und liegt in der Version V2.1 vor. Um möglichst sicher zu stellen, dass eine Applikation das Gerät komplett erreicht, darf das Aufspielen einer Anwendung nicht für längere Zeit unterbrochen werden. Entsteht nach dem Senden einer Nachrichten an den Programmlader eine Zeitlücke von mehr als 3 Sekunden, verwirft der Programmlader die aktuell übermittelte Anwendung. Folge ist, dass der Programmlader nach Abschluss des Programmiervorganges die Anwendung nicht aktiviert. Der Aufruf von EnterApplication bleibt wirkungslos und führt wieder zurück in das Programmladermodul. Man sollte also dem Hochladevorgang keine längeren Pausen gönnen.

12.1.2 Signalisierung

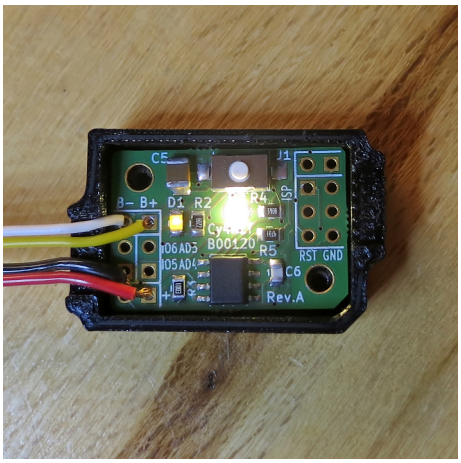


Abbildung 24: Cy4Nut-LEDs

Auf fast allen aktuellen Cy4NET-Boards finden sich LEDs. Welche vorhanden sind, ist boardspezifisch, welchem Zwecken sie dienen ist anwendungsspezifisch. Häufig werden sie genutzt um Systemzustände anzuzeigen oder um die Buskommunikation zu signalisieren. Wie in Abbildung 24 zu sehen, verfügt das Cy4Nut beispielsweise über zwei Leuchtmittel. Eine kleine, schwach leuchtende LED und eine große, helle, hier auch als Info-LED bezeichnet. Platziert sind beide als D1 und D2 auf der Vorderseite des PCBs. In der Referenzimplementierung A00727 kann mittels des Dienstes *SystemLedTask* die Aufgabe der kleine LED verändert werden. So ist es beispielsweise, möglich sie zur Signalisierung der Busaktivitäten einzusetzen. Sie leuchtet in dem Fall jedes mal für ca. 25 ms auf, wenn mit dem Gerät über den Bus kommuniziert wird. Auch die Aufgabe der Info-LED kann geändert werden. Der dafür zuständige Dienst lautet *InfoLedState*. Wie die Dienste einzusetzen sind, zeigt das Application-Modul der Referenzimplementierung und das Anwendungsbeispiel in Kapitel 14.

Auch im Programmlader A00001 wird, insofern vorhanden, eine LED zur Signalisierung eingesetzt. Welche LED das ist, wird in der Programmlader-Anwendung des jeweiligen Boards festgelegt. Sie dient als Aktivitätsanzeige. Solange das Programmladermodul aktiv ist, signalisiert die Anwendung den Zustand durch vier schnelle Lichtblitze, die sich im Abstand von ungefähr einer halben Sekunde wiederholen. Diese markante Signalfolge ist von außen gut wahrnehmbar.

12.1.3 Rettung in den Programmlader

Wie oben bereits erwähnt, ist nicht immer gewährleistet, dass der Programmlader auch mittels *EnterProgramLoader* erreicht werden kann. Im Falle einer nicht reagierenden Hauptanwendung ist es daher unerlässlich, eine Hintertür zu haben. Durch sie soll sichergestellt werden, dass der Programmlader jederzeit erreicht werden kann. Wie diese Hintertür realisiert ist, ist nicht im Programmlader, auch nicht im A00001 festgelegt, sondern hängt von der jeweiligen Hardware ab. Im Falle der aktuell mit Atmega-Controllern ausgestatteten Boards kommt dafür der Resetknopf zum Einsatz.

Die Resetlogik des Controllers kann feststellen, welche Ursache sein Neustart hatte. Neben internen Resets wie z.B. dem Watchdog, können auch von außen zugeführte Gegebenheiten als Neustart erkannt werden. Dazu gehört z.B. der Spannungseinbruch (brown-out) oder auch der Einschalt-Vorgang (Power-On). Die explizite Betätigung des auf dem Board befindlichen Resettasters gehört ebenfalls dazu. Als Ursache wird in dem Fall der externe Reset (External) diagnostiziert. Dieser Art des Resets wird nun dazu eingesetzt, den Programmlader zu aktivieren. Reagiert die Hauptapplikation nicht mehr, wird durch einen Druck auf den Resetknopf explizit der Programmlader aktiviert. Zu erkennen ist dieser Zustand dann an der oben beschriebenen Signalisierung.

Es ist übrigens nicht ausgeschlossen, dass nach dem Start (oder Neustart) des Controllers mehrere Resetursachen angegeben sind. Ist beispielsweise der Reseteingang des Controllers so beschaltet, dass dort das Resetsignal verzögert ausgelöst wird, kann die Reset-Logik als Ursache für den Start einerseits einen Power-On-Reset, zugleich aber auch einen externen Reset als Ursache melden. Bei Mehrfachnennungen sind die Angaben daher zu priorisieren.

13 Inbetriebnahme eines Cy4NET-Gerätes

Jedes Rechnersystem ist zu Anfang völlig handlungsunfähig. Um agieren zu können bedarf es einer Software. Wie in vorherigen Kapitel erklärt, ist das Vorhandensein eines Programmladers nötig um das System mit der gewünschte Firmware und den damit verbundenen Eigenschaften auszustatten. Man benötigt also eine softwaremäßige Grundausstattung um überhaupt Software aufspielen zu können. Die Frage zu diesem Henne-Ei-Problem lautet daher: Wie kommt der Programmlader auf ein zu Beginn komplett leeres Controllersystem?

Genauer müsste die Frage lauten: Wie befüllt man den Festwertspeicher eines Controllers initial mit Daten? Was in früheren Jahren ein mitunter aufwändigen Prozess darstellte, ist dank der heute üblichen Flashspeichertechnologie, wesentlich komfortabler geworden. Früher wie heute, sind dafür aber spezielle Programmiergeräte nötig, die jeder Hersteller zu seinen Controllerprodukten anbietet. Zwar gibt es mit JTAG [21] einen industriellen Standard für das Programmieren und das Testen von Schaltungen und auch ISP [22] ist weit verbreitet. Doch die konkrete Ausgestaltung in Sachen Anschlüssen und Pegeln ist meist herstellerspezifisch.

Für die Controller der ATmega-Reihe stehen drei Verfahren für die sogenannte *In-System*-Programmierung zur Verfügung: ISP, JTAG und DebugWire. Während JTAG und DebugWire neben der Programmierung auch sogenanntes On-Chip-Debugging ermöglichen, ist ISP rein für das Beschreiben des integrierten Programmspeicher vorgesehen. Die Programmierung mittels ISP wird von allen Controllern der ATmega-Reihe unterstützt, JTAG nur von den größeren Chips. DebugWire (auch debugWIRE geschrieben) ist eine proprietäre Programmiermethode des Herstellers Microchip (vormals Atmel). Als jüngste Methode steht sie auch erst bei den neueren Controller-Derivaten zur Verfügung. Weitere Methoden der In-System-Programmierung sind in der Entstehung.

13.1 Programmierung am Beispiel des Gateways

Der Controller des B00023 ist ein ATmega644P. Er gehört zu den größeren Atmega-Typen und bietet als Programmierschnittstelle ISP (Abbildung 25) sowie auch JTAG. Wie Anhang E zu entnehmen ist, sind beide als Steckerleisten H1 und H2 auf dem PCB verfügbar. Welche Schnittstelle für das Beschreiben des Programmladers genutzt wird, entscheidet sich meist durch den verfügbaren Programmieradapter. Abbildung 26 zeigt eine Auswahl unterschiedlicher Geräte. Einige stammen direkt vom Hersteller des Controllers, andere sind von Fremdherstellern.

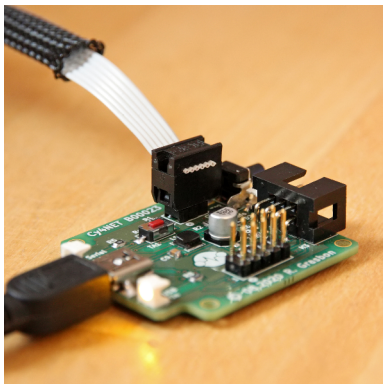


Abbildung 25: Gateway mit ISP-Verbindung

Abbildung 26: Verschiedene. Programmieradapter, v.l.: AVR-ISP-mk2, Olimex, Atmel-ICE

Um den Programmlader auf das Board zu bringen, bedarf es neben dem Programmieradapter noch einer unterstützenden Software. Auch hier gibt es mehrere Alternativen, zwei sind in den folgenden Abschnitten vorgestellt.

13.1.1 Microchip-Studio-IDE

Ist man daran interessiert den Code der Cy4NET-Projekte selbst zu sichten, zu übersetzen oder zu debuggen, erleichtert eine IDE die Vertiefung in die Materie. Da alle aktuellen Cy4NET-Anwendungen bis hin zur Referenzimplementierung als Microchip-Studio-Projekte vorliegen, wird das Aufspielen des Programmladers anhand dieser Entwicklungsumgebung erläutert. Aber auch jede andere IDE kann genutzt werden. Zur Funktionsweise und zum Umgang mit einer IDE sei auf weiterführende Dokumentation verwiesen.

Je nach Einstellung wird beim Übersetzungsvorgang das Kompilat im Unterverzeichnis Release oder Debug abgelegt. Um die resultierende Binärdatei aufzuspielen, verbindet man das bestromte Controllboard mit dem Programmieradapter und öffnet im Microchip-Studio unter *Tool/Device Programming* das Dialogfenster für die Programmierung. Nach Auswahl des Programmieradapter bei *Tool* und des Controllers unter *Device*, wechselt man auf die Karte *Memories*. Von dort aus lässt sich nun die in Release oder Debug befindliche Binärdatei auf den angeschlossenen Controller übertragen. In Abbildung 27 ist das entsprechende Fenster der IDE zu sehen.

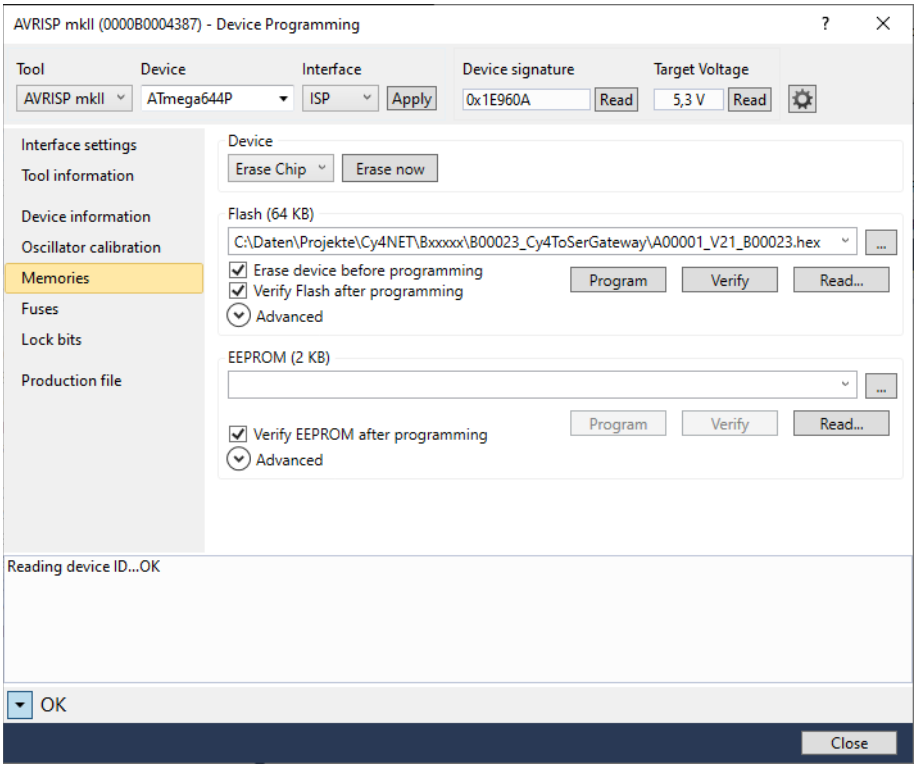


Abbildung 27: Das Device Programming Fenster im Atmel-Studio

Dabei ist darauf zu achten, dass man das Board mit der passenden Programmladersoftware bestückt. Eine entsprechenden Benennung nach dem Typcode (vergl. Kapitel 12.1.1) vermindert Fehler in diesem Zusammenhang.

Danach geht es an die *Fuses*, genauer gesagt, an die *Fuse*-Bits. Bei den *Fuses* handelt es sich nicht um Sicherungen, sondern um eine Möglichkeit, den Controller für seine Arbeit im jeweiligen Umfeld zu konfigurieren. Der Name *Fuses* soll verdeutlichen, dass vergleichsweise dem Verschmelzen (oder Trennen) von Kontaktbrücken bestimmte Einstellungen vorgenommen werden. Beispielsweise wird durch *Fuses* festgelegt, ob der Controller seinen internen Taktgeber nutzt oder extern mit Pulsen versorgt werden muss. *Fuses* bestimmen, welche Programmierschnittstellen aktiv sind oder ob der Reset-Pin nur als normaler Port-Pin agieren soll. Welche *Fuses* letztendlich für welchen Controllertyp genau zur Verfügung stehen und was sie bedeuten, kann dem jeweiligen Datenblatt entnommen werden. Abbildung 28 zeigt, welche *Fuses* im Falle des Gateways zu setzen sind.

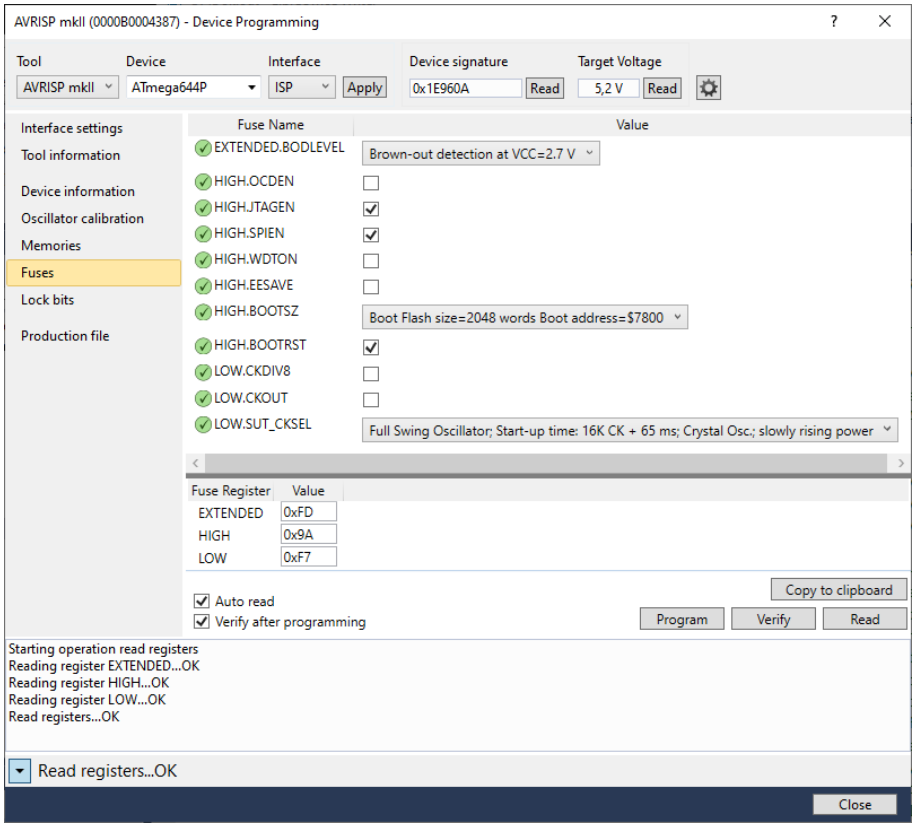


Abbildung 28: Fuse-Bits des Gateways

Untergebracht sind die *Fuse*-Bits in den *Fuse*-Registern. Derer gibt es beim ATmega644P drei. Ihre Bits repräsentieren die *Fuses*, die gegebenenfalls auch gruppiert sind. Nicht immer sind alle Bits der Register belegt, so dass zu einer Konfiguration manchmal mehrere Wertausprägungen existieren. Die drei *Fuses*-Register des Gateways haben die Werte FDh, 9Ah und F7h. Diese Einstellung bewirkt Folgendes:

BODLEVEL: Fällt die Versorgungsspannung auch nur kurzfristig unter eine bestimmten Grenze, führt das zu einem Brown-Out-Reset. Der Controller schützt sich damit gegen Minderspannungen, die undefiniertes Verhalten auslösen können. Das ist nicht zu unterschätzen, insbesondere zu Anfang, wo die Versorgungsspannung ansteigt bzw. am Ende, wo sie zusammenbricht. Beim Gateway ist ein *Brown-Out-Detection-Level* von 2,7 V ausgewählt.

Die *Fuses* JTAG und ISP aktivieren die gleichnamigen Programmierschnittstellen. Mindestens eine der beiden sollte immer aktiv bleiben, da ansonsten der Zugang zum Controller abgeschnitten ist.

Die BOOT-*Fuses* legen fest, dass der Prozessor nach einem Reset seine Arbeit nicht, wie üblich, an Adresse 0000h beginnt, sondern in diesem Fall erst bei 7800h. Da der ATmega bei der Adressierung hier mit Wortbreiten von 16 Bit rechnet, liegt seine Byte-Startadresse beim Doppelten, also F000h und damit 4 KiB vor Ende des Speichers. Dort ist der Programmlader angesiedelt, der auf diese Weise nach einem Start als erstes aufgerufen wird. Im Falle es Gateways nimmt er 4 KiB in Beschlag.

Die letzte Gruppe von *Fuses*, CKSEL, bestimmt, von wo aus der Controller seinen Takt bezieht. Im Ursprungszustand ist der interne Taktgeber aktiv. Da dieser jedoch verhältnismäßig ungenau ist, wird hier die Taktgebung auf einen externen Quarz verlegt. Darüber hinaus wird dem System ein wenig Zeit zum Einschwingen des Oszillators und zur Stabilisierung der Versorgungsspannung eingeräumt. In diesem Fall sind es 16000 Zyklen plus zusätzlicher 65 ms, bevor der Controller seine Arbeit aufnimmt.

Abschließend seien noch die *Lock*-Bits erwähnt. Sie sind eine besondere Gattung der *Fuse*-Bits und haben daher im Programmier-Dialogfenster des Microchip-Studios eine eigene Seite. Im Gegensatz zu *Fuses*, die die Arbeitskonfiguration des Controllers festlegen, bestimmen die *Lock*-Bits sicherheitsrelevante Aspekte. Durch sie wird festgelegt, welche Zugriffe auf welche Sektionen des Programmspeichers erlaubt sind. Gemeint sind Applikations- oder Boot-Sektion. So kann durch sie beispielsweise das nachträgliche Auslesen des Flashspeicher und damit das Exportieren des Programmcodes verhindert werden. Oder sie schützen Sektionen gegen unabsichtliche oder auch absichtliche Schreibvorgänge aus dem Programmcode heraus.

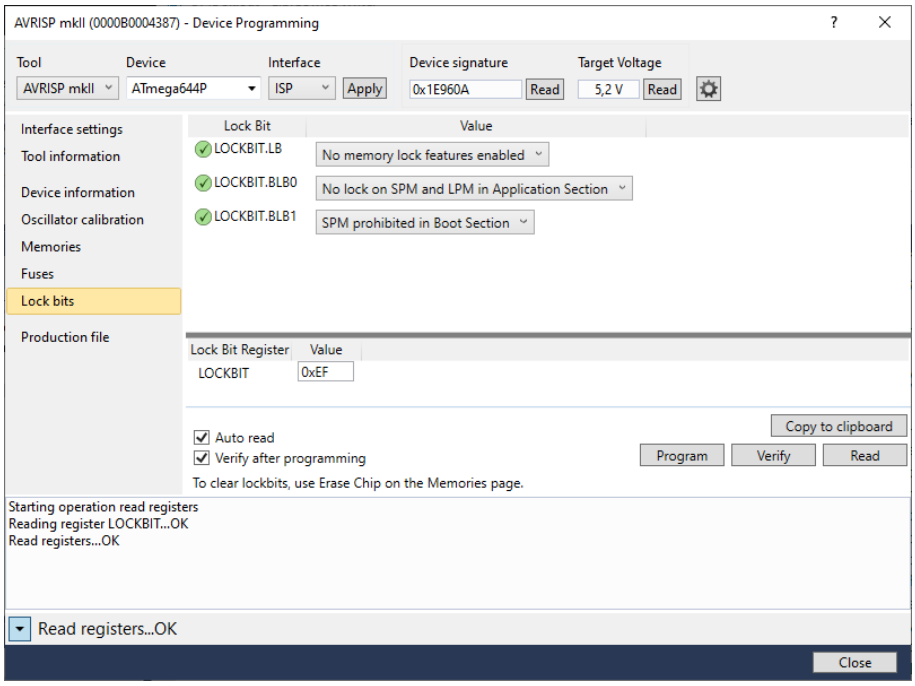


Abbildung 29: Mögliche Lock-Bit-Einstellungen des Gateways

Ob *Lock*-Bits gesetzt werden oder nicht, liegt im Ermessen des Entwicklers. Im Falle des Cy4Net müssen hinsichtlich des Programmcodes eigentlich keine Schutzmaßnahmen getroffen werden. Der Code ist frei verfügbar, das Auslesen des Programmspeichers zu verbieten ergibt für diese Anwendungen daher keinen Sinn. Überlegenswert ist jedoch, die Programmlader-Sektion gegen Schreibattacken aus der Applikations-Sektion zu schützen. Damit ist sichergestellt, dass das Programmladermodul stets unbeschädigt und damit einsatzfähig bleibt. Um das zu erreichen wird, wie in Abbildung 29 zu sehen, der Wert EFh ins *Lock*-Register geschrieben.

13.1.2 AVR-Dude

Wer sich nicht für den Tiefen der Programmierung interessiert, sondern seine Aktivität rein auf das Aufspielen der Firmware begrenzen möchte, sollte anstelle einer IDE lieber ein Kommandozeilenwerkzeug einsetzen. Ein hilfreiches Programm für diesen Zweck stammt aus dem Open-Source Projekt AVR-Dude [23]. Mit entsprechender Zusatzsoftware lässt es sich sogar um eine GUI ergänzen. Das Werkzeug gibt es als fertig ausführbares Programm für Linux und Windows. Eine Einführung zu AVR-Dude ist in der Artikelsammlung auf Mikrocontroller.net [24] zu finden.

Verzichtet man auf die GUI, besteht die eleganteste Methode darin, alle notwendigen Schritte für das Hochladen durch ein Script erledigen zu lassen. Die automatisierte Bestückung eines Boards mit der gewünschten Firmware und allen Konfigurationen ist wesentlich weniger fehlerträchtig, als in der IDE manuell alle Eingaben zu tätigen. Für das Gateway könnte folgendes Script dienen, hier als Beispiel für die DOS/Powershell:

```
@echo off
set DEVICE=m644p
set PROGRAMMER=avrispmkII
set HEXFILE=A00001_V21_B00023.hex
set FUSE_EXT=0xFD
set FUSE_HIGH=0x9A
set FUSE_LOW=0xF7
set LOCKBITS=0xEF
avrdude.exe -p %DEVICE% -c %PROGRAMMER% -U flash:w:%HEXFILE%:i -U lfuse:w: \
%FUSE_LOW%:m -U hfuse:w:%FUSE_HIGH%:m -U efuse:w:%FUSE_EXT%:m -U lock:w: \
%LOCKBITS%:m
if errorlevel 1 (
    echo Programmierung fehlgeschlagen
    pause
    exit /b %errorlevel%
)
echo Programmierung erfolgreich
pause
```

Als Erstes wird der Controller komplett zurückgesetzt. Das überschreibt nicht nur den Flashspeicher komplett mit Einsen (FFh), sondern versetzt auch die *Lock*-Bits in den Anfangszustand. Erst danach sind weitere Zugriffe auf den Speicher möglich. Anschließend wird das angegebene Binary aufgespielt gefolgt von Einstellungen der *Fuse*- und *Lock*-Bits. Nach jedem Arbeitsschritt wird geprüft, ob das Geschriebene mit dem Gewünschten übereinstimmt. Sollte das nicht der Fall sein, wird der Vorgang mit einem Fehler beendet.

> **Hinweis**: Da beide Methoden individuelle Adapter-Treiber installieren, können die oben genannten Methoden Microchip-Studio und AVR-Dude nicht gleichzeitig auf einem Rechner genutzt werden. Man muss sich in diesem Fall also für den Einsatz einer Alternativen entscheiden.

14 Anwendungsbeispiel

Als Abschluss folgt nun ein kleines Fallbeispiel. Dazu kehren wir kurz zurück an den Anfang dieser Dokumentation. In der Einführung ist in Abbildung 1 der Datenbus als grundlegende Struktur des Cy4NET dargestellt. Die Form der Busstationen in der Abbildung dort ist jedoch kein Zufall. Sie zeigen die Busstationen als stilisierte Form der Cy4Nut-Leiterplatte aus Anhang D. In dem nun folgenden Beispiel soll das Zusammenwirken zweier Busstationen demonstriert werden. Aufgabe eines jeden Gerätes soll darin besteht, die Info-LED des jeweils anderen blinken zu lassen. Es ist quasi das *Hello-World* der Mikrocontroller erweitert auf das Multimaster-Netzwerk des Cy4NET-Systems.

14.1 Ausgangssituation

Akteure in diesem Beispiel sind zwei Cy4Nut-Geräte, d.h. Boards vom Typ B00120. Die Geräte sind zuvor mittels der im vorangegangenen Kapitel vermittelten Inbetriebnahme mit einem Programmladermodul ausgestattet worden, weiteres ist noch nichts geschehen.

Beide Geräte sind an den Bus angeschlossen, auf den mittels eines Gateways und des Kommandozeilenwerkzeug `cy4cmd` extern zugegriffen werden kann. Abbildung 30 zeigt erneut die Situation aus Abbildung 1, hier jedoch mit zwei realen Busstationen.

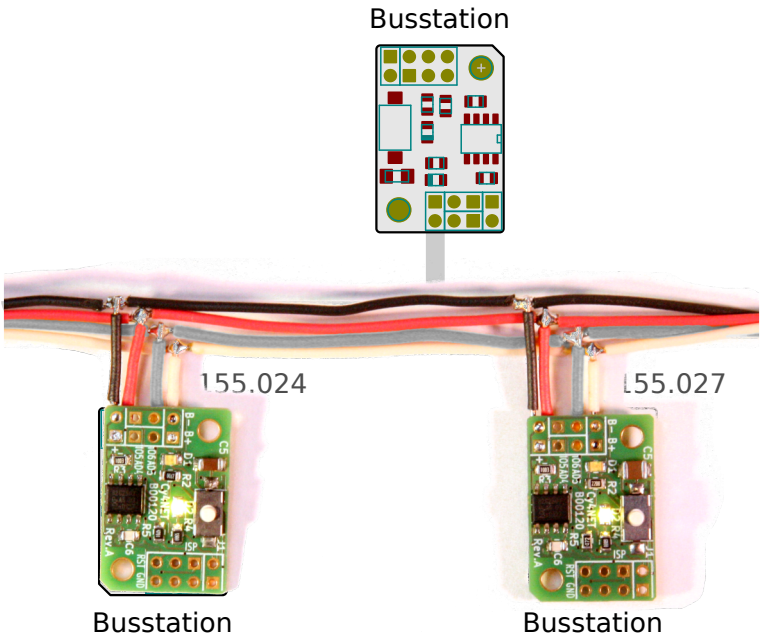


Abbildung 30: Bustopologie mit zwei Cy4Nut-Geräten

14.2 Vorbereitung

Da beide Geräte nach Inbetriebnahme die Werksadresse `255.120` haben, besteht die erste Aufgabe darin, die Geräte gemäß Konvention 3 von der Landing Page auf eine jeweils individuelle Adresse zu versetzen. Wenn, wie hier, beide Geräte bereits am Bus angeschlossen sind, muss zur Vergabe der Adressen einer der beiden temporär von der Buskommunikation ausgeschlossen werden. Erst dann kann die Adresse des anderen individuell versetzt werden. Das geschieht durch drücken und gedrückt halten der Reset-Taste an einem der Geräte. Für die Zeit des Reset-Signals ist das Gerät quasi eingefroren. Nun können wir die Adresse des anderen verändern:

```
.\cy4cmd address COM3 255.120 155.24
```

Danach ist das zweite Gerät an der Reihe. Die Reset-Taste kann losgelassen werden, die Blockade wird aufgehoben. Jetzt erhält das zweite Gerät seine neue Adresse:

```
.\cy4cmd address COM3 255.120 155.27
```

Beide Geräte befinden sich nun auf den individuellen Adressen `155.024` und `155.027`:

```
>.\cy4cmd scan COM3 155 20 30
155.020: ---
155.021: ---
155.022: ---
155.023: ---
155.024: A00001 V2.1 B00120 (Program loader)
155.025: ---
155.026: ---
155.027: A00001 V2.1 B00120 (Program loader)
155.028: ---
155.029: ---
155.030: ---
Probed: 11, found: 2.
```

Als Nächstes erhalten beide Busstationen ihre Hauptanwendung. Für das Cy4Net steht dafür die Referenzimplementierung `A00272_V11_B00120` zur Verfügung. Das Aufspielen der Firmware geschieht nacheinander:

```
>.\cy4cmd upload .\A00272_V11_B00120.hex COM3 155.24
155.024: Enter program loader.
155.024: Start upload.
Write 000000h:64 ... done   1%.
```

```
Write 000040h:64 ... done   2%.
…
Write 001DC0h:62 ... done 100%.
155.024: Upload completed.
155.024: Start application.
155.024: Programming device successful.

>.\cy4cmd upload .\A00272_V11_B00120.hex COM3 155.27
155.027: Enter program loader.
155.027: Start upload.
Write 000000h:64 ... done   1%.
Write 000040h:64 ... done   2%.
…
Write 001DC0h:62 ... done 100%.
155.027: Upload completed.
155.027: Start application.
155.027: Programming device successful.
```

Wenn das Hochladen fehlerfrei abgeschlossen wurden, sind beide Geräte nun mit ihrer Hauptanwendung ausgestattet. Man erkennt es daran, dass die farbige System-LED vergleichbar einem Herzschlag pulsiert.

```
>.\cy4cmd scan COM3 155 20 30
155.020: ---
155.021: ---
155.022: ---
155.023: ---
155.024: A00272 V1.1 B00120 "[Cy4Nut Reference Implementation]"
155.025: ---
155.026: ---
155.027: A00272 V1.1 B00120 "[Cy4Nut Reference Implementation]"
155.028: ---
155.029: ---
155.030: ---
Probed: 11, found: 2.
```

Damit sind die Vorarbeiten abgeschlossen.

14.3 Konfiguration

Welche Dienste Anwendung A000272 anbietet, kann im Application-Modul eingesehen werden. Um eine wechselseitige Kommunikation aufzubauen benötigen wir derer zwei. Der Erste schaltet die Info-LED, der Zweite kümmert sich um das Blinken.

InfoLedState	**100200h:1:RW**

Beschreibung: Liefert oder ändert den Zustand der Info-LED.

Belegung:

0
INF

INF	Zustand der Info-LED: 0 = Aus, 1 = Ein, x = Unverändert

Initial ist die Info-LED ausgeschaltet. Um die LED des Boards an `155.024` zum Leuchten zu bringen, genügt:

```
>.\cy4cmd write COM3 155.24 100200h 1
```

Eine 0 am Ende schaltet die LED wieder ab. Um also die Info-LED eines Boards blinken zu lassen, reicht es, periodisch auf `INF` dieses Dienstes abwechselnd den Wert 0 und 1 schreiben.

Der zweite Dienst ermöglicht genau dieses. Er löst in festgelegten Zeitabständen das Versenden einer Nachricht aus. Man beachte: Wie genau diese Nachricht aussieht, ist im Dienst selbst nicht festgelegt. Dort ist lediglich das Wiederholintervall und das Ziel dieser sogenannte *Push-Nachricht* spezifiziert.

PeriodicPushMessage	**100400h:6:RW**

Beschreibung: Zieladresse und Wiederholintervall der periodisch zu sendenden Push-Nachricht.

Belegung:

0	1	2	3	4	5
INT	NET	DEV	LC0	LC1	LC2

INT	Intervall für das Versenden der Push-Nachrichten: 1 .. 100 [s÷10], 0 = Aus
NET	Push-Nachricht Ziel, Netzadresse: 0 .. 255
DEV	Push-Nachricht Ziel, Geräteadresse: 0 .. 255
LCx	Push-Nachricht Ziel, Dienstort: 0 .. FFFFFFh

Die vorgesehene Push-Nachricht ist in Anwendung A00272 bereits so implementiert, dass sie bei jedem Aufruf alternierend 0 oder 1 in das durch *PeriodicPushMessage* bestimmte Ziel schreibt. Der entsprechende Nachrichtenausstatter heißt `SetupInfoLedStateMessage()` und ist ebenfalls im Application-Modul zu finden.

Als letzter Schritt bleibt nun, bei beiden Boards den Dienst *InfoLedState* des jeweils anderen als Push-Nachrichen Ziel des eigenen *PeriodicPushMessage* einzutragen, mitsamt des gewünschten

Blinkintervalls. Die Dateneingabe ist aufgrund der heterogenen Struktur des Dienstes etwas umständlich. Insbesondere den Dienstort (100200h) muss man, wenn man als Format die dezimale u8-Eingabe wählt, in die drei Werte 16, 02 und 00 zerlegen:

```
.\cy4cmd write COM3 155.24 100400h 5 155 27 16 02 00

.\cy4cmd write COM3 155.27 100400h 2 155 24 16 02 00
```

Das Gerät an `155.024` sendet nun nach Ablauf von 500 ms eine Nachricht an das zweite Gerät (an `155.027`) und wechselt damit den Zustand dessen Info-LED. In gleicher Weise verfährt das Zweite und lässt die LED des ersten blinken, allerdings mit 200 ms etwas schneller.

Die Sendevorgänge können sichtbar gemacht werden, indem die Aufgabe der System-LED von Herzschlag-Signal auf Busaktivität umgeschaltet wird. Der Dienst, der das ermöglicht, heißt *SystemLedTask*, er bestimmt die Aufgabe der System-LED.

SystemLedTask	**FFF020h:1:RW**

Beschreibung: Liefert oder ändert die Lichtsignale der System-LED. Die Einstellung wird gespeichert. Folgende Signale können ausgewählt werden:

- 0 = Kein Signal (Ausgeschaltet)
- 1 = Leuchtet durchgehend
- 2 = Busaktivität
- 3 = Herzschlag-Signal (Voreingestellt)
- 4 = Busaktivität und Herzschlag-Signal
- sonst = wirkungslos

Belegung:

0
TSK

TSK	Lichtsignale der System-LED: 0 .. 4

Damit lässt sich die Kommunikation auf dem Bus sichtbar machen:

```
.\cy4cmd write COM3 155.24 FFF020h 2

.\cy4cmd write COM3 155.27 FFF020h 2
```

Da in dieser Konstellation beide Geräte untereinander kommunizieren, zeigen beide LEDs die gleiche Aktivität an.

Beide Busstationen senden ihre Anforderungen selbstinitiiert und asynchron, ohne Koordinator als Multi-Master.

Anhang A Einteilung des Dienst-Speicherraumes

Damit unterschiedliche Geräten bei der Nutzung der Dienste keine unerwarteten Nebeneffekte hervorrufen, ist der Speicherraum der Dienste in verschiedene Funktionsbereiche unterteilt. Abbildung 31 liefert eine Übersicht der Segmentierung.

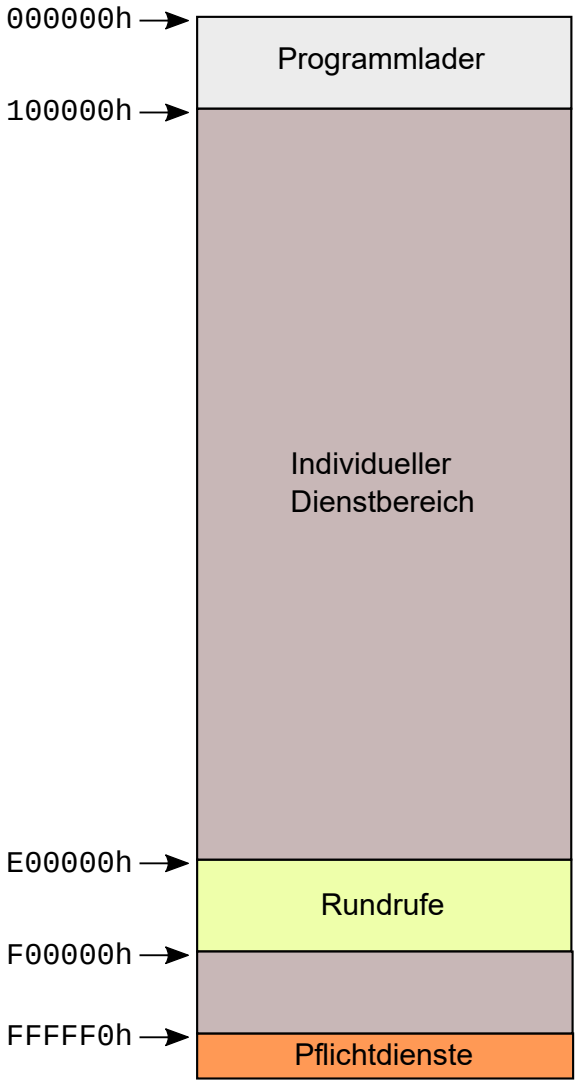


Abbildung 31: Funktionsbereiche im Dienst-Speicherraum

Anhang B Nutzung der Rundrufzone Exxxxxh

Grundsätzlich sind Rundruf nur im Dienstortbereich von E00000h bis EFFFFFh erlaubt. Um gemeinschaftliche Dienste als Rundrufe nutzen zu können, müssen sich alle Geräte über einen einheitlichen Satz an Diensten in der Rundruf-Zone verständigen.

Mit Einsatz bestehender Busstationen haben sich bereits einige Rundruf-Dienste etabliert. Insofern sie entsprechenden Geräte am Bus verfügbar sind, verteilen sie allgemein nützliche Informationen, wie Umgebungsdaten oder Zeitangaben. Die Rundrufe sind an 0.0 adressiert, können daher von jeder Station am Bus entgegengenommen werden. Die Liste vordefinierter Dienste ist nicht abgeschlossen, sondern wird sich mit der Zeit entwickeln. Einsatzort für allgemeine Mitteilungen wird der Bereich von E00000h bis E10000h sein. Um unerwünschte Nebeneffekte zu vermeiden sollten Gerätegruppen proprietäre gemeinschaftliche Rundrufdienste außerhalb dieses Bereiches ansiedeln.

In den Kästen unten sind die aktuellen Rundrufdienste aufgeführt. Beginnend mit dem Name und der Signatur folgt darunter die Beschreibung des Dienstes und die Belegung der Dienst-Bytes. Jedes Bytes identifiziert sich durch ein eindeutiges Kürzel, dessen Bedeutung der angeschlossenen Liste entnommen werden kann. Darin ist ebenfalls der Wertebereich angegeben und, insofern vorhanden, die Einheit in eckigen Klammern. Ein Doppelkreuz als Einheit bedeutet: Es ist ein nomineller Wert ohne spezifische Einheit. Mehrbytewerte führen in ihrem Kürzel h, l oder einen Zahlenwert. In der Erklärungsliste tauchen sie zusammengefasst mit n oder x auf. Es gibt Einträge, die zusätzlich die Wertausprägung im Fehlerfall angeben.

UniversalTime	**E00010h:7:WO**

Beschreibung: Rundmitteilung der Weltzeit als UTC. Der Rundruf findet einmal, ca. zu jeder vollen Stunde statt.

Belegung:

0	1	2	3	4	5	6
YYh	YYl	MON	DAY	HOR	MIN	SEC

YYx	Jahr: 1700 .. 9000
MON	Monat: 1 ..12
DAY	Tag: 1 ..31
HOR	Stunde: 0 ..23 [h]
MIN	Minute: 0 ..59 [min]
SEC	Sekunde: 0 ..59 [s]

LocalTime	**E00020h:7:WO**

Beschreibung: Rundmitteilung der Ortszeit. Der Rundruf findet einmal, ca. zu jeder vollen Stunde statt.

Belegung:

0	1	2	3	4	5	6
YYh	YYl	MON	DAY	HOR	MIN	SEC

YYx	Jahr: 1700 .. 9000
MON	Monat: 1 ..12
DAY	Tag: 1 ..31
HOR	Stunde: 0 ..23 [h]
MIN	Minute: 0 ..59 [min]
SEC	Sekunde: 0 ..59 [s]

AmbientTemperature — E00030h:1:WO

Beschreibung: Rundmitteilung der Außentemperatur. Der Wert wird übertragen, sobald er sich ändert, jedoch nicht häufiger als im 5-Minuten-Intervall.

Belegung:

0
TMP

TMP	Aktuelle Außentemperatur: -39 .. +60 [°C] (Fehler: -128)

AbsoluteAirPressure — E00040h:2:WO

Beschreibung: Rundmitteilung des absoluten Umgebungsluftdrucks. Der Wert wird übertragen, sobald er sich ändert, jedoch nicht häufiger als im 5-Minuten-Intervall.

Belegung:

0	1
APh	APl

APx	Aktueller Luftdruck: 300 .. 1100 [hPA] (Fehler: -32768)

RelativeAirPressure	**E00050h:2:WO**

Beschreibung: Rundmitteilung des relativen Umgebungsluftdrucks. Der Wert wird übertragen, sobald er sich ändert, jedoch nicht häufiger als im 5-Minuten-Intervall.

Belegung:

0	1
RPh	RPl

RPx	Aktueller Luftdruck: 300 .. 1100 [hPA] (Fehler: -32768)

AmbientBrightness	**E00060h:1:WO**

Beschreibung: Rundmitteilung der Umgebungshelligkeit. Der Wert wird übertragen, sobald er sich ändert, jedoch nicht häufiger als im 5-Minuten-Intervall.

Belegung:

0
BRT

BRT	Aktuelle Umgebungshelligkeit: 0 .. 100 [%] (Fehler: -128)

TextFeed	**E00100h:51:WO**

Beschreibung: Allgemeiner Rundmitteilungstext. Dieser Rundrufdienst ermöglicht, allgemeine Rundmitteilungstexte an alle Busstationen zu verteilen. Die üblicherweise UTF-8-kodierten Texte können eine Größe von bis zu 50 Bytes plus abschließendem Nullbyte haben. Die weitere Verarbeitung der Textmitteilungen ist nicht festgelegt. Entsprechende Busstationen können sie ausgeben, anzeigen oder aufzeichnen.

Belegung:

0	1	…	49	50
T01	T02	…	T50	0

TXX	Byte der Rundmitteilung: 0 .. 255

Anhang C CRC Ermittlung

Um die Korrektheit der Bytes in den Datagrammen zu gewährleisten werden sie mit einem 8-Bit CRC abgesichert. Das zugrundeliegende Polynom lautet $x^8+x^5+x^4+x^0$, auch bekannt als DOW-CRC (Dallas-One-Wire). Der CRC-Wert kann entweder berechnet oder aus einer Nachschlagetabelle abgelesen werden.

1. Berechnung:

```
void
UpdateCRC(UInt08 data, UInt08* crc)
{
    *crc ^= data;
    for (UInt08 i=0; i<8; i++) *crc = (*crc >> 1) ^ ((*crc & 1) ? 0x8C : 0 );
}
```

2. Nachschlagetabelle:

```
static void
UpdateCRC(UInt08 data, UInt08* crc)
{
    static const UInt08 table[] =
    {   0x00, 0x5E, 0xBC, 0xE2, 0x61, 0x3F, 0xDD, 0x83, 0xC2, 0x9C, 0x7E, 0x20, 0xA3,
        0xFD, 0x1F, 0x41, 0x9D, 0xC3, 0x21, 0x7F, 0xFC, 0xA2, 0x40, 0x1E, 0x5F, 0x01,
        0xE3, 0xBD, 0x3E, 0x60, 0x82, 0xDC, 0x23, 0x7D, 0x9F, 0xC1, 0x42, 0x1C, 0xFE,
        0xA0, 0xE1, 0xBF, 0x5D, 0x03, 0x80, 0xDE, 0x3C, 0x62, 0xBE, 0xE0, 0x02, 0x5C,
        0xDF, 0x81, 0x63, 0x3D, 0x7C, 0x22, 0xC0, 0x9E, 0x1D, 0x43, 0xA1, 0xFF, 0x46,
        0x18, 0xFA, 0xA4, 0x27, 0x79, 0x9B, 0xC5, 0x84, 0xDA, 0x38, 0x66, 0xE5, 0xBB,
        0x59, 0x07, 0xDB, 0x85, 0x67, 0x39, 0xBA, 0xE4, 0x06, 0x58, 0x19, 0x47, 0xA5,
        0xFB, 0x78, 0x26, 0xC4, 0x9A, 0x65, 0x3B, 0xD9, 0x87, 0x04, 0x5A, 0xB8, 0xE6,
        0xA7, 0xF9, 0x1B, 0x45, 0xC6, 0x98, 0x7A, 0x24, 0xF8, 0xA6, 0x44, 0x1A, 0x99,
        0xC7, 0x25, 0x7B, 0x3A, 0x64, 0x86, 0xD8, 0x5B, 0x05, 0xE7, 0xB9, 0x8C, 0xD2,
        0x30, 0x6E, 0xED, 0xB3, 0x51, 0x0F, 0x4E, 0x10, 0xF2, 0xAC, 0x2F, 0x71, 0x93,
        0xCD, 0x11, 0x4F, 0xAD, 0xF3, 0x70, 0x2E, 0xCC, 0x92, 0xD3, 0x8D, 0x6F, 0x31,
        0xB2, 0xEC, 0x0E, 0x50, 0xAF, 0xF1, 0x13, 0x4D, 0xCE, 0x90, 0x72, 0x2C, 0x6D,
        0x33, 0xD1, 0x8F, 0x0C, 0x52, 0xB0, 0xEE, 0x32, 0x6C, 0x8E, 0xD0, 0x53, 0x0D,
        0xEF, 0xB1, 0xF0, 0xAE, 0x4C, 0x12, 0x91, 0xCF, 0x2D, 0x73, 0xCA, 0x94, 0x76,
        0x28, 0xAB, 0xF5, 0x17, 0x49, 0x08, 0x56, 0xB4, 0xEA, 0x69, 0x37, 0xD5, 0x8B,
        0x57, 0x09, 0xEB, 0xB5, 0x36, 0x68, 0x8A, 0xD4, 0x95, 0xCB, 0x29, 0x77, 0xF4,
        0xAA, 0x48, 0x16, 0xE9, 0xB7, 0x55, 0x0B, 0x88, 0xD6, 0x34, 0x6A, 0x2B, 0x75,
        0x97, 0xC9, 0x4A, 0x14, 0xF6, 0xA8, 0x74, 0x2A, 0xC8, 0x96, 0x15, 0x4B, 0xA9,
        0xF7, 0xB6, 0xE8, 0x0A, 0x54, 0xD7, 0x89, 0x6B, 0x35
    };

    *crc = table[data^(*crc)];
}
```

Die Berechnungsmethode ist um den Faktor 4 langsamer, hat im Gegensatz dazu aber nur den halben Speicherverbrauch.

Ermittlung des CRCs am Beispiel mit 11 Werten:

```
UInt08 data[] = { 68, 187, 249, 124, 54, 204, 156, 88, 123, 254, 95 };
UInt08 crc = 255;
for (UInt08 i=0; i<sizeof(data); i++)
{   UpdateCRC(data[i], &crc);
}
// crc == 42
```

Startwert ist 255, das Ergebnis lautet: 42.

Übrigens: Bezieht man den ermittelten CRC als finalen Wert in die Berechnung mit ein, ergibt das stets den Wert 0.

Anhang D B00120 Cy4Nut

Abbildung 32 zeigt das Cy4Nut-Board. Es trägt den Namen aufgrund seiner Ausmaße von nur 25×17×5 mm, vergleichbar einer Walnuss. Das Gerät ist für keine spezifische Aufgabe vorgesehen, sondern kann für unterschiedliche Zwecke eingesetzt werden. Unter anderem dient das Board als Grundlage für die Referenzimplementierung. Es stehen zwei LEDs zur Verfügung, zusätzlich zur ISP-Schnittstelle sind vier GPIOs herausgeführt. Zwei davon können als ADC, zwei als PWM geschaltet werden. Die minimalistische Ausstattung ist seiner Größe geschuldet. So besitzt das Gerät keinen Spannungsregler, sondern wird direkt an 5V betrieben.

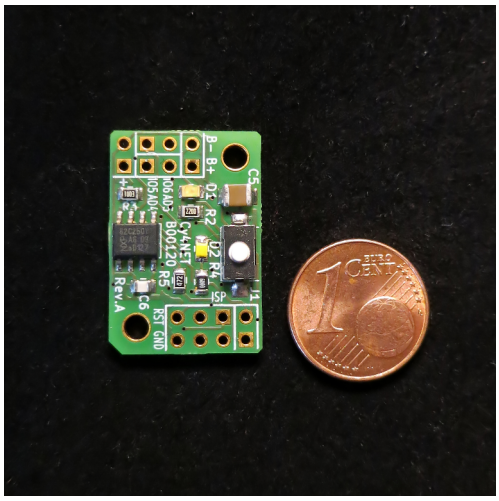


Abbildung 32: Cy4Nut-Board

Das Projekt, Schaltplan und PCB stehen in [25] zur Verfügung. Ein passendes Gehäuse ist als Step-Modell verfügbar.

Es folgen Stückliste, Schaltplan in Abbildung 33 und Bestückungsplan in Abbildung 34.

Referenz	Wert	Bauform/Typ
C1, C2	100nF	SMD Capacity 0805
C3, C4	22pF	SMD Capacity 0805
C6	33pF	SMD Capacity 0805
C5	10µF	SMD Capacity 0805
R2, R4	330	SMD Resistor 0805
R1	33K	SMD Resistor 0805
R3	100K	SMD Resistor 0805
R5	4K7	SMD Resistor 0805
D1, D2	LED	SMD Pol 0805
U1	PCA82C251	SMD SO8E
X1	11.0592 MHz	SMD Crystal 3.2x5 mm
TA1	Reset Key	SMD Pushbutton 6x4mm
IC1	ATmega328P	SMD TQFP-32
H2, H3	Pin-Header	THT Pin 2x1
H1	Pin-Header	THT Pin 3x2
J1	Jumper	THT Pin 2x1
P1, P2	Pin-Header/Connection field	THT Pin 2x1

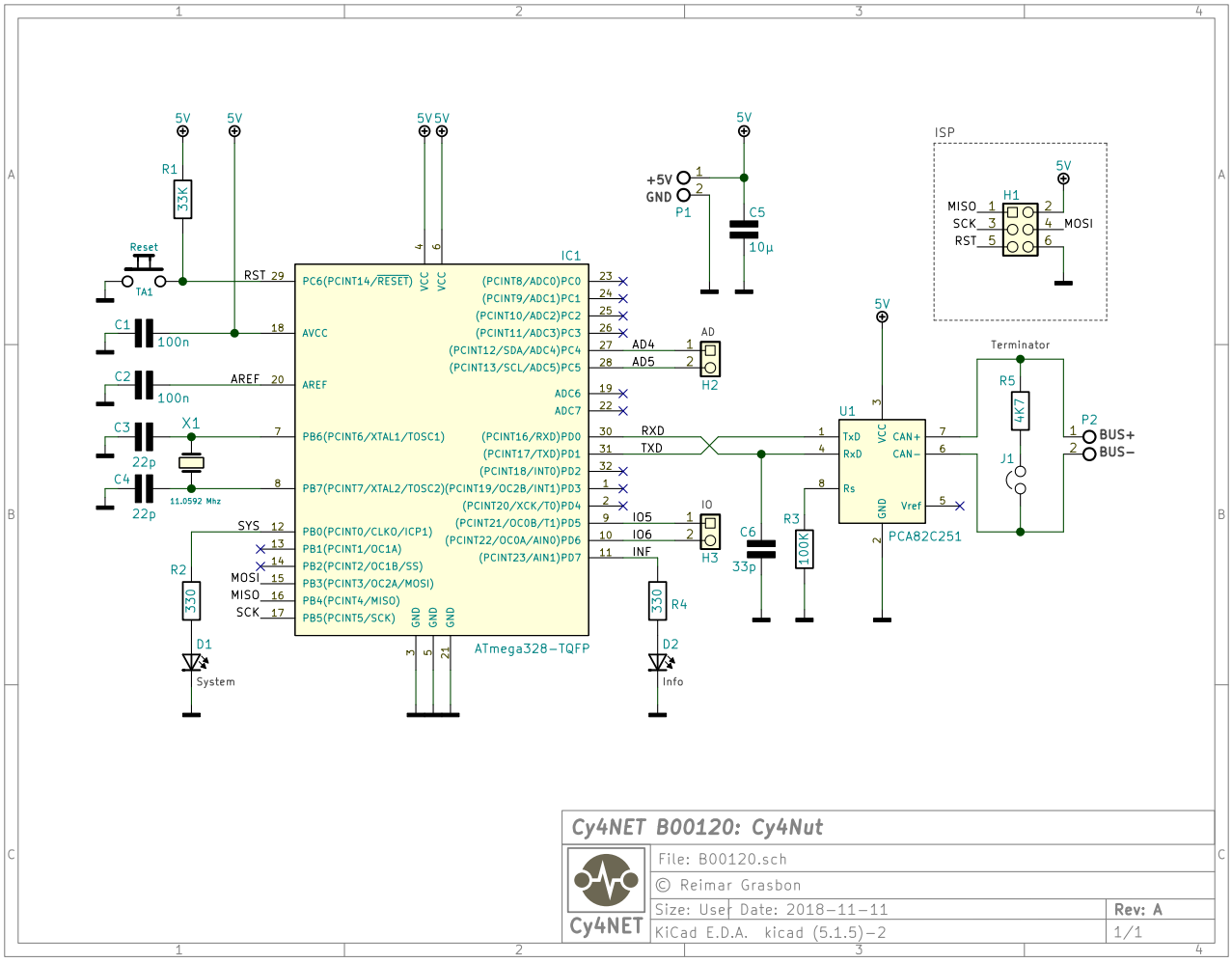


Abbildung 33: B00120 Schaltplan

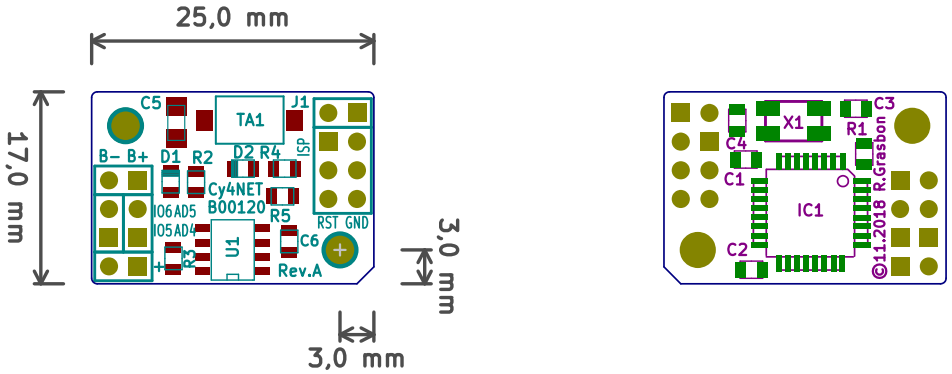


Abbildung 34: B00120 Leiterplatte, Bestückung Ober- und Unterseite

Anhang E B00023 Cy4-To-Serial Gateway

B00023 ermöglicht einen äußeren Zugang zum Cy4NET. Es ist ein Gateway. Als solches verfügt es neben der Busankopplung zusätzlich über eine serielle Verbindung, mittels derer das Gateway einem Rechner, z.B. einem PC, den Zugang verschafft. Kapitel 11.1 geht auf das Protokoll der Verbindung ein. Abbildung 35 zeigt die 3D-Ansicht des PCBs und das real bestückte B00023.

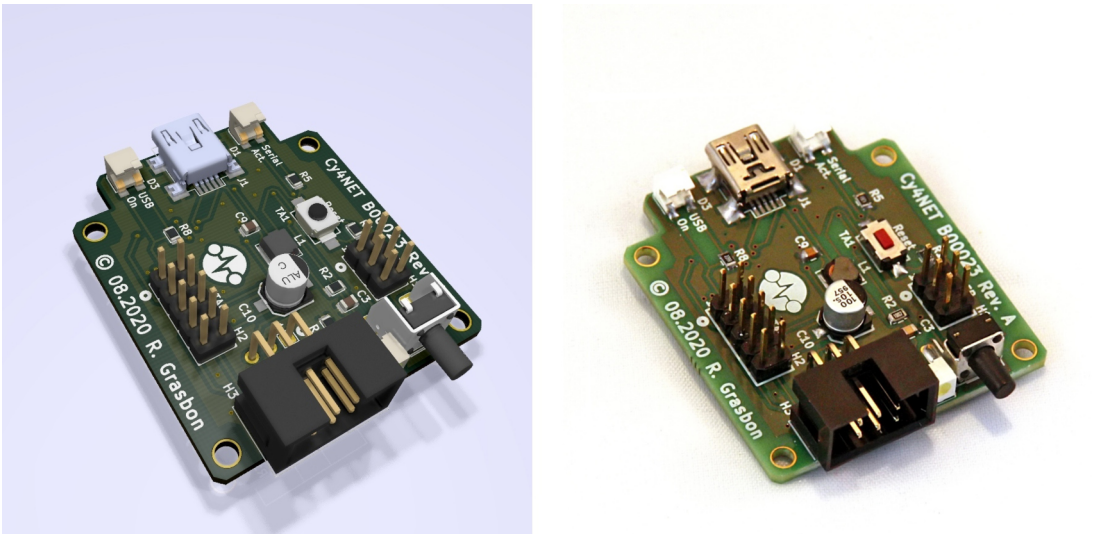


Abbildung 35: CAD-Modell und real bestücktes B00023

Da native serielle Schnittstellen bei heutigen Rechnern häufig nicht mehr zur Verfügung stehen, wird bei B00023 die serielle Verbindung mittels eines USB-zu-seriell-Konverters hergestellt. Angeschlossen wird das Gerät über eine Mini-USB-B-Buchse.

Herzstück ist neben dem USB-Konverter FT232RL ein ATmega644PA. Dabei ist der Suffix des Controllers von Bedeutung, also P oder PA. Es gibt auch ein Derivate namens ATmega644 ohne die Endung, doch im Vergleich zum P(A)-Typ fehlt diesem die, für diesen Zweck unabdingbare, zweite UART. Es ist also darauf zu achten, die richtige Variante zu wählen. Grundsätzlich hätte als Kernstück für das Gateway auch ein ATmega328PB dienen können, auch er verfügt über mehrere UARTs. Jedoch ist dieser Typ in der Bauform TQFP-32 schwerer erhältlich.

Es folgen der Schalt- und Bestückungsplan in Abbildung 36 und 37, gefolgt von der Stückliste. Das PCB findet sich als Projekt der quelloffenen CAD-Software KiCad zum herunterladen in [26].

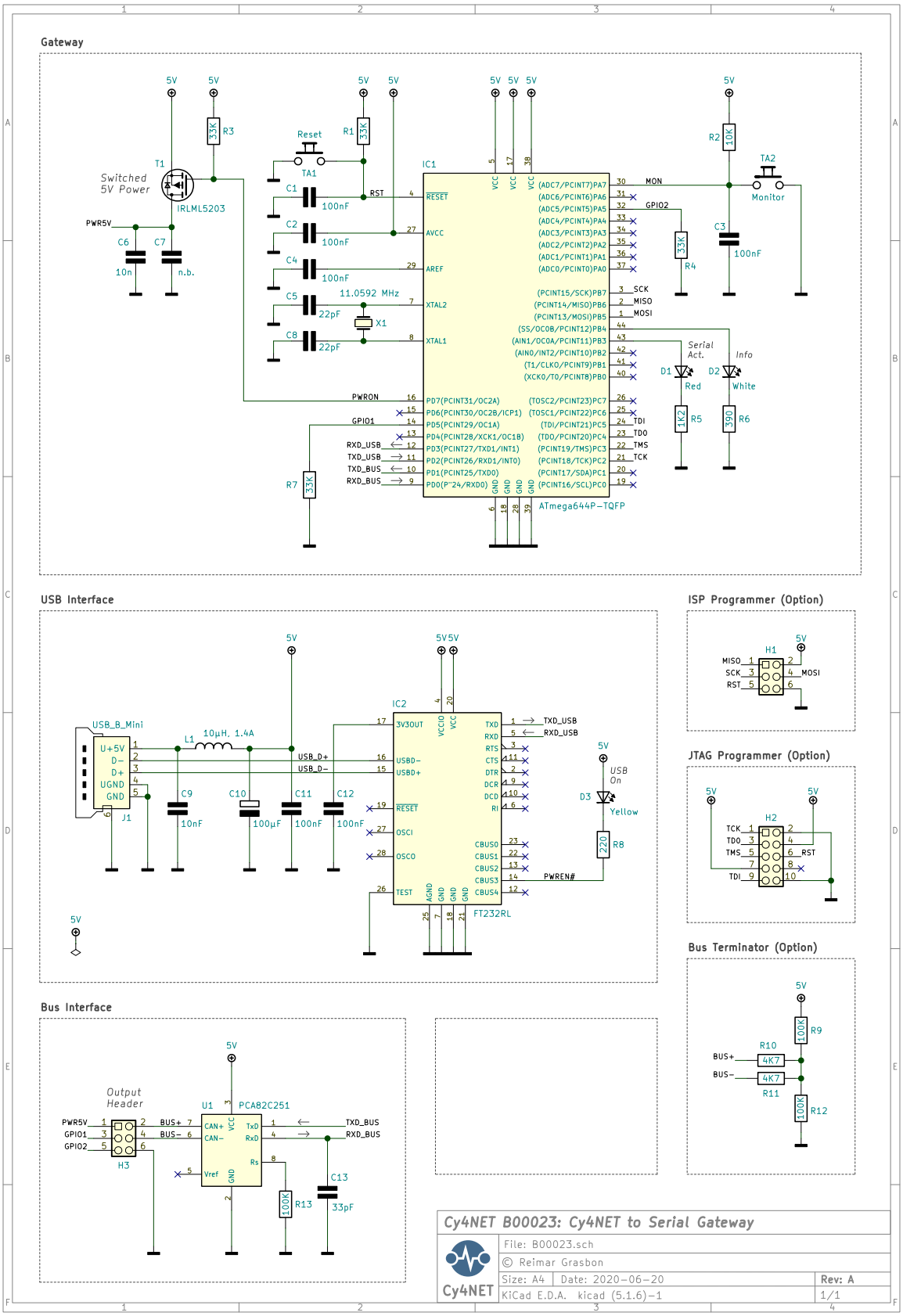


Abbildung 36: B00023 Schaltplan

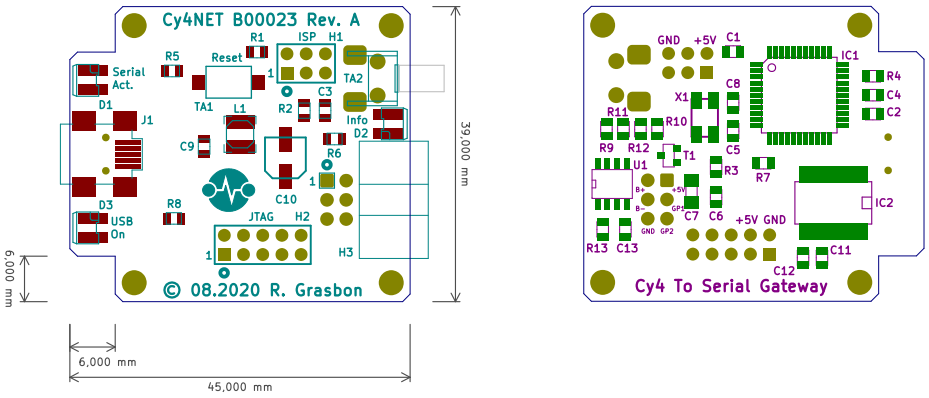


Abbildung 37: B00023 Leiterplatte, Bestückung Ober- und Unterseite

Stückliste:

Referenz	Wert	Bauform/Typ
C1, C2, C3, C4, C11, C12	100nF	SMD Capacity 0805
C5, C8	22pF	SMD Capacity 0805
C6, C9	10nF	SMD Capacity 0805
C7	n.c.	
C10	100µF	SMD Elko 5.3x5.3mm
C13	33pF	SMD Capacity 0805
D1	LED red	SMD LED 4242 SideView
D2	LED White	SMD LED 4242 SideView
D3	LED Yellow	SMD LED 4242 SideView
H1	Pin-Header	THT Pin 3x2
H2	Pin-Header	THT Pin 5x2
H3	Pin-Header	THT Case Pin 3x2 horizontal
IC1	ATmega644P	SMD TQFP-44
IC2	FT232RL	SMD SSOP-28
J1	USB Connector	SMD USB Mini-B
L1	10 µH, 1.4A	SMD Inductor 4.2x4.2 mm
R1, R3, R4, R7	33K	SMD Resistor 0805
R2	10K	SMD Resistor 0805
R5	1K2	SMD Resistor 0805
R6	390	SMD Resistor 0805
R8	220	SMD Resistor 0805
R9, R12, R13	100K	SMD Resistor 0805

R10, R11	4K7	SMD Resistor 0805
T1	IRLML5203	SMD SOT23
TA1	Reset Key	SMD Pushbutton 6x4 mm
TA2	Function Key	THT Pushbutton 6x6 mm horizontal
U1	PCA82C251	SMD SO8E
X1	11.0592 MHz	SMD Crystal 3.2x5 mm

Die Kontaktierung zum Bus geschieht mittels H3, einem um 90° gewinkelten 6-poligen Wannenstecker im Rastermaß 2,54 mm. Wie in Abbildung 38 zu sehen, können die Kontakte mittels passender Buchse abgegriffen werden. Zusätzlich zu den Busleitungen und GND, ist die 5V USB-Spannung herausgeführ99t (schaltbar mittels T1), sowie zwei frei nutzbare GPIOs. Bei Nutzung der herausgeführten 5V ist zu beachten, dass der Controller ebenfalls mit der USB-Spannung betrieben wird. Daher sollten man darauf verzichten mit der USB-Spannung größere externe Lasten zu versorgen. Auch ein nur kurzer Spannungseinbruch im Einschaltmoment externer Verbraucher versetzt den Controller in einen unkontrollierten Zustand, bestenfalls verursacht es einen Brown-Out-Reset.

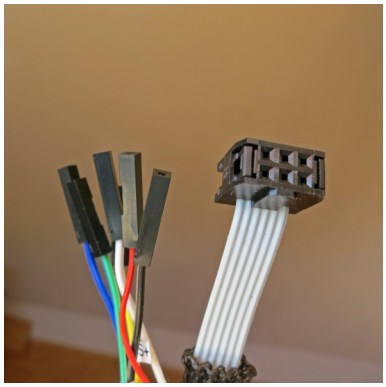


Abbildung 38: Kontaktierung

Wie bei B00120 gibt auch für dieses Projekt wieder ein passendes Gehäuse, dass mittels 3D-Druck hergestellt werden kann. Abbildung 39 zeigt das Objekt, die entsprechenden Step-Dateien stehen ebenfalls zur Verfügung.

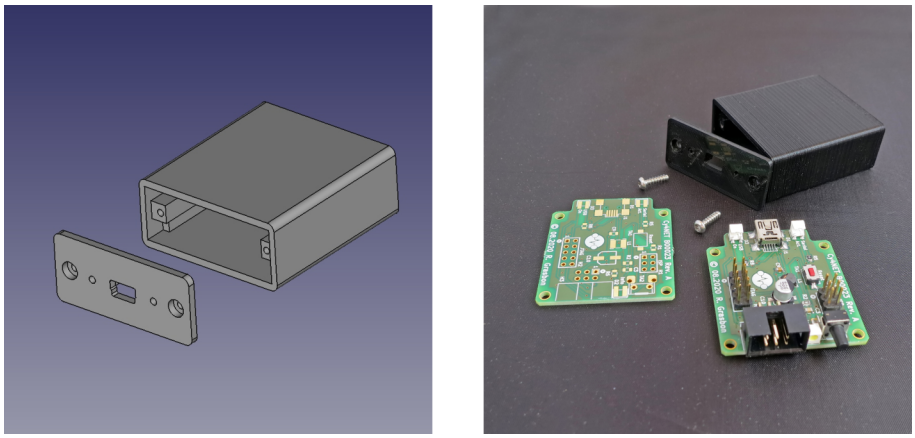


Abbildung 39: B00023 Gehäuse

Anhang F Dienste der Gateway-Anwendung A00111.

Anwendung A00111 ist die Standardfirmware für B00023. In Version 1.0 stellt sie folgende Dienste zur Verfügung:

OperationMode	**350000h:2:RW**

Beschreibung: Legt den Betriebsmodus des Gerätes fest.

Belegung:

0	1
MOD	SAV

MOD	Betriebsmodus: • 0 = Gateway • 1 = Monitor • sonst = Keine Funktion
SAV	Betriebsmodus bei Systemstart: • 0 = Gateway (Voreinstellung) • 1 = Monitor • sonst = Keine Funktion

MonitorOptions	**350020h:3:RW**

Beschreibung: Optionen des Monitorbetriebs. Die Einstellungen werden gespeichert.

Belegung:

0	1	2
RAW	PLT	NLT

RAW	Ausgabeform: • 0 = Aufbereitete Ausgabe (Voreinstellung) • 1 = Rohwertausgabe (Datenbytes der PCUs werden direkt als Hex-Zahlen ausgegeben) • sonst = Keine Funktion
PLT	Gekürzte Nutzlastausgabe. Bei mehr als 6 Nutzlastbytes Ausgabe mittels Auslassungspunkte kürzen: • 0 = Aus (Voreinstellung) • 1 = Ein • sonst = Keine Funktion
NLT	Zeilenwechsel: • 0 = Wagenrücklauf + Zeilenvorschub, CR + LF (Windows) (Voreinstellung), • 1 = Wagenrücklauf, CR (Mac), • 2 = Zeilenvorschub, LF (Linux/Unix), • sonst = Keine Funktion.

Function KeyTask	**350100h:1:RW**

Beschreibung: Legt die Aufgabe der Funktionstaste im Gatewaybetrieb fest. Die Einstellung wird gespeichert. Befindet sich das Gerät im Monitorbetrieb, bewirkt ein Druck auf die Funktionstaste, unabhängig von der hier eingestellten Aufgabe stets eine Rückkehr zum Gatewaybetrieb.

Belegung:

0
FKT

FKT	Tastenfunktion: • 0 = Keine. • 1 = Taste wechselt zwischen Gateway- und Monitorbetrieb (Voreinstellung). • 2 = Schaltet GPIO1 ein/aus (nur wenn GPIO1 als Ausgang konfiguriert ist). • 3 = Schaltet GPIO2 ein/aus (nur wenn GPIO2 als Ausgang konfiguriert ist). • sonst = Wirkungslos.

InfoLedSignaling	**350200h:2:RW**

Beschreibung: Legt die Aufgabe der Info-LED für den Gateway- und den Monitorbetrieb fest. Die getroffene Auswahl wird gespeichert. Folgende Lichtsignale können eingestellt werden:

- 0 = Kein Signal, ausgeschaltet
- 1 = Leuchtet durchgehend
- 2 = Blinkt
- 3 = Herzschlag-Signal
- 4 = Zeigt Aktivität auf dem Bus
- 5 = Leuchtet wenn die USB-Spannung durchgeschaltet ist
- 6 = Zeigt den Zustand von GPIO1
- 7 = Zeigt den Zustand von GPIO2
- sonst = wirkungslos

Belegung:

0	1
ISG	ISM

ISG	Lichtsignal im Gatewaybetrieb, Werte siehe oben, Voreingestellt ist 3.
ISM	Lichtsignal im Monitorbetrieb, Werte siehe oben, Voreingestellt ist 1.

USBPower	**350300h:1:RW**

Beschreibung: Bestimmt, ob das 5V USB-Testspannungssignal nach Systemstart an Pin 1 der Steckerleiste von H3 durchgeschaltet ist.

Belegung:

0
UPS

UPS	USB-Spannung: • 0 = USB-Testspannung liegt nach Systemstart nicht an P1 an (Voreinstellung). • 1 = USB-Testspannung liegt nach Systemstart an P1 an. • sonst = Ohne Funktion

GpioConfiguration	**350400h:4:RW**

Beschreibung: Konfiguriert die GPIOs. Folgende Möglichkeiten stehen zur Auswahl:

- 'T' = 84 = 54h : Eingabe, Tristate
- 'P' = 80 = 50h : Eingabe, Pullup
- 'H' = 72 = 48h : Ausgabe, High
- 'L' = 76 = 4Ch : Ausgabe, Low

Belegung:

0	1	2	3
GA1	GA2	SA1	SA2

GA1	Konfiguration GPIO1: Werte siehe oben.
GA2	Konfiguration GPIO2: Werte siehe oben.
SA1	Konfiguration GPIO1 nach Systemstart: Werte wie bei GA1, voreingestellt ist 'T'.
SA2	Konfiguration GPIO2 nach Systemstart: Werte wie bei GA2, voreingestellt ist 'T'.

DeviceLabel	**FFFF00h:61:RW**

Beschreibung: Benutzerdefinierte Gerätebezeichnung.

Belegung:

0	1	2	…	59	60
N01	N02	N03	…	N60	0

Nxx	Zeichen der allgemeinen Gerätebezeichnung: 0 .. 255 (ASCII/UTF-8)

FactorySettings	**FFFF40h:1:WO**

Beschreibung: Zurückstellen aller Einstellungen auf den Ursprungszustand.

Belegung:

0
FST

FST	Auslöser zur Wiederherstellung der Werksvoreinstellungen: 94, (5Eh, '^').

CallTypePlate	**FFFFF0h:7:RO**

Beschreibung: Geräte-Typenschild erfragen (Pflichtdienst).

Belegung:

0	1	2	3	4	5	6
AT2	AT1	AT0	AVE	BT2	BT1	BT0

ATx	Anwendungstypennummer: Axxxxxh, xxxxx in BCD: 00001 .. 99999
AVE	Anwendungsversion: 01 .. 99, Haupt/Unternummer in BCD (Bsp: 12h = V1.2)
BTx	Boardtypennummer: Bxxxxxh, xxxxx in BCD: 00001 .. 99999

EnterApplication	**FFFFF8h:1:WO**

Beschreibung: Startet die Anwendung neu (Pflichtdienst).

Belegung:

0
EAP

EAP	Auslöser zum (Neu-)Start der Applikation (Reset): 94, (5Eh, '^').

EnterProgramLoader	**FFFFFAh:1:WO**

Beschreibung: Startet das System neu und kehrt im Programmladermodul zurück (Pflichtdienst).

Belegung:

0
EPL

EPL	Auslöser zum Eintritt in den Programmlader: 94, (5Eh, '^').

ChangeCy4Address	**FFFFFCh:2:WO**

Beschreibung: Wechsel der Geräteadresse (Pflichtdienst).

Belegung:

0	1
NET	DEV

NET	Neue Netzadresse: 1 .. 255
DEV	Neue Geräteadresse: 1 .. 255

Anhang G B00101 Evaluierungsboard ATmega32 3M

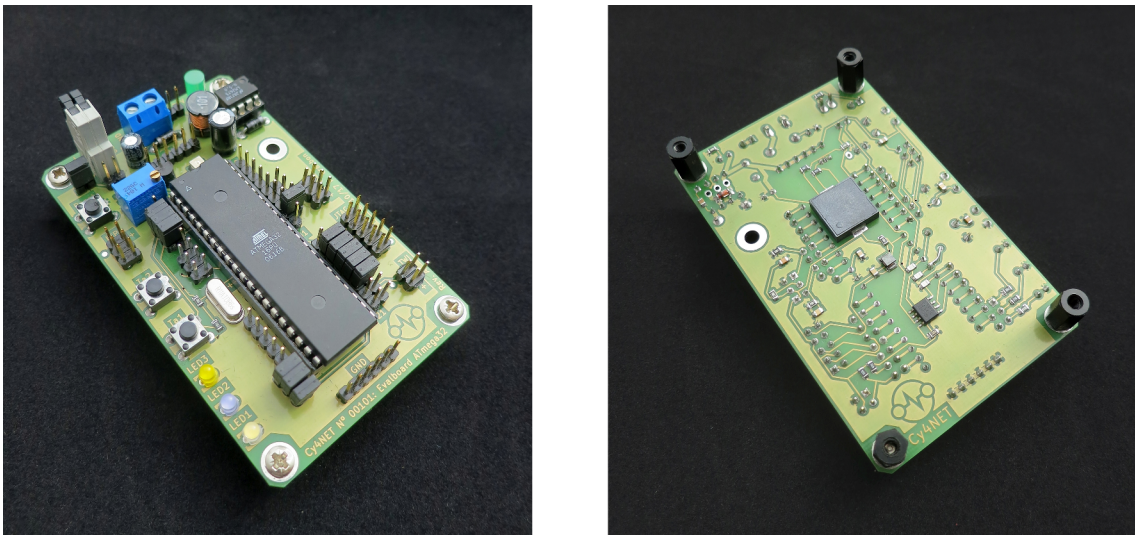


Abbildung 40: B00101 Oberseite und Unterseite

Das Board mit der Typennummer B00101 ist eine allgemeine Test- und Evaluierungsplattform. Abbildung 40 zeigt Ober- und Unterseite der bestückten Leiterplatte. Zum Einsatz kommt ein ATmega32, der mit 3,6864 MHz getaktet ist.

Alle Ein- und Ausgänge sind mittels Jumper konfigurierbar. Das Board bietet einen Piezosignalgeber, zwei allgemein nutzbare Taster, drei LEDs, einen Analogeingang und jeweils einen Pull-Up und Pull-Down Ein/Ausgang. Der TWI-Bus (I²C) ist herausgeführt. Das Board kann an bis zu 28V betrieben werden, Basisspannung beträgt 5V.

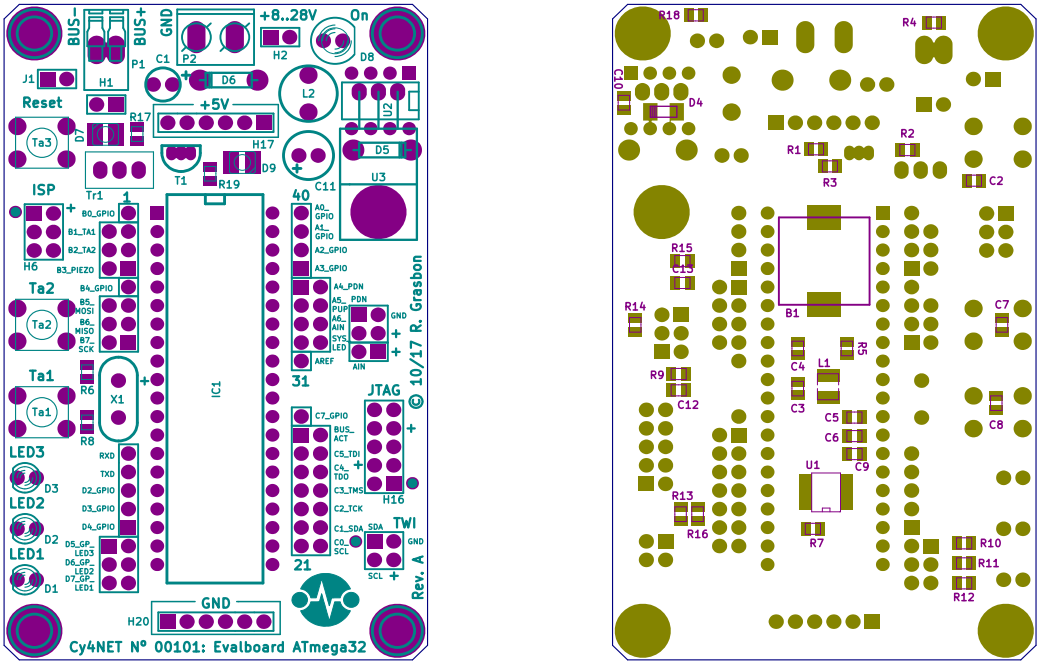


Abbildung 41: Leiterplatte, Bestückung Ober- und Unterseite

Wie B00120 oder B00023 steht auch dieses Projekt zum Herunterladen bereit [27].

Die Bestückung der Leiterplatte ist Abbildung 41 zu entnehmen, die Darstellung ist maßstabsgetreu. Es folgen die Stückliste und in Abbildung 42 der Schaltplan.

Referenz	Wert	Bauform/Typ
B1	Piezo Buzzer	P12A03
C1	10µF	THT Elcap Vert RM 5mm
C2, C3, C4, C7, C8, C12, C13	100nF	SMD Capacity 0805
C5, C6	22pF	SMD Capacity 0805
C9	33pF	SMD Capacity 0805
C10 (Option 1)	10nF	SMD Capacity 0805
C11 (Option 1)	100µF	THT Elcap RM 7
D1, D2, D3	LED	THT LED 3mm
D4 (Option 2)	1N4148	SMD Diode-3.5x1.5mm
D5 (Option 1)	LL5819	THT Diode RM 3
D6	1N4001	THT Diode RM 3
D7, D9	LED	SMD PLCC2
D8	LED	THT LED 5mm
H1, H7	Pin-Header	THT Pin 2x1
H2	Pin-Header	THT Pin 2x1
H3, H6, H13, H14	Pin-Header	THT Pin 3x2

H4, H9, H10, H11	Pin-Header	THT Pin 1
H5	Pin-Header	THT Pin 5x1
H8	Pin-Header	THT Pin 4x1
H12	Pin-Header	THT Pin 4x2
H15	Pin-Header	THT Pin 7x2
H16	Pin-Header	THT Pin 5x2
H17	Pin-Header	THT Pin 6x1
H18, H19	Pin-Header	THT Pin 2x2
H20	Pin-Header	THT Pin 6x1
IC1	ATmega32	THT DIP 40
J1	Jumper	THT Pin 2x1
L1	10µH	SMD Inductor 1210
L2 (Option 1)	100µH	THT Inductor 8mm
P1	Terminal	Spring terminal RIA AST021
P2	Terminal	THT Terminal RM 2
R1	22	SMD Resistor 0805
R2, R5, R7	100K	SMD Resistor 0805
R3, R4, R13, R16	4K7	SMD Resistor 0805
R6, R8, R9, R14, R15	10K	SMD Resistor 0805
R10	390	SMD Resistor 0805
R11	270	SMD Resistor 0805
R12	820	SMD Resistor 0805
R17, R18, R 19	220	SMD Resistor 0805
T1	BS170	THT TO92
Ta1, Ta2	TA	THT Pushbutton 6x6mm
Ta3	Reset Key	THT Pushbutton 6x6mm
Tr1	50K	THT TrimmPoti 1
U1	PCA82C251	SMD SO8E
U2 (Option 1)	LM2675-5.0	THT DIP 8
U3 (Option 2)	7805	THT TO220L
X1	3.6864MHz	THT Crystal HC49

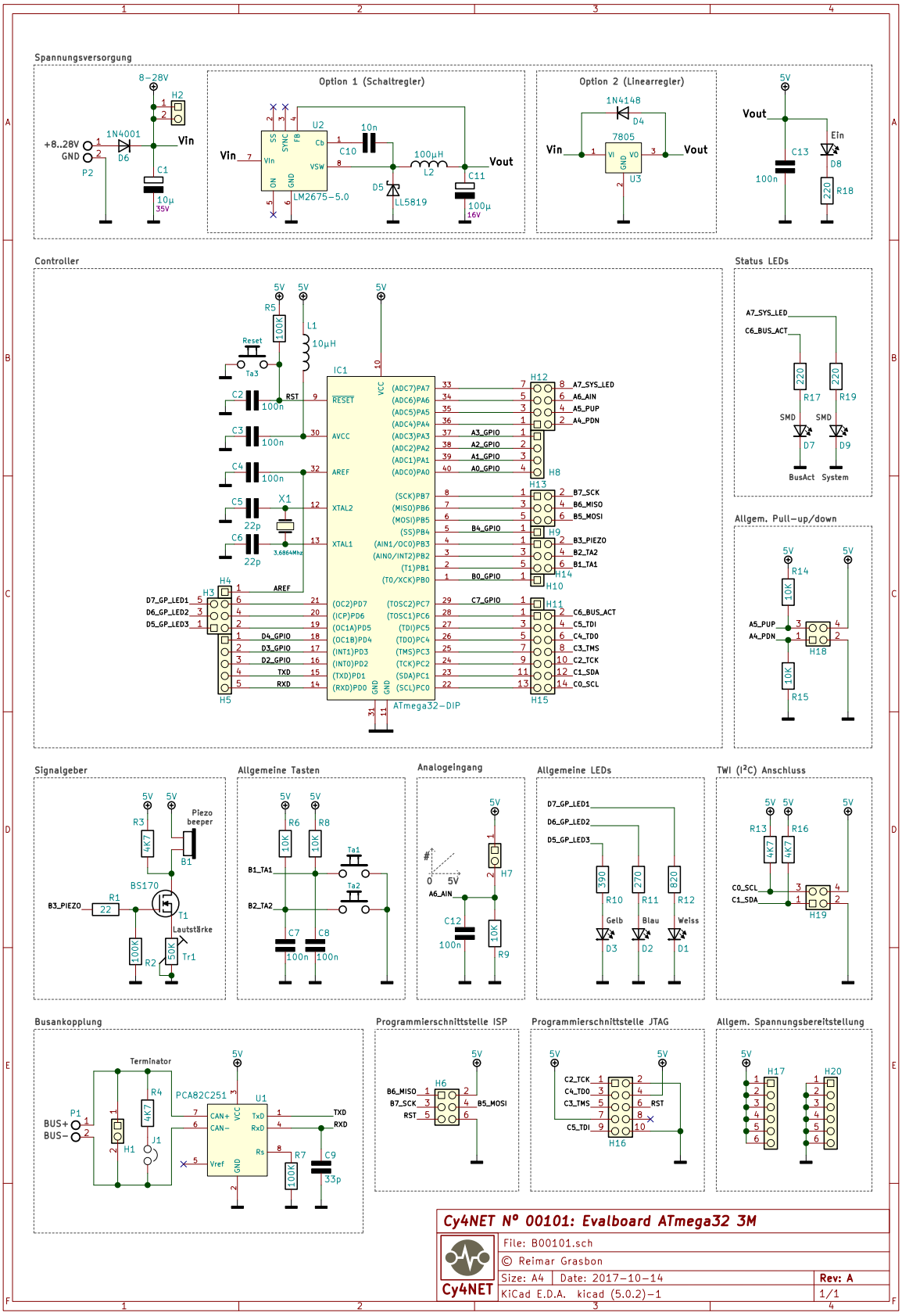


Abbildung 42: B00101 Schaltplan

Konventionen im Cy4NET

1. Cy4NET-Adresse 255.255 ist allgemein die Ausprägung für eine ungültige Adresse.
2. Wenn kein explizites Adressmanagement vorhanden ist, sollte die werksvoreingestellte Cy4NET-Addresse eines Gerätes 255.1xx lauten. Die xx der Geräteadresse (DEV) werden durch die beiden letzten Ziffern der Boardtypennummer bestimmt. Auf diese Weise erhalten neu in Betrieb genommene Geräte Werksadressen aus dem Bereich von 255.100 bis 255.199.
3. Cy4NET-Netzaddresse 255, also der Adressbereich von 255.1 bis 255.254, ist als *Landing Page* für neue oder zurückgesetzte Geräte vorgesehen. Um einer Kollision mit neuen oder weiteren Geräten entgegenzuwirken, sollten Geräte, deren Cy4NET-Addressen sich in diesem Bereich befinden, im Nachgang auf eine andere Adresse umplatziert werden. Siehe dazu auch Konvention 2.
4. Dienstort FFFFFFh ist reservierte Wertausprägung für einen ungültigen Dienstort.
5. Der allgemeine Wert für das Auslösen einfacher Ereignisse ist 94 bzw.5Eh.
6. Die Byte-Reihenfolge bei Mehrbytewerten ist Big-Endian, d.h. höherwertige Bytes kommt als Erstes.
7. Programmlader haben eine Applikationstypennummer im Bereich von 1 bis 99. Hat eine Applikation eine Typennummer unter 100, wird sie als Programmlader behandelt.
8. Typnummern sind öffentlich. Jedoch gibt es Cy4NET-Entwicklungen, die nicht öffentlich zugänglich gemacht werden sollen. Um Doppelbelegungen bei der Typennummerierung zu vermeiden, sollten Applikationen und Boards in einem solchen Fall ihre Typnummern aus dem Bereich von 90000 bis 99999 wählen. Es ist die private Zone der Typnummern.

Literatur und Bezugsquellen

1 Standard RS232 der seriellen Schnittstelle: de.wikipedia.org/wiki/RS-232

2 Carrier Sense Multiple Access Verfahren (CSMA): de.wikipedia.org/wiki/Carrier_Sense_Multiple_Access

3 Open Systems Interconnection model (OSI) für Netzwerkübertragungen: de.wikipedia.org/wiki/OSI-Modell

4 CAN Spezifikation, Robert Bosch GmbH: web.archive.org/web/20161213073159/http://www.bosch-semiconductors.de:80/media/ubk_semiconductors/pdf_1/ipmodules_1/can_fd/icc13_2012_paper_Hartwich.pdf

5 Modbus Protokoll: www.modbus.org

6 Datensätze der Netzwerktechnik, die Protokoll-Data-Unit: de.wikipedia.org/wiki/Service_Data_Unit

7 I²C Protokoll: de.wikipedia.org/wiki/I%C2%B2C

8 Die Sache mit der Funkzeit: c't Hacks / Make 02/2014, S. 84: www.heise.de/select/make/archiv/2014/2/seite-84

9 Mealy-Automat: de.wikipedia.org/wiki/Mealy-Automat

10 Microchip®, Hersteller der AVR®-MCUs: www.microchip.com/design-centers/8-bit

11 Anwendung A00272, die Referenzimplementierung des Cy4NET-Systems: cy4net.org/Doku/toCy4ReferenceAppl.html

12 Entwicklungsumgebung Microchip-Studio, V7: www.microchip.com/en-us/development-tools-tools-and-software/microchip-studio-for-avr-and-sam-devices

13 AVR®-Toolchain: www.microchip.com/mplab/avr-support/avr-and-arm-toolchains-c-compilers

14 Leistungsaufnahme und CoreMark-Vergleich des Raspberry Pi® 3: www.heise.de/ct/artikel/Raspberry-Pi-3-Leistungsaufnahme-und-CoreMark-Vergleich-3121139.html

15 Das Message Queuing Telemetry Transport Protokoll (MQTT): mqtt.org

16 Gateway-Anwendung A00111: cy4net.org/Doku/toCy4GatewayAppl.html

17 Powerstation: cy4net.org/Doku/toCy4PowerstationBrd.html

18 Kommandozeilenwerkzeug `cy4cmd`: cy4net.org/Doku/toCy4Cmd.html

19 Gleitkommaformat 32-Bit float nach IEEE-754: de.wikipedia.org/wiki/IEEE_754

20 Intel-Hex Format: de.wikipedia.org/wiki/Intel_HEX

21 JTAG Standard: de.wikipedia.org/wiki/Joint_Test_Action_Group

22 ISP-Programmierung: de.wikipedia.org/wiki/In-System-Programmierung

23 AVR-Dude: www.nongnu.org/avrdude/

24 Mikrocontroller-Netzwerk: www.mikrocontroller.net

Literatur und Bezugsquellen

25 Cy4Nut B00120: cy4net.org/Doku/toCy4NutBrd.html

26 Gateway B00023: cy4net.org/Doku/toCy4GatewayBrd.html

27 Evaluierungsboard ATmega32 3M B00101: cy4net.org/Doku/toCy4EvalBrd3MBrd.html