# Schnittstellen

## Serielle Schnittstelle, USB, Feldbusse, SPI, $I^2C$, 1-Wire etc.

**Jürgen Plate, 12. November 2017**

# Inhaltsverzeichnis

# 1

# Parallele Schnittstellen

Die E/A-Schnittstellen sind Bindeglied zwischen CPU und Peripherie. Sie sorgen unter anderem für Datentransport, zeitliche Anpasung, Signalumsetzung, Pegelumsetzung, Erweiterung der Belastbarkeit, Datenformat-Anpassung, Codeumsetzung etc.). Das Einsatzgebiet reicht von einfacher Byteorientierter Ein- und Ausgabe bis zur autonomen Verwaltung komplexer Protokolle, wie z. B. die Netzwerkanbindung. E/A-Komponenten werden in der Regel so entworfen, dass sie jeweils einen möglichst großen Bereich von Anwendungen abdecken. Die Spezialisierung für die Anwendung geschieht durch Initialisierung der Komponente im System, ein Vorgang, der in Datenblättern und im allgemeinen Sprachgebrauch „Programmierung“ genannt wird: Nach jedem Einschalten des Systems müssen bestimmte Steuerregister dieser Komponenten von der CPU aus mit bestimmten Werten initialisiert werden.

Nahezu jeder Controller verfügt über bitparallele Ein-/Ausgabe. Der Zugriff auf die Peripherie erfolgt entweder programmgesteuert (polling) oder interruptgesteuert. Auch Bus-Treiber können als sehr einfache E/A-Bausteine eingesetzt werden. So wurde z. B. die Standard-Druckerschnittstelle des IBM-PC mit zwei 8-Bit-Latches und zwei 8-Bit-Bustreibern realisiert (siehe auch *http://www.netzmafia.de/skripten/hardware/Parallel/*). Kaum komplexer sind auch Speicher-Register mit nachgeschalteten Leistungstreibern. Sie übernehmen auch die zeitliche Anpassung. Zum Teil sind sie mit Steuerlogik, Betriebsartenwahl (Ein/Aus) und Interrupt-Auslösung versehen.

Wesentlich flexibler im Einsatz sind programmierbare E/A-Komponenten („Ports“, deren Eigenschaften und Funktion sind per Software einstellbar ist. Programmierbare Eigenschaften können sein:

- Datenrichtung
- Übertragungsgeschwindigkeit
- Übertragungsformat
- Paritätsprüfung
- Art des Interrupt-Signals von der Peripherie
- Interrupt-Maske

Diese Ports erlauben meist sowohl programmgesteuerten E/A-Transfer als auch interruptgesteuerten E/A-Transfer. Für den einfachen E/A-Transfer gibt es zwei Grundtypen:

- Schnittstelle für parallelen Datenaustausch. In der Regel sind mehr als ein 8 Bit breiter Port vorhanden (meist zwei bis vier). Steuer- oder Statusregister sind meist pro Port separat vorhanden.

  Es existieren jedoch auch parallele Schnittstellen, die aus einem Gerät herausgeführt und standardisiert sind. Dazu gehören unter anderem der IEEE-Bus, der noch oft zur Ansteuerung von Messgeräten verwendet wird oder die parallele Druckerschnittstelle.

- Schnittstelle für bitseriellen Datenverkehr. Diese Komponenten enthalten je einen Parallel-Seriell- und Seriell-Parallel-Wandler (Schieberegister). SIe werden im folgenden Kapitel behandelt.

## 1.1 Grundlagen der Ansteuerung von Parallel-Schnittstellen

### 1.1.1 Digitalausgänge

Über Digitalausgänge werden beispielsweise Relais, Schütze, Anzeige- und Warnlampen geschaltet, Displays angesteuert und Geräte und Maschinen aller Art unter die Kontrolle des Mikrocomputers gebracht. Digitalausgänge gibt es bezüglich ihres Zeitverhalten in unterschiedlichen Formen:

- Statische Ausgänge, die über Register gepuffert sind. Durch den Ausgabebefehl wird ein n-Bit-Register geladen, dessen Inhalt in Form von Digitalsignalen nach außen übergeben wird.
- Pulsausgang mit festgelegter oder programmierbarer Dauer. Hier löst die Ausgabe einer 1 die Erzeugung eines Impulses aus, dessen Dauer durch die Programmierung oder durch die Hardware (Taktfrequenz) vorgegeben ist. Oft wird diese Form der digitalen Ausgabe durch die Anwendung von zwei Befehlen auf einen statischen Ausgang simuliert (Sequenz Bit ein - Bit aus).
- Pulsfolgen: Am Digitalausgang entsteht eine programmierbare Anzahl von Impulsen. Pulsfolgen werden im allgemeinen ebenfalls unter Programmkontrolle auf statischen Ausgängen erzeugt.

Ein wichtiger Parameter zur Beschreibung von Digitalausgängen ist die physikalische Spezifikation der Ausgangssignale. Dabei sind folgende Formen gebräuchlich:

- **Spannungsausgang**, der zunächst dem logischen Pegel des Rechnersystems (TTL, CMOS, ...) entspricht, der dann aber über Pegelkonverter an die entsprechenden Prozesseingänge angepaßt werden kann.
- **ECO-Ausgang** (ECO = Electronic-Contact), der einem gesteuerten Transistor mit offenem Kollektor entspricht. Ein *offener Kontakt* des ECO ist durch eine maximale zulässige Spannung und einen Reststrom spezifiziert. Der *geschlossene Kontakt* des ECO ergibt sich aus einer Kollektor-Restspannung und einer maximal zulässigen Strombelastung.

### 1.1.2 Digitaleingänge

Digitaleingänge dienen der direkten Eingabe digitaler Information und übernehmen vielfältige Aufgaben, darunter die Abfrage von Kontakten (Endkontakte, Tasten, Maschinenzustände, Stückzähler, Inkrementalweggeber...) und die Übernahme von digitalen Meßergebnissen (z. B. Digitalvoltmeter, Digitalzähler, Waagenergebnisse).

Auch Digitaleingänge werden unter Programmsteuerung abgefragt. Sie sind in Gruppen organisiert, die der Breite der Parallel-E/A-Bausteine entsprechen. Mit jedem Eingabebefehl wird die ganze Gruppe eingelesen. In seltenen Fällen ist die Datenrate so groß, daß ein DMA-Anschluß der Digitaleingabegruppe erforderlich wird.

Im Gegensatz zu den Digitalausgängen gibt es recht unterschiedliche Typen von Digitaleingängen:

- **Direkteingänge:** Die Zustände der Eingangsleitungen werden hier direkt übernommen. Ihr Zustand im Zeitpunkt des Lese-Befehls ist maßgebend (LD- oder IN-Befehl).
- **Gepufferte Eingänge:** Es gibt mehrere Arten der Pufferung, beispielsweise kann eine ganze Digitaleingabegruppe durch ein externes oder internes Übernahmesignal in ein Register abgespeichert werden. Hier ist der Zustand der Dateneingabeleitungen nicht zum Zeitpunkt der Abfrage durch das Programm, sondern zum Zeitpunkt des Taktes maßgebend.
- **Dynamische Eingänge:** Asynchron an den einzelnen Eingängen ankommende Ereignisse (Flanken oder Pulse) werden gespeichert; die betroffenen Speicher-Flipflops müssen diese Information solange behalten, bis sie vom Rechner übernommen werden und dort zu entsprechenden Aktionen führen. Da die Flip-Flops voneinander unabhängig zu beliebigen Zeitpunkten gesetzt werden können, stellt das Rücksetzen ein gewisses Problem dar: Solle genau in der Zeit zwischen Lesen der Eingangsinfo und dem Löschen der Flipflops eine neuer Impils eintreffen, würde dieser „übersehen“. Dies führt zu sporadischen Fehlern, die nur sehr schwer zu finden sind (wenn überhaupt!). Es dürfen also nur die Flipflops gelöscht werden, deren 1-Zustand bereits vom Rechner übernommen wurde. Dies ist nur möglich, wenn man die Ereignisse doppelt puffert. Eine Prinzipschaltung zeigt Bild 1.1.

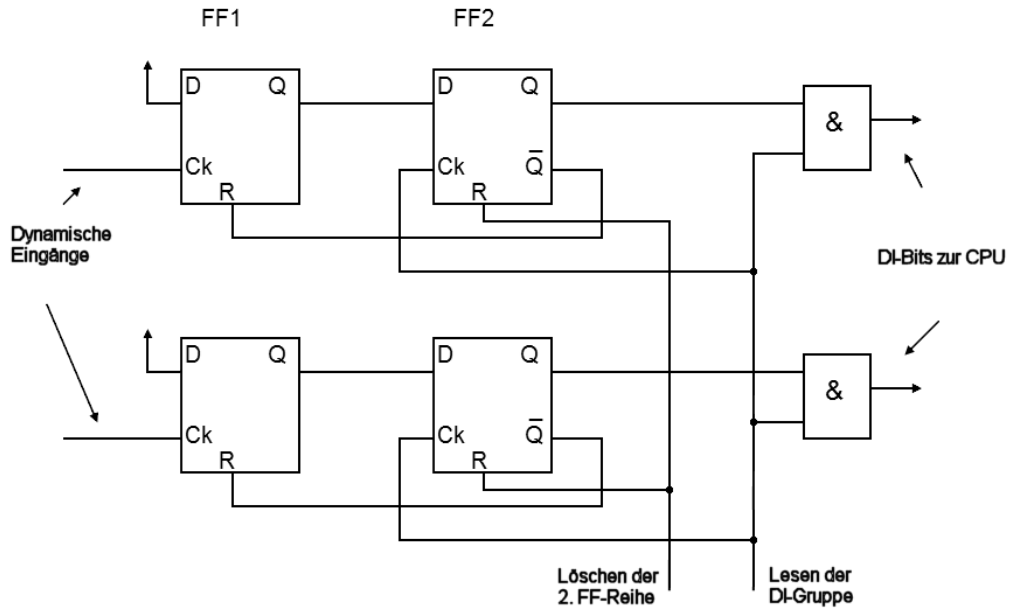



**Bild 1.1:** Doppelt gepufferte Impulseingänge

Mit der steigenden Flanke des Lese-Signals wird der Inhalt des eigentlichen Ereignisspeichers in die zweite Flipflop-Reihe übernommen. Der invertierte Ausgang der Zwischenspeicher wird zum Rücksetzen der Ereignisspeicher benutzt. Am Ende des Lesezyklus muss der Zwischenspeicher wieder gelöscht werden, damit das nächste Ereignis den Ereignisspeicher setzen kann.

- **Zähleingänge:** dienen dem Zählen von Ereignissen. Dabei sind folgende Realisierungen möglich:
  - Verwendung direkter Digitaleingänge und Abfrage per Programm in bestimmten Zeitintervallen (die Periode muß kleiner sein als der halbe minimale Ereignisabstand).
  - Verwendung von Interrupteingängen: Jedes Ereignis erzeugt einen Interrupt. DIe Interrupt service Routine inkrementiert einen Softwarezähler.
  - Verwendung eines Hardwarezählers, der das Zählen von sehr schnellen Ereignissen erlaubt. Häufig wird der Zählerüberlauf dazu benutzt, eine Programmunterbrechung auszulösen.

### 1.1.3 Ansteuerung

Grundsätzlich gibt es mehrere Möglichkeiten der Ansteuerung von Schnittstellen:

- **Programmgesteuerter E/A-Transfer (Polling):**
  Einfache E/A-Schnittstellen für programmierten E/A-Transfer bestehen wesentlich aus einem E/A-Datenregister (EADR), einem E/A-Steuer- und Statusregister (EASR) sowie der Adreß- und Steuerlogik . Das EADR dient zur Zwischenspeicherung des auszugebenden bzw. einzulesenden Datenwerts (siehe oben). Neben den zu übertragenden Daten braucht die Schnittstelle Signale zur richtigen Abwicklung des Datentransfers:
  - Statussignale von der Peripherie, z. B. Anzeige, dass das Peripheriegerät einen neuen Wert ins EADR geliefert hat oder bereit ist, den nächsten Wert zu übernehmen.
  - Steuersignale zur Peripherie, z. B. Anzeige, dass der vorhergehende Datenwert gelesen wurde und die Peripherie einen neuen Wert ins EADR einschreiben kann.
  - Steuersignale zur Schnittstelle selbst, z. B. Signale zum Einstellen bestimmter Betriebsmodi bei programmierbaren Schnittstellen.

  Diese Signale werden – soweit sie für längere Zeit zur Verfügung stehen müssen – im EASR gespeichert. Das EASR kann auch aus separaten Registern für Status- und Steuersignale bestehen. Im allgemeinen erhalten EADR und EASR eine eigene, separate Adresse. Die gesamte Initiative zur E/A geht vom Programm aus. Zeigt der Inhalt des EASR an, dass keine neuen Daten im EADR

stehen, muss das Programm erneut das EASR abfragen (Warteschleife, „Polling“). Dabei können zwei Aspekte Probleme verursachen: Ist die Zeit, die das Programm braucht, um einen Eingabewert zu bearbeiten, länger als die Zeitspanne innerhalb der sich das Eingangssignal ändern kann, werden Ereignisse „übersehen“. Bei „langsamen“ Peripheriegeräten ist der Computer während der Übertragung eines Datenblocks die meiste Zeit mit Abfragen bschäftigt, die in den meisten Fällen negativ ausfallen.

- **Direktspeicherzugriff (DMA = Direct Memory Access):**
  Beim DMA steuert die E/A-Schnittstelle den Transfer nach Anstoß durch die CPU selbstständig und ohne Zuhilfenahme der CPU Datentransport zwischen Speicher und Peripherie unter Umgehung der CPU. Selbst, wen die CPU in dieser Zeit nichts anderes tut, ist ein wesentlich schnellerer Datentransport möglich (weitgehender Wegfall der Verwaltungsarbeit). Daher ist DMA für den schnellen Transfer großer Datenmengen besonders geeignet. Zum Anstoßen des DMA-Transfers übermittelt die CPU der DMA-Schnittstelle:

  - die Anfangsadresse (im Arbeitsspeicher) des zu übertragenden Datenblocks
  - die Länge des zu übertragenden Datenblocks
  - Gegebenenfalls muss noch, wie beim programmierten Transfer, das EASR entsprechend geladen werden Initialisierung der Schnittstelle.

  Die DMA-Schnittstelle benötigt zwei weitere Register, das E/A-Adreßregister (EAAR) und den E/A-Blocklängenzähler (EAZR). Es ergibt sich eine Konfiguration wie in Bild 1.2.

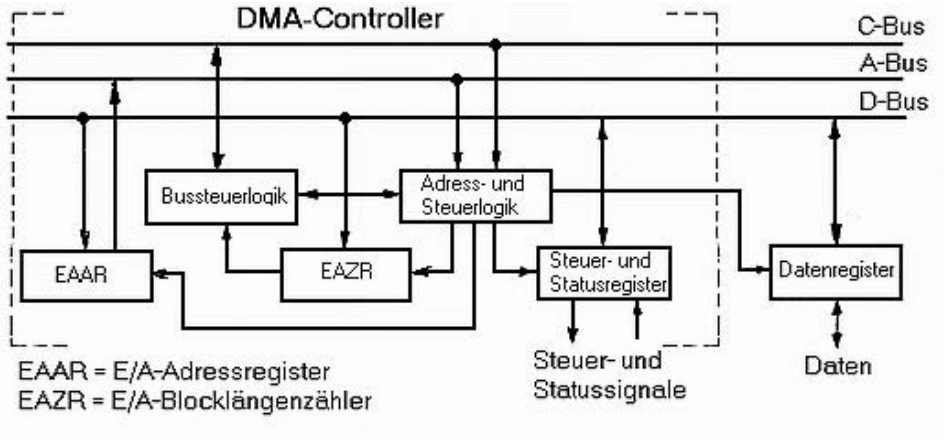



**Bild 1.2:** Prinzipieller Aufbau einer DMA-Schnittstelle

Der Ablauf eines DMA-Transfers besteht aus drei Schritten:

1. Initialisieren der DMA-Schnittstelle
2. Bus-Freigabe-Anforderung an die CPU (DMA-Request)
3. CPU bestätigt Bus-Freigabe (DMA-Acknowledge) (Die Bus-Freigabe der CPU geschieht durch Umschalten der Bustreiber-Ausgänge in den hochohmigen Zustand)

Nun bedient die DMA-Schnittstelle den Adress-, Daten- und Steuerbus auf die gleiche Art und Weise, wie die CPU. Der Inhalt des EAAR wird an den Adressbus gelegt und die für die Speichersteuerung notwendigen Signale auf den Steuerbus. Das EADR wird mit dem Datenbus verbunden (Bild 1.3).

- **Unterbrechungsgesteuerter E/A-Transfer:**
  Die E/A-Befehlsfolge wird durch einen von der E/A-Schnittstelle ausgelösten Interrupt gestartet. Der Interrupt wird von der Schnittstelle abgesetzt, wenn ein Datenwort empfangen wurde und im EADR steht (bzw. wenn ein Datenwort aus dem EADR abgeholt wurde). Die CPU braucht nicht in einer Programmschleife zu warten, sondern kann in der Zwischenzeit ein anderes Programm bearbeiten, das durch den Interrupt kurzzeitig unterbrochen wird. Der interruptgesteuerte Transfer gestattet zudem die quasi-simultane Bedienung mehrerer langsamer Schnittstellen. Zur Realisierung muß eine Interrupt-Leitung von der E/A-Schnittstelle (EASR) zum Unterbrechungswerk der CPU vorhanden sein.

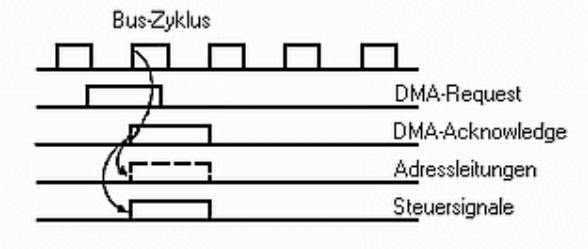



**Bild 1.3:** Prinzipieller Ablauf des DMA-Zyklus

Vorteil des programmierten Transfers: Sehr einfache Schnittstellen, die übersichtlich und leicht zu verwenden sind.

Nachteil des programmierten Transfers: Für den Transfer jedes einzelnen Datenworts wird die CPU benötigt. Diese muss dabei relativ viel Verwaltungsaufwand leisten (Laden/lesen EASR, Interrupt-Behandlung) die maximale Datenrate ist sehr begrenzt.

Durch den Interupt wird auch der Befehlszyklus erweitert. Nach Abarbeitung des Befehls wird das Vorhandensein eines Interrupts geprüft und ggebenenfalls in die Interrupt-Serviceroutine (ISR) verzweigt. Dabei werden die Register ganz oder teilweise auf dem Stack abgelegt und beim Beenden der ISR wieder restauriert.

## 1.2 Der IEC-Bus (IEEE-Bus)

Ein sehr bekanntes Bussystem zur Kopplung von Messgeräten an einen Rechner wurde schon Mitte der 70er Jahre von der Fa. Hewlett Packard als „HP-Interface-Bus“ auf den Markt gebracht. Nach geringfügiger Überarbeitung wurde er später mehrfach genormt und ist unter folgenden Bezeichnungen bekannt:

- IEC-Bus (internationale Norm IEC 625)
- IEEE-Bus (amerikanische Norm IEEE 488)
- HP-IB (Hewlett Packard Interface Bus)
- GPIB (General Purpose Interface Bus)

Der einzige relevante Unterschied besteht jedoch nur im Stecker. Laut IEC-Norm wird ein 25-poliger SUB-D-Stecker verwendet, nach der IEEE-Norm kommt ein 24-poliger AMP-Stecker zum Einsatz. Da die Geräte alle am Bus hängen, haben die Verbindungskabel an jedem Ende eine Stecker-Buchse-Kombination.

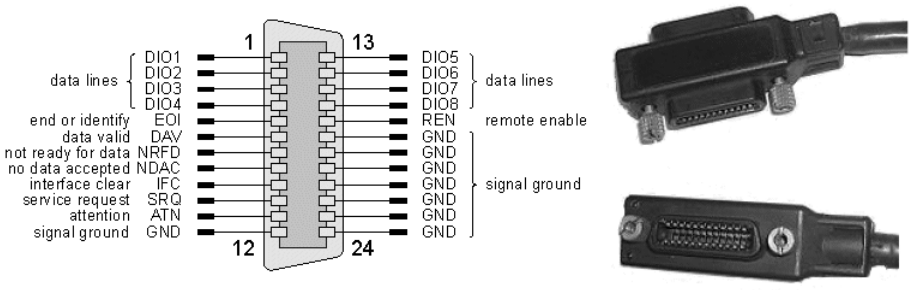



**Bild 1.4:** IEEE-Stecker mit Belegung

Nur durch die sehr strenge und genaue Definition des gesamten Busses, angefangen von der Stecker-Bauart und Signalbelegung bis hin zu den höheren Protokollen und sogar Programm-Bibliotheksroutinen zur Ansteuerung der angeschlossenen Geräte, konnte die volle Kompatibilität von IEC-Bus-fähigen Komponenten sichergestellt werden. Der IEC-Bus weist die typischen Merkmale eines sogenannten Laborbusses auf:

- Er verbindet programmierbare und nicht-programmierbare elektronische Meßgeräte zur Zusammenstellung von kompletten automatischen Meßsystemen.
- Es sollen maximal 15 Geräte über eine Gesamtlänge von max. 20 m verbunden werden.
- Die Übertragungsrate ist auf 1MBit/s beschränkt.
- Es werden Umgebungsbegingungen vorausgesetzt, wie sie in Laboratorien und Prüffeldern herrschen.
- Über eine relativ einfache Steckverbindung werden nur wenige Signale transportiert: acht Daten-, drei Handshake- und fünf Steuersignale.
- Um den Aufwand für die Teilnehmerschnittstellen zu begrenzen wurden drei verschiedene Teilnehmer-Grundfunktionen geschaffen. Jeder der Teilnehmer kann eine oder mehrere dieser Grundfunktionen besitzen.

Heute gibt es kaum mehr ein Meßgerät, das nicht zumindest die Nachrüstmöglichkeit für einen IEC-Bus-Anschluß besitzt. Als Steuereinheit wird in der Regel ein Computer mit IEC-Bus-Schnittstelle verwendet, an den die einzelnen Geräte angeschlossen werden. Es lassen sich bis zu 15 Geräte zu einem Meßplatz zusammenschalten, wobei die Länge der einzelnen Leitungen auf zwei Meter begrenzt ist. Der Bus arbeitet mit TTL-Pegeln, die Kopplung zum Gerät erfolgt über spezielle Treiberbausteine.

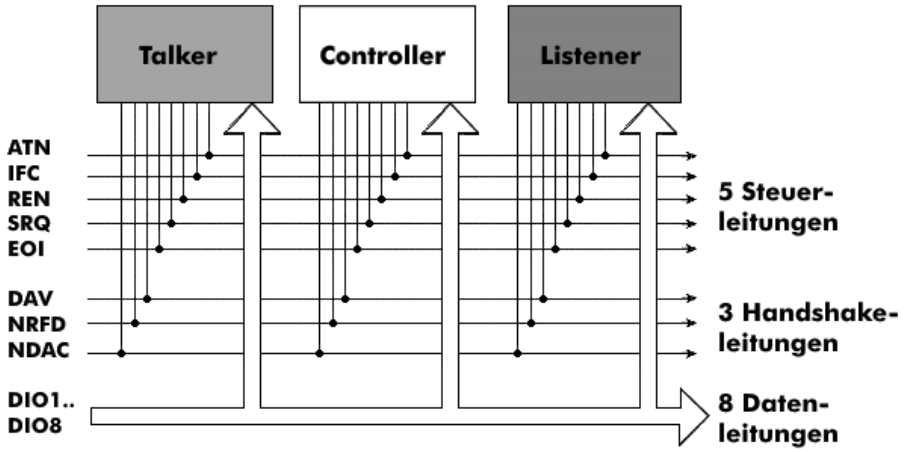



**Bild 1.5:** Aufbau des IEEE- oder IEC-Busses

Am IEEE-Bus (Bild 1.5) können drei verschiedene Gerätetypen aktiv sein: der Controller, der Listener (Hörer) und der Talker (Sprecher). Die Funktionen können beliebig von den angeschlossenen Geräten übernommen werden, sofern dies von der Implementierung der jeweiligen Schnittstelle erlaubt und sinnvoll ist. Es kann jedoch zu einer Zeit nur jeweils ein Controller aktiv sein. Der Controller überwacht die Kommunikation im Bus in ähnlicher Weise wie eine Telefonvermittlung: Bemerkt er, dass ein Gerät eine Datennachricht senden will, so wird dieser Talker mit dem entsprechenden Listener verbunden. Meist werden dabei durch den Controller Talker und Listener adressiert, noch bevor der Talker seine Nachricht zum Listener absetzen kann. Nach Abschluß dieser einzelnen Übertragung werden beide Geräte gewöhnlich wieder entadressiert. Gegebenenfalls ist für einige Buskonfigurationen kein Controller erforderlich; es gibt Geräte, die nur als Talker oder Listener arbeiten. Ein Controller ist aber immer dann notwendig, wenn der aktive oder adressierte Talker oder Listener geändert werden muss. Die Controllerfunktion wird in der Regel vom Computer wahrgenommen. Der aktive Controller kann von sich aus die Kontrolle einem anderen Gerät zuweisen, das dann die Funktion des Controllers übernimmt.

Beim IEEE-Bus handelt es sich um eine bit-parallele (byte-serielle) Kommunikationsschnittstelle mit acht Bit Wortbreite, die Datenübertragungsraten bis zu 1 MByte/s ermöglicht. Jedes Gerät, das am IEEE-Bus angeschlossen ist, hat eine eindeutige Adresse, die durch DIP-Switches an der jeweiligen Schnittstelle eingestellt wird. Über diese Adresse wird jedes Gerät angesprochen. Grundsätzlich hören alle Geräte auf dem Bus mit, wobei nur das angesprochene Gerät reagiert. Daher sind die Bustreiber mit Open-Collector-Ausgängen versehen. Im Ruhezustand liegen die Leitungen auf 1-Pegel

| | | |
|---|---|---|
| DIO1-DIO8 | Datenleitungen | Auf den acht bidirektionalen Datenleitungen werden die Daten transportiert. |
| NDAC | Not Data Accepted | Handshake-Leitung. Mit NDAC teilen die Listener mit, dass sie das Datenbyte auf den DIO-Leitungen noch nicht übernommen haben. |
| DAV | Data Valid | Handshake-Leitung. Die Daten auf DIO1-DIO8 sind gültig. Dieses Signal wird vom Talker eine kurze Zeit nach dem Anlegen der Daten aktiviert. |
| NRFD | Not Ready For Data | Handshake-Leitung. Listener teilen mit diesem Signal mit, dass sie das Datenbyte auf DIO1-8 noch nicht verarbeitet haben. |
| ATN | Attention | Wird vom Controller gesendet, um alle Talker zu unterbrechen und den Bus freizugeben. Die Datenleitungen enthalten ein Kommandobyte (z. B. eine Adresse). Wird zusammen mit EOI für Parallelabfragen (parallel poll) verwendet. |
| EOI | End Or Identify | Wird vom Controller oder Talker gesetzt, um anzuzeigen, dass die Datenübertragung zu Ende ist. |
| IFC | Interface Clear | Der Systemcontroller kann mit dieser Leitung den Bus zurücksetzen und sich als aktiver Controller etablieren. |
| REN | Remote Enable | Wird vom Controller gesetzt, um die Kontrolle an einen anderen Controller zu übergeben. Wird REN deaktiviert, gehen alle Teilnehmer zurück in den Lokalmodus. |
| SRQ | Service Request | Busteilnehmer können über dieses Signal dem aktiven Controller mitteilen, dass sie bedient werden wollen – etwa die Funktion einer Interrupt-Leitung bei einem Mikroprozessor. |

**Bild 1.6:** Signale des IEEE- oder IEC-Busses

(TTL, ca. 5 V) und werden gegebenenfalls auf 0-Pegel gezogen (active Low, wired-or für Low-Pegel). Die Bedeutung der Signale ist in Bild 1.6 aufgeführt.

Die Datenübertragung erfolgt in einzelnen Schritten. Das Handshake-Verfahren erlaubt, dass mehrere Listener von einem Talker Daten übernehmen können. Der Talker muss warten, bis alle Listener das Signal NRFD freigegeben haben. Dann kann er Daten auf die Datenleitungen legen und das Signal DAV aktivieren. Wenn alle Listener die Daten übernommen haben, wird das Signal NDAC freigegeben, und das nächste Datenbyte kann gesendet werden, wenn alle Listener das Signal NRFD freigegeben haben.

1. Die Datenquelle (z. B. Talker) wartet, bis NRFD=1 ist.
2. Die Datenquelle legt das erste Datenbyte auf die Datenleitungen DIO1 - DIO8.
3. Die Datenquelle zieht DAV auf 0.
4. Die Datenempfänger legen NRFD auf 0 (nicht bereit für weitere Daten).
5. Entsprechend ihrer Verarbeitungsgeschwindigkeit setzen die Empfänger ihren NDAC-Ausgang auf 1. Sind alle fertig, geht NDAC auf 1.
6. Sobald alle Datenempfänger die Daten intern verarbeitet haben, setzen sie Ihren NRFD-Ausgang auf 1. Sind alle fertig, geht NRFD auf 1. Die Datenquelle setzt DAV wieder auf 1.

Die Signale NRFD und NDAC werden, wie gesagt, über ein *wired or* erzeugt; damit werden die beiden Signale erst frei gegeben, wenn auch das letzte (langsamste) Gerät bereit für neue Daten ist (NRFD), bzw. die Daten übernommen hat (NDAC).

In Bild 1.7 ist gezeigt, wie eine typische Datenübertragung eingeleitet wird:

- Das vom Talker aktivierte ATN (Attention)-Signal zeigt an, daß zunächst Adressen (von Listener und Talker) übertragen werden,
- Liegt das Signal NRFD (not ready for data) auf **high** (also ready), so beginnt der Talker mit dem Anlegen der Listener-Adresse auf den Datenleitungen und zeigt die Gültigkeit des Datenbusses mit DAV (Data available) an.
- Der Listener zieht daraufhin das Signal NRFD auf „not ready“ und signalisiert die Übernahme der Datenbusinformation mit einem NDAC (not data accepted)-Signal, das er nach „accepted“ schaltet.
- Der Handshake ist abgeschlossen, wenn der Talker daraufhin das DAV-Signal deaktiviert, was wiederum vom Listener mit einem Deaktivieren des NDAC quittiert wird.
- Da beim Listener jetzt u. U. zeitraubende Aktivitäten nötig sind, wird er so lange das NRFD-Signal auf „not ready“ halten, bis er bereit ist, den nächsten Handshake (den Transport des nächsten Datenbytes) abzuwickeln.

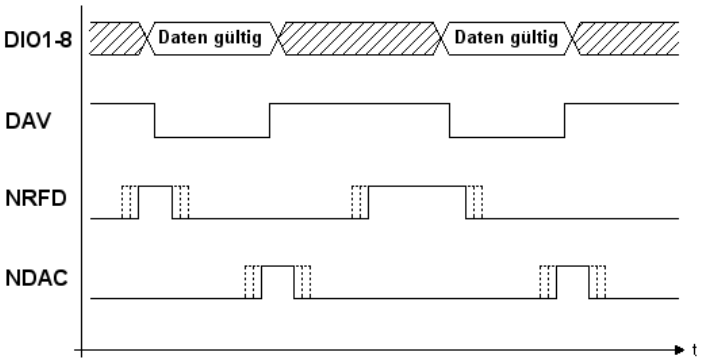



**Bild 1.7:** Datenübertragung auf dem IEEE-Bus

Die Schnittstellennachrichten bestehen aus adressierten Befehlen, Universal-Befehlen, die sich an mehrere Geräte richten, oder aus Adressen (Ansprechen eines bestimmten Geräts). Für die Geräteadressierung werden drei Typen von Adressen unterschieden (festgelegt durch die Bits 6 und 7):

```
Bedeutung    7 6 5 4 3 2 1 (Bitnummer)
Listener     0 1 x x x x x
Talker       1 0 x x x x x
Sekundär     1 1 x x x x x
Buskommando  0 0 x x x x x
```

Für die eigentliche Geräteadresse stehen dann 5 Bits (`x x x x x`) zur Verfügung, wobei für Primäradressen nur die Werte 0 bis 30 erlaubt sind. Innerhalb eines Gerätes können noch Unteradressen angesprochen werden (z. B. ein bestimmter Messkanal), deren Wertebereich auch wieder zwischen 0 und 30 liegen kann. Ein einzelnes Gerät darf bis zu sechs Unteradressen besitzen. Beim Multimeter Keithley 619 sieht die Aufteilung beispielsweise so aus (Adresse als ASCII-Zeichen):

```
Kanal A          "a"
Kanal B          "b"
Status Kanal A   "c"
Status Kanal B   "d"
```

Die Befehle vom Controller an die Geräte und Statusinformationen von den Geräten an den Controller sind in der Norm festgelegt. Welche Funktion jeweils an einem bestimmten Gerät ausgelöst wird, hängt von den Gerätespezifikationen ab.

Grundlage für die Programmierung von Busvorgängen und der angeschlossenen Geräte war zunächst nur eine Codetabelle, die eine Zuordnung des ASCII zu Adressen- und Befehlsgruppen vorsah. Bemühungen um eine Vereinheitlichung der Gerätebedienung führten schließlich zur Erweiterung des Busstandards, auf dessen Grundlage die „Standard Commands for Programmable Instruments“ (SCPI) zur Programmierung von Messgeräten entworfen wurden, die eine einheitliche Befehlsstruktur, Standardbefehle und definierte Datenformate beschreiben.

Bild 1.8 listet wesentliche allgemeingültige SCPI-Befehle auf. Sie sind durch das vorangestellte Zeichen „*“ erkennbar. Für die eigentliche Geräteprogrammierung wird eine Baumstruktur für Programmierbefehle benutzt (vom Bus-Systems als Daten mit ATN=Low übertragen). Die Befehle sind ASCII-Strings. Bei Abschluss mittels Semikolon können sie als ein verketteter String übertragen und im Gerät als Befehlsschlange abgearbeitet werden. Die einzelnen Befehlsebenen werden durch „:“ getrennt, ein anführender Doppelpunkt verweist auf die Wurzel. Die jeweils implementierten Strukturen und Befehle sind den zugehörigen Handbüchern der Geräte zu entnehmen.

| | | |
|---|---|---|
| *IDN? | Identification Query | Abfrage der Gerätekennung (Identitäts-String) |
| *RST | Reset Command | Rücksetzen in den Einschaltmodus der Gerätefunktion (geräteabhängig!) |
| *CLS | Clear Status Command | Löscht Ereignisbits im Statusregister |
| *TST? | Self-Test Query | Selbsttest-Abfrage: Fehlercode |
| *OPC | Operation Complete Command | Operationsende (setzt OPC-Bit) |
| *OPC? | Operation Complete Query | Abfrage Operationsende |
| *TRG | Trigger Command | Start der (vorprogrammierten) Aktion (z.B. Messung) |
| *WAI | Wait to Continue Command | Abarbeitung der Befehlsschlange erst nach Abschluß der vorherg. Befehle |
| *SAV {x} | Save Command | Speichert Geräteeinstellung {in x} |
| *RCL {x} | Recall Command | Reaktiviert Geräteeinstellung {aus x} |
| *ESE (wert) | Event Status Enable Command | Setzt Maske für Ereignisregister (wert = 0...255) |
| *ESE? | Event Status Enable Query | Abfrage des Maskenregisters (0...255) |
| *ESR? | Event Status Enable Query | Abfrage des Ereignisregisters (0...255) |
| *SRE (wert) | Service Request Enable Command | Setzt Maske für SRQ-auslösende Ereignisse (wert = 0...255) |
| *SRE? | Service Request Enable Query | Abfrage der SRQ-Maske (integer 0...255) |
| *STB? | Read Status Byte Query | Abfrage des Statusbytes (integer 0...255) |

**Bild 1.8:** Tabelle der SCPI-Befehle

Wie der Tabelle zu entnehmen ist, sind die Befehle mit unmittelbarer Rückantwort durch ein nachgesetztes Fragezeichen eindeutig gekennzeichnet. Generell gilt, dass gewünschte Daten explizit angefordert werden müssen. Nur dann werden sie im Gerät in den Ausgabe-Puffer übergeben und nur dann kann ein Lese-Vorgang erfolgreich gestartet werden.

Es ist auch immer sicher zu stellen, dass vom Gerät überhaupt lesbare Daten erzeugt und diese vom Controller (Listener) aufgenommen werden können. Fehlerquellen sind beispielsweise fehlende Betriebsbereitschaft von Geräten (ausgeschaltet, Interface nicht aktiviert), falsche Geräteadresse. fehlerhafte oder unvollständige Befehlsketten, unzureichende Empfangspuffer oder falsch eingestellte Messbedingungen.

Zum Schluß soll noch an einem Beispiel die Adressierung und Datenübertragung gezeigt werden. Es handelt sich um die Parametereinstellung eines Multimeters. Der Parameterstring lautet „F0R0X“, die Ergebnismeldung des Geräts „NDCV“. Zuerst die Sendung des Controllers an das Gerät:

```
Befehl    ASCII-Zeichen  Kommentar
          (sedezimal)

 UNL           3F        Unlisten (ATN=0) Gerät "ruhigstellen"
 UNZ           5F        Untalk (ATN=0)
  &            26        Primäradresse Listener (ATN=0)
  F            46        Einstellungen senden (ATN=1)
  0            30
  R            52
  0            30
  X            58
 <LF>          0A
 <CR>          0D        Ende des Blocks (EOI wechselt auf 0)
```

Dann die Antwort (Gerät an Controller)

| Befehl | ASCII-Zeichen (sedezimal) | Kommentar |
|---|---|---|
| N | 4E | Antwort des Geräts (ATN bleibt 1) |
| D | 44 | |
| C | 43 | |
| V | 56 | |
| <LF> | 0A | |
| <CR> | 0D | Ende des Blocks (EOI wechselt auf 0) |

# 2

# Die serielle Schnittstelle

Charakteristisch für die serielle Datenübertragung ist, dass die Bits über nur eine Datenleitung, zeitlich nacheinander (bitseriell), übertragen werden. Da die Mikroprozessoren intern die Daten bitparallel verarbeiten, erfolgt beim Sender eine Parallel-Seriell- und beim Empfänger eine Seriell- Parallel-Umsetzung. Diese Aufgabe übernehmen spezielle Sende- und Empfangsbausteine innerhalb des Controllers.

Aufgrund der heute sehr hohen Übertragungsraten spielt der erhöhte Zeitaufwand der seriellen Kommunikation in den meisten Fällen keine Rolle. Der verringerte Installations- und Kostenaufwand und die einfache Benutzung spricht hingegen für die serielle Übertragungstechnik.

Die serielle Datenübertragung eignet sich sowohl für die Kommunikation zwischen zwei Teilnehmern (Punkt-zu-Punkt) als auch für mehrere Teilnehmer. Charakteristisch für eine Übertragungsstrecke sind Richtung des Datenverkehrs und der Datendurchsatz, bzw. die maximal mögliche Übertragungsrate. Übertragungsstrecken unterscheiden sich darin, in welchen Richtungen und zu welchem Zeitpunkt Nachrichten übertragen werden können. Grundsätzlich werden drei verschiedene Nutzungsmöglichkeiten unterschieden (Bild 2.1).

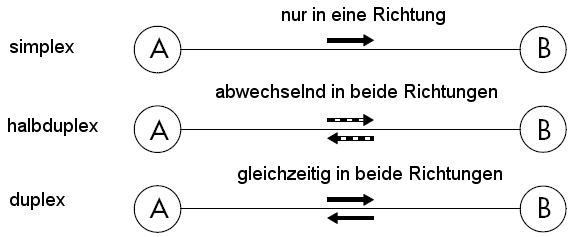



**Bild 2.1:** Übertragungsmöglichkeiten bei seriellem Datenverkehr

- Simplex- oder Richtungsverkehr: Der Datenaustausch erfolgt nur in einer Richtung.
- Halbduplex- oder Wechselverkehr: Daten können zeitlich versetzt in beide Richtungen übertragen werden.
- Vollduplex- oder Gegenverkehr: Der Datenaustausch kann in beide Richtungen gleichzeitig ablaufen.

Eine digitale Übertragung erscheint auf der Signalleitung als Bitstrom, aus der Sicht des Empfängers stellt sich dieser als eine Folge von Pegelwechseln dar. Um diesen Bitstrom wieder in die ursprünglichen Digitalwerte zurück zu verwandeln, muss der Empfänger wissen, zu welchem Zeitpunkt die Signale auf den Datenleitungen gültig sind, wann sie also ein Datenbit repräsentieren: Sender und Empfänger müssen sich also während der Übertragung miteinander synchronisieren. Diese Synchronisation kann entweder durch eine taktsynchrone Datenübergabe oder eine asynchrone, zeitgesteuerte Abtastung erreicht werden.

**Synchrone Übertragung:** Bei der synchronen Übertragung sind die Signale auf den Datenleitungen immer dann gültig, wenn ein von beiden Stationen genutztes Taktsignal einen bestimmten, vordefinierten Zustand einnimmt (z. B. Taktflankensteuerung gemäß Bild 2.2). Das Taktsignal wird entweder getrennt vom Datensignal auf einer zusätzlichen Leitung übertragen oder es kann aus dem Datensignal abgeleitet werden – sofern die Codierung dies zulässt (z. B. Manchester-Code). In jedem Fall erfolgt die Übertragung blockweise.

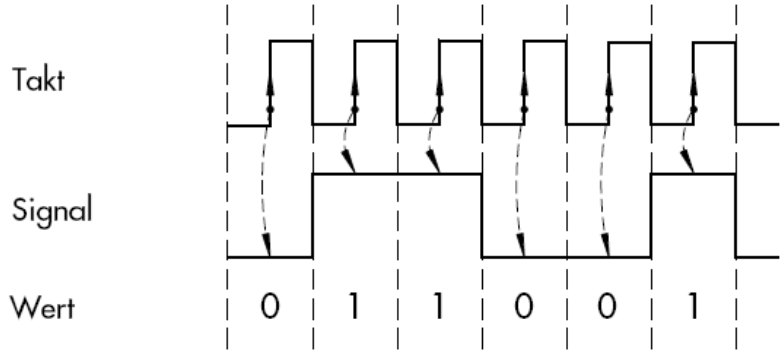



**Bild 2.2:** Synchrones Protokoll mit Takt- und Datenleitung

**Asynchrone Übertragung:** Bei der asynchronen Übertragung wird kein Taktsignal übertragen. Auch wenn Empfänger und Sender mit derselben Frequenz arbeiten, führen unvermeidliche Toleranzen (Bauteile, Temperaturdrift etc.) dazu, dass beide nach einiger Zeit nicht mehr synchron laufen. Daher muss sich der Empfänger in möglichst kurzen Abständen auf die Sendefrequenz synchronisiert.

Die Übertragung eines Bytes beginnt mit einem vorangestellten Startbit, das als logische 0 („SPACE“) gesendet wird. Anschließend werden nacheinander – je nach eingestelltem Format – fünf bis acht Datenbits, beginnend mit dem niederwertigen Bit (least significant bit, LSB), ausgegeben. Dem letzten Datenbit kann ein Paritätsbit folgen, das zur Erkennung von Übertragungsfehlern dient. Das Paritätsbit ergänzt das Datenbyte auf eine gerade (gerade Parität, even parity) oder ungerade (ungerade Parität, odd parity) Anzahl von 1-Bits. Das Ende des Zeichens wird durch ein oder zwei Stoppbits gebildet (Bild 2.3). Mit der ersten Signalflanke des Startbits synchronisiert der Empfänger seinen internen Bittakt auf die Empfangsdaten. Die folgenden Bits tastet er jeweils in der zeitlichen Bitmitte ab.

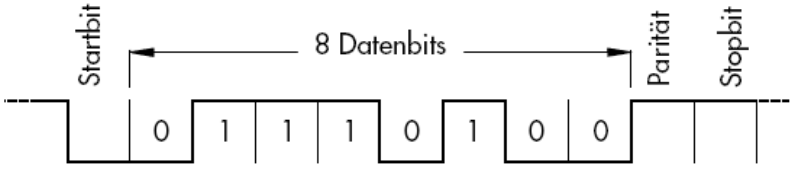



**Bild 2.3:** Die serielle Schnittstelle verwendet ein asynchrones Protokoll

Typisch Vertreter der synchronen Übertragung sind die Chip-Schnittstellen (siehe 4, die ab Seite 35 besprochen werden. Die Asynchrone serielle Schnittstelle wird normalerweise einfach als „serielle Schnittstelle“ bezeichnet. Fast jeder heutige Mikrocontroller verfügt über beide Schnittstellentypen. Im Notfall lassen sie sich sogar über parallele Schnittstellen nachbilden („soft serial“).

Eine wichtige Größe ist die Übertragungsgeschwindigkeit – also die Anzahl der Bits, die pro Sekunde übertragen werden. Bei den Chip-Schnittstellen hängt sie von Schnittstellendefintionen und Hardwareeigenschaften ab und ist nur selten genormt. Bei der asynchronen Schnittstelle haben sich bestimmte Datenrate historisch entwickelt – ausgehend von der Fernschreibtechnik. Hier wird die Datenrate in der Einheit *Bits pro Sekunde (bps)* bzw. *Baud* (nach dem französischen Ingenieur und Erfinder Jean Maurice Émile Baudot) angegeben. Dabei werden alle Bits (auch Start- und Stoppbit) gezählt und Lücken zwischen den Bytetransfers ignoriert. Deshalb ist die Baudrate der reziproke Wert der Länge eines Bits.

Bei den heute üblichen hohen Übertragungsraten sind ebenso die Einheiten *Kilobit pro Sekunde (kBit/s)* und *Megabit pro Sekunde (MBit/s)* gebräuchlich. Wird jedes Bit einzeln kodiert und übertragen, so muss die Übertragungsleitung in der Lage sein Frequenzen zu übertragen, die der halben Bitübertragungsrate entsprechen, z. B. beträgt bei einer Datenrate von 100 kBit/s die Übertragungsfrequenz 50 kHz. Als Datenraten sind folgende Werte üblich:

150, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600 und 115200

Nach dem Stoppbit kann sofort wieder eine neue Übertragung mit einem Startbit beginnen. Zur Vermeidung von Datenverlusten muss der Empfänger die Datenübertragung anhalten können, wenn keine weiteren Daten mehr verarbeitet werden können. Dieses sogenannte Handshake kann auf zwei Arten realisiert werden:

- Hardware-Handshake: Der Empfänger steuert über entsprechende Leitungen die Handshake-Eingänge des Senders (CTS, DSR) mit seinen Handshake-Ausgängen (DTR, RTS).
- Software-Handshake: Der Empfänger sendet zur Steuerung des Datenflusses spezielle Zeichen an den Sender (z. B. XON, XOFF).

Da die Pause zwischen zwei aufeinanderfolgenden Datenbytes beliebig lang sein darf, spricht man von einer „asynchronen“ Kommunikation. Für den Datenverkehr synchronisieren sich Sender und Empfänger bei der asynchronen Übertragung für jedes einzelne Zeichen neu. Vor jedem Zeichentransfer liegt auf der Übertragungsleitung das Signal auf 1-Pegel. Soll nun ein Zeichen übertragen werden, so wird dies dem Empfänger vom Sender durch ein Startbit angezeigt, indem für einen Taktzyklus das Signal auf 0 gelegt wird. Anhand der 0-1-Flanke kann der Empfänger den Beginn eines Datenbytes exakt erkennen. Damit sind Sender und Empfänger für dieses Zeichen synchronisiert. Anhand der Stoppbits erkennt der Empfänger das Ende des Zeichens, damit dient das Stoppbit ebenfalls der Synchronisation. Sender und Empfänger müssen sich zuvor auf die Anzahl der Stoppbits, der Datenbits, der Berechnung der Paritätsbits und auf die Frequenz des Übertragungstaktes (Baudrate) verständigen. Diese Parameter werden zumeist einmal in den Schnittstellen einprogrammiert und bleiben für die gesamte Dauer der Kommunikation unverändert. Normalerweise hat die serielle Schnittstelle am Controller dieselben Pegel wie auch alle anderen Pins – also digitale Pegel im Bereich von 3,3 bis 5 V). Für die Übertragung der Daten sind diese TTL- oder CMOS-kompatiblen Pegel aber nur für sehr kurze Strecken geeignet. Deshalb verwendet man zur Überwindung etwas größerer Entfernungen andere Pegel.

## 2.1 Die RS232C-Schnittstelle (V.24)

Diese häufig verwendete Schnittstelle nach der amerikanischen Norm RS232C ist in Europa mit fast identischen Eigenschaften unter V.24 genormt. Dieser Standard ist für zwei Kommunikationsgeräte konzipiert, die beide je eine Datenquelle (transmit, TX) und eine Datensenke (receive, RX) besitzen können. Zur bidirektionalen Datenübertragung werden mindestens drei Leitungen benötigt: eine Sendeleitung (TXD), eine Empfangsleitung (RXD) und eine Masseleitung (Ground). Die Signale der RS232 sind bipolar ausgelegt. Eine logische 0 wird bei den Datenleitungen durch eine Spannung von $+3$ bis $+15$ Volt, eine logische 1 durch $-3$ bis $-15$ Volt dargestellt. Bei den Steuerleitungen sind die Pegel genau umgekehrt (positive Spannung = 1, negative Spannung = 0). Das Signal-Störverhältnis ist damit wesentlich größer als bei TTL- oder CMOS-Pegeln. Dies ermöglicht eine störungsfreie Übertragung über größere Entfernungen. Die maximale Entfernung zwischen RS232-Geräten ist wie bei allen seriellen Übertragungsverfahren stark vom verwendeten Kabel und der Datenrate abhängig. Laut EIA-Norm definiert die RS232C die maximale Entfernung mit 15 Metern. Bei Verwendung von kapazitätsarmen Kabeln kann die maximale Distanz bis zu 50 Meter betragen. Je länger ein Kabel ist, umso größer gilt die Problematik der Potentialdifferenz zwischen beiden Endpunkten. Mit wachsenden Kabellängen sowie im industriellen Umfeld sollte grundsätzlich eine galvanische Trennung eingesetzt werden, damit unliebsame Störungen vermieden werden.

Neben der Masseleitung und den Datenleitungen gibt es noch eine ganze Reihe von Leitungen, die den Verkehr zwischen Rechner und Peripherie steuern. Meist interessieren aber nur einige Leitungen, um den Verkehr zwischen Computer und Peripherie oder zwischen zwei Computern aufrechtzuerhalten. Die anderen Leitungen bleiben unbeschaltet oder werden auf einen festen Pegel gelegt. Die wichtigsten Leitungen sind:

- GND bildet das gemeinsame Massepotential für die Datenleitungen.
- TXD führt die Sendedaten des Computers zum Peripheriegerät.
- RXD liefert die Daten vom Peripheriegerät zum Rechner.
- RTS zeigt der Peripherie die Übertragungsbereitschaft des Computers an.
- CTS signalisiert die Bereitschaft der Peripherie, Daten zu empfangen.
- DTR steuert bei Modems das Anschalten an die Telefonleitung.

Damit sind die sechs wichtigsten Leitungen aufgeführt. Oft sind noch die Leitungen DTR und DCD belegt, die dann meist auf die entsprechenden Anschlüsse des Schnittstellenbausteins führen. Die Norm definiert noch etliche weitere Signale, die am PC keine Entsprechung besitzen. Tabelle 2.1 liefert Aufschluss über Signalbezeichnungen, Steckerbelegungen (9-poliger und 25-poliger Stecker) sowie Signalbezeichnung der RS232-Schnittstelle.

**Tabelle 2.1:** V.24-Schnittstellenbelegung

| ITU | DIN | US | 25-pol. | 9-pol. | Beschreibung | Richt. |
|---|---|---|---|---|---|---|
| 101 | E 1 | – | 1 | – | Schutzerde | — |
| 102 | E 2 | GND | 7 | 5 | Signalerde (Ground) | — |
| 103 | D 1 | TXD | 2 | 3 | Sendedaten (Transmit Data) | ← |
| 104 | D 2 | RXD | 3 | 2 | Empfangsdaten (Receive Data) | → |
| 105 | S 2 | RTS | 4 | 7 | Sendeteil einschalten (Request To Send) | ← |
| 106 | M 2 | CTS | 5 | 8 | Sendebereitschaft (Clear To Send) | → |
| 107 | M 1 | DSR | 6 | 6 | Betriebsbereitschaft (Data Set Ready) | → |
| 108.2 | S 1.2 | DTR | 20 | 4 | Terminal ist bereit (Data Terminal Ready) | ← |
| 109 | M 5 | DCD | 8 | 1 | Empfangspegel (Data Carrier Detect) | → |
| 125 | M 3 | RI | 22 | 9 | Ankommender Ruf (Ring Indicator) | → |

Die Richtungsangabe bezieht sich auf die Peripherie (→ bedeutet Peripherie nach PC, ← entsprechend PC nach Peripherie). Werden Peripherie/Modem (DÜE = Datenübertragungseinrichtung) und Computer (DEE = Datenendeinrichtung) miteinander verbunden, verwendet man ein Kabel mit einer 1:1-Verbindung der wichtigsten Leitungen (Bild 2.4 links).

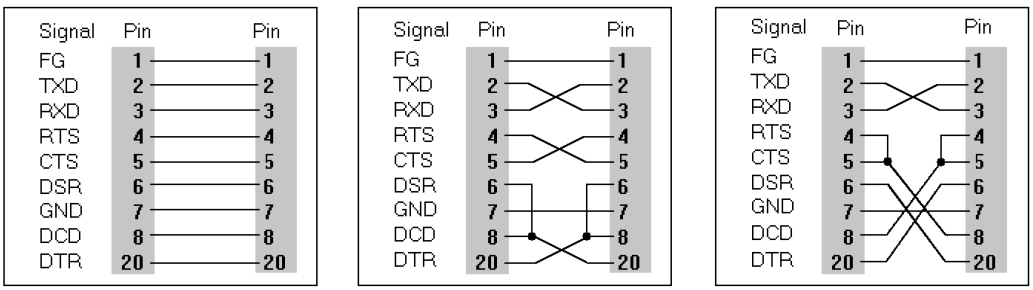
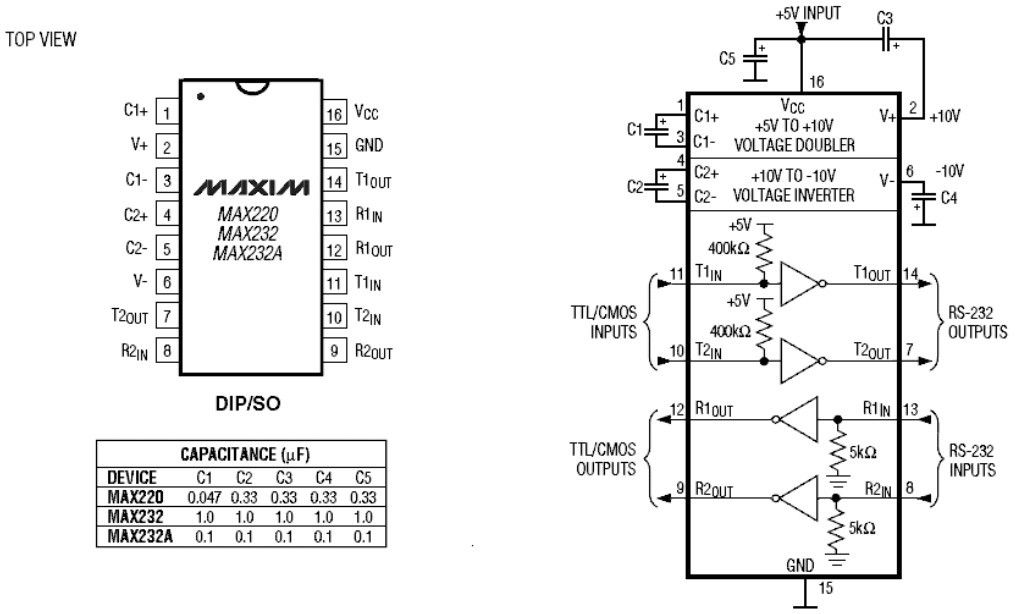



**Bild 2.4:** 1:1- und gekreuzte Verbindungen

Sollen hingegen zwei Computer direkt, also ohne Modem miteinander verbunden werden, müssen die Leitungen gekreuzt werden (Bild 2.4 rechts). Werden die Steuerleitungen gleich im Stecker zurückgeführt (RTS auf CTS, DTR auf DSR), kommt man mit einer dreiadrigen Verbindung aus. So eine direkte Verbindung zwischen zwei Computern wird allgemein auch als „Nullmodem“ bezeichnet, denn der Datenverkehr kann genauso ablaufen wie bei über Modem verbundenen Computern.

Bei der Pegelwandlung von TTL nach RS232 können wir praktischerweise auf einen bewährten integrierten Baustein zurückgreifen, den MAX232 von Maxim, um den sich inzwischen eine ganze Familie von Pegelwandlern mit verschiedenen Gehäusebauformen und unterschiedlichen elektrischen Eigenschaften gebildet hat. Der Baustein kann zwei Leitungen von TTL nach RS232 umsetzen und zwei

weitere Leitungen von RS232 nach TTL. Somit kann ein Baustein die Sende- und Empfangsleitung (TXD, RXD) und zwei Handshakeleitungen (z. B. CTS, RTS) umsetzen, was in vielen Fällen ausreicht. Zusätzlich sind im Chip noch zwei Ladungspumpen integriert, welche die benötigten Spannungen von $+12$ V und $-12$ V erzeugen. Für diese Spannungswandler werden als externe Beschaltung lediglich vier Kondensatoren benötigt (1-µF-Tantal-Elkos). In Bild 2.5 (nach Maxim-Unterlagen) sind die Innenschaltung des Bausteins, die Beschaltung mit den Kondensatoren sowie das Pin-Layout zu sehen. Sie werden diesen Baustein in etlichen Anwendungen wieder antreffen.




| CAPACITANCE (µF) | | | | | |
|---|---|---|---|---|---|
| DEVICE | C1 | C2 | C3 | C4 | C5 |
| MAX220 | 0.047 | 0.33 | 0.33 | 0.33 | 0.33 |
| MAX232 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| MAX232A | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |

**Bild 2.5:** Die Maxim-MAX232-Familie

### 2.1.1 Die RS422-Schnittstelle

Die RS422-Schnittstelle (und auch RS485) ist für die serielle Datenübertragung mit höherer Geschwindigkeit und über größere Entfernungen entwickelt worden und im industriellen Bereich weit verbreitet. Die seriellen Daten werden ohne Massebezug als Spannungsdifferenz zwischen zwei korrespondierenden Leitungen übertragen. Für jedes zu übertragende Signal existiert ein Adernpaar, das aus einer invertierten und einer nicht invertierten Signalleitung besteht.

Der Empfänger wertet die Differenz-Spannung zwischen beiden Leitungen aus, so dass Gleichtakt-Störungen auf der Übertragungsleitung nicht zu einer Verfälschung des Nutzsignals führen. Durch die Verwendung von abgeschirmten, paarweise verdrillten Cat-5-Kabeln lassen sich Distanzen von bis zu 1200 Metern (bei bis zu 100 000 bps) realisieren. RS422-Sender stellen unter Last Ausgangspegel von $\pm 2$ V zwischen den beiden Ausgängen zur Verfügung, die Empfängerbausteine erkennen Pegel von betragsmäßig 200 mV noch als gültiges Signal.

Bei RS422 gibt es also jeweils zwei Leiter für eine Übertragungsrichtung. Also muss das Kabel mindestens vier Adern haben, wobei es günstig ist, wenn jeweils zwei Adern paarweise verdrillt sind. An einen Sender können bis zu zehn Empfänger angeschlossen werden.

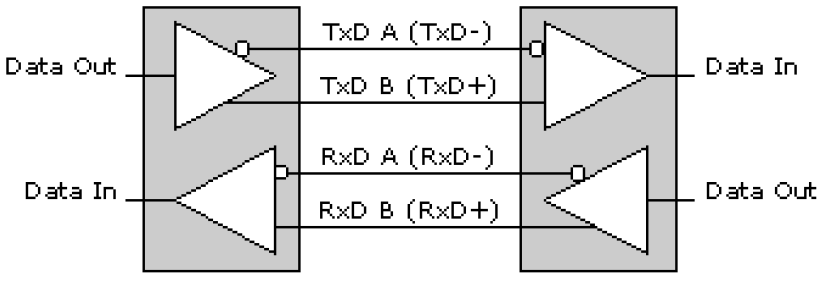



**Bild 2.6:** RS422-Verbindung

Die meisten externen Konverter von RS422 nach RS232 bieten aus Platzgründen keine weiteren Anschlüsse. In diesem Fall kann dann nur ein Software-Handshake genutzt werden. Stehen auf beiden Seiten Hardware-Handshakes (z. B. RTS/CTS) zur Verfügung, müssen noch jeweils vier weitere Verbindungen gezogen werden. Eine Terminierung des Kabels ist bei RS422-Verbindungen nur bei hohen Datenraten (mehr als 200 kbps) und großen Kabellängen erforderlich. In diesem Fall beugen relativ niederohmige Abschlusswiderstände (Größenordnung 120 Ω) Leitungsreflexionen vor.

### 2.1.2 Die RS485-Schnittstelle

Die RS485-Schnittstelle stellt eine Erweiterung der RS422-Definition dar. Während RS422 lediglich den unidirektionalen Anschluss von bis zu zehn Empfängern an einen Sendebaustein zulässt, ist die RS485-Schnittstelle als bidirektionales Bussystem mit bis zu 32 Teilnehmern konzipiert. Physikalisch unterscheiden sich beide Schnittstellen nur unwesentlich. Die Leitungen dieser Industrie-Bus-Schnittstelle werden wie bei RS422 im Gegentakt betrieben; es werden jedoch minimal nur zwei Leitungen benötigt, die halbduplex angesteuert werden. Der Vorteil der Zweidraht-Technik liegt hauptsächlich in der Multimaster-Fähigkeit: Jeder Teilnehmer kann prinzipiell mit jedem anderen Teilnehmer Daten austauschen. Die Norm sieht eine maximale Kabellänge von 500 Metern vor. Dank moderner, symmetrischer Leitungstreiber und kapazitäts- bzw. dämpfungsarmer Twisted-pair-Kabeln kann die Entfernung zwischen zwei Endgeräten wesentlich erhöht werden: RS485 unterstützt Kabellängen von bis zu 1,2 km und Datenübertragungsraten bis zu 1 Mbps. Bei langen Übertragungsstrecken kann zwischen der Betriebserde des Daten-Senders und der des Empfängers eine hohe Potentialdifferenz auftreten. Daher ist eine galvanische Trennung der Schnittstelle vom Rest der Schaltung (z. B. durch schnelle Daten-Optokoppler) zwingend vorgeschrieben. Da mehrere Sender auf einer gemeinsamen Leitung arbeiten, muss durch ein Protokoll sichergestellt werden, dass zu jedem Zeitpunkt maximal ein Datensender aktiv ist. Alle anderen Sender müssen sich zu dieser Zeit in hochohmigem Zustand befinden. Die Aktivierung der Senderbausteine kann durch Schalten einer Handshake-Leitung oder automatisch (abhängig vom Datenfluss) erfolgen. Eine Terminierung des Kabels ist bei RS485-Verbindungen grundsätzlich nötig.

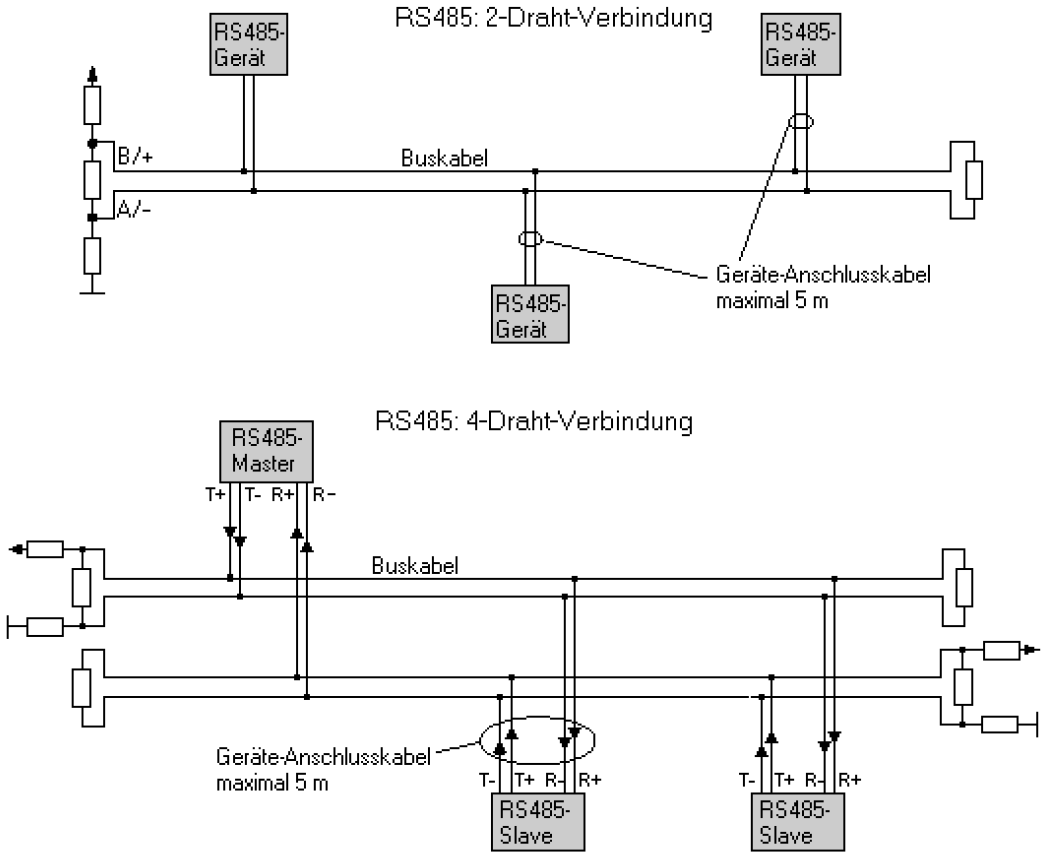



**Bild 2.7:** RS485-Verbindung (2-Draht und 4-Draht)

Beim RS485-2-Draht-Bus (Bild 2.7 oben) werden die Teilnehmer über eine maximal fünf Meter lange Stichleitung angeschlossen. Der Vorteil der Zweidraht-Technik liegt im Wesentlichen in der Multimaster-Fähigkeit, bei der jeder Teilnehmer prinzipiell mit jedem anderen Teilnehmer Daten austauschen kann. Der 2-Draht-Bus ist grundsätzlich nur halbduplexfähig, denn da nur ein Übertragungsweg zur Verfügung steht, kann immer nur ein Teilnehmer Daten senden. Erst nach Beendigung der Sendung können die Antworten anderer Teilnehmer erfolgen. Die wohl bekannteste auf der 2-Draht-Technik basierende Anwendung ist der Profibus.

Die beispielsweise vom DIN-Messbus genutzte 4-Draht-Technik (Bild 2.7 unten) ist für Master/Slave-Anwendungen geeignet. Wie im Bild zu sehen ist, wird der Datenausgang des Masters auf die Dateneingänge aller Slaves verdrahtet. Die Datenausgänge der Slaves sind zusammen auf den Dateneingang des Masters geführt.

Für die Pegelanpassung zwischen TTL und RS485 können wir wieder ein IC von Maxim einsetzen, den MAX485. Dieser Chip hat neben der Spannungsversorgung lediglich je zwei Ein- und Ausgangspins sowie zwei Pins, mit denen die Ein- und Ausgänge aktiviert werden können (wir erinnern uns: es darf nur einen Master geben). Da der PC jedoch keine TTL-Pegel liefert, sondern eine RS232-Schnittstelle besitzt, brauchen wir einen RS232-zu-RS485-Umsetzer – und Sie ahnen es sicher schon: die beiden MAX-Bausteine werden miteinander verheiratet. Die Schaltung in Bild 2.8 ist so auch recht einfach geworden.

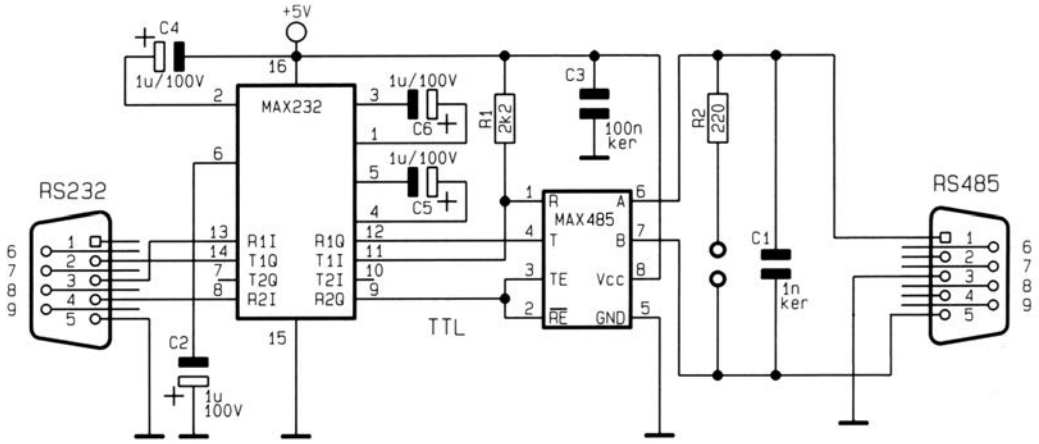



**Bild 2.8:** RS232- nach RS485-Umsetzer

Sie besteht nur aus den beiden Umsetzer-Bausteinen und einigen passiven Bauelementen. Die RS232-Signale werden vom MAX232 in TTL-Pegel umgewandelt. Diese gelangen dann an den MAX485, der die nötige Differenzspannung an den Ausgängen A und B erzeugt. Das DTR-Signal der RS232 steuert dabei die Richtung des Datenverkehrs. Als Ausgang wird auch auf der RS485-Seite ein 9-poliger Sub-D-Stecker vorgeschlagen, aber es kann auch jeder andere geeignete Steckverbinder zum Einsatz kommen. Befindet sich der Umsetzer an einem Ende der RS485-Busleitung, kann über eine Lötbrücke (oder einen Jumper) der Widerstand R2 aktiviert werden, womit auch gleich ein Leitungsabschluss vorhanden ist. Der Hersteller Maxim bietet eine ganze Reihe von Konverter-Bausteinen an (auch für die RS232-Schnittstelle), so z. B. MAX3440 – MAX3443 oder den MAX3535E, der intern eine galvanische Trennung realisiert und so den Computer von der RS485-Schnittstelle isoliert.

### 2.1.3 Die Stromschnittstelle(TTY)

Bei dieser Signalverbindung erfolgt die Datenübertragung nicht wie bei RS232 spannungsgesteuert, sondern durch einem eingeprägtem Linienstrom (typ. 20 bis 40 mA). Dadurch wirkt sich der Spannungsabfall auf der Datenleitung nicht wesentlich aus, so dass hier Kabellängen von bis zu einigen 100 m verwendet werden können. Diese Schnittstelle ist die Standardschnittstelle für Fernschreiber. Üblich ist eine Betriebsspannung von 12 bis 30 V, manchmal sogar bis zu 60 V. Im Ruhezustand bzw. während der Übertragung von 1-Bits fließt ein konstanter Strom, während 0-Bits durch eine Unterbrechung im Stromfluss gekennzeichnet sind.

Die Auskopplung der Nutzsignale aus der Stromschleife wird in der Regel über Optokoppler vorgenommen. Dies gewährleistet auch eine galvanische Trennung zwischen den verbundenen Geräten, so dass über TTY-Schnittstellen ohne weitere Schutzmaßnahmen eine isolierte Datenübertragung über

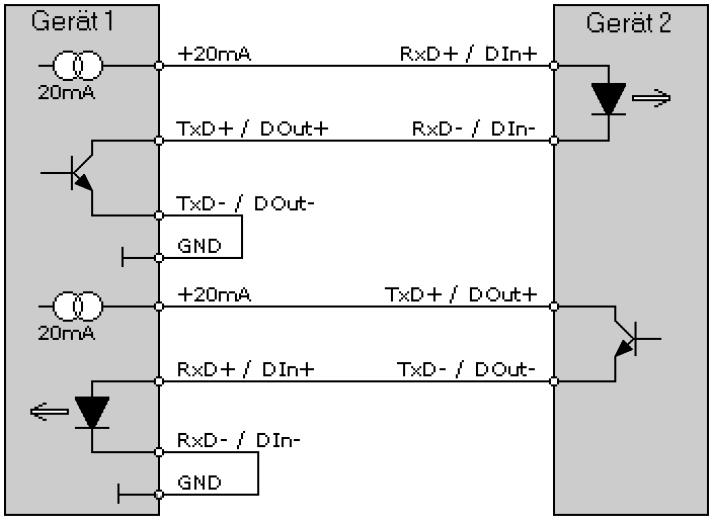



**Bild 2.9:** Prinzipschaltung der Stromschnittstelle

eine große Distanz möglich ist. Der Vorteil der relativ sicheren Übertragung wird bei dieser Schnittstelle jedoch mit vergleichsweise niedrigen Datenraten erkauft (bis 19 200 bps). Die maximale Entfernung wird mit 1 km bei 2400 bps angegeben.

Die Stromschnittstelle wird auch „20 mA“- bzw. „Current-Loop“-Schnittstelle genannt. Der Name „TTY-Schnittstelle“ wurde ihr nach ihrem ersten Anwendungsgebiet verliehen, dem Fernschreibverkehr (Teletype). Heute wird die TTY-Schnittstelle nahezu ausschließlich für den Datenverkehr zur Programmierung und Ankopplung von SPS, elektronischen Waagen, industriellen Großanzeige-Displays und Protokolldruckern verwendet.

Ein weiteres Problem ist die Auslegung der Schnittstelle. Innerhalb jeder Stromschleife darf lediglich ein angeschlossenes Gerät den erforderlichen Schleifenstrom liefern. Sowohl die Sendekomponente als auch der Empfangsteil können aktiv (eingebaute Stromquelle) oder passiv (Schalter/Schalttransistor bzw. Optokoppler-Eingang) ausgelegt sein. Trifft ein passiver Sender auf einen passiven Empfänger, tut sich gar nichts. Ebenso können zwei aktive Komponenten nicht miteinander spielen.

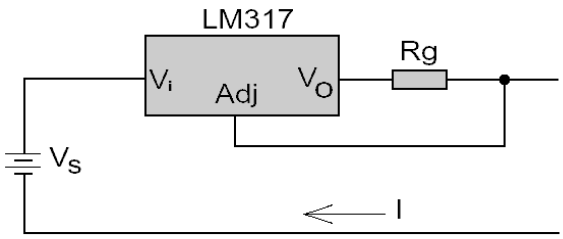



**Bild 2.10:** Stromquelle mit LM317

Dagegen ist die Schaltung für eine Stromquelle recht einfach. Mit dem integrierten Spannungsregler LM317 lässt sich recht einfach eine Stromquelle gemäß Bild 2.10 aufbauen. Für den Strom gilt

$$I = \frac{1.25}{R_g} \tag{2.1}$$

Für unseren Fall ergibt sich ein Wert von 20,1 mA bei einem Widerstandswert von 62 Ω. Die Spannung der Stromschleife beträgt maximal $V_s - 3$ Volt. Die Spannung muss so gewählt werden, dass alle Spannungsabfälle in der Stromschleife aufgefangen werden. Bei einem Optokoppler kann man von maximal 2 V ausgehen, am LM317 fallen nochmals 3 V ab, so dass man mit einer 5-V-Spannungsquelle nur noch knapp hinkommt. Auf der sicheren Seite ist man mit einer Versorgungsspannung von 12 V, bei langen Leitungen braucht man eventuell noch mehr.

## 2.2 Die USB-Schnittstelle

Die Abkürzung USB steht für „Universal Serial Bus“ und ist ein relativ neues Bus-System, das 1995 durch ein Konsortium von führenden Unternehmen der Computerbranche in Zusammenarbeit mit Intel entwickelt wurde. Jeder neue PC und jedes Notebook verfügen inzwischen über eine oder mehrere USB-Schnittstellen. An deser Stelle wird der USB nur kurz behandelt, ausführlichere Informationen dazu finden Sie unter: *http://www.netzmafia.de/hardware/USB/*.

Aber warum weicht man auf USB aus? Gründe gibt es viele: Neben der großen Marktverbreitung spricht vor allem die einfache Installation und Handhabung auf Benutzerseite dafür. Zwar sind die Steckverbinder nicht der Weisheit letzter Schluss und auch die Datenübertragungsrate wesentlich geringer als beim Netzwerk, aber die Vorteile überwiegen in vielen Anwendungsfällen: Hot plugging, Plug-and-Play sowie der Verzicht auf weitere I/O-Ports, DMA- und Interrupt-Kanäle. Die Installation ist zumindest auf USB-tauglichen Windows-Plattformen idiotensicher. USB-Geräte steckt man einfach während des Betriebs ein. Sofortiger Einsatz ist möglich, ein nerviger Reboot entfällt, und es lassen sich bis zu 127 Geräte anschließen. Zudem vermeiden USB-Geräte Ressourcenkonflikte. Der Controller im PC benutzt zwar generell einen Interrupt und 32 I/O-Ports, doch die angeschlossenen USB-Geräte belegen keine weiteren System-Ressourcen. Die Stromversorgung von bis zu 500 mA je USB-Port wird auch gleich mitgeliefert und erlaubt somit eine Vielfalt an Funktionen und Anwendungsmöglichkeiten zur Erweiterung von Schnittstellen. Durch die Kaskadierung von USB-Hubs lässt sich, wie bei Netzwerken, eine baumartige Struktur erzielen. Die maximale Länge eines Kabelsegmentes beträgt fünf Meter. Die Daten werden als Differenz-Signal mit einer Geschwindigkeit von 12 Mbit/s oder 1,5 Mbit/s übertragen.

An dieser Stelle wird nur ein Überblick der USB-Schnittstelle gegeben, ausführlichere Informationen finden Sie unter *http://www.netzmafia.de/hardware/USB/*.

### 2.2.1 USB-Hardware

Der USB kennt zwei verschiedene Steckertypen, „Typ A“ und „Typ B“ genannt. Das System ist so konzipiert, dass man sie auch nicht verkehrt herum einstecken kann. Kabel sind immer 1:1 verbunden, und die Belegung ist immer gleich:

**Pinbelegung und Adernfarben beim USB**

| Pin | Funktion | Adernfarbe |
|---|---|---|
| 1 | +5V | rot |
| 2 | Data- | weiss |
| 3 | Data+ | grün |
| 4 | Masse | schwarz |

An der Rückseite eines modernen PCs findet man zwei oder mehr Buchsen vom Typ A. Kleinere, langsame Geräte wie beispielsweise Mäuse verwenden ein dünnes, fest angebrachtes Kabel mit einem Stecker vom Typ A. In anderen Fällen besitzt das Gerät selbst eine USB-Buchse vom Typ B. Die Verbindung erfolgt dann mit einem Kabel vom Typ A–B. Bild 2.11 zeigt die Buchsenbelegung und typische Steckerformen.

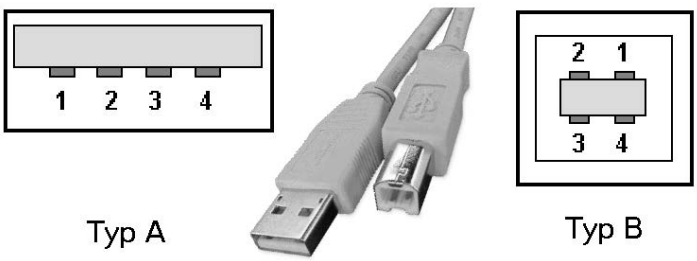



**Bild 2.11:** USB-Anschlussbuchsen

Die Kabel gab es nur fertig konfektioniert und vergossen zu kaufen. Einzelne USB-Stecker waren nicht erhältlich – man muss gegebenenfalls ein Kabel „schlachten“. Länge, Kabelquerschnitt, Abschirmung usw. sind genau vorgeschrieben. Inzwischen ist der Standard aufgeweicht, es gibt alle

USB-Komponenten einzeln und neben dem Stnadard-USB-Steckern haben sich Mini- und Mikro- Stecker und -Buchsen etabliert, zum Teil auch noch in mehreren Varianten.

Auch der Unterschied zwischen Fullspeed und Lowspeed spielt hier eine Rolle. Das System der vorgeschriebenen Kabel verhinderte zuverlässig, dass ein Lowspeed-Kabel für eine Fullspeed-Verbindung eingesetzt wird. Alle Verbindungskabel sind Fullspeed-Kabel, während Lowspeed-Kabel nur fest angebaut vorkommen. Für den Fall einer notwendigen Kabelverlängerung gibt es Verlängerungskabel vom Typ A-A und mittlerweile Adapter aller Art – die zum Teil dafür sorgen, dass die Datenübertragung gestört wird.

**Tipp**

Gleich eine erste Anwendung: Für Geräte mit einem relativ niedrigen Stromverbrauch eignet sich die USB-Schnittstelle bestens als „Versorgungsanschluss“. Sie besorgen sich ein USB-Kabel, knipsen den Typ-B-Stecker ab und haben auf der roten und schwarzen Kabelader stabilisierte 5 Volt zur Verfügung (die beiden USB-Datenleitungen werden dabei sorgfältig isoliert). Schon hat man ein Steckernetzteil und die Spannungs-Stabilisierung gespart. Andere mögliche 5-V-Quellen wären der Gameport und die Tastaturschnittstelle.

Die Signale auf den beiden Leitungen, $D+$ und $D-$, sind Differenzsignale mit Spannungspegeln von 0 und 3,3 Volt. Die Versorgungsspannung am USB kann bis zu 5,25 V betragen und bei starker Belastung bis auf 4,2 V abfallen. Man unterscheidet zwischen Geräten mit eigenem Netzteil und solchen, die über den USB versorgt werden. In vielen Fällen können beide Betriebsarten gewählt werden. Viele Geräte besitzen eine Netzteilbuchse, die zur Schonung des versorgenden PC mit einem externen Netzteil verbunden werden kann. Nach den USB-Spezifikationen ist die Stromaufnahme aus dem Bus automatisch begrenzt.

Der USB unterstützt drei Busgeschwindigkeiten: High-Speed mit 480 MBit/s, Full-Speed mit 12 MBit/s und Low-Speed mit 1,5 MBit/s. Die USB-Hostcontroller in modernen Rechnern unterstützen alle drei Geschwindigkeiten. Neben den Nutzdaten müssen jedoch Statussignale, Steuersignale und Prüfinfo zur Fehlersicherung übertragen werden. Auch nutzen alle Peripheriegeräte den Bus gemeinsam. Daher liegt die mit einem Gerät tatsächlich erreichbare Datentransferrate unterhalb der Busgeschwindigkeit. Die theoretisch maximale ereichbare Rate für einen einzelnen Transfer liegt bei etwa 53 MByte/s (High-Speed), 1,2 MByte/s (Full-Speed) bzw. 800 Byte/s (Low-Speed), siehe Bild 2.12). Die Geschwindigkeit wird vom Master vorgegeben, die angeschlossenen Geräte synchronisieren sich darauf. Da kein gesondertes Taktsignal übertragen wird, muss der Takt aus dem Datensignal zurückgewonnen werden (NRZI, Non-Return-to-Zero). Die Datencodierung und -decodierung erfolgt durch die USB-Hardware.

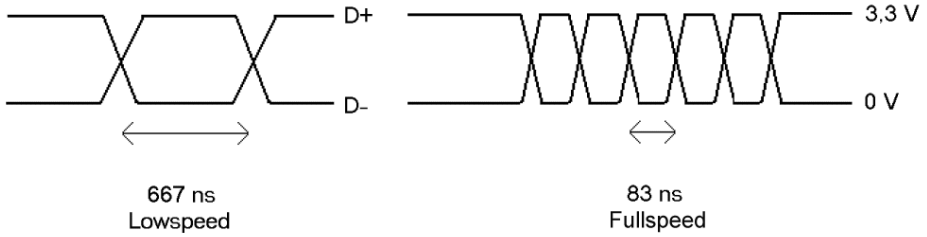



**Bild 2.12:** USB-Signale

Damit die Synchronisierung bei langen 1-Folgen nicht verlorengeht, wird nach sechs aufeinanderfolgenden Einsen automatisch eine 0 eingefügt (Bit Stuffing), um einen Pegelwechsel zu erzwingen. Der Empfänger entfernt diese 0 wieder aus dem Datenstrom. Jedes Datenpaket wird zum Zweck der Synchronisation von einem Sync-Byte angeführt.

Der USB besitzt nur einen Master, d. h. alle Aktivitäten gehen vom PC aus. Daten werden in kurzen Paketen von 8 bis 256 Bytes versandt und empfangen. Der gesamte Datenverkehr spielt sich in Frames von einer Millisekunde Dauer ab. Innerhalb eines Frames können aufeinanderfolgend Datenpakete für mehrere Geräte verarbeitet werden. Es dürfen auch die beiden Pakettypen, Lowspeed und Fullspeed, miteinander gemischt werden. Um mehrere Geräte mit dem USB zu verbinden, wird eine Verteilerkomponente, der sogenannte USB-Hub, benötigt. Normalerweise befindet sich mindestens ein Hub schon auf dem Mainboard des PC, der durch weitere Hubs kaskadiert werden kann. Der

Hub verhindert auch, dass Fullspeed-Signale an Lowspeedgeräte weitergeleitet werden. Die theoretische Obergrenze liegt bei sieben kaskadierten Hubs und 127 Geräten insgesamt. Bei jedem USB-Peripheriegerät wird eine der Datenleitungen über einen Widerstand von 1,5 kΩ mit 3,3 V verbunden, bei einem Fullspeed-Gerät ist dies die Leitung D+, bei einem Lowspeed-Gerät D-. Der Hub ist so in der Lage, den Typ des Geräts zu erkennen.

USB-Geräte können auf vier verschiedene Arten mit dem PC Daten austauschen.

- **Control-Transfer**: Dieser Transfertyp wird zur Gerätesteuerung benutzt. Es können sowohl Daten gelesen als auch geschrieben werden. Alle Geräte unterstützen diesen Transfertyp. Es kommt ein spezielles Protokoll mit doppelter Fehlersicherung zum Einsatz.
- **Interrupt-Transfer**: Dient der asynchronen Übertragung von Informationen vom Gerät zum Host. Dabei werden meist nur wenige Bytes übertragen (z. B. Mäuse und Tastaturen). Interrupttransfer dient der zyklischen Abfrage von Geräten. Im Fehlerfall wird die Übertragung wiederholt. Dabei wird vom Host sichergestellt, dass die Daten periodisch abgefragt werden.
- **Bulk-Transfer**: Für größere Datenmengen, die eine Fehlerüberwachung benötigen, aber nicht zeitkritisch sind (z. B. Drucker oder Scanner).
- **Isochronous-Transfer**: Übertragung großer Datenmengen mit garantierter Übertragungsrate und Latenzzeit in einer Richtung. Es ist dabei Lesen oder Schreiben möglich (z. B. Soundkarte). Bei Übertragungsfehlern findet keine Wiederholung statt.

Ein großer Vorteil von USB ist die automatische Erkennung (Plug and Play) neu angeschlossener Geräte. Das Betriebssystem muss dazu in der Lage sein, Informationen von einem Gerät abzufragen, um dann den passenden Treiber zu laden und das Gerät entsprechend anzusprechen. Ein neues Gerät wird dabei angemeldet (Enumeration), es erhält eine Bus-Adresse und wird durch einen speziellen Treiber unterstützt. Die Enumeration wird völlig selbstständig vom Betriebssystem ausgeführt. Den Anschluss eines neuen Gerätes erkennt der Hub Pull-Up-Widerstand der Datenleitung. Nun laufen folgende Schritte ab:

1. Der Hub informiert den Host über das neu angeschlossene Gerät.
2. Der Host erfragt vom Hub den Port des neuen Geräts.
3. Der Host schaltet per Befehl diesen Port ein und führt einen Bus-Reset aus.
4. Der Hub erzeugt ein Reset-Signal (10 ms) und gibt den Versorgungsstrom für das Gerät frei.
5. Das Gerät ist nun bereit und antwortet auf der Default-Adresse 0.
6. Der Host weist dem Gerät eine eigene Bus-Adresse zu.
7. Der Host liest alle Konfigurations-Informationen aus dem Gerät.
8. Der Host weist dem Gerät eine der möglichen Konfigurationen zu.

Sie sehen, die Geschichte ist recht komplex und jedes Gerät braucht seinen Treiber. Im Gegensatz zu Nicht-USB-Komponenten wird der Treiber eben nicht beim Booten oder gar manuell geladen, sondern vom USB-Subsystem automatisch aktiviert/installiert, wenn das Gerät am Bus auftaucht. Irgendwann stehen dann wieder die Standard-Dateifunktionen zum Ansprechen des Geräts zur Verfügung.

**Ausführlichere Informationen zu USB finden Sie unter: *http://www.netzmafia.de/hardware/USB/***

#### COM-Port-Adapter

Für Besitzer aktueller Rechner ohne COM-Port bietet der Fachhandel ein breites Sortiment an USB-zu-RS-232-Umsetzern an. Es gibt einfache Modelle mit einem Port ab zehn Euro ebenso wie Vier-Port-Geräte für knapp 70 Euro. Vor allem die günstigen Modelle ähneln sich sehr stark: Die Elektronik (USB-Chip, RS-232-Pegelwandler, passive Bauteile) steckt in einem etwas vergrößerten, meist vergossenen 9-poligen Steckergehäuse. Das Kabel ist entweder direkt angespritzt, oder der USB-Stecker besitzt eine Typ-B-Buchse. Der zentrale USB-Chip der meisten Geräte stammt entweder vom taiwanesischen Hersteller Prolific oder der britischen Firma Future Technology Devices International. Adapter mit diesen Chips sowie die Belkin-Mehrchip-Lösung gehören zu den gut von Linux unterstützten Geräten. Normalerweise erkennt sie das System automatisch, lädt das passende Kernel-Modul und stellte die serielle Schnittstelle `/dev/ttyUSB0` bereit (bei Mehrport-Adaptern je nach Typ zusätzlich `/dev/ttyUSB1` bis `/dev/ttyUSB3`). Für alle Adapter gilt, dass die Treiberversionen auf den beiliegenden CDs immer recht alt sind und man sich die neueste Version aus dem Netz holen sollte (sofern ein Treiber nötig ist). Zu empfehlen sind unter anderem die Adapter von Belkin, Conrad, Hama, Lindy, M-Cab, Pearl, VScom und Reichelt.

#### UM232R USB-Seriell-UART-Entwicklungsmodul

Das UM232R-Modul von FTDI (Future Technology Devices International, *http://www.ftdichip.com/*) ist ein TTL-Entwicklungsmodul mit einer USB-seriellen-UART-Schnittstelle für FT232R-USB-ICs mit vollem Hardware-Handshaking. Neben den Modemsteuerleitungen stehen noch einige frei programmierbare Leitungen zur Verfügung. Das Modul (Bild 2.13) eignet sich ideal für schnelle Prototypen und die Entwicklung von FT232R-Endprodukten geringer bis mittlerer Stückzahlen. Das UM232R wird auf einer Platine geliefert, die in einen 24-poligen DIP-Stecksockel gesteckt werden kann und den Zugriff auf alle UART- und CBUS-Schnittstellenpins des FT232R-Geräts ermöglicht. Das UM232R kann über den USB-Bus mit Energie versorgt werden oder mit einer separaten Spannungsquelle.

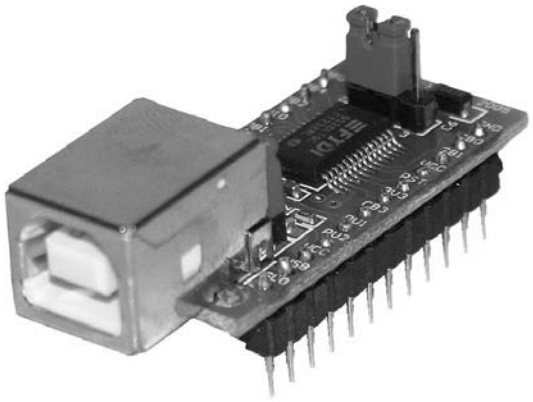


**Bild 2.13:** USB-Seriell-UART-Entwicklungsmodul UM232R

Die Ausgangspegel können an die angeschlossene Logik angepasst werden: per Konfigurationspin sind 5 V, 3,3 V oder andere Ausgangsspannungen möglich. Das UM232R kann zudem mit bis zu 3 MBaud bei TTL/CMOS-Pegeln kommunizieren. Zusätzlich bietet er eine automatische Sendepuffersteuerung für RS485-Anwendungen. Die freien Steuerleitungen sind ab Werk als Ausgänge für Kontroll-LEDs eingestellt. Das Modul wird vorprogrammiert mit Hersteller- und Geräte-ID und zusätzlich einer eindeutigen USB-Seriennummer geliefert. Der Anschluss an den PC erfolgt über ein USB-Standardkabel Typ A-auf-B. Erhältlich ist das Modul unter anderem bei Farnell (Best.-Nr. 114-6036). Weitere Informationen und kostenlose Treiber für alle Plattformen bietet die Webseite von FTDI, insbesondere folgende URLs sind von Interesse:

*http://www.ftdichip.com/Products/EvaluationKits/UM232R.htm*
*http://www.ftdichip.com/Documents/AppNotes/AN232R-01_FT232RBitBangModes.pdf*
*http://www.ftdichip.com/Documents/AppNotes/AN232R-02_FT232RChipID.pdf*
*http://www.ftdichip.com/Documents/ProgramGuides.htm*

# 3

# Feldbusse

Diese Bussysteme treten in der unteren Steuerungs-, Messwerterfassungs- und Regelungsebene auf. Sie dienen der Übertragung von Signalen zwischen Sensoren, Aktoren, Steuerungen (z. B. SPS = speicherprogrammierte Steuerungen), Computern und Prozessleitsystemen.

Beispiele hierfür sind der Profibus (DIN 19245) der Bitbus (IEEE 1118), der Interbus S für die unterste Ebene. Des weiteren existieren u. a. spezielle Bussysteme für Gebäude im Bereich der Haustechnik (KNX-Standard nach ISO/IEC 14543-3, früher (EIB) nach EN 50090) und Bussysteme für Kraftfahrzeuge, beispielsweise der CAN-Bus (CAN = Controller Area Network) oder der LIN-Bus LIN = Local Interconnect Network. Diese Bussysteme arbeiten störsicher, jedoch mit geringer Datenrate. LIN wurde speziell für die kostengünstige Kommunikation von intelligenten Sensoren und Aktoren in Kraftfahrzeugen entwickelt.

## 3.1 Der CAN-Bus

„CAN“ steht, wie schon erwähnt, für „Controller Area Network“; dieser Bus ist also räumlich begrenzt (z. B. innerhalb einer Maschine/Maschinengruppe oder in einem Fahrzeug). Er wurde speziell für den Einsatz innerhalb einer stark gestörten Umgebung konzipiert. Ein wesentliches Leistungsmerkmal von Bussen ist die Vergabe der Buszugriffsberechtigung (Arbitration), die beim CAN-Bus anders erfolgt als bei gebräuchlichen Netzen. Der CAN-Bus arbeitet mit CSMA/CR (Carrier Sense Multiple Access/Collision Resolution). In der Literatur wird das Verfahren oft als CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance) bebezeichnet. Die Daten sind NRZ-kodiert. Der Bus ist entweder mit Kupferleitungen oder über Glasfaser ausgeführt. Der CAN-Bus arbeitet nach dem Multi-Master-Prinzip, bei dem mehrere gleichberechtigte Busteilnehmer durch ein Bussystem miteinander verbunden sind.

Im Falle von Kupferleitungen arbeitet der CAN-Bus bei höheren Datenraten mit Differenzsignalen. Die Differenzsignale werden normalerweise mit zwei oder drei Leitungen ausgeführt: CAN-HIGH, CAN-LOW und optional CAN-GND (Masse). CAN-LOW und CAN-HIGH haben immer komplementäre Pegel gegen Masse (Unterdrückung von Gleichtaktstörungen).

Bei langsameren Varianten reicht oft ein Eindrahtsystem mit der Karosserie als Masse. Praktisch wird es meistens doch als Zweidrahtsystem ausgeführt, verwendet aber beim Lowspeed-CAN im Fehlerfall eines Aderbruchs den Eindrahtbetrieb als Rückfallebene.

Bei CAN bezeichnet ein sogenannter Objekt-Identifier immer den Inhalt einer Nachricht, nicht ein bestimmtes Gerät. So könnte in einem Messsystem den Parametern „Temperatur“, „Drehzahl“ und „Druck“ jeweils ein eigener Identifier zugewiesen sein. Es lassen sich auch mehrere Parameter unter einem Identifier zusammenfassen, solange die maximale Länge eines Datenfeldes nicht überschritten wird. Die Empfänger entscheiden anhand des Identifiers, ob die Nachricht für sie relevant ist oder nicht. Auch dient der Objekt-Identifier der Priorisierung der Nachrichten. Die Spezifikation definiert zwei verschiedene Identifier-Formate:

- 11-Bit-Identifier, auch „Base frame format“ genannt (CAN 2.0A)

- 29-Bit-Identifier, auch „Extended frame format“ genannt (CAN 2.0B).

Ein Teilnehmer kann Empfänger und Sender von Nachrichten mit beliebig vielen Identifiern sein, aber umgekehrt darf es zu einem Identifier immer nur maximal einen Sender geben, damit die Arbitrierung funktioniert. Dazu erfolgt die Aufschaltung der Teilnehmer so, dass die logische **0** gegenüber der **1** Priorität besitzt. Dies erreicht man beispielsweise durch Open-Collector-Treiber mit Pullup-Widerstand (Bild 3.1).

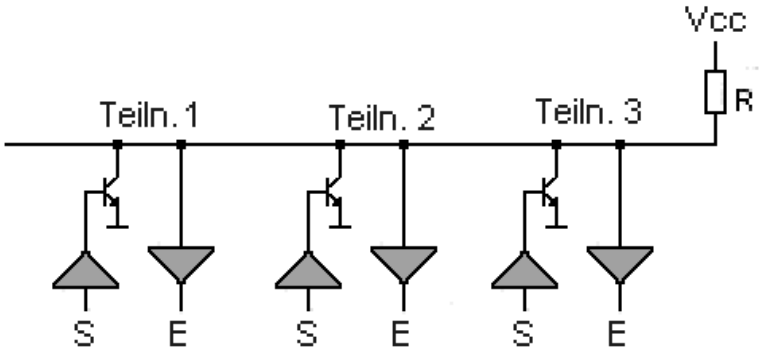



**Bild 3.1:** CAN-Busankopplung über Open Collector

Die Arbitrierung erfolgt bitweise mit gleichzeitigem Rücklesen und Vergleichen der gesendeten Daten durch den Empfangsteil des Senders. Dazu überwacht jeder Sender den Bus, während er den Identifier sendet. Senden zwei Teilnehmer gleichzeitig, so überschreibt das erste dominante Bit eines der beiden das entsprechend rezessive Bit des anderen, was dieser erkennt und darauf hin seinen Übertragungsversuch beendet. Durch dieses Verfahren wird auch eine Hierarchie der Nachrichten untereinander festgelegt. Die Nachricht mit dem zahlenmäßig kleinsten Identifier darf immer übertragen werden. Für die Übertragung von zeitkritischen Nachrichten kann also ein Identifier hoher Priorität (niedrige ID, z. B. 0x010, 0x001, 0x000) vergeben werden, um ihnen so Vorrang bei der Übertragung zu gewähren (Bild 3.2.

Dennoch kann selbst bei hochprioren Botschaften der Sendezeitpunkt zeitlich nicht genau vorhergesagt werden, da gerade in Übertragung befindliche Nachrichten nicht unterbrochen werden können und sich so der Startzeitpunkt einer Sendung bis zur maximalen Nachrichtenlänge verzögert.

In Bild 3.3 ist das CAN-Bus-Telegramm abgebildet. Es beginnt mit einem Synchronisationsbit, mit dessen Hilfe sich alle sendewilligen und empfangsbereiten Stationen synchronisieren. Danach folgt das 12 Bit lange Arbitrationsfeld, dessen vorderen 11 Bits eine Kennung darstellen. Wie schon erwähnt hängt die Priorität auf dem Bus nicht von den Stationen ab, sondern vom Identifier der Nachricht, die versendet wird. Die Vergabe der Prioritäten muss also schon während der Entwicklungsphase erfolgen. Es lassen sich bei 11 Bit insgesamt 2048 Meldungen codieren, wobei man den Code auch in zwei logische Teile aus Nachrichtenkennung und Stationskennung aufteilen kann.

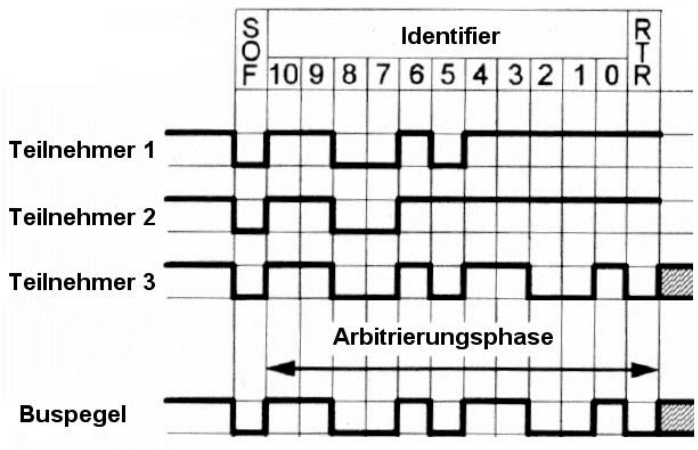



**Bild 3.2:** CAN-Arbitrierungsphase

Auch der Datenempfang entspricht nicht althergebrachten Denkweisen. Wichtig ist nicht ein bestimmter Empfänger der Nachricht, sondern die Art der Nachricht. Jede Station besitzt ein sogenanntes Empfangskennungsregister, in dem eine bestimmte Kennung abgelegt wird. Stimmt die empfangene Kennung mit der Empfangskennung überein, werden die Daten übernommen. Die lieferbaren CAN-Controller erlauben das Maskieren einzelner Bits, die dann bei der Empfangskennung keine Rolle spielen. So lassen sich auch Gruppen von Telegrammen (Frames) auswerten.

Das Datenfeld kann bis zu 8 Bytes lang sein. Die Länge wird durch vier Bits im Control-Feld gemeldet.

Es gibt vier verschiedene Arten von Telegrammen (Frames):

- Der **Daten-Frame** dient dem Transport von bis zu acht Bytes an Daten.
- Der **Remote-Frame** dient der Anforderung eines Daten-Frames von einem anderen Teilnehmer.
- Der **Error-Frame** signalisiert allen Teilnehmern eine erkannte Fehlerbedingung in der Übertragung.
- Der **Overload-Frame** dient als Zwangspause zwischen Daten- und Remote-Frames.

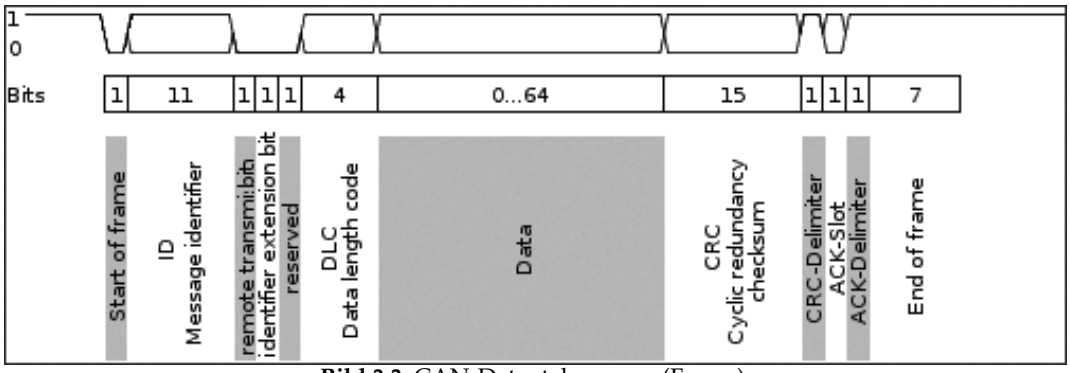



**Bild 3.3:** CAN-Datentelegramm (Frame)

Der Acknowledge-Slot quittiert den Empfang eines korrekten CAN-Frames. Jeder Empfänger, der keinen Fehler feststellen konnte, setzt einen dominanten Pegel an der Stelle des ACK-Slots und überschreibt somit den rezessiven Pegel des Senders. Im Falle einer negativen Quittung (rezessiver Pegel) muss der fehlererkennende Knoten nach dem ACK-Delimiter ein Error-Flag auflegen, damit erstens der Sender vom Übertragungsfehler in Kenntnis gesetzt wird und zweitens um netzweite Datenkonsistenz sicherzustellen. Erkennt ein Empfänger eine Fehlerbedingung, sendet er einen Error-Frame und veranlasst so alle Teilnehmer, den Frame zu verwerfen. Der Sender wiederholt nach dem Error-Frame seine Datenübertragung.

CANopen ist ein auf CAN basierendes Schicht-7-Kommunikationsprotokoll, welches anfänglich in der Automatisierungstechnik verwendet wurde, mittlerweile aber vorwiegend in Embedded Systemen eingesetzt wird. CANopen wurde vorwiegend von deutschen mittelständischen Firmen initiiert und im Rahmen eines ESPRIT-Projektes unter Leitung von Bosch erarbeitet. Seit 1995 wird es von der CiA (CAN in Automation, *http://www.can-cia.org*) gepflegt.

## 3.2 Der LIN-Bus

Das Local Interconnect Network (LIN) oder auch LIN-Bus ist die Spezifikation für ein serielles Kommunikationssystem. LIN wurde speziell für die kostengünstige Kommunikation von intelligenten Sensoren und Aktoren in Kraftfahrzeugen entwickelt. LIN basiert auf einem Eindraht-Bus (plus Fahrzeugmasse). Typische Anwendungsbeispiele sind die Vernetzung innerhalb einer Tür oder Steuerung der Innenbeleuchtung (LEDs mit LIN-Interface). Es wird dort eingesetzt, wo die Leistungsfähigkeit von CAN nicht benötigt wird.

Ein LIN-Bus setzt sich aus einem Master (meist ein CAN-zu-LIN-Interface) und einem oder mehreren Slaves zusammen. Der Master steuert die zeitliche Reihenfolge aller zu übertragenden Daten. Diese Daten werden von den entsprechenden Slaves dann übertragen, wenn sie vom Master dazu aufgefordert werden. Dazu sendet der Master einen Header (Identifier), der durch eine bestimmte

Nachrichtenadresse gekennzeichnet ist. Im Anschluss daran legt der Slave die Datenbytes auf den Bus.

Ein LIN-Frame (Bild 3.4)besteht aus dem Identifier und maximal acht Nutzbytes Daten), sie steht grundsätzlich jedem LIN-Slave zur Verfügung. Die Übernahme einer LIN-Botschaft hängt einzig von der Entscheidung des jeweiligen Slave ab (wie CAN ein empfängerselektives System). Die Übernahme wird über sogenannte Akzeptanz- bzw. Nachrichtenfilter gesteuert. Eine LIN-Botschaft kann so von einem, mehreren oder allen Slaves zur Weiterverarbeitung übernommen werden.

Zu jedem Zeitpunkt wird immer nur eine LIN-Botschaft übertragen, wodurch kein Mechanismus zur Auflösung von Buskollisionen erforderlich ist. Alle zu übertragenden Botschaften werden einmal innerhalb eines Zyklus übertragen. Die zeitliche Reihenfolge der Botschaften ist in einer sogenannten Schedule festgehalten. Diese kann je nach Bedarf gewechselt werden.

Um die Buslast durch Abfrageframes für seltene Ereignisse zu reduzieren, besteht die Möglichkeit, dass Slaves auf einen Identifier nur dann antworten, wenn sie neue Daten weiterzugeben haben. Dabei werden mögliche Buskollisionen vom Master erkannt. In einem solchen Fall werden nach der Kollision Abfrageframes gesendet, auf die nur jeweils ein Slave antworten darf.

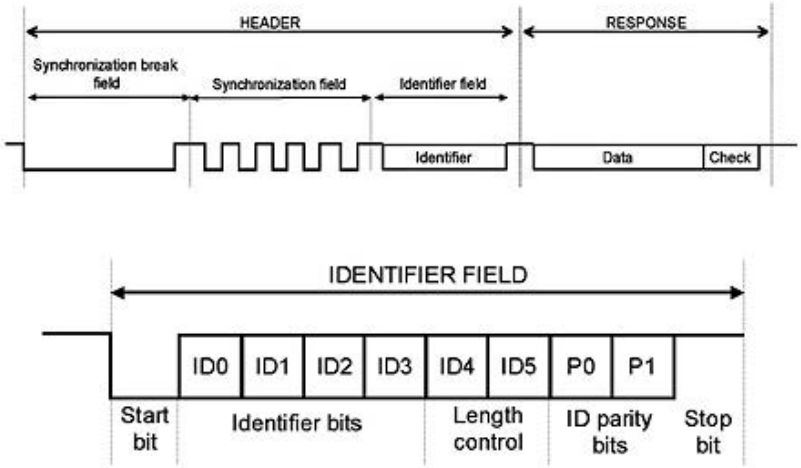



**Bild 3.4:** Aufbau eines LIN-Datensatzes

Die Signalzustände sind beim LIN-Bus wie beim CAN-Bus rezessiv und dominant. Die Bordnetzspannung stellt den rezessiven Zustand sowie den Ruhezustand dar, und Massepotenzial den dominanten Zustand. Die maximale Bruttodatenrate beträgt 20000 bps. Empfohlene Datenraten sind 2400 bps, 9600 bpss und 19200 bps.

Das LIN-Protokoll sieht keine Fehlersignalisierung vor. Es gibt jedoch zwei Mechanismen, um Übertragungsfehler erkennen zu können: Parity und Checksumme. Fehlerhafte Botschaften werden als nicht gesendet betrachtet und verworfen. Im LIN-Bus existiert keine Fehlersignalisierung, erkannte Fehler werden also nicht über das Protokoll signalisiert.

LIN bietet einen Mechanismus, um Geräte in einen Ruhezustand zu versetzen und damit Energie zu sparen. Bei der Spezifikation LIN 2.0 können alle Slaves in einen Sleep-Modus versetzt werden, indem der Master einen Master-Frame mit ID = 60 und dem ersten Daten-Byte = 0 sendet. Slaves werden auch automatisch in den Sleep-Modus versetzt, wenn die LIN-Kommunikation länger als vier Sekunden inaktiv ist.

LIN verfügt entsprechend auch über einen Wakeup-Modus für Geräte am Bus. Der Wakeup-Modus kann von jedem Knoten, Slave oder Master, initiiert werden. Der Wakeup-Modus wird initiiert, indem der Bus für 0,25 ... 5 ms dominant gemacht wird. Jeder Slave muss die Wakeup-Anfrage erkennen und bereit sein, einen Header innerhalb von 100 ms zu verarbeiten. Auch der Master sollte die Wakeup-Anfrage erkennen und Header senden, sobald die Slave-Knoten bereit sind (innerhalb von 100 ms ... 150 ms nach dem Empfang der Wakeup-Anforderung).

Die Spezifikationen des LIN-Bus werden vom LIN-Konsortium (*http://www.lin-subbus.org/*) gepflegt.

## 3.3 KNX

KNX (KONNEX) ist Nachfolger des europäischen Installationsbusses (EIB) und vollkommen kompatibel dazu. KN beschreibt, wie bei einer Installation Sensoren und Aktoren in einem Haus miteinander verbunden werden können,und legt fest, wie Sensoren und Aktoren miteinander kommunizieren müssen (das Protokoll). KNX steuert zum Beispiel die Beleuchtung und Beschattungseinrichtungen, die Gebäudeheizung sowie die Schließ- und Alarmanlage. Mittels KNX 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. KNX wird derzeit vor allem bei neuen Wohn- und Zweckbauten installiert, kann jedoch auch bei der Modernisierung von Altbauten nachträglich eingebaut werden. KNX ist der erste offene Weltstandard für Haus- und Gebäudeautomation (weshalb er namentlich EIB abgelöst hat, der ja „Europa“ im Namen trägt). Die Standardisierung ist durch DIN EN 50090 und ISO/IEC 14543-3 erfolgt.

Traditionell werden Elektrogeräte über die Stromversorgung an- oder ausgeschaltet, etwa mittels Parallel- oder Reihenschaltung. Versorgung und Steuerung erfolgte über das 230-V-Wechselspannungsnetz. Mit KNX werden Versorgung und Steuerung der Geräte getrennt /Bild 3.5). Es gibt nun zwei Netze: das Stromnetz mit 230 V Wechselspannung und das Steuerungsnetz (KNX-Netz), das mit maximal 30 V Gleichspannung betrieben wird. Diese werden unabhängig voneinander, meist parallel im Haus verlegt. Zudem gibt es Powernet-KNX, bei dem die Steuersignale über das normale Stromnetz gesendet werden (überwiegend für nachträglichen Einbau).

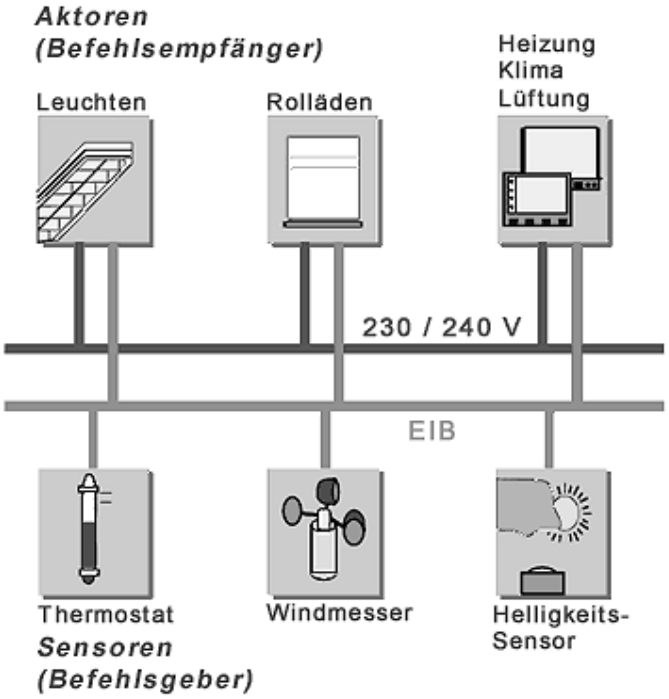



**Bild 3.5:** Prinzipieller Aufbau bei KNX (Quelle: Siemens)

Mit KNX kann nun erstmals jeder elektrische Verbraucher gesteuert werden. Durch Neuprogrammierung der KNX-Zentrale kann jede Art von Anschluss neu definiert werden. Ein Schalter, der gerade noch zum Anschalten einer Deckenleuchte diente, kann in wenigen Minuten zum Einschalten der Gartenbewässerung umprogrammiert werden. Auch die Zahl der Sensoren zur Steuerung eines Verbrauchers ist prinzipiell nicht begrenzt.

So könnten neben dem obligatorischen Schalter die Daten eines Windmessers genutzt werden, um Jalousien oder Markisen zu steuern oder alle Fenster und Türen ab einer bestimmten Windstärke automatisch zu schließen. Aktionen und beteiligte Sensoren lassen sich durch die Programmierung der Anlage flexibel festlegen. KNX verknüpft auch zahlreiche Gewerke miteinander: Heizung, Belüftung, Lichtanlage etc.

Zwischen einem beliebigen Elektrogerät und der Netzspannung wird eine Steuerung (Aktor) eingebaut. Der Aktor ist an das KNX-Netz angeschlossen und darüber seine Befehle. Diese stammen entweder direkt von Sensoren (Schalter, Helligkeitssensor, Temperatursensor etc.) oder indirekt von ei-

nem Computer oder Wanddisplay. Die KNX-Leitungen selbst bestehen in der Regel aus zwei Adernpaaren (rot-schwarz und weiß-gelb), wovon jedoch nur die rote und die schwarze Ader verwendet wird. Diese Leitungen müssen natürlich zu allen Aktoren und Sensoren führen. Die Versorgungsspannung der KNX-ANlage versorgt die Busankoppler, über die jedes KNX-Gerät am Bus angeschlossen ist.

Der Datenaustausch zwischen den KNX-Geräten erfolgt ebenfalls über Telegramme (Frames).Durch das Zugriffsverfahren CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance) werden Datenverluste im Falle von Kollisionen ausgeschlossen. Der KNX-Bus kommuniziert mit einer Übertragungsrate von 9600 bps. Für die Anbindung an lokale Computernetze (Ethernet) existieren IP-KNX-Koppler, so dass auch über deutlich schnellere Ethernetverbindungen kommuniziert werden kann.

KNX ist aufgeteilt in 15 Bereiche mit jeweils 15 Linien und 256 Teilnehmern pro Linie. Prinzipiell können damit ca. 60000 Busteilnehmer einzeln gesteuert werden. Die Adressierung entspricht der dreistufigen Hierarchie: die Adresse 8.7.233 bezeichnet z. B. Bereich 8, Linie 7, Teilnehmer 233. Koppler erhalten stets die Teilnehmernummer 0. Pro Linie können 64 Busteilnehmer direkt mit Strom versorgt werden. Zur Erweiterung können maximal drei separat versorgte Linienverstärker eingesetzt werden. Es ergeben sich damit vier Liniensegmente mit bis zu 64 Teilnehmern. Über eine Bereichsline (Backbone) können schließlich noch maximal 15 Bereiche miteinander verbunden werden.

Zusammengehörige Aktoren und Sensoren werden mit einer sogenannten Gruppenadresse verbunden, die einprogrammiert werden kann. Die Kommunikation der Geräte erfolgt über standardisierte Befehle, was sichergestellt, dass Geräte verschiedener Hersteller zusammenarbeiten. Bild 3.6 zeigt die typische Struktur eines KNX-Telegramms.

| Position | Länge in Bit | Bezeichnung |
|---|---|---|
| 1 | 8 | Kontrollfeld |
| 2 | 16 | Adressfeld |
| 3 | 17 | Ziel-Byte |
| 4 | 3 | Routingzähler |
| 5 | 4 | Länge der Nutzinformationen |
| 6 | 0*8...16*8 | Nutzinformationen |
| 7 | 8 | Sicherung |
| **Summe** | **56+(1...16)*8** | = 72...192 bit = 9...23 Byte |

**Bild 3.6:** Struktur eines KNX-Telegramms

Das Kontrollbyte bestimmt die Paket-Priorität und unterscheidet zwischen Standard- und erweitertem Paket (Tabelle 3.1):

**Tabelle 3.1:** Kontrollbyte bei KNX

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Inhalt | 1 | 0 | R | 1 | p1 | p0 | 0 | 0 |

Das Wiederholungsbit R ist beim erstmaligem Senden des Paketes 1, bei einer Wiederholung 0, so dass Teilnehmer, die das Paket bereits korrekt empfangen haben, die Wiederholung ignorieren können. Die Prioritäts-Level werden durch die Bits P1 und P0 festgelegt (Tabelle 3.2):

**Tabelle 3.2:** Prioritätslevel bei KNX

| p1 | p0 | Bedeutung |
|---|---|---|
| 0 | 0 | Systemfunktion |
| 1 | 0 | Alarmfunktion |
| 0 | 1 | hohe Priorität |
| 1 | 1 | normale Priorität |

Die Quelladresse (typische Schreibweise: *Bereich.Linie.Teilnehmer*) besteht aus zwei Byte, wobei das MSB zuerst übertragen wird. Das MSB enthält in den oberen vier Bits den Bereich und in den unteren vier Bits die Linie, das LSB enthält die Teilnehmernummer.

Die Zieladresse adressiert entweder einen individuellen Empfänger (unicast) oder eine Gruppe (multicast). Hier ist die typische Schreibweise: *Hauptgruppe/Mittelgruppe/Untergruppe*. Der Typ der Ziel-Adresse wird im DRL-Byte gesetzt. Bei einer physikalischen Adresse entspricht die Kodierung der Quelladresse. Eine Gruppenadresse wird anders kodiert. Das MSB enthält wieder in den oberen vier Bits die Hauptgruppe, wobei hier das höchste Bit 0 ist. Die unteren vier Bits enthalten die Mittelgruppe. Im LSB ist die Untergruppe gespeichert.

Das DRL-Byte (Destination Address, Routing Counter, Length) ist aus drei Teilen aufgebaut. Das höchstwertige Bit D entscheidet über die Adressierung (D = 0: phys. Adresse, D = 1: Gruppenadresse), danach folgt ein dreistelliger Routing-Counter, der mit 6 initialisiert und von jedem Linien- oder Bereichskoppler dekrementiert wird. Ein Paket mit dem Wert 0 wird verworfen. Ein Wert von 7 verhindert eine Dekrementierung und lässt das Paket beliebig oft weiterleiten. Die letzten vier Bits geben die Länge der folgenden Nutzdaten minus zwei an, d. h. eine Länge von 0 entspricht zwei Bytes und eine Länge von 15 entspricht 17 Bytes.

Die Transport Layer Protocol Control Information (TPCI) beschreibt die Kommunikation auf dem Transport Layer und die Application Layer Protocol Control Information (APCI) sind für die Application Layer Services (Lesen, Schreiben, Antwort, ...) zuständig. Nach den Nutzdaten folgt dann noch die Checksumme, eine invertierte, bitweise XOR-Verknüpfung aller vorher gesendeter Bytes des Paketes.

Quittiert wird ein Paket mit NAK (0x0C), BUSY (0xC0) oder ACK (0xCC).

Weitere Informationen liefern die Webseiten *http://www.knx.org* und *http://www.knx.de*.

# 4

# Chip-Schnittstellen

Häufig erfolgt die Verbindung von integrierten Schaltkreisen innerhalb eines Geräts über einfache serielle Bussysteme. Die Datenübertragung erfolgt dabei mit relativ geringer Geschwindigkeit und einem exakt definierten Protokoll. Für die Verbindung selbst sind nur sehr wenige Signalleitungen erforderlich. Meist reichen schon drei bis vier Leitungen. Es gibt sogar Lösungen mit einer oder zwei Leitungen. Die meisten Systeme erlauben sogar Kabelverbindungen über Boardgrenzen hinweg. An dieser Stelle will ich diese Schnittstellen kurz vorstellen.

## 4.1 Die SPI-Schnittstelle

Der weit verbreitete SPI-Bus (SPI = Serial Peripheral Interface) wurde vor einigen Jahren von Motorola definiert und hat sich inzwischen in der Industrie etabliert. Es gibt zahlreiche integrierte Schaltkreise (EEPROMs, digitale Temperatursensoren, A/D- und D/A-Wandler usw.), die über ein integriertes SPI-Interface verfügen. Auch viele Microcontroller besitzen eine solche Schnittstelle.

### 4.1.1 Arbeitsweise des SPI

Über die SPI-Schnittstelle werden Daten synchron seriell ausgegeben und gleichzeitig eingelesen. Der Bus wird für kurze Distanzen verwendet und arbeitet prinzipiell wie ein Schieberegister. Er hat vier logische Signale:

- SCLK: Serial Clock (output from master).
- MOSI: Master Output, Slave Input (DO, output from master).
- MISO: Master Input, Slave Output (DI, output from slave).
- CS: Chip Select (active low, output from master).

Es ist auch möglich, dass über SPI Daten zwischen zwei Prozessoren ausgetauscht werden. Diese Schnittstelle lässt auf kurzen Übertragungswegen hohe Übertragungsgeschwindigkeiten zu (Bild 4.1). Sender und Empfänger für die SPI-Übertragung lassen sich mit geringem Aufwand (Schieberegister) aufbauen.

Das Grundkonzept des SPI-Bus basiert auf einem Master/Slave-Verfahren und benutzt vier Signalleitungen. Die Datenübertragung erfolgt synchron zu einem Taktsignal, welches stets vom SPI-Master erzeugt wird. Es ist jeweils eine Datensende- und Empfangsleitung vorhanden. Weiterhin gibt es ein Chip-Select-Signal, um einen Slave auszuwählen. Der Master startet die Datenübertragung, indem die Chip-Select-Leitung des Slave auf 0 gelegt wird. Anschließend gibt der Master auf der Taktleitung SCLK eine festgelegte Anzahl von Impulsen aus. Auf der Datenleitung *Master Out Slave In* (MOSI, SIN) gibt der Master bei jedem einzelnen Taktsignal ein Bit des zu übertragenden Bytes aus. Die Datenübernahme durch den Slave erfolgt mit steigender Taktflanke. Gleichzeitig liest der Master

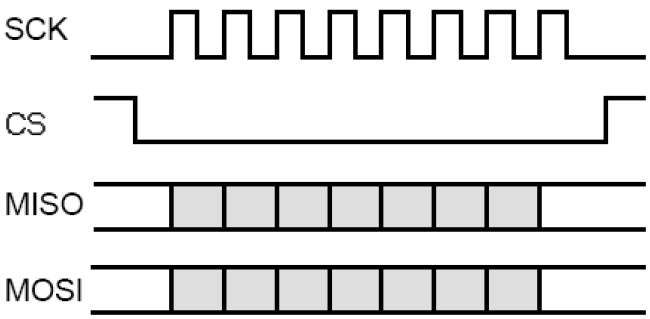



**Bild 4.1:** Serielle SPI-Datenübertragung

bei jedem Taktsignal ein Bit von der Leitung *Master In Slave Out* (MISO, SOUT) ein. Hier erfolgt die Datenübernahme durch den Master mit der fallenden Flanke des Taktes.

Das Senden und Empfangen geschieht also immer gleichzeitig, bei jedem Takt wird ein Bit vom MOSI des Masters versendet und über MISO ein Bit des Slaves empfangen. Schickt also der Master ein Kommando an den Slave, erhält er die Antwort auf die vorherige Anforderung.

Je nach Betriebsart ist ein SPI-Baustein unterschiedlich komplex. Das kann von einem einfachen Schieberegister bis hin zu einem eigenen Subsystem reichen. Das Grundprinzip des Schieberegisters ist allerdings bei jedem Baustein anzutreffen. Die Länge der Schieberegister ist nicht festgelegt, sondern kann von Baustein zu Baustein variieren. Normalerweise sind die Schieberegister acht Bit breit bzw. ein ganzzahlig Vielfaches davon.

Für die Übertragungsparameter gibt es keine Festlegung. So kann die Übertragungsgeschwindigkeit beliebig gewählt und die Bitreihenfolge (MSB oder LSB zuerst) je nach Anwendung eingestellt werden. Für die Synchronisierung der Daten mit dem Taktsignal sind mehrere Möglichkeiten vorgesehen: positive oder negative Flanke, Polarität usw. Die maximale Taktfrequenz hängt von den einzelnen Slavetypen ab.

Eine einfache Slaveschaltung kann mit zwei Schieberegistern aufgebaut werden. Das Empfangs-SR liest die Daten seriell vom Prozessor ein und gibt sie parallel aus, während ein zweites SR die Daten parallel einliest und sie seriell an die CPU weitergibt. Mit dem Signal CS selektiert der Prozessor zuerst die beiden Schieberegister. Anschließend werden durch das Clocksignal SCK die Daten geschrieben bzw. gelesen.

Mehrere Slaves können parallel an die Leitungen SCK, MOSI, MISO angeschlossen werden. Jeder Slave wird jedoch über eine separate CS-Leitung selektiert. Die Slaves versetzen ihre Leitungen in den hochohmigen Zustand, solange sie nicht selektiert sind. So kann der Master mit jeweils einem Slave Daten austauschen. Eine andere Möglichkeit ist die Hintereinanderschaltung mehrerer Slaves (alle CS-Eingänge sind miteinander verbunden). Für den Master erscheint diese Kette dann wie ein sehr langes Schieberegister.

Durch den SPI-Bus ist ein Embedded System recht einfach erweiterbar. Das Embedded System selbst bildet dabei den SPI-Master. Die Programmierung einer SPI-Schnittstelle unter Linux ist sehr einfach, weil in der Regel nur parallele E/A-Bits benötigt werden. Somit sind nur die elementaren E/A-Funktionen notwendig, um aus einem Linux-Programm auf einen SPI-Slave zuzugreifen. Einziger Nachteil: die Leitung zwischen E/A-Port und Peripherie sollte höchstens einige Zentimeter betragen. Man kann aber tricksen und RS232- oder RS485-Pegelwandler einsetzen, um die Verbindung zu verlängern.

SPI-Bausteine lassen sich grob in folgende Kategorien unterteilen:

- Analog-Digital- und Digital-Analog-Wandler
- Speicher (EEPROM und FLASH)
- Real Time Clocks (RTC)
- Sonstige (Signalmixer, Potentiometer, LCD-Controller, UART, Sensoren usw.)

In den ersten drei Kategorien sind die meisten Vertreter zu finden.

### 4.1.2 Programmierung

Als Beispiel für die Programmierung soll der Port-Expander MCP23S17 dienen, der unter anderem zwei 8-Bit-Ports bietet. Das IC wird per SPI an ein Rasberry-Pi-System angeschlossen und ist dank eines SPI-Treibers und einer SPI-Library recht einfach zu programmieren. Das Pin-Layout ist in Bild 4.2 zu sehen. Gegebenfalls hilft ein Blick ins Datenblatt weiter:

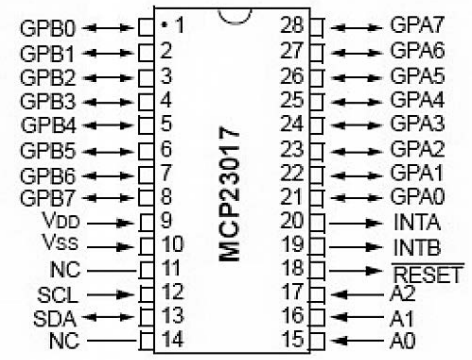



**Bild 4.2:** Der Port-Expander MCP23S17

Um den MCP23S17 ohne Bibliothek per Python-Programm anzusteuern, werden jeweils drei Bytes geschrieben:

1. Die Adresse des MCP23S17. Im folgenden Beispiel wird die Adresse 0x40 verwendet.
2. Das Register, in das geschrieben wird. Im Beispiel ist dies das Register GPIOB für die Steuerung der acht Pins GPB0 bis GPB7.
3. Der Wert, der in das Register geschrieben werden soll.

```
from time import sleep

SPIDEV = '/dev/spidev0.0'
ADDRESS = 0x40
DELAY = 0.1

def write(DEV, Addr, Register, Byte):
#       SPI-Device, Adresse, Register, Daten
  handle = open(DEV, 'w+')
  try:
    data = chr(Addr)+chr(Register)+chr(Byte)
    handle.write(data)
    handle.close
    return True
  except:
    print("Error writing to SPI Bus")
    return False


# MCP23S17-Register GPIOB auf Output schalten
write(SPIDEV,ADDRESS,0x01,0x00)
while True:
  write(SPIDEV,ADDRESS,0x13,0xff)
  sleep(DELAY)
  write(SPIDEV,ADDRESS,0x13,0)
  sleep(DELAY)
```

Das zweite Beispiel verwendet einen Analog-Digital-Wandler MCP3008. Es ist ein 8-Kanal-10-Bit-ADC. Wenn der ADC Spannungen von 0 bis 3,3 V misst, entspricht jeder Schritt im Ausgangswert einer Änderung von 0,003 Volt. Dem Datenblatt kann das SPI-Protokoll entnommen werden (Bild 4.3).

Das erste Programm verwendet, wie schon das vorherige, die SPI-Bibliothek von Python und ist daher sehr einfach:

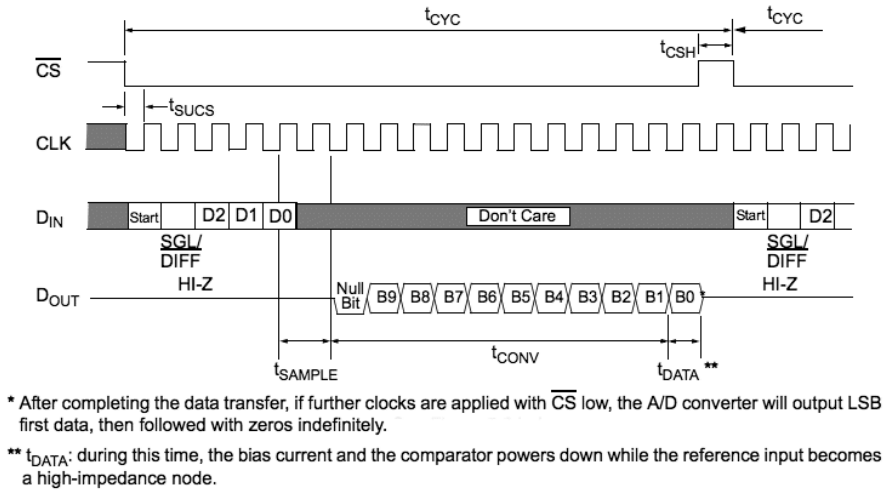



**Bild 4.3:** Ablauf der Kommunikation beim MCP3008

```
#!/usr/bin/env python
import time
import sys
import spidev

spi = spidev.SpiDev()
spi.open(0,0)

def readAdc(channel):
  if ((channel > 7) or (channel < 0)):
    return -1
  result = spi.xfer2([0x01, 0x08|(channel << 4), 0])
  return (result[1] & 0x03) << 8) + result[2]

try:
  while True:
    val = readAdc(0)
    print ("Wert: " + str(val))
    time.sleep(5)
except KeyboardInterrupt:
  spi.close()
  sys.exit(0)
```

Das zweite Beispiel zeigt, wie es auch ohne Library ginge („Bit-Banging“), indem Takt und Daten vom Programm aus gesteuert werden. Dies erfordert eine genauere Kenntnis des Protokolls und eignet sich, um SPI besser kennenzulernen. Der Ablauf stellt sich dann wie folgt dar:

- Die CS-Leitung muss auf High, der Clock-Pin auf Low und der MOSI-Pin Low stehen.
- Setzen der CS-Leitung auf Low, um die Übertragung zu starten.
- Nun fünf Bits an den ADC senden: ein Startbit, gefolgt von einem Befehlsbit, dann die Kanalnummer: Für jedes Bit wird der MOSI-Pin entsprechnd gesetzt und danach ein Taktimpuls erzeugt.
- Nun erfolgt das Einlesen des Analogwertes. Insgesamt sind dazu 12 Taktimpulse notwendig, wobei jedes Mal ein Bit vom MISO-Pin gelesen wird. Das erste Bit ist ein leeres Bit, gefolgt von einem NULL-Bit (das ignoriert werden kann). Die nächsten 10 Bits sind der digitale Wert.
- CS wird nun auf Low gezogen, um die Übertragung zu beenden.

```
#!/usr/bin/env python
import RPi.GPIO as GPIO
import time
import sys

# Port-Definitionen
```

```
CLK = 18
MISO = 23
MOSI = 24
CS = 25

def setupSpi(clkPin, misoPin, mosiPin, csPin):
    # Pins als Eingang/Ausgang definieren
    GPIO.setup(clkPin, GPIO.OUT)
    GPIO.setup(misoPin, GPIO.IN)
    GPIO.setup(mosiPin, GPIO.OUT)
    GPIO.setup(csPin, GPIO.OUT)


def readAdc(channel, clkPin, misoPin, mosiPin, csPin):
  if (channel < 0) or (channel > 7):
    print "Invalid ADC Channel number, must be between [0,7]"
    return -1

  GPIO.output(csPin, GPIO.HIGH)
  GPIO.output(csPin, GPIO.LOW)
  GPIO.output(clkPin, GPIO.HIGH)
  # Steuerbist fuer NAECHSTE Wandlung setzen
  sendBits((0x18 | channel), 5, clkPin, mosiPin)
  # Daten einlesen
  adcValue = recvBits(12, clkPin, misoPin)
  GPIO.output(csPin, GPIO.HIGH)
  return adcValue

def sendBits(data, numBits, clkPin, mosiPin):
  data = data << (8 - numBits)
  for bit in range(numBits):
    # Ausgang setzen
    if data & 0x80:
      GPIO.output(mosiPin, GPIO.HIGH)
    else:
      GPIO.output(mosiPin, GPIO.LOW)
    # Taktimpuls
    GPIO.output(clkPin, GPIO.HIGH)
    GPIO.output(clkPin, GPIO.LOW)
    # naechstes Bit
    data = data << 1

def recvBits(numBits, clkPin, misoPin):
  data = 0
  for bit in range(numBits):
    # Taktimpuls
    GPIO.output(clkPin, GPIO.HIGH)
    GPIO.output(clkPin, GPIO.LOW)
    # Datenbit lesen
    if GPIO.input(misoPin):
      data = data | 1
    # naechstes Bit
    data = data << 1

  # NULL-Bit eliminieren
  return (data/2)

try:
  GPIO.setmode(GPIO.BCM)
  setupSpi(CLK, MISO, MOSI, CS)
  while True:
    val = readAdc(0, CLK, MISO, MOSI, CS)
    print "ADC Result: ", str(val)
    time.sleep(5)
except KeyboardInterrupt:
  GPIO.cleanup()
  sys.exit(0)
```

## 4.2 Der I²C-Bus

Philips entwickelte diese serielle bidirektionale Zweidraht-Schnittstelle für die Kommunikation zwischen verschiedenen ICs in elektronischen Systemen. Daher auch der Name „Inter-IC-Bus“, kurz

„IIC-Bus“ oder I²C-Bus. Das Interface besteht aus den beiden Signalleitungen SDA (serial data) und SCL (serial clock) sowie einer gemeinsamen Masseleitung (ground). Am Bus können mehrere Teilnehmer angeschlossen werden. Jeder der Teilnehmer hat eine vorgegebene sieben Bit lange Adresse (das achte Bit entscheidet, ob gelesen oder geschrieben wird). In den meisten Fällen kontrolliert ein Master die verschiedenen Slaves. Es ist aber auch möglich, mehrere Master an einen I²C-Bus anzuschließen.

Eine weitere Bezeichnung für diese Schnittstelle ist „Two-Wire Interface“ (TWI). Viele Hersteller verwenden diese Bezeichnung, da I²C eine eingetragene Marke des damaligen Erfinders Philips ist. Technisch sind beide Systeme identisch und können auch miteinander kommunizieren.

Eine Abwandlung von I²C ist der SMBus, der System Management Bus. Er ist hardwaretechnisch identisch mit I²C, definiert darauf aber ein anderes Übertragungsprotokoll. Er findet sich auf fast allen modernen PC-Boards, um z. B. die Temperatur der CPU zu messen.

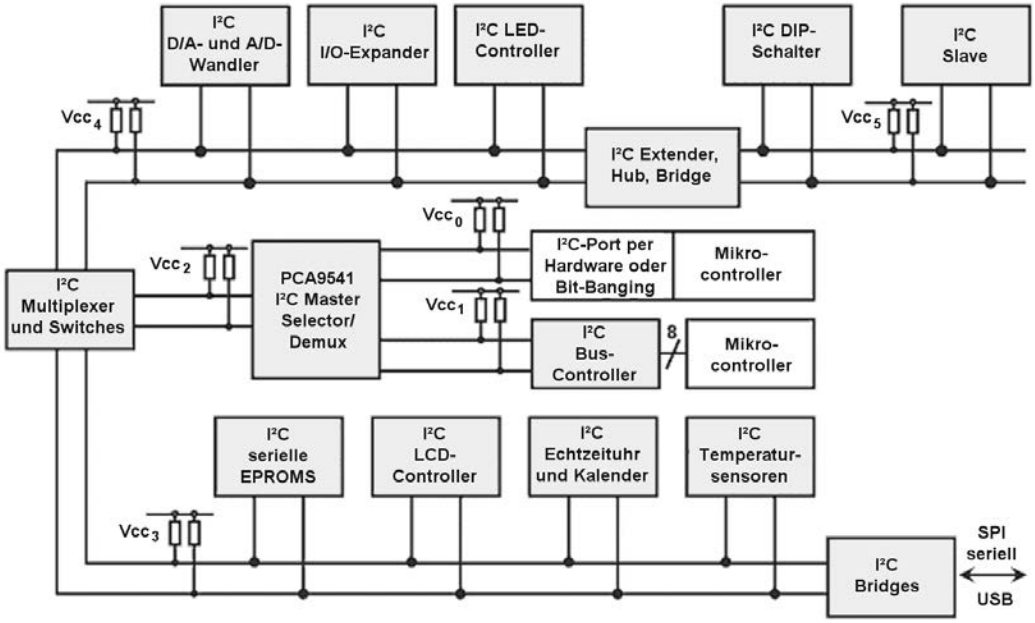



**Bild 4.4:** Überblick der verschiedene Komponenten des I²C-Busses

### 4.2.1 ²C-Bus-Technik

Ursprünglich wurde der I²C Bus entwickelt, um verschiedene Chips in Fernsehgeräten zu steuern (Bild 4.4). Mittlerweile ist der Bus zu einer viel genutzten Kommunikationslösung zwischen Komponenten in Controller-Systemen geworden. Anfangs betrug die Übertragungsrate nur 100 kBit/s bei einer zulässigen Bus-Kapazität von 400 pF. Den ständig steigenden Leistungsanforderungen folgend, wurde die Übertragungsrate später angehoben. Im Laufe der Entwicklung des Busses gab es deshalb drei wichtige Schritte, die sich hauptsächlich auf die Übertragungsrate beziehen:

- 1992, Version 1.0: Erhöhung der Geschwindigkeit von 100 kbit/s auf 400 kbit/s; Erweiterung des Adressraums auf 10 Bit.
- 1998, Version 2.0: Erhöhung der Geschwindigkeit auf maximal 3,4 Mbit/s mittels „Hochgeschwindigkeits-Modus“; weniger Strom- und Spannungsanforderungen.
- 2007, Version 3.0: Erhöhung der Geschwindigkeit auf bis zu 1 Mbit/s.

Ein Teilnehmer kann nur Empfänger oder Sender und Empfänger sein. Der Master kontrolliert die Datenflussrichtung und gibt mit dem Clock-Signal die Übertragungsgeschwindigkeit vor. Die SDA- und die SCL-Leitung werden mit Pull-Up-Widerständen versehen, liegen also im Ruhezustand auf High-Pegel. Die Bus-Teilnehmer sind mit Open-Drain-Schaltungen an den Leitungen angeschlossen und ziehen im aktiven Zustand die Leitungen gegen GND.

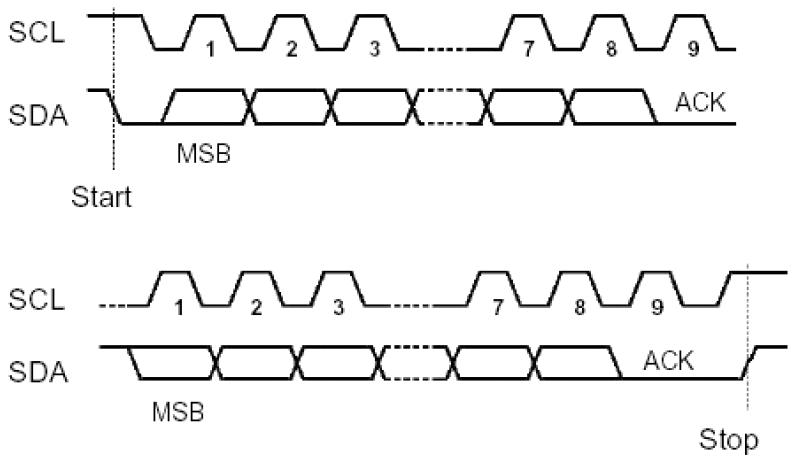



**Bild 4.5:** Serielle I²C-Datenübertragung: Start, Stopp und Acknowledge

Der I²C-Bus nutzt ein einfaches Master-Slave-Konzept. Der Master initialisiert eine Datenübertragung (Start-Bedingung), generiert den Takt (SCL) und beendet die Übertragung (Stopp-Bedingung). Geräte, die vom Master adressiert werden, bezeichnet man als Slaves. Master und Slaves können als Sender und Empfänger arbeiten. Der Master initiiert eine Schreib- oder Lesesequenz. Dabei beginnt jede Übertragungssequenz mit einer sogenannten Start-Bedingung (SDA geht vor SCL auf 0) und endet mit einer sogenannten Stopp-Bedingung (SCL geht vor SDL auf 1). Bild 4.5 zeigt das Verhalten auf dem Bus und Bild 4.6 die Start-und Stopp-Sequenzen. So lange SCL = 1 ist, müssen die Daten stabil anliegen, sie dürfen sich nur ändern, wenn SCL = 0 ist.

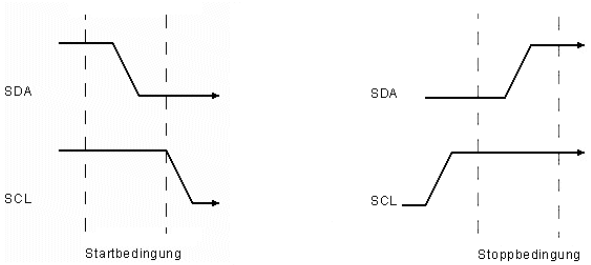



**Bild 4.6:** Serielle I²C-Datenübertragung: Start- und Stopp-Bedingung

Nach der Startbedingung folgt ein Byte mit der Slave-Adresse (Bit 1 bis 7) und dem Read/Write-Bit (Bit 0), das angibt, ob zum Slave geschrieben wird (0) oder von ihm gelesen werden soll (1). Die Kommunikation beginnt immer mit einer Schreibsequenz des Masters. Soll der Slave Daten liefern, folgt anschließend eine Lesesequenz. Nach Senden der Startbedingung gibt der Master das Adressbyte mit R/W = 0 aus. Der Slave antwortet mit ACK (er zieht während Bit 9 der SDA-Leitung auf 0). Danach sendet der Master ein oder mehrere Datenbytes. Nach jedem Datenbyte gibt der Slave eine Bestätigung (ACK). Die Übertragung endet mit einer Stop-Bedingung (Bild 4.7). Das Lesen von Daten erfolgt nach dem gleichen Schema.

Eine Besonderheit ist die Möglichkeit einer erneuten Start-Bedingung des Masters ohne vorherige Stopp-Bedingung. Hierdurch wird eine neue Übertragungssequenz eingeleitet, ohne dass der Bus vorher freigegeben wurde.

Das erste Byte, das der Master nach der Startbedingung verschickt, ist die Adresse des angesprochenen Slaves. Die ursprüngliche 7-Bit-Adressierung ermöglicht bis zu 128 (2¡sup¿7¡/sup¿) Geräte an einem Bus. Damit man nicht nur ein IC eines Typs am Bus haben kann, sondern bei Bedarf auch mehrere, sind bei den meisten Bausteinen die drei unteren Adressbits einstellbar. Dadurch kann man bis zu acht ICs derselben Bauart an den Bus schalten. Eine Adresse besteht somit oft aus einem festen Teil und einer 3-Bit-Subadresse. Das achte Bit des ersten Bytes gibt die Datenrichtung an - es legt fest, ob der Master Daten empfangen oder schicken möchte (R/W-Bit).

Einige Adressen wurden reserviert, um den Bus ausbaufähig zu halten oder um Problemen vorzubeugen.

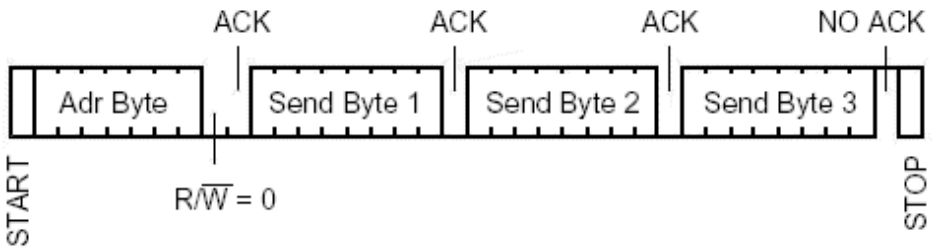



**Bild 4.7:** I²C-Datenformat

| **Adresse** | **R/W-Bit** | **Beschreibung** |
|---|---|---|
| 0000000 | 0 | General Call Adresse |
| 0000001 | X | CBUS Adresse |
| 0000010 | X | Reserviert für ein anderes Busformat |
| 0000011 | X | Für zukünftige Erweiterungen reserviert |
| 00001XX | X | Für zukünftige Erweiterungen reserviert |
| 11111XX | X | Für zukünftige Erweiterungen reserviert |
| 11110XX | X | 10-Bit Adressierung |

Damit man nicht nur ein IC eines Typs am Bus haben kann, sondern bei Bedarf auch mehrere, sind bei den meisten Bausteinen die unteren Adressbits einstellbar. Dadurch kann man bis zu acht ICs derselben Bauart an den Bus schalten. Deswegen und weil die Adressen nur sieben Bit lang sind, ergibt sich ein ziemliches Gedränge im Adressraum. Zudem sind zwei Achterblöcke reserviert. So gibt es kaum eine der dann noch möglichen 112 Adressen, die nicht doppelt oder dreifach belegt sind. Aus diesem Grund wurde später ein erweiterter Addressierungsmodus eingeführt, der 10-Bit-Adressen erlaubt. Jedoch gibt es nicht viele ICs mit einer solchen Adresse. Bild 4.8 zeigt die Adressierung mit 10 Bit.

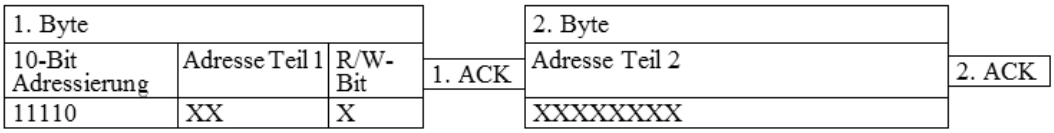



**Bild 4.8:** 10-Bit-Adressierung bei I²C

Jedes über den Bus gesendete Byte muss vom Empfänger bestätigt werden. Der Master muss für die Bestätigung einen extra Taktpuls generieren. Der Sender setzt nach dem übertragenen Byte die Datenleitung auf 1. Zur Bestätigung zieht der Empfänger die Datenleitung auf 0. Möchte (oder kann) der Empfänger kein weiteres Byte mehr empfangen, lässt er SDA auf 1 und sendet damit keine Bestätigung. Der Master reagiert darauf mit einer Stopp-Bedingung und beendet die Datenübertragung. Das bedeutet auch, dass der Master für jedes Byte neun Taktimpulse erzeugt: acht für den Datentransfer und einen für die Bestätigung.

### 4.2.2 5 V ⇔ 3,3 V Pegelwandler

Um 5-V-Peripherie an einem Controller mit 3,3-V-Stroimversorgung zu betreiben (oder auch umgekehrt), ist ein Pegelwandler (neudeutsch „level shifter“) notwendig. Soll die Wandlung nur in einer Signalrichtung erfolgen, ist die Lösung recht einfach mit einem Schalttransistor zu realisieren. Etwas anders verhält es sich mit bidirektionalen Verbindungen wie z. B. beim I²C-Bus. Hier gibt es eine Lösung mittels Feldeffekttransistoren. Bild 4.9 zeigt die Schaltung mit jeweils einem N-Kanal-MOSFET pro Leitung.

Die Gate-Anschlüsse der MOSFETs werden mit 3,3 V verbunden, die Sorce-Anschlüsse mit den Signalleitungen der 3,3-V-Seite und die Drain-Anschlüsse mit den Signalleitungen auf der 5-V-Seite. Dabei werden MOSFETS eingesetzt, deren Substrat bereits mit dem Source-Anschluss intern verbunden ist: Andernfalls müsste diese Verbindung extern erfolgen. Die im Schaltbild gezeichnete Diode

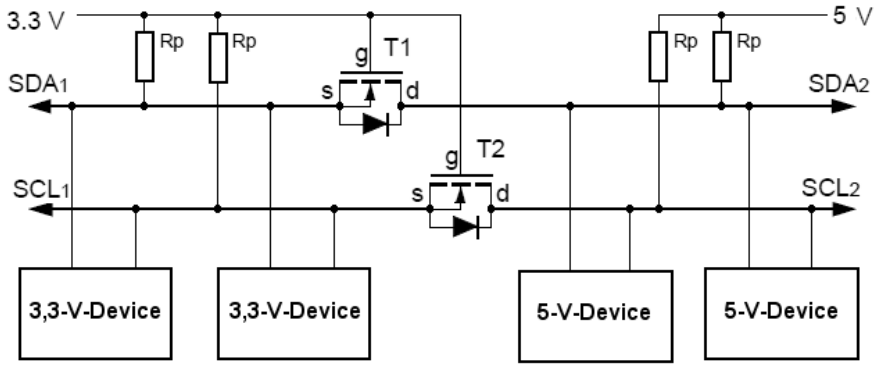



**Bild 4.9:** Pegelwandlung mit nur zwei MOSFETs

zwischen Source und Drain ist innerhalb des MOSFET als PN-Übergang zwischen Substrat und Drain enthalten. Der MOSFET leitet, wenn der Source-Anschluss (3,3-V-Seite) auf Massepegel gezogen wird (bzw. unter etwa 1 V sinkt). Damit gibt es drei mögliche Schaltzustände:

- Liegt der Bus beidseitig auf High-Pegel, sperrt der MOSFET und die beiden Seiten liegen über die Pullup-Widerstände auf 3,3 V bzw. 5 V.
- Wird die 3,3-V-Seite auf Low gezogen leitet der MOSFET, weil das Gate ja weiter auf 3,3 V liegt. Damit zieht er auch die 5-V-Seite auf Low.
- Wenn dagegen die 5-V-Seite auf Low gezogen wird, leitet die Drain-Substrat-Diode im MOSFET und zieht den Source-Anschluss auf der 3,3-V-Seite auf etwa 0,7 V (Low), so dass der MOSFET leitet und so auch die 3,3-V-Seite auf Low-Level zieht.

Das Prinzip funktioniert natürlich auch mit anderen Spannungen. Die höchste Schaltfrequenz wird durch die parasitären Kapazitäten der MOSFETs bestimmt. Für die Pegelwandlung beim I²C-Bus sind alle Typen mit folgenden Eigenschaften geeignet:

| | |
|---|---|
| Typ: | N-Kanal MOSFET |
| Gate threshold voltage: | $V_{GS}$(th) min. 0.1 V max. 2 V |
| On resistance: | $R_{DS}$(on) max. 100 Ω bei $I_D$ 3 mA, $V_{GS}$ 2.5 V |
| Input capacitance: | max. 100 pF bei $V_{DS}$ 1 V, $V_{GS}$ 0 V |
| Switching times: | $T_{on}/T_{off}$ max. 50 ns |
| Allowed drain current: | $>$ 10 mA |

Geeignete Typen sind BSN10, BSN20, BSS83, BSS88, BSS138. Der Wert der Pull-Up-Widerstände ist unkritisch, er hängt vom maximal benötigten Strombedarf ab. Hier haben sich Werte von 4,7 bis 10 Kiloohm bewährt.

Die 3,3-V-Seite kann abgeschaltet werden, ohne dass der Betrieb auf der 5-V-Seite beeinträchtigt wird. Die linke Seite (3,3 V) ist somit beim Abschalten der Versorgung isoliert. Insofern kann die Schaltung auch verwendet werden um „externe“ Komponenten zu entkoppeln. In diesem Fall können auf beiden Seiten auch identische Spannungen zu Versorgung dienen.

Sollen beide Seiten unabhängig voneinander abgeschaltet werden können, muss auch die 5-V-Seite isoliert werden. Dazu muss die Schaltung wie in Bild 4.10 erweitert werden.

Die Schaltung arbeit wie die vorhergehende. Wenn jedoch hier die 5-V-Versorgung abgeschaltet wird, ist die rechte Seite isoliert, ohne dass die linke Seite (3,3 V) davon beeinträchtigt wird. Die Schaltung ist nun auch symmetrisch, so dass es nun egal ist, auf welcher Seite 3,3 V und 5 V angeschlossen werden. Jede der beiden Seiten ist von der anderen isoliert, sobald deren Versorgung abgeschaltet wird.

Die Schaltung nach Bild 4.10 arbeitet in der dargestellten Form einwandfrei. Mann kann jedoch, um das „Floating“ der Drain-Anschlüsse im High-Zustand zu vermeiden, die Drain-Anschlüsse von T1/T3 und T2/T4 über Pullup-Widerstände mit der höheren der beiden Versorgungsspannungen (meist 5 V) verbinden (im Bild strichliert gezeichnet).

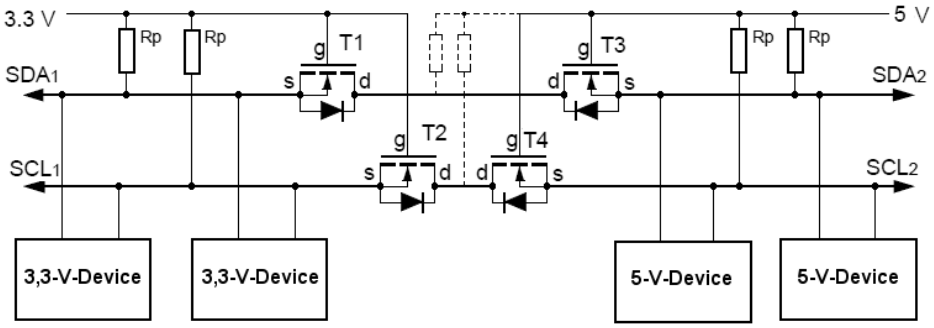



**Bild 4.10:** Symmetrische Pegelwandler-Schaltung

Aber es gibt auch massenhaft integrierte Bausteine für diesen Zweck, etwa den Level-Shifter TCA9406 von Texas Instruments. Er arbeitet bidirektional, daher ist es egal, auf welcher Seite welche Spannung verwendet wird. Über einen Output-Enable-Pin kann er entweder vom Master aktiviert werden (Bild 4.11) oder man legt den Pin einfach auf High-Potential.

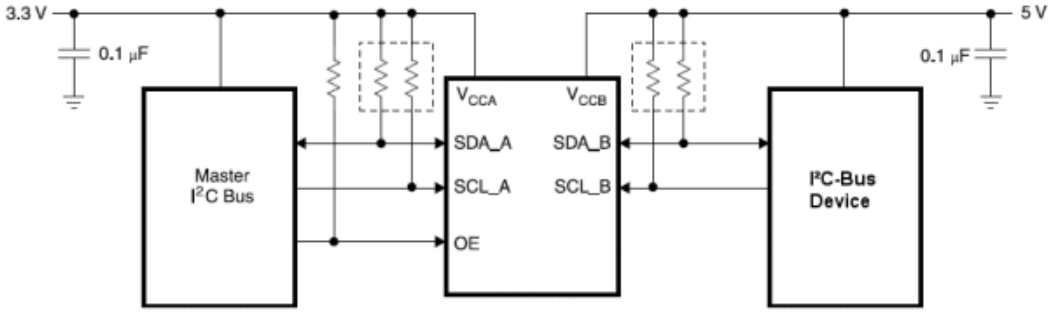



**Bild 4.11:** Integrierte Pegelwandler-Schaltung TCA9406

### 4.2.3 Lange Leitung

Bei vielen Projekten, die den I²C-Bus über Entfernungen von mehreren Metern einsetzen, wird ein Leitungsverstärker vom Typ P82B715 nötig. Dieser Chip vergrößert den Busstrom und damit die Reichweite. Der P82B715 puffert hochkapazitive I²C-Bussysteme und unterstützt die bidirektionale Datenübertragung über den I²C-Bus.Er ermöglicht die Erweiterung des I²C-Busses unter Beibehaltung aller Betriebsmodi und Funktionen des I²C-Systems. Jeweils nach dem Master und vor jedem Slave muss ein solcher Verstärker eingebaut werden.

Beim P82B715 handelt es sich um einen sogenannten I²C-Bus-Extender, der die Reichweite des Busses erweitert. Er ist im DIL8- oder SO8-Gehäuse erhältlich. Bild 4.12 zeigt die Pinbelegung des Bausteins. Die Pins Sx und Sy werden an den vorhandenen seriellen Bus angeschlossen, die Pins Lx und Ly sind für die Leitung vorgesehen. Dabei ist es gleichgültig, welchen von beiden Pins Sie für Daten- oder Taktleitung verwenden, sie dürfen nur keine Kreuzung produzieren.

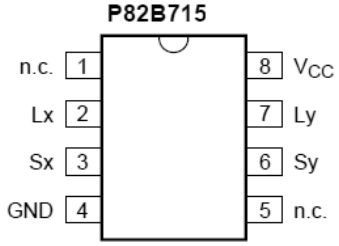



**Bild 4.12:** Integrierter Busextender P82B715

Seine Eigenschaften sind:

- Versorgungsspannungsbereich von 3 V bis 12 V

- Unterstützt die bidirektionale Datenübertragung von I²C-Bus-Signalen (Bild 4.13)
- Erlaubt Buskapazität von 400 pF auf Haupt-I²C-Bus-Seite (Sx/Sy) und 3000 pF auf Leitungsseite (Lx/Ly)
- Dual bidirektionaler Unity-Voltage-Gain-Puffer ohne externe Richtungssteuerung
- 10 mal niederohmigere Busverdrahtung für verbesserte Störfestigkeit
- Multi-Drop-Verteilung von I²C-Signalen unter Verwendung von kostengünstigen Twisted-Pair-Kabeln
- I²C Busbetrieb über 50 Meter Twisted-Pair-Leitung
- Latchup-Leistung übertrifft 100 mA

Der P82B715 eignet sich für den Einsatz mit vielen seriellen Bussen und beschränkt sich keineswegs ausschließlich auf den I²C-Bus. Daher reicht seine Betriebsspannung von 3 bis 12 Volt, er unterstützt Busse mit bis zu 400 kHz Taktrate. Zur Berechnung der Pullup-Widerstände gibt das Datenblatt wertvolle Hinweise (Default: 470 Ohm).

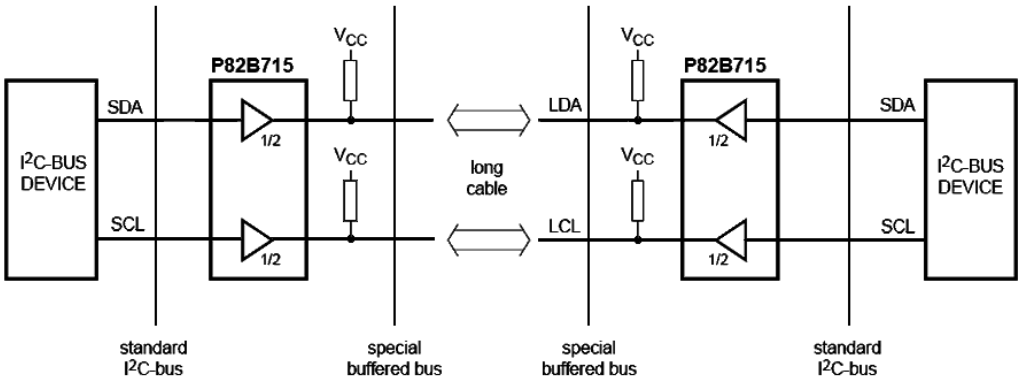



**Bild 4.13:** Anschlussschema des Busextenders P82B715

Es sei noch darauf hingewiesen, dass der I²C-Bus inzwischen auch auf vielen Mainboards von PCs zu finden ist (z. B. für die Temperaturüberwachung der CPU oder ähnliche Aufgaben).
Weitere Informationen zum I²C-Bus:

- I2C-Bus.org: *http://www.i2c-bus.org/*
- The I2C-Bus Specification, Version 5, October 2012:
  *http://www.nxp.com/documents/user_manual/UM10204.pdf*
- The I2C-bus and how to use it
  *http://www.mcc-us.com/i2cHowToUseIt1995.pdf*
- Philips Application Note AN97055 (Bi-directional level shifter for I²C-bus and other systems)
  *http://ics.nxp.com/support/documents/interface/pdf/an97055.pdf*

### 4.2.4 Programmierung

Die Programmierung ist beim I²C-Bus komplexer als bei SPI, weshalb man hier nahezu immer auf einen Treiber und ggf. auf eine Bibliothek zurückgreift. Grundvoraussetzung ist jedoch, dass ein passender Treiber installiert ist, sonst bliebe nur eine komplexe Programmierung im Bitbang-Modus übrig. Ist ein Treiber eingebunden werden für die Programmierung in C im Prinzip nur fünf Funktionen benötigt:

- `open()` für den Zugriff auf den I²-Geräteknoten,
- `ioctl()` für die Einstellung von Parametern des I²-Slaves,

- `read()` und write() für die eigentliche Kommunikation mit dem Slave und
- `close()` für die Beendigung des Zugriffs auf den $I^2$-Geräteknoten.

Unter Linux wird die Header-Datei `linux/i2c-dev.h` benötigt. Mit deren Hilfe wird der Zugriff auf den Geräteknoten mit `open()` realisiert. Danach muss man noch die Slave-Adresse mit einem Aufruf von `ioctl()` setzen:

```
# include <linux/i2c-dev.h>
  ...
fd = open("/dev/i2c-1", O_RDWR);
ioctl(fd, I2C_SLAVE, 0x48);
  ...
```

Nun kann man mit `read()` und `write()` lesen und schreiben.

Auch für Python gibt es eine passende Bibliothek, die man auf dem Raspberry Pi gegebenenfalls mittels `sudo apt-get install python-smbus` nachinstallieren muss. Das erste Beispiel soll aber wieder zeigen, dass es genügt, den Treiber über die Dateischnittstelle anzusprechen. Dieses Beispiel und das folgende lesen einen Luftfeuchte- und Temperatursensor HDC1008 aus (für nähere Informationen sei auch hier auf das Datenblatt verwiesen):

```
#!/usr/bin/env python
import struct, array, time, io, fcntl, sys

I2C_SLAVE=0x0703
HDC1008_ADDR = 0x43
bus = 1

fr = io.open("/dev/i2c-"+str(bus), "rb", buffering=0)
fw = io.open("/dev/i2c-"+str(bus), "wb", buffering=0)

# set device address
fcntl.ioctl(fr, I2C_SLAVE, HDC1008_ADDR)
fcntl.ioctl(fw, I2C_SLAVE, HDC1008_ADDR)
time.sleep(0.015) # 15ms startup time

# set config register
s = [0x02,0x02,0x00]
s2 = bytearray(s)
fw.write(s2) #sending config register bytes
time.sleep(0.015) # From the data sheet

# read temperature
s = [0x00]
s2 = bytearray(s)
fw.write(s2)
time.sleep(0.0625) # From the data sheet
data = fr.read(2)  # read 2 byte temperature data
buf = array.array('B', data)
temp = ((((buf[0]<<8) + (buf[1]))/65536.0)*165.0) - 40.0
time.sleep(0.015)  # From the data sheet

# read humidity
s = [0x01]
s2 = bytearray(s)
fw.write(s2)
time.sleep(0.0625) # From the data sheet
data = fr.read(2)  # read 2 byte temperature data
buf = array.array('B', data)
humid = ((((buf[0]<<8) + (buf[1]))/65536.0)*100.0)

print("%7x %7.2f %7.2f" % (HDC1008_ADDR, temp, humid))
fr.close()
fw.close()
```

Mit der SMBus-Bibliothek wird das Programm wesentlich kompakter:

```
#!/usr/bin/env python
import smbus
import time

# number of I2C bus
```

```
BUS = 1
HDC1008 = 0x43

# Get I2C bus
bus = smbus.SMBus(BUS)

# set config register
bus.write_byte_data(HDC1008, 0x02, 0x00)
time.sleep(0.015) # From the data sheet

# read temperature
bus.write_byte(HDC1008, 0x00)
time.sleep(0.0625)  # From the data sheet

# read temp data
data0 = bus.read_byte(HDC1008)
data1 = bus.read_byte(HDC1008)

# convert the data
temp = ((((data0<<8) + data1)/65536.0)*165.0) - 40.0
time.sleep(0.015)  # From the data sheet

# read humidity
bus.write_byte(HDC1008, 0x01)
time.sleep(0.0625)  # From the data sheet

# convert the data
data0 = bus.read_byte(HDC1008)
data1 = bus.read_byte(HDC1008)
humid = (((data0<<8) + data1)/65536.0)*100.0

print("%7x %7.2f %7.2f" % (HDC1008_ADDR, temp, humid))
```

## 4.3 Der 1-Wire-Bus

1-Wire ist ein digitaler, serieller Bus des Herstellers Maxim (ehem. Dallas), der mit einer Datenader und einer Masseleitung auskommt. Die Bezeichnung 1-Wire leitet sich daraus ab, dass zu der ohnehin vorhandenen Masseleitung nur eine weitere Ader für die gesamte Buskommunikation und die Energieversorgung erforderlich ist. Die Form der Energieübertragung bei diesem Bus heißt „parasitic power“: Wenn gerade keine Daten übertragen werden, saugt sich der Chip seine Energie aus der aktiven Leitung und speichert sie in einem kleinen Kondensator, der während der Sendeaktivität zur Überbrückung dient.

1-Wire wird auch als „MicroLAN“ bzw. „Single Wire Serial Interface“ bezeichnet. 1-Wire eignet sich insbesondere für Sensorik (Temperaturmessung, Akkuüberwachung, Spannung, Temperatur, Stromfluss), zur Steuerung und Meldung und für Identifikation durch einmalige, eindeutige und nicht veränderbare 64-Bit-Seriennummern. Jeder Chip besitzt eine eigene Zeitbasis, die die ankommenden Signale nach ihrem logischen Gehalt unterscheiden kann. Für den Einsatz in Zahlungssystemen und industriellen Zugangskontrollen wurde eine Art elektronischer Metall-Tablette mit der Bezeichnung „iButton“ entwickelt, die äußerst widerstandsfähig gegenüber äußeren Einflüssen ist (Bild 4.14, Info unter *http://www.ibutton.com*).

Ursprünglich für die Kommunikation zwischen den Komponenten eines Gerätes bzw. Schaltschrankes entwickelt, wurde der 1-Wire-Bus zu einem einfach handhabbaren Bussystem für Strecken bis zu mehreren hundert Metern erweitert. Dies wurde insbesondere durch verbesserte Hostinterfaces mit Kontrolle der Signalanstiegszeit und aktivem Pull-Up Widerstand sowie durch Rauschfilter und ein optimiertes Timing erreicht. Jeder Chip besitzt eine eigene Zeitbasis, die die ankommenden Signale nach ihrem logischen Gehalt unterscheiden kann. Die Datenübertragung erfolgt in Blöcken zu 64 Bit und ist bidirektional, seriell, asynchron und Halbduplex, da dieselbe Datenleitung für Senden und Empfangen benutzt wird. Die gesamte Kommunikation wird durch den Busmaster gesteuert. Hierbei können mehrere Dutzend Slaves an einem Busmaster angeschlossen werden. Die Geschwindigkeit der Datenübertragung reicht von 15,4 kbps (Standard) bis 125 kbps (Overdrive). Hierbei ist „parasitic Power“ nur bei der Standard-Übertragungsrate möglich, welche jedoch völlig ausreichend ist:


**Bild 4.14:** Dallas/Maxim iButton

- Um eine logische 1 an den Chip zu senden, wird der Pegel der Datenleitung vom Master für maximal 15 µs auf Masse gezogen.
- Eine logische 0 entspricht einem Low-Pegel von mindestens 60 $\mu$s.

Der zeitliche Abstand zwischen beiden Zuständen ist groß genug, um eventuelle Produkttoleranzen zu überbrücken. Der 1-Wire-Bus ist also ein PWM-kodiertes Bussystem.

Der Bus unterstützt bis zu $2,8 * 10^{14}$ verschiedene Chips der gleichen Familie. Zur Unterscheidung besitzt jedes Chip eine eindeutige 64 Bit lange Identifikationsnummer: eine 8-Bit-CRC-Prüfinfo, eine 48-Bit-Seriennummer und einen 8-Bit-Family-Code. Der erste Dallas-Chip war ein (serielles) ROM, das nur die obige Information enthielt (DS2401) und den Chip zu einer Art elektronischer Seriennummer zum Beispiel für Leiterplatten oder andere elektronische Geräte machte. Neben diesen reinen ID-Chips existiert eine recht große und stetig wachsende Anzahl von verschiedensten 1-Wire-Chips, angefangen vom einfachen EEPROM über Echtzeituhren mit Eventcounter, Temperatursensoren bis hin zum leistungsfähigen Mehrfach-Analog-Digitalwandler.

Der Busmaster kann unter anderem eigenständig nach allen mit dem Bus verbundenen Chip-IDs suchen oder allen angeschlossenen Chips gleichzeitig einen Befehl schicken. Je mehr Slaves sich in einem seriellen Bus befinden, desto stärker wird er durch den Datenfluss elektrisch und informationstechnisch belastet – die Chips sind ja zur Versorgung auf eine Mindestpausenzeit angewiesen. Abhilfe schaffen beim 1-Wire-Bus spezielle Busteilerbausteine, die den Bus in mehrere kleinere Äste unterteilen.

Bevor vom Master ein Kommando an einen Slave gesendet wird, muss der Slave per ROM-Adresse selektiert werden. Es kann, wie oben erwähnt, Auch an alle Slaves ein Kommando gesendet werden („skip ROM"-Kommando). Dies ist nur zulässig für Kommandos, die keine Antwort von allen Slaves erzeugt – mit einer Ausnahme: wenn der Master die IDs der angeschlossenen Slaves ermitteln will (siehe unten).

Um 1-Wire-Sensoren mit „parasitic Power" zu betreiben, muss anhand des Datenblatts geprüft werden, ob ein 1-Wire-Sensor für die Stromversorgung über die Datenleitung geeignet ist. Dies ist insbesondere bei Multisensoren (Temperatur, Luftfeuchte, Druck, Umgebungslicht) nicht immer der Fall. Es werden lediglich zwei Adern am Busmaster angeschlossen: GND und DATA (DQ). Der Versorgungsan-schluss Vcc eines jeden Sensors ist mit GND zu verbinden. Der Sensor bezieht seine Spannungsversorgung hierbei parasitär aus der Datenleitung. Damit hier ein ausreichend großer Strom geliefert werden kann ist ein Busmaster mit Strong-Pullup erforderlich. Bild 4.15 zeigt das Schema.

Zu beachten ist, dass nur Sensoren deren Datenblatt diesen Betrieb vorsieht, verwendet werden können. Weiterhin muss die gesamte Kabellänge pro Bus unter 100 m betragen und es sind maximal 20 Sensoren pro Bus möglich. Um alle Möglichkeiten zur späteren Erweiterung für Sensoren die eine zusätzliche Spannungsversorgung benötigen offen zu halten, sollte man möglichst Kabel mit drei oder vier Adern verlegen.

Bei Verwendung anderer Sensoren müssen separat 5 V (stabilisiert) eingespeist werden. Dazu werden alle drei Anschlüsse an das separate Netzteil und den Busmaster entsprechend folgender Abbildung

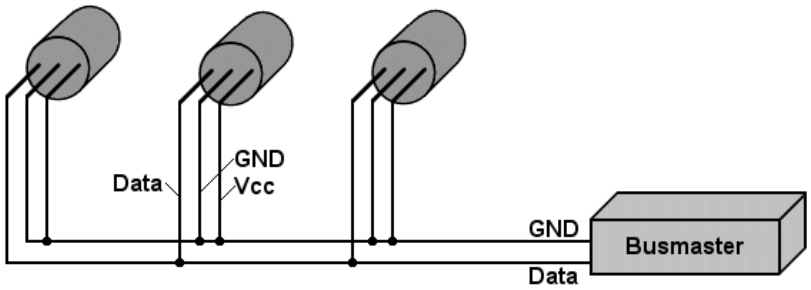



**Bild 4.15:** 1-Wire-Sensoren mit „parsitic power“

angeschlossen. Der GND-Anschluss des Netzteiles wird mit GND des Busmaster verbunden. Beide Installationsvarianten können auf dem gleichen Bus auch gemischt betrieben werden. Bei den parasitär betriebenen Sensoren wird hierbei lediglich der Vcc-Anschluss auf GND gelegt (BIld 4.16).

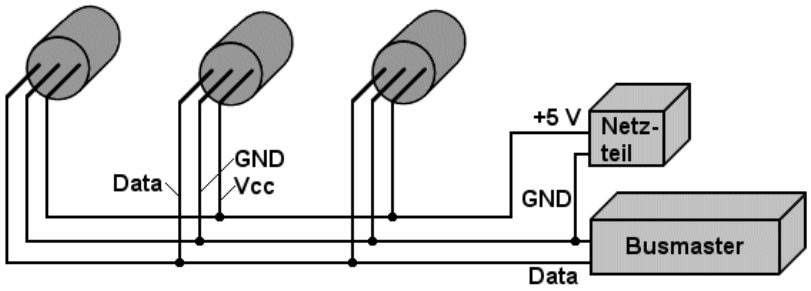



**Bild 4.16:** 1-Wire-Sensoren mit separater Stromversorgung

Zur Verkabelung kann beispielsweise normale Fernmeldeleitung (Telefonkabel) J-Y(St)Y 2x2x0,6 verwendet werden. Für Data und GND ist jeweils ein verdrilltes Paar zu verwenden (bei J-Y(St)Y sind dies meist jeweils rot/schwarz und gelb/weiß, bei anderen Kabeln braun/weiß und gelb/grün). Sollen die Sensoren nicht fest verdrahtet werden, haben sich RJ12/RJ45-Stecker als Quasi-Standard herausgebildet. Die folgende Tabelle zeigt ein der möglichen Belegungen:

| **Farbe (J-Y(St)Y)** | **Funktion** | **RJ-12** | **RJ-45** |
|---|---|---|---|
| Schwarz (sw) | GND | 4 | 5 |
| Rot (rt) | Data | 3 | 4 |
| Gelb (ge) | Vcc | 1 | 2 |

Zum Aufbau einer Masterstation für einen 1-Wire-Bus gibt es grundsätzlich drei verschiedene Möglichkeiten:

- Der Mikrocontroller selber übernimmt direkt die Rolle des Masters. Es ist also auf der Mikrocontroller-Seite lediglich ein freier digitaler bidirektionaler I/O-Port-Pin mit Open-Drain-Eigenschaft notwendig, der dann extern über einen 4,7-Kiloohm-Pullup-Widerstand an die 5V-Betriebsspannung gelegt wird. Das gesamte 1-Wire-Datenübertragungsprotokoll wird vom Anwender selbst auf dem Mikrocontroller programmiert. Diese ist die einfachste und preiswerteste Realisierung für einen 1-Wire-Master.
- Es wird ein spezieller externer Chip als 1-Wire-Master eingesetzt (z. B. der DS2482 integrated 1-Wire Driver), der an den Mikrocontroller über eine I2C-Bus-Verbindung a ngekoppelt ist. Dieser Baustein entlastet den Mikrocontroller von der kompletten Abwicklung des 1-Wire-Datentransfers und es werden nur die eigentlichen Nutzdaten zwischen den beiden Bausteinen ausgetauscht, die der 1-Wire Driver dann normgerecht aussendet bzw. bereits empfangen hat.

- Maxim stellt dem Anwender auch eine Funktionsbibliothek bzw. Funktionsbeschreibung für einen kompletten 1-Wire-Master zur Verfügung, die dieser dann in einen selbst erstellen, applikationsspezifischen ASIC-Baustein einbinden und somit die 1-Wire-Funktionalität mit in diesen Chip integrieren kann.

Weiterführende Informationen bietet die Webseite von Maxim/Dallas unter *http://www.maxim-ic.com/products/1-wire* bzw. unter *http://www.ibutton.com* .

Mit dem Kommando **Search ROM** lassen sich die IDs der angeschlossenen Slaves ermitteln. Durch dieses Kommando und einen schlauen Algorithmus lassen sich Schritt für Schritt alle ROM-IDs der Slaves auslesen. Einmaliges Ausführen des Kommandos und dessen Datenaustausch liefert eine ROM-ID. Um alle IDs zu erfahren, muss mehrmals nach dem Search-ROM-Algorithmus gesucht werden. Der prinzipielle Ablauf des Algorithmus besteht aus folgenden drei Schritten, die beim LSB der ID beginnen und entsprechend oft wiederholt werden müssen:

- Der Master liest ein Bit der ROM-ID.
- Der Master liest das komplementäre Bit.
- Der Master sendet das ausgewählte Bit.

Sind mehrere Slaves angeschlossen (müssen also mehrere ROM-IDs ermittelt werden), geschieht folgendes: Nach der Anfrag des Masters senden alle Slaves ihr erstes Bit (LSB, niederwertigstes Bit). Da die Leitung per Pullup-Widerstand auf logisch „1“ liegt, ergibt sich eine sogenannte Wired-AND-Verknüpfung (nur wenn alles Slaves eine „1“ liefern, ergibt sich eine „1“ als Ergebnis). Bei Übertragung einer „0“ wird die Datenleitung länger auf LOW gehalten, als bei einer „1“. Bei gleichzeitiger Übertragung einer „1“ dominiert die „0“ und der Master empfängt eine „0“. Anschließend übertragen die Slaves das Komplement des ersten Bits. Nun können vier Fälle auftreten (jeweils für jede Bit-Position):

| **Slave-Bit** | **Komplement** | **Resultat** |
|---|---|---|
| 0 | 0 | Es gibt sowohl „0“- als auch „1“-Bits |
| 1 | 0 | Alle Bits sind „1“ |
| 0 | 1 | Alle Bits sind „0“ |
| 1 | 1 | Es gibt kein Device in diesem Zweig |

Im ersten Fall (0, 0) muss die Software zwei Zweige verfolgen, zuerst alle Slaves mit einer „1“ an dieser Bitposition und dann alle Slaves mit einer „0“. Im zweiten und dritten Fall geht es nur in einer Richtung weiter. Im letzten Fall (1, 1) gibt es keinen Slave, der an der Suche teilnimmt. Nach jedem gelesenen Bit muss sich der Master also für eine Suchrichtung entscheiden, indem er nun selbst ein Bit sendet:

- Sendet er eine „0“, so nehmen am Identifikationsprozess nur noch die Slaves weiterhin teil, deren ROM-ID an dieser Bitposition eine „0“ haben.
- Sendet er eine „1“, so nehmen am Identifikationsprozess nur noch die Slaves weiterhin teil, deren ROM-ID an dieser Bitposition eine „1“ haben.

Die nicht selektierten Slave gehen bis zum nächsten Reset-Kommando in einen „Sleep“- oder „Wait“-Zustand und nehmen nicht mehr an der Kommunikation teil.

Würde der Master nun bei Stellen, die eine Wahl zwischen „0“ und „1“ erlauben (wo allso beide Werte an der Bitposition vorkommen) immer wieder den „0“-Zweig verfolgen, hätte er nach 64 Schritten die ID des ersten Slaves ermittelt. Im nächsten Durchlauf kann er dann an der ersten „Wegegabelung“ die „1“ verfolgen und dann wieder die „0“, um so zur zweiten ID zu gelangen. Das kann so lange wiederholt werden, bis die IDs aller Slaves ermittelt sind.

Wie bei mehreren Slaves deren ID ermittelt wird, steht ganz ausführlich in der Applikation Note von Maxim: „1-Wire Search Algorithm“ unter *http://www.maximintegrated.com/en/app-notes/index.mvp/id/187* .

## 4.4 ZACwire

ZACwire von Innovative Sensor Technology ist dem 1-Wire-Protokoll sehr ähnlich. Es gibt derzeit nur Temperatursensoren für dieses Protokoll. Auch hier spielt das Tastverhältnis zwischen 0- und 1-Signalen eine Rolle. Im Gegensatz zum 1-Wire ist hier das Datenprotokoll Byte-orientiert. Um ein Byte zu senden wird – ähnlich wie bei RS232 – ein Startbit vorangestellt. Statt des Stoppbits folgt jedoch nach acht Datenbits ein Paritätsbit. Werden mehrere Bytes hintereinander gesendet, wird nach jedem Byte eine Stoppbit von einer halben Bitperiode Dauer eingeschoben, das logisch 1 ist.

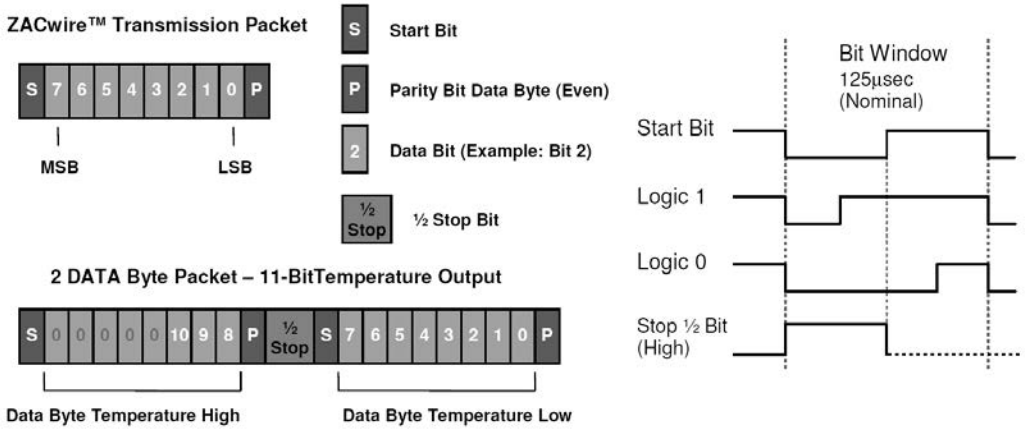



**Bild 4.17:** Das Datenformat von ZACwire

De Sensoren TSic206 oder TSic306 senden die Temperaturdaten, sobald die Versorgungsspannung anliegt. Die Daten bestehen aus zwei Bytes. Die Datenleitung ist im Ruhezustand auf 1-Pegel. Die Übertragungsdauer für ein Bit ist ca. 125 Mikrosekunden. Der Unterschied zwischen logisch 0 und logisch 1 wird über das Tastverhältnis gesteuert. Bei 0 beträgt dieses 25%, 1 ist es 75%. Das Startbit hat ein Tastverhältnis von 50%. Es legt auch das Timing des Datentelegramms fest und sollte vor jedem übertragenen Byte erneut festgestellt werden. Bild 4.17 zeigt links das Datenformat für ein Byte und für die Temperatursensoren und rechts das Timing für die einzelnen Bits.

Für das Lesen der Daten empfiehlt das Datenblatt folgendes Vorgehen: Wenn die fallende Flanke des Startbits auftritt, misst man die Zeit bis zur steigende Flanke des Startbits. Diese Zeit (Tstrobe) entspricht einer halben Bitperiode. Tritt die nächste fallende Flanke auf, wartet man eine Zeitdauer gleich Tstrobe und testet dann das Signal. Der vorliegende Datenwert entspricht zu diesem Zeitpunkt dem Wert des übertragenen Bits. Bild 4.18 zeigt das 2-Byte-Signal des oben genannten Temperatursensors.

Da jedes Bit mit einer fallenden Flanke beginnt, kann das Abtastfenster jeweils nachjustiert werden. Wenn das Lesen nicht flankengetriggert, sondern durch kontinuierliches Abtasten erfolgt, wird empfohlen, die Abtastrate mindestens das 16-fache der Zeit Tstrobe betragen soll. Da die nominale Schrittgeschwindigkeit 8 kHz beträgt, sollte daher mit mindestens 128 kHz abgetastet werden.

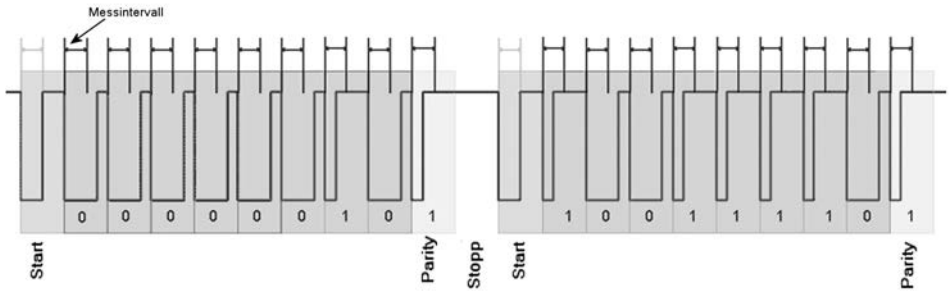



**Bild 4.18:** Das Datenformat einer Messung des Temperatursensors

Weitere Informationen liefert das Dokument *http://www.ist-ag.com/eh/ist-ag/resource.nsf/imgref/Download_ZACWireAppNotes.pdf/$FILE/ZACWireAppNotes.pdf*

# 5

# S0-Schnittstelle

Die S0-Schnittstelle („S-Null-Schnittstelle“) ist eine Hardware-Schnittstelle für die Messwertübermittlung in der Gebäudeautomatisierung. Sie ist in DIN 43864 bzw. EN 62053-31 definiert. Sie hat trotz des gleichen Namens nichts zu tun mit dem S0-Bus bei einer ISDN-Telefon-Installation.

Die Übertragung der Daten erfolgt mit Hilfe von gewichteten Impulsen, was nichts weiter bedeutet, als das pro Kilowattstunde oder Kubikmeter ein Impuls übertragen wird. Die Gewichtung ist vom verwendeten Zählertyp abhängig. Die nachfolgenden DDC-Einheiten (DCC: Direct Digital Control) sammeln die Impulse und generieren daraus einen darstellbaren Verbrauchswert. Die Schnittstelle ist in Wasserzählern, Gaszählern, Stromzählern und Wärmezählern anzutreffen (Bild 5.1).

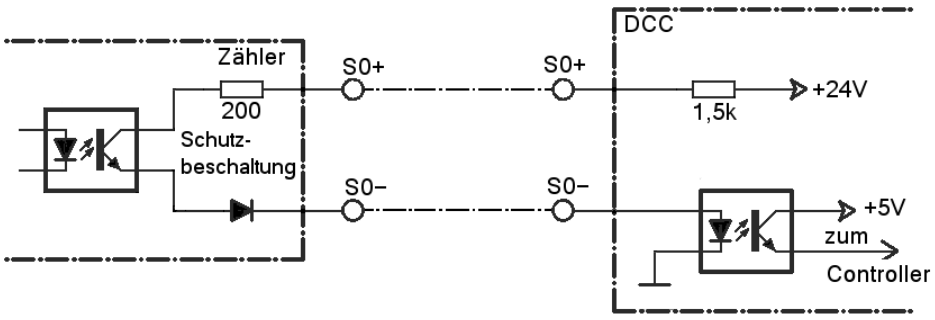



**Bild 5.1:** Schaltung der S0-Schnittstelle

Der Ausgang ist der offene Kollektor des Fototransistors eines Optokopplers. Die Schnittstelle liefert daher kein elektrisches Potential und beim Anschluss muss auf die Polarität geachtet werden. Es gibt zwei Klassen der S0-Schnittstelle, A für lange und B für kurze Übertragungswege. In Klasse B können bis zu 15 V Gleichspannung, in Klasse A bis zu 27 V Gleichsspannung angeschlossen werden. Der maximale Stromfluss wird mit 15 mA bzw. 27 mA angegeben, dies entspricht einem Widerstand von 1 Kiloohm. Ein Stromfluss von kleiner 2 mA entspricht einer logischen „0“, ein Strom größer als 10 mA einer logischen „1“. Gängige DDCs arbeiten einwandfrei mit diesen Schwellen. Das Signal wird vom Zähler als Impulsfolge ausgegeben, wobei ein einzelner Impuls mindestens 30 ms andauert. Die folgenden Tabellen fassen alle wichtigen Werte zusammen.

| **Maximalwerte** | | **Timing** | |
|---|---|---|---|
| Spannungsfestigkeit | 5 kV | Anstiegszeit | $\leq$ 5 ms |
| Maximale Spannung | DC 27 V | Abfallzeit | $\leq$ 5 ms |
| Maximaler Strom | DC 27 mA | Impulsdauer | $\geq$ 30 ms |
| Maximale Leitungslänge | 0,5 m (nach DIN) | Impulspause | $\geq$ 30 ms |

Auch wenn nach DIN die Leitungslänge nur bis zu einem halben Meter betragen darf, geben manche Hersteller eine maximale Leitungslänge von bis zu 10 m an.

# Anhang

## A.1 Literatur

### A.1.1 Schnittstellen


- Jan Axelson: *Serial Port Complete*, Lakeview Research
- Jan Axelson: *USB Complete*, Lakeview Research
- Jürgen Hulzebosch: *USB in der Elektronik*, Franzis
- H.-J. Kelm (Hrsg.): *USB 2.0*, Franzis
- S. Furchtbar, W. Hackländer: *$I^2C$-Bus angewandt*, Elektor


### A.1.2 Schaltungstechnik


- Dieter Zastrow: *Elektronik*, Vieweg-Verlag
- G. Koß, W. Reinhold, F. Hoppe: *Lehr- und Übungsbuch Elektronik*, Fachbuchverlag Leipzig
- U. Tietze, Ch. Schenk: *Halbleiter-Schaltungstechnik*, Springer-Verlag
- Helmut Lindner: *Taschenbuch der Elektrotechnik und Elektronik*, Hanser
- E. Prohaska: *Digitaltechnik für Ingenieure*, Oldenbourg
- Ch. Siemers, A. Sikora: *Taschenbuch der Digitaltechnik*, Hanser
- Don Lancaster: *Das CMOS-Kochbuch*, VMI Buch AG
- Don Lancaster: *TTL-Cookbook*, Sams Publishing
- Hans-Dieter Stölting, Eberhard Kallenbach: *Handbuch Elektrische Kleinantriebe*, Hanser
- Elmar Schrüfer: *Elektrische Messtechnik*, Hanser
- *Zeitschrift Elektor*, Elektor-Verlag, Aachen
- *Elrad-Archiv 1977–1997 DVD*, eMedia GmbH, Hannover


## A.2 Links

**PC-Schnitstellen:** *http://www.netzmafia.de/skripten/hardware/PC-Schnittstellen/*

**Serielle Schnitstelle:** *http://www.netzmafia.de/skripten/hardware/PC-Schnittstellen/seriell.html*

**Raspberry Pi:** *http://www.netzmafia.de/skripten/hardware/RasPi/*

**Das Elektronik-Kompendium:** *http://www.elektronik-kompendium.de/*

**Kabel- und Stecker-FAQ:** *http://www.kabelfaq.de/*

**Maxim-Datenblätter:** *http://www.maxim-ic.com* und *http://datasheets.maxim-ic.com*

**Datenblätter aller Art:** *http://www.datasheets.org.uk/* und *http://www.alldatasheet.com*

**Einführung in SPS:** *http://www.studet.fh-muenster.de/~diefrie/einfh.html*

# Stichwortverzeichnis