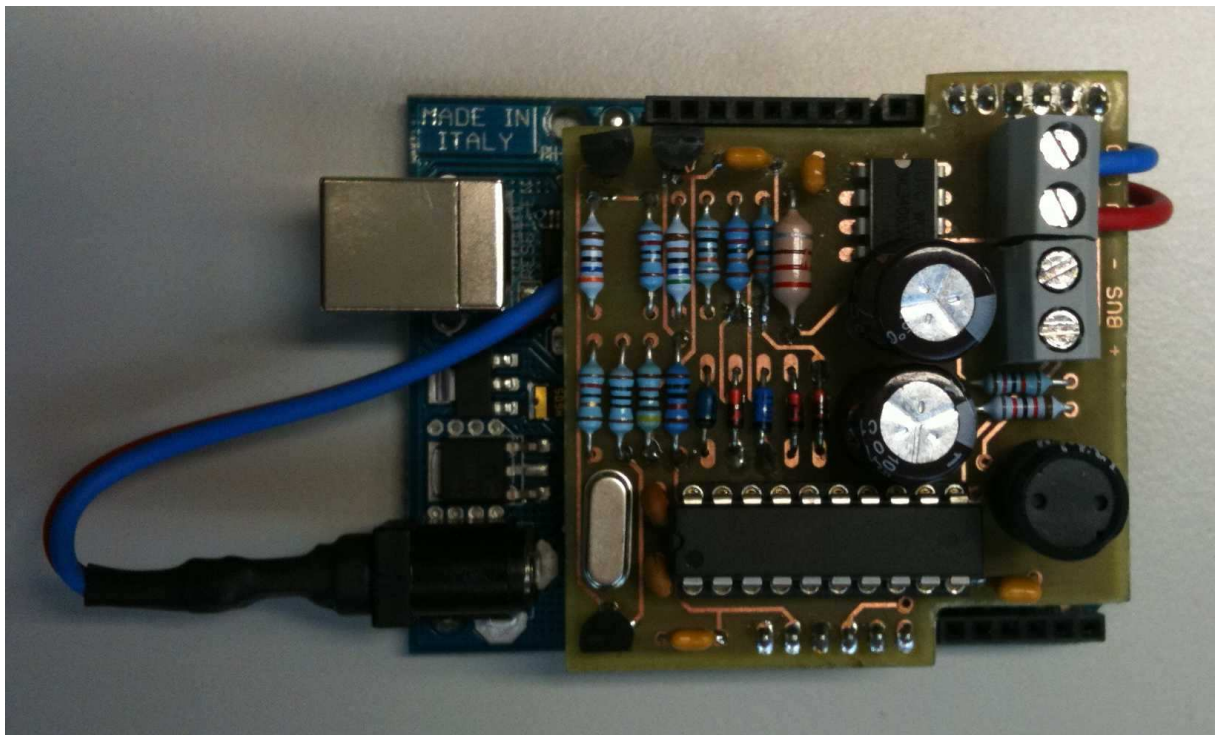


Studienarbeit

EIB-Sensor aus einem Arduino-Board



Andreas Straub

Kurs: **TEL07AT – 5. Semester**

Betreuer: **Prof. Dr. Georg Richter**

Abgabedatum: **15. Jan 2010**

1 Inhaltsverzeichnis

1	Inhaltsverzeichnis	2
2	Einleitung	4
2.1	Vorwort.....	4
2.2	Aufgabenstellung.....	4
3	Arduino – Plattform.....	5
3.1	Hardware.....	5
3.1.1	Arduino Duemilanove	7
3.1.2	Bootloader.....	11
3.2	Software	12
4	EIB	14
4.1	Technik.....	14
4.2	Struktur.....	15
4.2.1	Physikalische Struktur (= Netztopologie)	15
4.2.2	Logische Struktur (frei programmierbar)	16
4.3	Bits und Bytes.....	17
4.3.1	Wie kommunizieren die Teilnehmer untereinander?	17
4.3.2	Was passiert denn nun auf dem Bus?	17
4.3.3	Wie sendet der Mikrocontroller ein Bit?.....	19
4.3.4	Wie sieht denn ein SendeByte aus?	20
4.3.5	Kollisionen vermeiden.....	21
4.4	Telegramm-Aufbau	23
4.4.1	Kontrollbyte	23
4.4.2	Quelladresse	24
4.4.3	Zieladresse.....	24
4.4.4	DRL-Byte	25

4.4.5	Nutzdaten.....	25
4.4.6	Prüfbyte.....	27
5	Buskoppler.....	28
5.1	Aufbau.....	29
5.1.1	Die Schutzstufe:.....	30
5.1.2	Der Schaltregler:.....	30
5.1.3	Die Empfangsstufe:	31
5.1.4	Die Sendestufe:.....	32
5.2	Firmware.....	33
5.2.1	Telegramme senden:	33
5.2.2	Konfiguration:.....	34
5.2.3	Telegramme empfangen:.....	34
5.3	Signalverlauf	35
5.3.1	Der original Signalverlauf.....	35
5.3.2	Der Buskoppler Signalverlauf	36
6	Arduino Sensor.....	37
6.1	Quellcodeauszug:.....	38
7	Alternativen	39
8	Abbildungsverzeichnis.....	40
9	Erklärung	41
10	Quellenverzeichnis.....	42
11	Anhang.....	43

2 Einleitung

2.1 Vorwort

Das Projekt „Aufbau eines EIB-Sensors aus einem Arduino-Board mit einem EIB - Shield“ ist eigentlich für eine Bearbeitungszeit von 6 Monaten ausgelegt. Da meine Bearbeitungszeit für diese Studienarbeit aber nur 3 Monate betrug, setzte ich mir als Ziel, in dieser Zeit, die bestmögliche, umsetzbare Lösung zu finden.

Dank an:

die Herren von Freebus.org, besonders an Herr Andreas Krebs, die mir bei der Lösung der Aufgabenstellung mit Rat und Tat zur Seite gestanden sind.

2.2 Aufgabenstellung

Entwicklung eines EIB – Anschlusses (sog. Shield) an einen Arduino-M μ -Board und Programmierung/Portierung des Kommunikationsprotokolls.

3 Arduino – Plattform

Das Arduino-System ist eine aus Soft- und Hardware bestehende sog. Physical-Computing-Plattform. Beide Komponenten sind im Sinne von Open Source „queltoffen“. Die Hardware besteht aus einem einfachen I/O-Board mit einem Mikrocontroller und analogen und digitalen Ein- und Ausgängen (siehe 3.1).

Die Entwicklungsumgebung beruht auf Processing und Wiring, die insbesondere Künstlern, Designern, Hobbyisten und anderen Interessierten den Zugang zur Programmierung und zu Mikrocontrollern erleichtern soll.

Arduino kann benutzt werden, um eigenständige interaktive Objekte zu steuern oder um mit Softwareanwendungen auf Computern zu interagieren

3.1 Hardware

Die Hardware basiert auf einem Atmel AVR-Mikrocontroller, der über einen FTDI-USB-Seriell-Konverter mit dem PC verbunden werden kann. Bei den älteren Produkten wird ein ATmega168 benutzt, neuerdings, so wie auch hier, wird der ATmega328 eingebaut. Alle Boards werden entweder über USB oder eine externe Stromquelle mit 5-Volt versorgt und verfügen über einen 16 MHz Quarzoszillator. Der Mikrocontroller ist mit einem Boot-Loader vorprogrammiert.

Die Programmierung erfolgt direkt über die USB-Schnittstelle, ein externer Programmierer ist nicht erforderlich.

Konzeptionell werden alle Boards über eine serielle RS-232-Schnittstelle programmiert. Serielle Arduino Boards besitzen einen simplen Inverter Schaltkreis, der die RS-232 Level auf TTL Level übersetzt.

Bei dem hier verwendeten Board geschieht die Umsetzung von USB nach seriell über einen Adapterchip (FTDI FT232).



Abbildung 1: Arduino Duemilanove Board

Die Arduino-Boards stellen die meisten I/O-Pins des Mikrocontrollers zur Nutzung für oder durch andere elektronische Schaltungen zur Verfügung. Die aktuell gängigen Boards bieten 14 digitale I/O-Pins, von denen 6 PWM-Signale ausgeben können und zusätzlich 6 analoge Eingänge. Diese Pins sind auf der Oberseite des Boards angebracht mit weiblichen Steckern im 0.1 Zoll Raster. Es gibt zahlreiche kommerzielle "Shields". Das sind Plug-In Boards für verschiedene Anwendungen die auf diesen Pins montiert werden können. Um die Aufgabenstellung korrekt zu lösen, musste ein eigenes „Shield“ entwickelt werden (siehe Abs. 5).

3.1.1 Arduino Duemilanove

Der in der Studienarbeit eingesetzte Arduino Duemilanove ist das neueste Modell der normalen Arduino-Boards (noch neuer ist der Arduino Mega für komplexere Projekte). "Duemilanove" ist italienisch und bedeutet "2009", was das Jahr des Erscheinens dieses Modells ist.

Das "Duemilanove"-Board basiert auf einem ATmega328-Mikrocontroller. Ebenso besitzt es 14 digitale Ein-/Ausgänge (6 davon können als PWM Kanäle genutzt werden), 6 analoge Eingänge, ein 16 MHz Quartz, eine USB-Schnittstelle zur Verbindung mit dem PC, einen 6 poligen ISP-Anschluss und einen Resetknopf.

Der Arduino kann per USB Verbindung oder regulären Adapterkabel mit Strom versorgt werden. Die Umschaltung erfolgt automatisch. Externe (nicht-USB) Strom kann entweder per Gleichstromadapter oder Batterie erfolgen. Der Adapterstecker kann mit einem 2.1 mm Stecker angeschlossen werden, der innen eine positive Spannung hat. Die Batterieleitungen können auch direkt mit den GND und VIN Pin Headern Verbunden werden.

Das Board kann mit einer externen Spannung von 6 bis 20 Volt versorgt werden. Wenn eine Spannung von weniger als 7 Volt verwendet wird, so kann der 5V Pin weniger als 5V bereitstellen und das Board kann instabil laufen. Wenn mehr als 12 Volt Spannung anliegt, so kann der Voltregulator überhitzen und kann das Board beschädigen. Die Empfohlene Spannung liegt deshalb zwischen 7 und 12 Volt.

Die Strompins sind wie folgt:

VIN: Die Eingangsspannung für das Arduino Board wenn eine externe Spannungsquelle anliegt (im Gegensatz zu den 5 Volt von der USB Verbindung oder anderen regulierten Stromquellen). Strom kann auch durch diesen Pin bereitgestellt werden oder von hier abgegriffen werden.

5V: Eine regulierte Stromquelle, mit der der Microcontroller und andere Komponenten auf dem Board versorgen. Diese kann entweder vom VIN per on-board Regulator stammen oder per USB oder anderen regulierten 5V Quellen stammen.

3V3: Eine 3,3 Volt Stromquelle, die vom FTDI Chip erzeugt wird. Maximale Stromstärke sind 50 mA.

GND: Masse Pin (Ground).

Speicher:

Der ATmega hat 32 KB Flash Speicher um Code zu speichern, von denen 2 KB für den Bootloader (3.1.2) belegt wird. Er hat 2 KB RAM und 1 KB EEPROM (Der mit Hilfe der EEPROM Bibliothek geschrieben und gelesen werden kann).

Ein- und Ausgänge:

Jede der digitalen Pins auf dem Arduino kann als Eingänge oder Ausgänge genutzt werden mit pinMode(), digitalWrite() und digitalRead() Funktionen. Der Betrieb läuft mit 5 Volt. Jeder der Pins kann ein Maximum von 40 mA senden und empfangen und hat einen internen Pull-Up Widerstand von 20-50 kOhm (per default deaktiviert).

Zusätzlich haben einige Pins spezialisierte Funktionen:

Serial 0 (RX) und 1 (TX): Wird genutzt um serielle TTL Daten zu Empfangen (RX) und zu Senden (TX). Diese Pins sind mit den passenden Pins des FTDI USB-to-TTL seriellen chips verbunden.

Externe Interrupts 2 und 3: Diese Pins können konfiguriert werden Interrupts bei geringen Werten, ansteigenden oder fallenden Flanken oder bei einer Wertänderung zu triggern.

PWM 2 bis 13: Stellen 8-bit PWM Ausgänge mit der analogWrite() Funktion zur Verfügung.

LED 13: Es gibt eine eingebaute LED, die mit dem digitalen Pin 13 verbunden ist. Wenn der Pin HIGH Wert hat ist die LED an, sie ist aus bei einem LOW Wert.

Kommunikation:

Der Arduino hat eine Anzahl von Einrichtungen um mit einem Computer, anderen Arduinos oder anderen Microcontrollern zu kommunizieren. Der ATmega stellt eine UART TTL (5V) serielle Kommunikation bereit, die auf den digitalen Pins 0 (RX) und 1 (TX), sowie 14 bis 19 liegt. Ein FTDI FT232RL auf dem Board kanalisiert diese serielle Kommunikation über USB und den FTDI Treibern (in der Arduino Software enthalten) um dem Computer einen virtuellen COM Port bereitzustellen. Die Arduino Software umfasst eine Monitorfunktion des seriellen Ports und erlaubt den Empfang und Versand von simplen Textdaten vom und zum Arduino Board.

Eine SoftwareSerial Bibliothek erlaubt serielle Datenverbindung über jeden beliebigen digitalen Pin des Arduino.

Der ATmega unterstützt auch I2C (TWI) und SPI Kommunikation. Die Arduino Software enthält eine Wire Bibliothek um die Verwendung des I2C Busses zu vereinfachen; Details kann man in der Dokumentation auf der Wiring Website finden. Um die SPI Kommunikation zu verwenden findet man Details im ATmega1280 Datenblatt.

Programmierung:

Der Arduino kann mit der Arduino Software programmiert werden.

Der ATmega auf dem Arduino kommt mit gebranntem Bootloader, der es ermöglicht neuen Code auf den Chip zu schreiben ohne externe Hardware Programmer. Die Kommunikation in diesem Prozess wird unter Verwendung des original STK500 Protokolls durchgeführt.

Man kann den Bootloader auch umgehen und den ATmega über die ICSP (In-Circuit Serial Programming) Header programmieren.

3.1.2 Bootloader

Der Bootloader ist ein kleines Stück Software, das bereits auf dem Microcontroller Chip des Arduinos geladen ist. Es erlaubt das Hochladen von Sketches auf den Arduino ohne die Verwendung von zusätzlicher, externer Hardware.

Wenn man den Reset Knopf vom Arduino Board drückt, so wird der Bootloader (sofern vorhanden) geladen. Der Bootloader sendet ein Blinksignal an Pin 13 als Bestätigung seiner Funktionstüchtigkeit. Du kannst eine LED and Pin 13 anschließen, um es zu überprüfen. Der Bootloader ist dann empfangsbereit für Signale oder Daten vom Computer. Normalerweise ist dies ein Sketch, das der Bootloader dann in den Flashspeicher des ATmega168 oder ATmega8 Chips schreibt. Danach wird das neu upgedatete Programm gestartet. Wenn der Computer kein Sketch gesendet hat, so wird das zuletzt upgedatete Programm gestartet. Wenn der Chip noch jungfräulich ist, ist der Bootloader das einzige Programm auf dem Chip und startet sich selber.

Warum verwenden wir einen Bootloader?

Die Verwendung eines Bootloaders vermeidet externe, zusätzliche Hardware Programmer. Den Bootloader allerdings auf den Chip zu schreiben erfordert solche externen Programmer.

3.2 Software

Die Arduino-IDE ist eine plattformunabhängige Java-Anwendung, sie dient als Code-Editor und -Compiler und ist auch in der Lage, die Programme (hier: Sketches) auf die Hardware zu übertragen. Sie basiert auf Processing, einer auf die Einsatzbereiche Grafik, Simulation und Animation spezialisierten objektorientierten, stark typisierten Programmiersprache. Sie ist von Wiring abgeleitet, einer C-ähnlichen Sprache. Programmiert wird mit gcc, unter Verwendung der avr-gcc-Library und weiteren Arduino-Libraries, die die Programmierung in C/C++ stark vereinfachen.

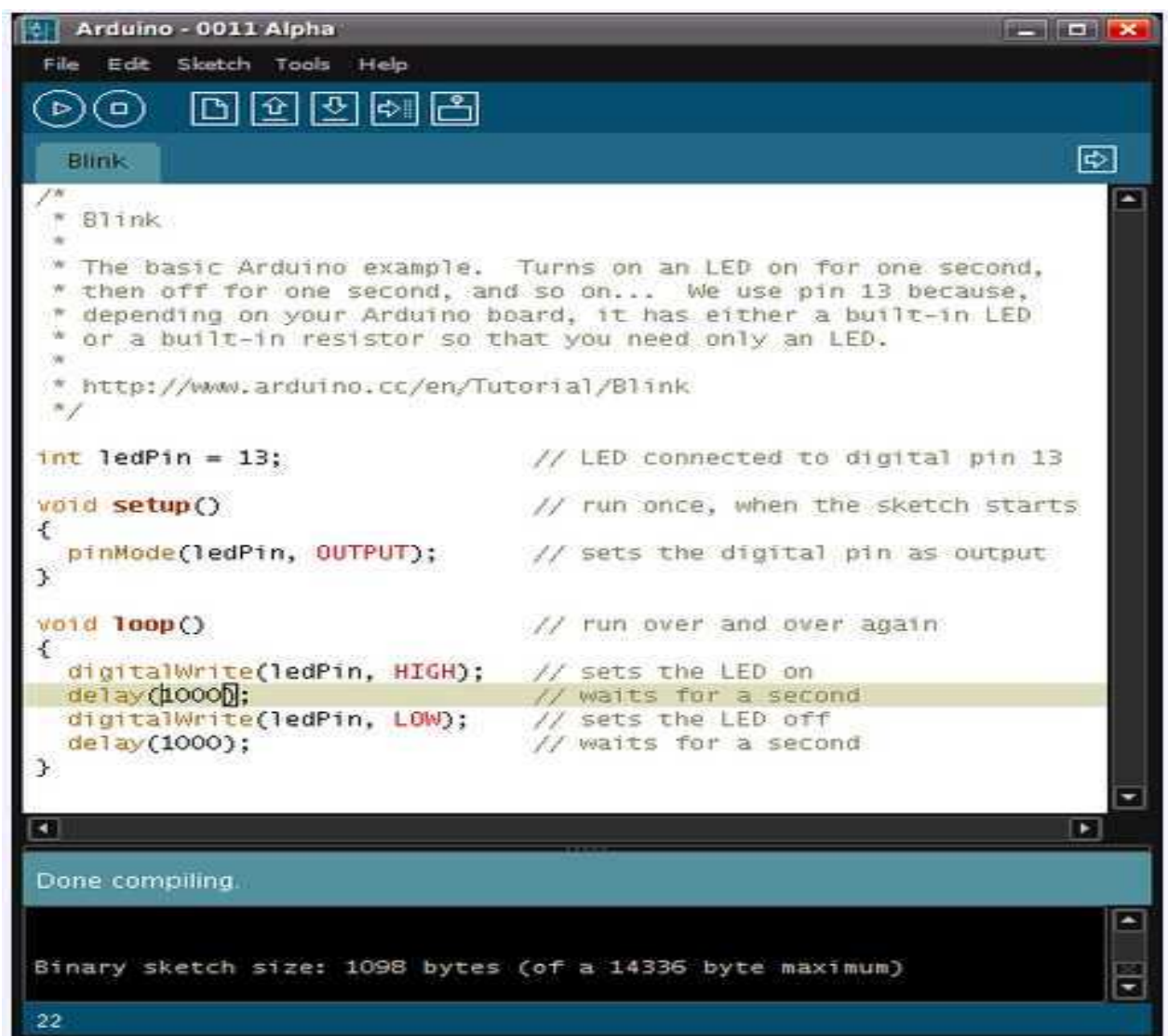


Abbildung 2: Arduino-IDE

Für ein funktionstüchtiges Programm genügt es, zwei Funktionen zu definieren:

- **setup()** - wird beim Start des Programmes angerufen, nützlich um z.B. Pins als Eingang oder Ausgang zu definieren
- **loop()** - wird durchgehend angerufen, solange bis das Arduino-Board ausgeschaltet wird

4 EIB

Der Europäische Installationsbus (EIB) ist ein Standard nach EN 50090, in der aktuellen Version als KNX-Standard auch nach ISO/IEC 14543-3, der beschreibt, wie bei einer Installation Sensoren und Aktoren in einem Haus miteinander verbunden werden können, der festlegt, wie Sensoren und Aktoren miteinander kommunizieren müssen (=Protokoll). Der EIB steuert zum Beispiel die Beleuchtung und Jalousien beziehungsweise Beschattungseinrichtungen, die Heizung sowie die Schließ- und Alarmanlage. Mittels EIB ist auch die Fernüberwachung und -steuerung eines Gebäudes möglich. Eine Steuerung erfolgt dabei über den Benutzer selbst oder über einen mit entsprechender Software ausgerüsteten Computer.

4.1 Technik

Zwischen dem Verbraucher (zum Beispiel Elektrogerät, Lampe, Fensteröffner) und der Netzspannung wird ein Steuerungsgerät, „Aktor“ genannt, eingebaut. Der Aktor ist an das EIB-Netz angeschlossen und erhält von diesem Daten. Diese Daten stammen entweder direkt von einem „Sensor“ (zum Beispiel Schalter, Helligkeit, Temperatur) oder indirekt von einem Computer (z. B. regelt dieser zeitgesteuerte Schaltungen, sonstige Auswertung von Sensordaten je nach seiner Programmierung).

Erhält der Aktor den Befehl, dem Verbraucher Spannung zuzuführen, so schaltet er die Netzspannung an das Gerät durch. Der Befehl kann von unterschiedlichen Sensoren kommen. Die EIB-Leitung (Bezeichnung u.a. : J-Y (St) Y 2x2x0,8 EIB bzw. YCYM 2x2x0,8 EIB) besteht in der Regel aus zwei Aderpaaren (rot-schwarz und weiß-gelb), wovon jedoch nur rot-schwarz verwendet wird.

Die EIB-Anlage wird von einer Spannungsversorgung mit 30 V Nennspannung (Gleichspannung) versorgt. Diese Spannung versorgt die Busankoppler, über die jedes EIB-Gerät mit den anderen vernetzten EIB-Geräten kommuniziert. Der Datenaustausch zwischen den EIB-Geräten erfolgt über Telegramme (siehe 4.4). Durch das Zugriffsverfahren CSMA/CA werden Telegrammverluste im Falle von Kollisionen (Abs. 4.3.5) ausgeschlossen. Der EIB-Bus kommuniziert mit einer Übertragungsrate von 9,6 kBit/s, was bei korrekter Programmierung auch für mehrere 10.000 Geräte ausreichend ist.

4.2 Struktur

4.2.1 Physikalische Struktur (= Netztopologie)

Der EIB ist aufgeteilt in 15 Bereiche mit jeweils 15 Linien und nochmals 255 Teilnehmern pro Linie. Somit können bis zu $15 \times 15 \times 255 = 57.375$ Busteilnehmer einzeln gesteuert werden. Damit bezeichnet zum Beispiel die Physikalische Adresse 8.7.233 in Bereich 8, Linie 7, den Teilnehmer 233.

Auf einer Buslinie können eine begrenzte Anzahl von Busteilnehmern (TN oder TLN) angeschlossen werden. Jede Linie kann aus bis zu vier Liniensegmenten bestehen, an jedes Liniensegment können bis zu 64 TLN angeschlossen werden. Dadurch sind max. 255 TLN je Linie möglich (Linienkoppler nicht mitgerechnet). Jedes Liniensegment benötigt eine eigene Spannungsversorgung.

Um die Linien in ihrer Struktur zu erweitern, werden sie über eine so genannte Hauptlinie miteinander verbunden. Hierzu werden die Linien über Linienkoppler mit der Hauptlinie verbunden. Die Hauptlinie selbst benötigt wiederum mindestens eine Spannungsversorgung und kann maximal 255 Busteilnehmer beinhalten. Eine Hauptlinie verbindet maximal 15 Linien miteinander.

Hauptlinien können weiters über eine so genannte Bereichslinie verbunden und somit erweitert werden. Auch mit dieser lassen sich 15 Hauptlinien miteinander verbinden. Weitere 255 Teilnehmer lassen sich ebenfalls auf dieser einbinden.

Auf den übergeordneten Linien, Hauptlinie und Bereichslinie, werden meist Geräte, die Zentralfunktionen bieten, eingebunden. Dies sind z. B. physikalische Sensoren, eine Visualisierung, Logikkomponenten und Aktoren in Verteilern, die Schaltausgänge für Sensoren aus verschiedenen Linien zur Verfügung stellen.

4.2.2 Logische Struktur (frei programmierbar)

Zusammengehörige Aktoren und Sensoren werden mit einer sogenannten Gruppenadresse verbunden, die einfach einprogrammiert werden kann. Dadurch ergibt sich die Möglichkeit, die Zusammengehörigkeit von zum Beispiel Schaltern und Lampen jederzeit zu ändern, ohne neue Leitungen verlegen zu müssen.

Die Kommunikation der Geräte erfolgt mit standardisierten Befehlen. So ist sichergestellt, dass Geräte verschiedener Hersteller zusammen arbeiten. Damit wurde erstmals ein einheitlicher Standard geschaffen, der offen ist für alle Hersteller von Elektrogeräten bzw. Steuerkomponenten. Mittlerweile wurden weltweit mehrere hunderttausend Gebäude mit einer EIB-Anlage ausgestattet. Entsprechend groß ist auch die Vielfalt der Steuergeräte der verschiedenen Hersteller.

EIB ist ein offener Standard, d. h. jeder Hersteller/ Entwickler hat vollen Zugriff auf alle notwendigen technischen Informationen, die er für die Weiterentwicklung benötigt. Allerdings erfordert dies die beitragspflichtige Mitgliedschaft in der offenen Vereinigung Konnex Association. Daher wird kritisiert, dass dies kein wirklich offener Standard sei, da durch die Mitgliedschaft grundsätzlich Kosten entstehen. Erst wenn diese Mitgliedschaft auch kostenfrei ist, könne von einem „offenen Standard“ die Rede sein.

4.3 Bits und Bytes

4.3.1 Wie kommunizieren die Teilnehmer untereinander?

Die Kommunikation erfolgt seriell mit einer Geschwindigkeit von 9600 Bit/s. Es gibt keinen Master und keine Slaves, alle Geräte auf dem Bus sind prinzipiell gleichberechtigt. Man kann jedoch verschiedene Sendeprioritäten vergeben. Da ja nur zwei Leitungen benutzt werden, nämlich + und – vom Netzteil kommend, kommt hier nur das Halbduplex-Verfahren in Betracht. Es kann also nicht gleichzeitig gesendet und empfangen werden. Von daher müssen die Geräte verschiedene Timeout-Situationen überwachen und entsprechend reagieren.

4.3.2 Was passiert denn nun auf dem Bus?

Im Ruhezustand, wenn also kein Datenverkehr stattfindet, liegt der Pegel auf dem Niveau der Spannungsversorgung, typischerweise bei 30V. Bevor wir uns Bytes oder gar ganze Telegramme anschauen, betrachten wir zunächst einmal ein einzelnes Bit.

Soll nun ein Bit gesendet werden, gibt es zwei Möglichkeiten:

- Eine logische 1 wird gesendet, indem „gar nichts“ passiert. Der Pegel bleibt unverändert während der Dauer von 104µs auf ca. 30V.
- Eine logische 0 wird gesendet, indem das sendende Gerät für die Dauer von 35µs einen gewissen Strom fließen lässt. Dabei sinkt die Spannung auf typischerweise 21V, kann aber bis auf minimal 19V heruntergehen. Da zum Netzteil hin der Bus durch Drosseln angekoppelt ist, entsteht nach den 35µs eine höhere Spannung, da die durch den Stromfluss verursachte magnetische Energie in den Drossel nun wieder auf den Bus gelangt. Dieser Zustand klingt langsam ab, so dass nach spätestens 104µs seit der negativen Flanke das Ausgangs-Niveau erreicht ist und das zweite Bit gesendet werden kann.

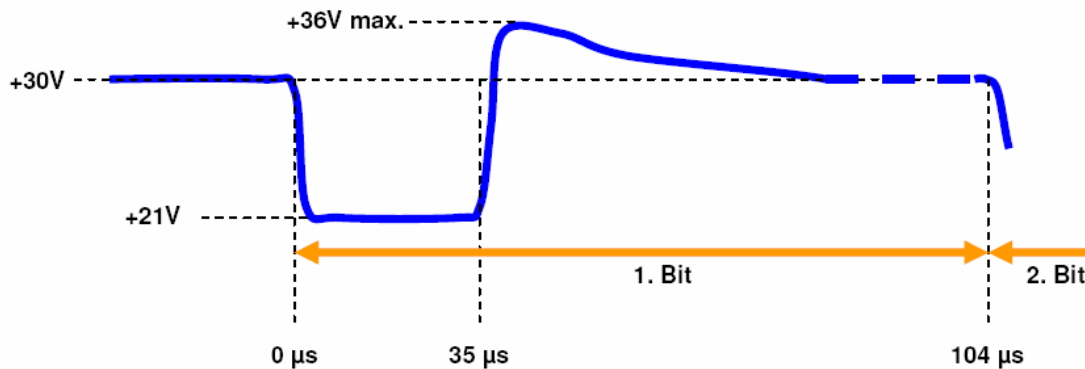


Abbildung 3: Spannungsverlauf

Zum Lesen der über den Bus gesendeten Bits braucht man also nur während der ersten 35µs prüfen, ob eine fallende Flanke vorliegt. Das Niveau während der logischen 0 ist dabei unerheblich und kann gewaltig variieren.

Um diese Signale für den Mikrocontroller aufzubereiten kann man folgende Eingangsschaltung für den Empfang benutzen:

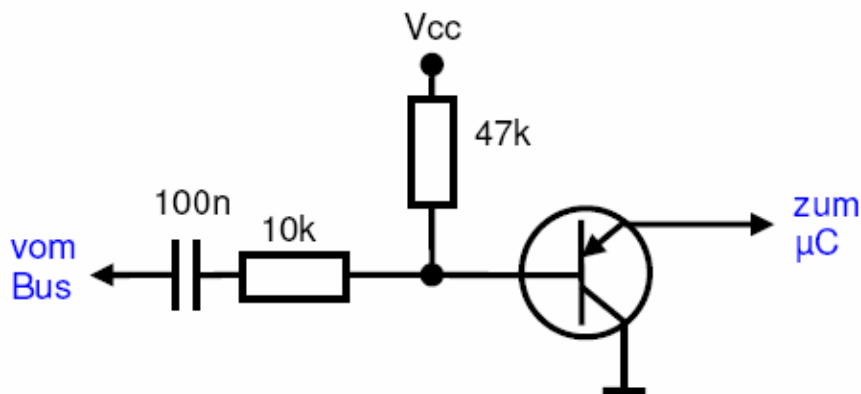


Abbildung 4: Empfangs-Schaltung

Dabei entsteht ein sauberes Rechteck-Signal mit gleicher Polarität wie das Original, d.h. im Ruhezustand ist der Controllerpin auf High. Sofern der verwendete Mikrocontroller keinen internen pull-up Widerstand hat, muss man diesen extern vorsehen, typischerweise 10k gegen Vcc. Das Niveau von ca. 21V auf dem Bus (s.o.), also eine logische Null entspricht dann 0V am Controllerpin, also Low.

4.3.3 Wie sendet der Mikrocontroller ein Bit?

Zum Senden eines Bits muss der Mikrocontroller eigentlich nur bei einer logischen Null etwas tun. „Eigentlich“ deswegen, weil zum Senden einer logischen 1 zwar das Bussignal nicht aktiv verändert wird, während dieser Zeit jedoch geprüft wird, ob eine Kollision vorliegt. Dazu aber später mehr. Zum Senden einer Null muss wie gesagt ein gewisser Strom fließen um die Busspannung absinken zu lassen. Das wird mittlerweile nicht mehr durch Übertrager realisiert, sondern über einen FET. Der folgende Schaltplan zeigt die Sende-Schaltung:

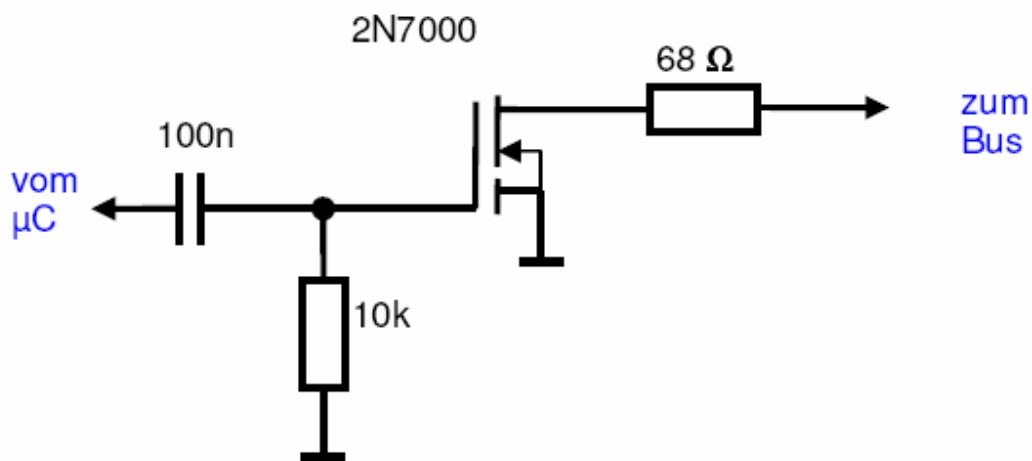


Abbildung 5: Sende-Schaltung

Während $35\mu\text{s}$ sendet der μC einen High-Pegel, der den FET durchschalten lässt. über den 68Ω Widerstand fließt nun der nötige Strom um den Spannungspegel auf den geforderten Wert absinken zu lassen. Da nur für sehr kurze Zeit ein Strom fließt reicht ein normaler 1/4-Watt Widerstand.

Sicherheitshalber verhindert der Kondensator, dass bei permanentem High-Pegel der FET dauerhaft durchschaltet und den Widerstand zum Glühen bringt. Der $10\text{k}\Omega$ Widerstand sorgt für ein sicheres Sperren des FET.

4.3.4 Wie sieht denn ein SendeByte aus?

Übertragen werden immer 11 Bits. Ein Startbit (Null), 8 Datenbits (das niedrigste Bit zuerst), ein Parity-Bit und ein Stopbit (immer 1). An diese 11 Bits schließt sich dann eine Pause von 2 weiteren Bit-Längen an, bevor das nächste Byte gesendet wird.

Das Parity-Bit wird gebildet, indem die Anzahl Einsen der 8 Datenbits gezählt wird. Ist sie ungerade ist das Parity-Bit 1.

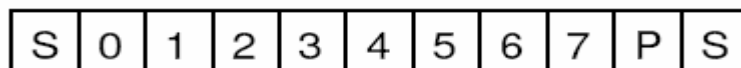
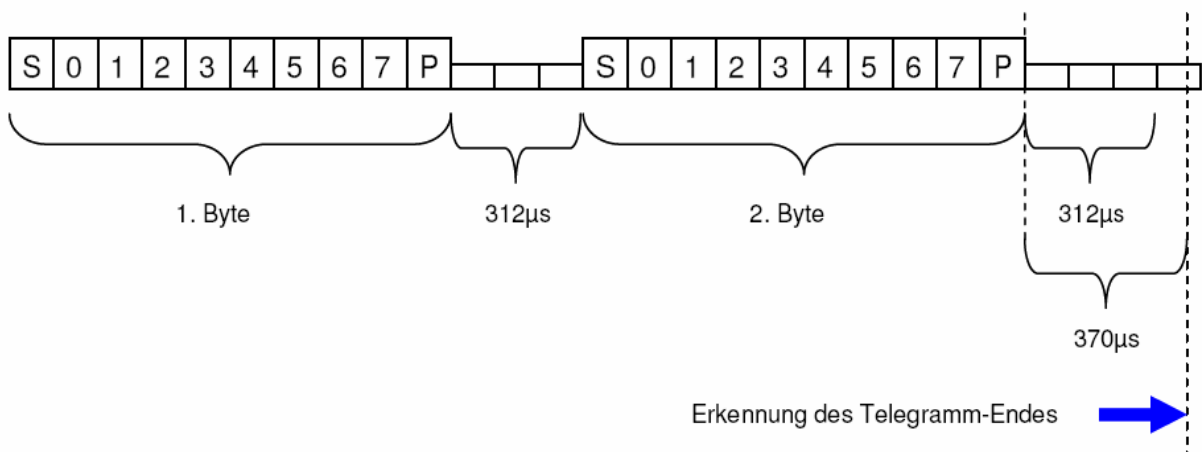


Abbildung 6: Sende-Byte

Damit dauert das Senden eines Byte $(11+2) \times 104\mu\text{s} = 1352\mu\text{s}$.

In der Praxis wird man das Stopbit nicht bemerken, da es eine logische 1 ist und sich somit von der anschließenden Pause nicht unterscheidet. Also wird man nach dem Senden des

Parity-Bit einfach 312µs Pause machen und dann mit dem Senden des nächsten Byte fortfahren. Das genaue Einhalten der Pausenlänge ist wichtig. Wird nämlich während einer Zeit von 370µs nach Ende des Parity-Bits nichts übertragen, so geht man davon aus, dass das Telegramm vollständig ist. Sollte hingegen ein weiteres Byte gesendet werden, wird dessen Startbit nach 312µs den „count-down“ stoppen.



4.3.5 Kollisionen vermeiden

Das Buszugriffsverfahren, das hier verwendet wird nennt sich CSMA/CA. Dabei ist das CA für „collision avoidance“ entscheidend. Es gibt auch z.B. ein CSMA/CD Verfahren, bei dem Kollisionen auftreten können, die die Daten unbrauchbar machen. Dies wird dort erkannt und die Daten nochmals gesendet. Bei uns geht es mit CSMA/CA aber von Anfang an darum Kollisionen zu vermeiden. Kollisionen werden auf Bit-Ebene erkannt und dementsprechend auch während einer Bitübertragung reagiert. Eine Kollision liegt also nur dann vor, wenn ein Teilnehmer eine 1 und ein anderer eine 0 zur gleichen Zeit sendet. Es stehen uns also nur die ersten 35µs eines Bits zur Verfügung dies zu erkennen. Und die Aufgabe hat jeder Teilnehmer, der eine 1 sendet, denn dabei sollte sich auf dem Bus ja nichts ändern (s.o.). Diese Teilnehmer lesen also im Wirklichkeit während des „Sendevorgangs“. Wenn eine 0 gelesen wird liegt also eine Kollision vor und der Teilnehmer bricht seinen Sendevorgang sofort ab. Der Teilnehmer, der die 0 gesendet hat, kriegt von alle dem nichts mit und fährt einfach fort.

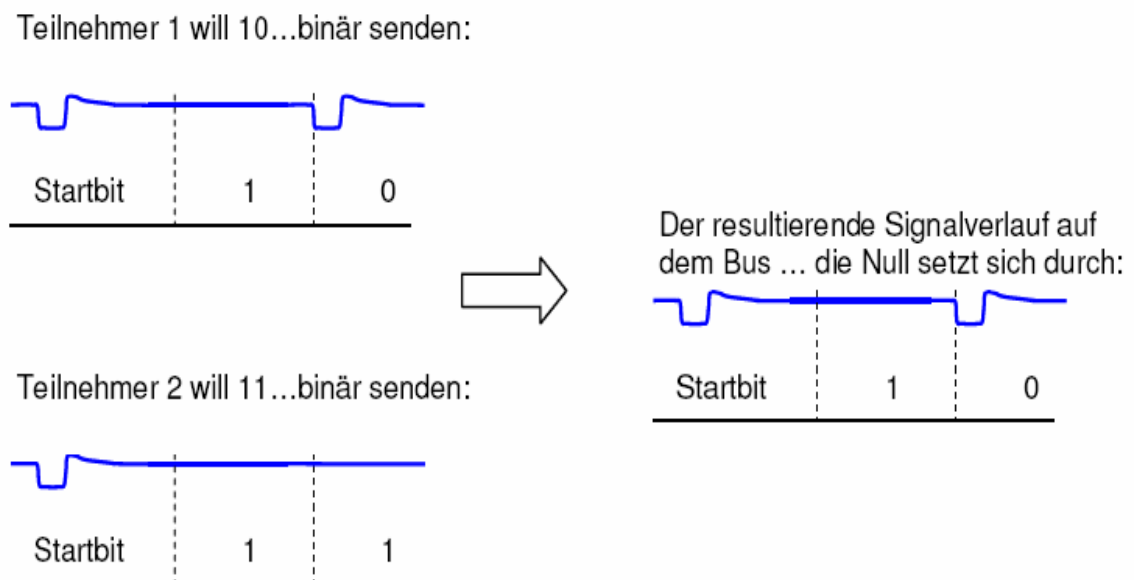


Abbildung 7: Kollisionsverhalten

In diesem Beispiel muss der Teilnehmer 2 erkennen, dass während des Sendens der zweiten 1 auf dem Bus tatsächlich eine 0 übertragen wurde. Er bricht daraufhin seinen eigenen Sendevorgang ab und wiederholt diesen später, sobald der Bus frei ist. Teilnehmer 1 hingegen setzt seinen Sendevorgang ungestört fort und bekommt von alledem nichts mit. Wie man sieht, ist dies also ein sehr effektives Verfahren, denn eigentlich findet ja gar keine richtige Kollision statt. Somit verliert man nicht unnötig kostbare Zeit. Jetzt könnte man sich die Frage stellen, ob o.g. Beispiel nicht der Idealzustand einer Kollision ist, denn die beiden Teilnehmer senden ihre Bits ja jeweils synchron zur gleichen Zeit. Könnte es also sein, dass die Teilnehmer versetzt gegeneinander senden und somit die Bits des Einen in die Pausen des Anderen rutschen? Um es vorwegzunehmen... eigentlich nicht. Es gibt nämlich eine Spielregel, die da besagt: „Bevor ich auf den Bus senden darf, muss ich lauschen und während 5,1ms keine Aktivität wahrnehmen. Dann ist der Bus frei und ich kann senden.“ Das heißt, dass wenn ein Teilnehmer bereits angefangen hat sein Startbit zu senden, darf kein Anderer einen Sendeversuch starten. Zwei Teilnehmer können also nur dann beide senden wollen, wenn sie exakt zur gleichen Zeit anfangen die Busaktivität zu überwachen. Nur dann würden beide gleichzeitig ihren Sendevorgang starten und damit lägen die Bits exakt übereinander.

4.4 Telegramm-Aufbau

Die gesamte Kommunikation zwischen Teilnehmern erfolgt über Telegramme, die einen recht einfachen Aufbau haben. Ein solches Telegramm besteht aus mindestens 7 und maximal 23 Bytes. Die meisten Aktionen auf dem Bus, wie z.B. Schalten, Dimmen, etc., lassen.

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8...22	Byte 9..23
Kontrollbyte	Quelladresse		Zieladresse		DRL	Nutzdaten		Checksum

Abbildung 8: Telegrammaufbau

4.4.1 Kontrollbyte

Das Kontrollbyte beinhaltet das Wiederholungsbit und die Priorität des Telegramms:

1	0	R	1	p1	p0	0	0
7	6	5	4	3	2	1	0

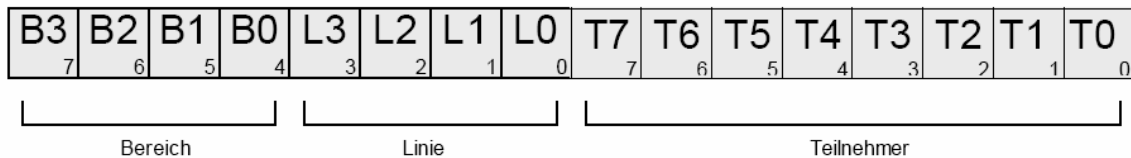
Es gibt 4 verschiedene Prioritäts-Levels.
Systemfunktionen haben die höchste Priorität.

<u>p1</u>	<u>p0</u>	<u>Bezeichnung</u>
0	0	Systemfunktion
0	1	Alarmfunktion
1	0	hohe Priorität
1	1	normale Priorität

Wird ein Telegramm das erste mal gesendet, so ist das Wiederholungsbit = 1. Wird ein Telegramm wiederholt, z.B. weil ein Empfänger es beim ersten Mal nicht verarbeiten konnte, ist dieses Bit = 0. Somit können die Teilnehmer, die das Telegramm bereits beim ersten Mal korrekt empfangen haben, es bei der Wiederholung ignorieren.

4.4.2 Quelladresse

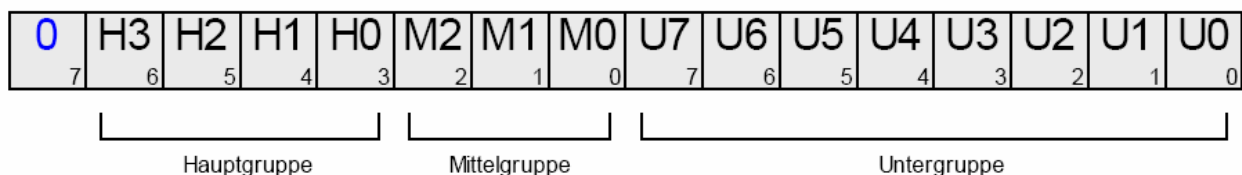
Die Quelladresse besteht aus zwei Byte, wobei erst das MSB dann das LSB übertragen wird:



Die Quelladresse ist immer eine physikalische Adresse eines Gerätes. Die Schreibweise ist dezimal <Bereich>.<Linie>.<Teilnehmer>, also z.B. 1.1.34

4.4.3 Zieladresse

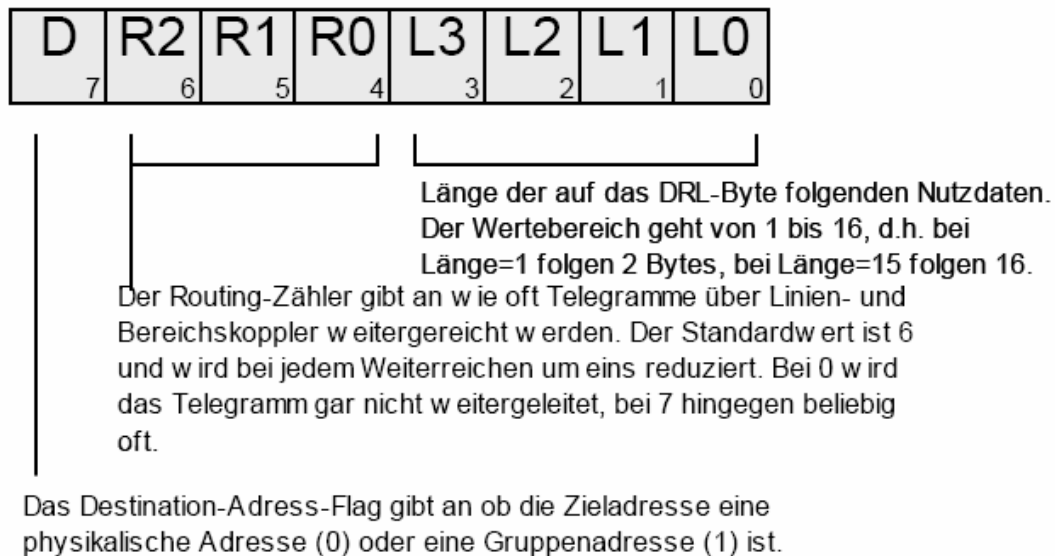
Die Zieladresse kann entweder eine physikalische Adresse oder eine Gruppenadresse sein. Das wird im DRL-Byte festgelegt (s.u.) Im Falle einer physikalischen Adresse ist die Bit-Aufteilung so wie unter Quelladresse beschrieben. Handelt es sich um eine Gruppenadresse sieht die Aufteilung in der Regel wie folgt aus:



Dies ist die Einteilung in 3 Ebenen, die Schreibweise der Gruppenadresse ist z.B. 1/3/43. Es gibt auch die (selten genutzte) Möglichkeit die Gruppenadresse in nur 2 Ebenen aufzuteilen. Dann gibt es nur die 4 Bit breite Hauptgruppe und die aus 11 Bit bestehende Untergruppe. Das erste Bit ist immer 0 für die Standard-Adressierung. Es gibt auch Sonderadressierungen, die interessieren uns aber erstmal überhaupt nicht.

4.4.4 DRL-Byte

DRL kommt von Destination-adress-flag, Routing-counter, Length und das sind bereits die drei Bestandteile dieses Steuerbytes:



4.4.5 Nutzdaten

Die Nutzdaten bestehen aus mindestens einem, maximal 16 Bytes. Dabei haben die ersten beiden Bytes eine besondere Bedeutung, denn zum Einen ist in ihnen der auszuführende Befehl kodiert, zum Anderen kann man viele Aufgaben schon mit diesen 2-Byte erledigen. Es existieren auch Befehle mit nur einem Byte.

Es gibt, grob gesagt, zwei mögliche Varianten für die Nutzdaten:

1. Der so genannte EIS (EIB Interworking Standard) ist ein Standard zur Kommunikation von Geräten unterschiedlicher Hersteller. Der Befehlssatz ist ziemlich mächtig und erlaubt so ziemlich alles an Informationen auf Gruppenadress-Ebene zu übertragen.

Es gibt 15 verschiedene EIS Formate für die folgenden Funktionen:

- EIS 1 Schalten
- EIS 2 Dimmen
- EIS 3 Uhrzeit
- EIS 4 Datum
- EIS 5 Wert, Zahl mit Nachkommastellen
- EIS 6 Relativwert, 0-100%
- EIS 7 Antriebssteuerung
- EIS 8 Zwangssteuerung
- EIS 9 Zahl mit Nachkommastellen nach IEEE
- EIS 10 16-Bit Wert
- EIS 11 32-Bit Wert
- EIS 12 Zugangskontrolle
- EIS 13 ASCII Zeichen
- EIS 14 8-Bit Wert
- EIS 15 Zeichenkette

2. Die zweite mögliche Nutzung sind Befehle zur Kommunikation mit nur einem Teilnehmer. Dies sind alle möglichen Befehle zur Programmierung, Parametrierung, zum Lesen und Schreiben des EEPROM eines Teilnehmers, etc.

4.4.6 Prüfbyte

Im Anschluss an die Nutzdaten wird immer das Prüfbyte gesendet. Es handelt sich dabei um die invertierte, bitweise EXOR-Verknüpfung aus allen vorher gesendeten Bytes des Telegramms. Auf der Empfängerseite kann man sich die Prüfung ziemlich einfach machen. Es reicht dabei, alle empfangenen Bytes inklusive des Prüfbytes ohne Übertrag zu addieren. Nur wenn am Ende 0xFF herauskommt ist das empfangene Telegramm OK.

Beispiel:

Das folgende Beispiel ist ein EIS 1 Telegramm vom Teilnehmer mit der physikalischen Adresse 1.1.1, der z.B. eine Lampe an einem Aktor-Ausgang mit der Gruppenadresse 1/2/5 einschaltet:

BC	1011 1100	Kontrollbyte
11	0001 0001	Quelladresse 1.1.1
01	0000 0001	
0A	0000 1010	Zieladresse 1/2/5
05	0000 0101	
E1	1110 0001	DRL Länge=1, d.h. Es folgen 2 Nutzbytes!
00	0000 0000	z.B. EIS 1
81	1000 0001	einschalten (zum ausschalten stünde hier 0x80)
2D	0011 1100	Prüfsumme

Ausserdem können alle Telegramme, die von anderen Geräten gesendet werden, empfangen werden. Für die Parametrierung wird keine Software wie z.B. ETS benötigt. Die physikalische Adresse wird beispielsweise einfach über einen Befehl über die serielle Schnittstelle gesetzt.

5.1 Aufbau

Dies ist die Grundsaltung für die Ankopplung von Mikrocontrollern an den EIB - Bus. Es handelt sich hier noch um eine vorläufige Version, die noch einiger Tests bedarf, aber bis jetzt sieht es so als, dass es die endgültige Version wird:

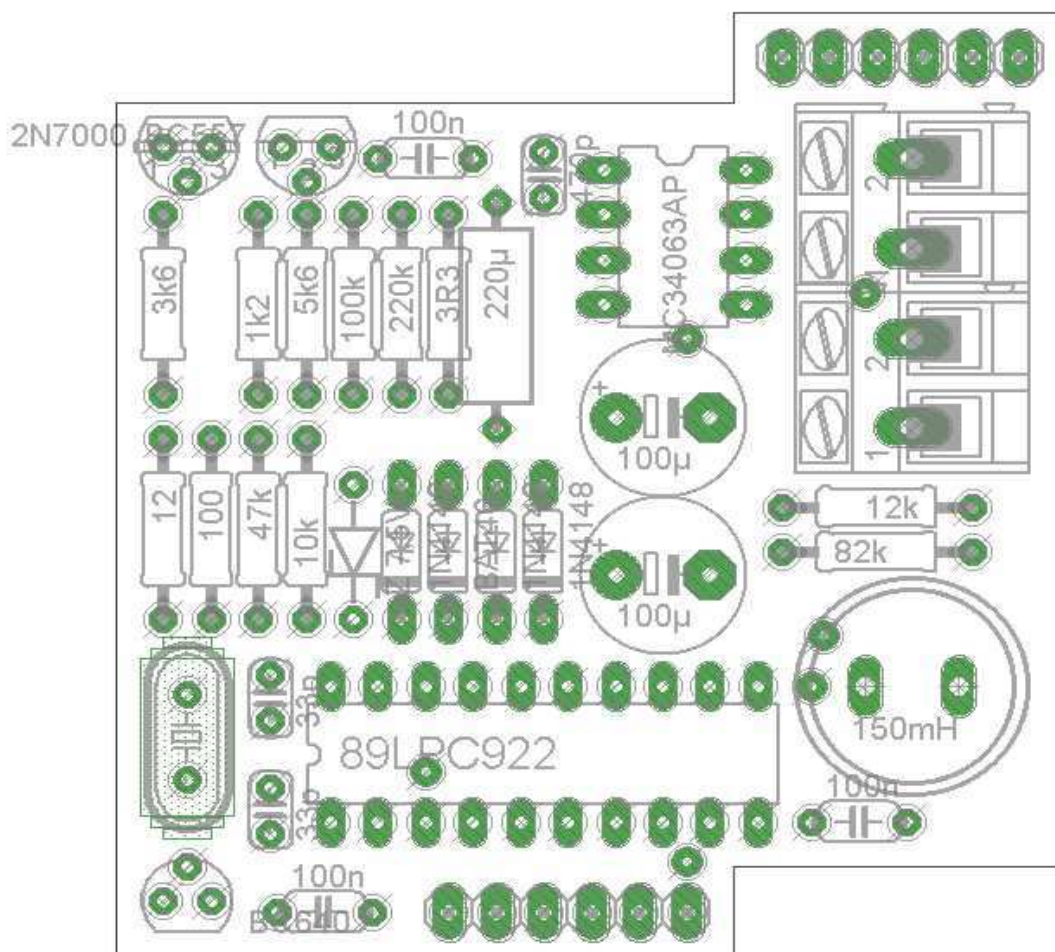
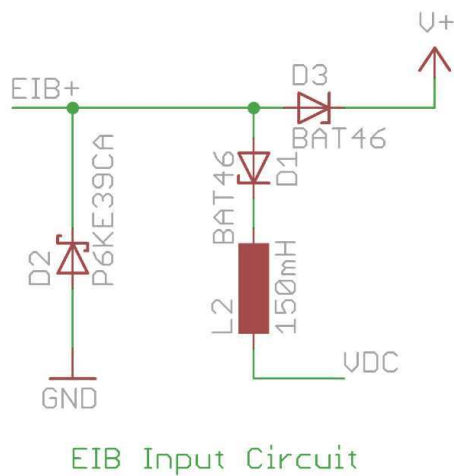


Abbildung 10: Platinenlayout Buskoppler

5.1.1 Die Schutzstufe:



Zunächst sorgt D1 und D3 für den Verpolungsschutz. Die Diode D2 sorgt dafür dass bei Spannungsspitzen die Schaltung geschützt ist. Der Pin **V+** geht dann auf die Empfangs- und Sendeschaltung. **Vext** geht zum Schaltregler und liefert den DC-Level für die Sendestufe.

Abbildung 11: Schutzstufe

5.1.2 Der Schaltregler:

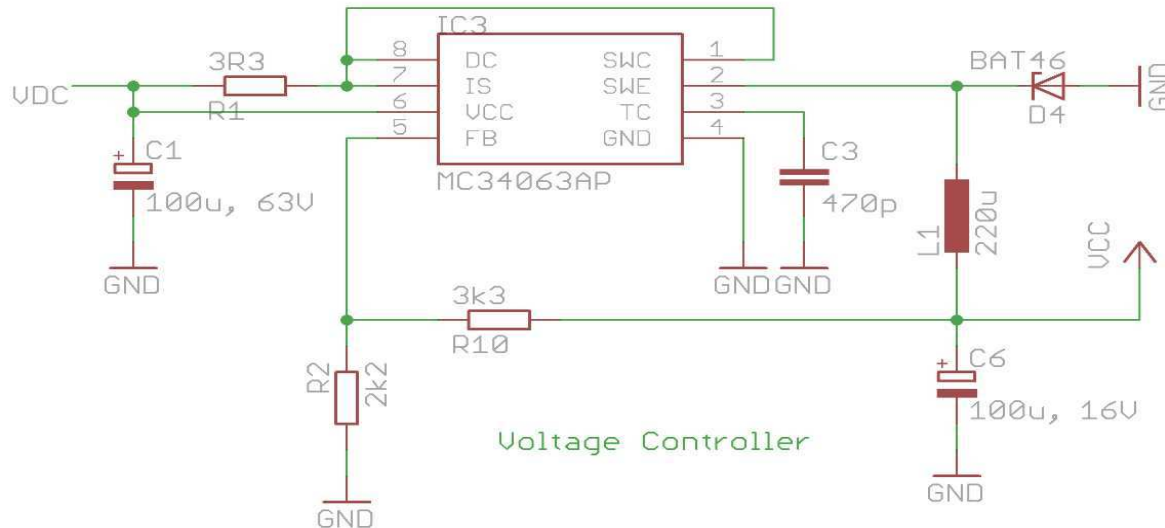
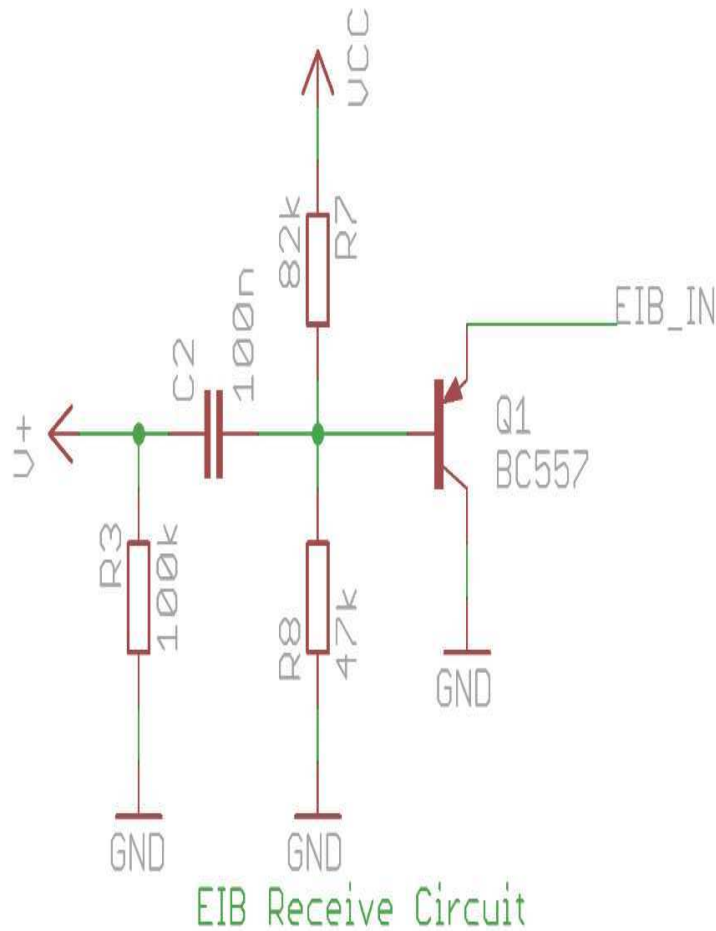


Abbildung 12: Schaltregler

Der Schaltregler MC34063AP sorgt dafür, dass aus den 30V Bussspannung, die 5V Betriebsspannung für den korrekten Betrieb des Arduino's werden. Somit ist ein Inselbetrieb des Arduino-Boards möglich und er benötigt keine zusätzliche externe Stromversorgung mehr.

5.1.3 Die Empfangsstufe:



Der Kondensator C2 sorgt dafür dass der Gleichstromanteil ausgekoppelt wird. Der Spannungsteiler R7, R8 kümmert sich darum, dass die richtige Spannung am Transistor Q1 anliegt um eine saubere Flanke für den Eingangspin am Mikrokontroller zu erzeugen. Falls der Mikrokontroller keinen internen Pull-Up Widerstand hat ist am Pin **EIB_IN** noch ein Widerstand gegen VCC anzubringen (typischerweise 10k).

Abbildung 13: Empfangsstufe

5.1.4 Die Sendestufe:

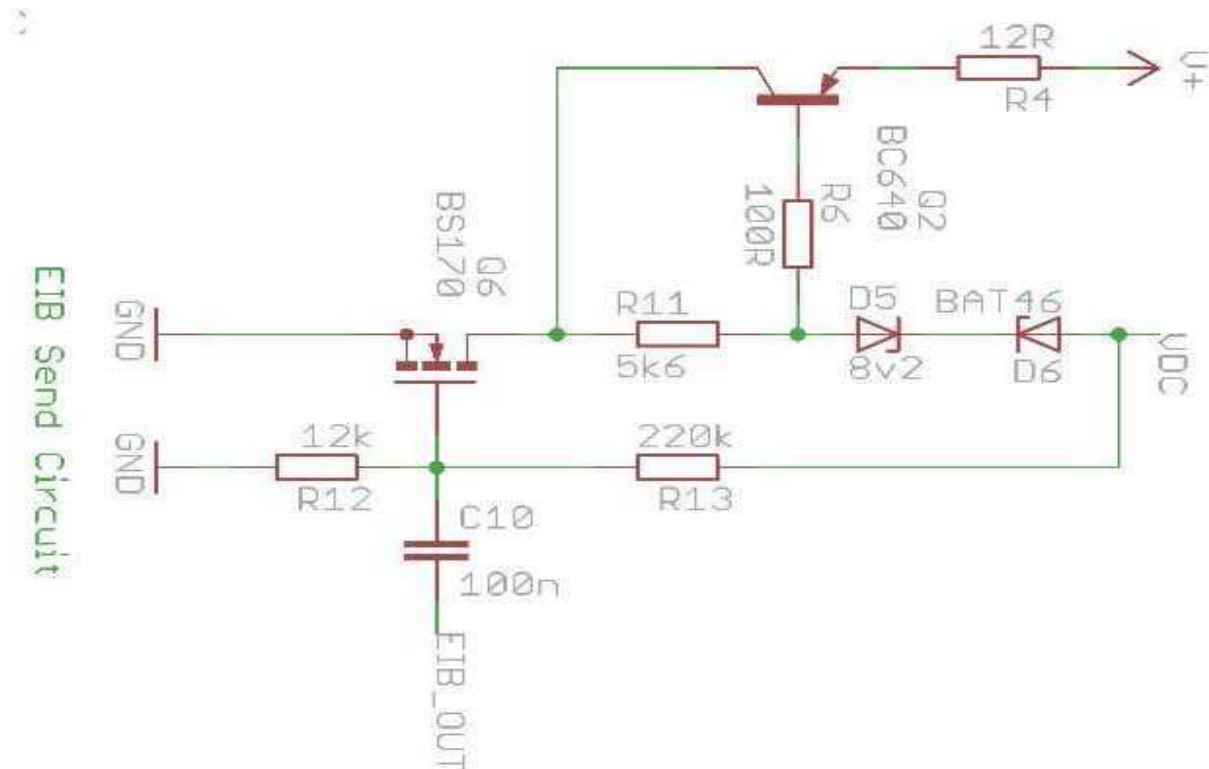


Abbildung 14: Sendestufe

Die Sendestufe wurde dahingehend überarbeitet, dass der Hub beim Senden einer logischen Null einen definierten Wert von ca. -7V hat. C10 sorgt dafür, dass es im Fehlerfall des μ Controllers nicht zu Dauersenden kommt und die Schaltung „schaden“ nimmt. Mit R12 und R13 wird das Gate von Q6 auf ca. 4V gehoben wenn ein positiver Puls vom μ Controller kommt. Damit wird Q6 durchgeschaltet der bei diesen 4V einen maximalen Strom I_{ds} von ca. 350mA fließen lässt.

Die Schaltung um Q2 sorgt dafür, dass der Strom nur so groß wird, dass die Busspannung nicht unter den ursprünglichen Wert minus 7V sinkt.

Natürlich sind die Signale 100% EIB-kompatibel, so dass die jeweilige Applikation problemlos in jede existierende EIB-Installation integriert werden kann. Ein Blick auf die Signale des Originals und der Schaltung bestätigt das (siehe 5.3).

5.2 Firmware

Die Firmware für das RS-Interface ist eine Schnittstelle zwischen dem Bus und einem PC oder µController und wird seriell angeschlossen. Auf den LPC wird eine eigens von Freebus entwickelte Software als Hex-Datei geflasht, die den späteren Telegrammaufbau erledigt und sich um das Versenden und Empfangen kümmert.

Leider ist diese Hex-Datei nicht als C-Code erhältlich, da hier das Know-how von mehreren Jahren Entwicklungsarbeit dahinter steckt.

Die Schnittstelle ist auf 115.200 Baud , No Parity , 8 Datenbits , 1 Stoppbit eingestellt.

Wenn ein Terminalprogramm am PC verwendet werden soll, muss es auf CR und LF eingestellt werden.

5.2.1 Telegramme senden:

Folgende Kommandos können an das Interface durch den Arduino gesendet werden, jeweils durch ein CR LF abgeschlossen:

- **fbs01/BB/L/TTT=0** oder **1** : sendet ein EIS1 Telegramm (Schalten) auf den Bus
BB/L/TTT ist die Gruppenadresse genau in dem Format (ggf. mit Nullen auffüllen!)
Beispiel: fbs01/01/4/007=1[CR][LF] schaltet Gruppenadresse 1/4/7 ein
- **fbs06/BB/L/TTT=XXX** : sendet ein EIS6 Telegramm (Dimmwert) auf den Bus
BB/L/TTT ist die Gruppenadresse genau in dem Format (ggf. mit Nullen auffüllen), XXX ist ein Byte Wert für die Helligkeitsstufe zwischen 0 und 255
Beispiel: fbs06/01/4/007=192[CR][LF] dimmt die Lampe mit der Gruppenadresse 1/4/7 auf 75%

5.2.2 Konfiguration:

- **fbrpa**: lesen der physikalischen Adresse des Adapters
- **fbspaXX.XX.XXX** : Setzt die physikalische Adresse des Adapters (ggf. mit Nullen auffüllen)

5.2.3 Telegramme empfangen:

Alle empfangene Multicast-Telegramme vom Bus werden seriell in der Form BB/L/TTT=xxxxl vom LPC an den Arduino ausgegeben.

Zusätzlich speichert das RS-Interface laufend den Wert von max. 265

Gruppenadressen in einer internen Tabelle. Diese wird beim Neustart gelöscht.

Wenn bereits 256 verschiedene Gruppenadressen gespeichert sind, werden neue verworfen. Man kann also jederzeit den Wert einer Gruppenadresse abfragen, ohne ein read_value_request Telegramm senden zu müssen. Das geschieht mit dem Befehl

fbrgaBB/L/TTT

Mit **fbdump** kann man sich die Tabelle auch komplett in hex ausgeben lassen.

5.3 Signalverlauf

5.3.1 Der original Signalverlauf

Von einem original EIB Gerät erzeugter Signalverlauf. Deutlich zu sehen ist die Normal-Spannung von 30V. (10V/Div) Bei einer Null sollte die Spannung bis auf minimal 19V zurückgehen, hier ca.24V. Die Ausschläge über 30V am Ende der Null sind normal und durch die Induktivitäten im Netzgerät (Drossel) bedingt.

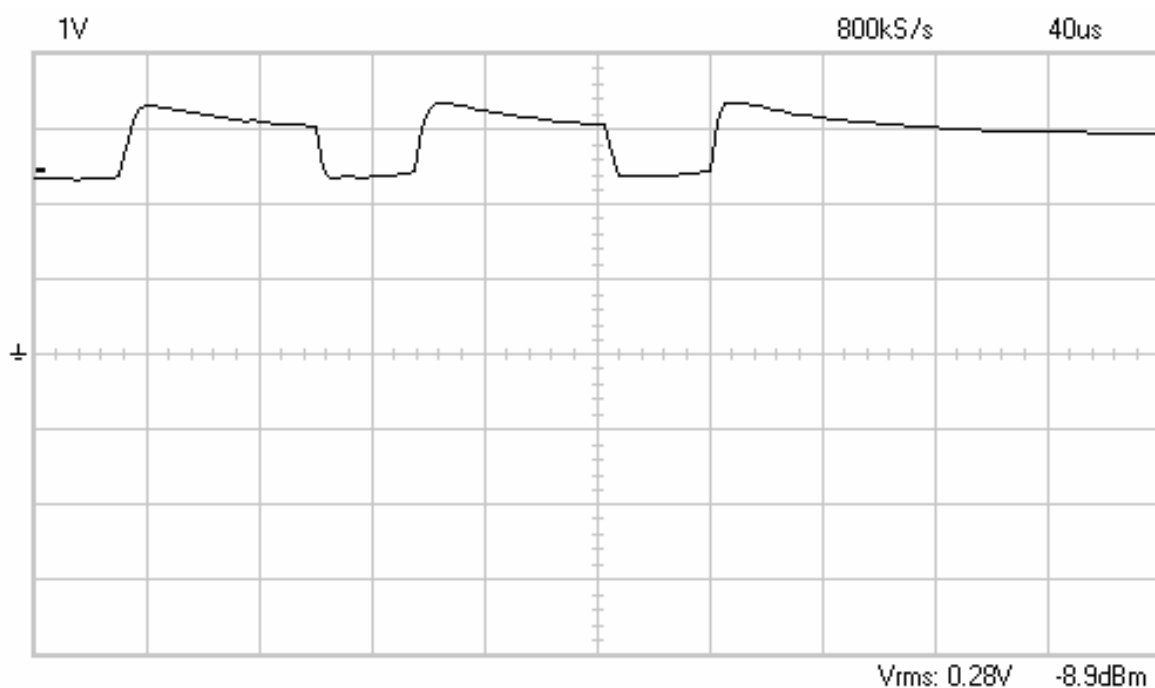


Abbildung 15: Original Signalverlauf

5.3.2 Der Buskoppler Signalverlauf

Der Signalverlauf dieser Schaltung ist fast völlig identisch mit dem Original. Die Flankensteilheit ist einen Hauch flacher, was aber völlig innerhalb der zulässigen Toleranz ist.

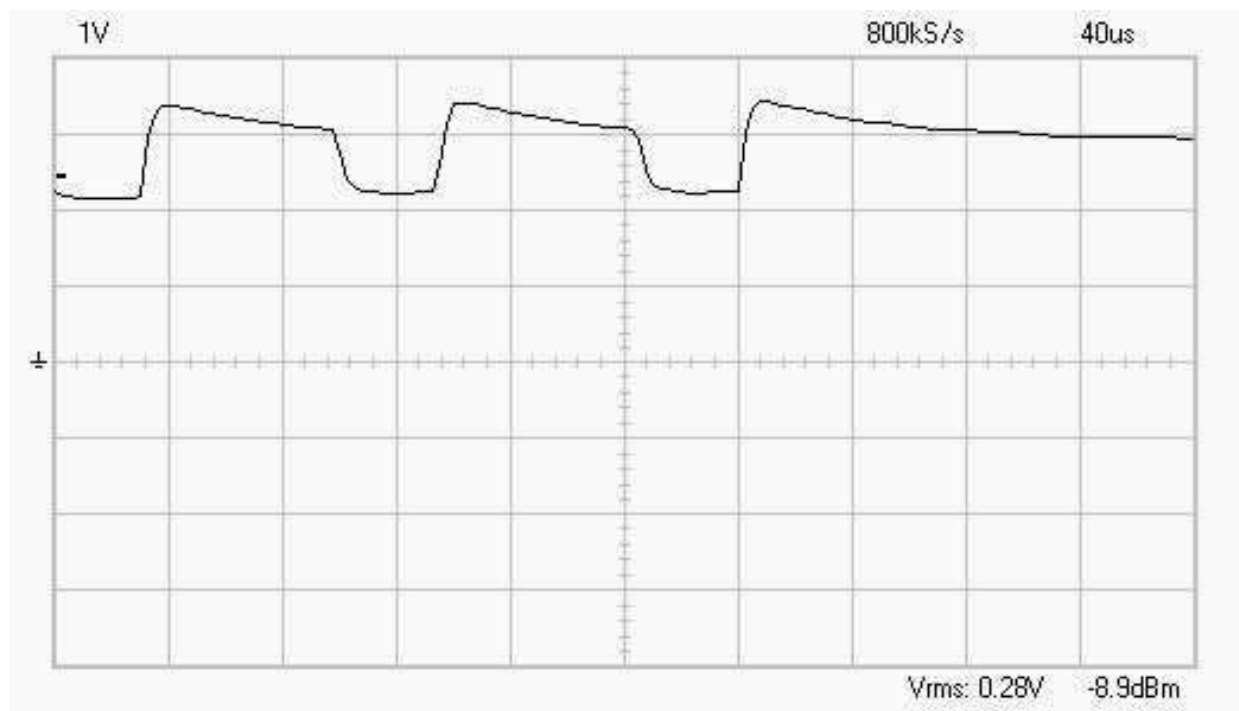


Abbildung 16: Buskoppler Signalverlauf

6 Arduino Sensor

Als Sensor dient ein Anwenderprogramm, welches dem Anwender erlaubt, verschiedene Sensorbefehle über den PC und die Tastatur einzugeben und zu steuern.

Über die programmierte Menüführung ist es möglich folgende Aktionen auszuwählen und auszuführen:

1. ZIELADRESSE:
 - Neue Gruppenadresse des Zielobjektes eingeben
 - Zuletzt Gespeicherte Gruppenadresse verwenden
 - Zuletzt Verwendete Gruppenadresse verwenden

2. QUELLADRESSE:
 - Absenderadresse des Sensors anzeigen
 - Absenderadresse des Sensors ändern

3. FUNKTIONEN:
 - Ziel EIN-Schalten
 - Ziel AUS-Schalten
 - Ziel DIMMEN

6.1 Quellcodeauszug:

```
Serial.println("AKTOR EIN:");
Serial.print("fbs01/");          //Ausgabe EINSTRING PART I
for (d=0;d<8;d++)
{ Serial.print(Adresse[d]); }   //Ausgabe EINSTRING PART II (Adresse)
Serial.println("=1");           //Ausgabe EINSTRING PART III

-----

Serial.println("AKTOR AUS:");
Serial.print("fbs01/");          //Ausgabe AUSSTRING PART I
for (d=0;d<8;d++)
{ Serial.print(Adresse[d]); }   //Ausgabe AUSSTRING PART II (Adresse)
Serial.println("=0");           //Ausgabe AUSSTRING PART III

-----

Serial.println("AKTOR DIMMEN:"); //Auswählbare Dimmwerte
Serial.print("fbs06/");          //Ausgabe DIMMSTRING PART I
for (d=0;d<8;d++)
{ Serial.print(Adresse[d]); }   //Ausgabe DIMMSTRING PART II (Adresse)
Serial.print("=");
for (d=0;d<3;d++)
{ Serial.print(Dimm[d]); }       //Ausgabe DIMMSTRING PART III (Dimmwert)
```

7 Alternativen

Neben der oben beschriebenen Lösung gibt es noch einige weitere gute Lösungswege für die gegebene Aufgabenstellung, jedoch sind diese zeitlich ausgeprägter und kamen daher für mich nicht in Erwägung.

Die Besten unter diesen Lösungsmöglichkeiten fand ich auf der Seite von Herrn **Prof. Dr. Franz Graf**.

Er und sein Team von der FH-Regensburg beschäftigen sich seit Jahren mit dem EIB-BUS und der Open-Source Fähigkeit.

<http://homepages.fh-regensburg.de/~scg39398/eibteam/eibteam.htm>

8 Abbildungsverzeichnis

Abbildung 1: Arduino Duemilanove Board	6
Abbildung 2: Arduino-IDE	12
Abbildung 3: Spannungsverlauf	18
Abbildung 4: Empfangs-Schaltung	18
Abbildung 5: Sende-Schaltung	19
Abbildung 6: Sende-Byte	20
Abbildung 7: Kollisionsverhalten	21
Abbildung 8: Telegrammaufbau	23
Abbildung 9: Schaltplan Buskoppler	28
Abbildung 10: Platinenlayout Buskoppler	29
Abbildung 11: Schutzstufe	30
Abbildung 12: Schaltregler	30
Abbildung 13: Empfangsstufe	31
Abbildung 14: Sendestufe	32
Abbildung 15: Original Signalverlauf	35
Abbildung 16: Buskoppler Signalverlauf	36

9 Erklärung

gemäß § 5 (2) der „Studien- und Prüfungsordnung DHBW Technik“
vom 18. Mai 2009.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die
unten angegebenen Quellen und Hilfsmittel verwendet.

Karlsruhe, 11.01.2010

(Andreas Straub)

10 Quellenverzeichnis

Arduino:

- www.Freeduino.de
- www.arduino.cc
- <http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial>

EIB:

- www.Freebus.org
- www.eib-home.de
- www.knx.de
- http://www.knx-developer.de/bcu2_help_1/help_java.htm
- <http://www.auto.tuwien.ac.at/~mkoegler/index.php/bcus>
- <http://www.eiba-software.com/eibacom/Volume1.zip>
- <http://homepages.fh-regensburg.de/~scg39398/eibteam/eibteam.htm>

11 Anhang

Auf der CD im Anhang befinden sich in digitaler Form der C-Code des Sensors, die Datenblätter zu den entsprechenden Bauteilen, die Eagle-Dateien, sowie diese Studienarbeit im .pdf Format.

Die Datenstruktur ist zur Ablage auf dem zentralen Server der DH-Karlsruhe gemäß der Vorgabe „Abgabe von Studienarbeiten in elektronischer Form“ geeignet.

\C-Code

[Sensor.c](#) (C-Code des Sensors)
[Sensor.pde](#) (Sketch Code des Sensors für die Arduino DIE)
[rs.hex](#) (Hex Datei für den LPC -> RS Interface)

\Bericht (enthält die Studienarbeit im PDF-Format)
[Studienarbeit Arduino.pdf](#)

\Datasheets (enthält die Datenblätter der Bauteile)
[LPC](#)
[Atmel](#)
[Mosfet](#)
[Transistoren](#)
[Spannungsregler](#)

\Quellen (enthält PDF-Dateien von Quellen)

\prg (enthält die EAGLE-Dateien des Buskopplers)
[BUS.sch](#)
[BUS.brd](#)