Mikrocontrollertechnik

MODUL D

Copyright ©

Inhaltsverzeichnis MODUL D

D0 Wiederholung

Die Grundlagen der Assembler-Programmierung aus dem Modul C sollen anhand einiger Programmieraufgaben gefestigt werden.

Kurze Zusammenfassung EIA232

- Die **serielle Schnittstelle** (EIA-232) wird auch heute noch sehr häufig eingesetzt. Damit eine Kommunikation richtig ablaufen kann, muss das **gleiche Datenformat** (Anzahl der Datenbits, gleiche Parität, gleiche Anzahl von Stoppbits) und die **gleiche Übertragungsgeschwindigkeit** (Baud bzw. Bit/s) verwendet werden. Werden zwei Endgeräte (z.B. PC und Controller) eingesetzt, so müssen die Leitungen gekreuzt werden (Null-Modem-Kabel). Oft werden bei proprietären Lösungen auch Steuerleitungen (z.B. **CTS**, **RTS**) eingesetzt. Diese sind richtig anzuschließen.
  Die Schnittstelle arbeitet mit anderen Pegeln wie der Controller!! Es muss ein Schnittstellen-Wandlerbaustein (z.B. MAX232) eingesetzt werden, da sonst der Controller zerstört werden kann.
- Die meisten ATmega-Controller besitzen eine Hardware-Schnittstelle. Diese wird mit **UART** oder **USART** bezeichnet (*Universal Synchronous/Asynchronous Receiver/Transmitter*). Die USART des ATmega32A wird mit Hilfe von 4 SF-Registern initialisiert. Für die Baudrate wird ein Teilerwert in dem Doppelregister **UBRRH**:**UBRRL** abgelegt wird. Das Datenformat wird im Register **UCSRC** eingestellt. Sender und Empfänger sowie die Interrupts werden in **UCSRB** eingeschaltet. Beim Polling werden die Flags im Statusregister **UCSRA** abgefragt. Das Datenregister heißt **UDR**.
- Daten werden am Pin **PD0** (**RxD**) eingelesen, und am Pin **PD1** (**TxD**) ausgegeben. Wurde der Sende- bzw. Empfängerbaustein eingeschaltet so ist das entsprechende Pin automatisch richtig als Eingang oder Ausgang initialisiert.
- Die Übertragungsgeschwindigkeit der seriellen Schnittstelle wird beim Controller durch Teilung seiner Taktfrequenz eingestellt. Für eine sichere Übertragung über die serielle Schnittstelle sollte ein externer Quarz verwendet werden und die Abweichung zur erwünschten Übertragungsgeschwindigkeit nach Datenblatt 0,5 %[1] nicht überschreiten, da sonst häufiger Fehler auftreten können. Bei kurzen Leitungen können aber ohne weiteres höhere Abweichungen (±2 %) toleriert werden.

1 Nach EIA-232 Standard sollte die Schnittstelle bis zu einer Abweichung von ±4% bei Baudraten unter 19200 Baud funktionieren.

Software-UART

Ein wichtiger Schritt bei der Programmierung ist die Fehlersuche (*Debugging*). Für die Ausgabe von Fehlermeldungen oder Registerinhalten ist die serielle Schnittstelle erste Wahl. Leider ist die einzige serielle Hardware-Schnittstelle oft schon vom Projekt in Beschlag genommen. Eine Lösung ist die Verwendung eines AVR-Controllers mit zwei seriellen Hardware-Schnittstellen wie der zum ATmega32A pinkompatible und sparsame ATmega644p[2].

Eine einfachere alternative Lösung ist eine Software-Emulation der Schnittstelle. Sie hat zudem folgende Vorteile:

- Jedes beliebige Pin kann zum Senden verwendet werden. Dies gilt auch beim Empfang mittels Polling.
- Jede Baudrate ist bei guter Programmierung mit geringer Abweichung realisierbar unabhängig vom verwendeten Quarz.

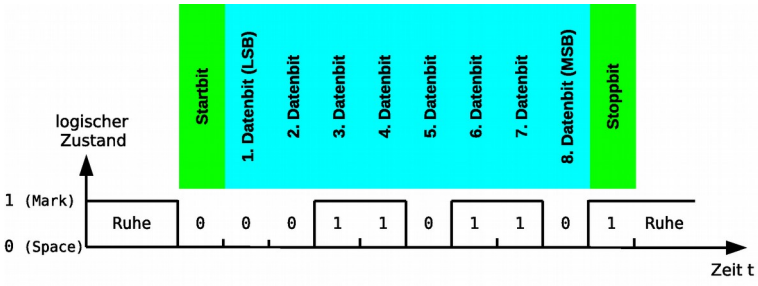
Natürlich gibt es auch Nachteile, da sonst keine Hardware-UARTs existieren würden:

- Bei der Emulation kann der Controller während des Sendens oder des Empfangs mittels Polling sonst keine Aufgaben wahrnehmen.
- Bei hohen Übertragungsgeschwindigkeiten müssen eventuell spezielle Unterprogramme, mit entsprechenden Zeitschleifen, geschrieben werden, um die Fehlerquote klein zu halten.
- Der Code fällt umfangreicher aus.

Software-UART zum Senden.

Das folgende Unterprogramm zeigt eine mögliche Lösung. Die Baudrate kann mittels einer Zuweisung beliebig eingestellt werden. Das Datenformat beträgt fest 8N1, es sind also 10 Bit zu senden. Nach dem Startbit kommt das Zeichen (LSB *first*) und dann ein Stoppbit.

Beispiel:



Es wird eine 16 Bit Zeitschleife verwendet. Zur Berechnung des Anfangswerts des Schleifenzähler G wird die genaue Formel benutzt:

2 Beim ATmega644p ist darauf zu achten, dass die SF-Register sich teilweise im SRAM befinden. Sie müssen dann mit **`sts`** und **`lds`** anstatt von **`out`** und **`in`** adressiert werden. **`sbi`**, **`cbi`**, **`sbic`** und **`sbis`** können dann nicht verwendet werden. Es muss auf die Bitmaskierung zurückgegriffen werden.

$$G = \frac{t_{Bit} - t_T}{4t_T} = \frac{t_{Bit}}{4t_T} - \frac{1}{4}$$

Mit

$$t_{Bit} = \frac{1}{Baud} \quad und \quad t_T = \frac{1}{Takt}$$

ergibt sich:

$$G = \frac{Takt}{4\text{Baud}} - \frac{1}{4}$$

„Takt“ ist die Taktfrequenz des Controllers (bei uns 16 MHz) und „Baud“ die Bitrate der seriellen Schnittstelle). Beim Startbit und beim Stoppbit benötigt der **`cbi`** bzw. der **`sbi`**-Befehl einen Taktzyklus, so dass ein Viertel (Zeit eines Taktzyklus) abgezogen werden muss. Zur Berechnung des **Zeitschleifenzähler**s ergibt sich damit für das **Start-** bzw. das **Stoppbit** die folgende Formel:

$$G_{Start} = G_{Stopp} = \frac{Takt}{4\text{Baud}} - \frac{2}{4}$$

Im Hauptteil des Unterprogramms wird das Zeichen, das sich im Register **`r24`**[3] befinden muss, 8 mal (Zähler in **`r25`**) mit dem Schiebebefehl **`lsr`** nach Rechts geschoben. Das niederwertigste Bit (2^0) wird dabei in das Übertragsbit (**C**arry, Statusregister **`SREG`**) geschoben. Ein bedingter Sprungbefehl bewirkt dann, dass die Leitung bei einer Null auf Masse bzw. bei einer Eins auf VCC gezogen wird. Nach 8 Schiebeoperationen wurde dann das ganze Zeichen versendet.

Da bei der Verzweigung, je nach Bit eine unterschiedliche Anzahl von Befehlen verwendet wird, muss ein **`nop`**-Befehle als Zeitausgleich eingefügt werden. Die Verarbeitung der Daten benötigt 8 Taktzyklen. Für den **Zeitschleifenzähler** der **8 Datenbit** ergibt sich folgende Formel:

$$G_{Datenbit} = \frac{Takt}{4\text{Baud}} - \frac{9}{4}$$

Da die Berechnungen in Assembler mittels einer Ganzzahldivision erfolgt, findet keine Rundung statt und es kann ein Fehler von 4 Taktzyklen (1 Schleifendurchgang) entstehen. Dies ist bei einer Baudrate von 115200 Baud nicht hinnehmbar, da ein Fehler von 4 Taktzyklen/139 Taktzyklen*100 = 2,9 % entstehen könnte. Um dies zu vermeiden wird die Berechnung mit 10 multipliziert (und später wieder durch 10 geteilt). Durch eine Addition von 5 kann dann eine richtige Rundung durchgeführt werden.

Erweiterte Formeln für den Zeitschleifenzähler:

3 Das Doppelregister **`r24`**:**`r25`** ist hier für die Parameterübergabe reserviert und wird deshalb auch nicht auf den Stapel gerettet.

$$G_{Start} = G_{Stopp} = \frac{\frac{Takt*10}{4Baud} - \frac{2*10}{4} + 5}{10}$$

$$G_{Datenbit} = \frac{\frac{Takt*10}{4Baud} - \frac{9*10}{4} + 5}{10}$$

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Zuweisungen
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EQU    TXDDR   = DDRC          ;DDR-Register fuers Senden
.EQU    TXPORT  = PORTC         ;Port-Register fuers Senden
.EQU    TXPNr   = 1             ;Pin Nummer fuers Senden

.EQU    Baud    = 115200        ;Baudrate
.EQU    Clock   = 16000000      ;Systemtakt (CPU)

;-------------------------------------------------------------------------------
;       Sende 1 Byte (r24) ueber die serielle Schnittstelle (8N1)
;-------------------------------------------------------------------------------
SSNDC:  push    XL              ;rette verwendete Register
        push    XH

        sbi     TXDDR,TXPNr     ;Ausgang
        sbi     TXPORT,TXPNr    ;Leitung auf 1
        ldi     r25,8           ;Zaehler = 8

        cbi     TXPORT,TXPNr    ;Startbit senden
        ;Zeitschleife fuer das Startbit
        ldi     XL,LOW(((((10*Clock)/(4*Baud))-((2*10)/4)+5)/10))
        ldi     XH,HIGH(((((10*Clock)/(4*Baud))-((2*10)/4)+5)/10))
SSNDC1: sbiw    XL,1
        brne    SSNDC1
        ;Sende Zeichen LSB first!
SSNDC2: lsr     r24             ;Bit in den Uebertrag (Carry) schieben
        brcs    SSNDC3          ;Carry-Bit testen
        cbi     TXPORT,TXPNr    ;Null
        rjmp    SSNDC4          ;zur Zeitschleife
SSNDC3: sbi     TXPORT,TXPNr    ;Eins
        nop                     ;Zeitausgleich
SSNDC4: ;Zeitschleife fuer ein Datenbit
        ldi     XL,LOW(((((10*Clock)/(4*Baud))-((9*10)/4)+5)/10))
        ldi     XH,HIGH(((((10*Clock)/(4*Baud))-((9*10)/4)+5)/10))
SSNDC5: sbiw    XL,1
        brne    SSNDC5
        dec     r25             ;falls nicht alle Bits durch, dann naechstes Bit
        brne    SSNDC2

        nop                     ;Zeitausgleich für Sprungbefehl (MSB)
        sbi     TXPORT,TXPNr    ;Stoppbit senden
        ;Zeitschleife fuer das Stoppbit
        ldi     XL,LOW(((((10*Clock)/(4*Baud))-((2*10)/4)+5)/10))
        ldi     XH,HIGH(((((10*Clock)/(4*Baud))-((2*10)/4)+5)/10))
SSNDC6: sbiw    XL,1
        brne    SSNDC6

        pop     XH              ;Register wiederherstellen
        pop     XL
        ret                     ;zurueck
```

⌂ **D000**

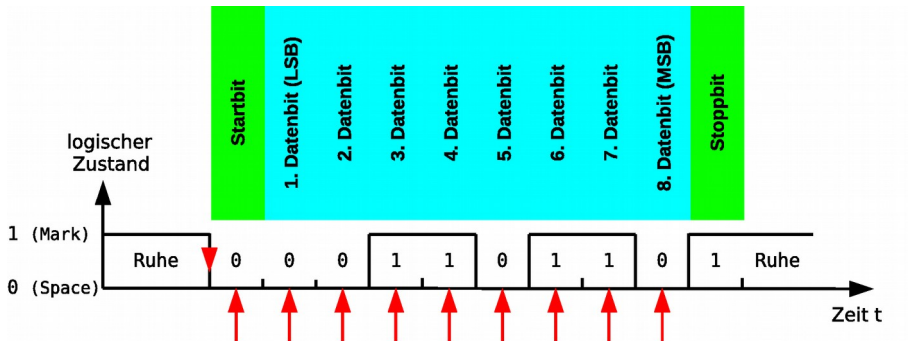
a) Zeichne das Flussdiagramm zum Unterprogramm.
b) Das obige Unterprogramm befindet sich in der Bibliothek: "**SR_UART_SOFT_sample.asm**".
Schreibe ein Hauptprogramm zum Testen des Unterprogramms. Gib dem Hauptprogramm den Namen "**D000_UART_SOFT_test1.asm**".
c) Welche maximale Abweichung ergibt sich bei einer Baudrate von 115200 Baud und bei einer Baudrate von 9600 Baud? Sind diese Abweichungen hinnehmbar? Begründe.

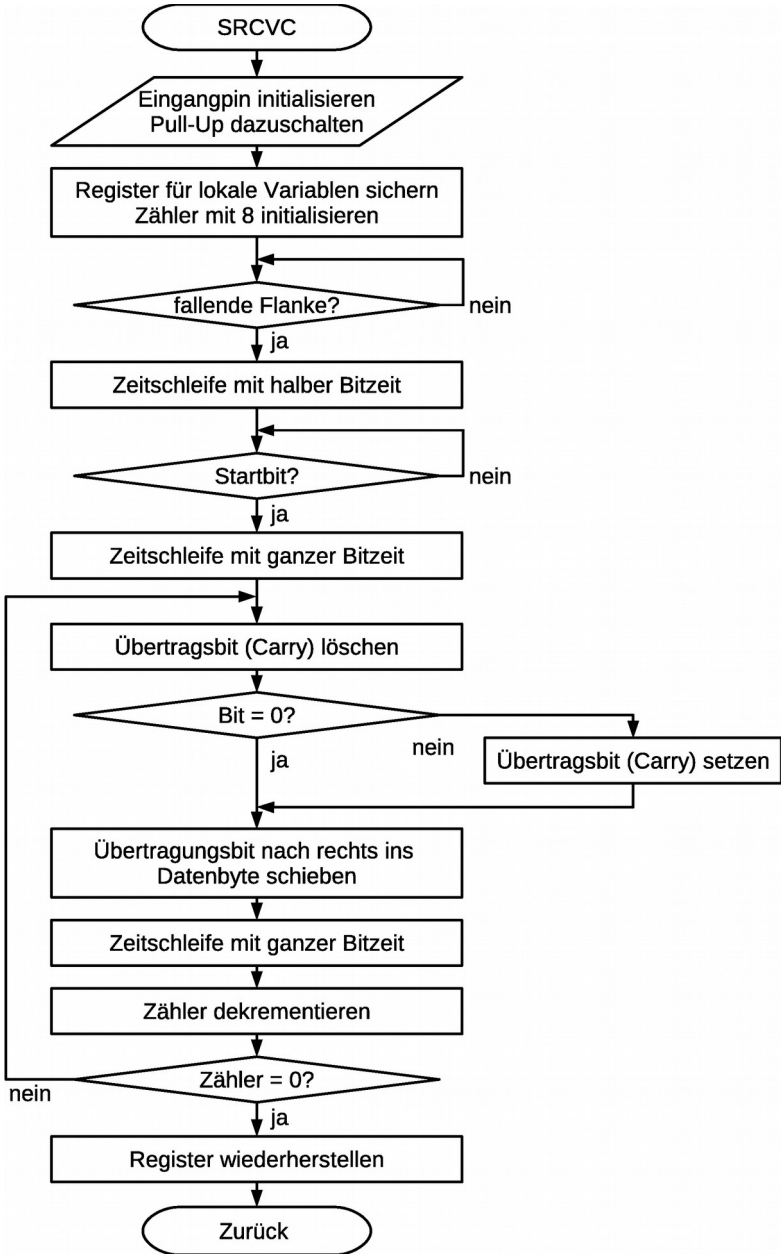
⌂ **D001**

Die Bibliothek soll um ein ein Unterprogramm mit dem symbolischen Namen **SSNDC11** für eine feste Baudrate von 115200 Baud (16 MHz Quarz) erweitert werden. Es ist eine einfache 8-Bit Zeitschleife zu verwenden. Ein Zeitfehler soll mit **nop**- Befehlen ausgeglichen werden.
a) Teste das Unterprogramm. Nenne die Bibliothek "**SR_UART_SOFT.asm**" und das Hauptprogramm "**D001_UART_SOFT_test2.asm**".
b) Sende das Zeichen **'U'** und miss die Baudrate mit dem Oszilloskop. Der Fehler darf maximal 0,5 % betragen!

Software-UART zum Empfangen.

Ein Unterprogramm für die Bibliothek soll ähnlich wie beim Senden den Empfang mittels Software ermöglichen. Das Abtasten soll zur Mitte der Bitzeit erfolgen. Die Schiebeoperation und die Berechnung der Zeitschleifen erfolgt ähnlich wie beim Senden.





⌂ **D002**

a) Die Bibliothek soll um ein Unterprogramm mit dem symbolischen Namen **SRCVC** erweitert werden. Programmiere und teste das Unterprogramm mit 115200 Baud.
b) Im Hauptprogramm wird dazu, das von der PC-Tastatur empfangene Byte als Echo zurück zum PC geschickt wird. Schreibe das Hauptprogramm und nenne es "**D002_UART_SOFT_test3.asm**".

Software-UART mit Interrupt

Besonders beim Empfang wird der Controller lange blockiert, da nicht gewusst ist, wann Daten eintreffen werden. Hier bietet es sich an einen externen Interrupt zu opfern um auf eintreffende Daten reagieren zu können.

⌂ **D003**

a) Die Bibliothek soll um ein eine Interruptroutine mit dem symbolischen Namen **ISRSRC** erweitert werden. Die negative Flanke des Startbit an **PB2** (**INT2**) soll den Empfang des Datenbyte auslösen. Programmiere und teste die ISR.

b) Im Hauptprogramm wird dazu, das von der PC-Tastatur empfangene Byte als Echo zurück zum PC geschickt wird. Schreibe das Hauptprogramm und nenne es "**D003_UART_SOFT_test4.asm**".

Bemerkungen:

Falls möglich sollte beim Software-UART und auch beim Debuggen immer mit der höchstmöglichen Baudrate gearbeitet werden. Setzt man die serielle Schnittstelle zum Debuggen ein, so muss man darauf achten das Timing des eigentlichen Programms nicht beeinflusst wird.
Um weiter Arbeitszeit des Controllers einzusparen, können die Zeitschleifen durch Timerinterrupts ersetzt werden.

Durch Mehrfachtastung könnten Fehler entdeckt werden und die Übertragung bei großen Leitungslängen (bzw. hoher Störstrahlung) verbessert werden.

Wie schon erwähnt besitzt der pinkompatible ATmega644p zwei Hardware-USARTs. Er bietet aber auch die Möglichkeit an jedem beliebigen Pin einen externen Interrupt auszulösen und somit interruptgesteuert Daten zu empfangen.

D1 Die I²C Schnittstelle (TWI)

Einführung

Die von Philipps entwickelte **synchrone I²C-Master-Slave-Schnittstelle**, (***Inter-Integrated Circuit***, gesprochen **I-Quadrat-C**) dient der Kommunikation zwischen verschiedenen integrierten Schaltkreisen (*IC, Integrated Circuit*). Sie wurde für die Unterhaltungselektronik entwickelt (Fernsehgeräte) und ist dort weit verbreitet (viele ansteuerbare Spezial-ICs). Bei einigen AVR-Controllern wird die Schnittstelle hardwaremäßig unterstützt und heißt aus Urheberrechtsgründen nicht I²C- sondern **TWI**-Schnittstelle (***Two Wire Bus***).

Vorteile des I²C-Busses ist der geringe Verdrahtungsaufwand und die geringen Kosten bei der Entwicklung eines Gerätes. Es werden nur drei Leitungen benötigt. Ein Mikrocontroller kann so ein ganzes Netzwerk an ICs mit nur drei Leitungen und einfacher Software kontrollieren. Dies senkt die Kosten des zu entwickelnden Gerätes. Während des Betriebes können Chips zum Bus hinzugefügt oder entfernt werden (*hot-plugging*).

Nachteile des I²C-Busses sind die geringe Geschwindigkeit und die geringe überbrückbare Distanz. Daten können nur abwechselnd über die Datenleitung gesendet werden (Halbduplex) und zusätzlich zu den Daten müssen die Adressen der Bausteine versendet werden.

Verwendung: Der I²C-Bus wird meist zur Übertragung von Steuer- und Konfigurationsdaten verwendet, da es dabei meist nicht auf Schnelligkeit ankommt. Er wird zum Beispiel verwendet bei Echtzeituhren, Lautstärkereglern, Sensoren, A/D- un D/A-Wandlern mit niedriger Abtastrate, EEPROM Speicherbausteinen oder bidirektionale Schaltern und Multiplexern. Große Bedeutung hatte das I²C-Protokoll in der Vergangenheit im Chipkartenbereich. Er ist nicht geeignet für größere Entfernungen, da die Störsicherheit gering ist und zu Fehlern in der Übertragung führt.

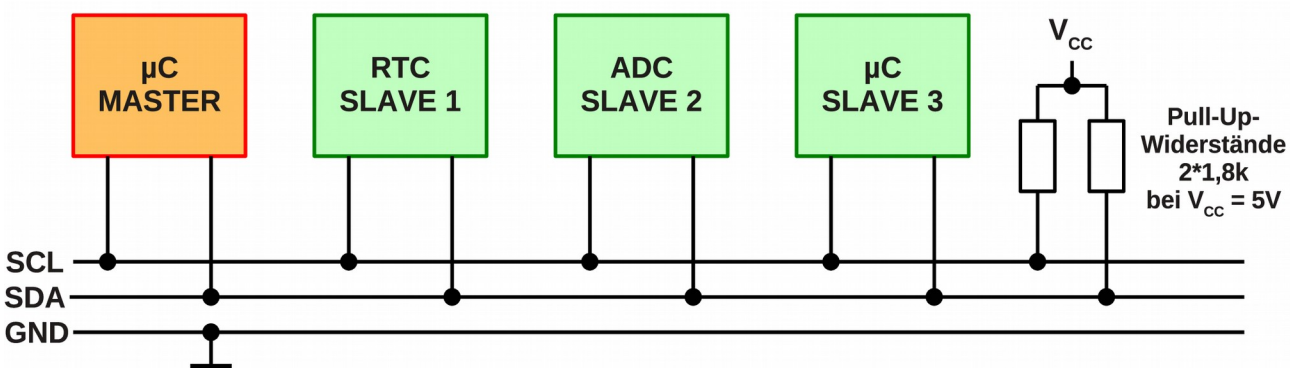
Der I²C-Bus ist als synchrone Master-Slave-Schnittstelle konzipiert. Es ist allerdings auch möglich mehrere Master einzusetzen (*multimaster mode*). Die Buszuteilung ist dabei in den Spezifikation geregelt und verhindert Kollisionen. Im Normalfall sendet der Master und ein Slave reagiert darauf.

Es können je nach verwendeten ICs vier Geschwindigkeiten eingesetzt werden:

- **100 kHz** (***standard mode***)
- **400 kHz** (***fast mode***; erweiterter Adressraum)
- **1 MHz** (***fast mode plus***)
- **3,4 MHz** (***high-speed mode***).

Beim ATmega32A (ATmega8A) sind maximal 400 kHz zulässig.

Der synchrone I²C-Bus



Im obigen Bild sind ein Master und drei Slaves eingezeichnet. Der synchrone I²C-Bus benötigt eine **Taktleitung** (*serial clock line*, **SCL**) und eine **Datenleitung** (*serial data line*, **SDA**). Die Pull-Up-Widerstände an der Takt- und Datenleitung ziehen beide Leitungen im Ruhezustand auf High-Pegel[4].

Alle am Bus angeschlossene Bausteine besitzen einen ***Open-Collector-*** oder ***Open-Drain-*Ausgang** (der Kollektor bzw. Drain eines Transistors ist unbeschaltet (offen) und wird durch den gemeinsamen Pull-Up Widerstand des Busses mit V_{CC} verbunden). Ist der bipolare Transistoren bzw. der FET durchgeschaltet, so wird der Bus auf Masse gezogen. Man nennt einen solche Verschaltung auch noch eine **Wired-And-Verknüpfung** da die Schaltung wie ein Und-Gatter wirkt.

Die I²C-Datenübertragung

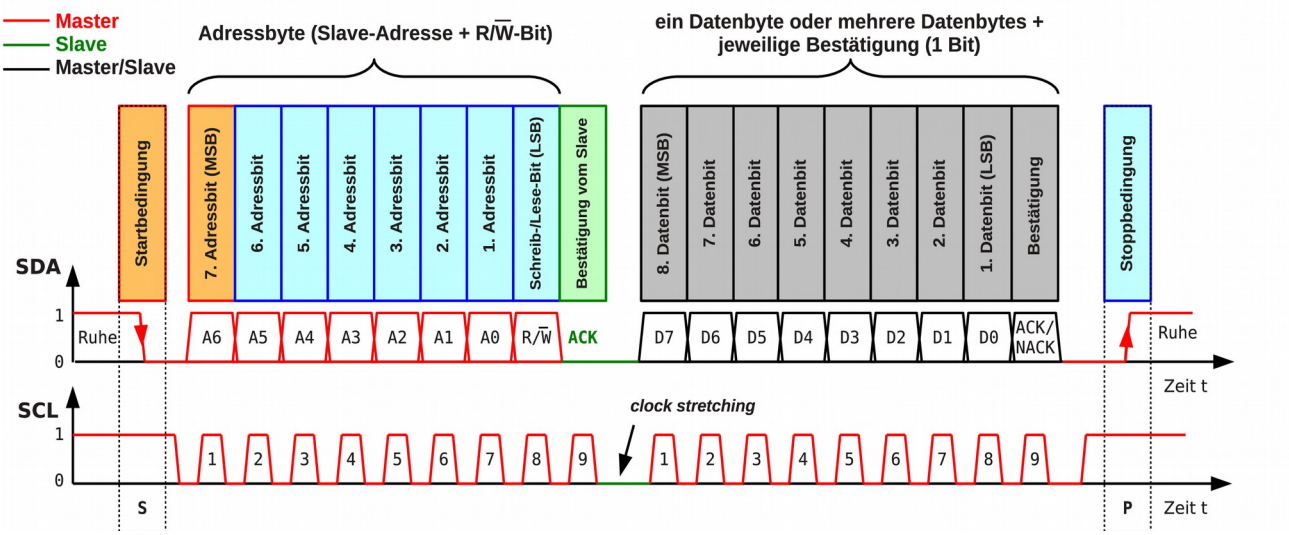
Der Master schreibt auf der Datenleitung **SDA** zum Slave oder liest von ihm. Der Takt auf der Taktleitung **SCL** wird vom Master erzeugt[5]. Damit ein Datenbit gültig ist, darf sich sein Pegel während einer Clock-High-Phase nicht ändern (gilt nicht für Start-, Stopp und *Repeated* Start-Signal). Nach einer Start-Bedingung ist der Bus belegt (*busy*) und kein anderer Master, als der Master der die Start-Bedingung erzeugt hat, darf den Bus stoppen. Nach einer Stopp-Bedingung ist der Bus wieder leer (*idle*).

Das I²C-Protokoll

Mit einer fallende Flanke auf SDA (SCL = *High*) startet der Master die Kommunikation. Nach dem Startbit sendet der Master als erstes das Adressbyte an den Slave. Das Adressbyte besteht aus einer 7-Bit Slave-Adresse und einem Schreib-Lese-Bit, welches die Richtung der Kommunikation festlegt. Der Slave bestätigt den korrekten Empfang mit einem **ACK**-Bestätigungsbit (*ACKnowledgement*). Der Master erzeugt die 9 Taktimpulse und liest dann die Taktleitung. Hier

4 Durch die Pull-Up-Widerstände soll ein Strom von maximal 3 mA fließen. Bei niedrigen Geschwindigkeiten kann ein höherer Widerstand (geringerer Energieverbrauch) verwendet werden. Bei V_{CC} = 5 V sind 4,7 kΩ im *standard mode* und 1,8 kΩ im *fast mode* gute Werte (bis zu einer Bus-Kapazität von 200 pF). Weitere Berechnungen findet man in den I²C-Specs bzw im Datenblatt des verwendeten Controllers.
(http://www.nxp.com/acrobat_download/usermanuals/UM10204_3.pdf).

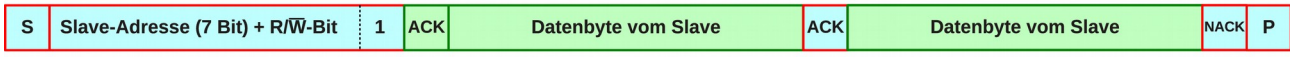
5 Im Bedarfsfall kann der Slave die Low-Phase des Takts verlängern indem er SCL auf Low-Pegel zieht (*clock stretching*) und so den Master bremsen.



kann dann ein langsamer Slave mit einem Low-Pegel eine Wartezeit erzwingen (*clock stretching*). Je nach Richtung der Kommunikation sendet jetzt der Master oder der Slave beliebig viele Datenbytes (8 Bit, MSB *first*). Jedes Datenbyte wird vom Gegenüber mit einem ACK-Bit (Low-Pegel) bestätigt. Die Übertragung wird durch das Senden eines **NACK**-Bits (*Not ACK*nowledge, High-Pegel) vom Master oder Slave abgebrochen. Mit einer steigenden Flanke auf SDA (SCL = *High*) gibt der Master den Bus wieder frei (Stoppbit).

Master sendet Daten (n Datenbyte + jeweilige Bestätigung) zum Slave

Master liest Daten (n Datenbyte + jeweilige Bestätigung) vom Slave



Um Zeit zu sparen kann der Master auch den Bus nicht freigeben (kein Stoppbit) und gleich mit einem weiteren Startbit (*Repeated Start*) eine neue Kommunikation starten. Die Kommunikationsrichtung kann hierbei natürlich beliebig geändert werden.

Die Adressierung

Das vom Master gesendete Adressbyte besteht, wie beschrieben, aus sieben Bit die die eigentliche Adresse des Slave darstellen und einem achten Bit das die Lese- oder Schreibrichtung festlegt. Die I²C-Schnittstelle nutzt einen Adressraum von 7 Bit, womit gleichzeitig 112 Bausteine auf einem Bus angesprochen werden können (16 der 128 möglichen Adressen sind für Sonderzwecke reserviert). Jeder I²C-fähige Baustein (IC) hat eine festgelegte Adresse. Bei manchen ICs kann ein Teil der Adresse hardwaremäßig mittels Steuerpins festgelegt werden. So können z.B. bis zu acht gleichartige ICs an einem I²C-Bus betrieben werden. Immer häufiger kann die Adresse aber auch softwaremäßig umprogrammiert werden (z.B. bei digitalen Sensoren).

Es besteht auch noch eine neuere alternative 10 Bit-Adressierung (1136 Bausteine). Sie ist abwärtskompatibel zum 7 Bit-Standard (nutzt zusätzlich 4 der 16 reservierten Adressen).

Die TWI-Schnittstelle des ATmega32A

Die hardwaremäßige TWI-Schnittstelle des ATmega-Controller entspricht der I^2C-Schnittstelle. Der Controller kann dabei sowohl als Master als auch als Slave programmiert werden. Ein Multi-Master-Mode ist möglich, soll aber hier nicht behandelt werden (siehe Datenblatt). Beim ATmega32A werden die Pins **PC0** (**SCL**) und **PC1** (**SDA**) verwendet. Diese Pins brauchen bei eingeschalteter Schnittstelle nicht mehr speziell konfiguriert zu werden.

Die Initialisierung der TWI-Schnittstelle

Zur **Initialisierung und Statusabfrage** der Schnittstelle dienen die Register:

- **TWCR** (**TW**I **C**ontrol **R**egister) und **TWSR** (**TW**I **S**tatus **R**egister).

Die **Geschwindigkeit** (Bitrate) der Schnittstelle wird beim Betrieb im Master-Modus über das Register:

- **TWBR** (**TW**I **B**it **R**ate Register) und 2 Bit (Vorteiler) im **TWSR**-Register festgelegt.

Daten werden über das Register:

- **TWDR** (**TW**I **D**ata **R**egister) ausgegeben bzw. eingelesen.

Wird der Controller als **Slave** genutzt, so wird seine Adresse im Register:

- **TWAR** (**TW**I Slave **A**ddress **R**egister) abgelegt.

Das interessanteste Register ist das Kontrollregister **TWCR**. Mit dem Bit **TWEN** wird der Baustein eingeschaltet.

Polling

Mit dem Bit **TWINT** wird eine Aktion gestartet, die vorher vorbereitet wurde:

- Beim vorigen Setzen des Bits **TWSTA** gibt der Controller als Master ein Startbit aus.
- Beim vorigen Setzen des Bits **TWSTO** gibt der Controller als Master ein Stoppbit aus.
- Beim vorigen Setzen des Bits **TWEA** gibt der Controller als Master eine Bestätigung (ACK) aus.
- Beim vorigen Ablegen von Daten im Datenregister **TWDR** gibt der Controller ein Datenbyte aus.

Mit **TWINT** = 1 wird die Aktion also eingeleitet. Die Steuerung setzt **TWINT** = 0 und führt die Operation dann aus. Als Flag zeigt **TWINT** dann das Ende der Operation an. Im Statusregister wird mit 5 Statusbit ermittelt werden, ob die Operation erfolgreich war.

Im einfachen Pollingbetrieb zeigt das Flag **TWINT** also an wenn die Operation abgeschlossen ist.

Interrupt

Mit dem Bit **TWIE** kann auch ein Interrupt freigegeben werden. Die ISR, welche dann nach Beendigung der Operation aufgerufen wird, muss sich selbst um das Löschen des Flag **TWINT** durch Schreiben einer Eins kümmern. Dadurch wird allerdings auch die nächste Operation gestartet und die entsprechenden Register müssen schon vorbereitet sein.

Die Bitrate der Schnittstelle (TWBR und TWSR)

Wird der Controller als Slave betrieben, so sind hier keine Einstellungen erforderlich, allerdings muss die Taktfrequenz des Controller 16 mal höher als die Bitrate des Busses liegen!

Im **Masterbetrieb** errechnet sich die Bitrate mit:

$$f_{SCL} = \frac{\text{Systemtakt}}{16+2\cdot \text{TWBR}\cdot \text{Vorteiler}}$$

TWBR ist dabei der Wert des Bitraten-Registers **TWBR**. Der **Vorteiler** errechnet sich mit 4^{TWPS} aus dem Inhalt der beiden Bit **TWPS0** und **TWPS1** im Statusregister **TWSR**. Damit ergeben sich folgende Werte für den Vorteiler:

TWPS1	TWPS0	TWPS	Vorteiler (4^{TWPS})
0	0	**0**	**1**
0	1	**1**	**4**
1	0	**2**	**16**
1	1	**3**	**64**

Der Wert des Bitraten-Register errechnet sich dann mit einem erwählten Vorteiler aus:

$$\text{TWBR} = \frac{\left(\frac{\text{Systemtakt}}{f_{SCL}} - 16\right)}{2\cdot \text{Vorteiler}}$$

⌂ **D100** Berechne den Wert von TWBR und den Wert des Vorteilers für den „*standard mode*“ bei einem Quarz von 16 MHz. Notiere die Zeilen, die zur Initialisierung nötig wären.

Die SF-Register der TWI-Schnittstelle

Das Kontroll-Register TWCR

TWCR-Register: SF-Register-Adresse **0x36** (SRAM-Adresse **0x0056**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)

TWCR = TWI Control Register

Bit	7	6	5	4	3	2	1	0
TWCR 0x36	**TWINT**	**TWEA**	**TWSTA**	**TWSTO**	**TWWC**	**TWEN**	**-**	**TWIE**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W

TWINT ***TWI Interrupt Flag***

0 es wurde eine Operation gestartet, die noch nicht beendet ist.

1 wird von der Hardware auf Eins gesetzt, sobald eine TWI Operation beendet ist. Gleichzeitig wird ein TWI Interrupt ausgelöst, wenn Interrupts global erlaubt sind (**I** im **SREG**) und das **TWIE**-Bit gesetzt wurde.
Durch das Schreiben einer Eins muss das Bit softwaremäßig gelöscht werden!
Dies startet auch eine neue Operation. Vor dem Löschen müssen deshalb alle Register, die für die Operation benötigt werden schon beschrieben sein!
Auch beim Verwenden des Interrupts muss das Bit manuell gelöscht werden! Dies geschieht hier nicht automatisch durch die Hardware!

TWEA ***TWI Enable Acknowledge Bit***

0 **sendet ein NACK**, nachdem Daten (Slave Adresse, globaler Anruf) angekommen sind.

1 **sendet ein ACK**, nachdem Daten (Slave Adresse, globaler Anruf) angekommen sind.

TWSTA ***TWI START Condition Bit***

0 muss per Software wieder zurückgesetzt werden, wenn das Startbit erfolgreich versendet wurde!

1 **sendet eine Startbedingung** als Master, falls der Bus frei ist (wartet sonst auf eine Stoppbedingung).

TWSTO ***TWI STOP Condition Bit***

0 wird nach dem Senden einer Stoppbedingung automatisch rückgesetzt.

1 **sendet eine Stoppbedingung** als Master. Kann im Slave-Modus benutzt werden, um im Fehlerfall einen neuen definierten Zustand zu erreichen (SCL und SDA hochohmig).

TWCC ***TWI Write Collision Flag***

0 wird automatisch gelöscht, wenn Daten in das Datenregister **TWDR** geschrieben werden und keine Operation anhängig ist (**TWINT** = 1).

1 **Kollision!** Zeigt an, dass versucht wurde in das Datenregister **TWDR** zu schreiben, obschon die vorherige Operation noch nicht beendet wurde (**TWINT** = 0).

TWEN ***TWI Enable***

0 schaltet die TWI-**Schnittstelle aus** (Baugruppe verbraucht keinen Strom!).

1 schaltet die TWI-**Schnittstelle ein**.

ADIE ***TWI Interrupt Enable***

0 der TWI Interrupt ist gesperrt.

1 Das Setzen dieses Bit **ermöglicht das Auslösen eines Interrupts** in dem Moment, wo die TWI-Operation beendet ist (**TWINT**-Flag = 1 im **TWCR**).

Das Status-Register TWSR

TWSR-Register: SF-Register-Adresse **0x01** (SRAM-Adresse **0x0021**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**

TWSR = TWI Status Register

Bit	7	6	5	4	3	2	1	0
TWSR 0x01	**TWS7**	**TWS6**	**TWS5**	**TWS4**	**TWS3**	**-**	**TWPS1**	**TWPS0**
Startwert	1	1	1	1	1	0	0	0
Read/Write	R	R	R	R	R	R	R/W	R/W

TWS ***TWI Status TWS7-TWS3***

Die 5 Statusbits müssen aus dem Register ausmaskiert werden! Sie geben an ob eine Operation erfolgreich durchgeführt wurde und sollten kontrolliert werden.

Für den **Mastermode** gelten folgende Codes:

TWS	TWSR (maskiert)	Bedeutung
00001	**0x08**	Startbedingung wurde gesendet.
00010	**0x10**	Wiederholte Startbedingung gesendet (*repeated start*).
00011	**0x18**	Slave-Adresse fürs Schreiben gesendet. **ACK** empfangen.
00100	**0x20**	Slave-Adresse fürs Schreiben gesendet. **NACK** empfangen.
00101	**0x28**	Datenbyte gesendet. **ACK** empfangen.
00110	**0x30**	Datenbyte gesendet. **NACK** empfangen.
00111	**0x38**	Datenverlust!
01000	**0x40**	Slave-Adresse fürs Lesen gesendet. **ACK** empfangen.
01001	**0x48**	Slave-Adresse fürs Lesen gesendet. **NACK** empfangen.
01010	**0x50**	Datenbyte empfangen. **ACK** wurde gesendet.
01011	**0x58**	Datenbyte empfangen. **NACK** wurde gesendet.

TWPS ***TWI Prescaler Bits TWPS1,TWPS0***

Diese beiden Bits legen den Vorteiler für die Bitrate fest:

TWPS1	TWPS0	TWPS	Vorteiler (4^{TWPS})
0	0	**0**	**1**
0	1	**1**	**4**
1	0	**2**	**16**
1	1	**3**	**64**

Das Bitraten-Register TWBR

TWBR-Register: SF-Register-Adresse **0x00** (SRAM-Adresse **0x0020**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**

TWBR = TWI Bit Rate Register

Bit	7	6	5	4	3	2	1	0
TWBR 0x00	**TWBR7**	**TWBR6**	**TWBR5**	**TWBR4**	**TWBR3**	**TWBR2**	**TWBR1**	**TWBR0**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

TWBRn ***TWI Bit Rate Register Bit n***
Im Masterbetrieb errechnet sich die Bitrate mit:

$$f_{SCL} = \frac{\text{Systemtakt}}{16+2\cdot \text{TWBR}\cdot \text{Vorteiler}}$$

TWBR ist der Wert des Bitraten-Registers **TWBR**. Der Vorteiler wurde im Statusregister **TWSR** festgelegt. **TWBR** ermittelt man mit:

$$\text{TWBR} = \frac{\left(\frac{\text{Systemtakt}}{f_{SCL}} - 16\right)}{2\cdot \text{Vorteiler}}$$

Das Datenregister TWDR

TWDR-Register: SF-Register-Adresse **0x03** (SRAM-Adresse **0x0023**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**

TWDR = TWI Data Register

Bit	7	6	5	4	3	2	1	0
TWDR 0x03	**TWDR7**	**TWDR6**	**TWDR5**	**TWDR4**	**TWDR3**	**TWDR2**	**TWDR1**	**TWDR0**
Startwert	1	1	1	1	1	1	1	1
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

TWDRn ***TWI Data Register Bit n***
Das Datenbyte enthält das letzte zu sendende Byte oder das zuletzt empfangene Byte. Es kann nur beschrieben werden, wenn **TWINT** = 1 (**TWCR**).

Das Adress-Register TWAR

Dieses Register wird nur im Slave-Modus benötigt!

TWAR-Register: SF-Register-Adresse **0x02** (SRAM-Adresse **0x0022**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**

TWAR = TWI (Slave) Adress Register

Bit	7	6	5	4	3	2	1	0
TWAR 0x02	**TWA6**	**TWA5**	**TWA4**	**TWA3**	**TWA2**	**TWA1**	**TWA0**	**TWGCE**
Startwert	1	1	1	1	1	1	1	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

TWAn ***TWI (Slave) Adress Register Bit n***
Im Slavemodus wird in diesem Register die Slave-Adresse eingetragen

TWGCE ***TWI General Call Recognition Enable Bit***
Erlaubt das Erkennen eines globalen Rundrufs auf dem Bus.

Die I²C-Bibliothek

Im Folgenden soll eine kleine Bibliothek zum Senden und Empfangen von Daten als Master mittels Polling (ständiges Abfragen) über den I²C Bus erstellt werden. Die Bibliothek besteht aus 6 kleinen Unterprogrammen. Zur Parameterübergabe wird das Doppelregister **r25**:**r24** verwendet. Daten werden mit **r24** übermittelt (**r24** dient manchmal auch als Zwischenspeicher). Die, aus fünf Bit bestehende Statusinformation (siehe Statusregister **TWSR**) befindet sich in **r25**.
Die Bibliothek findet man auf: http://weigu.lu/tutorials/avr_assembler (**"SR_I2C.asm"**)

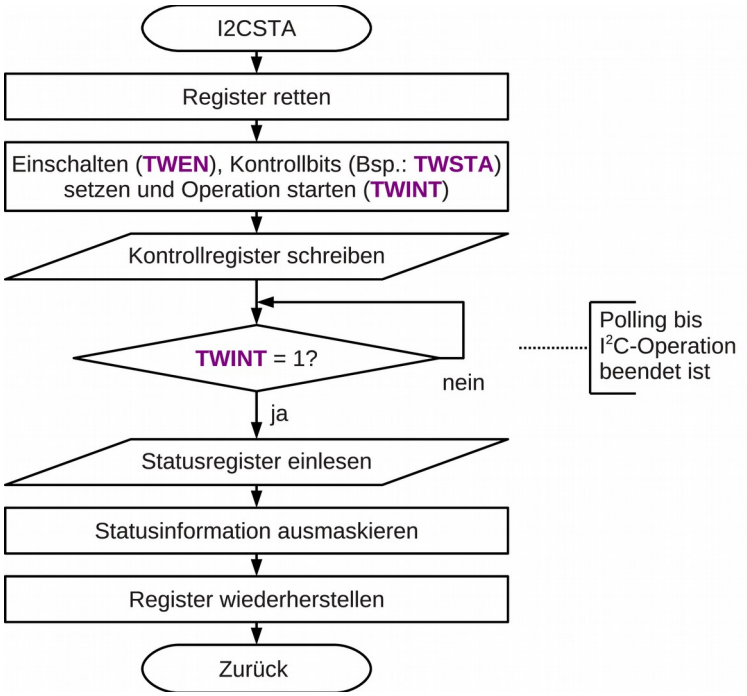
Bus initialisieren mit I2CINI

Die Initialisierung beschränkt sich auf das Festlegen der Bitrate. Dieses Unterprogramm muss als erstes im Hauptprogramm aufgerufen werden. Die Zuweisungen können auch im Hauptprogramm erfolgen.

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Zuweisungen
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EQU    I2C_Clock = 100000      ;I2C Geschwindigkeit (nur 100000 oder 400000)
.EQU    CPU_Clock = 16000000    ;Systemtakt
                                ;Achtung! Je nach Systemtakt muss der Vorteiler
                                ;und die Formel im UP I2CINI angepasst werden
;-------------------------------------------------------------------------------
;       I2C (TWI) initialisieren
;-------------------------------------------------------------------------------
I2CINI: push    r16
        clr     r16             ;TWPS = 0 (Vorteiler = 1)
        out     TWSR,r16        ;TWBR=((Systemtakt/I2C Takt)-16)/(2*Vorteiler)
        ldi     r16,((CPU_Clock/I2C_Clock)-16)/2
        out     TWBR,r16        ;das Einschalten der Baugruppe (TWEN in TWCR)
        pop     r16             ;erfolgt jeweils beim Ansteuern
        ret
```

Startbit senden mit I2CSTA

Als Erstes muss ein Startbit gesendet werden (wird das Unterprogramm ohne voriges Stoppen des Busses verwendet, so erfolgt ein *Repeated Start*).
Dieses Unterprogramm zeigt die Struktur, welche auch in den folgenden Unterprogrammen verwendet wird.




```
;-------------------------------------------------------------------------------
;       Startbit senden
;-------------------------------------------------------------------------------
I2CSTA: push    r24
        ldi     r24,0xA4        ;TWCR=0b10100100 TWINT,TWSTA u. TWEN setzen
        out     TWCR,r24
I2CST1: in      r24,TWCR        ;warten bis TWINT wieder 1
        sbrs    r24,TWINT       ;Operation beendet
        rjmp    I2CST1
        in      r25,TWSR        ;Status-Code in r25
        andi    r25,0xF8
        pop     r24
        ret
```

Daten senden mit I2CSND

In diesem Unterprogramm zum Senden (*send*) von Daten werden die Daten zuerst in das Datenregister kopiert. Die Bestätigung (ACK) kommt vom Slave.

```
;-------------------------------------------------------------------------------
;       Daten senden
;-------------------------------------------------------------------------------
I2CSND: push    r24
        out     TWDR,r24        ;r24 (Daten) ins Datenregister
        ldi     r24,0x84        ;TWCR=0b10000100 TWINT u. TWEN setzen TWSTA
        out     TWCR,r24        ;ruecksetzen
I2CSN1: in      r24,TWCR        ;warten bis TWINT wieder 1
        sbrs    r24,TWINT       ;Operation beendet
        rjmp    I2CSN1
        in      r25,TWSR        ;Status-Code in r25
        andi    r25,0xF8
        pop     r24
        ret
```

Daten empfangen und ACK senden mit I2CRAK

Es werden zwei Unterprogramme zum Empfang (*receive*) von Daten benötigt. Das erste Unterprogramm (I2CRAK) sendet nach dem Empfang eine positive Empfangsbestätigung (ACK). Das empfangene Datenbyte befinden sich in **r24**.

```
;-------------------------------------------------------------------------
;       Daten empfangen mit ACK
;-------------------------------------------------------------------------
I2CRAK: ldi     r24,0xC4        ;TWCR=0b11000100 TWINT, TWEA u. TWEN setzen
        out     TWCR,r24
I2CRA1: in      r24,TWCR        ;warten bis TWINT wieder 1
        sbrs    r24,TWINT       ;(Start-Operation beendet)
        rjmp    I2CRA1
        in      r24,TWDR        ;Daten in r24
        in      r25,TWSR        ;Status-Code in r25
        andi    r25,0xF8
        ret
```

Daten empfangen und NACK senden mit I2CRNA

Das zweite Unterprogramm sendet nach dem Empfang eine negative Empfangsbestätigung (NACK) und beendet damit die Übertragung.

```
;-------------------------------------------------------------------------
;       Daten empfangen ohne ACK (NACK)
;-------------------------------------------------------------------------
I2CRNA: ldi     r24,0x84        ;TWCR=0b10000100 TWINT u. TWEN setzen
        out     TWCR,r24
I2CRN1: in      r24,TWCR        ;warten bis TWINT wieder 1
        sbrs    r24,TWINT       ;(Start-Operation beendet)
        rjmp    I2CRN1
        in      r24,TWDR        ;Daten in r24
        in      r25,TWSR        ;Status-Code in r25
        andi    r25,0xF8
        ret
```

Stoppbit senden mit I2CSTO

Mit dem Stopbit gibt der Master den Bus wieder frei. Ohne Stoppbit kann der Master mit einem neuen Startbit (*repeated start*) den Bus weiter besetzen.

```
;-------------------------------------------------------------------------
;       Stoppbit senden
;-------------------------------------------------------------------------
I2CSTO: push    r16
        ldi     r16,0x94         ;TWCR=0b10010100 TWINT, TWSTO u. TWEN setzen
        out     TWCR,r16
        pop     r16
        ret
```

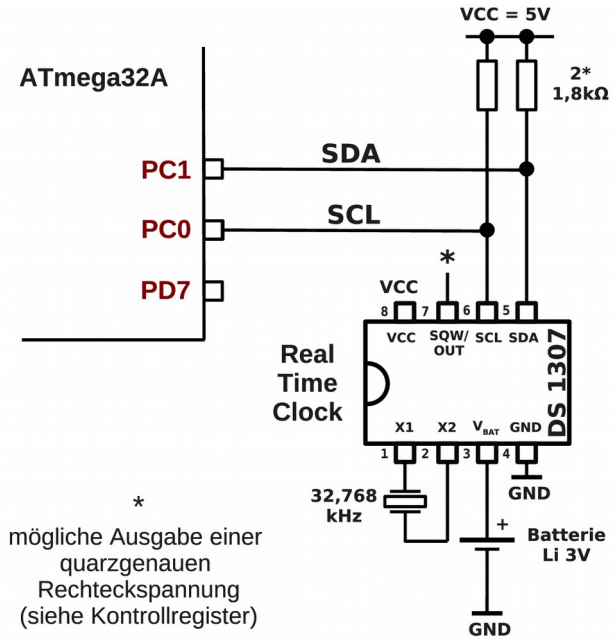
Die Echtzeituhr (RTC) DS1307

Das Senden und Empfangen von Daten über den I^2C-Bus soll mit einer Echtzeituhr (*Real Time Clock*) getestet werden. Eine Echtzeituhr läuft auch ohne externe Spannungsversorgung mit einer Batterie (üblicherweise Lithium-Knopfzelle mit 3 V) weiter.

Es wird der I²C Baustein DS1307 von Maxim als Echtzeituhr verwendet. Zur äußeren Beschaltung wird nur ein Uhrenquarz (32,768 kHz) und die Batterie benötigt. Der DS1307 besitzt auch einen Ausgang (SQW/OUT) mit dem ein quarzgenauer Takt ausgegeben werden kann, um zum Beispiel einen Interrupt im Sekundentakt auszulösen. Die I²C-Adresse des Bausteins ist **`0b1101000 (0x68)`**.

Bemerkungen: Wird keine Batterie angeschlossen, so muss Pin 3 mit Masse verbunden werden, damit der Baustein angesprochen werden kann.

Der DS1307 arbeitet nur im Standard Mode (100 kHz).



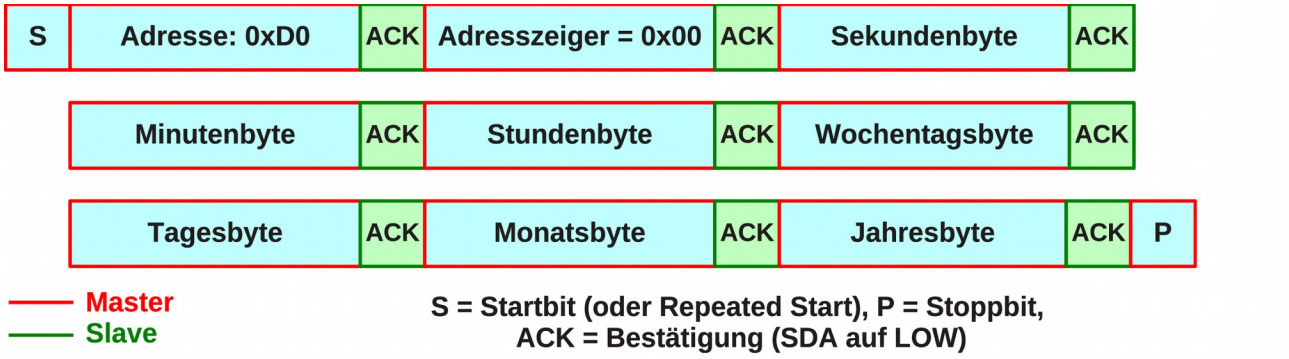
Der Uhrenbaustein besitzt 64 Register (RAM-Speicherzellen), welche über eine Registeradresse angesprochen werden können. Die ersten sieben Register enthalten die Daten der Uhr (Uhrzeit (3), Wochentag (1) und Datum (3)). Das achte Register dient als Kontrollregister. Die restlichen 56 Speicherzellen können beliebig beschrieben und gelesen werden (gepuffertes RAM).

Uns interessieren hier besonders die ersten sieben Register. Sobald das Sekundenregister beschrieben wurde (Bit 7 (CH) = 0) läuft die Uhr. Die Daten werden im BCD-Code[6] abgelegt.

6 Der BCD-Code (*Binary Coded Decimal*) ist ein Code mit dual kodierten Dezimalziffern. 4 Bit (*Nibble*) stellen eine Dezimalziffer (0-9) im Dualcode dar (**`0b0000-0b1001`**).

ADDRESS	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	FUNCTION	RANGE
00H	CH	10 Seconds			Seconds				Seconds	00–59
01H	0	10 Minutes			Minutes				Minutes	00–59
02H	0	12	10 Hour	10 Hour	Hours				Hours	1–12 +AM/PM 00–23
		24	PM/AM							
03H	0	0	0	0	0	DAY			Day	01–07
04H	0	0	10 Date		Date				Date	01–31
05H	0	0	0	10 Month	Month				Month	01–12
06H	10 Year				Year				Year	00–99
07H	OUT	0	0	SQWE	0	0	RS1	RS0	Control	—
08H-3FH									RAM 56 x 8	00H–FFH

I²C-Daten zur Echtzeituhr senden



Zum Senden von Daten an den Baustein wird das erste Bit (Bit 0, LSB) des Baustein-Adressbytes auf Null fürs Schreiben gesetzt. Mit der 7-Bit-Adresse des DS1307 erhält man dann **`0b11010000`** (**`0xD0`**) als Gesamt-Adressbyte. Nach dem Senden der Baustein-Adresse muss beim DS1307 als erstes Datenbyte die Adresse der zu adressierenden Speicherzelle gesendet werden. Praktischerweise wird die Adresse (Adresszeiger) automatisch nach dem Schreiben eines Registerinhalts erhöht, so dass nacheinander alle 7 Registerinhalte gesendet werden können.

Es ist sinnvoll die Daten vorher in einer SRAM-Tabelle (hier symbolische Adresse (Label): **`RTCTAB`**) abzulegen. Das folgende Unterprogramm zeigt die Vorgehensweise beim Beschreiben des Uhrenbausteins. Tritt ein Fehler bei der Übertragung auf, so wird dies mit einer LED signalisiert.

Das Unterprogramm befindet sich in der Bibliothek "**`SR_RTC_DS1307_sample.asm`**" (http://weigu.lu/tutorials/avr_assembler).

```
;-------------------------------------------------------------------------
;       Daten senden (7 Datenbyte, Tabelle RTCTAB im SRAM)
;-------------------------------------------------------------------------
RTCSND: rcall   I2CSTA          ;Startbit senden + Kontrolle
        cpi     r25,0x08
        brne    RTCSNE
        ldi     r24,0xD0        ;Adresse des RTC DS1307 (Maxim 1101000) senden
        rcall   I2CSND          ;(Bit 2^0 = 0 schreiben) + Kontrolle
        cpi     r25,0x18
        brne    RTCSNE
        ldi     r24,0x00        ;Adresszeiger = 0 senden + Kontrolle
        rcall   I2CSND
        cpi     r25,0x28
        brne    RTCSNE
        ;7 Datenbyte senden + Kontrolle
```

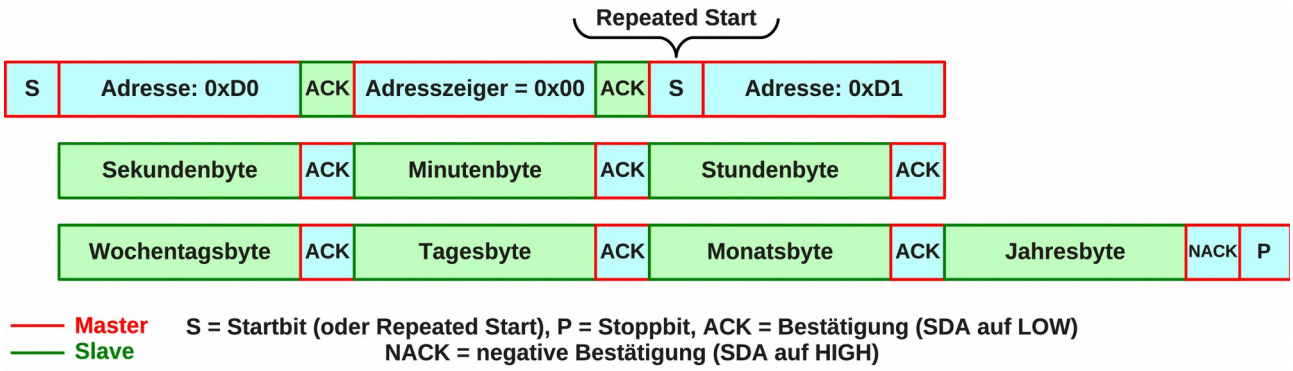
```
        lds     r24,RTCTAB+0    ;Sekunden
        rcall   I2CSND
        cpi     r25,0x28
        brne    RTCSNE
        lds     r24,RTCTAB+1    ;Minuten
        rcall   I2CSND
        cpi     r25,0x28
        brne    RTCSNE
        lds     r24,RTCTAB+2    ;Stunden
        rcall   I2CSND
        cpi     r25,0x28
        brne    RTCSNE
        lds     r24,RTCTAB+3    ;Wochentag
        rcall   I2CSND
        cpi     r25,0x28
        brne    RTCSNE
        lds     r24,RTCTAB+4    ;Tag
        rcall   I2CSND
        cpi     r25,0x28
        brne    RTCSNE
        lds     r24,RTCTAB+5    ;Monat
        rcall   I2CSND
        cpi     r25,0x28
        brne    RTCSNE
        lds     r24,RTCTAB+6    ;Jahr
        rcall   I2CSND
        cpi     r25,0x28
        brne    RTCSNE
        rjmp    RTCSNR
RTCSNE: sbi     ERRDDR,ERRPNr   ;Ausgang fuer Fehler LED initialisieren
        sbi     ERRPORT,ERRPNr  ;Fehler LED ein
RTCSNR: rcall   I2CSTO          ;Stoppbit senden
        ret
```

I²C-Daten von der Echtzeituhr empfangen

⌂ **D101**

a) Die Bibliothek soll um ein Unterprogramm mit dem symbolischen Namen **`RTCREC`** erweitert werden. Das Unterprogramm soll die abgespeicherten 7 Datenbyte aus dem RTC auszulesen. Die Vorgehensweise kann aus dem unteren Diagramm ausgelesen werden.

b) Schreibe ein Hauptprogramm, das bei der Initialisierung die Uhr setzt, und dann in einer Schleife den Minuten- und den Sekundenwert ausliest und auf dem 7-Segmentdiplay darstellt.
Nenne das Hauptprogramm "**`D101_I2C_RTC_test.asm`**".



Bemerkungen: Wird sofort nach einem Stoppbit ein erneutes Startbit gesendet, so wird dies auch als *Repeated Start* erkannt. Bei der Fehlerbehandlung (Kontrolle) ist dies zu beachten. Es kann natürlich auch zwischen Stoppbit und Startbit eine kurze Zeitschleife eingefügt werden.

Auch beim Lesen wird der Adresszeiger im RTC automatisch mit jeder Leseoperation erhöht.

Um den Lesezyklus des DS1307 zu stoppen, benötigt dieser eine negative Bestätigung (NACK) vom Master.

Das EEPROM 24LC256

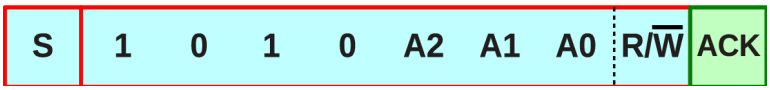
Als zweites Beispiel soll hier ein EEPROM mit 256 Kibit Speicher (32 KiB) verwendet werden. Ziel ist es einige Sekunden Sprache mit dem Electret-Mikrofon (A/D-Wandler) aufzunehmen und dann wieder per D/A-Wandler auszugeben.

Ein EEPROM ist natürlich auch nützlich um zum Beispiel Zeichenketten (mehrsprachige Menüführung am LCD-Display) oder Graphiken (Webserver) abzuspeichern.

Der Baustein 24LC256 beherrscht den Fast Mode (400 kHz). Wir werden diesen Modus nutzen um eine möglichst hohe Frequenz zu erreichen.

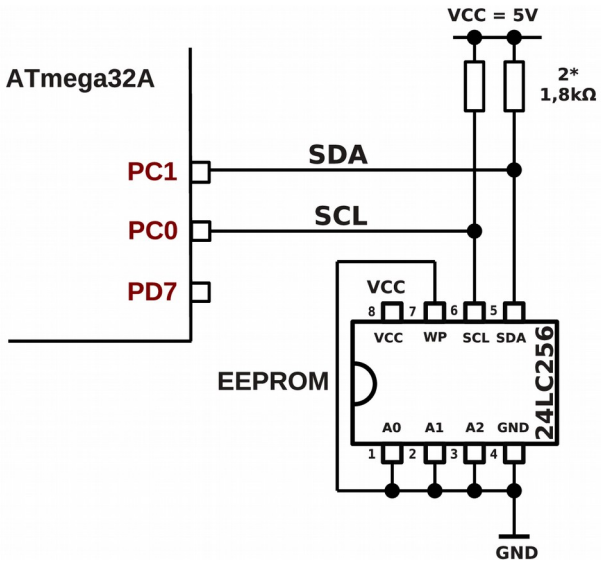
Das EEPROM 24LC256 besitzt 3 Pins, um die Adresse per Hardware festzulegen. Damit besteht die Möglichkeit 8 Bausteine am Bus zu betreiben (2 Mibit (256 KiB) Speicher). Die Grundadresse beträgt **`0b1010`** gefolgt von den drei per Hardware kodierten Bits A2, A1 und A0.

Adressbyte des EEPROM 24LC256



In unserem Beispiel legen wir die drei Pins auf Masse und erhalten **`0b1010000`** als Adresse. Damit ergibt sich **`0xA0`** als gesamtes Adressbyte beim **Schreiben** und **`0xA1`** beim **Lesen**.

Das Pin WP (*Write Protect*) muss mit Masse verbunden werden, wenn Lesen und Schreiben erlaubt werden sollen. Wird es mit VCC verbunden, so ist nur das Lesen erlaubt.



I²C-Daten zum EEPROM senden

Da in diesem EEPROM mehr als 256 Byte angesprochen werden muss eine Word (16 Bit) zur Adressierung der Speicherzellen verwendet werden. Es bestehen zwei Möglichkeiten:

- Schreiben von einzelnen Bytes (***Byte Write***). Dabei muss jedes Mal die 16-Bit-Adresse mit übermittelt werden.
- Sequentielles Schreiben einer ganze Seite (***Page Write***), wobei eine Seite aus 64 Byte besteht (insgesamt 512 Seiten mit je 64 Byte).

Quelle: Datenblatt 24LC256 (Microchip)

FIGURE 6-1: BYTE WRITE

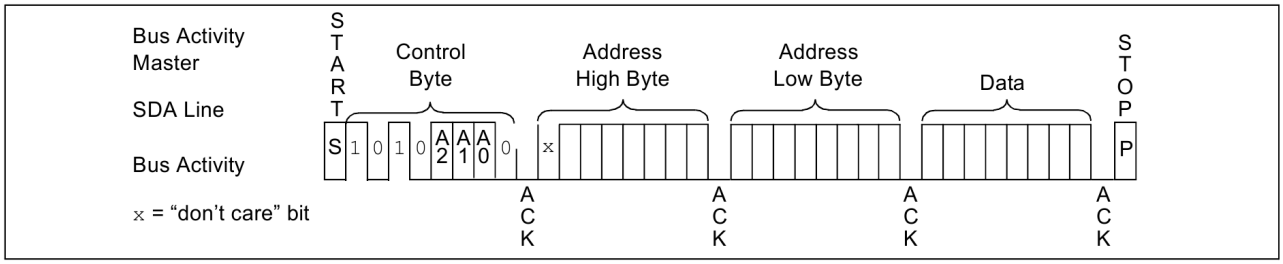
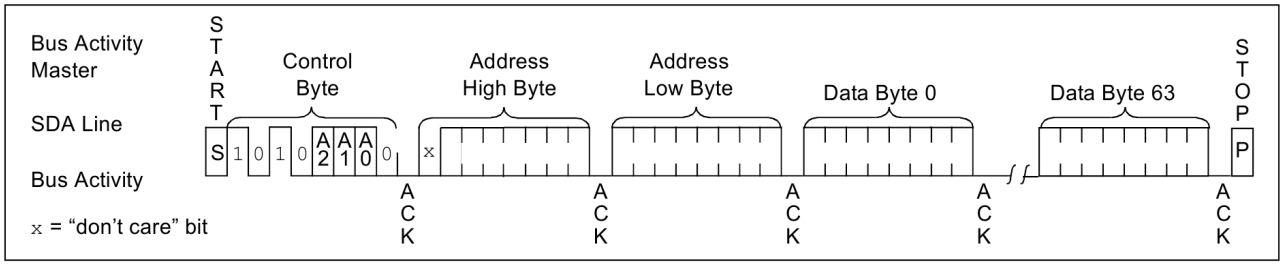


FIGURE 6-2: PAGE WRITE



Die zweite Möglichkeit soll hier genutzt werden, da sie effektiver ist.

Auch wenn nur 1 Byte geschrieben zu schreiben ist, wird im EEPROM eine ganze Seite (64 Byte) beschrieben. Dieser Schreibzyklus braucht nach Datenblatt maximal 5 ms. Während dieser Zeit sendet der Slave keine ACK Impulse!

Es muss also nach jedem Schreiben eine Wartezeit von 5 ms eingelegt werden!

⌂ **D102** Es soll eine Bibliothek mit dem Namen "**SR_EEPROM_24LC256.asm**" erstellt werden:

a) Schreibe ein Unterprogramm mit dem symbolischen Namen **E24SB** um ein einzelnes Byte in den EEPROM-Speicher zu schreiben. Die Adresse und das Datenbyte befinden sich im SRAM (Label: **E24ADL**, **E24ADH** und **E24BYTE**). Das Unterprogramm soll auch die Wartezeit von 5 ms berücksichtigen.

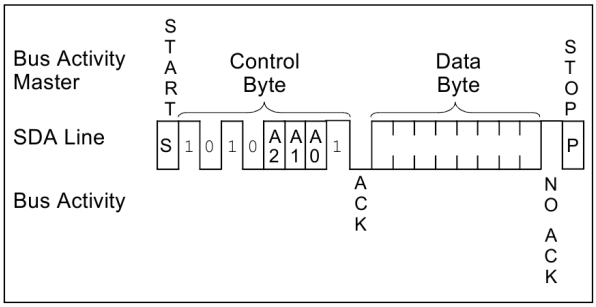
b) Schreibe ein zweites Unterprogramm mit dem symbolischen Namen **E24SP** um eine ganze Seite (Page, 64 Byte) in den EEPROM-Speicher zu schreiben. Die Adresse und eine Datentabelle mit 64 Byte befinden sich im SRAM (Label: **E24ADL**, **E24ADH** und **E24PAGE**). Das Unterprogramm soll auch die Wartezeit von 5 ms berücksichtigen.

I²C-Daten vom EEPROM empfangen

Beim Lesen der Daten gibt es drei Möglichkeiten:

- Lesen des Datenbyte an der momentanen Adresse (***Current Address Read***). Der interne Adresszeiger wird beim Lesen kontinuierlich um 1 erhöht. War die zuletzt gelesene Adresse n, so wird der Inhalt der Adresse n+1 gelesen.
- Lesen eines Bytes an einer beliebigen Adresse (***Random Read***). Die 16 Bit-Adresse muss zuerst in das EEPROM geschrieben werden.
- Sequentielles Lesen von mehreren Bytes (***Sequential Read***). Genau wie beim Lesen eines Bytes muss zuerst der Adresszeiger initialisiert werden. Wird aber nach dem ersten Byte kein Stoppbit, sondern ein ACK gesendet, so wird der Adresszeiger automatisch inkrementiert und es können beliebig viele Bytes gelesen werden.

FIGURE 8-1: CURRENT ADDRESS READ



Quelle: Datenblatt 24LC256 (Microchip)

FIGURE 8-2: RANDOM READ

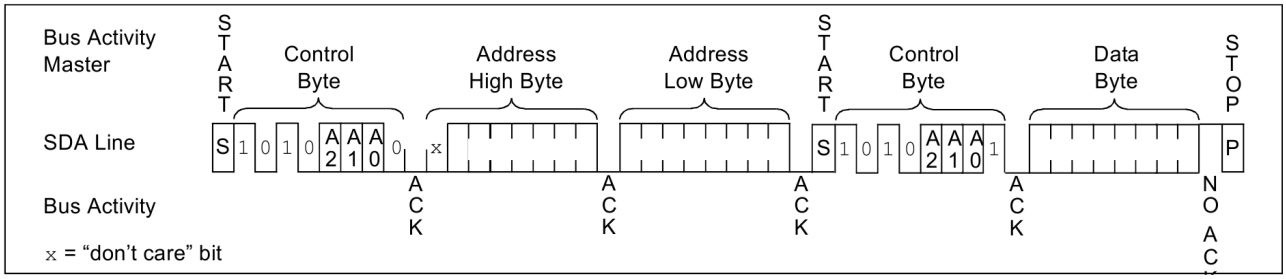
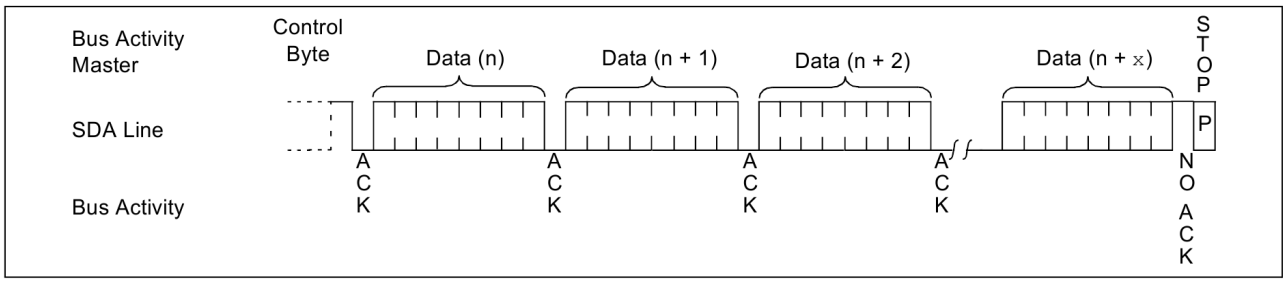


FIGURE 8-3: SEQUENTIAL READ



⌂ **D103** Die Bibliothek "**SR_EEPROM_24LC256.asm**" soll um zwei Unterprogramme erweitert werden:

a) Schreibe ein Unterprogramm mit dem symbolischen Namen **E24RR** um ein einzelnes Byte aus dem EEPROM-Speicher zu lesen (*Random Read*). Die Adresse befindet sich zur Parameterübergabe im SRAM (Label: **E24ADL**, **E24ADH**). Das empfangene Datenbyte soll im SRAM abgespeichert werden (Label: **E24BYTE**).

b) Schreibe ein zweites Unterprogramm mit dem symbolischen Namen **E24RS** um eine ganze Seite (Page, 64 Byte) aus dem EEPROM-Speicher zu lesen (*Sequential Read*). Die Adresse befindet im SRAM (Label: **E24ADL**, **E24ADH**). Die 64 empfangenen Datenbyte sollen im SRAM abgespeichert werden (Label: **E24BYTE**).

⌂ **D104** Teste das Abspeichern und Einlesen eines Byte. Die Speicherzelle **E24BYTE** muss nach dem Abspeichern gelöscht werden! Das empfangene Datenbyte soll auf dem 7-Segmentdisplay dargestellt werden.
Nenne dein Hauptprogramm "**D104_I2C_EEPROM_test1.asm**".

⌂ **D105** Teste das Abspeichern und Einlesen einer Seite (64 Byte). Die Tabelle **E24PAGE** muss nach dem Abspeichern gelöscht werden! Die empfangenen Datenbyte sollen in einer Schleife auf dem 7-Segmentdisplay dargestellt werden.
Nenne dein Hauptprogramm "**D105_I2C_EEPROM_test2.asm**".

Weitere Aufgaben (für Fortgeschrittene)

Mit dem I^2C-Speicherchip lässt sich zusammen mit einem D/A-Wandler und einem Mikrofon (MICES2-Board) ein Aufnahmegerät programmieren, das ungefähr 4 Sekunden Sprache oder Musik aufnimmt und diese wieder abspielt. Schließt man zusätzlich ein Potentiometer (0-5 V) an einen zweiten A/D-Wandler-Eingang (**PA1**) an, so lässt sich die Abspielgeschwindigkeit verändern.

Hier das vollständige Programm:

```
;*****************************************************************************
;*
;*      Titel:  Aufnahmegeraet mit EEPROM (24LC256)
;*      (C306_I2C_EEPROM_ADDA.asm)
;*      Datum:  25/01/11        Version:        0.1
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*
;*      Prozessor: ATmega32A                    Quarzfrequenz:  16MHz
;*      Eingaenge: Potentiometer (0-5V) an PA1, Mikrofonschaltung an PA0 (0-5V)
;*                 Aufnahmetaster an PC2
;*      Ausgaenge: D/A-Wandler an PortB, EEPROM (I2C) an SCL und SDA
;*                 (PC0 und PC1), LED Aufnahme an PC2, LED Abspielen an PC4
;*
;*      Informationen zur Funktionsweise:
;*
;*****************************************************************************

;-----------------------------------------------------------------------------
;       Einbinden der controllerspezifischen Definitionsdatei
;-----------------------------------------------------------------------------
.NOLIST                         ;List-Output ausschalten
.INCLUDE "m32def.inc"           ;AVR-Definitionsdatei einbinden
.LIST                           ;List-Output wieder einschalten

;-----------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
;-----------------------------------------------------------------------------
.DSEG                           ;was ab hier folgt kommt in den SRAM-Speicher
E24ADL:  .BYTE 1                ;16 Bit Adressezeiger EEPROM
E24ADH:  .BYTE 1
E24BYTE: .BYTE 1                ;Datenbyte EEPROM
E24PAGE: .BYTE 64               ;64 Byte Seite (Page) im EEPROM
TCNT:    .BYTE 1                ;Tabellenzeiger
TAB1:    .BYTE 128              ;128 Byte Tabelle

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                           ;was ab hier folgt kommt in den FLASH-Speicher
.ORG    0x0000                  ;Programm beginnt an der FLASH-Adresse 0x0000
RESET1: rjmp    INIT            ;springe nach INIT (ueberspringe ISR Vektoren)

;-----------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-----------------------------------------------------------------------------
        ;Vektortabelle (im Flash-Speicher)
.ORG    OC2addr                 ;interner Vektor fuer Timer2 Compare
        rjmp    ISR_T2          ;Bei TCNT2=OCR2 springe zur ISR Timer2
.ORG    OC0addr                 ;interner Vektor fuer Timer0 Compare
        rjmp    ISR_T0          ;Bei TCNT0=OCR0 springe zur ISR Timer0
.ORG    ADCCaddr                ;interner Vektor fuer ADCC (alt.: .ORG 0x0020)
                                ;AD-Wandlung vollstaendig ADC complete
        rjmp    ISR_AD          ;Springe zur ISR von ADCC

;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 1 wird zum Rechnen benoetigt
        clr     r15             ;und mit Null belegt
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
.DEF    Tmp2 = r17              ;Register 17 dient als zweiter Zwischenspeicher
.DEF    Cnt1 = r18              ;Register 18 dient als Zaehler
.DEF    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister W und zur Parameteruebergabe
```

```
        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
        ldi     Tmp1,HIGH(RAMEND)       ;RAMEND (SRAM) ist in der Definitions-
        out     SPH,Tmp1                ;datei festgelegt
        ldi     Tmp1,LOW(RAMEND)
        out     SPL,Tmp1

        ;ADC initialisieren
        ldi     Tmp1,0b01100000 ;REFS = 01 AVCC als Referenzspannung
        out     ADMUX,Tmp1      ;ADLAR = 1 linksb. (obere 8 Bit in ADCH)
                                ;MUX = 00000 unsymetrisch PA0 (ADC0)
        ldi     Tmp1,0b10100100 ;ADEN = 1 (ADC einschalten)
        out     ADCSRA,Tmp1     ;ADATE = 1 (Auto Trigger Modus ein)
                                ;ADIE = 0  (noch kein Interrupt)
                                ;ADPS = 100 (16MHz/16 = 1MHz)
                                ;andere Bits per default = 0
        in      Tmp1,SFIOR      ;ADTS = 011 Timer 0 Compare
        andi    Tmp1,0b01111111
        ori     Tmp1,0b01100000
        out     SFIOR,Tmp1

        ;Timer0 einschalten und Vorteiler festlegen (SAVE)
        ldi     Tmp1,0x0B       ;TCCR0 = 0b00001011 (Systemtakt/64)
        out     TCCR0,Tmp1      ;WGM = 01 CTC-Modus
        ;Vergleichswert T0 fuer den Zaehler festlegen
        ldi     Tmp1,36         ;36 Zaehlschritte = 6,8kHz; fs = 3,4kHz
        out     OCR0,Tmp1

        ;Timer2 einschalten und Vorteiler festlegen (PLAY)
        ldi     Tmp1,0x0C       ;TCCR0 = 0b00001100 (Systemtakt/64)
                                ;!! CS2 <> CS0 anders als Timer0!!)
        out     TCCR2,Tmp1      ;WGM = 01 CTC-Modus
        ;Vergleichswert T2 fuer den Zaehler festlegen
        ldi     Tmp1,36        ;50 Zaehlschritte = 5000Hz
        out     OCR2,Tmp1

        ;D/A-Wandler initialisieren
        ser     Tmp1            ;D/A-Wandler an PORTB
        out     DDRB,Tmp1

        ;Aufnahmetaster und LEDs initialisieren
        cbi     DDRC,2          ;Aufnahmetaster an PC2
        sbi     PORTC,2         ;Pull-Up
        sbi     DDRC,3          ;AufnahmeLED (SAVE) an PC3
        sbi     DDRC,4          ;AbspieLED (PLAY) an PC4

        ;I2C Bus initialisieren (400 kHz)
        rcall   I2CINI

        ;Interrupts global erlauben
        sei

;-------------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------------
MAIN:   ;T2 Interrupt (PLAY) ausschalten
        in      Tmp1,TIMSK      ;
        andi    Tmp1,0x7F       ;Mit einer UND-Maskierung (0b01111111)
                                ;Bit 7 (OCIE2) im TIMSK-Register loeschen
        out     TIMSK,Tmp1      ;
        cbi     PORTC,4         ;Abspiel LED loeschen

        ;Potentiometer zum Verstellen der Abspielgeschwindigkeit an PA1
        ;Option! dieser Teil kann falls erwuenscht geloescht werden
        ldi     Tmp1,0b01100001 ;REFS = 01 AVCC als Referenzspannung
        out     ADMUX,Tmp1      ;ADLAR = 1 linksb. (obere 8 Bit in ADCH)
        ldi     Tmp1,0b10000100 ;ADEN = 1 (ADC einschalten)
        out     ADCSRA,Tmp1     ;ADPS = 100 (16MHz/16 = 1MHz)
        sbi     ADCSRA,ADSC     ;eine AD-Wandlung ausloesen
LOOPV:  sbic    ADCSRA,ADSC     ;Polling bis Wandlung fertig
        rjmp    LOOPV
```

```
        in      Tmp1,ADCH       ;oberste 8 Bit einlesen und ausgeben
        lsr     Tmp1            ;Wert durch 4 teilen
        lsr     Tmp1
        brne    LOOPG           ;auf Endstellung (0)
        ldi     Tmp1,36         ;normale Abspielgeschwindigkeit
LOOPG:  out     OCR2,Tmp1
        ldi     Tmp1,0b01100000 ;A/D Wandler zuruecksetzen (PA0 Mikrofoneingang)
        out     ADMUX,Tmp1
        ldi     Tmp1,0b10100100
        out     ADCSRA,Tmp1

        sbic    PINC,2          ;Aufnahmetaster
        rjmp    PLAY

SAVE:   ;Timer0 (SAVE) Interrupt einschalten
        in      Tmp1,TIMSK      ;
        ori     Tmp1,0x02       ;Mit einer ODER-Maskierung (0b00000010)
                                ;Bit 2 (OCIE0) im TIMSK-Register auf eins setzen
        out     TIMSK,Tmp1      ;
        sbi     ADCSRA,ADIE

        ;Anfangsadresse und Tabellenzaehler initialisieren, AufnahmeLED ein
        clr     Tmp1
        sts     E24ADL,Tmp1
        sts     E24ADH,Tmp1
        sts     TCNT,Tmp1
        sbi     PORTC,3

SAVE0:  lds     Tmp1,TCNT       ;zweite Haelfte der Tabelle?
        cpi     Tmp1,64
        brge    SAVE1
        rjmp    SAVE0

SAVE1:  clr     Cnt1            ;PAGE (64 Byte) schreiben
        ldi     XL,LOW(E24PAGE)
        ldi     XH,HIGH(E24PAGE)
        ldi     YL,LOW(TAB1)
        ldi     YH,HIGH(TAB1)
LOOP2:  ld      Tmp1,Y+
        st      X+,Tmp1
        inc     Cnt1
        cpi     Cnt1,64
        brne    LOOP2
        rcall   E24SP

        ;Pageadresse erhoehen
        lds     XL,E24ADL
        lds     XH,E24ADH
        adiw    XL,32           ;64 addieren (naechste Seite)
        adiw    XL,32
        sts     E24ADL,XL
        sts     E24ADH,XH

LOOP3:  lds     Tmp1,TCNT       ;erste Haelfte der Tabelle?
        cpi     Tmp1,64
        brlo    SAVE2
        rjmp    LOOP3

SAVE2:  clr     Cnt1            ;PAGE (64 Byte) schreiben
        ldi     XL,LOW(E24PAGE)
        ldi     XH,HIGH(E24PAGE)
        ldi     YL,LOW(TAB1+64)
        ldi     YH,HIGH(TAB1+64)
LOOP4: ld      Tmp1,Y+
        st      X+,Tmp1
        inc     Cnt1
        cpi     Cnt1,64
        brne    LOOP4
        rcall   E24SP

        ;Pageadresse erhoehen
```

```
        lds     XL,E24ADL
        lds     XH,E24ADH
        adiw    XL,32           ;64 addieren (naechste Seite)
        adiw    XL,32
        brmi    PLAY            ;Falls EEPROM voll (Bit15 = 1) dann Raus
        sts     E24ADL,XL
        sts     E24ADH,XH
        rjmp    SAVE0

PLAY:   ;T0 Interrupt (SAVE) ausschalten
        in      Tmp1,TIMSK      ;
        andi    Tmp1,0xFD       ;Mit einer UND-Maskierung (0b11111101)
                                ;Bit 1 (OCIE0) im TIMSK-Register loeschen
        out     TIMSK,Tmp1      ;
        cbi     ADCSRA,ADIE

        ;Anfangsadresse und Tabellenzaehler initialisieren, Abspiel LED ein
        clr     Tmp1
        sts     E24ADL,Tmp1
        sts     E24ADH,Tmp1
        sts     TCNT,Tmp1
        sbi     PORTC,4

PLAY2:  rcall   E24RS           ;Werte an der gleichen Adresse abholen (1,53ms)
        clr     Cnt1            ;und ausgeben
        ldi     XL,LOW(E24PAGE)
        ldi     XH,HIGH(E24PAGE)
        ldi     YL,LOW(TAB1)
        ldi     YH,HIGH(TAB1)
LOOP5:  ld      Tmp1,X+
        st      Y+,Tmp1
        inc     Cnt1
        cpi     Cnt1,64
        brne    LOOP5

        ;Pageadresse erhoehen
        lds     XL,E24ADL
        lds     XH,E24ADH
        adiw    XL,32           ;64 addieren (naechste Seite)
        adiw    XL,32
        ;brmi    MAINR          ;Falls EEPROM voll (Bit15 = 1) dann Raus
        sts     E24ADL,XL
        sts     E24ADH,XH

LOOP6:  lds     Tmp1,TCNT       ;erste Haelfte der Tabelle?
        cpi     Tmp1,64
        brlo    PLAY3
        rjmp    LOOP6

PLAY3:  lds     Tmp1,E24ADL     ;erstes Mal?
        cpi     Tmp1,0x40
        brne    PLAY4
        lds     Tmp1,E24ADH
        cpi     Tmp1,0x00
        brne    PLAY4

        ;T2 Interrupt einschalten
        in      Tmp1,TIMSK      ;
        ori     Tmp1,0x80       ;Mit einer ODER-Maskierung (0b10000000)
                                ;Bit 7 (OCIE2) im TIMSK-Register auf eins setzen
        out     TIMSK,Tmp1      ;
        cbi     PORTC,3

PLAY4:  rcall   E24RS           ;Werte an der gleichen Adresse abholen (1,53ms)
        clr     Cnt1            ;und ausgeben
        ldi     XL,LOW(E24PAGE)
        ldi     XH,HIGH(E24PAGE)
        ldi     YL,LOW(TAB1+64)
        ldi     YH,HIGH(TAB1+64)
LOOP7:  ld      Tmp1,X+
        st      Y+,Tmp1
```

```
        inc     Cnt1
        cpi     Cnt1,64
        brne    LOOP7

        ;Pageadresse erhoehen
        lds     XL,E24ADL
        lds     XH,E24ADH
        adiw    XL,32           ;64 addieren (naechste Seite)
        adiw    XL,32
        brmi    MAINR           ;Falls EEPROM voll (Bit15 = 1) dann Raus
        sts     E24ADL,XL
        sts     E24ADH,XH

LOOP8:  lds     Tmp1,TCNT       ;zweite Haelfte der Tabelle?
        cpi     Tmp1,64
        brge    PLAYR
        rjmp    LOOP8

PLAYR:  rjmp    PLAY2           ;Endlosschleife

MAINR:  rjmp    MAIN

;-----------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-----------------------------------------------------------------------------
; Interrupt-Behandlungsroutine Timer2 (PLAY)
ISR_T2: push    r16             ;benutzte Reg. retten (r16 = Zwischenspeicher)
        in      r16,SREG        ;Statusregister einlesen
        push    r16             ;Statusregister retten
        push    r17
        push    YL
        push    YH

        cbi     PORTC,4         ;dient der Visualisierung mittels Oszilloskop
        nop
        sbi     PORTC,4

        lds     r17,TCNT
        ldi     YL,LOW(TAB1)
        ldi     YH,HIGH(TAB1)
        add     YL,r17
        adc     YH,Zero
        ld      r16,Y
        out     PORTB,r16       ;Wert aus Tabelle ausgeben
        inc     r17
        cpi     r17,128
        breq    ISR_TF
        sts     TCNT,r17
        rjmp    ISR_TR
ISR_TF: clr     r17
        sts     TCNT,r17

ISR_TR: pop     YH
        pop     YL
        pop     r17
        pop     r16             ;Werte der geretteten Register wieder-
        out     SREG,r16        ;herstellen
        pop     r16
        reti                    ;Ruecksprung aus einer ISR

;Dummy Interrupt-Behandlungsroutine
ISR_T0: reti                    ;Ruecksprung aus einer ISR

;Interrupt-Behandlungsroutine ADC (SAVE)
ISR_AD: push    r16             ;benutzte Reg. retten (r16 = Zwischenspeicher)
        in      r16,SREG        ;Statusregister einlesen
        push    r16             ;Statusregister retten
        push    r17
        push    YL
        push    YH
```

```
        cbi     PORTC,3         ;dient der Visualisierung mittels Oszilloskop
        nop
        sbi     PORTC,3

        lds     r17,TCNT
        ldi     YL,LOW(TAB1)
        ldi     YH,HIGH(TAB1)
        add     YL,r17
        adc     YH,Zero
        in      r16,ADCH        ;oberste 8 Bit einlesen und
        st      Y,r16           ;abspeichern
        inc     r17
        cpi     r17,128
        breq    ISR_AF
        sts     TCNT,r17
        rjmp    ISR_AR
ISR_AF: clr     r17
        sts     TCNT,r17

ISR_AR: pop     YH
        pop     YL
        pop     r17
        pop     r16             ;Werte der geretteten Register wieder-
        out     SREG,r16        ;herstellen
        pop     r16
        reti                    ;Ruecksprung aus einer ISR

.INCLUDE "SR_I2C.asm"                   ;I2C-Bibliothek einbinden
.INCLUDE "SR_EEPROM_24LC256.asm"        ;EEPROM-Bibliothek einbinden
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EXIT                           ;Ende des Quelltextes
```

⌂ **D106** Das obige Programm "**D106_I2C_EEPROM_ADDA.asm**" findet man auf der Seite http://weigu.lu/tutorials/avr_assembler.

a) Programmiere die Aufgabe und teste sie (Voraussetzung ist eine funktionierende EEPROM-Bibliothek).

b) Die Kommentare des Programms sind absichtlich lückenhaft. Ermittle die genaue Funktionsweise des Programms und ergänze die Kommentare. (Um die ISRs zu visualisieren kann man an den beiden LEDs ein Oszilloskop anschließen und so das genaue Timing beobachten.)

c) Zeichne die entsprechenden Flussdiagramme.

D2 Die SPI Schnittstelle

Einführung

Motorola entwickelte die **synchrone SPI-Master-Slave Schnittstelle**, (***Serial Periphal Interface***) für die Kommunikation zwischen Mikrocontrollern. Ein ähnliches Bus-System existiert von National Semiconductor und nennt sich **Microwire**. Die Schnittstelle arbeitet synchron in Vollduplex mit hoher Taktgeschwindigkeit (bis zu mehreren 10 MHz). Die SPI-Schnittstelle wird bei der AVR-Familie zur *In-System*-Programmierung genutzt und meist hardwaremäßig unterstützt.

Im Gegensatz zur asynchronen seriellen EIA-232 Schnittstelle werden keine Start-, Stopp- oder Paritätsbits verwendet. Auch wird keine Adresse wie bei der synchronen I^2C-Schnittstelle benötigt.

Vorteile Schnell, da Vollduplex. Die Taktrate kann bis zur Hälfte des Systemtakts des Controllers betragen. Es werden keine Steuerbits benötigt. Es können mehrere Master abwechselnd am Bus arbeiten.

Nachteile Die überbrückbare Distanz ist gering. Der Verdrahtungsaufwand ist meist höher als bei I^2C (3 bis 5 Leitungen).

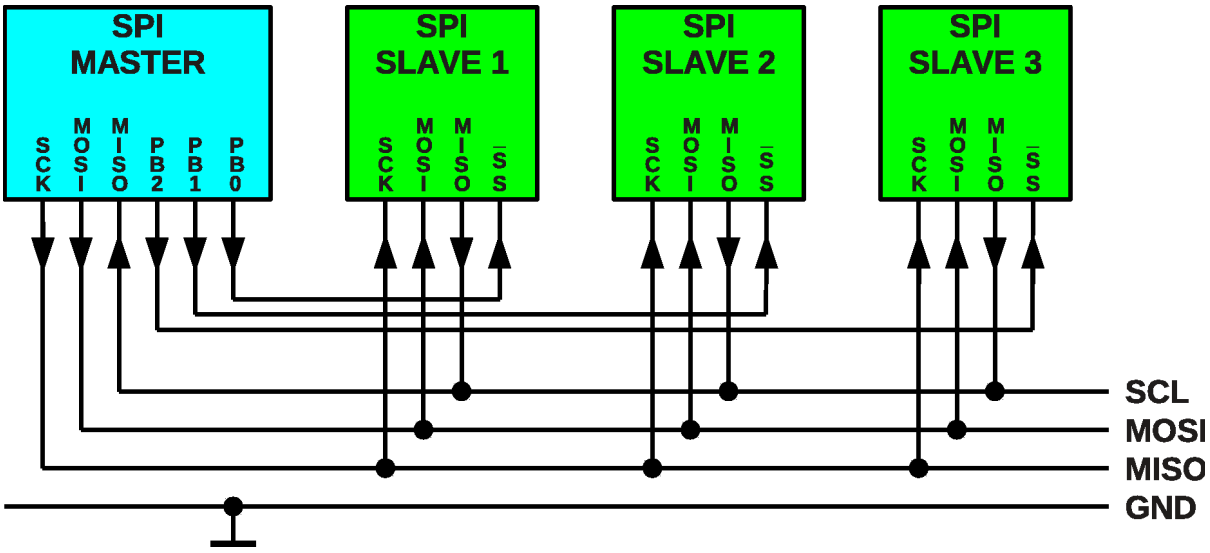
Verwendung: Datenaustausch zwischen Mikrocontrollern, aber auch mit anderen ICs (z.B. EEPROM) oder Sensoren welche über diese Schnittstelle verfügen. Zusätzliche Ein/Ausgabeports mittels TTL-Schieberegistern (Wandlung seriell-parallel).

Die SPI Schnittstelle arbeitet mit 8-Bit Schieberegistern. Es kann frei gewählt werden, mit welcher Geschwindigkeit (mit Vorteiler aus dem Systemtakt abgeleitet) gearbeitet wird und ob MSB oder LSB zuerst gesendet werden.

Aufbau der SPI-Schnittstelle

Verdrahtung und Schieberichtung

Im folgenden Bild sind ein Master und drei Slaves eingezeichnet. Beliebige Ausgänge (hier PB0-PB2) dienen dem Master, zum Auswählen des anzusteuernden Bausteins (***/SS Slave Select***). Die "Slave Select"-Eingänge sind dabei **Low-Aktiv!** Bemerkt der Slave, dass seine SS-Leitung auf Low gezogen wurde (fallende Flanke), so beginnt für ihn die Übertragung. Diese Leitung dient also auch der Synchronisation der Übertragung zwischen Master und Slave.



Die SPI-Schnittstelle arbeitet synchron, d.h. mit einer gemeinsamen Taktleitung. Auf der ***Serial Clock*** Leitung (**SCK**) legt der Master das Taktsignal an, solange wie SS Low ist. Dazu muss allerdings mit einem Schreiben in das SPI-Datenregister die Kommunikation gestartet werden (siehe später).

MOSI bedeutet ***Master Out Slave In*** und wird 1:1 verbunden. Für den Master ist diese Leitung ein Ausgang (muss initialisiert werden) und beim Slave automatisch ein Eingang. Der Master sendet über diese Leitung. **MISO** bedeutet ***Master In Slave Out*** und wird auch 1:1 verbunden. MISO ist für den Master automatisch ein Eingang. Er empfängt Daten vom Slave über diese Leitung. Vollduplex ist also mit der SPI Schnittstelle möglich. Beim Slave muss dieses Pin als Ausgang initialisiert werden.

Pin	Master	Slave
MOSI	als Ausgang zu initialisieren	automatisch Eingang
MISO	automatisch Eingang	als Ausgang zu initialisieren
SCK	als Ausgang zu initialisieren	automatisch Eingang
/SS	als Ausgang zu initialisieren	automatisch Eingang

Es soll hier nicht auf den möglichen Multimasterbetrieb eingegangen werden. Dass /SS Pin des Masters muss für den Single Master-Betrieb als Ausgang gesetzt werden!

Je nach angeschlossenen Slave muss die Schnittstelle flexibel konfigurierbar sein. Mit Hilfe des **DORD**-Bit im Kontrollregister kann die Schieberichtung festgelegt werden. Es besteht die Möglichkeit das höchstwertige Bit zuerst (**DORD** = 0) rauszuschieben, oder mit dem niederwertigsten Bit zu beginnen(**DORD** = 1).

Mit zwei weiteren Bits lässt sich auch die Taktleitung und der Zeitpunkt der Übernahme von Daten beeinflussen:

Die SPI-Modi und die Geschwindigkeit

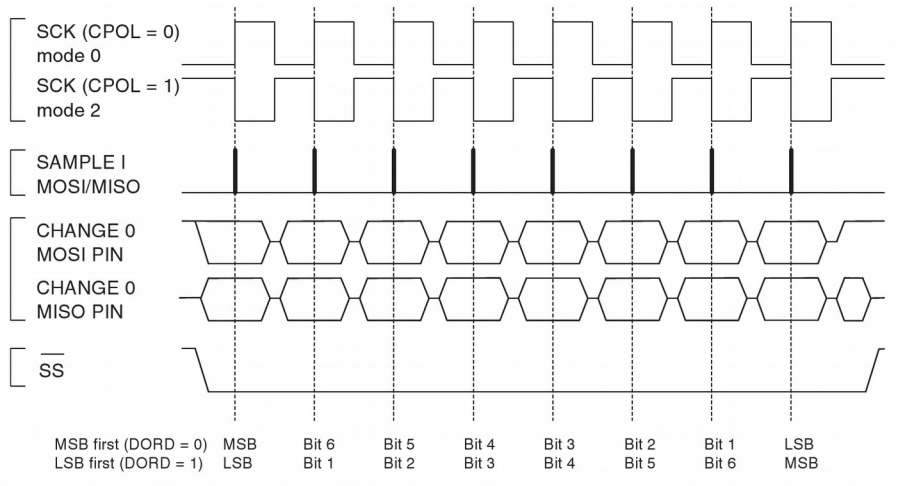
Mit den beiden Bits **CPOL** und **CPHA** kann einer von vier SPI-Modi ausgewählt werden. **CPOL** bestimmt dabei die Polarität der SCK-Leitung (0: Ruhezustand der Taktleitung Low, 1: Ruhezustand High) und **CPHA** die Phasenlage, d. h. bei welcher Flanke die Daten übernommen werden sollen (0: erste Flanke sofort, 1: zweite Flanke nach halber Taktzeit (180°)).

Zwischen dem Abtasten (sample) und der Ausgabe (Schieben) des Bits bleibt immer eine halbe Periode Zeit, damit die Signale sich stabilisieren können.

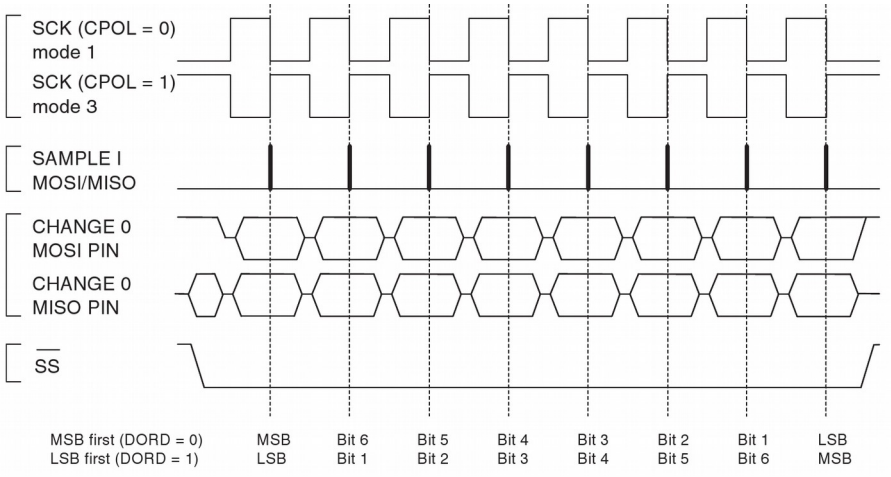
SPI-Modus	CPOL	CPHA	erste Flanke	zweite Flanke
0	0	0	Bit abtasten (steigende Flanke)	Bit ausgeben (fallende Flanke)
1	0	1	Bit ausgeben (steigende Flanke)	Bit abtasten (fallende Flanke)
2	1	0	Bit abtasten (fallende Flanke)	Bit ausgeben (steigende Flanke)
3	1	1	Bit ausgeben (fallende Flanke)	Bit abtasten (steigende Flanke)

Im Modus 0 legt der Slave seine Daten schon beim Runterziehen von SS an MISO an. Der Master kann sie dann beim ersten Flankenwechsel übernehmen. Hier ist auf das Timing zu achten (ausreichen Zeit zwischen beiden Aktionen damit kein Bit verloren geht).

CPHA = 0



CPHA = 1



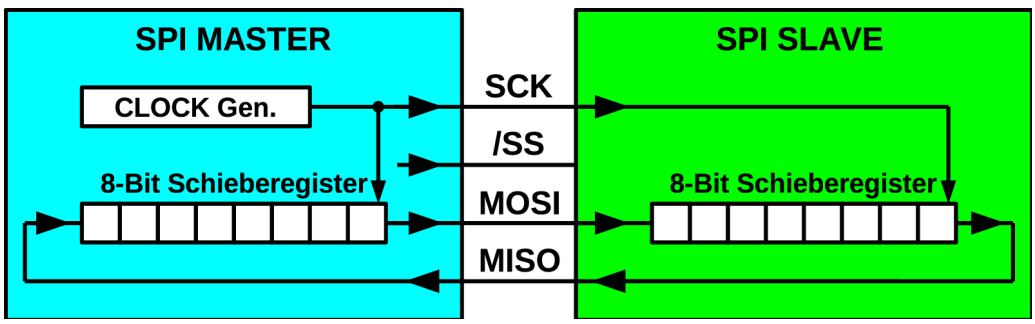
Quelle: Datenblatt ATmega32A

Die Frequenz des SPI-Taktsignals ist an sich beliebig und wird durch einen Teilungsfaktor festgelegt. Die Taktfrequenz des Controllers (Systemtakt) wird dabei durch 2, 4, 8, 16, 32, 64 oder 128 geteilt. Die maximale Geschwindigkeit des Masters liegt mit 16 MHz Quarz also bei 8 MHz. Die Geschwindigkeit des Slave soll allerdings nicht höher als Systemtakt/4 sein!

Funktionsweise der SPI-Schnittstelle

Der Master legt den Schiebetakt auf die SCK-Leitung, nachdem er ein Byte im Schieberegister abgelegt hat. Mit jeder Periode des Taktsignals wird 1 Bit vom Master-Schieberegister ins Slave-Schieberegister und gleichzeitig vom Slave-Schieberegister ins Master-Schieberegister geschoben. Nach 8 Taktzyklen ist die Übertragung abgeschlossen. Die beiden 8-Bit Schieberegister können als großes 16-Bit-Schieberegister betrachtet werden!!

Bei jeder Datenübertragung wird immer gleichzeitig ein Byte gesendet und empfangen!



Will der Master nur Senden, so ignoriert er die vom Slave ankommenden Bytes, d.h er liest sie nicht ein. Will er nur Empfangen, so ignoriert der Slave die nichtssagenden ankommenden Bytes vom Master.

Sowohl beim Master als auch beim Slave wird das Ende der Übertragung eines Byte, nach 8 Taktzyklen durch das SPI-Interrupt-Flag **SPIF** im SPI-Statusregister angezeigt. Dieses kann mittels Polling abgefragt werden oder einen Interrupt auslösen.

Daten können im SPI-Datenregister **SPDR** abgelegt oder empfangen werden, nachdem **SPIF** gesetzt wurde.

Schreibt man in das Datenregister **SPDR**, so werden die Daten im Sendepuffer abgelegt, bevor sie ins Schieberegister wandern. Da der Sendepuffer nur ein Byte groß ist, sind weitere Schreibzugriffe ins Datenregister während einer Übertragung gesperrt. Wird dies trotzdem versucht, so wird die Übertragung nicht gestört, sondern die Schreib-Kollision wird mit dem Flag **WCOL** (*Write COLlision*) im Statusregister **SPSR**. Bei Bedarf kann dies kontrolliert werden. Um Kollisionen zu vermeiden, soll immer unmittelbar nach einer Übertragung (SPIF = 1) ins Datenregister geschrieben werden.

Der Lesebuffer beträgt zwei Bytes. Liest man Daten aus dem Datenregister **SPDR**, so kann dies auch noch während der Übertragung des folgenden Bytes geschehen. Das Lesen muss allerdings abgeschlossen sein bevor die Übertragung beendet wird.

Die Initialisierung der SPI-Schnittstelle

Damit eine Kommunikation per SPI möglich ist, müssen Sender und Empfänger richtig initialisiert werden.

1. Die Schnittstelle muss eingeschaltet werden, und es muss festgelegt werden ob der Controller als Master oder Slave arbeitet (**SPE**, **MSTR**).

2. Beim Master müssen **MOSI**, **SCK** und **/SS** als Ausgang initialisiert werden, beim Slave nur **MISO** als Ausgang.

3. Die Schieberichtung und der verwendete SPI-Modus müssen festgelegt werden (**DORD**, **CPOL**, **CPHA**).

4. Die Geschwindigkeit muss festgelegt werden (**SPR0**, **SPR1**[7])

5. Wird mit Interrupts statt Polling gearbeitet, so muss neben der Initialisierung des Interrupt-Vektors, dem globalen Freischalten von Interrupts auch der SPI-Interrupt freigeschaltet werden (**SPIE**)

Die SPI Schnittstelle benötigt nur drei SF-Register.

- **SPCR** Die gesamte Initialisierung kann im **SPI-Kontrollregister** vorgenommen werden.
- **SPSR** Das **SPI-Statusregister** dient hauptsächlich der Abfrage des SPIInterrupt-Flags **SPIF** beim Polling.

7 Nur im Fall, dass die höchst mögliche Geschwindigkeit nötig ist oder der Teilungsfaktor 8 bzw. 32 benötigt wird, muss zusätzlich das Bit zur Verdopplung der Geschwindigkeit im SPI-Statusregister gesetzt werden (SPI2X in SPSR)

- **SPDR** Daten werden über das **SPI-Datenregister** in den Sendepuffer geschrieben bzw. aus dem Empfangspuffer gelesen.

Die SF-Register der SPI-Schnittstelle

Das SPI Control Register SPCR

Das **SPI Control Register** befindet sich auf der SRAM-Adresse **0x002D** (SF-Register-Adresse **0x0D**) und wird mit der Abkürzung "**SPCR**" angesprochen (Definitionsdatei). Die Befehle **sbi**, **cbi**, **sbic** und **sbis** können verwendet werden.

SPCR = SPI Control Register

Bit	7	6	5	4	3	2	1	0
SPCR 0x0D	**SPIE**	**SPE**	**DORD**	**MSTR**	**CPOL**	**CPHA**	**SPR1**	**SPR0**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

SPIE ***SPI Interrupt Enable***

0 kein SPI-Interrupt erlaubt.

1 In dem Moment, wo die Übertragung aller 8 Bit beendet ist, wird das SPI Interrupt-Flag **SPIF** im Statusregister **SPSR** gesetzt. Das Setzen des **SPIE**-Flag **ermöglicht das Auslösen eines Interrupts**, sobald **SPIF** gesetzt wurde. Interrupts müssen dazu global frei gegeben sein (**I** = 1 im **SREG** mit "**sei**"). **SPIF** wird hardwaremäßig gelöscht wenn der Interrupt abgearbeitet wird.

SPE ***SPI Enable***

0 schaltet SPI aus.

1 schaltet SPI ein. Muss gesetzt werden, damit SPI-Operationen durchgeführt werden.

DORD ***Data ORDer***

0 das höchstwertigste Bit **MSB** wird als Erstes gesendet.

1 das niederwertigste Bit **LSB** wird als Erstes gesendet.

MSTR ***Master/Slave Select***

0 Controller ist Slave.

1 Controller ist Master. Das Bit muss vor oder gleichzeitig mit dem **SPE** Bit gesetzt werden!

CPOL ***Clock Polarity***

0 Ruhezustand der Taktleitung ist Low.

1 Ruhezustand der Taktleitung ist High.

CPHA ***Clock PHAse***

0 Daten werden sofort bei der ersten Flanke übernommen.

1 Daten werden bei der zweiten Flanke nach der halben Taktzeit (180°) übernommen.

Mit **CPOL** und **CPHA** wird der SPI-Modus festgelegt:

CPOL CPHA	SPI Modus:
00	0
01	1

10	2
11	3

SPRn ***SPI Clock Rate Select*** ***SPR2X, SPR1, SPR0***

Mit drei Bit wird die Taktgeschwindigkeit (Frequenz) des Busses **SCK** festgelegt. Dies betrifft nur den Master. Das Bit **SPR2X** befindet sich im SPI Statusregister **SPSR** (Bit 0).

SPR2X SPR1 SPR0 2^2 2^1 2^0	SCK Frequenz:
000	**Systemtakt / 4**
001	**Systemtakt / 16**
010	**Systemtakt / 64**
011	**Systemtakt / 128**
100	Systemtakt / 2
101	Systemtakt / 8
110	Systemtakt / 32
111	Systemtakt / 64

Weitere Informationen findet man im Datenblatt S 137.

Das SPI Status Register SPSR

Das **SPI Status Register** befindet sich auf der SRAM-Adresse **0x002E** (SF-Register-Adresse **0x0E**) und wird mit der Abkürzung "**SPSR**" angesprochen (Definitionsdatei). Die Befehle **sbi**, **cbi**, **sbic** und **sbis** können verwendet werden.

SPSR = SPI Status Register

Bit	7	6	5	4	3	2	1	0
SPSR 0x0E	**SPIF**	**WCOL**	**-**	**-**	**-**	**-**	**-**	**SPI2X**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R	R	R	R	R	R	R	R/W

SPIF ***SPI Interrupt Flag***

0 Übertragung noch nicht beendet.

1 In dem Moment, wo die Übertragung aller 8 Bit beendet ist wird das SPI Interrupt-Flag **SPIF** gesetzt. Das Setzen des **SPIE**-Flag in **SPCR** ermöglicht das Auslösen eines Interrupts. **SPIF** wird hardwaremäßig gelöscht wenn der Interrupt abgearbeitet wird. Wird **SPIF** im Polling-Betrieb abgefragt, so kann das Flag durch Lesen des Statusregisters **SPSR** bei gesetztem **SPIF** und anschließendem Lesen oder Schreiben des Datenregisters **SPDR** gelöscht werden.

WCOL ***Write COLlision Flag***

0 keine Schreibkollision aufgetreten.

1 Wird **SPDR** während der Datenübertragung beschrieben, so meldet das **WCOL**-Flag eine Schreibkollision. Das Flag kann (zusammen mit **SPIF**) durch Lesen des Statusregisters **SPSR** bei

gesetztem **WCOL** und anschließendem Lesen oder Schreiben des Datenregisters **SPDR** gelöscht werden.

SPI2X ***Double SPI Speed Bit***

0 keine Verdopplung der Taktgeschwindigkeit des Busses.

1 Verdopplung der Taktgeschwindigkeit des Busses (siehe Tabelle **SPCR**).

Das SPI Data Register SPDR

Das **SPI Data Register** befindet sich auf der SRAM-Adresse **0x002F** (SF-Register-Adresse **0x0F**) und wird mit der Abkürzung "**SPDR**" angesprochen (Definitionsdatei). Die Befehle **sbi**, **cbi**, **sbic** und **sbis** können verwendet werden. Das Register ist lese- und schreibbar.

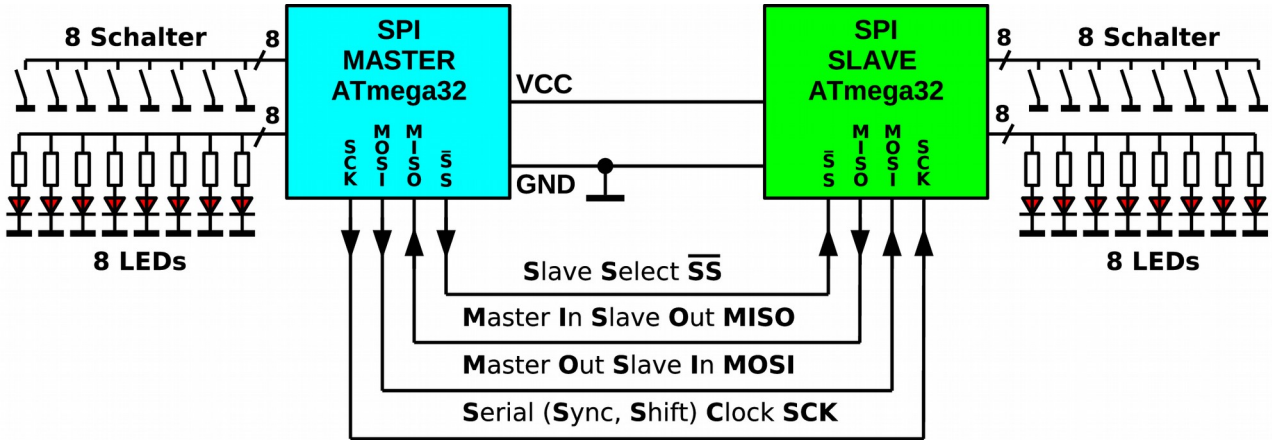
SPDR = SPI Data Register

Bit	7 (MSB)	6	5	4	3	2	1	0 (LSB)
SPDR 0x0F	**SPDR7**	**SPDR6**	**SPDR5**	**SPDR4**	**SPDR3**	**SPDR2**	**SPDR1**	**SPDR0**
Startwert	-	-	-	-	-	-	-	-
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Der Startwert ist undefiniert.

Ansteuerung eines Slaves mittels Polling

Mittels zweier ATmega32A soll eine SPI-Übertragung in beide Richtungen zwischen einem Master und einem Slave realisiert werden. Beide Controller besitzen 8 Schalter und 8 LEDs. Die Schalterstellung des Master soll an den LEDs des Slave sichtbar sein und umgekehrt.



Programm für den Master:

```
        ;Schalter und LEDs initialisieren
        clr     Tmp1            ;8 Schalter an PORTA
        out     DDRA,Tmp1
        ser     Tmp1
        out     PORTA,Tmp1      ;Pull-Ups
        out     DDRC,Tmp1       ;8 LEDs an PORTC
```

```
        ;SPI Master initialisieren
        ;SPI-Ausgaenge initialisieren (PB6 (MISO) automatisch Eingang)
        sbi     DDRB,4          ;PB4 Ausgang (/SS) muss beim Master Mode vor dem
                                ;Einschalten der Schnittstelle geschehen!
        sbi     DDRB,5          ;PB5 Ausgang (MOSI)
        sbi     DDRB,7          ;PB7 Ausgang (SCK)
        ;SPI einschalten, Master mit SCK = Systemtakt/128, MSB first, SPI Mode 1
        ldi     Tmp1,0x57       ;SPCR = 0b01010111
        out     SPCR,Tmp1       ;B7 SPIE=0 kein Interrupt
                                ;B6 SPE=1 SPI einschalten
                                ;B5 DORD=0 MSB zuerst
                                ;B4 MSTR=1 Master
                                ;B32 CPOL=0 CPHA=1 SPI Modus 1
                                ;B10 SPR1=1 SPR0=1 Systemtakt/128 (125khz)
        ;Slave aktivieren
        cbi     PORTB,4         ;Slave aktivieren

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   in      Tmp1,PINA       ;Schalter einlesen und ueber SPI uebertragen
        out     SPDR,Tmp1       ;Starte die Uebertragung
W_SPIF: sbis    SPSR,SPIF       ;Warte bis Uebertragung fertig
        rjmp    W_SPIF
        in      Tmp1,SPDR       ;Empfange Byte vom Slave
        out     PORTC,Tmp1      ;und Ausgabe an LEDs
        rjmp    MAIN            ;Endlosschleife
```

Programm für den Slave:

```
        ;Schalter und LEDs initialisieren
        clr     Tmp1            ;8 Schalter an PORTA
        out     DDRA,Tmp1
        ser     Tmp1
        out     PORTA,Tmp1      ;Pull-Ups
        out     DDRC,Tmp1       ;8 LEDs an PORTC

        ;SPI Slave initialisieren
        ;SPI-Ausgang initialisieren
        ;(PB4(/SS), PB5(MOSI) und PB7(SCK) automatisch Eingang)
        sbi     DDRB,6          ;PB6 Ausgang (MISO)
        ;SPI einschalten, Slave mit SCK = Systemtakt/128, MSB first, SPI Mode 1
        ldi     r16,0x47        ;SPCR = 0b01000111
        out     SPCR,r16        ;B7 SPIE=0 kein Interrupt
                                ;B6 SPE=1 SPI einschalten
                                ;B5 DORD=0 MSB zuerst
                                ;B4 MSTR=0 Slave
                                ;B32 CPOL=0 CPHA=1 SPI Modus 1
                                ;B10 SPR1=0 SPR0=1 Systemtakt/128 (125khz)

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   in      Tmp1,PINA       ;Schalter einlesen
W_SPIF: sbis    SPSR,SPIF       ;Warte bis Empfang fertig
        rjmp    W_SPIF
        out     SPDR,Tmp1       ;Daten ueber SPI uebertragen
        in      Tmp1,SPDR       ;Empfangene Daten einlesen
        out     PORTC,Tmp1      ;und auf LEDs ausgeben
        rjmp    MAIN            ;Endlosschleife
```

⌂ **D200** Teste die Übertragung mit den beiden vorgegebenen Programme[8]. Schreibe die Programme so um, dass die Initialisierung und das Polling jeweils in einem Unterprogramm erfolgen.

8 Download auf www.weigu.lu/a/asm

Nenne die neuen Programme "**D200_SPI_master_polling.asm**" und "**D200_SPI_slave_polling.asm**".

Bemerkungen: Nach dem Programmieren ist ein Reset des Masters notwendig, damit der Slave richtig synchronisiert.
Bei zu hohen Frequenzen (16 MHz/4 = 4 MHz) und mit CHPA = 0 (Modus 0) funktionierte die Synchronisation nicht zuverlässig. Abhilfe schafft das konsequente Einsetzen der /SS Leitung um den Slave vor der Übertragung eines jeden Bytes zu synchronisieren.
Acht auf eine gemeinsame Masse zwischen beiden Schaltungen!

Ansteuerung eines Slaves mittels Interrupt

Die gleiche Aufgabe soll nun mit Interrupts erledigt werden.

Programm für den Master:

```
;-----------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-----------------------------------------------------------------------------
        ;Vektortabelle (im Flash-Speicher)
.ORG    SPIaddr                 ;interner Vektor fuer SPI (alt.: .ORG 0x0018)
        rjmp    ISRSPI          ;Springe zur ISR von SPI Transfer Ready
```

...

```
        ;Schalter und LEDs initialisieren
        clr     Tmp1            ;8 Schalter an PORTA
        out     DDRA,Tmp1
        ser     Tmp1
        out     PORTA,Tmp1      ;Pull-Ups
        out     DDRC,Tmp1       ;8 LEDs an PORTC
        ;SPI Master initialisieren
        ;SPI-Ausgaenge initialisieren (PB6 (MOSI) automatisch Eingang)
        sbi     DDRB,4          ;PB4 Ausgang (/SS) muss beim Master Mode vor dem
                                ;Einschalten der Schnittstelle geschehen!
        sbi     DDRB,5          ;PB5 Ausgang (MOSI)
        sbi     DDRB,7          ;PB7 Ausgang (SCK)
        ;SPI einschalten, Master mit SCK = Systemtakt/128, MSB first, SPI Mode 1
        ldi     r16,0xD7        ;SPCR = 0b11010111
        out     SPCR,r16        ;B7 SPIE=1 Interrupt !!
                                ;B6 SPE=1 SPI einschalten
                                ;B5 DORD=0 MSB zuerst
                                ;B4 MSTR=1 Master
                                ;B32 CPOL=0 CPHA=1 SPI Modus 1
                                ;B10 SPR1=0 SPR0=1 Systemtakt/128 (125 kHz)
        ;Slave aktivieren
        cbi     PORTB,4         ;Slave aktivieren

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   sei                     ;Interrupts Global erlauben
        clr     Tmp1
        out     SPDR,Tmp1       ;Uebertragung starten
END:    rjmp    END             ;Endlosschleife

;-----------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-----------------------------------------------------------------------------
;Interruptroutine zum Senden und Empfangen über den SPI Bus
ISRSPI: in      r16,PINA        ;Schalter einlesen und ueber SPI versenden
```

```
        out     SPDR,r16
        in      r16,SPDR        ;Empfange Byte vom Slave an den LEDs ausgeben
        out     PORTC,r16
        reti                    ;Rücksprung ins Hauptprogramm
```

Programm für den Slave:

```
;-----------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-----------------------------------------------------------------------------
        ;Vektortabelle (im Flash-Speicher)
.ORG    SPIaddr                 ;interner Vektor fuer SPI (alt.: .ORG 0x0018)
        rjmp    ISRSPI          ;Springe zur ISR von SPI Transfer Ready
```

...

```
        ;Schalter und LEDs initialisieren
        clr     Tmp1            ;8 Schalter an PORTA
        out     DDRA,Tmp1
        ser     Tmp1
        out     PORTA,Tmp1      ;Pull-Ups
        out     DDRC,Tmp1       ;8 LEDs an PORTC

        ;SPI Slave initialisieren
        ;SPI-Ausgang initialisieren
        ;(PB4(/SS), PB5(MOSI) und PB7(SCK) automatisch Eingang)
        sbi     DDRB,6          ;PB6 Ausgang (MISO)
        ;SPI einschalten, Slave mit SCK = Systemtakt/128, MSB first, SPI Mode 1
        ldi     r16,0xC7        ;SPCR = 0b11000111
        out     SPCR,r16        ;B7 SPIE=0 kein Interrupt
                                ;B6 SPE=1 SPI einschalten
                                ;B5 DORD=0 MSB zuerst
                                ;B4 MSTR=0 Slave
                                ;B32 CPOL=0 CPHA=1 SPI Modus 1
                                ;B10 SPR1=0 SPR0=1 Systemtakt/128 (125kHz)

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   sei                     ;Interrupts Global erlauben
END:    rjmp    END             ;Endlosschleife

;-----------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-----------------------------------------------------------------------------
;Interruptroutine zum Senden und Empfangen über den SPI Bus
ISRSPI: in      r16,PINA        ;Schalter einlesen und
        out     SPDR,r16        ;ueber SPI uebertragen
        in      r16,SPDR        ;Empfangene Daten auf LEDs ausgeben
        out     PORTC,r16
        reti                    ;Rücksprung ins Hauptprogramm
```

⌂ **D201** Teste die Übertragung mit den beiden vorgegebenen Programme[9]. Schreibe die Programme so um, dass die Initialisierung jeweils in einem Unterprogramm erfolgt.

Nenne die neuen Programme "**D201_SPI_master_interrupt.asm**" und "**D201_SPI_slave_interrupt.asm**".

9 Download auf www.weigu.lu/a/asm

Weitere Aufgaben

⌂ **D202** Im SRAM eines Slave befinden sich 10 Werte eines Sensors. Ein Master aktiviert den Slave, um die 10 Werte abzufragen (Polling). Dabei wird für jedes Byte die /SS-Leitung aktiviert. Der Master speichert die 10 Werte im SRAM und gibt sie dann auf dem Display aus. Der Slave übermittelt die 10 Werte mit Hilfe einer Interruptroutine.

Nenne die neuen Programme "**4D22_SPI_master_polling_10.asm**" und "**D202_SPI_slave_interrupt_10.asm**".

⌂ **C203** Die folgende Aufgabe stammt aus dem Lehrbuch von Roland Walter (www.rowalt.de). Das Brenngerät (Programmer) dient in dieser Aufgabe als Master. Damit unser Slave nicht umprogrammiert wird, sondern die Daten über SPI richtig empfängt, muss die Reset Leitung des Programmiergerätes mit dem /SS-Pin statt dem Reset-Pin des AVR verbunden werden. Bastle dazu einen Adapter. Die Befehle des Programmiergerätes können dann eingelesen werden und über die serielle Schnittstelle an einen PC übermittelt werden. Weitere Informationen zu den Befehlen findet man im Datenblatt des ATmega32A (bzw. ATmega8A) unter *Memory Programmimg*, *Serial Downloading*, *SPI Serial Programmimg Instruction Set.* Teste das Programm und finde heraus, welche Befehle das Programmiergerät sendet. Nenne das neue Programm "**D203_SPI_slave_interrupt_programmer.asm**".

⌂ **C204** Die folgende Aufgabe stammt ebenfalls aus dem Lehrbuch von Roland Walter (www.rowalt.de). Es soll ein Master so programmiert werden, dass er einen beliebigen anderen Controller als Brenngerät (Programmer) anspricht. Der Slave wird dabei vom Master mit Spannung versorgt. Seine Reset-Leitung wird über einen Ausgang (z.B. PB4,/SS) aktiviert. Die Befehle für das Programmiergerät findet man im Datenblatt des ATmega32A (bzw. ATmega8A) unter *Memory Programmimg*, *Serial Downloading*, *SPI Serial Programmimg Instruction Set.* Das Programm soll den Slave in den Programmiermodus versetzen, dann die drei Signatur-Bytes auslesen und dann den Programmiermodus wieder beenden. Nenne das Programm "**D204_SPI_master_polling_programmer.asm**".

D3 Die USB-Schnittstelle

In diesem Kapitel soll die Kommunikation über USB möglichst einfach dargestellt werden. Es wird ebenfalls eine kleine USB-Bibliothek für ATMEL®-USB-AVRs® vorgestellt. Die Bibliothek verwendet keine Standardklassen. Sie arbeitet auf der PC-Seite mit der freien „libusb“. Die Software und weitere Erklärungen findet man unter www.weigu.lu/usb.

Kurze Einführung zu USB

USB-Transfer

Die USB-Übertragung zwischen PC (*host*) und Gerät (*device*, *function*) besteht aus mehreren Protokollschichten. Die gesamte Übertragung (Kommunikationsanforderung herstellen und ausführen) wird als **USB-Transfer** bezeichnet. Es existieren **vier** unterschiedliche **Transfertypen**[10]:

- **Control**-Transfer:
  zur Identifikation und Steuerung des Gerätes und für herstellerspezifische Anforderungen (muss von jedem USB-Gerät unterstützt werden und arbeitet immer mit Endpunkt 0 (siehe weiter unten))
- **Bulk**-Transfer:
  zur Übertragung großer Datenmengen ohne garantierte Geschwindigkeit (Bsp.: Laufwerk)
- **Interrupt**-Transfer:
  mit garantierter Bandbreite für geringes Datenvolumen (Bsp.: Tastatur)
- **Isochron**-Transfer:
  mit garantierter Bandbreite aber ohne Fehlerkorrektur (Bsp.: Streaming von Audiodaten)

Ein Transfer besteht aus einer oder mehreren **Transaktionen** (Übergabe eines Dienstes an einen Endpunkt). Es gibt drei Arten von Transaktionen[11]:

- **SETUP (Control)**-Transaktion: bidirektionale Nachrichten
- **IN**-Transaktion: Daten vom Gerät zum PC
- **OUT**-Transaktion: Daten vom PC zum Gerät

Jede Transaktion muss ohne Unterbrechung abgeschlossen werden. Jede Transaktion besteht aus drei Paketen (Phasen)[12]:

- **TOKEN**-Paket:
  gibt an um welchen Transaktionstyp es sich handelt (SETUP, IN, OUT)

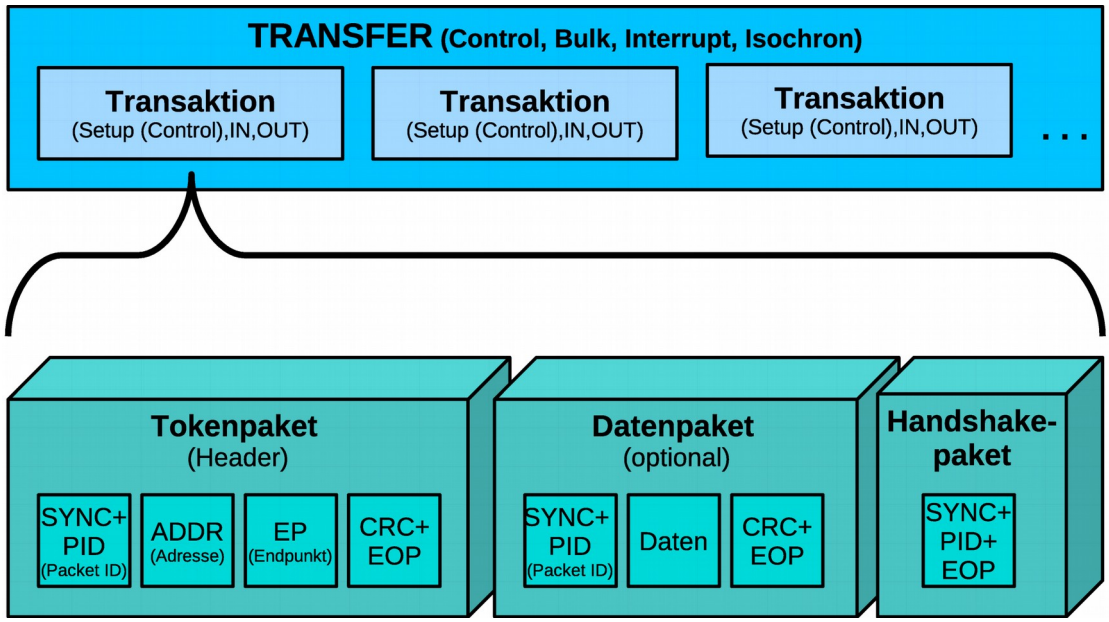
10 Control-Transfers nutzen bidirektionale Nachrichten-Kanäle (Pipes). Die anderen Transfers nutzen unidirektionale Datenstrom-Kanäle.

11 Zur Zeitsteuerung wird auch ein Start-Of-Frame-Transfer bzw. Transaktion durchgeführt. Ebenfalls existiert also ein SOF-Token. Es überträgt eine Zeitreferent (1ms oder 125µs).

12 Es existiert auch eine vierte "Special"-Phase. Auf sie soll hier nicht eingegangen werden.

- (optionale) **DATA**-Paket:
  enthält Daten oder Statusinformationen (DATA0-2, MDATA)
  Speziell: Beim erfolgreichen Abschließen eines Control-Transfers wird als Status-Information (siehe Handshake) ein Datenpaket ohne Daten (**ZLP**, **Z**ero **L**ength data **P**acket) anstelle einer Bestätigung (ACK) versendet.
- **STATUS** (Handshake)-Paket:
  Feedback zur Kommunikation: "Erfolg", "bin beschäftigt (warten)" und "nicht unterstützt" (ACK, NAK, STALL (NYET)).

Die Paketzusammenstellung wird von der Hardware bewerkstelligt. Auf sie soll nicht weiter eingegangen werden.



Endpunkte

Der Datenaustausch passiert zwischen dem PC und einem spezifischen Geräteendpunkt (*endpoint* EP). Ein Endpunkt ist ein Datenspeicher (FIFO[13], Puffer, Speicherbank) im Gerät, der meist nur aus 8-256 Bytes besteht. Jedes Gerät hat mehrere Endpunkte die über die Endpunktadresse (015) angesprochen werden. Diese Adresse enthält in Bit 7 ebenfalls die Richtung der Datenkommunikation[14]. Der Control-Endpunkt 0 ist in jedem Gerät vorhanden und wird für zum Beispiel für die Enumeration benötigt. Er ist als einziger Endpunkt bidirektional. Bei USB 1.1 hat er einen 8 Byte großen FIFO-Speicher. Die Anzahl und mögliche Größe der anderen Endpunkte variiert. Jede Transaktion ist an eine Geräte und eine Endpunktadresse gebunden.

13 FIFO (*First In First Out*): Speicherbank, welche zuerst abgelegte Daten beim Lesen auch als erstes wieder ausgibt, im Gegensatz zum LIFO (*Last In First Out*), der zum Beispiel beim AVR Stapel verwendet wird.

14 Außer dem Control-Endpunkt, da dieser bidirektional; Bit 7 = 0: OUT-Endpunkt; Bit 7 = 1: IN-Endpunkt

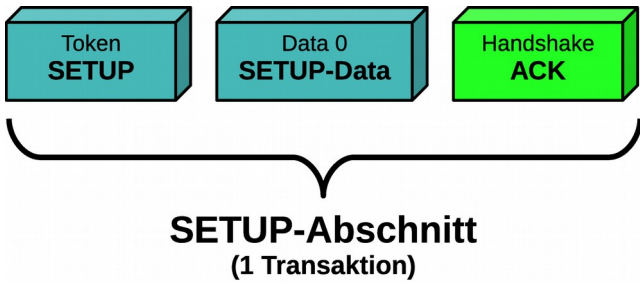
Control Transfer

Die gesamte Enumeration der Gerätes erfolgt mittels Control-Transfers (Standard Anfragen). Control-Transfers können auch für herstellerspezifische Anforderungen genutzt werden. Der Control-Transfer besteht aus drei Abschnitten (*stages*):

6. Der **SETUP-Abschnitt** besteht aus einer Transaktion.
7. Der **DATEN-Abschnitt** kann wegfallen. Er besteht aus keiner, einer oder mehreren Transaktionen.
8. Der **STATUS-Abschnitt (Handshake)** besteht aus einer Transaktion.

Der SETUP-Abschnitt (1 Transaktion)

Die Anfrage (request) erfolgt in der Setup-Token. Der PC sendet das Setup-Token und das Setup-Paket (Daten-Paket). Das Gerät antwortet mit ACK.



Alle Informationen zur Anfrage befinden sich im 8 Byte großen Setup-Paket[15]:

Feld	Bytes
bmRequestType	1
bRequest	1
wValue	2
wIndex	2
wLength	2

bmRequestType

Bit	7	6	5	4	3	2	1	0
	DIR	TYPE1	TYPE0	REC4	REC3	REC2	REC1	REC0

DIR ***Data Phase Transfer Direction*** **0 PC zum Gerät (OUT)**
1 Gerät zum PC (IN)

TYPE ***TYPE1, TYPE0*** **00 Standard**

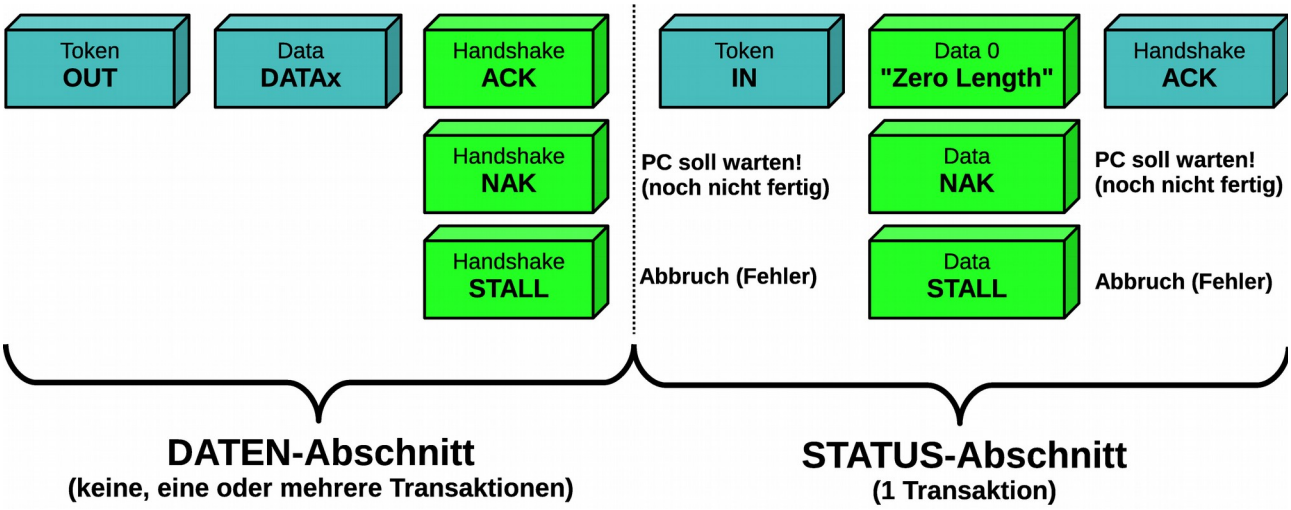
15 Benutzte Präfixe:
b = Byte, w = Word, bm = Bitmap, bcd = binaercodierte Dezimalzahl, i = Index, id = Kennung

	01	Klasse
	10	Vendor (herstellerspezifische Anfrage)
	11	Reserviert
REC *Recipient REC1, REC0*	**00**	**Device**
	01	**Interface**
	10	**Endpoint**
	11	otherReserviert

Der DATEN- und der STATUS-Abschnitt (Handshake)

Der schreibende Control Transfer (Data OUT)

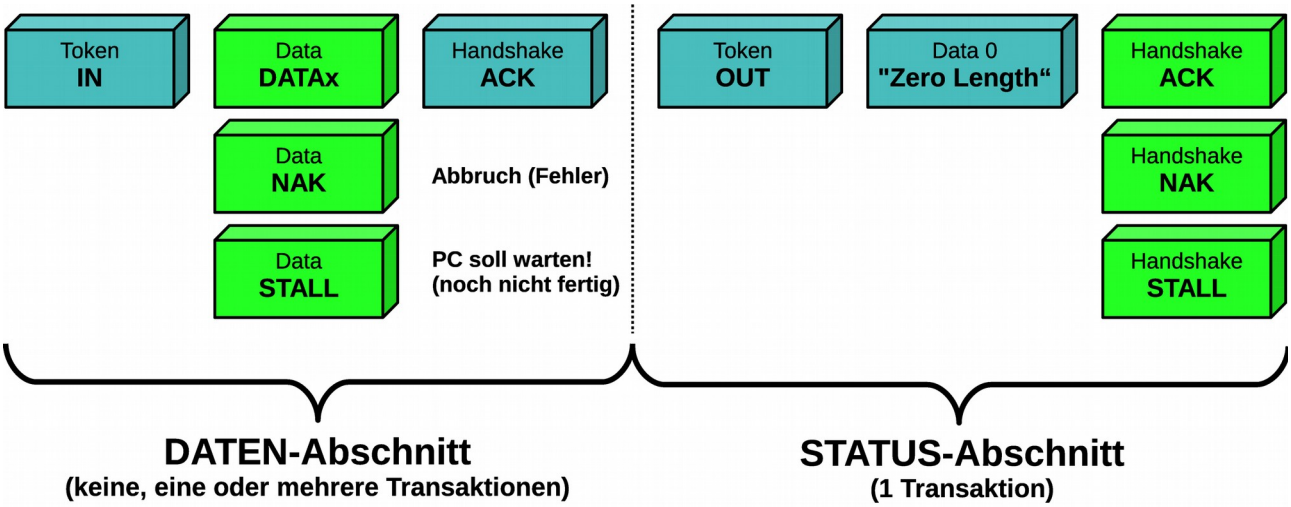
Bei diesem Transfer teilt der PC dem Gerät im Setup-Abschnitt mit, dass er Daten zum Gerät senden will. Bei der Datenphase gibt es nach dem Empfang der Daten dann drei Möglichkeiten. Das Gerät kann den Empfang bestätigen (ACK), den PC auffordern zu warten (NAK) oder Abbrechen (STALL). Tritt beim Token oder Datenpaket ein Fehler auf, so wird das Paket ignoriert. Der DATEN-Abschnitt kann aus mehreren Transaktionen bestehen!



War der Transfer erfolgreich, so sendet das Gerät in der Handshake-Phase ein "Zero Length"-Paket an den PC zurück. Der PC antwortet mit ACK. Trat ein Fehler beim Endpunkt auf, so sendet das Gerät einen STALL. Ist es noch beschäftigt, so sendet es ein NAK. Tritt sonst ein Fehler auf, so wird das Paket ignoriert.

Der lesende Control Transfer (Data IN)

Bei diesem Transfer teilt der PC dem Gerät im SETUP-Abschnitt mit, dass er Daten vom Gerät lesen möchte. Hier gibt es ebenfalls drei Möglichkeiten. Das Gerät kann den IN Token annehmen und die Daten bereitstellen. Der PC antwortet mit ACK. Ist das Gerät noch beschäftigt, so sendet es ein NAK. Trat ein Fehler beim Endpunkt auf, so sendet es einen STALL. Tritt sonst ein Fehler auf, so wird das Paket ignoriert.



In der Handshake-Phase meldet der PC mit einem "Zero Length"-Paket ob er die Daten erfolgreich erhalten hat. Das Gerät antwortet mit ACK. Fehler oder ein beschäftigtes Gerät geben STALL bzw. NAK zurück. Bei sonstigen Fehlern wird das Paket ignoriert.

Die Enumeration

Der Vorgang bei dem der PC dem Gerät eine Adresse zuordnet, alle Informationen über das Gerät erfragt, den richtigen Treiber lädt und dann eine Konfiguration auswählt, wird **Enumeration** genannt.

Mittels Standard-Anfragen (*standard request*) wird vom PC die Adresse des Gerätes festgelegt und mehrere sogenannte Deskriptoren (Beschreibungen) erfragt. Jedes Gerät muss mindestens 4 Deskriptoren liefern:

- **Geräte-Deskriptor (d*evice descriptor*)**
- **Konfigurations-Deskriptor (*configuration descriptor*)**
- **Schnittstellen-Deskriptor (*interface descriptor)***
- **Endpunkt-Deskriptor (*endpoint descriptor)***

Es sollen hier nur die aller nötigsten Schritte einer Enumeration vereinfacht beschrieben werden. Wir gehen dabei von nur einer Konfiguration und einem Interface ohne alternative Einstellungen aus.

Nachdem das Gerät an den Bus angeschlossen wurde, ermittelt der PC anhand der Spannungen der beiden Signalleitungen ob ein Low- oder Full-Speed angeschlossen wurde. Der PC resetet dann das Gerät (beide Datenleitungen auf Low) und stellt dabei fest ob es sich um ein Highspeed-Gerät handelt. Nach dem Reset des Gerätes ist dies bereit über Endpunkt 0 auf der Adresse 0 angesprochen zu werden.

Die Standard Anfragen werden dann mittels SETUP-Paketen (siehe Control Transfer) durchgeführt.

Das erste SETUP-Paket (**Get_Deskriptor**) des PC erfragt (Adresse Null, Endpunkt 0) 64 Byte des Geräte-Deskriptor um die maximale Paketgröße von Endpunkt 0 zu ermitteln[16]. Nach 8 Byte (hier befindet sich die bMaxPacketSize) bricht der PC ab und führt ein neues Reset des Gerätes aus.

Mit dem zweiten SETUP-Paket (**Set_Adress**) sendet der PC eine Geräte-Adresse. Diese wird von der Firmware dem Gerät zugewiesen. Das dritte SETUP-Paket (Get_Deskriptor) erfragt die 18 Byte des Geräte-Deskriptors. Dann werden mit einem vierten SETUP-Paket (Get_Deskriptor) die 9 Byte des Konfigurations-Deskriptors erfragt. Dieses vermittelt die Gesamtlänge des Konfigurations-, Interface- und aller Endpunkt-Deskriptoren. Ein fünftes SETUP-Paket (Get_Deskriptor) erfragt all diese Deskriptoren in einer Aktion. Weitere SETUP-Pakete erfragen optionale Deskriptoren (z.B. String-Deskriptoren).

Der PC lädt jetzt anhand von Vendor-ID und Product-ID und der ".INF" Datei den entsprechenden Gerätetreiber.
Ein letztes SETUP-Paket der Emuneration (**Set_Configuration**) bewirkt das Initialisieren und Aktivieren der User-Endpunkte (Endpunkt1-15). Andere, von einer Firmware nicht unterstützte Anfragen des PCs mittels SETUP-Paketen, werden einfach mit STALL abgewiesen.

Der Geräte-Deskriptor

Es gibt pro Gerät nur einen Geräte-Deskriptor[17].
Hier der Geräte-Deskriptor mit den Werten für unsere minimale Firmware:

Feldbezeichnung	Byte	Wert	Beschreibung
bLength	1	18 (0x12)	Größe des Deskriptors in Byte
bDescriptorType	1	1	Geräte Deskriptor = 1 (Konstante)
bcdUSB	2	0x0110	USB_Spec1_1
bDeviceClass	1	0xFF	Klassencode (hier anbieterspezifisch = 0xFF)
bDeviceSubClass	1	0xFF	Unterklassencode (hier anbietersp. = 0xFF)
bDeviceProtocoll	1	0xFF	Protokollcode (hier anbieterspezifisch = 0xFF)
bMaxPacketSize	1	8	max. Paketgröße Endpunkt 0 (EP0_FS)
idVendor	2	0x03eb	Atmel Code durch usb.org vergeben
idProduct	2	0x0001	Produkt ID beliebig
bcdDevice	2	0x0001	Release Nummer Gerät
iManufacturer	1	1	Index für String-Deskriptor Hersteller
iProduct	1	2	Index für String-Deskriptor Produkt
iSerialNumber	1	3	Index für String-Deskriptor Seriennummer
bNumConfigurations	1	1	Anzahl möglicher Konfigurationen

Der Konfigurations-Deskriptor

Ein Gerät kann mehrere Konfigurationen besitzen (bNumConfigurations). Ein Gerät kann zum Beispiel eine Konfiguration für die Stromversorgung per Bus besitzen und eine zweite

16 Diese kann 8, 16, 32, 64 Byte betragen.

17 Außer bei Verbundgeräten (*composite device*), die über die Schnittstellen-Deskriptoren statt über den Geräte-Deskriptor beschrieben werden.

Konfiguration für eine eigenständige Versorgung. Der Gerätetreiber wählt dann die entsprechende Konfiguration aus. Es ist immer nur eine Konfiguration aktiv.

Im Feld **wTotalLength** wird dem PC die Gesamtzahl der Bytes aller Konfigurations-, Schnittstellen- und Endpunkt-Deskriptoren mitgeteilt. In unserem Fall sind das 9 Byte (1 Konfigurations-Deskriptor) + 9 Byte (1 Schnittstellen-Deskriptor) + 3*7 Byte (3 Endpunkte neben Endpunkt 0) = 39 Byte.

Feldbezeichnung	Byte	Wert	Beschreibung
bLength	1	9	Größe des Deskriptors in Byte
bDescriptorType	1	2	Konfigurations-Deskriptor = 2 (Konstante)
wTotalLength	2	39 (0x27)	Länge des Konfigurations-Deskriptors und aller untergeordneter Deskriptoren
bNumInterfaces	1	1	Anzahl der Schnittstellen
bConfigurationValue	1	1	Nummer um diese Konfiguration auszu-wählen (darf nicht Null sein, sonst geht Gerät in den nicht-konfigurierten Zustand)
iConfiguration	1	0	Index für String-Deskriptor dieser Konfiguration (0 = kein Text)
bmAttributes	1	0x80	D7 = 1 Versorgung durch Bus, D6 = 1 Selbstversorgung, D5 = 1 Remote Wakeup
bMaxPower	1	50	Max. Strombezug vom Bus in 2mA Schritten

Der Schnittstellen-Deskriptor

Der Schnittstellen-Deskriptor bündelt mehrere Endpunkte zu einer Funktionsgruppe. Es können mehrere Schnittstellen gleichzeitig aktiv sein (Beispiel, Fax-Schnittstelle, Druck-Schnittstelle und Scan-Schnittstelle bei Multifunktionsdrucker). Eine Schnittstelle kann mit der gleichen Schnittstellennummer mehrere alternative Einstellungen beherbergen, welche allerdings nicht gleichzeitig aktiviert werden können. Ein schnelles Umschalten zwischen den alternativen Einstellungen ist möglich.

Feldbezeichnung	Byte	Wert	Beschreibung
bLength	1	9	Größe des Deskriptors in Byte
bDescriptorType	1	4	Schnittstellen-Deskriptor = 4 (Konstante)
bInterfaceNumber	1	0	Anzahl der Schnittstellen
bAlternateSetting	1	0	Nummer um alternative Einstellungen zu wählen
bNumEndpoints	1	3	Anzahl der Endpunkte außer Endpunkt 0
bInterfaceClass	1	0xFF	Klassencode (hier anbieterspezifisch = 0xFF)
bInterfaceSubClass	1	0xFF	Unterklassencode (hier anbietersp. = 0xFF)
bInterfaceProtocol	1	0xFF	Protokollcode (hier anbieterspezifisch = 0xFF)
iInterface	1	0	Index für String-Deskriptor dieser Schnittstelle (0 = kein Text)

Der Endpunkt-Deskriptor

Die Endpunkt-Deskriptoren beschreiben die Endpunkt (außer Endpunkt 0). Wichtig ist, dass bei der Endpunktadresse auch die Richtung mittels Bit 7 angegeben werden muss.

Hier der Deskriptor für Endpunkt 1 (IN, Bulk, FIFO 8 Byte)

Feldbezeichnung	Byte	Wert	Beschreibung
bLength	1	7	Größe des Deskriptors in Byte
bDescriptorType	1	5	Schnittstellen-Deskriptor = 5 (Konstante)
bEndpointAddress	1	0x81	Bit 7 = 1 (IN), Bit 0-3 Endpunkt-Nummer (andere 0)
bmAttributes	1	2	Transfertyp = Bulk (contr. = 0, iso. = 1, int. = 3)
wMaxPacketSize	2	8	FIFO Größe des Endpunkts in Byte
bInterval	1	0	Polling Interval = 0 (ignoriert für Bulk und Control), 1 für Iso, 1-255 für Interrupt

Die String-Deskriptoren

Sie sind nicht unbedingt nötig, liefern aber zusätzliche lesbare Informationen. Wird ein String-Deskriptor nicht benötigt, so wird sein Index auf 0 gesetzt.

Strings sind in Unicode (16 Bit) kodiert. Es werden viele unterschiedliche Sprachen unterstützt. Im String-Deskriptor mit dem Index 0 werden die unterstützten Sprachen festgelegt. Hier als Beispiel eine Unterstützung für Englisch und Deutsch:

Feldbezeichnung	Byte	Wert	Beschreibung
bLength	1	6	Größe des Deskriptors in Byte
bDescriptorType	1	3	String-Deskriptor = 3 (Konstante)
wLANGID[0]	2	0x0409	English USA (Standard)
wLANGID[1]	2	0x0407	Deutsch Standard

Als nächster Deskriptor folgt dann der Deskriptor mit Index Eins (hier Hersteller String-Deskriptor)

Feldbezeichnung	Byte	Wert	Beschreibung
bLength	1	18	Größe des Deskriptors in Byte
bDescriptorType	1	3	String-Deskriptor = 3 (Konstante)
bString	16	0x0057,0x0045,.....	"W","E","I","G","U",".","L","U"

Firmware AT90USBKEY

Kurze Beschreibung der Firmware

Der AT90USBKEY ist eine billige kleine Entwicklungsplatine von ATMEL mit einem AT90USB1287-Controller.

Die Firmware besteht aus 2 Dateien (Hauptprogramm, USB-Bibliothek).

Es wird nur eine Konfiguration und ein Interface verwendet. Es stehen neben dem zwingend vorgeschriebenem Endpunkt Null (EP0) noch zwei weitere Endpunkte zur Verfügung. Ein OUT-Endpunkt (EP1) erlaubt dem PC Daten zum Gerät zu senden. Ein IN-Endpunkt (EP2) ermöglicht es dem PC Daten vom Gerät zu lesen.

Hauptprogramm:

Eine blinkende LED zeigt, dass die Firmware läuft. Mit einem Flag kann das Blinken abgestellt werden.

USB-Bibliothek

Treiberfunktionen:

Nach dem Anstecken führt der PC (Host) ein Reset das Gerät (device) aus. Der USB-Teil und die interne PLL des AT90USB1287 werden eingeschaltet und initialisiert. Tritt dann ein End of Reset Interrupt (**`EORSTI`**) auf, so kann der bei jedem USB-Gerät vorhandene bidirektionelle Control Endpunkt 0 (EP0) initialisiert und aktiviert werden.
Als nächstes sendet der PC ein SETUP-Paket an Endpunkt 0, das durch ein Received SETUP Interrupt (**`RXSTPI`**) erkannt wird.

Enumeration:

Standard Requests werden vom PC mittels SETUP-Paketen erfragt.
Das erste SETUP-Paket (**Get_Deskriptor**) des PC erfragt (Adresse Null, Endpunkt 0) 64 Byte des Geräte-Deskriptor um die maximale Paketgröße von Endpunkt 0 zu ermitteln. Nach 8 Byte (hier befindet sich die bMaxPacketSize) bricht der PC ab und führt ein neues Reset des Gerätes aus.
Mit dem folgenden SETUP-Paket (**Set_Adress**) sendet der PC eine Geräte-Adresse. Diese wird von der Firmware dem Gerät zugewiesen. Das dritte SETUP-Paket erfragt die 18 Byte des Geräte-Deskriptors. Dann werden mit einem vierten SETUP-Paket die 9 Byte des Konfigurations-Deskriptors erfragt. Dieses vermittelt die Gesamtlänge des Konfigurations-, Interface- und aller Endpunkt-Deskriptoren. Ein fünftes SETUP-Paket erfragt all diese Deskriptoren (hier 5) in einer Aktion. Weitere SETUP-Pakete erfragen erfragen die 4 String-Deskriptoren.
Der PC kann jetzt anhand von Vendor-ID und Product-ID und der ".INF" Datei den entsprechenden Gerätetreiber laden.
Ein letztes SETUP-Paket der Emuneration (**Set_Configuration**) bewirkt das Initialisieren und Aktivieren der zwei User-Endpunkte.

Die Firmware antwortet auch noch auf Statusanfragen (**Get_Status**).[18] Andere, von dieser Firmware nicht unterstützte Anfragen des PC mittels SETUP-Paket werden mit STALL abgewiesen.

18 Das Behandeln dieser Anfrage besteht nur um unschöne Fehlermeldungen bei **`lsusb -v`** unter Linux zu vermeiden (kann bei Bedarf gelöscht werden).

Anwendungskommunikation:

Folgende Aktionen können nun von der PC-Software gestartet werden:

a) Ein- und Ausschalten einzelner Bits (4 LEDs an PD4-PD7 und das Blink-Flag PD0) eines Ports (1 Byte) über Endpunkt 1 (OUT-Aktion (PC ⇨ Gerät)).
b) Analoge Daten (Analog-Digitalwandler) über den FIFO des Endpunkt 2 einlesen (IN-Aktion (Gerät ⇨ PC)).

Zur Kommunikation mit den Endpunkten:

Control Endpunkt 0

SETUP-Abschnitt:

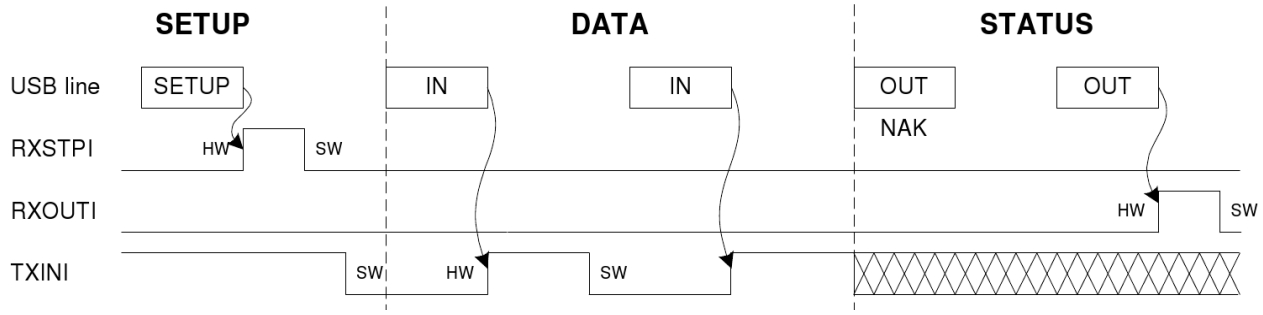
Das **R**eceived **SETUP I**nterrupt (**RXSTPI**) Flag wird gesetzt wenn ein SETUP-Paket eintrifft (SETUP-Abschnitt). In der Firmware wird ein Interrupt ausgelöst. In der Interruptroutine wird ein Unterprogramm zur Behandlung des SETUP-Pakets aufgerufen. In diesem Unterprogrsamm löscht die Firmware mit **CBI(UEINTX,RXSTPI)**[19] das Interrupt-Flag um das SETUP-Paket zu bestätigen (ACK) und den FIFO-Puffer des Endpunktes zu löschen.

DATA IN und STATUS-Abschnitt:

Das **TXINI**-Flag (Transmitter Ready Interrupt-Flag) zeigt, dass der FIFO-Puffer frei ist und vom Controller gefüllt werden kann. Nachdem dies geschehen ist wird mit **CBI(UEINTX,TXINI)** das bis zu 8 Byte große Paket (FIFO-Größe EP0) abgeschickt und der FIFO wird wieder gelöscht, so dass er neue Daten aufnehmen kann. Wurden mehr als 8 Byte angefragt, so können mehrere Pakete vor dem Status-Abschnitt gesendet werden. Dazu muss jedes Mal überprüft werden ob der FIFO-Speicher leer ist (**TXINI** = 1).

Das erste OUT-Paket vom PC wird von der Hardware automatisch mit NAK (PC soll warten) beantwortet. Das nächste OUT-Paket setzt das **RXOUTI**-Flag (**R**eceived **OUT** Data **I**nterrupt). Das Flag teilt mit, wann das Zero-Length-Paket vom PC angekommen ist. Per Polling wird auf das ZLP gewartet und dann das Flag wieder gelöscht **CBI(UEINTX,RXOUTI)**.

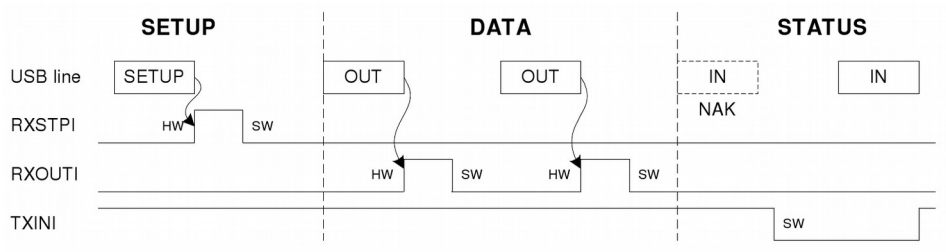
19 **cbi (UEINTX,RXSTPI)** steht für lösche Bit **RXSTPI** im SF-Register **UEINTX**.
cbi uns sbi (setze Bit) existieren so nur für die untersten 32 SF-Register. Alle USB Register befinden sich im erweiterten Bereich und müssen mittels direkter SRAM Adressierung und Maskierung angesprochen werden.



Quelle: Datenblatt AT90USB128

DATA OUT und STATUS-Abschnitt:

Soll der PC Daten über den Daten-Abschnitt des Endpunkt 0 an das Gerät senden (was in der Firmware nicht vorkommt), so kann der Datenfluss ebenfalls mit **TXINI** und **RXOUTI** gesteuert werden.



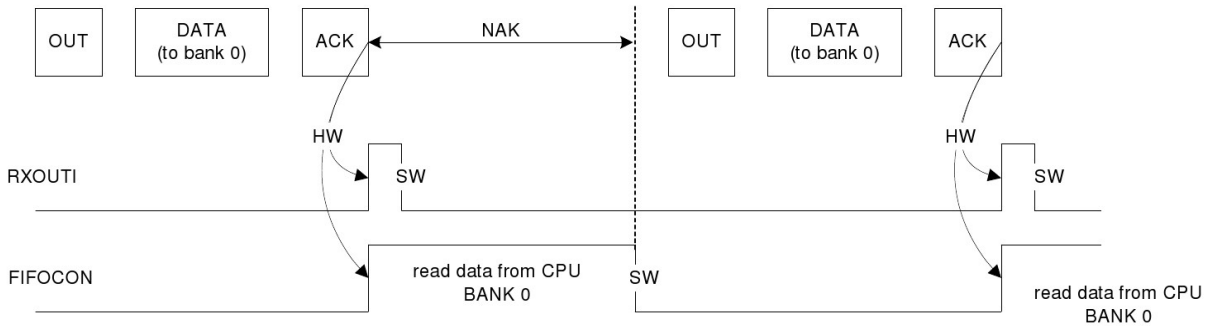
Quelle: Datenblatt AT90USB128

Ein Spezialfall tritt auf wenn keine Daten vorliegen wie bei der **Get_Adress** Anfrage. Es gibt also neben dem SETUP- nur ein STATUS-Abschnitt. Mit **CBI(UEINTX,TXINI)** wird im Statuspaket ein ZLP versendet. Dann wird gewartet bis der Speicher wieder frei ist (**TXINI** = 1, ACK vom PC).

OUT Endpunkt

Das **R**eceived **OUT** Data **I**nterrupt (**RXOUTI**) Flag wird gesetzt wenn ein OUT-Datenpaket eintrifft. Dieses wird von der Hardware bestätigt. Gleichzeitig setzt die Hardware das **FIFOCON**-Flag (FIFO CONtrol Bit). In der Firmware wird ein **RXOUT-I**nterrupt ausgelöst. In der Interruptroutine wird ein Unterprogramm zur Behandlung des OUT-Pakets aufgerufen. In diesem Unterprogrsamm löscht die Firmware mit **CBI(UEINTX,RXOUTI)** das Flag um das Interrupt zu bestätigen. Das **RWAL**-Flag im gleichen SF-Register zeigt den Zustand des FIFO. **RWAL** = 1 bedeutet, dass Daten im FIFO vorhanden sind. Sind keine Daten vorhanden, so löscht die Hardware das Flag. Die Firmware überprüft **RWAL** und liest dann die Daten und löscht mit **CBI(UEINTX,FIFOCON)** das **FIFOCON**-Flag um den FIFO-Puffer wieder frei zu geben.

Quelle: Datenblatt AT90USB128



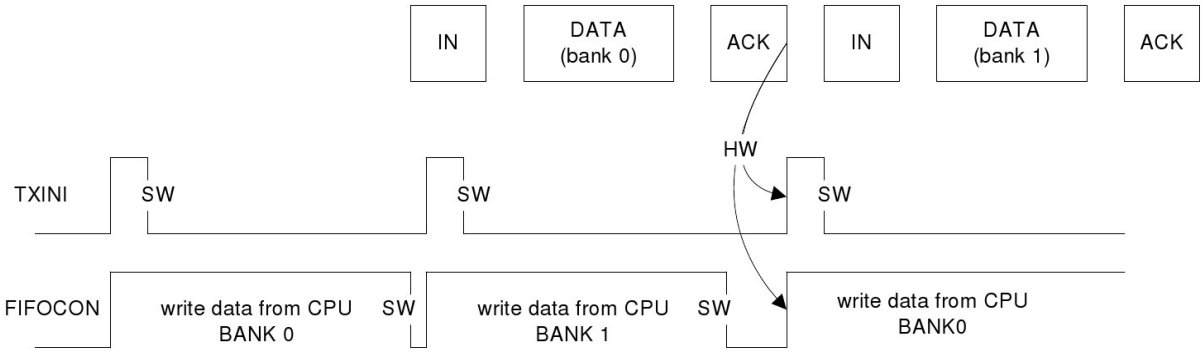
IN Endpunkt

Benötigt der PC ein Paket, so wird dies über das **NAK IN** Received **I**nterrupt (**NAKINI**) Flag gemeldet. In der Firmware wird ein Interrupt ausgelöst. In der Interruptroutine wird ein Unterprogramm zur Behandlung der IN-Anfrage aufgerufen. In diesem Unterprogrsamm löscht die Firmware mit **CBI(UEINTX,NAKINI)** das Interrupt-Flag.

Das **TXINI**-Flag (Transmitter Ready Interrupt-Flag) zeigt, dass der FIFO-Puffer frei ist und vom Controller gefüllt werden kann. Gleichzeitig wird das **FIFOCON**-Flag (FIFO CONtrol Bit) von der Hardware gesetzt.

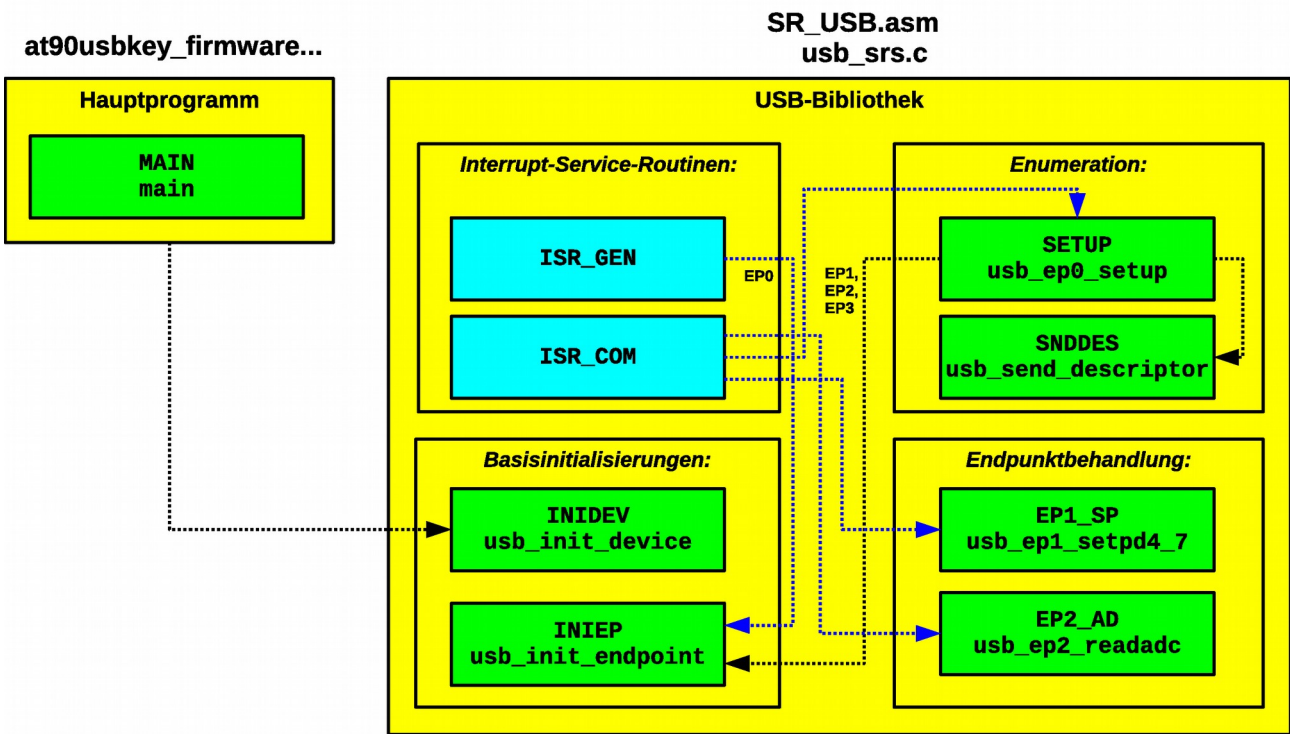
Nachdem dies geschehen ist wird mit **CBI(UEINTX,TXINI)** sofort gelöscht. Der Controller füllt dann den FIFO und erlaubt mit dem Löschen des **FIFOCON**-Flag (**CBI(UEINTX,FIFOCON)**) dem Controller die Daten abzuschicken. Besteht der Endpunkt aus einer Dobbelspeicherbank, so wird automatisch auf die nächste Bank umgeschaltet.

(Das **RWAL**-Flag zeigt auch hier den Zustand des FIFO. **RWAL** = 1 bedeutet, dass der FIFO voll ist.)



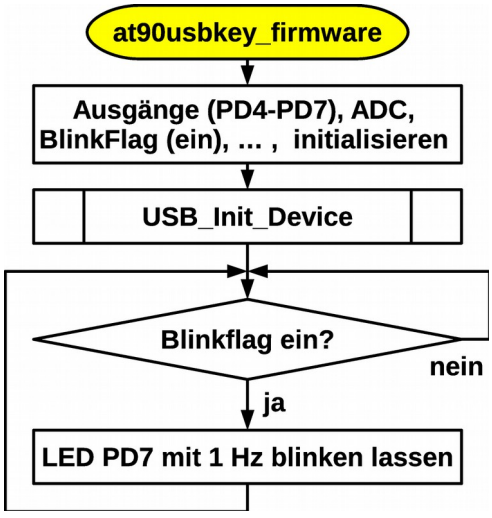
Quelle: Datenblatt AT90USB128

Übersicht zur Firmware



Das Hauptprogramm (at90usbkey_firmware...)

Im Hauptprogramm zeigt eine LED (**PD4**) durch Blinken mit 1 Hz, dass dieses arbeitet. Ein Flag (**BLINKFLAG**) ermöglicht es das Blinken abzuschalten. Dieses Flag kann später über den Endpunkt 1 mit **PD0** ab. bzw. eingeschaltet werden.



Einzige wichtige Aktion im Hauptprogramm ist der Aufruf des Unterprogramms **INIDEV** (**usb_init_device**), das sich in sr_usb.asm (usb_srs.c) befindet.

Im Hauptprogramm wird ebenfalls der AD-Wandler initialisiert. Im Assembler muss auch die Vektortabelle im Hauptprogramm initialisiert werden.

Die USB-Bibliothek (SR_USB.asm, usb_srs.c)

In der USB-Bibliothek werden die Interrupts abgefangen und behandelt, die Basisinitialisierungen des USB-Teils im ATmega vorgenommen (Treiberfunktionen) , die Enumeration wird durchgeführt und die die Endpunkte werden behandelt (Anwenderkommunikation).

Die Interrupt-Service-Routinen

In der USB-Bibliothek befinden sich **zwei Interrupt-Service-Routinen** (Datenblatt S 252ff).

- Einmal wird der **USB GENeral Interrupt Vektor** behandelt. Dieser Interrupt Vektor kann von 21 verschiedenen Interrupts (General, Device und Host) angesprungen werden. Uns interessiert allerdings nur der **E**nd **O**f **R**e**S**e**T** **I**nterrupt (**EORSTI**), welcher das Ende des vom PC auslösten Reset[20] nach dem Anstecken des Gerätes anzeigt (das Gerät hat den Reset Zustand wieder verlassen).
- Die zweite ISR behandelt den **USB Endpoint/Pipe COMmunication Interrupt Vektor (USB COM Interrupt)**. Hier interessieren uns nur drei Interrupts, welche die Endpunkte betreffen[21]. Für **Endpunkt 0** ist das der **R**eceived **S**E**T**U**P** **I**nterrupt (**RXSTPI**), für den bzw. die **OUT-Endpunkte** der **R**eceived **OUT** Data **I**nterrupt (**RXOUTI**) und für einen oder mehrere **IN-Endpunkte** der **NAK IN** Received **I**nterrupt (**NAKINI**).

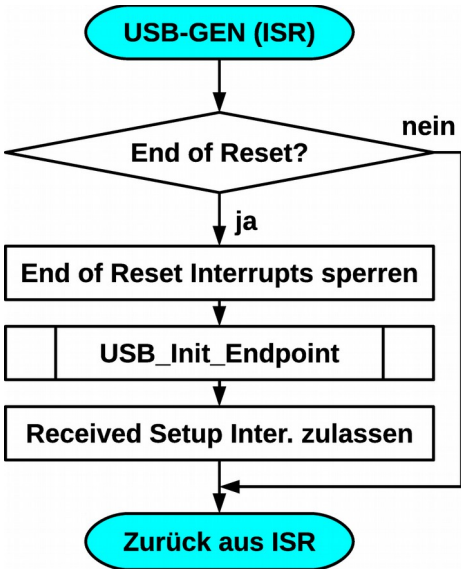
USB_GEN (USB GENeral Interrupt)

In der Routine für den USB General Interrupt (**ISR_GEN**) wird überprüft ob sie von einem **E**nd **O**f **R**e**S**e**T** **I**nterrupt (**EORSTI**), ausgelöst wurde. War ein anderer Interrupt der Auslöser, so wird die Routine ohne Aktion verlassen.
Wurde der Interrupt erkannt, so wird er gesperrt (**CBI**[22] **(UDINT,EORSTI)**), damit er nicht mehr auftreten kann. Der Control **E**nd**P**unkt **0** (**EP0**) wird über das Unterprogramm **INIEP** (**usb_init_endpoint**) initialisiert. Danach wird der Endpoint 0 **R**eceived **S**E**T**U**P** Interrupt freigeschaltet (**SBI(UEIENX,RXSTPE)**), damit eintreffende SETUP-Pakete erkannt werden können.

20 Der PC zieht dazu beide Datenleitungen gleichzeitig auf Null.

21 Da jeder dieser Interrupts an allen sieben Endpunkten auftreten kann ist es wichtig, vor der Behandlung den richtigen Endpunkt mit dem **ENUM** Register auszuwählen.

22 cbi (**UDINT**,**EORSTI**) steht für lösche Bit **EORSTI** im SF-Register **UDINT**.
cbi uns sbi (setze Bit) existieren so nur für die untersten 32 SF-Register. Alle USB Register befinden sich im erweiterten Bereich und müssen mittels direkter SRAM Adressierung und Maskierung angesprochen werden.



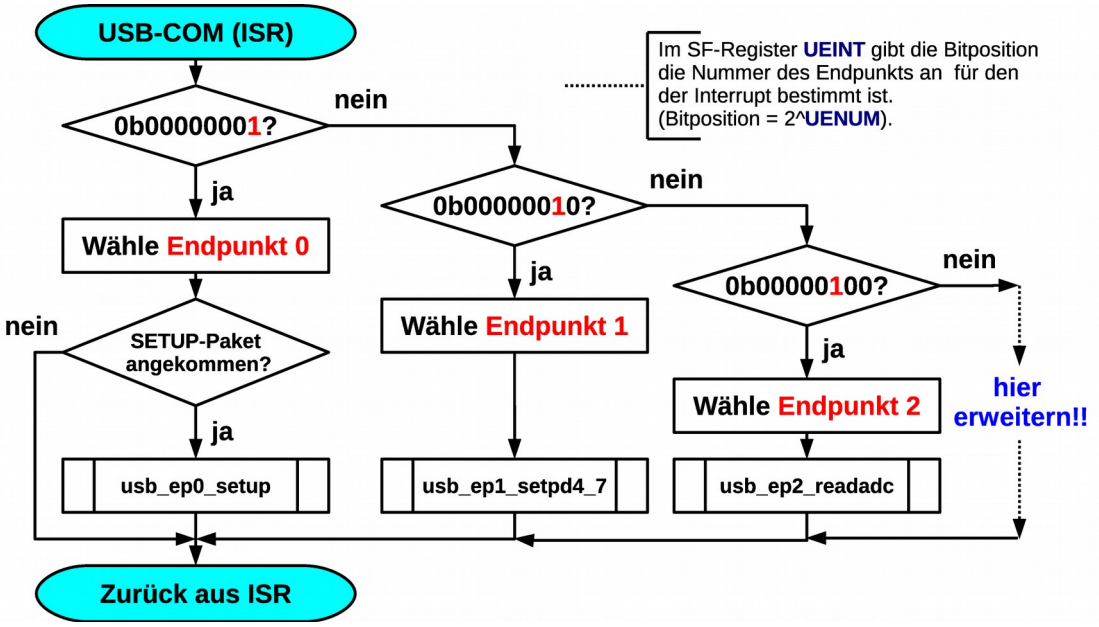
USB_COM (USB Endpoint/Pipe COMmunication Interrupt)

Die Routine für den USB Endpoint/Pipe COMmunication Interrupt (**ISR_COM**) reagiert auf die Endpoint Interrupts. Mit dem SF-Register **UEINT** wird erkannt um welchen Endpunkt es sich handelt. Die Bitposition gibt den Endpunkt an. Mit dem SF-Reg **UENUM** muss dann der jeweilige Endpunkt ausgewählt werden. Je nach Endpunkt wird dann eine entsprechende Unterprogramm aufzurufen.

EP0: Nur falls ein **R**eceived **SETUP** **I**nterrupt (**RXSTPI**) vorliegt wird das Unterprogramm **SETUP** (**usb_ep0_setup**) aufgerufen.

EP1: Ein **R**eceived **OUT** Data **I**nterrupt für Endpunkt 1 bewirkt den Aufruf des Unterprogramms **EP1_SP** (**usb_ep1_setpd4_7**, im Unterprogramm **SETUP** wurde der **RXOUT** **I**nterrupt freigeschaltet).

EP2: Ein **NAK IN** **I**nterrupt für Endpunkt 2 bewirkt den Aufruf des Unterprogramms **EP2_AD** (**usb_ep2_readadc**, im Unterprogramm **SETUP** wurde der **NAKIN** **I**nterrupt freigeschaltet).



Die Unterprogramme

Basisinitialisierungen mit INIDEV und INIEP

INIDEV (usb_init_device)

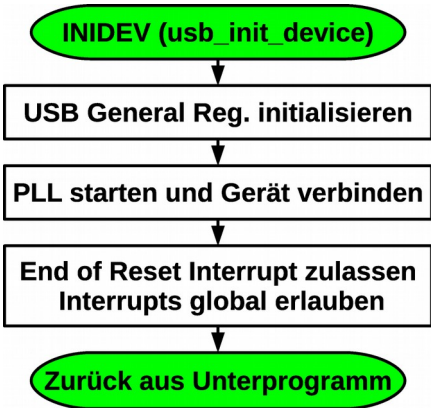
Hier werden einige Basis-Initialisierungen vorgenommen, damit ein **E**nd **O**f **R**e**S**e**T** **I**nterrupt ausgelöst werden kann.

UHWCON (USB HardWare CONfiguration)**= 0x81**
Per Software wird "Device" d.h. Gerät ausgewählt (PIN UID nicht genutzt!) da unser Gerät keine Host-Funktionen (OTG) übernehmen soll.
Bit 7 (**UIMOD**) = 1 wählt Device-Modus, Bit 0 (**UVREGE** = 1)schaltet den Regelkreis zur Versorgung der D+ und D- Leitung (= pads) mit 3-3,6V ein.

USBCON (USB CONfiguration) **= 0xB0**
Bit 7 **USBE** = 1 schaltet USB Sender/Empf. und Takt ein. Bit 5 **FRZCLK** bleibt 1 (default Strom sparen). Bit 4 **OTGPADE** =1 schaltet VBUS-Leitung ein. Muss auch im Device Modus eingeschaltet werden, damit Einstecken und Abstecken vom Bus erkannt wird! Der Takt muss kurzzeitig einschalten werden (**FRZCLK** = 0, **USBCON = 0x90**), damit ein Transfer möglich wird (WAKEUP Interrupt im Flag gespeichert) und somit **EORSTI** auslösen kann (getestet!), kann danach aber gleich wieder abgeschaltet werden um den den Stromverbrauch zu verringern (**USBCON = 0xB0**). **UDIEN = 0x08** (USB Device Interrupt ENable) erlaubt den **E**nd **O**f **R**e**S**e**T** **I**nterrupt.

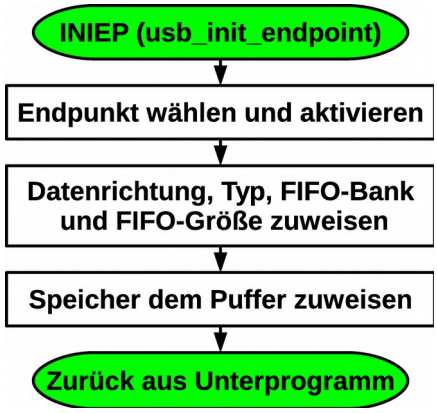
Als nächstes muss die PLL gestartet werden. Die PLL Frequenz wird aus einem 2 MHz Signal durch Multiplikation mit 24 erzeugt. Die 2 MHz werden durch Teilen der Quarzfrequenz erzeugt. Der Vorteiler wird mit den Bits **PLLP0**-**PLLP2** (3 Bit, 2^2-2^4) im Register **PLLCSR** (PLL Control and Status Register) initialisiert (0b011 bei 8 MHz-Quarz (:4)) **PLLCSR = 0x0C**. Nach dem Starten der PLL (**PLLE** = 1, **PLLCSR = 0x0E**)) benötigt die PLL rund 100 ms bevor sie einrastet. Das Flag **PLOCK** im Register **PLLCSR** meldet wenn das Einrasten erfolgt ist. Es wird einfach mittels Polling abgefragt. Mit **FRZCLK** = 0 (**USBCON = 0x90**) wird der Takt aktiviert!

Das Gerät muss mit **UDCON = 0** (**DETACH** = 0) noch physikalisch verbunden werden (verbindet internen Pull-Up mit der D+ Datenleitung (Full Speed)) und Interrupts mit **sei** global erlaubt werden.



INIEP (usb_init_endpoint)

Das UP **INIEP** dient der Initialisierung und Aktivierung der Endpunkte.
Nach der Auswahl des Endpunkts mit **UENUM** wird dieser aktiviert (**SBI(UECONX,EPEN)**).Über die jeweiligen Bits in **UECFG0X** und **UECFG1X** werden Typ, Richtung, FIFO-Größe und die Art des FIFO-Puffers (einzeln oder doppelt) festgelegt. Mit dem **ALLOC**-Bit wird der Speicher dem Puffer zugewiesen (**SBI(UECFG1X,ALLOC)**)[23].



	TYPE	DIR	SIZE (FIFO)	BANK
EP0	Control (0)	IN/OUT (0)	8 Byte (0)	1 (0)
EP1	Bulk (2)	OUT (0)	64 Byte (0)	1 (0)
EP2	Bulk (2)	IN (0)	64 Byte (3)	2 (1)

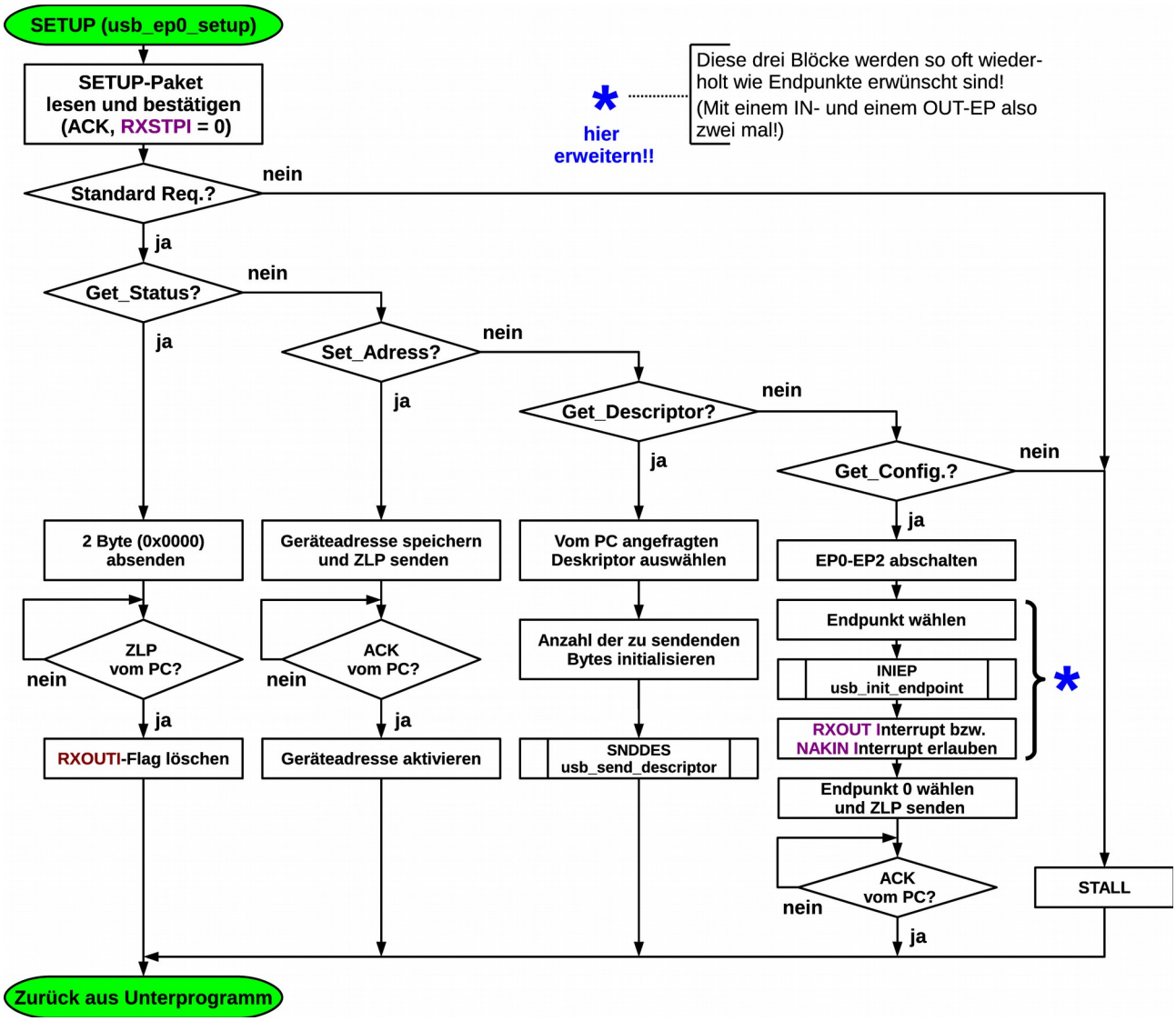
Die Enumeration mit SETUP

Die Enumeration wird durch das Senden von Anfragen (Service Requests) vom PC aus durchgeführt. Der PC sendet SETUP-Pakete mit denen er die Adresse an das Gerät vergibt, Informationen über das Gerät erfragt und die Konfiguration aktiviert. Das Gerät beantwortet die Informationsanfragen mit Deskriptoren. Da jedes Gerät mindestens aus einer Konfiguration und einem Interface bestehen muss werden hier also ein Gerätedeskriptor, ein Konfigurationsdeskriptor, ein Interfacedeskriptor und mehrere Endpunktdeskriptoren erfragt. Zusätzlich können noch Stringdeskriptoren Auskunft über das Gerät oder den Hersteller liefern.

SETUP (usb_ep0_setup)

Dieses Unterprogramm beantworte SETUP-Pakete an Endpunkt 0 für die Enumeration. Das 8 Byte große SETUP-Paket wird eingelesen, im SRAM abgespeichert und mit ACK bestätigt (**RXSTPI** löschen). Liegt ein Standard Request vor, so wird es sich um einen von den vier in unserer Firmware unterstützten Anfragen handelt. Falls nicht antwortet das Gerät mit STALL.

23 Wie aus dem Flussdiagramm zur Aktivierung eines Endpunkt S269 im Datenblatt ersichtlich kann mit dem Bit **CFGOK** im Register **UECFG0X** dann noch überprüft ob die Zuweisung erfolgreich war. Dies wird hier nicht unterstützt.



Get_Status: Dieser Request muss nicht implementiert werden (Zeilen können also gelöscht werden), verhindert aber eine unschöne Fehlermeldung mit „`lsusb -v`“ unter Linux. Es werden zwei leere Byte im FIFO abgelegt (`0x0000`, Flag *remote wakeup* (2^1)= 0), wenn das Gerät vom Bus mit Strom versorgt wird (*bus powered*). Hat es eine eigene Versorgung (*self powered*), so muss `0x0001` ins FIFO geschrieben werden. Mit (`CBI(UEINTX,TXINI)` wird das Paket abgeschickt und das FIFO wieder gelöscht. Danach wird auf die Bestätigung des PC gewartet (ZLP).

Set_Adress: Die Adresse wird ermittelt und dem Gerät zugewiesen. Ein Zero-Length Paket (ZLP) meldet den Erfolg. Wurde das ZLP erfolgreich versendet (ACK vom PC), so wird die Adresse aktiviert.

Get_Descriptor: Es wird überprüft, ob es sich um einen Device, Configuration oder String-Deskriptor handelt. Die zu übermittelnden Deskriptoren befinden sich im Flash. Ein Unterprogramm `SNDDES` (`usb_send_descriptor`) kümmert sich um das Versenden der Deskriptoren. Dem UP wird in `r18` die Länge des Deskriptors in Byte und in `Z` der Zeiger auf den Deskriptor übergeben.

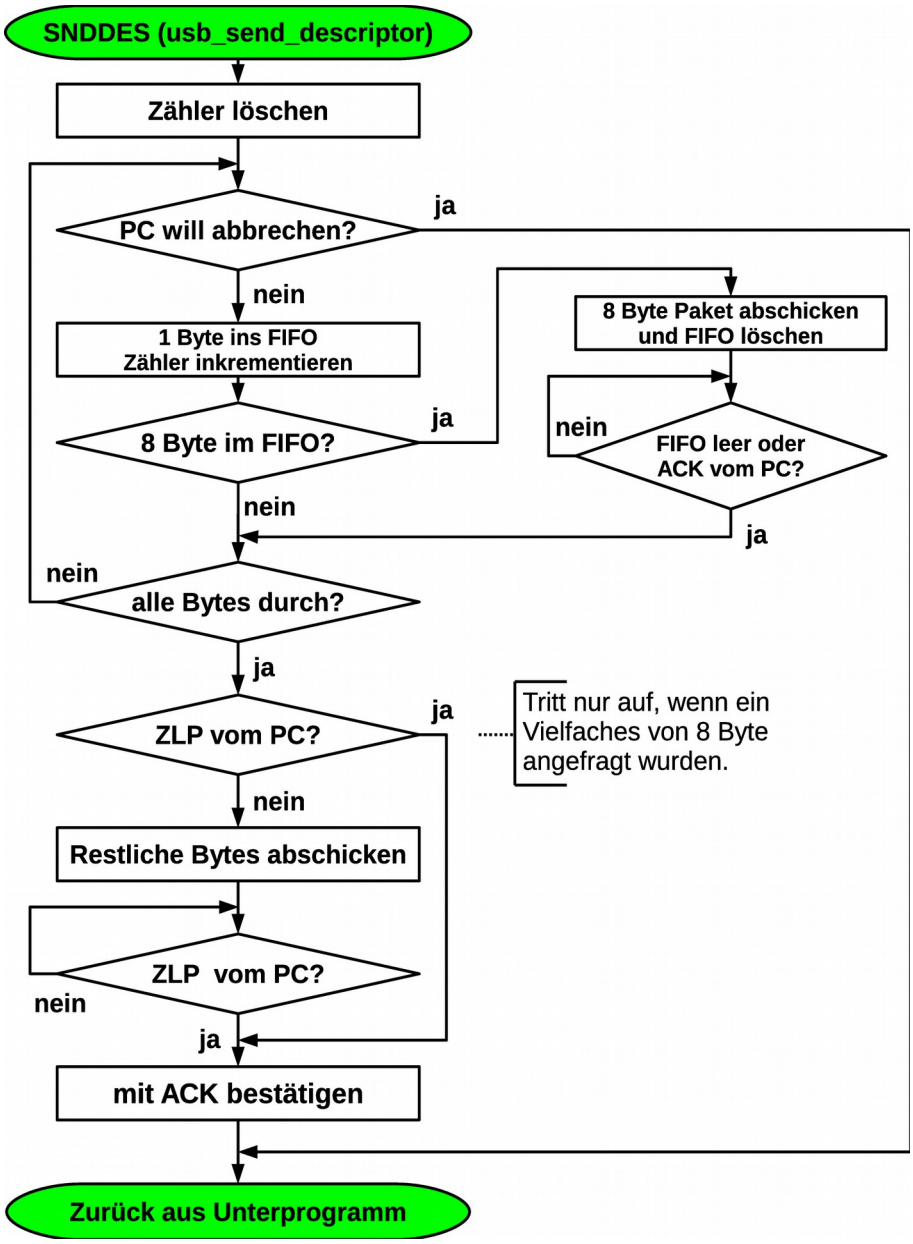
Set_Configuration: Um die benutzten Endpunkte zu Initialisieren müssen zuerst alle benutzten Endpunkte abgeschaltet werden und ihre Speicherbänke freigegeben werden. Dies passiert in einer Schleife. Mit dem schon bekannten Unterprogramm **INIEP** (**usb_init_endpoint**) werden dann die einzelnen Endpunkte initialisiert.
Nach der Initialisierung eines jeweiligen OUT-Endpunktes muss der **RXOUT**[24] **I**nterrupt für diesen OUT-Endpunkt (**UENUM** = Endpunkt-nummer!) erlaubt werden. Dadurch kann das Gerät einen neuen COM Interrupt auslösen, wenn Daten vom PC angekommen sind, und dann das zum OUT Endpunkt gehörende Unterprogramm aufrufen, um die Informationen abzuholen.
Der **NAKIN**[25] **I**nterrupt wird für den jeweiligen IN Endpunkt erlaubt (**UENUM** = Endpunktnummer!). Dadurch kann bei einer IN Anfrage des PC ein neuer COM Interrupt ausgelöst werden, der dann das entsprechende Unterprogramm zum IN Endpunkt aufrufen kann, um die angeforderten Informationen zu liefern.
Wurden alle Endpunkte erfolgreich konfiguriert, so wird ein ZLP an den PC gesendet und auf eine Bestätigung (ACK) vom PC gewartet.

SNDDES (usb_send_descriptor)

In einer Schleife wird der FIFO (Endpunkt 0) mit dem Deskriptor gefüllt. Tritt dabei eine Unterbrechung durch den PC (**RXOUT I**nterrupt) auf, so wird abgebrochen. Im Assembler befindet sich die Anzahl der zu schreibenden Bytes in **r18**, der Zeiger auf den Deskriptor in **Z**.
Da der FIFO 8 Byte groß ist, wird er mit jeweils 8 Byte gefüllt. Dann wird die Anfrage des PC mit einem ZLP bestätigt und darauf gewartet, dass die Speicherbank wieder frei ist (**TXIN I**nterrupt Flag) oder die Daten erfolgreich beim PC angekommen sind (ZLP vom PC, **RXOUT I**nterrupt Flag), bevor die nächsten 8 Byte gefüllt werden.
Sind alle angefragten Bytes im FIFO gelandet, so werden die restlichen Bytes abgeschickt, außer es war ein Vielfaches von 8 Byte angefragt worden. Es wird dann das ZLP vom PC gewartet und dieses dann mit einem ACK bestätigt.

24 Wird von Hardware gesetzt wenn OUT Daten vorhanden sind.

25 Wird von Hardware gesetzt wenn IN Anfrage von PC mit NAK beantwortet wurde.



Die Anwendungskommunikation

Die folgenden zwei Unterprogramme dienen der Anwenderkommunikation. UP **EP1_SP** und **EP2_AD** werden vom COM Interrupt aus aufgerufen.

Folgende Aktionen werden dabei ausgelöst:

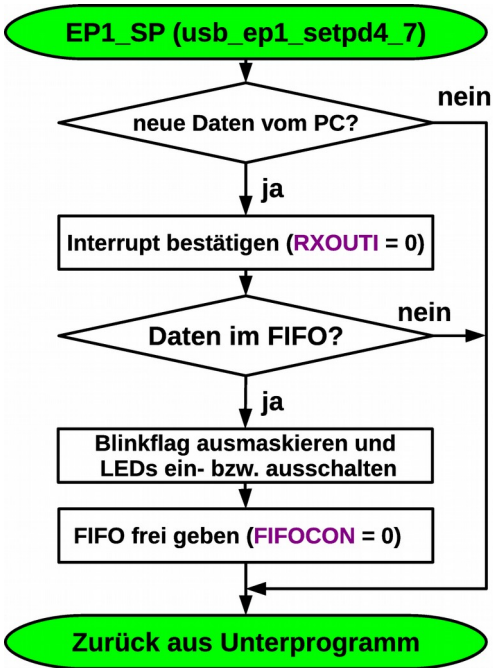
EP1_SP: Ein- und Ausschalten einzelner Bits (4 LEDs an PD4-PD7 und das Blink-Flag PD0) eines Ports (1 Byte) über Endpunkt 1 (OUT-Aktion (PC ⇨ Gerät)).

EP2_AD: Analoge Daten (Analog-Digitalwandler) über den FIFO des Endpunkt 2 einlesen (IN-Aktion (Gerät ⇨ PC)).

EP1_SP (usb_ep1_setpd4_7)

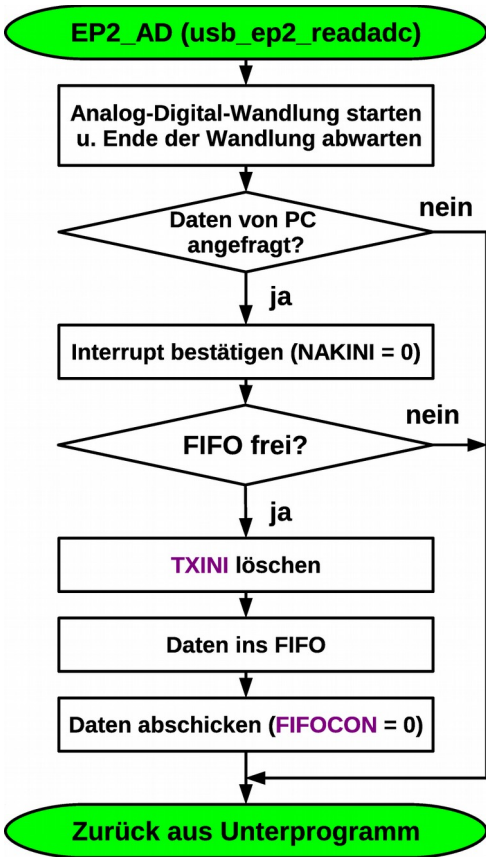
Lese Portbyte von PC und schalte Portbits ein bzw. aus.
Falls der PC Daten für Endpunkt 1 gesendet hat, so wird in der Interruptroutine dieses Unterprogramm aufgerufen. Das Interruptflag (**RXOUTI**) wird gelöscht und mit dem **RWAL**-Flag

wird überprüft ob die Daten im FIFO vorliegen und das Gerät die Erlaubnis hat die Daten zu lesen. Das Portbyte wird gelesen und die entsprechenden Aktionen werden durchgeführt (Ein- bzw. Ausschalten des Blinkens und der 4 LEDs).Mit dem Löschen des **FIFO CON**trol Bit wird gemeldet, dass die Speicherbank jetzt wieder frei ist.



EP2_AD (usb_ep2_readadc)

Sende analoge Daten an den PC.
Falls der PC Daten am Endpunkt 2 anfragt, so wird in der Interruptroutine dieses Unterprogramm aufgerufen. Als erstes wird eine AD-Wandlung ausgelöst und es wird mittels Polling auf das Ende der Wandlung gewartet. Das Interruptflag (**NAKINI**) wird dann gelöscht und es wird überprüft ob die Speicherbank frei ist. Zwei Byte (10 Bit-Wandlung) werden ins Fifo geschrieben. Das Abschicken der Daten wird durch Löschen des **FIFO CON**trol Bit erlaubt.


EP2_AD (usb_ep2_readadc)
Analog-Digital-Wandlung starten u. Ende der Wandlung abwarten
Daten von PC angefragt?
nein
ja
Interrupt bestätigen (NAKINI = 0)
FIFO frei?
nein
ja
TXINI löschen
Daten ins FIFO
Daten abschicken (FIFOCON = 0)
Zurück aus Unterprogramm