

Protokoll zum Mikrocomputertechnik-Praktikum

Thema: LIN-Kommunikation



<http://mbecker-tech.de>

Verfasser: Markus Becker

Datum: 30. Juni 2010

Unterschrift des Verfassers:

Mit meiner Unterschrift bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen verwendet habe.

Diese Dokument entstand im Rahmen des Praktikums zur
Mikrocomputertechnik-Vorlesung an der FH-Ingolstadt und darf gemäß
Creative-Commons-Lizenz (by-nc-sa) verwendet werden, sofern folgende Punkte
eingehalten werden: Namensnennung des Autors, nicht kommerziell, Weitergabe unter
gleichen Bedingungen (selbes Lizenzmodell).

Zusammenfassung

Inhalt dieses Protokolls ist die Realisierung einer LIN-Kommunikation auf einem xc161-Mikrocontroller im Rahmen des Praktikums zur Vorlesung *Mikrocomputertechnik* im Sommersemester 2010.

Inhaltsverzeichnis

1	Einleitung	3
2	Grundlagen des LIN-Busses	3
2.1	LIN - Local Interconnect Network	3
2.2	Physikalische Schicht (Physical Layer)	4
2.3	Sicherungsschicht (Data Link Layer)	5
3	Konfiguration des ASC-Moduls	6
3.1	Konfigurationsregister ASC1_CON	6
3.2	Konfiguration der Portpins	7
3.3	Generierung der Baudrate	7
3.4	Interrupts	7
3.5	Initialisierung des LIN-Transceivers	7
4	Implementierung	8
4.1	Datenstrukturen und Konstanten	8
4.2	Bereitstellen der Daten	9
4.3	Erkennung des Rahmen-Anfangs	9
4.4	Empfangen des Headers	10
4.5	Senden der Daten	11
4.6	Berechnung der Checksumme	11
5	Ergebnis	12
	Literatur	13
	Abbildungsverzeichnis	13
	Listings	13

1 Einleitung

Aufgabe dieses Versuchs war die Realisierung einer Kommunikation über einen LIN-Bus, wobei der Master-Knoten und das Bus-Protokoll (Zuordnung von Daten zu Identifiern) bereits gegeben war. Hierzu mussten UART-Modul, Ports sowie Interrupts des xc161-Mikrocontrollers entsprechend konfiguriert und programmiert werden.

Im Folgenden werden zuerst die theoretischen Rahmenbedingungen des LIN-Protokolls aufgezeigt und anschließend die technische Realisierung und Implementierung auf der xc16x-Plattform beschrieben.

2 Grundlagen des LIN-Busses

2.1 LIN - Local Interconnect Network

Der LIN-Bus ist ein auf dem UART basierender Eindraht-Bus, der im Automotive-Bereich als Kompromiss zwischen diskreter Verkabelung und „höheren“ Busanbindungen (z.B. CAN) bei Sensoren und Aktoren verwendet wird.

Die maximale Rohdatenrate beträgt $20 \frac{kbit}{s}$, wobei in der Regel lediglich UART-typische Datenraten von $2.4 \frac{kbit}{s}$, $9.6 \frac{kbit}{s}$ und $19.2 \frac{kbit}{s}$ verwendet werden [3].

Entwickelt wurde der LIN-Bus vom *LIN Consortium*, einem Zusammenschluss von u.a. Automobil- und Halbleiterherstellern sowie Zulieferfirmen.

Gründe für die Einführung:

- niedrigere Kosten im Vergleich zum CAN-Bus durch:
 - günstigere Transceiver
 - eine Datenübertragungsleitung weniger, keine teure Twisted-Pair-Leitung
 - geringere Ansprüche an Genauigkeit der Oszillator-Frequenz, dadurch Einsatz günstigerer Komponenten (RC-Kombinationen statt Quarze) möglich

- niedrigere Kosten im Vergleich zu diskreter Verkabelung durch:
 - Zeitmultiplex: Mehrere Signale über *eine* LIN-Leitung statt einer Leitung pro Signal
 - Ersparnis durch Reduzierung der Einzelleitungen und Pins an Steuergeräten
 - Beispiel: Stellmotor für Klimaanlage:

	Diskrete Verkabelung	LIN-Vernetzung
zum Betrieb notwendige Leitungen	2 für Potentiometer, 2 für Motor	2 für Stromversorgung, 1 für LIN-Bus
Anzahl Pins am Stellmotor	4	3
Anzahl Pins am Steuergerät	pro angeschlossenem Stellmotor 4	insgesamt 3

Tabelle 2.1: Einsparungen durch LIN-Vernetzung anstatt diskreter Verkabelung

2.2 Physikalische Schicht (Physical Layer)

Im Gegensatz zu schnelleren Bussystemen wie CAN, FlexRay, TTP oder Ethernet wird bei LIN lediglich ein Eindraht-Bus verwendet. Das Signal wird dabei auf einer Leitung übertragen, als Bezugspotential wird im Automotive-Bereich die Fahrzeug-Masse verwendet. Als Leitungscodierung wird ein *Non-Return-to-Zero-Code* eingesetzt, dessen Buspegel als prozentuale Werte der Versorgungsspannung definiert sind:

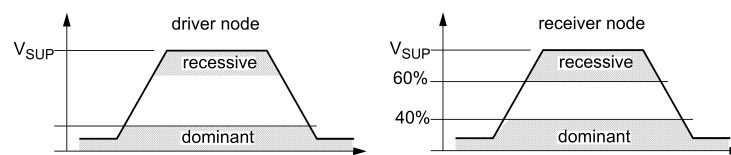


Abbildung 2.1: Buspegel am Sender und Empfänger [2]

Eine logische 0 gilt dabei als dominanter Wert, eine logische 1 als rezessiver Wert. Dies bedeutet, dass sich bei Sendekollisionen (gleichzeitiges Senden mehrerer Knoten) die logische 0 gegen eine 1 immer durchsetzt.

2.3 Sicherungsschicht (Data Link Layer)

Master/Slave-Konzept: Ein LIN-Netzwerk enthält stets ein *Master*-Steuergerät, das die Kommunikation initialisiert und kontrolliert, sowie maximal 64 *Slaves*, wobei nicht mehr als 15 empfohlen werden [3]. Auf dem Master-Steuergerät wird die *Master-Task* sowie eine *Slave-Task*, auf den Slave-Steuergeräten eine *Slave-Task* ausgeführt. Diese Tasks stellen folgende Funktionen dar:

Master-Task

- sendet Header (Sync-Break, Sync-Byte und Identifier)
- sendet und empfängt Wakeup-Befehle
- synchronisiert den Bus

Slave-Task

- wartet auf Sync-Break
- synchronisiert sich auf Sync-Byte
- wertet Identifier aus und empfängt oder sendet Daten
- kann Wakeup-Befehle senden

Kommunikationskonzept: Das LIN-Protokoll sieht vor, dass die Master-Task in regelmäßigen Abständen gemäß einer Schedule-Table *Header* auf dem Bus sendet, die unter anderem einen Identifier (ID) enthalten.

Jede der 64 möglichen IDs adressiert dabei eine Nachricht. Erkennt eine Slave-Task eine ID, deren Botschaft von ihr zu senden ist, sendet sie unmittelbar nach dem Header der Master-Task ihre *Response*:

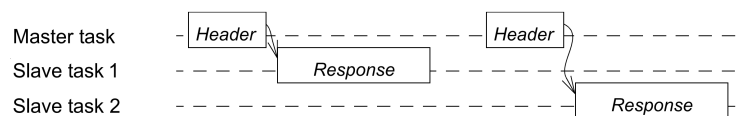


Abbildung 2.2: Header und Response in zeitlicher Abfolge [2]

Frameformat: Wie bereits bei der Darstellung des Kommunikationskonzepts beschrieben ist ein LIN-Frame grundsätzlich aus 2 Teilen aufgebaut, dem Header der Master-Task und der Response der Slave-Task.

Der Header enthält einen *Sync-Break* (mindestens 13 dominante Bit → Rahmenverletzung) zur Erkennung eines Rahmenanfangs, ein *Sync-Byte* zur Synchronisierung der Teilnehmer und zur automatischen Erkennung der Baudrate sowie einen *Protected Identifier*. Dieser Protected Identifier enthält eine 6 bit lange *ID* sowie zwei Paritätsbits zur Absicherung.

Die Response enthält 0...8 Datenbytes, gefolgt von einer Checksumme. Diese Checksumme wird (seit Protokoll-Spec. 2.1) über die Protected-ID sowie über alle Datenbytes berechnet.

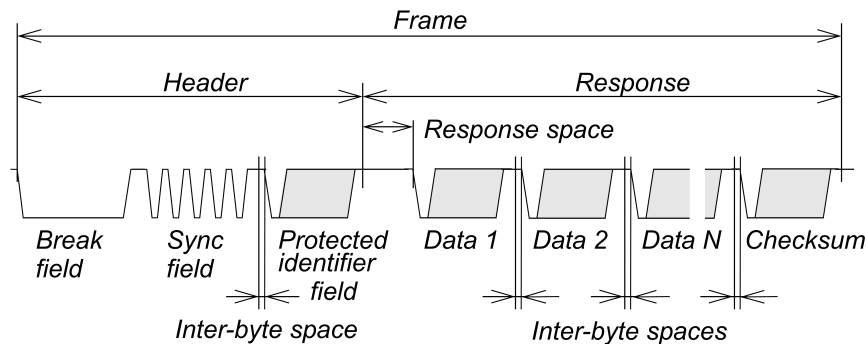


Abbildung 2.3: LIN-Frame [2]

Die so übertragenen Bytes sind UART-kompatibel mit einem Startbit (logische 0), 8 Datenbits und einem Stoppbit (logische 1) codiert.

3 Konfiguration des ASC-Moduls

Zur Realisierung der LIN-Kommunikation wird das UART-Modul *ASC1* des xc161-Mikrocontrollers verwendet, das wie folgt zu konfigurieren ist (nicht genannte Einstellungen / Bits sind als `false` bzw. 0 zu verstehen):

3.1 Konfigurationsregister *ASC1_CON*

Im Konfigurationsregister des UART-Moduls, *ASC1_CON*, sind folgende Einstellungen zu setzen:

- Übertragungsmodus 8 Bit Daten, asynchron: Modus auf `0x001`
- Anzahl der Stopp Bits: 1 gemäß LIN-Spezifikation [2]
- Receiver Enabled Bit auf 1, um Empfänger zu aktivieren
- Frame Check Bit auf 1, um den Sync-Break am Frame-Anfang als Rahmenverletzung zu erkennen und im Interrupt behandeln zu können
- Baudratengenerator Run Bit auf 1, um den Baudraten-Timer zu starten

3.2 Konfiguration der Portpins

Die Pins P2.15 (NSLP-Pin des LIN-Transceivers) und P3.0 (TxD-Pin) sind als Ausgang zu konfigurieren. Außerdem ist im ALTSELOP3-Register die Alternativfunktion für Pin P3.0 einzustellen, um diesen als TxD-Pin verwenden zu können.

3.3 Generierung der Baudrate

Um die Baudrate einzustellen, ist der Reload-Wert des Baudraten-Timers nach folgender Formel zu ermitteln:

$$\text{ReloadWert} = \frac{f_{ASC}}{32 \cdot \text{Baudrate}} - 1$$

Bei einer Frequenz $f_{ASC} = 40\text{MHz}$ und der geforderten Baudrate von $19.2\frac{\text{kbit}}{\text{s}}$ ergibt sich somit ein Wert von 64, der in das Register ASC1.BG zu schreiben ist.

3.4 Interrupts

Um die in der Protokoll-Spezifikation geforderten Maximalzeiten *Response-Space* und *Inter-Byte-Space* (vgl. Abbildung 2.3) einzuhalten, wird die Kommunikation im Interrupt statt im Polling realisiert.

Hierzu sind folgende Interrupts zu konfigurieren:

- Error-Interrupt bei Frame-Check (vgl. Kapitel 3.1, Konfiguration des UART-Moduls) zur Erkennung des Sync-Breaks am Anfang eines Frames
- Receive-Interrupt bei Empfang eines Bytes, um es sofort lesen zu können
- Transmit-Interrupt bei vollständigem Senden eines Bytes, um sofort das nächste Byte abschicken zu können

3.5 Initialisierung des LIN-Transceivers

Der LIN-Transceiver ist gemäß Datenblatt [1] aus dem *SLEEP*-Modus in den *NORMAL SLOPE*-Modus zu versetzen. Dies geschieht durch Anlegen eines High-Pegels auf dem TxD-Pin, gefolgt von einer High-Low-Flanke auf dem NSLP-Pin.

4 Implementierung

4.1 Datenstrukturen und Konstanten

Zur Speicherung von Sende/Empfangspuffer, Checksumme, Länge der zu sendenden Daten, Empfangszähler und aktuellem Index sowie der Nutzdaten wurden folgende Datenstrukturen definiert:

```
1 struct linMsg_type {
2     unsigned char index; // index in data-array
3     unsigned char rxcount; // Rx-Counter
4     unsigned char len; // length of bytes to send
5     unsigned int chk; // checksum
6     unsigned char data[11]; // Rx- & Tx-data
7 };
8 struct linData_type {
9     unsigned char temperature_lsb;
10    unsigned char temperature_msb;
11    unsigned char freq_1;
12    unsigned char freq_2;
13    unsigned char freq_3;
14    unsigned char freq_4;
15    unsigned char keyState_lsb;
16    unsigned char keyState_msb;
17 };
```

Listing 4.1: Verwendete Datenstrukturen

Außerdem wurden Konstanten für die im Identifier enthaltene Gruppennummer und Bot-schaftsnummern definiert:

```
1 #define LIN_ID_SLAVE 0x3
2 #define LIN_ID_TEMP (1 << 4)
3 #define LIN_ID_FREQ (1 << 5)
4 #define LIN_ID_KEY (1 << 4) | (1 << 5)
```

Listing 4.2: Verwendete Konstanten

4.2 Bereitstellen der Daten

Die zu sendenden Daten werden in der Polling-Schleife des Hauptprogramms erzeugt und in der globalen Struktur `linData` gespeichert. Um die Konsistenz dieser Daten zu gewährleisten, ist während dieses Schreibvorgangs der Empfangs-Interrupt zu maskieren.

```
1 // ***** save variables to linData *****
2 ASC1_RIC_IE = 0; // disable Rec-Interrupt
3 linData.temperature_lsb = (unsigned char) ((temp & 0x00FF) >> 0);
4 linData.temperature_msb = (unsigned char) ((temp & 0xFF00) >> 8);
5 linData.freq_1 = (unsigned char) ((frequency & 0x000000FF) >> 0);
6 linData.freq_2 = (unsigned char) ((frequency & 0x0000FF00) >> 8);
7 linData.freq_3 = (unsigned char) ((frequency & 0x00FF0000) >> 16);
8 linData.freq_4 = (unsigned char) ((frequency & 0xFF000000) >> 24);
9 linData.keyState_lsb = (unsigned char) ((keyState & 0x00FF) >> 0);
10 linData.keyState_msb = (unsigned char) ((keyState & 0xFF00) >> 8);
11 ASC1_RIC_IE = 1; // enable Rec-Interrupt
```

Listing 4.3: Bereitstellung der Nutzdaten

4.3 Erkennung des Rahmen-Anfangs

Bei Erkennung eines Sync-Breaks durch Rahmenverletzung werden die internen Zähler der `linMsg`-Struktur auf null gesetzt.

```
1 void ASC1_Error (void) interrupt 0x4A {
2     linMsg.index = 0;
3     linMsg.len = 0;
4     linMsg.rxcount = 0;
5 }
```

Listing 4.4: Error-Interrupt-Servicefunktion

4.4 Empfangen des Headers

Bei Empfang eines Bytes wird dieses zwischengespeichert und der Empfangszähler inkrementiert.

Wurde das 2. Byte nach dem Sync-Break empfangen, wird dieses als Identifier interpretiert und bei Übereinstimmung mit der eigenen Gruppennummer das Senden vorbereitet.

Hierzu werden die vom Master angeforderten Daten aus der globalen Datenstruktur gelesen und in einem Sendepuffer gespeichert. Anschließend wird mit dem Senden des ersten Bytes und der Berechnung der Checksumme begonnen.

```
1 void ASC1_Rx (void) interrupt 0x49 {
2   linMsg.data[linMsg.index] = (unsigned char) ASC1_RBUF;
3   linMsg.rxcount ++;
4   if((linMsg.index == 0) && (linMsg.rxcount == 3)) {
5     // 3rd byte of frame is identifier
6     if ((linMsg.data[0] & 0x0F) == LIN_ID_SLAVE) {
7       // group-no. in identifier
8       linMsg.index ++;
9       switch (linMsg.data[0] & 0x3F) {
10        case LIN_ID_TEMP : // send temperature
11          linMsg.data[1] = linData.temperature_lsb;
12          linMsg.data[2] = linData.temperature_msb;
13          linMsg.len += 2;
14          break;
15        case LIN_ID_FREQ : // send frequency
16          linMsg.data[1] = linData.freq_1;
17          linMsg.data[2] = linData.freq_2;
18          linMsg.data[3] = linData.freq_3;
19          linMsg.data[4] = linData.freq_4;
20          linMsg.len += 4;
21          break;
22        case LIN_ID_KEY : // send keyState
23          linMsg.data[1] = linData.keyState_lsb;
24          linMsg.data[2] = linData.keyState_msb;
25          linMsg.len += 2;
26          break;
27      }
```

```
28     linMsg.chk = linMsg.data[0]; // begin checksum with id
29     // send first byte:
30     ASC1_TBUF = linMsg.data[1];
31     lin_calc_chk ();
32 }
33 }
34 }
```

Listing 4.5: Rx-Interrupt-Servicefunktion

4.5 Senden der Daten

Wurde ein Byte fertig übertragen, so wird ein Interrupt ausgelöst, in dessen Servicefunktion das nächste Byte (sofern vorhanden) und am Ende die Checksumme gesendet wird.

```
1 void ASC1_Tx (void) interrupt 0x48 {
2     if (linMsg.index < linMsg.len) { // send next byte
3         linMsg.index ++;
4         ASC1_TBUF = linMsg.data[linMsg.index];
5         lin_calc_chk ();
6     }
7     else if (linMsg.index == linMsg.len) { // send inverted checksum
8         linMsg.index ++;
9         ASC1_TBUF = ~(linMsg.chk);
10    }
11 }
```

Listing 4.6: Tx-Interrupt-Servicefunktion

4.6 Berechnung der Checksumme

Die Checksumme wird fortlaufend bei Versand eines Bytes berechnet.

```
1 void lin_calc_chk (void) {
2     extern struct linMsg_type linMsg;
3     linMsg.chk += linMsg.data[linMsg.index];
4     linMsg.chk = (linMsg.chk & 0x00FF) + (linMsg.chk >> 8);
5 }
```

Listing 4.7: Funktion zur Berechnung der Checksumme

5 Ergebnis

Neben funktionalen Kriterien (Versand der korrekten Daten, zum Header passende Response, richtige Checksumme) ist es beim LIN-Bus auch obligatorisch, die maximale Framelänge einzuhalten. Diese liegt gemäß LIN-Protokoll-Spezifikation [2] beim 1.4-fachen der nominalen Zeitdauer, d.h. der kürzesten Zeitdauer einer Nachricht.

Die reale Zeitdauer verlängert sich hauptsächlich durch den Zeitbedarf zur Auswertung des Headers (Zeit zwischen Header und Response, *response-space* in Abbildung 2.3) sowie durch Wartezeiten zwischen dem Versand der einzelnen Datenbytes (*inter-byte-space* in Abbildung 2.3).

Durch die Implementierung der Kommunikation im Interrupt-Betrieb ergeben sich minimale Reaktionszeiten auf die Ereignisse „Header empfangen“ und „Byte komplett gesendet“. Dadurch werden die *inter-byte-spaces* in erster Näherung null und der *response-space* besteht lediglich aus der Zeit, die zur Auswertung des Headers und Schreiben des Sendepuffers erforderlich ist. Wie in Abbildung 5.1 zu sehen ist, liegt diese Zeit bei ca. $\frac{1}{2}$ Bitzeit bzw. $25\mu\text{s}$. Die reale Framelänge ist somit lediglich um den Faktor 1.006 größer als die nominelle Framelänge, maximal zulässig wäre dagegen ein Faktor von 1.4.



Abbildung 5.1: LIN-Frame (ID 0x23) mit vernachlässigbaren *inter-byte-spaces*

Literatur

- [1] *TJA1020 LIN transceiver - DATA SHEET*. Philips Semiconductors, 2002.
- [2] *LIN Specification Package (Revision 2.1)*. LIN Consortium, 2006.
- [3] *Serielle Fahrzeugbussysteme - Skript zur Vorlesung*. FH-Ingolstadt, 2009.

Abbildungsverzeichnis

2.1	Buspegel am Sender und Empfänger [2]	4
2.2	Header und Response in zeitlicher Abfolge [2]	5
2.3	LIN-Frame [2]	6
5.1	LIN-Frame (ID 0x23) mit vernachlässigbaren <i>inter-byte-spaces</i>	12

Listings

4.1	Verwendete Datenstrukturen	8
4.2	Verwendete Konstanten	8
4.3	Bereitstellung der Nutzdaten	9
4.4	Error-Interrupt-Servicefunktion	9
4.5	Rx-Interrupt-Servicefunktion	10
4.6	Tx-Interrupt-Servicefunktion	11
4.7	Funktion zur Berechnung der Checksumme	11