SPI

Das Serial Peripheral Interface (kurz SPI) ist ein Bus-System, mit dem digitale Schaltungen nach dem Master-Slave-Prinzip miteinander verbunden werden können. Dieses Bus-System ist inzwischen weit verbreitet; so taucht es bei Mikrocontrollern, SD-Karten, Funkmodulen und vielen anderen Peripheriegeräten auf.

Das SPI-Bus-System ähnelt dem bereits betrachteten I^2C-Bus. Während der I^2C-Bus jedoch mit einer einzige Datenleitung auskommt, besitzt das SPI-System zwei Datenleitungen, MOSI und MISO genannt. Auf ihre Bedeutungen kommen wir gleich zu sprechen.

Zusammen mit der Taktleitung (SCK = Serial Clock) benötigt das SPI-Bus-System also 3 Leitungen; deswegen spricht man auch von einem 3-wire-Bus (wire = Draht). Häufig tauchen neben diesen drei Leitungen noch weitere Leitungen auf, über welche die angeschlossenen Slave-Bausteine ausgewählt werden. In den folgenden Beispielen betrachten wir aber immer nur Systeme mit einem einzigen Master und einem einzigen Slave; in diesem Fall spielen diese Adressierleitungen dann keine Rolle.

Eine SPI-Verbindung kann z. B. so wie in der Abb. 1 aussehen. Master und Slave sind durch eine Takt- und zwei Datenleitungen miteinander verbunden. Daten können gleichzeitig vom Master auf den Slave und vom Slave auf den Master übertragen werden. Die Übertragung erfolgt seriell: Mit jedem Taktsignal sendet der Master ein Bit an den Slave; gleichzeitig empfängt der Master ein Bit vom Slave.

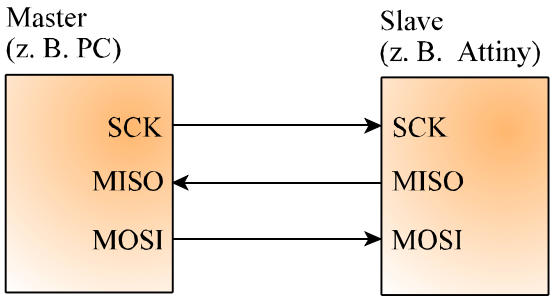

Abbildung 1

Master und Slave unterscheiden sich nur darin, wer den Takt angibt: Master ist der Baustein, welcher das Taktsignal gibt. Wir werden zunächst nur Beispiele betrachten, in denen ein PC als Master und unser Attiny 2313 als Slave eingesetzt werden. An dieser Stelle soll aber nicht verschwiegen werden, dass auch ein Mikrocontroller als Master eingesetzt werden kann.

Das Timing-Diagramm für die Übertragung eines Bytes ist in der folgenden Abb. 2 dargestellt:

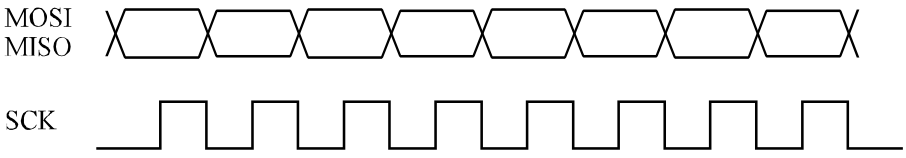

Abbildung 2

Ein Taktsignal besteht aus einer positiven und negativen Flanke. Bei der aufsteigenden Flanke des Taktsignals werden die einzelnen Bits von Master und Slave empfangen: Der Master kontrolliert seine MISO-Leitung (*Master-In*-Slave-Out) und der Slave seine MOSI-Leitung (Master-Out-*Slave-In*). Bei der absteigenden Flanke werden die nächsten Bits an die Ausgänge gelegt: Der Master gibt den Zustand des nächsten Bits an MOSI (*Master-Out*-Slave-In) aus und der Slave an MISO (Master-In-*Slave-Out*). Die Übertragung beginnt auf beiden Seiten mit dem Bit 7. Als letztes wird Bit 0 übertragen. Nach 8 Taktsignalen ist ein Byte vollständig übertragen.

Neben dem gerade geschilderten Übertragungsprotokoll existieren auch andere Varianten; so findet man z. B. SPI-Bus-Systeme, bei denen das SCK invertiert ist. Für weitere Informationen dazu sei auf die Spezialliteratur verwiesen; wir werden hier nicht weiter darauf eingehen.

SPI-Übertragung bei der Attiny-Platine

Bei unserer Attiny-Platine können SPI-Signale über die serielle Schnittstelle mit dem PC ausgetauscht werden. Dazu wird die Platine wie üblich mit einem COM-Kabel (oder ein USB-COM-Kabel) an den PC angeschlossen. Für den PC benutzen wir das Programm spi.exe. Dieses Programm gibt Taktsignale an die RTS-Leitung der COM-Schnittstelle und macht damit den PC zum Master. Bei jeder positiven Flanke liest es den Zustand des CTS-Signalleitung (MISO) und bei jeder negativen Flanke gibt es den Bitwert an die DTR-Leitung (MOSI) weiter.

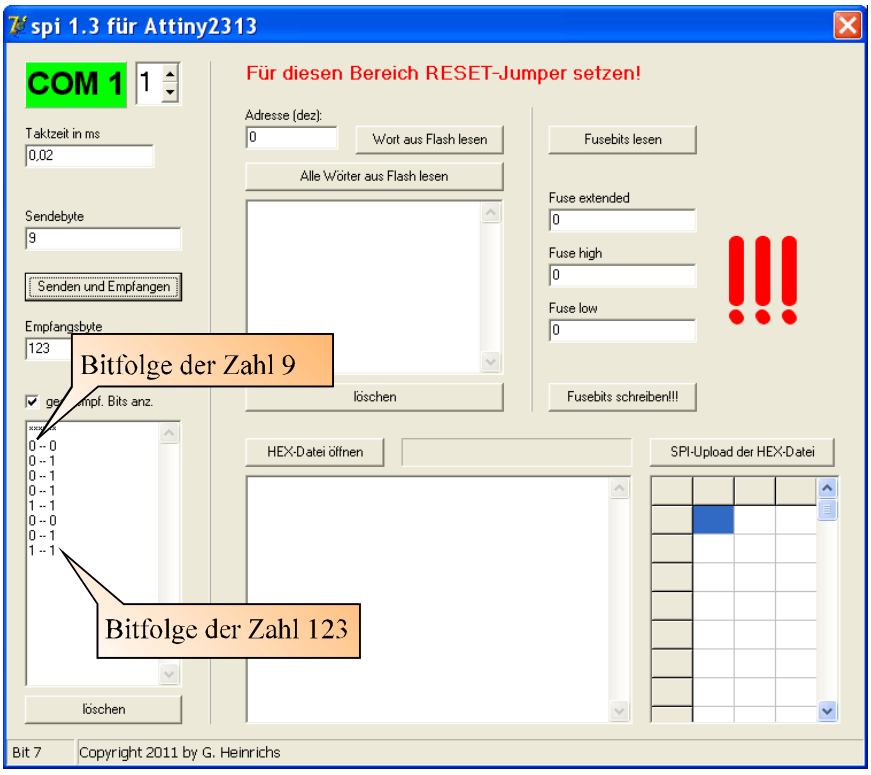

Abbildung 3: Das Master-Programm spi.exe

In dem Beispiel aus Abb. 3 hat das Programm spi.exe die Zahl 9 mit der Taktrate von 1 Bit pro 0,02 ms an den Attiny gesendet; gleichzeitig hat es vom Attiny die Zahl 123 empfangen. Die entsprechenden Bit-Folgen sind im linken unteren Textfeld zu erkennen. Auf die anderen Elemente des Formulars und ihre Bedeutung werden wir später noch zu sprechen kommen.

Zunächst einmal sollen hier die wesentlichen Zeilen des Quellcodes von spi.exe angegeben werden. Die Kommentare bei den entscheidenden Befehlen erübrigen weitere Erläuterungen.

```
function TForm1.spi(s: byte): byte;
type sztyp = array [0..7] of integer;
const stufenzahl: sztyp = (128, 64, 32, 16, 8, 4, 2, 1);
var halbetaktzeit: real;
var i, e, sbit, ebit: byte;
begin
  taktzeit := 0.01;
  e := 0;
  for i := 0 to 7 do begin
    sbit := s div stufenzahl[i]; //i-tes Bit
    s := s mod stufenzahl[i];    //Rest
    com.DTR := sbit;             //MOSI setzen
    com.delay(halbetaktzeit);
    com.RTS := 1;                //steigende Flanke bei SCK
    ebit := com.CTS;             //MISO lesen
    e := e + ebit * stufenzahl[i];
    com.delay(halbetaktzeit);
    com.RTS := 0;                //fallende Flanke bei SCK
  end;
  result := e;
end;
```

Damit die SPI-Signale vom Attiny2313 verarbeitet werden können, müssen die Taktleitung (SCK = RTS) und die Datenleitungen MISO = CTS sowie MOSI = DTR mit den Pins PB7, PB6, PB5 über Kabel verbunden werden (vgl. Abb. 4). Warum gerade diese Anschlüsse des Attiny2313 ausgesucht wurden, das werden wir im nächsten Abschnitt erklären.

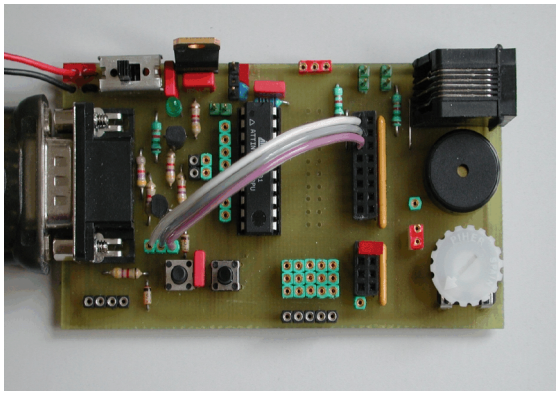

Abbildung 4

Die USI-Einheit des Attiny2313

Wir wollen unseren Attiny2313 nun so programmieren, dass er ein über SPI empfangenes Byte zunächst an PortB mittels LEDs ausgibt und dann bei der nächsten Übertragung dieses Byte wieder an den Master sendet. Da wir PB5-PB7 schon für das Bus-System benutzen, können mittels der LEDs natürlich keine großen Zahlen angezeigt werden; die Übertragung selbst gelingt natürlich mit allen Zahlen.

Bei den großen Brüdern der Atmega-Serie ist bereits eine komplette SPI-Schnittstelle (vergleichbar mit der USART für die COM-Schnittstelle) fest eingebaut (hardwareseitig implementiert). Eine derartige Komponente steht beim Attiny nicht zur Verfügung. Allerdings muss jetzt die Kontrolle der Taktleitung und die entsprechende Steuerung der Ein- und Ausgabe der Datenbits nicht von Grund auf neu programmiert werden; bei dieser Aufgabe erhalten wir nämlich Unterstützung von der USI-Einheit des Attiny2313 (USI = Universal Serial Interface).

Was leistet diese USI-Einheit und wie arbeitet man mit ihr? Um dies zu klären, betrachten wir zunächst ein vereinfachtes Modell dieser Einheit (Abb. 5).

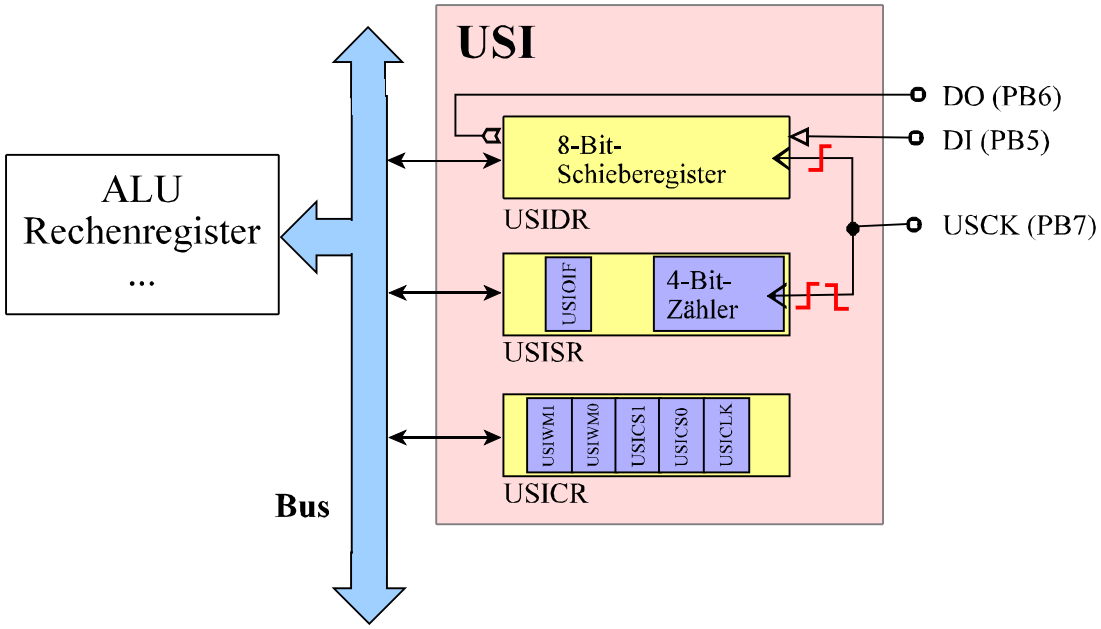

Abbildung 5: Vereinfachtes Modell der USI-Einheit

Die USI-Einheit besteht im Wesentlichen aus dem USI-Datenregister USIDR, dem USI-Statusregister USISR und dem USI-Kontrollregister USICR. Dabei ist das USIDR als Schiebregister ausgelegt: Bei jeder positiven Flanke am USCK-Eingang (USI-Clock) werden die einzelnen Bits dieses Registers nach links geschoben; dabei gelangen die am Eingang DI liegenden Signale Bit für Bit in das USIDR; gleichzeitig werden die schon im USI befindlichen Inhalte an den Ausgang DO gelegt und können dann Bit für Bit vom Master gelesen werden. Nach 8 Taktsignalen ist also der alte Inhalt des USIDR vollständig über DO ausgegeben und durch die vom Master über DI erhaltenen Bits ersetzt worden. Ausgabe und Eingabe erfolgen somit gleichzeitig.

Die ersten 4 Bits des USISR stellen einen 4-Bit-Zähler dar. Dieser Zähler erhöht seinen Wert jeweils um 1, wenn am Eingang USCK eine positive oder eine negative Flanke ankommt. Die Übertragung eines Bytes beginnt nun mit dem Zählerstand 0. Bei jedem Bit wird der Zählerstand um 2 erhöht. Nach dem letzten Bit gibt es einen Overflow, denn der 4-Bit-Zähler geht nur bis 15. Durch diesen Overflow wird das USIOIF-Bit (USI Overflow Interrupt Flag) gesetzt. Das

USIOIF = 1 zeigt demnach an, dass ein Byte vollständig übertragen worden ist. Davon werden wir bei der Programmierung der SPI-Schnittstelle Gebrauch machen. Die weiteren Status-Bits des USISR sind für uns hier nicht von Interesse.

Schauen wir uns jetzt noch das USICR genauer an:

Bit	7	6	5	4	3	2	1	0	
	USISIE	USIOIE	USIWM1	USIWM0	USICS1	USICS0	USICLK	USITC	USICR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	W	W	
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 6

Die USI-Einheit kann in verschiedenen Betriebsarten benutzt werden; neben dem hier dargestellten 3-wire-mode gibt es auch noch andere. Die Bitkombination USIWM1 = 0 und USIWM0 = 1 stellt den gewünschten 3-wire-mode ein. Die Bitkombination USICS1 = 1, USICS0 = 0 und USICLK = 0 legt fest, dass das USIDR und das USISR so wie oben beschrieben auf das Taktsignal reagieren. Wie die folgende Tabelle zeigt, kann die USI-Einheit auch ganz anders eingestellt werden und so z. B. an Master angepasst werden, die ein anderes Timing-Diagramm als in Abb. 2 besitzen.

USICS1	USICS0	USICLK	Shift Register Clock Source	4-bit Counter Clock Source
0	0	0	No Clock	No Clock
0	0	1	Software clock strobe (USICLK)	Software clock strobe (USICLK)
0	1	X	Timer/Counter0 overflow	Timer/Counter0 overflow
1	0	0	External, positive edge	External, both edges
1	1	0	External, negative edge	External, both edges
1	0	1	External, positive edge	Software clock strobe (USITC)
1	1	1	External, negative edge	Software clock strobe (USITC)

Abbildung 7: Die Taktmöglichkeiten der USI-Einheit (*aus: Attiny2313-Manual*)

Mit diesen Kenntnissen über die Bedeutung der USI-Register ist es nicht schwer, dass gewünschte Programm zu schreiben. Zunächst werden die nötigen Initialisierungen vorgenommen:

```
Ddrb = &B01011111   'PB5=DI und PB7=USCK als Eingang
Usicr = &B00011000  '3-wire-mode  und  Reaktion von USIDR und
                    USISR auf Takt
```

Nun wird das zu sendende Byte in das USIDR gelegt, der Zähler auf 0 gesetzt und das USIOIF-Bit gelöscht:

```
Usidr = S                'Sendebyte in USI-Datenregister
Usisr = &B01000000       'USIOIF löschen und USI-Zähler auf 0
```

Hier kommt eine Merkwürdigkeit zum Tragen: Das Löschen des USIOIF-Bits geschieht dadurch, dass es auf 1(!) gelegt wird. Nun wird so lange gewartet, bis dieses Flag durch einen Overflow des Zählers von USISR auf 1 gesetzt wird; dies signalisiert nämlich, dass die Übertragung komplett ist. Der Wert des empfangenen Bytes steht jetzt im USIDR und kann in eine Variable übertragen werden.

```
Do                        'Warten, bis 8 Takte vorbei
  Loop Until Usisr.usioif = 1
E = Usidr                 'Empfangsbyte aus USI-Datenregister
```

Jetzt wird das Ergebnis über Port B an die LEDs ausgegeben:

```
Portb = E
```

Nun legen wir das empfangene Byte in der Variablen S ab:

```
S = E
```

Auf diese Weise wird das empfangene Byte als Echo an den Master gesendet, wenn die oben stehenden Befehle ein weiteres Mal ausgeführt werden.

In der Abb. 8 ist das Programm nahezu vollständig angegeben; es fehlen lediglich die Deklarationen. Der Startwert der Variablen S ist 123; diese Zahl wird also als erstes vom Attiny an den Master übertragen.

Master und Slave bei der Arbeit

Schauen wir uns nun an, wie der Master (der PC mit dem Programm spi.exe) und der Slave (Attiny2313) miteinander arbeiten. Zuerst schließen wir die Platine an den PC an und laden das Programm Programm spi_slave.hex auf den Attiny. Anschließend schalten wir die Attiny-Platine wieder aus, stecken die 4 LEDs in PB0 bis PB3 und ziehen die nötigen Kabel wie in Abb. 4 dargestellt. Nun starten wir das Programm spi.exe auf dem PC und stellen die richtige COM-Nummer in der linken oberen Ecke des Formulars ein (Abb. 3). Erst jetzt schalten wir die Platine wieder ein. Diese Reihenfolge kann wichtig sein: Die COM-Schnittstelle kann nämlich ggf. bei ihrer Initialisierung schon einige Signale ausgeben, die vom Attiny fälschlicherweise als SPI-Signale gedeutet werden könnten.

Nun geben wir die Zahl 7 in das Feld “Sendebyte” ein und betätigen die Schaltfläche “Senden und Empfangen”. Der Attiny reagiert praktisch sofort und lässt die LEDs bei PB0, PB1 und PB2 aufleuchten. Außerdem entdecken wir im Feld “Empfangsbyte” die Zahl 123. Den nächsten Übertragungsvorgang wollen wir uns etwas genauer anschauen: Dazu stellen wir die Taktzeit auf

```
'************************************************************
'******************** Hauptprogramm ************************

PORTB = &B11111111
Waitms 100
PORTB = &B00000000
Waitms 100

S = 123

Do
  Call Spi
  S = E
Loop


'************************************************************
'******************** Unterprogramme ************************

Sub Spi
  DDRB = &B01011111
  Usicr = &B00011000
  Usidr = S
  Usisr = &B01000000
  Do
  Loop Until Usisr.usioif = 1
  E = Usidr
  Waitms 10
  DDRB = &B11111111
  PORTB = E
End Sub
```

Abbildung 9: Der Quellcode des Echo-Programms: spi_slave.bas

1000 ms und aktivieren die Anzeige der gesendeten und empfangenen Bits. Als Sendebyte geben wir die Zahl 9 ein. Wenn wir nun die Schaltfläche “Senden und Empfangen” betätigen, werden die einzelnen Bits im Sekundentakt übertragen; wir können dies in dem Textbereich unten links verfolgen. Links stehen die gesendeten und rechts die gleichzeitig empfangenen Bits.

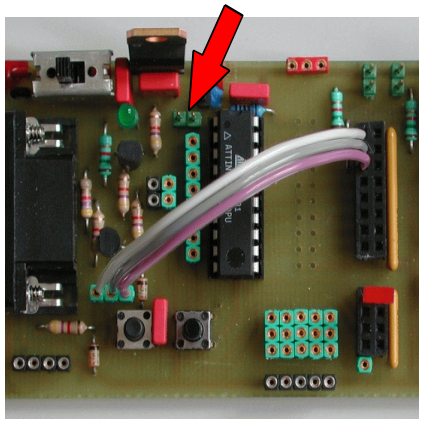

Abbildung 8

Das empfangene Bitmuster ist 00000111; das entspricht gerade der Zahl 7, welche bei der vorangegangen Übertagung an den Attiny gesendet worden war. Das Echo ist beim Master erfolgreich eingetroffen!

SPI-Programmierung des Attiny2313

Das SPI-Bus-System kann auch zur Programmierung des Mikrocontrollers benutzt werden. Dazu muss der Reset-Eingang des Mikrocontrollers auf Low gelegt werden; auf

der Attiny-Platine wird dafür ein Jumper wie in Abb. 9 eingesetzt. Wenn die Platine dann eingeschaltet wird, wird im Attiny ein fest eingebautes Programm gestartet, welches mithilfe von Steuerbefehlen durch den Master kontrolliert wird. Diese Steuerbefehle bestehen jeweils aus einer Folge von 4 Bytes.

Instruction	Instruction Format				Operation
	Byte 1	Byte 2	Byte 3	Byte4	
Programming Enable	1010 1100	0101 0011	xxxx xxxx	xxxx xxxx	Enable Serial Programming after $\overline{\text{RESET}}$ goes low.
Chip Erase	1010 1100	100x xxxx	xxxx xxxx	xxxx xxxx	Chip Erase EEPROM and Flash.
Read Program Memory	0010 **H**000	0000 00**aa**	**bbbb bbbb**	**oooo oooo**	Read **H** (high or low) data **o** from Program memory at word address **a**:**b**.
Load Program Memory Page	0100 **H**000	000x xxxx	xxxx **bbbb**	**iiii iiii**	Write **H** (high or low) data **i** to Program Memory page at word address **b**. Data low byte must be loaded before Data high byte is applied within the same address.
Write Program Memory Page	0100 1100	0000 00**aa**	**bbbb** xxxx	xxxx xxxx	Write Program Memory Page at address **a**:**b**.
Read EEPROM Memory	1010 0000	000x xxxx	x**bbb bbbb**	**oooo oooo**	Read data **o** from EEPROM memory at address **b**.
Write EEPROM Memory	1100 0000	000x xxxx	x**bbb bbbb**	**iiii iiii**	Write data **i** to EEPROM memory at address **b**.
Load EEPROM Memory Page (page access)	1100 0001	0000 0000	0000 00bb	iiii iiii	Load data **i** to EEPROM memory page buffer. After data is loaded, program EEPROM page.
Write EEPROM Memory Page (page access)	1100 0010	00xx xxxx	xbbb bb00	xxxx xxxx	Write EEPROM page at address **b**.
Read Lock bits	0101 1000	0000 0000	xxxx xxxx	xx**oo oooo**	Read Lock bits. "0" = programmed, "1" = unprogrammed. See Table 64 on page 158 for details.
Write Lock bits	1010 1100	111x xxxx	xxxx xxxx	11**ii iiii**	Write Lock bits. Set bits = "0" to program Lock bits. See Table 64 on page 158 for details.
Read Signature Byte	0011 0000	000x xxxx	xxxx xx**bb**	**oooo oooo**	Read Signature Byte **o** at address **b**.
Write Fuse bits	1010 1100	1010 0000	xxxx xxxx	**iiii iiii**	Set bits = "0" to program, "1" to unprogram.
Write Fuse High bits	1010 1100	1010 1000	xxxx xxxx	**iiii iiii**	Set bits = "0" to program, "1" to unprogram.
Write Extended Fuse Bits	1010 1100	1010 0100	xxxx xxxx	xxxx xxxi	Set bits = "0" to program, "1" to unprogram.
Read Fuse bits	0101 0000	0000 0000	xxxx xxxx	**oooo oooo**	Read Fuse bits. "0" = programmed, "1" = unprogrammed.
Read Fuse High bits	0101 1000	0000 1000	xxxx xxxx	**oooo oooo**	Read Fuse High bits. "0" = programmed, "1" = unprogrammed.
Read Extended Fuse Bits	0101 0000	0000 1000	xxxx xxxx	oooo oooo	Read Extended Fuse bits. "0" = programmed, "1" = unprogrammed.
Read Calibration Byte	0011 1000	000x xxxx	0000 000**b**	**oooo oooo**	Read Calibration Byte at address **b**.
Poll RDY/$\overline{\text{BSY}}$	1111 0000	0000 0000	xxxx xxxx	xxxx xxxo	If **o** = "1", a programming operation is still busy. Wait until this bit returns to "0" before applying another command.

Note: **a** = address high bits, **b** = address low bits, **H** = 0 - Low byte, 1 - High Byte, **o** = data out, **i** = data in, x = don't care

Abbildung 10: Tabelle der SPI-Programmier-Instruktionen (*aus: Attiny2313-Datenblatt*)

An einigen Beispielen soll erklärt werden, wie man mit diesen Instruktionen arbeitet; dabei greifen wir auf die oben angegebene Funktion `spi` zurück. Ansonsten werden die Anweisungen

umgangssprachlich beschrieben.

Beispiel 1

Der Programmspeicher soll ausgelesen werden, z. B. an der Adresse dec780. Dabei muss bedacht werden, dass jeder Programmspeicherplatz aus einem Wort, d. h. aus zwei Bytes besteht.

1. Nach dem Einschalten mindestens 20 ms warten

2. Programming enable - Befehl: spi($AC); spi($53); spi($00); spi($00)

3. Der Lesevorgang besteht aus 2 Read-Sequenzen: Zuerst wird das Lowbyte des Programmspeichers gelesen, dann das Highbyte. Im Byte 1 wird festgelegt, ob das Highbyte oder das Lowbyte gelesen werden soll.

   Die Adressen 0 bis 1023 des Attiny lassen sich als 10-Bit-Dualzahl darstellen; in unserem Fall ist dec780 = bin1100001100 = $30C . Die ersten 2 Bit werden mit dem Byte 2 der Read-Sequenz übertragen, die restlichen mit dem Byte 3. Was als 4. Byte gesendet wird, ist egal; wir können hier z. B. eine 0 senden. Entscheidend ist hier der Wert, der gleichzeitig vom Attiny geliefert wird.

   spi($40); spi($03); spi($0C); lowbyte = spi($00)
   spi($48); spi($03); spi($0C); highbyte = spi($00)

Wenn Sie das austesten wollen, können Sie diese Bytes mit dem Programm spi.exe einzeln der Reihe nach an den Attiny übertragen; das Lowbyte und das Highbyte des Programmspeichers 780 finden Sie dann im Feld “empfangenes Byte”. Leichter ist es, wenn Sie direkt die Adresse 780 in das Adressfeld eingeben und die Schaltfläche “Wort aus Flash lesen” betätigen. Dann führt spi.exe die obigen Sequenzen durch und Sie finden das Ergebnis im Speicherinhaltsfeld.

Beispiel 2

Einige Fusebits sollen gesetzt werden. Das “high fuse byte” soll die Bitfolge &B11011111 = $DF erhalten. **Achtung: Das Setzen von Fuse-Bits ist gefährlich!** Ein Fehler kann den Mikrocontroller unbrauchbar machen.

1. Nach dem Einschalten mindestens 20 ms warten

2. Programming enable - Befehl: spi($AC); spi($53); spi($00); spi($00)

3. spi($AC); spi($A4); spi($00); spi($DF)

4. Die letzte Instruktion erfordert eine gewisse Zeit, ähnlich wie das Schreiben eines Wertes

ins EEPROM. Wenn noch weitere Instruktionen folgen sollen, muss also gewartet werden, bis der Mikrocontroller wieder bereit ist. Dies kann mit der Poll ready/Busy-Instruktion geschehen:

```
Wiederhole
        spi($F0); spi($00); spi($00); bereit = spi($00);
        bereit = bereit and 1
bis bereit = 0
```

Eine solche Warteschleife muss auch nach anderen Schreibbefehlen durchgeführt werden.

Beispiel 3

Ein Programm soll in den Flash-Speicher des Attiny hochgeladen werden. Der Programmspeicher des Attiny213 ist aus 64 Seiten (Pages genannt) zu je 16 Wörtern aufgebaut. Das Hochladen geschieht Page für Page. Wir beschreiben hier, wie ein Programmabschnitt in die Page mit der Nummer 43 = $2B geladen wird.

1. Nach dem Einschalten mindestens 20 ms warten

2. Programming enable - Befehl: spi($AC); spi($53); spi($00); spi($00)

3. Speicher löschen mit spi($AC); spi($80); spi($00); spi($00) Dabei werden alle Bits auf 1 gesetzt. Die folgenden Schreibbefehle können nur aus solchen Einsen Nullen machen, aber nicht aus Nullen Einsen!

4. Warten bis Attiny bereit (vgl. Punkt 4 von Beispiel 2)

5. Eine Page hochladen; dabei gelangen die 16 Wörter zunächst in einen Zwischenspeicher des Attiny. Erst im nächsten Schritt werden die Daten in den Flash-Speicher geschrieben; deswegen ist hier die Pagenummer auch noch nicht relevant.

   ```
   Für z = 0 bis 15
           spi($40); spi(0); spi(z); spi(lowbyte)
           spi($48); spi(0); spi(z); spi(highbyte)
   nächtes z
   ```

6. Page in Flash-Speicher schreiben. Die Pagenummer $2B muss in zwei Nibbles (halbe Bytes) aufgeteilt werden: highnibble = $2; lownibble = $B.

   ```
   spi($4C); spi($02); spi($B0); spi(0);
   warten bis Attiny bereit
   ```

7. Gegebenenfalls weitere Pages hochladen und in den Flash-Speicher schreiben.