Wie kommunizieren die Teilnehmer untereinander ?

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

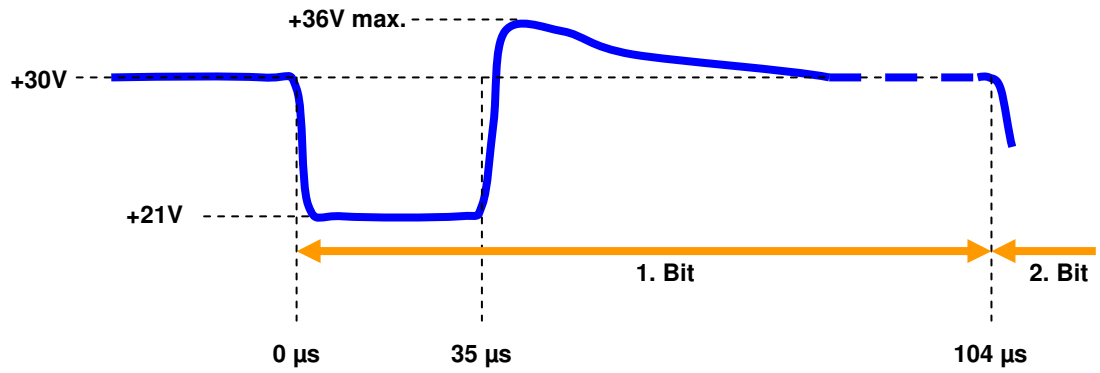
Was passiert denn nun auf dem Bus ?

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

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

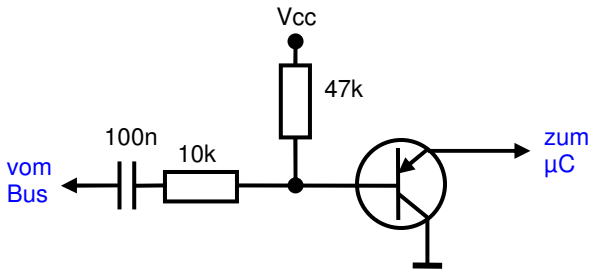
- Eine **logische 1** wird gesendet, indem „gar nichts“ passiert. Der Pegel bleibt unverändert während der Dauer von 104µs auf ca. 30V.

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



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

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

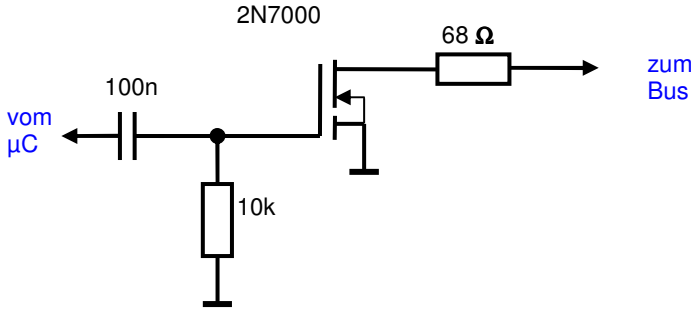


Dabei entsteht ein sauberes Rechteck-Signal mit gleicher Polarität wie das Original, d.h. im Ruhezustand ist der Controllerpin auf High. Sofern der verwendete Mikrocontroller keinen internen pull-up Widerstand hat, muss man diesen extern vorsehen, typischerweise 10k gegen Vcc. Das Niveau von ca. 21V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

Wie sendet der Mikrocontroller ein Bit ?

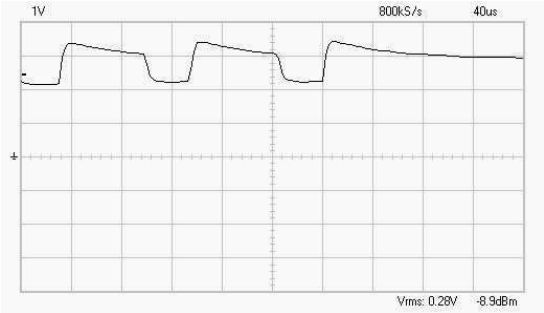
Zum Senden eines Bits muss der Mikrocontroller eigentlich nur bei einer logischen Null etwas tun. „Eigentlich“ deswegen, weil zum Senden einer logischen 1 zwar das Bussignal nicht aktiv verändert wird, während dieser Zeit jedoch geprüft wird, ob eine Kollision vorliegt. Dazu aber später mehr.

Zum Senden einer Null muss wie gesagt ein gewisser Strom fließen um die Busspannung absinken zu lassen. Das wird mittlerweile nicht mehr durch Übertrager realisiert, sondern über einen FET. Der folgende Schaltplan zeigt die Sende-Schaltung:



Während 35µs sendet der µC einen High-Pegel, der den FET durchschalten lässt. über den 68Ω Widerstand fließt nun der nötige Strom um den Spannungspegel auf den geforderten Wert absinken zu lassen. Da nur für sehr kurze Zeit ein Strom fließt reicht ein normaler 1/4-Watt Widerstand. Sicherheitshalber verhindert der Kondensator, dass bei permanentem High-Pegel der FET dauerhaft durchschaltet und den Widerstand zum Glühen bringt. Der 10k Widerstand sorgt für ein sicheres Sperren des FET.

Die mit dieser Schaltung erreichte Signalform auf dem Bus ist im folgenden Oszillogramm (Tastkopf 1:10) zu sehen und identisch mit dem Idealverlauf:



Bits… schön und gut – aber wie sieht denn nun ein Byte aus ?

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

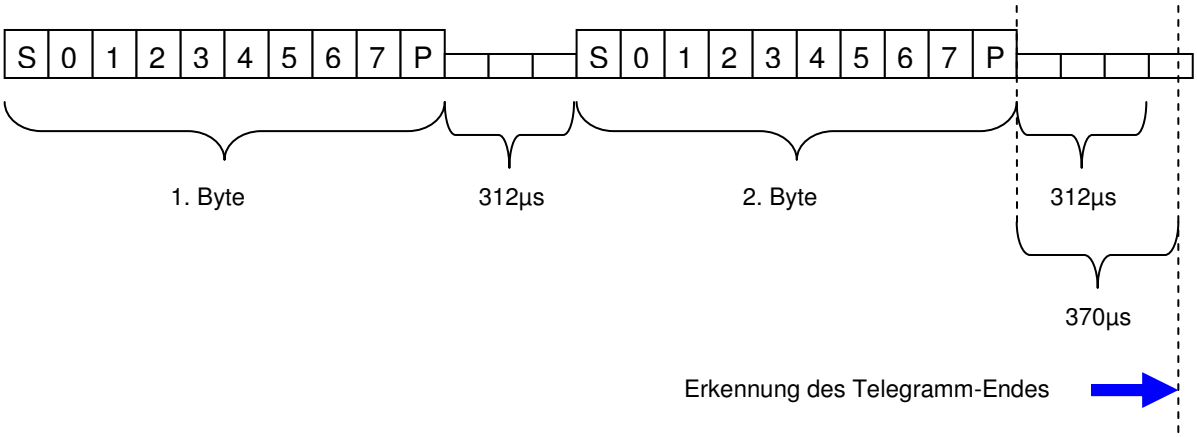
Das Parity-Bit wird gebildet, indem die Anzahl Einsen der 8 Datenbits gezählt wird. Ist sie ungerade ist das Parity-Bit 1.

S	0	1	2	3	4	5	6	7	P	S

Damit dauert das Senden eines Byte (11+2) x 104µs = 1352µs.

In der Praxis wird man das Stopbit nicht bemerken, da es eine logische 1 ist und sich somit von der anschließenden Pause nicht unterscheidet. Also wird man nach dem Senden des Parity-Bit einfach 312µs Pause machen und dann mit dem Senden des nächsten Byte fortfahren.

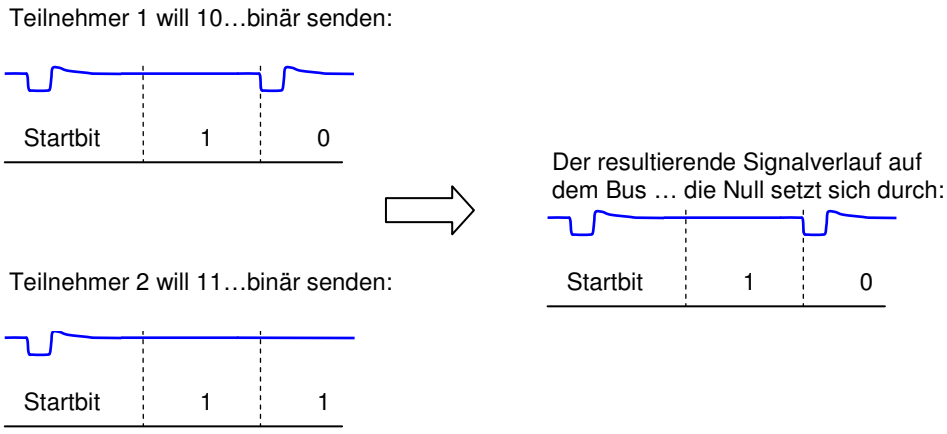
Das genaue Einhalten der Pausenlänge ist wichtig. Wird nämlich während einer Zeit von 370µs nach Ende des Parity-Bits nichts übertragen, so geht man davon aus, dass das Telegramm vollständig ist. Sollte hingegen ein weiteres Byte gesendet werden, wird dessen Startbit nach 312µs den „count-down“ stoppen.



Kollisionen vermeiden

Das Buszugriffsverfahren, das hier verwendet wird nennt sich CSMA/CA. Dabei ist das CA für „collision avoidance“ entscheidend. Es gibt auch z.B. ein CSMA/CD Verfahren, bei dem Kollisionen auftreten können, die die Daten unbrauchbar machen. Dies wird dort erkannt und die Daten nochmals gesendet. Bei uns geht es mit CSMA/CA aber von Anfang an darum Kollisionen zu vermeiden.

Kollisionen werden auf Bit-Ebebne erkannt und dementsprechend auch während einer Bit-Übertragung reagiert. Eine Kollision liegt also nur dann vor, wenn ein Teilnehmer eine 1 und ein anderer eine 0 zur gleichen Zeit sendet. Es stehen uns also nur die ersten 35µs eines Bits zur Verfügung dies zu erkennen. Und die Aufgabe hat jeder Teilnehmer, der eine 1 sendet, denn dabei sollte sich auf dem Bus ja nichts ändern (s.o.). Diese Teilnehmer lesen also im Wirklichkeit währen des „Sendevorgangs“. Wenn eine 0 gelesen wird liegt also eine Kollision vor und der Teilnehmer bricht seinen Sendevorgang sofort ab. Der Teilnehmer, der die 0 gesendet hat, kriegt von alle dem nichts mit und fährt einfach fort.



In diesem Beispiel muss der Teilnehmer 2 erkennen, dass während des Sendens der zweiten 1 auf dem Bus tatsächlich eine 0 übertragen wurde. Er bricht daraufhin seinen eigenen Sendevorgang ab und wiederholt diesen später, sobald der Bus frei ist. Teilnehmer 1 hingegen setzt seinen Sendevorgang ungestört fort und bekommt von alledem nichts mit.

Wie man sieht, ist dies also ein sehr effektives Verfahren, denn eigentlich findet ja gar keine richtige Kollision statt. Somit verliert man nicht unnötig kostbare Zeit.

Jetzt könnte man sich die Frage stellen, ob o.g. Beispiel nicht der Idealzustand einer Kollision ist, denn die beiden Teilnehmer senden ihre Bits ja jeweils synchron zur gleichen Zeit. Könnte es also sein, dass die Teilnehmer versetzt gegeneinander senden und somit die Bits des Einen in die Pausen des Anderen rutschen? Um es vorwegzunehmen… eigentlich nicht. Es gibt nämlich eine Spielregel, die da besagt: „Bevor ich auf den Bus senden darf, muss ich lauschen und während 5,1ms keine Aktivität wahrnehmen. Dann ist der Bus frei und ich kann senden.“ Das heißt, dass wenn ein Teilnehmer bereits angefangen hat sein Startbit zu senden, darf kein Anderer einen Sendeversuch starten. Zwei Teilnehmer können also nur dann beide senden wollen, wenn sie exakt zur gleichen Zeit anfangen die Busaktivität zu überwachen. Nur dann würden beide gleichzeitig ihren Sendevorgang starten und damit lägen die Bits exakt übereinander.