nChannel-Dimmer

Ein Entwicklungsbericht von Dipl.-Ing. (FH) Peter Salomon

1. Grundkonzept

Das Konzept für einen Mehrkanal-Dimmer geht von einer AVR-Mikrocontroller-Steuerung aus, wobei für jeden Kanal – im Maximalfall in der einfachen Variante sind das 8 Kanäle – der Dimmwert einstellbar sein soll. Als Einstell-Element soll ein einfacher Drehgeber verwendet werden, bei dem je nach Drehrichtung +/- der Dimmwert für den betreffenden Kanal erhöht, bzw. vermindert wird.

Der Dimmwert des aktuellen Kanals wird mit einem LCD-Display angezeigt und zwar als numerischer Wert in % und als Balkendiagramm für alle Kanäle. Die Bestätigung und Abspeicherung des Dimmwertes für den aktuellen Kanal erfolgt durch Drücken der im Drehgeber integrierten Taste. Dabei wird automatisch der nächste Kanal zum Einstellen angewählt. Nach dem 8. Kanal kann wieder der 1. Kanal eingestellt werden.

Durch längeres Drücken (> 1 sec) kann man den Einstellmodus verlassen werden, bzw. in den Einstellmodus gelangen.

Bei Unterbrechung der Betriebsspannung werden automatisch die gespeicherten Dimmwerte in den nichtflüchtigen Speicher (EEPROM) des Mikrocontrollers gesichert. Beim Einschalten werden die gesicherten Dimmwerte aus dem EEPROM in den Arbeitsspeicher der Mikrocontroller-Steuerung wieder zurückgeholt.

2. Erweiterung

In einer Erweiterung ist es denkbar, einzelne 8-Channel-Dimmer mittels einer Master-Slave-Konzeption zu einem nChannel-Dimmer beliebig zu erweitern. Die Bedienung und Anzeige (HDI) erfolgt dann ausschließlich vom Master aus. Als Anzeige wird in diesem Fall ein LCD-Grafik-Display eingesetzt und evtl. weitere Tasten zur Slave-Auswahl. Auf den Slaves sind keinerlei Bedien- und Anzeigeelemente notwendig. Die Kommunikation erfolgt über eine SPI-Verbindung (Serial-Peripheral-Interface), womit weitere Slaves angeschlossen werden können.

3. Realisierung 8-Channel-Dimmer

Da die AVRs nur maximal 4 Timerkanäle haben und es auch hierbei verschiedene Einschränkungen gibt, wurde hier die Aufgabe mit einem völlig anderen Konzept gelöst. Wenn davon ausgegangen wird, dass für eine PWM-Ansteuerung von LED-Beleuchtungseinheiten eine Auflösung von 256 Schritte völlig ausreicht, ist es naheliegend, die betreffenden 256 HIGH-/LOW-Werte in einem RAM-Bereich als Array abzulegen und diesen dann immer zyklisch auslesend auf einen AVR-Port auszugeben. Da die AVR-Ports in der Regel 8 Bits beinhalten, können auch bis zu 8 Bits (= 8 Kanäle) immer gleichzeitig ausgegeben werden.

3.1. Kernroutine

Kern der Software-Routine ist die Umwandlung von (max.) 8 Dimmwerten mit einem Zahlenbereich von jeweils 0 bis 255 in ein Array von 256 x 8 Bit, oder 256 Byte.
Dabei wird sowohl eine Schreibfunktion für das Einschreiben der Dimmwerte im Einstell-Modus unterstützt, wie auch die zyklische Lesefunktion zur Ausgabe auf den Port, z.B. Port B.
Beim Einschalten und bei Unterbrechung der Betriebspannung werden die gleichen Funktionen benutzt, um die Dimmwerte im EEPROM zu sichern, bzw. wieder zurück zu schreiben.

3.2. Eingabe

Die Eingabe der Dimmwerte erfolgt durch Zählung der Drehgeber-Impulse, wobei bis maximal +/- 255 gezählt werden. Dann fängt der Zähler automatisch wieder von Null an. Ein größerer Zählbereich ist in Anbetracht der Auflösung von 256 auch nicht sinnvoll.
Der Zählerstand wird aktuell im LCD-Display als %-Zahlenwert angezeigt, aber erst mit Bestätigung durch Drücken der Drehknopf-Taste in das RAM-Array eingetragen und damit auch im Balkendiagramm auf dem LCD-Display dargestellt.
Die einzelnen Dimmwerte sind in den Registern R1 bis R8 des AVRs abgelegt. Darauf Zugriff haben die Routinen der Eingabe, der Array-Übertragung, der Anzeige und auch der Datenrettung bzw. –Wiederherstellung.

Die Drehgeber-Signale A und B liegen am PortD, Bit0 und Bit7, die der Drehknopf-Taste auf Bit1 (gegen Masse).

Um in den Programm-Modus zu gelangen, muß zunächst die Taste länger als 1s gedrückt werden. Beginnend mit Kanal 1 kann nun durch Drehen des Drehgeber-Knopfes links oder rechts das bestehende PWM-Verhältnis des Kanals geändert werden.
Zur Auswertung der Drehgeber-Signale wird mit jeder H/L-Flanke von A ein Interrupt erzeugt und in der betreffenden ISR der dabei aktuelle Zustand an B abgefragt.

Achtung Änderung!
~~Je nach dem, ob HIGH oder LOW, wird in einem Arbeitsregister, z.B. R16, in das vorher der aktuelle Dimmwert aus einem der Register R1 bis R8 kopiert wurde, der Register-Inhalt erhöht oder vermindert. Die %-Anzeige erfolgt deshalb vom Arbeitsregister R16. Mit kurzem Drücken der Drehgeber-Taste (<< 1s) wird der neue Dimmwert in das betreffende Register R1…R8 zurückgeschrieben und danach automatisch der nächste Kanal angewählt.~~
Je nach dem, ob HIGH oder LOW, wird der Dimmwert in dem aktuell angewähltem Channel erhöht oder vermindert. Gleichzeitig wird dabei die %-Anzeige aktualisiert. Die 8 Dimmwerte sind im Registersatz R1 … R8 abgelegt, welche über das Zeigerregister R15 verwaltet werden. Mit kurzem Drücken der Drehgeber-Taste (<<1s) wird der nächste Kanal angewählt. Durch längeres Drücken (>1s) der Drehgeber-Taste wird der Programm-Modus wieder verlassen.
(Ende der Änderung)

In der LCD-Anzeige wird im Progamm-Modus sowohl alphanumerisch der zu bearbeitende Kanal angezeigt, wie auch im Balkendiagram die Veränderung in Echtzeit dargestellt. Im Normal-Modus erscheint nur das Balkendiagramm.

Der Programm-Modus kann jederzeit durch Drücken des Drehgeber-Knopfes >1s beendet werden. Dann wird mittels der Kernroutine das RAM-Array aktualisiert, damit der neuen Dimmwerte der soeben eingestellten Kanäle wirksam werden können.

3.3 Anzeige

Die Anzeige der Dimmwerte erfolgt mit einer 4x20 LCD-Textanzeige. Die Möglichkeiten des internen Anzeigecontrollers ausnutzend werden hier sowohl alphanumerische Zeichen aus dem internen Zeichengenerator dargestellt, als auch mit benutzerdefinierten Zeichen

(max. 8 Stück) eine quasi-grafische Balkenanzeige realisiert.
Die Anzeige erfolgt in der oben beschriebenen Weise, wobei für die Balkenanzeige auf die aktuellen Dimmwerte aus den Registern R1 bis R8 zugegriffen wird.

Die benutzerdefinierten Zeichen für die Balkenanzeige werden bei der Initialisierung der LCD-Anzeige als Daten-Array aus dem AVR-Programmspeicher in den Anzeigespeicher der LCD-Anzeige geschrieben.
Die LCD-Anzeige wird über das PortA im 4Bit-Modus betrieben.

3.3.1 LCD-Anzeige

Die 4-zeilige LCD-Anzeige hat folgendes Design
(2. veränderte Version mit Leistungs-Angabe in Watt):

```
   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1  PS                  :  :  :  :  :  :  :  :
2  nCh-Dimmer          :  :  :  :  :  :  :  :
3  Channel  x          :  :  :  :  :  :  :  :
4  x x x %   x x xW     1  2  3  4  5  6  7  8
```

*) xxx%-Anzeige
nur im ProgModus
(d.h. wenn Channel-Nr
aktiviert ist)

3.3.2 Übersicht Port-Belegung am AVR128 (Master)

	b7	b6	b5	b4	b3	b2	b1	b0	
Port A:	RS	R/W	E		D0/4	D1/5	D2/6	D3/7	LCD-Anschlüsse
Port B:					MISO	MOSI	CLK	/SS	SPI-Master
Port C:	~~Ch7~~	~~Ch6~~	~~Ch5~~	~~Ch4~~	~~Ch3~~	~~Ch2~~	~~Ch1~~	~~Ch0~~	~~8-Channel-PWM~~
Port D:	B				PW	PD	S	A	ABS=DG-Signale PD=Power-Down PW=LED-Betriebsspannung
Port E:									
Port F:	ADC0	ADC1	ADC2	ADC3	ADC4	ADC5	ADC6	ADC7	Analog-Steuereingänge

DG = Drehgeber (S = Schalter)

Das PowerDown-Signal wird über einen Spannungsteiler und ggf. Z-Diodenbegrenzung direkt vor dem Längsregler abgenommen. Kommt es zum PowerDown, muß der Stütz-

Elko nach dem Längsregler noch solange Energie für den AVR liefern, bis der die 8 Bytes aus den Registern R1…R8 in den EEPROM kopiert hat. Dazu dient die PowerDown-ISR.

3.3.3 Übersicht Port-Belegung am AVR16 (Slave)

	b7	b6	b5	b4	b3	b2	b1	b0	
Port A:							ADC1	ADC0	AD-Eingänge für Leistungsmessung ADC0: U ADC1: I
Port B:	CLK	MISO	MOSI	/SS				PW	SPI-Slave, PW: LED-Betriebspannung
Port C:	Ch7	Ch6	Ch5	Ch4	Ch3	Ch2	Ch1	Ch0	8-Channel-PWM
Port D:		LED2	LED1		INT1	INT0			LED1**:** SPI-Status LED2: MCrun (nur EvaBoard) INT0: PwDown-Eingang INT1: Wake-Up

Zusätzlich sollte auf dem Master-, wie auf dem Slave-Board je eine ISP-Steckverbindung vorhanden sein, um die AVRs auf dem Board programmieren zu können.

3.4. Dual-Chip-Lösung

Bei der Erprobung der ersten Variante mit nur einem ATMega64/128 stellte sich heraus, dass es bei Aktivitäten im ProgModus zu erheblichen Störungen der Kernroutine kommt. Dieses äußert sich in unregelmäßigen PWM-Zyklen und somit z.B. im Flackern der angeschlossenen LEDs.
Ursache ist hierfür die doch recht lange Unterbrechung der Kernroutine durch die ISR des Drehgebers, da die neuen Einstellwerte in Echtzeit auf dem LCD-Display angezeigt werden sollen. Insbesondere die recht häufig zu verwendeten Wait-Zyklen schlagen hierbei negativ zu Buche. Als Abhilfe könnten zwei Strategien verfolgt werden:

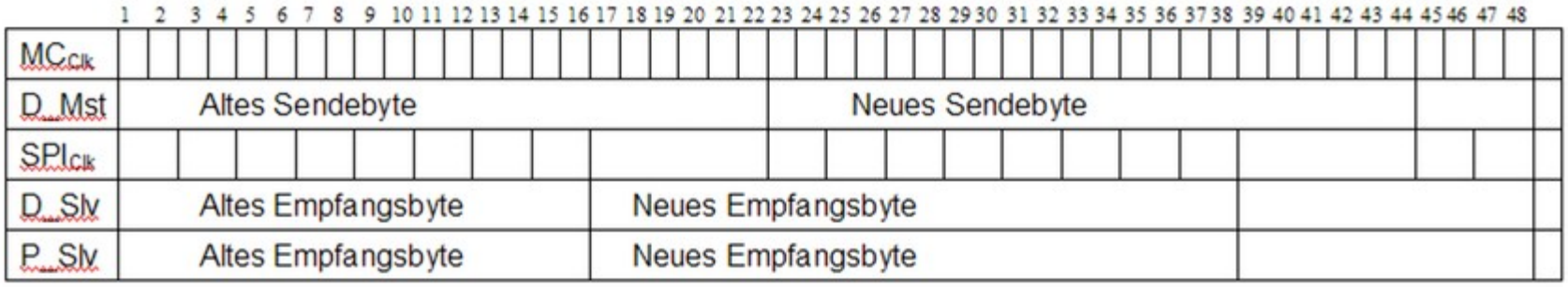
(1) Die Verwendung einer BF-(Busy-Flag) gestützten Basis-Routine für den Datenverkehr mit dem LCD

(2) Die Verwendung einer SPI-gekoppelten 2-Chip-Lösung, wobei im Master die gesamten Bedien- und Anzeigefunktionen (HDI) mit dem LCD realisiert werden

und im Slave lediglich die 8-Kanal-PWM-Kernroutine läuft, welche nur sehr kurz für die mit sehr hoher Geschwindigkeit ablaufende Datenübertragung vom Master-RAM zum Slave-RAM unterbrochen wird.
Unterbrechungen < 10ms sind nicht sichtbar.

Die zweite Variante hat außerdem den Vorteil, dass die leistungsintensiven LED-Treiber von der Bedienkonsole (HDI) getrennt sind und somit diese abgesetzt mit einem einfachen Netzwerkkabel mit dem Slave (Leistungsteil) verbunden werden kann. Handelsübliche RJ45-Steckverbinder ermöglichen eine sehr preiswerte Lösung.
Die SPI-Schnittstelle im Master- und auch im Slave muß allerdings hardwaremäßig mit Treiber-ICs ausgerüstet werden, da die AVR-internen Treiber sicherlich keine Leitungen treiben können.

Es gibt jedoch bei einer Lösung mit SPI-Master/Slave Timing-Probleme, die beachtet werden müssen. Da beim SPI-Master/Slave kein Handshaking vorgesehen ist, weis der Master nicht, wenn er seriell ein Byte übertragen hat, ob der Slave bereit ist, sofort das nächste zu empfangen. Obwohl das Empfangsregister doppelt gepuffert ist, ist nicht sicher, ob dem Slave genügend Zeit bleibt, die notwendigen Befehle zur Verarbeitung des soeben empfangenen Bytes auszuführen. Um eine Abschätzung der möglichen Übertragungsrate (die im Master und im Slave mit einigen Stufen einstellbar ist) soll nachstehendes Taktdiagramm dienen:


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
MC_Clk
D_Mst
Altes Sendebyte
Neues Sendebyte
SPI_Clk
D_Slv
Altes Empfangsbyte
Neues Empfangsbyte
P_Slv
Altes Empfangsbyte
Neues Empfangsbyte

Nach dem Einschreiben des Sende-Bytes in das Datenregisters des Masters beginnt mit dem 1. MC-Takt das Schieben in den Slave, im Minimum mit MC_{Clk} / 2. Wenn die 8 Bits in den Slave geschoben sind, wird die SPI-Taktung unterbrochen und sowohl im Master, als auch im Slave wird das SPIF-Bit (SPI-Interrupt Flag) gesetzt. Dieses kann in seiner ISR, wenn das SPIE-Bit (SPI Interrupt Enable) gesetzt ist und global Interrupts freigegeben sind, oder auch im Polling-Verfahren das nächste Byte in das Sende-Register laden. Dazu werden im Minimum 6 MC_{Clk} benötigt. Das empfangene Byte wird sofort in den Empfangspuffer kopiert, so dass der Slave bis zum vollständigen Eintreffen des nächsten Bytes im Minimum 16+6 MC_{Clk} zur Verfügung hat, um seine Daten-Bearbeitung durchzuführen. Danach würde das noch im Puffer liegende Byte durch das neue Empfangsbyte überschrieben. Für einfache, auch fortlaufende Daten-Transfers - auch ins RAM über den Umweg eines Universal-Registers - ist das allemal ausreichend. Kritischer sieht es allerdings aus, wenn zuvor jedes Mal erst die Zeigerregister manipuliert werden müssen, oder gar dazu noch Berechnungen notwendig sind.
Abhilfe schafft dann die Transfer-Geschwindigkeit herunterzusetzen,
also z.B. auf MC_{Clk} / 4.

Gleichzeitig zur Übertragung Master -> Slave findet eine Übertragung Slave -> Master statt. Wenn also sinnvolle Daten vom Slave zum Master übertragen werden sollen, müssen diese vor dem Beginn des Transfers ins SPI-Schieberegister eingeschrieben werden. Aus der Sicht des Slaves ist jedoch nicht bekannt, wann ein solcher Transfer stattfindet. Auch die Auswertung des /SS-Signal würde zu spät kommen, weil unmittelbar danach die Schiebe-Aktion beginnen kann (im Minimum 2x M_{Clk}). Sicherer ist deshalb unmittelbar nach dem vollständigen Empfang des vorhergehenden Bytes, gekennzeichnet durch Setzen des SPIF-Flags, das Schieberegister mit den zu übertragenden Daten für den nächsten Transfer zu beschreiben.
Umgekehrt muß dann im Master zunächst natürlich zuerst auch das Schieberegister gelesen werden, bevor neue Sende-Daten dort eingeschrieben werden können. Auch damit wird der Übertragungszyklus verlängert.

3.4.1. Softwarekonzept zum Übertragungsverfahren

Bei der Dual-Chip Lösung werden die Funktionen des HDI (Eingabe, Anzeige) und der eigentlichen Mehrkanal-PWM auf zwei getrennte Controller verteilt. Das hat den Vorteil, dass bei längeren Bearbeitungs-Sequenzen für Datenerfassung des Drehgebers und/oder

die LCD-Ausgabe keine merkliche Beeinflussung der kontinuierlichen PWM-Ausgabe stattfindet.
Realisiert wird das durch Kopieren der Dimmwerte des Daten-Registersatzes vom Master, in welchem nur noch die HDI-Funktionen implementiert sind, in den Slave, der nur noch die PWM-Ausgabe bewerkstelligt. Eine Aufbereitung der Dimmwerte für das RAM-Array zur Mehrkanal-PWM-Ausgabe wird somit im Master nicht mehr benötigt.

Andererseits sollen in einer weiteren Option Analogwerte (als Digital-Signal) über die aktuelle Gesamt-Leistung der angeschlossenen LED-Beleuchtungen vom Slave (dort befinden sich auch die Leistungsstufen) an den Master übertragen werden. Diese Analogwerte werden dann auf dem LCD-Display mit angezeigt.

Da sowohl im Master, als auch im Slave eine Datensicherung in den EEPROM beim PowerDown erfolgt und beim Wiedereinschalten diese gespeicherten Daten dann in den Registersatz R1…R8 zurückgeholt werden, braucht eine SPI-Übertragung nur zu erfolgen, wenn die Dimmwerte durch neue Einstellungen verändert werden. Werden also mittels der DGcounter-Routine neue Dimmwerte in einem Daten-Register erzeugt, so wird auch das aktuelle Register in das betreffende Register des Slaves übertragen. Rückwirkend wird nach einer kleinen Pause (zum Messen, Umsetzen im ADU usw.) der Analogwert der aktuellen Leistung (als Digital-Signal) vom Slave in den Master übertragen und dort angezeigt.

Die SPI-Übertragung erfolgt also in 3 (+1) Sequenzen:
- Übertragung des Registerzeigers für die Auswahl des aktuellen Datenregisters,
  die 1. Rückübertragung Analogwert wird ignoriert
- Übertragung des Dimmwertes in das mit dem Registerzeiger ausgewählten Datenregisters,
  die 2. Rückübertragung Analogwert wird ignoriert (nach kurzer Pause –>
  wird realisiert durch Master-interne Verarbeitungszeit für %-Ausgabe)
- nochmalige Übertragung des gleichen Dimmwertes ->
  wird im Slave ignoriert, aber die 3. Rückübertragung enthält den Spannungswert der neuen Leistung zur Bearbeitung und nach Übertragung einer zusätzlichen
- 4. Sequenz steht dann der Stromwert zur Berechnung der Leistung zur Verfügung, die dann auf dem LCD angezeigt werden kann.

(weitere Details im PAP 4.6.)

Alternativ könnte die Leistungsberechnung auch im Slave erfolgen, jedoch zu Lasten der Zeitscheibe für die PWM. Inwiefern dies mit größeren Störungen verbunden ist, wäre noch zu untersuchen. Vorteilhaft ist hierbei, dass nur 3 Sequenzen übertragen werden müssen.

Problem:

Bei der Slave-Realisierung mit einem ATMega16 liegt die JTAG-Schnittstelle ebenfalls auf dem C-Port, so dass kein OnChip-Debugging mit gleichzeitiger PWM-Ausgabe möglich ist. Das betrifft jedoch nur die von JTAG belegten 4 Pins vom C-Port, auf den anderen Pins ist durchaus trotzdem eine PWM-Ausgabe möglich. Der C-Port braucht auch nicht extra anders initialisiert werden. Die Umschaltung geschieht automatisch beim Setzen des JTAG-FuseBits.

Für den normalen Betrieb muß deshalb dafür gesorgt werden, dass das JTAG-FuseBit nicht gesetzt ist. Dafür ist - im Gegensatz zum Betrieb mit der JTAG-Schnittstelle – ein normaler AVR-Programmer, z.B. der „AVRISPmkII“ (oder ein entsprechender Clone) einsetzbar. Beide, sowohl der JTAG-Adapter, als auch der „AVRISPmkII“ arbeiten über den USB-Anschluß, jedoch auf unterschiedlichen Ports. Welche das sind, muß im Gerätemanager des PC abgefragt werden, da ggf. dieser Wert explizit beim Start in einem Konfigurations-Dialogfenster eingetragen werden muß. Die „Auto“-Portzuweisung funktioniert nicht immer zuverlässig.

Über JTAG läßt sich der SPI-Interrupt nicht abarbeiten, d.h. bei gesetztem SPIF wird – im Gegensatz zum Simulator - die ISR nicht angesprungen – warum???

3.4.2 SPI-Connections

3.4.2.1 Versuchsaufbau mit Pollin-LP (ATMega16) als Slave
ATMega128Evalboard als Master

Pollin-LP Stv J4		Belegung	Prüfstecker		ATMega128Eval-Board - PortB		Belegung
1	2	2: PB6(MISO)	9	10	1	2	1: PB0(/SS)
3	4	4: PB7(SCK)					2: PB1(SCK
5	6		7	8	3	4	3: PB2(MOSI)
7	8						4: PB3(MISO)
9	10		5	6	5	6	
11	12						
13	14		3	4	7	8	
15	16						
17	18		1	2	9	10	9: Vcc
19	20						10: Masse
21	22						
23	24						
25	26	25: PB4(/SS)					
27	28	27: PB5(MOSI)	(jetzt richtig !)				
29	30						
31	32						
33	34						

3.4.2.2 Probleme bei der SPI-Übertragung

Nach Programmierung des Masters und des Slaves mit der betreffenden Firmware stellt sich heraus, daß zwar eine SPI-Übertragung zustande kam, aber die einzutragenden Dimm-Werte in den einzelnen Kanälen waren mehr oder weniger zufällig. Es konnte zunächst nicht verifiziert werden, worauf diese Fehlererscheinung beruht. Ein OnChip-Debugging kann mit der ebenfalls unter SPI laufenden JTAG-Schnittstelle nicht durchgeführt werden.

Mittels eines Logik-Analysers konnte zwar die Wirkungsweise der SPI-Übertragung nachverfolgt werden, aber das Fehlerbild ließ sich auch damit nicht erkennen. Es zeigt sich lediglich, dass die 8 Sende-Takte auf der SCK-Leitung teilweise mit einem schmalen Nachtakt „ergänzt“ wurden, so dass die Vermutung einer Reflexion auf der Taktleitung nahe lag. Mit diesen „Nachtakten“ kam natürlich das Empfangs-Schieberegister des Slaves aus dem Takt und somit ist die willkürliche Datenübermittlung erklärbar.

Ein Dämpfungswiderstand von ca. 100 Ohm in Reihe vor dem Takteingang des Slaves sollte Abhilfe schaffen. Dem war aber leider nicht so!

Woher kamen also die „Nachtakte“?

Um das signaltechnisch untersuchen zu können, wird ein sog. Prüfprogramm benötigt, welches kontinuierlich immer wieder den gleichen „Dimm“-Wert über das SPI schickt. Mit einem Oszi konnte dann sehr schön der Zeitverlauf sowohl der Takt-Leitung, als auch der anderen Leitungen untersucht werden. Der Dämpfungswiderstand hatte auch wirklich die gewünschte Verminderung des Überschwingens zur Folge, aber die eigentliche Ursache war damit noch nicht gefunden. Die /SS-Leitung wird vom Master zur Auswahl des Slaves auf Low-Pegel gehalten. Sind jedoch auf dieser Leitung kapazitiv-induktive Einstreuungen möglich, wie es hier durch die direkt parallel verlaufenden SCK- und /SS-Leitungen innerhalb des Bandkabels der Fall ist, so können bereits kleinste Spikes in positiver Richtung auf der /SS-Leitung das Slave-SPI zum RESET veranlassen. Die Folge davon ist ebenfalls ein „außer Tritt kommen“ des Slave-SPI.

Die /SS-Leitung wurde direkt am Slave-Eingang mittels eines kleinen Kondensators von ca. 500pF soweit „entstört“, dass der Effekt nicht mehr auftritt. Das gilt jedoch wahrscheinlich nur im Zusammenhang mit dem bereits positiven Effekt des Dämpfungswiderstandes in der SCK-Leitung, welcher bereits steile Taktflanken verhindert.

3.5. Sleep-Funktion

Die Forderung nach einer Sleep-Funktion, d.h. das gezielte Abschalten aller Dimm-Kanäle und nach Bedarf Wiedereinschalten mit dem einzigsten Tastknopf, der in dieser Konzeption vorgesehen ist, ist nur auf recht kompliziertem Wege zu erfüllen.

3.5.1. Master-Seite

Mit dem im Drehgeber integriertem Taster wurden bisher bereits die Funktionen
- der Eintritt in den Programmiermodus (längeres Drücken ca. 1s) und
- die Kanalwahl im Programmiermodus (kurzes Drücken << 1s), sowie wieder
- die Beendigung des Programmiermodus (längeres Drücken ca. 1s)

realisiert.
Somit ist auf einfache Weise, d.h. ohne das bisher bestehende Gesamtkonzept des Tasten-Interface zu verwerfen, nur noch die Funktion „Kurzes Drücken“ außerhalb des Programmiermodus frei. Das würde sich auch gut in die Bedien-Philosophie einfügen, welche ein kurzes Drücken der Taste die Gesamt-Anlage ein- oder ausschaltet (Toggle-Funktion).
Im Master sind dazu verschieden Maßnahmen, sowohl auf der Hardware-Seite, als auch softwaremäßig notwendig:

<u>Hardware</u>

- ein Ausgang für die Steuerung der LCD-Stromversorgung, z.B. PortB6
- ein Low-Level-Interrupt-Eingang für die Tasten-Abfrage zum Aufwachen, z.B. INT1, welcher bereits für die Kanalwahl und den Programmiermodus vorgesehen ist. Somit sind dafür keine Hardware-Ergänzungen notwendig. Dieser Funktionsteil kann softwaremäßig implementiert werden.

<u>Software</u>

Außerhalb des Programmier-Modus bewirkt jede fallende Flanke am INT1 den Eintritt in den Sleep-Zustand (PowerDown, nicht zu verwechseln mit der PowerDown-Funktion bei Ausfall der Betriebsspannung), oder wieder das Erwachen daraus. Dazu sind verschiedene Maßnahmen zu realisieren:

Ausschalten

- Übermittlung an den Slaves über das SPI, dass der Sleep-Zustand eingeleitet wurde, dazu wird als 1. SPI-Sequenz die Adresse „0“ gesendet, die der Slave als „Sleep-Befehl“ interpretiert und alles weitere in Eigenregie abwickelt.
- nach vollständiger SPI Umkonfigurierung des PortB2: SPI -> normaler Ausgang,
- Abschalten der LCD-Stromversorgung,
- Selbstabschaltung (Befehl SLEEP).

Einschalten

- eine fallende Flanke an PortD1 bewirkt mit INT1 ein „Aufwachen“ des AVR aus dem Sleep-Zustand.
- Einschalten der LCD-Stromversorgung
- an PortB2 „High“ ausgeben, als Signal für den Slave aus dem Sleep-Zustand zu erwachen.
- Umkonfiguration von PortB2: normaler Ausgang -> SPI

3.5.2 Slave-Seite

Der Slave ist darauf angewiesen, was er vom Master über das SPI gesendet bekommt. Aber auch hier sind außer den Software-Ergänzungen hardwareseitige Maßnahmen vorzusehen.

<u>Hardware</u>

- ein Ausgang für die Steuerung der LED-Stromversorung
- ein Low-Level-Interupt-Eingang für die Abfrage „MOSI“-Leitung zum Aufwachen aus dem Sleep-Zustand. Da beim AVR16 nur zwei externe Interrupt-Eingange vorhanden sind – und die zufälligerweise nicht auf dem gleichen Port wie das SPI liegen, muß hier eine Parallel-Anschaltung an PortD3 (INT1) vorgenommen werden.
  INT0 ist bereits belegt durch den PowerDown-Interrupt bei Unterbrechung der Stromversorgung.
  PortD3 ist nur aktiv, wenn sich der Slave im Sleep-Zustand befindet, d.h. die Umkonfiguration wird softwaremäßig vorgenommen.

Software

Alle Funktionen des Slaves (mit Ausnahme vom PowerDown-Interrupt) werden vom Master über das SPI gesteuert. Dazu werden verschiedene Sequenzen übermittelt (siehe dazu Pkt. 3.4):

- 1.Sequenz -> Auswahl des Dimm-Kanals, Zahlenbereich von 1…8,
  hier ergibt sich die Möglichkeit durch die Zahl „0“ den Sleep-Befehl zu übermitteln.
- 2. bis 5.Sequenz sind hierbei ohne Bedeutung.

Andere Befehlszahlen für die 1.Sequenz im Bereich 10…255 sind zukünftig für optionale Slave-Adressen reseriert.

Ausschalten

Wird eine „0“ als 1.Sequenz über das SPI übermittelt, bedeutet das für den Slave:

- über PortB0 die Betriebspannung aller LED-Kanäle abschalten,
- Umkonfiguration PortB3: INT1 für die Wake-Up-Funktion
- Selbstabschaltung durch SLEEP-Befehl

Einschalten

Wird an PortB3 eine steigende Flanke empfangen, wird das als Interrupt zum Wake-Up des Slaves interpretiert. Dazu werden folgende Schritte durchgeführt:

- Umkonfiguration des PortB3: normaler Eingang, ohne Bedeutung
- über PortB0 wird die Betriebspannung der LED-Kanäle eingeschaltet.

3.6. Leistungsanzeige

3.6.1. Grundsätzliche Betrachtungen zur Leistungs-Anzeige

Mit dem ADU des AVR können Analogwerte mit 8Bit, bzw. im Höchstfall mit 10Bit Auflösung erfasst werden. Zur Anwendung der 10Bit Auflösung werden neben anderen Einschränkungen auch 2 Byte für die Übertragung benötigt. Das verkompliziert ungemein die weitere Verarbeitung, so z.B. statt 8Bit- eine 16Bit-Multiplikation.

Deshalb wäre zunächst abzuschätzen, inwieweit für den hier vorliegenden Anwendungsfall bereits eine 8Bit-Auflöung ausreicht.

Bei einer LED-Betriebsspannung von 24V würde sich eine Bit-Spannung von 24V/256 ˜ 0,1V ergeben, was eine Genauigkeit von 0,4% bedeutet.

Für den vorgesehenen Leistungsbereich von im Maximum 999W (8x ˜ 125W) würde sich somit ein maximaler Summenstrom von 999W/24V = 41,6A ergeben. Das bedeutet einen Bit-Strom von ˜ 0,16A, was wiederum einer minimal anzeigbaren Leistung von 3,9W entspricht, bzw. einem gleichgroßen Leistungsintervall.
Obwohl auch hier eine Genauigkeit von ˜ 0,4% erreicht wird, sollte in Anbetracht dessen, dass nicht in jedem Fall davon ausgegangen werden kann, dass alle 8 Kanäle angeschlossen sind und auch ein Leistungsintervall von fast 4W als zu hoch empfunden wird, die Auflösung wenigstens 1W betragen. Das würde dann eine Genauigkeit von 0,1% bedeuten.
Als Fazit muß also festgestellt werden, dass für die Spannungserfassung 8Bit ausreichen, jedoch für die des Stromes eine 10Bit-Genauigkeit notwendig ist.

3.6.2. Slave-Seite

Es sind maximal 8 Kanäle vorhanden. Es wird eine Summen-Erfassung vorgenommen, d.h. in der gemeinsamen Summenleitung aller Kanäle – ob in der Minus- oder Plusleitung, muß hardwaremäßig noch geklärt werden – wird der Summenstrom über einen Shunt-Widerstand gemessen. Gleichzeitig wird in einem zweiten Analog-Kanal die aktuelle Betriebs-Spannung der LED-Leuchten (oder –Leisten, Ketten usw.) gemessen. Beide Werte werden über den Analogmultiplexer und AD-Wandler des AVR-Slaves an den AVR-Master zur Berechnung und Anzeige der aktuellen Leistung übertragen.
Dazu werden in der 3. und 4. Sequenz eines Zyklus der SPI-Kommunikation zwischen dem Master und dem Slave die Daten rückwirkend vom Slave zum Master gesendet. Dazu sendet der Master drei mal hintereinander nochmals den aktuellen Dimmwert. Diese wird vom Slave zwar als Dimmwert ignoriert, aber dadurch wird signalisiert, dass jetzt der Slave zuerst den 8Bit-Spannungswerte (3. Sequenz) und anschließend in einer 4. Sequenz und 5. Sequenz die Stromwerte (Low- und High-Wert) an den Master senden soll.
Nach Empfang der 2.Sequenz (aktueller Dimmwert) wird im Anschluß an die PWM-Aktualisierung nach einer kurzen Wartezeit zur Stabilisierung der Messung der ADU veranlasst, den aktuellen Messwert in das SPI-Register zu transferieren. So können diese Daten dann rückwirkend mit der 3. Sequenz zum Master gesendet werden.
Analog dazu wird auch mit der 4. und 5. Sequenz verfahren.

3.6.3 Master-Seite

Nachdem der Master den aktuellen Dimmwert als 2. Sequenz gesendet hat (1. Sequenz ist immer die Übermittlung des aktuellen Kanals), wird nach einer notwendigen Wartezeit, um die Strom- und Spannungs-Messungen als stabile Werte im Slave bereitstellen zu können, eine 3. Sequenz mit dem gleichen Dimmwert an den Slave gesendet.
Rückwirkend wird dadurch vom Slave der 8Bit-Spannungswert übertragen und mit der 4. und 5. Sequenz – nochmals der aktuelle Dimmwert gesendet – liegt dann auch der 10Bit-Stromwert (Low- und High-Teil) im Master vor.
Daraus wird die aktuelle Leistung berechnet und im Display angezeigt.
(siehe dazu 3.3.1)

Spannungs- und Stromwerte werden als Integer-Zahlen zunächst in den drei aufeinander folgenden Registern R14:R13:R12 abgelegt. Eine Leistungsberechnung und Anzeige kann innerhalb der DGcounter_ISR.asm nicht erfolgen. Das würde den Zeitrahmen zu weit ausdehnen und die DG-Impulserfassung nachteilig beeinflussen. Da bereits die %-Anzeige in Echtzeit erfolgt, kann dieses nicht unbedingt als Nachteil angesehen werden.

Die Berechnung erfolgt mittels der AVR-Hardware-Multiplikation und deren Ergebnis wird in den Registern R9:R10:R11 als 24Bit-Binärwert abgelegt. Siehe Pkt. 4.7.
Somit kann auch gewährleistet werden, dass bei einem PowerDown-Ereignis diese Werte mit im EEPROM gesichert werden können (dazu muß die EEPROM-Routine noch erweitert werden).
Beim Start wird dann auch die letzte Leistungsmessung angezeigt, ohne über den ProgModus mit dem Drehgeber erst einen neuer Dimmwert einzustellen, um eine aktuelle Messung zu veranlassen.
Bei zwischenzeitlichen Änderungen der Lastverhältnisse an den einzelnen Dimmer-Channels kann dies natürlich vorübergehend zu fehlerhafter Anzeige führen, weshalb in solchen Fällen immer durch Bedienen des Drehgebers eine aktuelle Messung veranlasst werden muß.

Die Aktualisierung der Leistungsanzeige erfolgt alternativ zur %-Anzeige, d.h. bei Beenden des ProgModus wird die in Echtzeit erfolgte %-Anzeige durch die Leistungsanzeige ersetzt.

3.7. Analogwert-Eingabe

Für eine Fernsteuerung ist im Master die Möglichkeit einer analogen Schnittstelle 0…10V für maximal 8 Kanäle vorgesehen. Dazu wird der 8-Kanal-gemultiplexte AD-Wandler des AVR benutzt.

Zur Umschaltung in den Fernsteuer-Modus mit dem vorhandenen HDI (Ein-Kopfbedienung) wird ein weiterer „Channel: A“ eingeführt, welcher im Programmiermodus bei der Kanalwahl zwischen dem 8. und dem 1. Kanal im Display erscheint und als Fernsteuer-Betriebsart interpretiert wird, wenn man in dieser Einstellung durch längeres Drücken (ca. 1s) des Bedienknopfes aus dem Programmiermodus geht.

Die Handeinstell-Möglichkeiten bleiben davon unberührt, nur das Analog-Interface wird aktiviert. Erkennbar ist das daran, dass auch nach dem Verlassen des Programmiermodus die Anzeige „Channel: A“ nicht verschwindet. Andererseits wird in diesem Modus durch erneutes Aufrufen der Programmierfunktion (ca. 1s langes Drücken des Bedienknopfes), anschließender Wahl eines anderen, jedoch nicht „Channel: A“ und anschließend gleich wieder aus dem Programmiermodus rausgehen (wiederum ca. 1s langes Drücken des Bedienknopfes) die Fernbedienfunktion deaktiviert. Die Anzeige „Channel: A“ verschwindet und es ist wieder der normale Arbeitszustand des Masters eingestellt.

Zu speziellen Programmdetails siehe unter „4.3.1. Tasten-Interrupt (neu)“ und „5.8. Umschaltung in den Analog-Modus“.

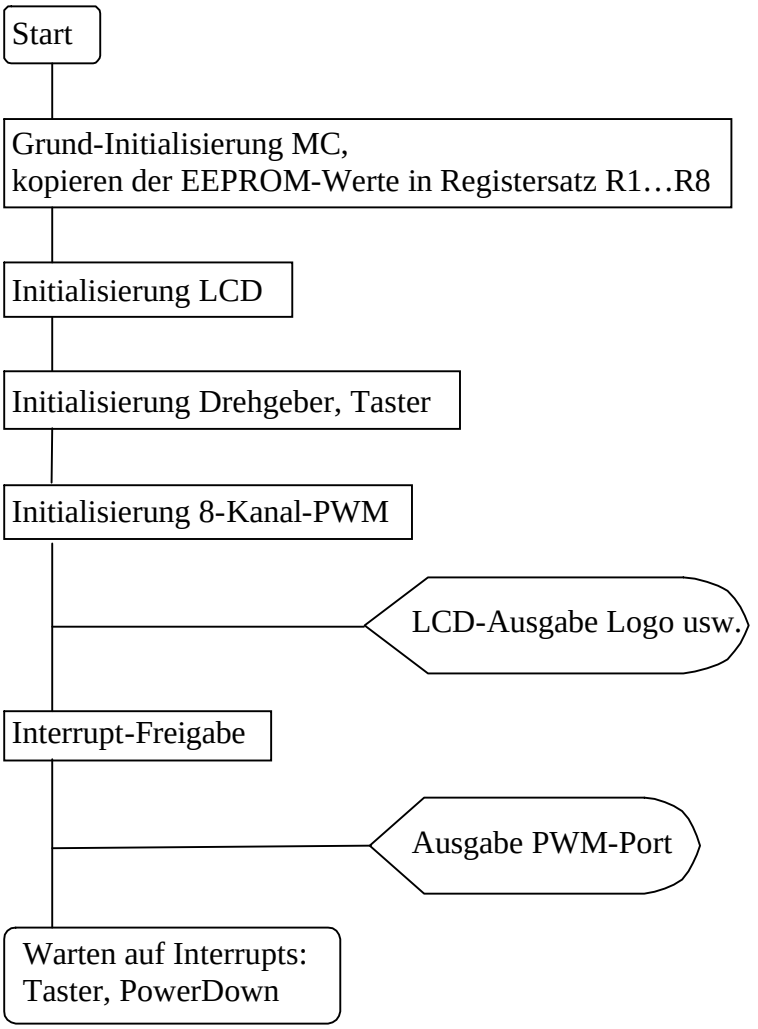
Im Analogwert-Fernsteuermodus wird der eingehende Analogwert für den aktuellen Kanal mit dem bestehenden Wert verglichen. Bei Abweichung (nach + oder -) wird der neue Dimmwert in das betreffende Register eingetragen, mit den 5 Sequenzen (siehe 4.7.2) dieser neue Dimmwert an den Slave geschickt und rückwirkend die dann auftretenden Messwerte zur Leistungsermittlung abgespeichert. Nach der Leistungsberechnung wird der numerische Wert in „Watt“ und die Balkenanzeige vorbereitet und mit einem LCD-Refresh werden dann die neuen Werte angezeigt.

3.8. Beschleunigung der DG-Eingabe

(ist nicht mehr bearbeitet worden)

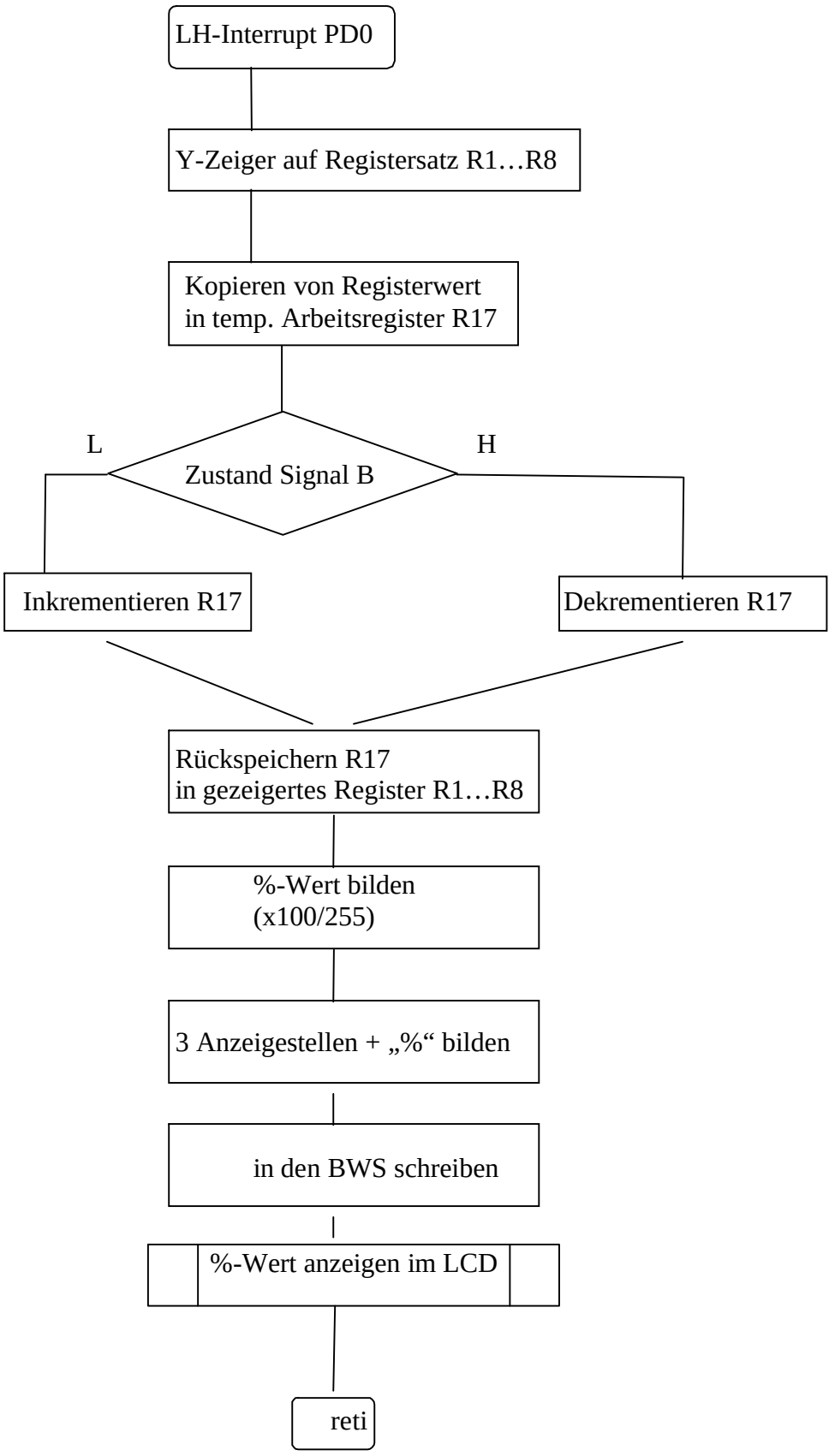
4. Programmablaufplan (PAP)

4.1. Grundroutine



4.2. Drehgeber-ISR

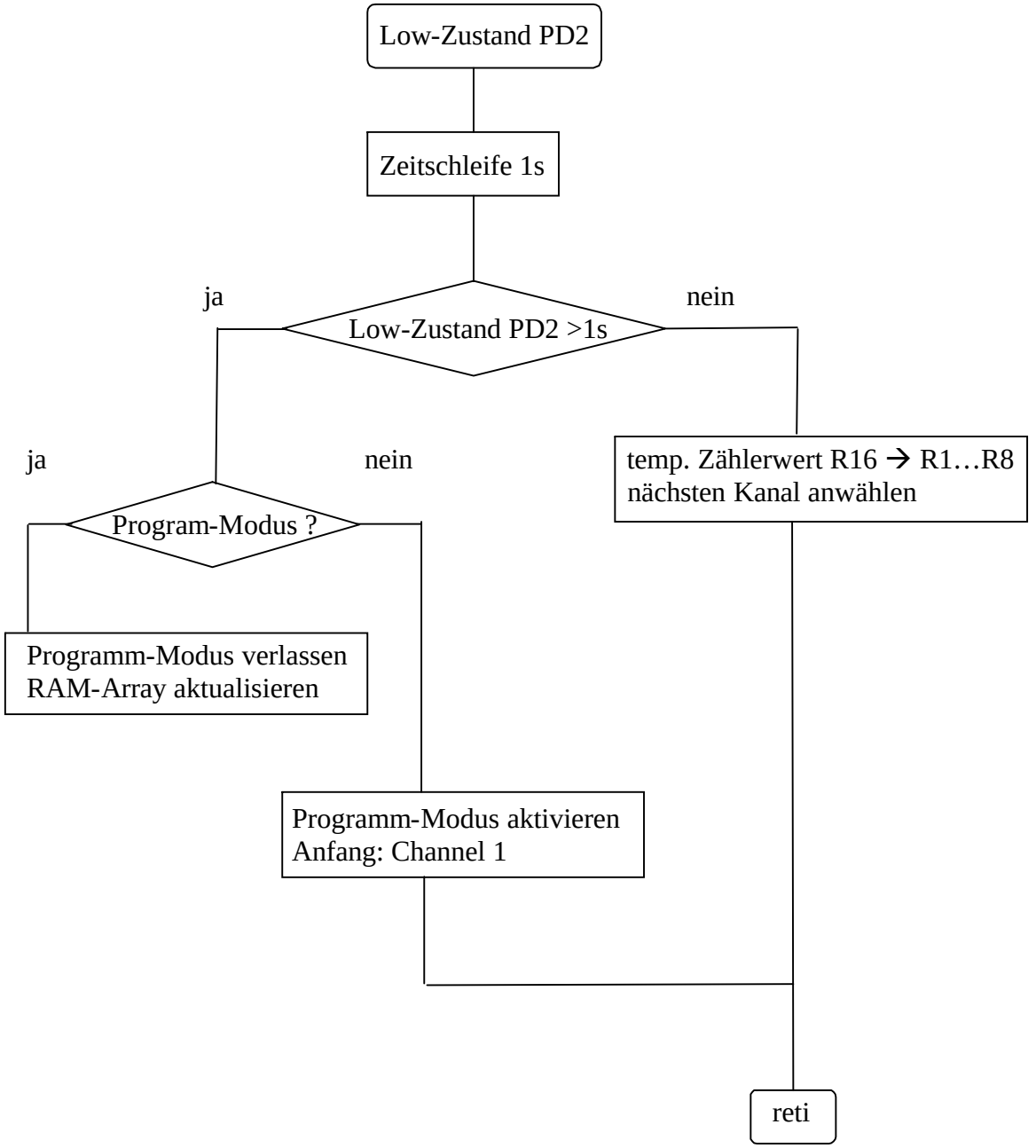
Der Drehgeber-Interrupt wird erst freigegeben, wenn die Drehgeber-Taste länger als 1 Sekunde gedrückt wurde (siehe 4.3 Tasten-Interrupt).



4.3. Tasten-Interrupt

(alte Variante – ist nicht implementiert!)

Der Tasten-Interrupt ist zustandsgesteuert, d.h. der Interrupt hält solange an, wie die Taste gedrückt ist.



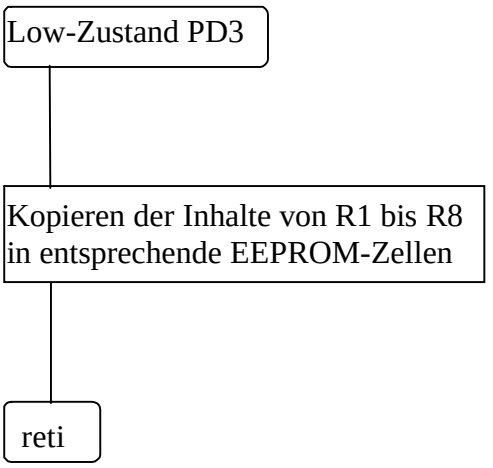
Die Merkerzelle für „Programm-Modus“ kann z.B. im T-Flag von SREG liegen.

Die neue Variante ist ebenfalls High/Low-flankengesteuert (siehe Taster_ISR.asm).

4.4. PowerDown-Interrupt

Damit beim Ausschalten, oder Unterbrechung der Betriebsspannung die eingestellten Dimmwerte nicht verloren gehen, müssen diese nach Erkennung des Zusammenbruchs der Betriebsspannung am Netzteil noch schnell im EEPROM gesichert werden. Solange muß ein Dioden-entkoppelter Stützkondensator noch Energie für den MC liefern.
Beim Einschalten verhindert eine späte Interrupt-Freigabe undefinierte Zustände.

Auch hier wird ein zustandsgesteuerter Interrupt verwendet.



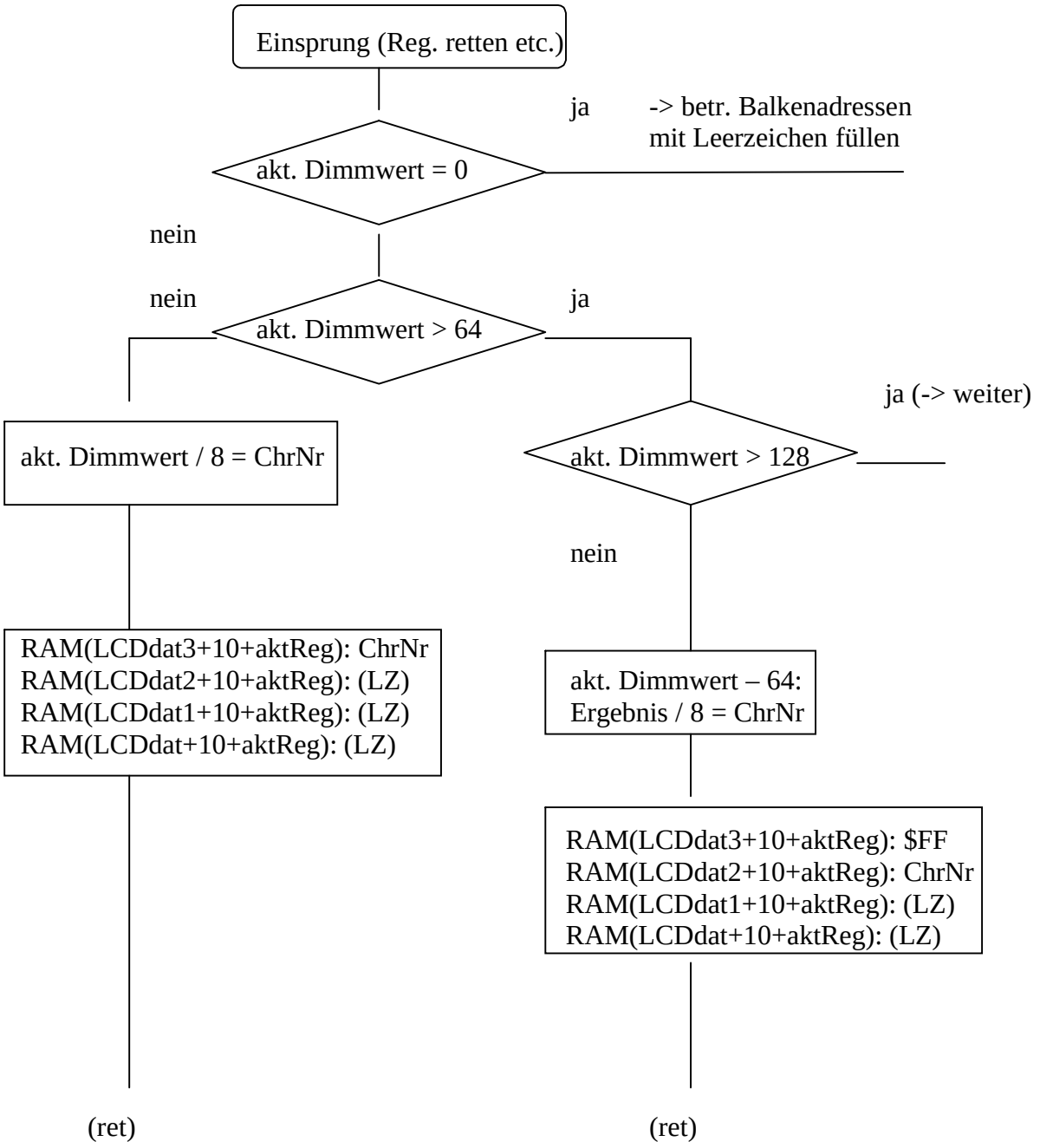
Beim Einschalten werden die gespeicherten Werte vom EEPROM wieder zurück in den Registersatz R1 bis R8 geladen.

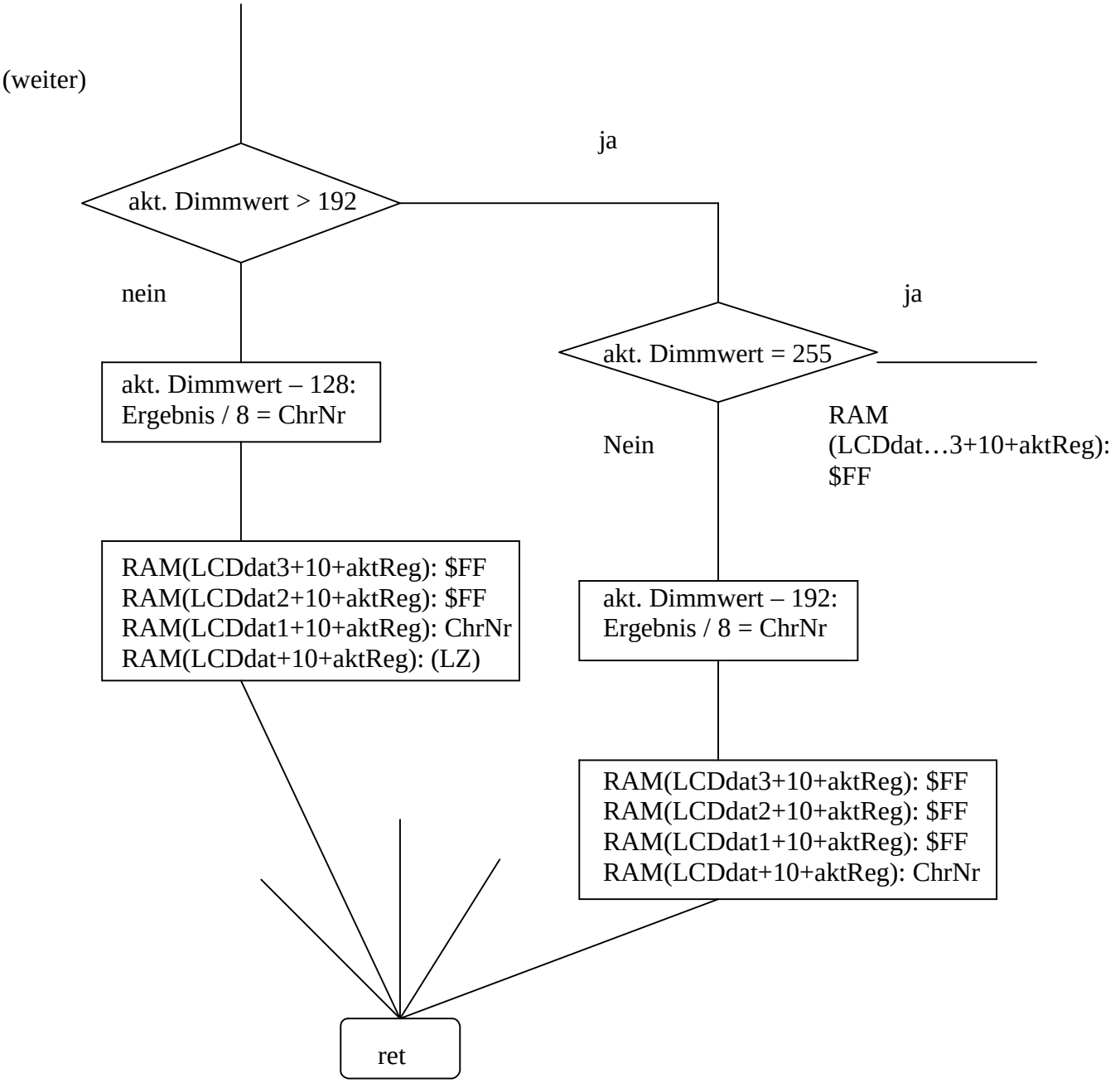
4.5. Balkenanzeige

Voraussetzung ist, dass im LCD-CharakterRAM die benutzerdefinierten Zeichen entspr. 5.5.1 abgelegt sind. Diese 8 Zeichen stehen ab der Adr. 00 zu Verfügung.

Die Balkenanzeige wird erstmals nach der Initialisierung kurz vor dem Einsprung in die Hauptroutine aufgerufen, da zu diesem Zeitpunkt bereits die Dimmwerte aus dem EEPROM im Registersatz R1…R8 zurückgeschrieben zu Verfügung stehen.

Des Weiteren wird nach jedem Beenden des ProgModus ebenfalls die Routine der Balkenanzeige aufgerufen, um die LCD-Anzeige zu aktualisieren. Ein Einbinden der Balkenanzeige zur Echtzeitdarstellung bei der DG-Eingabe erscheint aus heutiger Sicht zu problematisch, wegen der damit verbundenen Verlängerung der DGcounter_ISR. Dann wären nur noch langsamere Dreh-Eingaben möglich.
(die nachfolgende Routine ist falsch: Die Balkenanzeige geht nur über 3 Zeilen!)

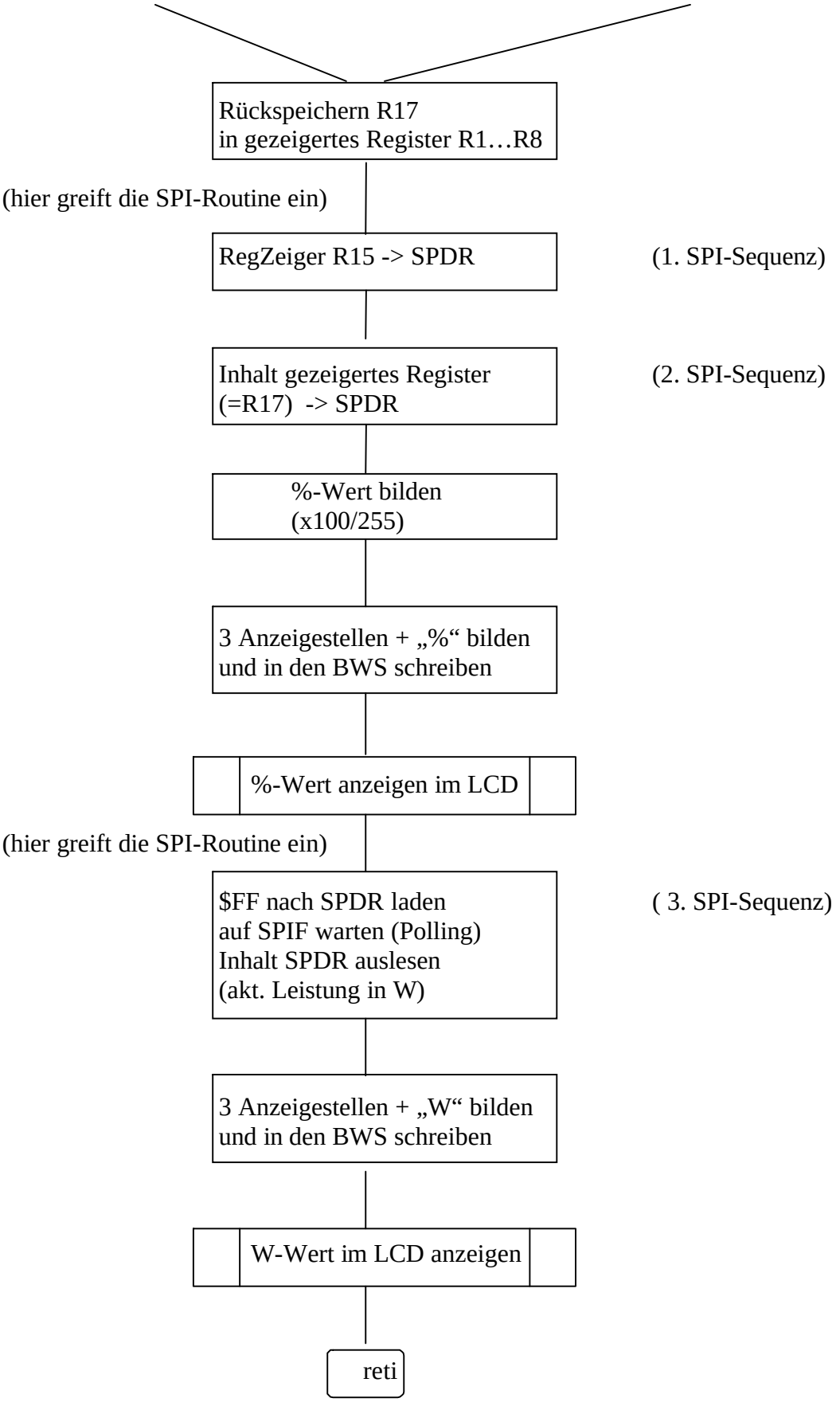



(weiter)
akt. Dimmwert > 192
ja
nein
akt. Dimmwert = 255
ja
akt. Dimmwert – 128:
Ergebnis / 8 = ChrNr
Nein
RAM
(LCDdat…3+10+aktReg):
$FF
RAM(LCDdat3+10+aktReg): $FF
RAM(LCDdat2+10+aktReg): $FF
RAM(LCDdat1+10+aktReg): ChrNr
RAM(LCDdat+10+aktReg): (LZ)
akt. Dimmwert – 192:
Ergebnis / 8 = ChrNr
RAM(LCDdat3+10+aktReg): $FF
RAM(LCDdat2+10+aktReg): $FF
RAM(LCDdat1+10+aktReg): $FF
RAM(LCDdat+10+aktReg): ChrNr
ret

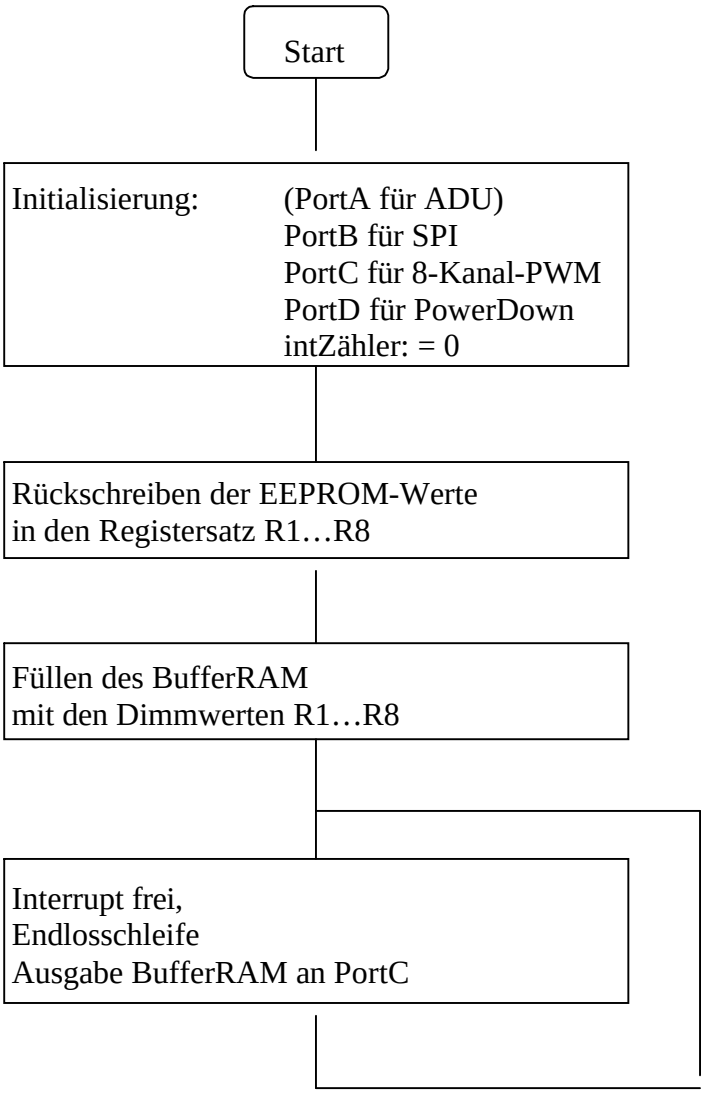
4.6. SPI-Übertragung (Dual-Chip Lösung)

4.6.1. Master

(Ausschnitt des PAP zur DG_ISR) – Achtung! – Ist nicht mehr aktuell!



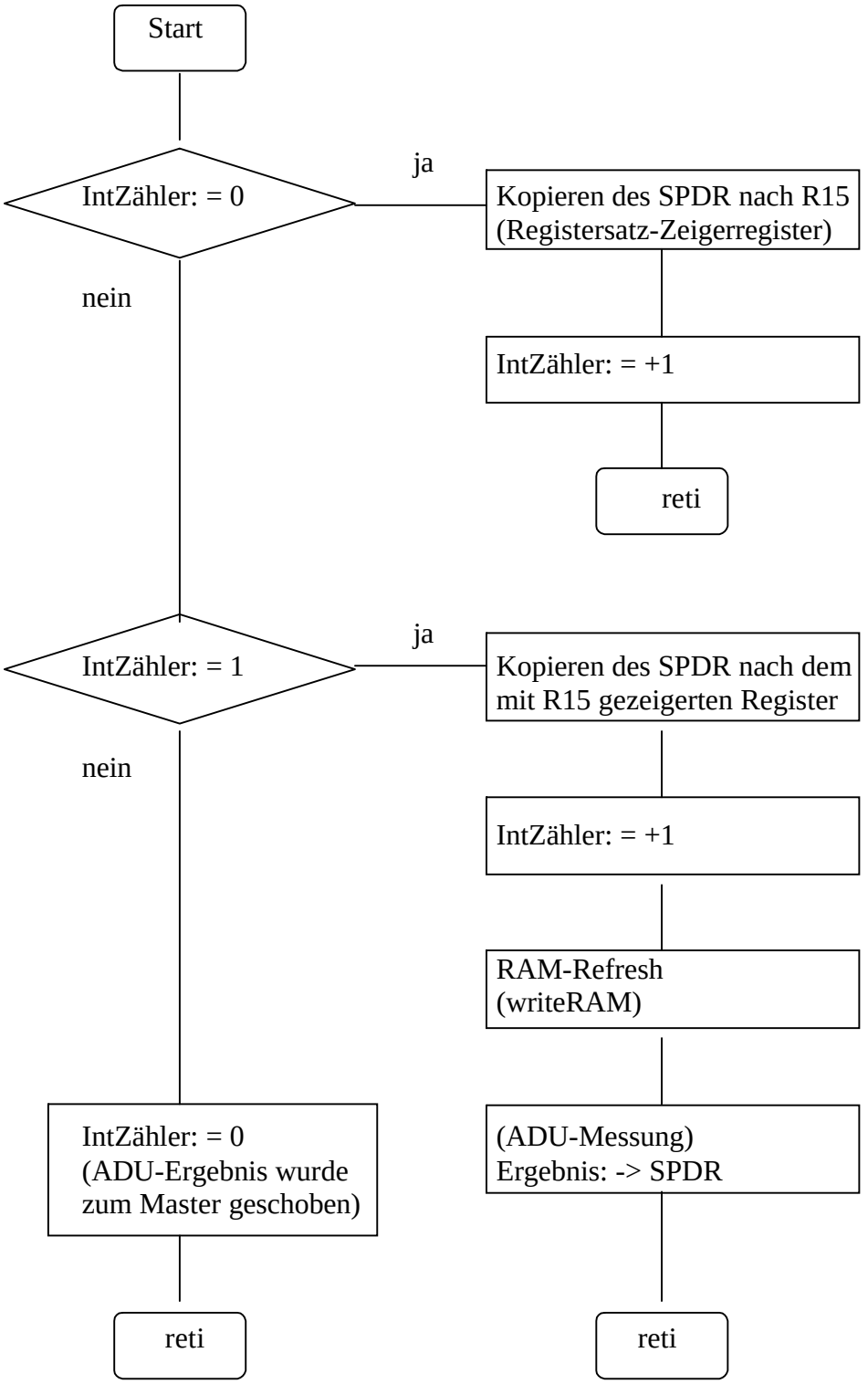
4.6.2. Slave
4.6.2.1. Initialisierung und Basisroutine



4.6.2.2. SPI-ISR

(wird aufgerufen bei Ende des Schiebevorgangs: Setzen von SPIF)

– Achtung! Ist nicht mehr aktuell! -



4.7. Leistungsanzeige (Master-Slave Version)

4.7.1 Master-Seite

Die Spannungs- und Stromwerte werden als Rückmeldedaten vom Slave an den Master gesendet und in den Registern (Achtung – neue Registerbelegung !)

~~R16~~ R14 Spannungswert

~~R17~~ R13 Stromwert (High-Teil)

~~R18~~ R12 Stromwert (Low-Teil)

abgelegt.

Der ADU im Slave ist für den Spannungswert so parametriert, dass ein links-bündiges Ergebnis erzeugt wird. So kann der Spannungswert als einfache 8Bit-Größe übertragen werden. Für den Stromwert muß hingegen ein rechtsbündiges Ergebnis erzeugt und auch so übertragen werden.

Zur Multiplikation der Spannungs- und Stromwerte soll hier aus Performance-Gründen der spezielle AVR-Befehl `mul` verwendet werden. Dieser ist fest mit dem 1. Registerpaar R1:R0 verbunden und kann „nur“ zwei 8Bit-Werte multiplizieren mit dem Ergebnis in R1:R0. Multiplikationen mit >8Bit werden dann genauso berechnet, als wenn man eine schriftliche Multiplikation vornimmt. Siehe dazu die Application Note AVR201 unter Pkt. 5.9.

Da R1…R8 mit den aktuellen Dimmwerten belegt sind, muß zunächst R1 gesichert werden.

R1 -> Stapel

	Spannungswert =: ~~R16~~		R14	x	Stromwert (Low) =: ~~R18~~		R12
z.B.	10V	=: 10.0(00) mV	$64		1000mA =: 1000		$(03)E8
		=: 100 (Zahlenwert)					
	Ergebnis =: R1:R0	->	R10:R9 (Low)				
			$5AA0				
	Spannungswert =: ~~R16~~		R14	x	Stromwert (High) =: ~~R17~~		R13
			$64				$03
	Ergebnis =: R1:R0						
			$012C				

R1 -> R11

R10 =: R10 add R0 $86

R0 =: 0

Endergebnis =: R11 adc R0; -> R11:R10:R9
$0186A0 =: 100.000 (Zahlenwert)
=: 10.000.0(00) µW

Stapel -> R1

In ~~R21:R20:R18~~ R11:R10:R9 steht jetzt ein 24Bit-Wert, der die Leistung in mV(*100) x mA = µW(*100) ausdrückt! Der Spannungswert liegt im Zahlenbereich 0…255(00)mV vor - deshalb *100 - und der Stromwert im Zahlenbereich 0…25500mA, das ergibt dann maximal 650.250.0(*100)µW, d.h. 6502500 / 10000 = 650W, oder rund 80W/Kanal.
Der 24Bit-Wert der berechneten Leistung kann jetzt auch mit in den EEPROM gesichert werden.

Schwieriger ist jetzt die Anzeige.
Zunächst muß der 24Bit-Binärwert der Leistung entsprechend dem oben genannten Rechenweg durch 10.000 dividiert werden. Dazu wird wie bei der schriftlichen Division vorgegangen, d.h. als erstes wird das Doppelbyte R11:R10 mit einer 16Bit-Division durch 10.000 ($2710) dividiert. Das 1. Teil-Ergebnis steht in R17:R16 und der Rest in R22:R21.
Dieses Verfahren hat sich leider als falsch erwiesen!
Da eine fertige 24Bit/16Bit-Divisionsroutine nicht zu finden war, wurde versucht eine solche „mal schnell“ selbst zu kreieren. Dies ging trotz nachstehender Vorlage für ein einfaches Verfahren, allerdings nur für 16/8Bit nicht mit einem Erfolg aus. Die Routine „div24_16.asm“ rechnet leider nur richtig, solange als Ergebnis keine größere Zahl als 255 herauskommt.
(weiter nach diesem Exkurs in die „Weiten“ der binären Division…) -> siehe:
http://www-user.tu-chemnitz.de/~heha/Mikrocontroller/Division.htm

Denkmodell 2: Praktische Registeraufteilung für Mikrocontroller
Man beachte beim Subtraktionstest, dass ein vorher herausgeschobenes Übertragsbit *auf jeden Fall eine Subtraktion erzwingt*; das ist in diesem Beispiel nie der Fall.

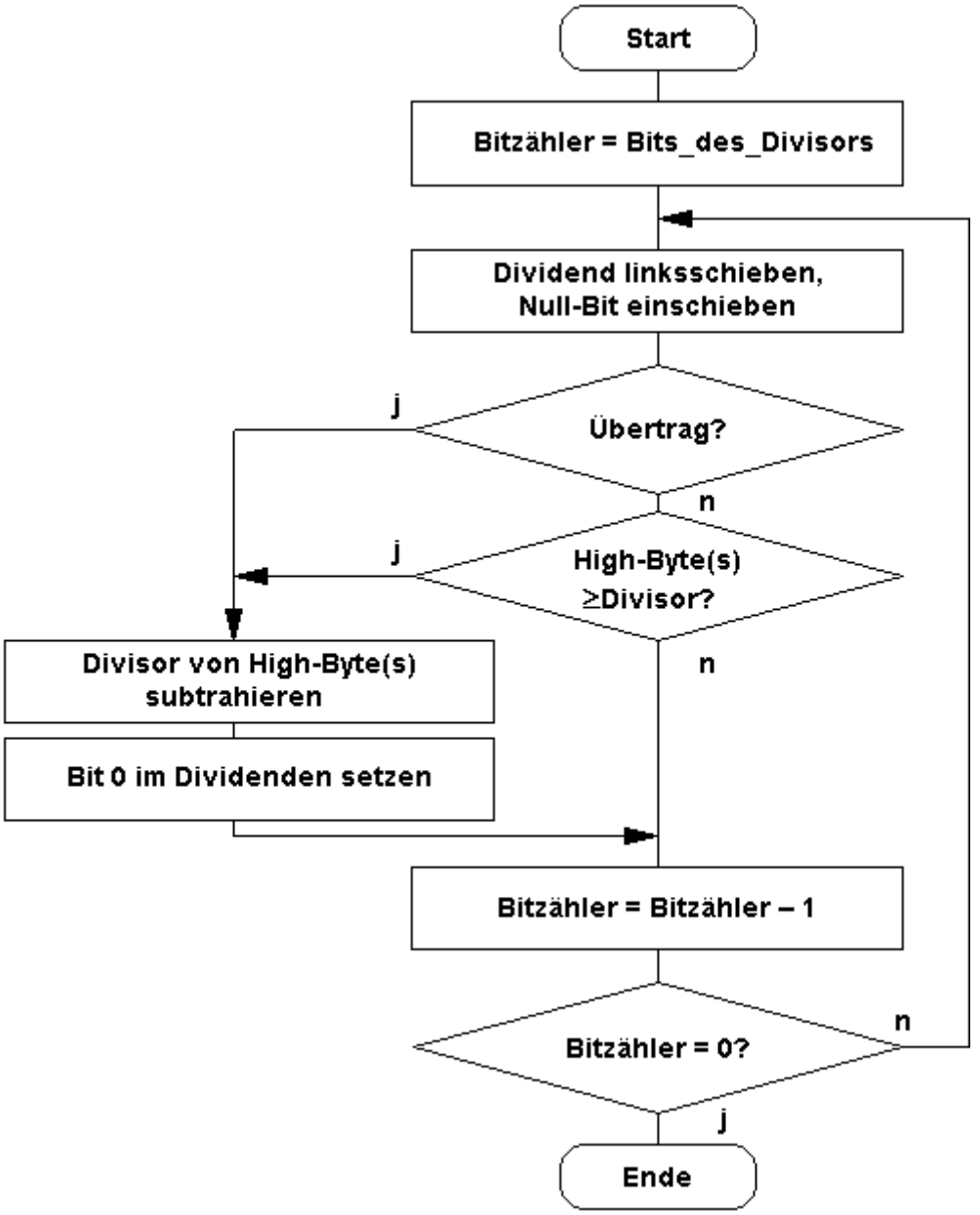
Dividend	3039h	0	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1	
linksschieben	6072h	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1	**0**	kein Übertrag!
Subtraktion testen	60h-43h	0	1	0	0	0	0	1	1									geht! 1
subtrahieren, Bit setzen	6073h	0	0	0	1	1	1	0	1	0	1	1	1	0	0	1	**1**	

linksschieben	3AE6h	0	0	1	1	1	0	1	0	1	1	1	0	0	1	1	**0**	kein Übertrag!	
Subtraktion testen	3Ah-43h	0	1	0	0	0	0	1	1									geht nicht!	0
unverändert lassen	3AE6h	0	0	1	1	1	0	1	0	1	1	1	0	0	1	1	0		
linksschieben	75CCh	0	1	1	1	0	1	0	1	1	1	0	0	1	1	0	**0**	kein Übertrag!	
Subtraktion testen	75h-43h	0	1	0	0	0	0	1	1									geht!	1
subtrahieren, Bit setzen	32CDh	0	0	1	1	0	0	1	0	1	1	0	0	1	1	0	**1**		
linksschieben	659Ah	0	1	1	0	0	1	0	1	1	0	0	1	1	0	1	**0**	kein Übertrag!	
Subtraktion testen	65h-43h	0	1	0	0	0	0	1	1									geht!	1
subtrahieren, Bit setzen	229Bh	0	0	1	0	0	0	1	0	1	0	0	1	1	0	1	**1**		
linksschieben	4536h	0	1	0	0	0	1	0	1	0	0	1	1	0	1	1	**0**	kein Übertrag!	
Subtraktion testen	45h-43h	0	1	0	0	0	0	1	1									geht!	1
subtrahieren, Bit setzen	0237h	0	0	0	0	0	0	1	0	0	0	1	1	0	1	1	**1**		
linksschieben	046Eh	0	0	0	0	0	1	0	0	0	1	1	0	1	1	1	**0**	kein Übertrag!	
Subtraktion testen	04h-43h	0	1	0	0	0	0	1	1									geht nicht!	0
unverändert lassen	046Eh	0	0	0	0	0	1	0	0	0	1	1	0	1	1	1	0		
linksschieben	08DCh	0	0	0	0	1	0	0	0	1	1	0	1	1	1	0	**0**	kein Übertrag!	
Subtraktion testen	08h-43h	0	1	0	0	0	0	1	1									geht nicht!	0
unverändert lassen	08DCh	0	0	0	0	1	0	0	0	1	1	0	1	1	1	0	0		
linksschieben	11B8h	0	0	0	1	0	0	0	1	1	0	1	1	1	0	0	**0**	kein Übertrag!	
Subtraktion testen	11h-43h	0	1	0	0	0	0	1	1									geht nicht!	0
unverändert lassen	11B8h	0	0	0	1	0	0	0	1	1	0	1	1	1	0	0	0		
Ergebnis im Dividenden!!	11B8h	00010001 = 11h = 17								10111000 = 0B8h = 184									

Daraus ergibt sich, dass der Rest stets im High-Teil des Ergebnisses steht, auch beim eingebauten Divisionsbefehl ist das immer so!

Programmablaufplan

Der Algorithmus lässt sich leicht auf längere Dividenden und etwas schwieriger (wegen der langen Subtraktion) auf längere Divisoren übertragen.



Die Umsetzung dieses PAP für mehr als 16-stellige Binärzahlen führt beim AVR zu einigen Problemen. Von den zwei vorhandenen Links-Schiebebefehlen `lsl` und `rol` erweist sich der `rol` als prinzipiell geeignet:

Alle Bits des betreffenden Registers werden um 1 Position nach links geschoben, der vorhandene Wert des Carry-Bits (von SREG) wird in das Bit0 kopiert und das Bit7 wird in das Carry-Bit übertragen, d.h. kopiert.

Beispiel:

Dividend	Reg	R18	R17	R16	Cy=0
	Hex	2	46	A0	
	Bin	00000010	01000110	10100000	= 149152
Divisor Reg		R20	R19		
	Hex	27	10		= 10000
	Bin	00100111	00010000		
(1)	`sub R17, R19`		00110110		Cy=0
	`sbc R18, R20`	11011011			Cy=1
ergäbe negative Zahl, deshalb `sub/sbc` rückgängig, Ergebnis „0“					
(1.1)	`rol R16`			01000000	Cy=1
	`rol R17`		10001101	01000000	Cy=0
	`rol R18`	00000100	10001101	01000000	Cy=0
Zykluszähler dekrementieren					
(2)	`sub R17, R19`		01111101		Cy=0
	`sbc R18, R20`	11011101			Cy=1
ergäbe negative Zahl, deshalb `sub/sbc` rückgängig, Ergebnis „0“					
(2.1)	`rol R16`			10000000	Cy=0
	`rol R17`		00011010		Cy=1
	`rol R18`	00001001			Cy=0
Zykluszähler dekrementieren					
(3)	`sub R17, R19`		00001010		Cy=0
	`sbc R18, R20`	11100010			Cy=1
ergäbe negative Zahl, deshalb `sub/sbc` rückgängig, Ergebnis „0“					
(3.1)	`rol R16`			00000000	Cy=1
	`rol R17`		00110111		Cy=0
	`rol R18`	00010010			Cy=0
Zykluszähler dekrementieren					
(4)	`sub R17, R19`		00100111		Cy=0
	`sbc R18, R20`	00010010			Cy=1
ergäbe negative Zahl, deshalb `sub/sbc` rückgängig, Ergebnis „0“					
(4.1)	`rol R16`			00000000	Cy=1
	`rol R17`		01101010		Cy=0
	`rol R18`	00100100			Cy=0
Zykluszähler dekrementieren					
(5)	`sub R17, R19`		01011010		Cy=0
	`sbc R18, R20`	11111101			Cy=1
ergäbe negative Zahl, deshalb `sub/sbc` rückgängig, Ergebnis „0“					
(5.1)	`rol R16`			00000000	Cy=0
	`rol R17`		11010100		Cy=0
	`rol R18`	01001000			Cy=0
Zykluszähler dekrementieren					
(6)	`sub R17, R19`		11000100		Cy=0
	`sbc R18, R20`	00100001			Cy=0
ergibt diesmal „1“, Cy setzen					
(6.1)	`rol R16`			00000001	Cy=0
	`rol R17`		10001000		Cy=0

```
                rol R18          01000011                        Cy=0
Zykluszähler dekrementieren
(7)             sub R17, R19                 01111000            Cy=0
                sbc R18, R20     00011100                        Cy=0
ergibt diesmal „1“, Cy setzen
(7.1)           rol R16                                 00000011 Cy=0
                rol R17                      11110000            Cy=0
                rol R18          00111000                        Cy=0
Zykluszähler dekrementieren
(8)             sub R17, R19                 11100000            Cy=0
                sbc R18, R20     00010001                        Cy=0
ergibt diesmal „1“, Cy setzen
(8.1)           rol R16                                 00000111 Cy=0
                rol R17                      11000000            Cy=1
                rol R18          00100011                        Cy=0
Zykluszähler dekrementieren
(9)             sub R17, R19                 10110000            Cy=0
                sbc R18, R20     11111100                        Cy=1
ergäbe negative Zahl, deshalb sub/sbc rückgängig, Ergebnis „0“
(9.1)           rol R16                                 00001110 Cy=0
                rol R17                      10000000            Cy=1
                rol R18          01000111                        Cy=0
Zykluszähler dekrementieren
(10)            sub R17, R19                 01110000            Cy=0
                sbc R18, R20     11111100                        Cy=0
ergibt diesmal „1“, Cy setzen
(10.1)          rol R16                                 00011101 Cy=0
                rol R17                      11100000            Cy=0
                rol R18          01000000                        Cy=0
Zykluszähler dekrementieren
(11)            sub R17, R19                 11010000            Cy=0
                sbc R18, R20     00011001                        Cy=0
ergibt diesmal „1“, Cy setzen
(11.1)          rol R16                                 00111011 Cy=0
                rol R17                      10100000            Cy=1
                rol R18          00110011                        Cy=0
Zykluszähler dekrementieren
(12)            sub R17, R19                 10010000            Cy=0
                sbc R18, R20     00001100                        Cy=0
ergibt diesmal „1“, Cy setzen
(12.1)          rol R16                                 01110111 Cy=0
                rol R17                      00100000            Cy=1
                rol R18          00011001                        Cy=0
Zykluszähler dekrementieren
(13)            sub R17, R19                 00010000            Cy=0
                sbc R18, R20     11110010                        Cy=1
ergäbe negative Zahl, deshalb sub/sbc rückgängig, Ergebnis „0“
(13.1)          rol R16                                 11101110 Cy=0
                rol R17                      01000000            Cy=0
                rol R18          00110010                        Cy=0
Zykluszähler dekrementieren
(14)            sub R17, R19                 00110000            Cy=0
                sbc R18, R20     00001011                        Cy=0
ergibt diesmal „1“, Cy setzen
(14.1)          rol R16                                 11011101 Cy=1
```

Weitere Arithmetik-Routinen:
http://www.umnicom.de/Elektronik/Mikrokontroller/Atmel/AtSoft/AtMathe.html#5.1

(weiter im Bericht)

Zu einem Ausweg könnte folgende Überlegung führen:
Wenn der Leistungswert, der mit 24Bit Genauigkeit vorliegt, sowieso durch „10.000“ dividiert werden muß, um eine einfach handhabbare dreistellige Anzeige zu erhalten, muß es auch möglich sein, durch Weglassen des Low-Bytes und einer Division des High-Mid-Bytes durch „39“ (=10.000/256) das gleiche Ergebnis, allerdings mit weniger Genauigkeit zu erhalten. Diese ist hier aber auch nicht notwendig und eine solche Division ist mit der vorhandenen „div16_8.asm“-Routine leicht möglich.

Für die Anzeige des daraus resultierenden Ergebnisses als 3-stellige Zahl liegt bereits eine Routine für die %-Anzeige vor, so dass hier auf weitere Erläuterungen verzichtet werden kann.
Diese Anzeige wird alternativ zur %-Anzeige auf dem LCD dargestellt, d.h. nach Beendigung des Prog-Modus wird die %-Anzeige des aktuellen Kanals verlassen und dafür die Summen-Leistung aller Kanäle angezeigt.

4.7.2. Slave-Seite

Der Slave wird durch verschiedene Sequenzen, die über SPI übertragen werden, gesteuert.

1. Sequenz: Übermittlung der Kanal-Adresse,
zur Vorbereitung und Erkennung der 2. Sequenz wird das T-Flag gesetzt,
R18 pushen (Sequenzzähler) und auf Null setzen

2. Sequenz: Übermittlung des aktuellen Dimmwertes,
Sequenz 2 erkannt: T-Flag ist gesetzt:
Eintragen in das vorher bestimmte Speicherregister R1…R8,
die PWM-Routine arbeitet ab da mit dem neuen Wert;
nach 1ms „Einschwingzeit“ ADU für eine Umsetzung starten,
wenn fertig, den U-Wert vom ADCH im SPDR ablegen,
Sequenzzähler inkrementieren (=1)

3. Sequenz: Übermittlung des gleichen Dimmwertes als „Dummy“,
der im SPDR bereitliegende U-Wert wird über SPI zum Master übertragen,

Sequenz 3 erkannt (Sequenzzähler gleich 2):
ADU umprogrammieren: rechtsbündig für 10Bit-Ausgabe,
ADU für eine Umsetzung starten, warten bis ADU mit Wandelung fertig,
ADCH im SPDR ablegen,
Sequenzzähler inkrementieren

4. Sequenz: Übermittlung des gleichen Dimmwertes als „Dummy“,
der im SPDR bereitliegende IH-Wert wird über SPI zum Master übertragen,
Sequenz 4 erkannt (Sequenzzähler gleich 3):
ADCL im SPDR ablegen,
Sequenzzähler inkrementieren

5. Sequenz: Übermittlung des gleichen Dimmwertes als „Dummy“,
der im SPDR bereitliegende IL-Wert wird über SPI zum Master übertragen,
Sequenzzähler zurücksetzen,
T-Flag löschen

Diese Verfahrensweise – jede Sequenz einzeln mit der SPI-ISR zu bearbeiten, hat sich nicht als sicher erwiesen. Die T-Flag-Steuerung kam bei der Übermittlung von „0“-Daten durcheinander. „0“-Daten gibt es sowohl als Adresse, dann ist es das Sleep-Signal, als auch für den Dimmwert. Wird in der korrekten Reihenfolge eine Sequenz nicht erkannt, kann das zu Fehlinterpretationen führen.
Deshalb wird in der neuen Variante der SPI-ISR_3.asm jedes Mal die 5 Sequenzen komplett verarbeitet. Die Sequenzen werden vom Master mit einer 1ms-Folge gesendet, so dass eigentlich genügend Zeit bleiben sollte, die Abarbeitung vollständig zu vollziehen. Da zwischen Master und Slave kein Hand-Shaking besteht, ist das Timing kritisch. Trotzdem könnte es noch Optimierungs-Reseven geben.
Versuche, eine Timing-Optimierung durch jeweils einen Warteschleifen-Zyklus bis zur Fertigstellung der SPI-Übertragung durchzuführen, hatten dann unsichere Übertragungs-Zyklen zur Folge. Der Grund ist wiederum, dass der Warteschleifen-Zyklus zwar feststellt, ob das Sender-/Empfänger-SPI mit der Übertragung fertig ist, aber wiederum keine Aussage für die Bereitschaft der Gegenstelle, die danach anstehenden Daten schon wieder verarbeiten zu können.

Abhilfe könnte hier evtl. ein probeweises Senden eines Prüfbytes sein, welches über den Umweg eines Registertransfers, z.B. invertiert zurückgesendet wird und somit vom Master als „Slave ist zum Datenempfang bereit“ ausgewertet werden kann.
Die Alternative mit einer obligatorischen z.B. 1ms-Warteschleife ist bei nicht so zeitkritischen Anwendungen jedoch einfacher (wird hier angewendet).
Die 1ms-Warteschleifen werden in allen 5 SPI-Übertragungs-Zyklen bei allen SPI-Übertragungsfunktionen eingesetzt:

- beim Start zur Übertragung der Start-Dimmwerte (EEPROM) vom Master zum Slave
- bei der Übertragung neuer Dimmwerte im ProgModus
- bei der Übertragung abweichender Dimmwerte im AnalogModus

Die neue SPI-ISR sieht dann so aus:

1. Sequenz: Übermittlung der Kanal-Adresse,
falls „Null“ übermittelt wurde -> Sleep-Modus

2. Sequenz: Übermittlung des aktuellen Dimmwertes,
Eintragen in das vorher bestimmte Speicherregister R1…R8,
die PWM-Routine arbeitet ab da mit dem neuen Wert;
nach 1ms „Einschwingzeit“ ADU für eine Umsetzung starten,
wenn fertig, den U-Wert vom ADCH im SPDR ablegen,

3. Sequenz: Übermittlung des gleichen Dimmwertes als „Dummy“,
der im SPDR bereitliegende U-Wert wird gleichzeitig über SPI zum Master übertragen, ADU umprogrammieren: rechtsbündig für 10Bit-Ausgabe,
ADU für eine Umsetzung starten, warten bis ADU mit Wandelung fertig,
warten bis SPI 3.Sequenz fertig, dann ADCH im SPDR ablegen,

4. Sequenz: Übermittlung des gleichen Dimmwertes als „Dummy“,
der im SPDR bereitliegende IH-Wert wird über SPI zum Master übertragen,
warten bis SPI 4.Sequenz fertig, dann ADCL im SPDR ablegen,

5. Sequenz: Übermittlung des gleichen Dimmwertes als „Dummy“,
der im SPDR bereitliegende IL-Wert wird über SPI zum Master übertragen,

Mit dieser neuen SPI-Kommunikation ergibt sich auch die Frage nach der Sinnfälligkeit der doppelten EEPROM-Ablage der Dimmwerte im Master und im Slave.
Da geplant ist, die Stromversorgung des Masters über das Kabel (normales 8-pol.-Netzwerkkabel) für die SPI-Kommunikation zu führen, ist es ausreichend, die bei Stromausfall oder RESET gesicherten Dimmwerte nur im Master-EEPROM zu halten und bei jedem Neustart als Paket über die SPI-Kommunikation in den Registersatz R1…R8 des Slaves neu zu schreiben.

Der Slave-EEPROM wird nicht mehr benutzt.

5. Programm-Details

5.1. LCD-CGRAM mit Sonderzeichen belegen

(dazu folgende Übersetzung des „Optex-Dmcman_full“, S22f)

Der CG-RAM ist ein 64 x 8 Bit RAM, in dem der Benutzer Muster selbstdefinierter Zeichen programmieren kann. Entweder können Muster von 8 - 5 x 7 Zeichen oder 4 - 5 x 10 Zeichen-Muster geschrieben werden.

Zum Schreiben der zuvor im CG-RAM programmierten Zeichen in den DD-RAM werden die Zeichencodes 0H bis 7H verwendet. (Siehe Schriftart Tabelle 5.3)

Die Beziehungen zwischen den CG-RAM-Adresse und den Daten der angezeigten Zeichen ist in den Tabellen 5.1 und 5.2 dargestellt.

Die ungenutzte CG RAM-Standorte können als RAM für allgemeine Zwecke verwendet werden.

Zum Programmieren eines Muster von 5 x 8 Zeichen an eine bestimmte Position im CG-RAM # 2 sollten die folgenden Schritte ergriffen werden:

A. Verwenden Sie den "Set CG RAM-Adresse"-Befehl zum Positionieren des CG-RAM-Zeiger auf die n-te Zeile von Zeichen # 2.

RS	R/W	DB_7	DB_6		. . .			DB_1	DB_0
0	0	0	1	0	0	1	Y	Y	Y
Set CG RAM Address Command				Character Location #2			n_{th} row of character #2 (initially YYY=000)		

B. Verwenden Sie den "Schreiben von Daten auf CG- oder DD-RAM"-Befehl zum Schreiben der obersten Zeile der benutzerdefinierten Charakter.

RS	R/W	DB_7	DB_6		. . .			DB_1	DB_0
1	0	x	x	x	y	y	y	y	y
Write Data Command		Don’t Care			5 bits of character bit pattern for row n of 7.				

C. Der YYY-Teil der "Set CG RAM-Adresse"-Befehl in Schritt 1 wird automatisch erhöht, wenn das I / O-Bit im "Entry Mode Set"-Befehl gesetzt ist. Wenn dies der Fall ist, zurück zu Schritt 2, bis alle Zeilen des Charakters geschrieben sind.

D. Nachdem alle 7 Reihen von Daten eingeschrieben sind, verwenden Sie den "Set DD RAM-Adresse"-Befehl, um wieder den Adress-Zähler auf einen neuen Standort im DD-RAM zu setzten.

E. Zum Anzeigen der benutzerdefinierte Zeichen wie oben beschrieben, benutzt das "Schreiben von Daten auf den CG- oder DD-RAM"-Befehl. Das Schreiben von 02H-Daten wird benutzt zum Anzeigen der Zeichen der DD-RAM-Adressen, wie in Schritt 4. (??)

Anmerkungen:

1. Zeichencode Bits 0 ~ 2 entsprechen den CG RAM-Adressen Bit 3 ~ 5. (3 Bits: 8 Typen)

2. CG RAM-Adresse Bit 0 ~ 2 bezeichnet die Stellung der Linie in einem Zeichen-Muster. Die 8. Zeile ist die Cursor-Position und Anzeige wird bestimmt durch ein logisches Oder der 8. Zeile mit dem Cursor. Die Unterstützung der Zeilen-Daten, die entsprechend den Cursor- Position angezeigt werden sollen, werden nur im "0"-Status der Cursor-Anzeige dargestellt. Wenn von den Daten der 8. Zeile Bit 1 ist "1" leuchtet auf, und zwar unabhängig von der Existenz des Cursors.

3. Die Zeichen-Muster der Zeilen-Positionen entsprechen den CG-RAM-Datenbits 0 ~ 4 wie in der Abbildung gezeigt (Bit 4 ist am Ende links).
Wenn CG-RAM-Datenbits 5 ~ 7 nicht zur Anzeige benötigt werden, können sie für die allgemeinen Daten verwendet werden, wie als ob ein RAM-Speicher noch existiert.

4. Wie in Tabelle 5.1 gezeigt, werden die Charakter Muster im CG-RAM ausgewählt, wenn Zeichencode-Bits 4 ~ 7 sind alle "0" sind.
Jedoch als Zeichencode Bit-3 ist untauglich das "R" anzuzeigen im Beispiel des Zeichen-Musters, wenn der Zeichencode "00" (hexadezimal) oder "08" (hexadezimal) ausgewählt ist. (??)

5. "1" für CG-RAM-Daten entspricht ausgewählte Pixel und "0" ist für nicht ausgewählte.

5.1.1. Im LCD abgelegten Sonderzeichen

Zur Darstellung der Balkendiagramme für die Dimmwerte der 8 Kanäle werden auf den benutzerdefinierbaren Zeichenadressen 0 … 7 folgende Sonderzeichen abgelegt:

Adr.	0	1	2	3	4	5	6	7
	----	---- ----	---- ---- ----	---- ---- ---- ----	---- ---- ---- ---- ----	---- ---- ---- ---- ---- ----	---- ---- ---- ---- ---- ---- ----	---- ---- ---- ---- ---- ---- ---- ----

5.2. LCD-Ansteuerung im 4-Bit-Modus

Beim Anschalten der Betriebsspannung erfolgt als interne Routine im LCD eine Initialisierungs-Sequenz. Nach ca. 10ms ist das LCD in der Standard-Initialisierung wie folgt betriebsbereit:

(1) Display Clear
(2) Function Set
- DL = 1 : 8-bit interface data
- N = 0 : 1-line display
- F = 0 : 5x7-dot character font

(3) Display On/Off Control
- D = 0 : Display Off
- C = 0 : Cursor Off
- B = 0 : Blink Off

(4) Entry Mode Set
- I/D = 1 : +1 (Increment)
- S = 0 : No Shift

Während der internen Initialisierung bleibt das Busy-Flag im „High“-Zustand.

Um Port-Ausgänge zu sparen ist es möglich, das LCD in den sog. 4-Bit-Modus umzuschalten. Vorraussetzung ist jedoch, dass die Datenbits d0…d3 hardwaremäßig auf Masse liegen, d.h. dort dauerhaft „Low“ anliegt. Nur dann ist es möglich mit dem Befehl SET-FUNCTION

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
L	L	L	L	H	DL	N	F	X	X

Wichtig ist hierbei der Zustand von DB4: →Low = 4-Bit-Modus, die Belegung von DB0…DB3 ist hierbei egal, jedoch hardwaremäßig durch „Low“ vorgegeben. Dieser Befehl wird noch im grundinitialisierten 8-Bit-Modus verarbeitet. Ab diesem Zeitpunkt

sind dann alle weiteren Befehl-/Datentransfers nur noch im 4-Bit-Modus (erst High-, dann Low-Nibble) vorzunehmen.
Vielfach in Beispielprogrammen – und auch die im Datenblatt - angegebenen Vorsequenzen mit mehrfachen Ausgabe von

0 0 0 0 1 1 * * * *

hat sich als nicht notwendig erwiesen.
Nach einer Wartezeit von mind. 15ms nach dem Einschalten kann sofort der o.g. Umschaltbefehl gegeben werden und danach sind die speziellen Einstellungen zum Zeichensatz, Zeilenzahl, Cursor usw. bereits im 4-Bit-Modus möglich.

In Beispielprogrammen wird vielfach auf die Auswertung des `Busy`-Flags bei der Kommunikation mit dem LCD verzichtet. Dann sind dort umfangreiche Warte-Zyklen eingebaut, die sicherstellen, dass die LCD-interne Verarbeitung beendet ist, bevor ein neuer Befehl/Daten gesendet werden. Solche Warte-Zyklen sind Ressourcen-fressend und bei zeitkritischen Applikationen nicht zu empfehlen.

Eine Variante mit BF-Auswertung ist noch nicht erarbeitet bzw. noch nicht getestet.

5.3. Bildwiederholspeicher des LCD im RAM

Der Bildwiederholspeicher für das LCD ist im SRAM des AVR ab $100 definiert.
Im Gegensatz zum LCD-internem RAM ist der BWS linear aufgebaut, d.h.:

RAM-Adr.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
LCD-Adr.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
	P	S	–	T	e	c	h	.		(	B	a	l	k	e	n	)			

RAM-Adr.	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27
LCD-Adr.	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53
	n	C	h	-	D	i	m	m	e	r		(	B	a	l	k	e	n	)	

RAM-Adr.	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B
LCD-Adr.	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27
	C	h	a	n	n	e	l	:		(1)		(	B	a	l	k	e	n	)	

RAM-Adr.	3C	3D	3E	3F	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
LCD-Adr.	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	60	61	62	63	64	65	66	67
	1	0	0	%								1	2	3	4	5	6	7	8	

5.4. RESET und Interruptvektoren beim AVR128

Zur Programmierung des AVR128 sind im Gegensatz zu den „kleineren“ Devices einige Besonderheiten zu beachten.
Zunächst läßt sich wegen der möglichen Implementierung eines Boot-Bereiches im Programmspeicher der RESET-Einsprungpunkt verschieben. Die dazu notwendigen Einstellungen werden mit den Fuses festgelegt:

`BOOTRST (1)` = unprogrammiert, d.h. RESET liegt bei $0000
`(0)` = gesetzt, würde RESET je nach Einstellung von `BOOTSZ0` und `BOOTSZ1` am Anfang des Bootbereichs liegen

Wichtig ist außerdem, dass der AVR103-Kompatibilitätsmodus ausgeschaltet wird, d.h. die Fuses `M103C` muß von „0“ (Auslieferungszustand) auf „1“ gesetzt werden.
(siehe Tabellen 117 – 119 auf Seite 287ff des AVR128-Datasheet)

Der Sprungvektor für RESET muß - im Gegensatz zu den kleineren AVRs, wo ein „`rjump`“ verwendet wird - mit einem `JMP` erfolgen. Ansonsten sind die nachfolgenden Interruptvektoren nicht korrekt. Werden diese benutzt, sind auch sie mit einem `JMP` auszuführen. Werden sie nicht benutzt, ist an dieser Stelle wie üblich ein „`reti`“ einzutragen, gefolgt von einem „`nop`“. Offensichtlich ist die Interruptvektor-Tabelle in Abweichung vom sonst Üblichen als Doppelwort-Struktur aufgebaut. Wird dass nicht beachtet, verweisen Interrupt-Anforderungen auf falsche, d.h. ungültige Interruptvektoren und somit falsche bzw. gar keine Interrupt-Service-Routinen (ISR).

5.5. Relativ bedingte Sprünge

Durch die Einbindung von Programmteilen mittels der `.include`-Anweisung - aber auch sonst - kann es vorkommen, dass relativ bedingte Sprünge, wie z.B. `breq, brne` usw. mit ihrer maximalen Sprungdistance von +63/-64 nicht ausreichen.
Die Lösung ist eine in der Nähe zu platzierende Zwischenmarke, die dann mit einem `jmp` zur wirklichen Sprungadresse verlassen werden kann. Diese Zwischenmarke darf im normalen Programmablauf nicht erreicht werden, deshalb ist diese zweckmäßigerweise gleich nach einem `rjmp` des normalen Programms anzuordnen.

5.5.1 Vergleiche – Sprungbedingungen (Übersicht)

Ausgehend von der Vergleichsabfrage `cp Rd, Rr` ergeben sich folgende Sprungbedingungen:

a) für vorzeichenlose Zahlen

Abfrage	**Boolean**	**Mnemonic**	**Komplementär**	**Boolean**	**Mnemonic**	**Bemerk.**
Rd > Rr	C+Z = 0	`Brlo` (1)	Rd ≤ Rr	C+Z = 1	`Brsh` (1)	Vz-los
Rd ≥ Rr	C = 0	`Brsh/Brcc`	Rd < Rr	C = 1	`Brlo/Brcs`	Vz-los
Rd = Rr	Z = 1	`Breq`	Rd ≠ Rr	Z = 0	`Brne`	Vz-los
Rd ≤ Rr	C+Z = 1	`Brsh` (1)	Rd > Rr	C+Z = 0	`Brlo` (1)	Vz-los
Rd < Rr	C = 1	`Brlo/Brcs`	Rd ≥ Rr	C = 0	`Brsh/Brcc`	Vz-los

b) für Zahlen mit Vorzeichen (2er Komplement)

Abfrage	**Boolean**	**Mnemonic**	**Komplementär**	**Boolean**	**Mnemonic**	**Bemerk.**
Rd > Rr	Z●(N⊕V)=0	`Brlt` (1)	Rd ≤ Rr	Z●(N⊕V)=1	`Brge` (1)	mit Vz
Rd ≥ Rr	(N⊕V) = 0	`Brge`	Rd < Rr	(N⊕V) = 1	`Brlt`	mit Vz
Rd = Rr	Z = 1	`Breq`	Rd ≠ Rr	Z = 0	`Brne`	mit Vz
Rd ≤ Rr	Z+(N⊕V)=1	`Brge` (1)	Rd > Rr	Z●(N⊕V)=0	`Brlt` (1)	mit Vz
Rd < Rr	(N⊕V) = 1	`Brlt`	Rd ≥ Rr	(N⊕V) = 0	`Brge`	mit Vz

Sprungbedingungen können auch wie folgt definiert werden:

Statusbit gesetzt	BRBS b7, k127
Statusbit rückgesetzt	BRBC b7, k127
Springe bei gleich	BREQ k127
Springe bei ungleich	BRNE k127
Springe bei Überlauf	BRCS k127
Springe bei Carry=0	BRCC k127
Springe bei gleich oder größer	BRSH k127
Springe bei kleiner	BRLO k127
Springe bei negativ	BRMI k127
Springe bei positiv	BRPL k127
Springe bei größer oder gleich (Vorzeichen)	BRGE k127
Springe bei kleiner Null (Vorzeichen)	BRLT k127
Springe bei Halbübertrag	BRHS k127
Springe bei HalfCarry=0	BRHC k127
Springe bei gesetztem T-Bit	BRTS k127

Springe bei gelöschtem T-Bit	BRTC k127	
Springe bei Zweierkomplementüberlauf	BRVS k127	
Springe bei Zweierkomplement-Flag=0	BRVC k127	
Springe bei Interrupts eingeschaltet	BRIE k127	
Springe bei Interrupts ausgeschaltet	BRID k127	

Bedingte Sprünge zum übernächsten Befehl

Registerbit=0	SBRC r1, b7	
Registerbit=1	SBRS r1, b7	
Portbit=0	SBIC pl, b7	
Portbit=1	SBIS pl, b7	
Vergleiche &, Sprung bei gleich	CPSE r1, r2	

Quellen für Sprungbedingungen

Addition

8 Bit, +1	INC r1	Z N V
8 Bit	ADD r1, r2	Z C N V H
8 Bit+Carry	ADC r1,r2	Z C N V H
16 Bit, Konstante	ADIW rd, k63	Z C N V S

Subtraktion

8 Bit, -1	DEC r1	Z N V
8 Bit	SUB r1, r2	Z C N V H
8 Bit, Konstante	SUBI rh, k255	Z C N V H
8 Bit - Carry	SBC r1, r2	Z C N V H
8 Bit - Carry, Konstante	SBCI rh, k255	Z C N V H
16 Bit	SBIW rd, k63	Z C N V S

Schieben

Logisch, links	LSL r1	Z C N V
Logisch, rechts	LSR r1	Z C N V
Rotieren, links über Carry	ROL r1	Z C N V
Rotieren, rechts über Carry	ROR r1	Z C N V
Arithmetisch, rechts	ASR r1	Z C N V
Nibble-Tausch	SWAP r1	

Binär

Und	AND r1, r2	Z N V
Und, Konstante	ANDI rh, k255	Z N V
Oder	OR r1,r2	Z N V
Oder, Konstante	ORI rh, k255	Z N V
Exklusiv-Oder	EOR r1, r2	Z N V
Einer-Komplement	COM r1	Z C N V
Zweier-Komplement	NEG r1	Z C N V H

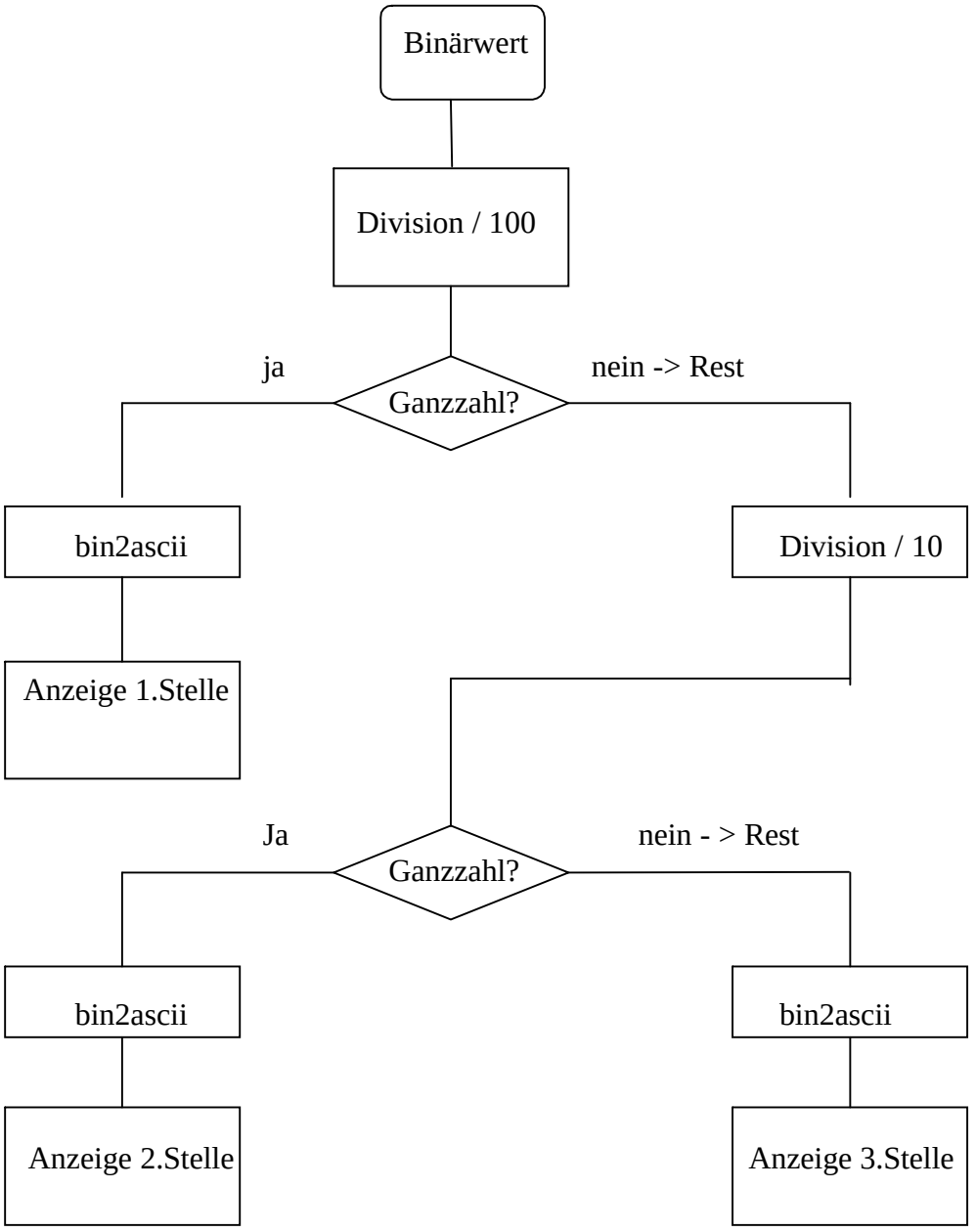
Vergleiche

Register, Register	CP r1, r2	Z C N V H
Register, Register + Carry	CPC r1, r2	Z C N V H
Register, Konstante	CPI rh, k255	Z C N V H
Register = 0	TST r1	Z N V

5.6. Darstellung einer Binärzahl als 3-stellige Dezimal-Ausgabe

Der Dimmwert des aktuell angezeigten Kanals soll als Prozentwert angezeigt werden. Die im Registersatz R1…R8 abgelegten Dimmwerte liegen aber als Binärwert in einem Bereich von 0…$FF (Hex), bzw. 0…255 (dez) vor.

Der Binärwert muß entsprechend umgewandelt werden, dazu folgendes Flussdiagramm:



Bei Ausführung dieser Routine würde aber lediglich der binäre Dimmwert 0…255 angezeigt werden, nicht jedoch der verlangte %-Wert. Deshalb muß der binäre Dimmwert zunächst mit 100 multipliziert und dann durch 255 dividiert werden. Andersherum würden sich Dezimalbrüche ergeben, die nicht mit Ganzzahl-Division realisiert werden können.

5.7. SPI - Grundlagen

(dazu die Übersetzung der Atmel-Application-Note AVR151)

5.7.1. Setup und Anwendung des SPI

Merkmale

• SPI-Pin-Funktionalität

• Multi-Slave-Systeme

• SPI-Timing

• SPI Übermittlung Konflikte

• Emulation der SPI

• Code-Beispiele für befragten Betrieb

• Code-Beispiele für Interrupt Betrieb

1. Einführung

Diese Application-Note beschreibt die Installation, Konfiguration und Anwendung des On-Chip Serial Peripheral Interface (SPI) der AVR-Micro-Controller. Die meisten AVR-Geräte werden mit einem SPI an Bord hergestellt, welches nach diesem Dokument individuell konfiguriert werden kann. Nach einem theoretischen Hintergrund wird gezeigt, wie die Konfiguration des SPI, um in den beiden Master-und Slave-Modus-Modus

Figure 1. Master and Slave Interface.

(Seite 2)

5.7.2. Allgemeine Beschreibung der SPI

Das SPI ermöglicht eine High-Speed-synchrone Datenübertragung zwischen dem AVR und Peripheriegeräten, oder zwischen mehreren AVR-Geräten. Auf den meisten Gräten hat das SPI einen zweiten Zweck, wenn es für In-System Programming (ISP) benutzt wird. Für Details siehe dazu die Application Note AVR910.

Die Zusammenschaltung zwischen zwei SPI-Geräten passiert immer zwischen einem Master-Gerät und einem Slave-Gerät. Im Vergleich zu einigen Peripherie-Geräte, wie z.B. Sensoren, die nur in Slave-Modus betrieben werden können, kann der SPI des AVR für beide, den Master- und Slave-Modus eingesetzt werden. Der Modus, unter welchem der AVR läuft, wird durch die Einstellungen des Master-Bit (MSTR) im SPI-Control-Register (SPCR) festgelegt. Besondere Erwägungen müssen bei dem /SS-Pin berücksichtigt werden. Dies wird

beschrieben im Abschnitt "Multi-Slave-Systeme - /SS Pin-Funktionalität" auf Seite 3. Der Master ist der aktive Teil in diesem System und generiert ein Takt-Signal, worauf eine serielle Daten-Übertragung basiert. Der Slave ist nicht in der Lage ein Taktsignal zu generieren und kann daher nicht selbst aktiv werden. Der Slave sendet und empfängt Daten nur, wenn der Master das notwendige Taktsignal generiert. Der Master generiert das Taktsignal jedoch nur beim Senden von Daten. Das bedeutet, dass der Master Daten zum Slave senden muss, um Daten vom Slave lesen zu können.

Hinweis:
Dies kann verwirrend sein, vor allem, wenn "passive" Peripheriegeräte, wie Sensoren verwendet werden. Die Notwendigkeit, zufälligen Daten an einen Sensor senden zu müssen, nur um seine Daten zu lesen, ist nicht immer klar.

5.7.3. Die Datenübertragung zwischen Master und Slave

Die Interaktion zwischen einem Master- und einem Slave-AVR ist in Abbildung 1 auf Seite 1 gezeigt. Zwei identische SPI-Einheiten sind dargestellt. Das linke Gerät ist als Master konfiguriert, während das rechte Gerät als Slave konfiguriert ist.
Die MISO-, MOSI- und SCK-Leitungen sind mit den entsprechenden Leitungen der anderen Seite verbunden. Der Modus, in dem ein Teil ausgeführt wird, bestimmt, welche Signalleitungen Eingang oder Ausgang sind. Da gleichzeitig ein Bit vom Master auf den Slave und vom Slave zum Master in einem Taktzyklus beider 8-Bit-Shift Register verschoben wird, kann man das auch als ein 16-Bit-Ringschieberegister betrachten. Dies bedeutet, dass nach acht SCK-Taktimpulsen die Daten zwischen Master und Slave ausgetauscht werden.

Das System ist in die Sende-Richtung einfach gepuffert und in Empfangs-Richtung doppelt gepuffert. Dies beeinflusst das Daten-Handling auf folgendem Wege:

1. Neue Bytes, die gesendet werden sollen, können nicht in das Daten-Register (SPDR) / Shift Register geschrieben werden, bevor der gesamte Schiebe-Zyklus abgeschlossen ist.
2. Empfangene Bytes werden in den Empfänger-Puffer geschrieben, unmittelbar nach dem die Übertragung abgeschlossen ist.
3. Der Empfänger-Puffer ist zu lesen, bevor die nächste Übertragung beendet ist oder Daten gehen verloren.

4. Das Lesen des SPDR gibt die Daten des Empfänger-Puffer wieder.

Nachdem eine Übertragung abgeschlossen ist, wird das SPI-Interrupt Flag (SPIF) im SPI Status Register (SPSR) gesetzt. Dies führt dazu, dass die entsprechende ISR ausgeführt wird, wenn dieser Interrupt und die Interrupts globale aktiviert sind. Das Setzen des SPI-Interrupt-Enable (SPIE) Bit im SPCR aktiviert den SPI-Interrupt, solange das I-Bit im SREG globale Interrupts ermöglicht.

(Seite 3)

5.7.4. Pins des SPI

Das SPI besteht aus vier verschiedenen Signalleitungen. Diese Leitungen sind:
die Takt-Leitung (SCK),
die Master Out Slave In Leitung (MOSI),
die Master-In-Slave-Out Leitung (MISO) und
die Low-aktive Slave Auswahl Leitung (/SS).
Wenn das SPI aktiviert ist, werden die Daten in Richtung des MOSI, MISO, SCK und SS-Pins entsprechend der folgenden Tabelle überschrieben.

Table 2-1. SPI Pin overrides

Pin	Direction Master Mode	Direction Slave Mode
MOSI	User Defined	Input
MISO	Input	User Defined
SCK	User Defined	Input
SS	User Defined	Input

Diese Tabelle zeigt, dass nur die Input-Pins automatisch konfiguriert werden. Die Output-Pins müssen manuell durch die Software initialisiert werden. Der Grund dafür ist, Schäden zu vermeiden, z. B. durch Treiber-Festlegungen.

5.7.5. Multi-Slave-Systeme - /SS Pin-Funktionalität

Das Slave-Auswahl-Pin (/SS) spielt eine zentrale Rolle bei der SPI-Konfiguration. Je nach Modus, in welchem das Teil läuft und der Konfiguration dieses Pins, kann sie zum Aktivieren oder zu Deaktivieren der Geräte verwendet werden. Das /SS-Pin kann mit einem Chip-Select-Pin verglichen werden, während es noch einige zusätzliche Features hat.

Im Master-Modus muss das /SS-Pin auf High gehalten werden, um den SPI Master-Betrieb sicherzustellen, wenn dieses Pin als Input Pin konfiguriert ist. Ein Low-Niveau schaltet in den SPI-Slave-Modus um und die SPI-Hardware wird folgenden Aktionen

durchführen:

1. Die Master-Bit (MSTR) in der SPI Control Register (SPCR) wird gelöscht, und das SPI-System wird zu einem Slave. Die Richtung der Pins wird umgeschaltet nach Tabelle 2-1.
   2. Die SPI-Interrupt Flag (SPIF) im SPI-Status Register (SPSR) wird gesetzt. Wenn der SPI-Interrupt und Interrupts global aktiviert sind, wird die entsprechende Interrupt-Routine ausgeführt.
   Dies kann in Systemen mit mehr als einem Master nützlich sein zur Vermeidung, dass zwei Meister zur gleichen Zeit den Zugriff auf den SPI-Bus bekommen. Wenn das /SS-Pin als Output-Pin konfiguriert ist, kann es als ein allgemein verwendbares Output-Pin verwendet werden, welches keinen Effekt im SPI-System hat.

2. Hinweis:

In Fällen, in denen der AVR für den Master-Modus konfiguriert ist und es kann nicht gewährleistet werden, dass das /SS-Pin zwischen zwei Übertragungen High bleibt, muss der Status der MSTR-Bit geprüft werden, bevor ein neues Byte geschrieben wird. Sobald das MSTR-Bit gelöscht wurde von einem Low auf der /SS-Leitung, muss erst der SPI-Master-Modus durch die Anwendung wieder aktivieren werden.

Im Slave-Modus ist das /SS-Pin ist immer ein Eingang. Wenn /SS = Low wird, ist das SPI aktiviert und MISO wird Ausgang, wenn das durch den Benutzer so konfiguriert worden ist. Alle anderen Pins sind Eingänge. Bei der /SS = High sind alle Pins Eingänge und das SPI ist passiv, was bedeutet, dass keine eingehenden Daten erhalten werden. Tabelle 2-2 zeigt einen Überblick über die /SS-Pin-Funktionalität.

Hinweis:

Im Slave-Modus wird ein Reset der SPI-Logik veranlasst, sobald das /SS-Pin auf High gebracht wird. Wenn das /SS-Pin während einer Übertragung auf High gebracht wird, stoppt das SPI sofort das Senden und Empfangen und beide Daten, die empfangenen und gesendeten müssen als verloren angesehen werden.

(Seite 4)

Table 2-2. Overview of the SS pin functionality

Mode	/SS-Configuration	/SS-Pin-level	Description
Slave	Always Input	High	Slave deactivated (deselected)
		Low	Slave activated (selected)
Master	Input	High	Master activated (selected)
		Low	Master deactivated, switched to slave mode
	Output	High	Master activated (selected)
		Low	(“ “)

Wie in Tabelle 2-2 gezeigt, ist der /SS-Pin im Slave-Modus immer ein Eingangs-Pin. Ein Low aktiviert das SPI des Geräts, während ein High ihre Deaktivierung verursacht. Ein Single Master Multiple Slave-System mit AVR und im Master-Modus konfiguriert mit /SS als Output-Pin ist in Abbildung 2-1 dargestellt. Die Anzahl der Slaves, die an diesem AVR angeschlossen werden können, ist nur begrenzt durch die Anzahl der I/O-Pins zum Generieren der Slave-Auswahlsignale.

Figure 2-1. Multi Slave System

Die Fähigkeit zum Verbindung mehrerer Geräte auf dem gleichen SPI-Bus basiert auf der Tatsache, dass nur ein Master- und nur ein Slave gleichzeitig aktiv sein können. Die MISO-, MOSI- und SCK-Leitungen aller anderen Slaves sind „TriStated“ (konfiguriert als Input-Pins mit hoher Impedanz ohne aktivierte Pullup-Widerständen). Eine falscher Anwendung (z. B. wenn zwei Slaves zur gleichen Zeit aktiviert sind) kann dazu führen, dass ein Treiber-Konflikt zum Zustand eines CMOS-Latchup führen kann und dass muss vermieden werden. Widerstände von 1 bis 10 k Ohm in Reihe mit den Pins des SPI können verwendet werden, um ein System-LatchUp zu verhindern. Jedoch wirkt sich dieses auf die maximal nutzbare Datenrate aus, die abhängig ist von der Last-Kapazität an den SPI-Pins.

Unidirektionale SPI-Geräte benötigen nur die Takt-Leitung und eine der Datenleitungen. Ob das Gerät mit Hilfe der MISO- oder die MOSI-Leitung arbeitet, hängt von ihren Zweck ab. Einfache Sensoren zum Beispiel senden nur Daten (siehe S2 in Abbildung 2-1), während ein externer DAC in der Regel nur Daten empfängt (siehe S3 in Abbildung 2-1).

(Seite 5)

5.7.6. SPI-Timing

Das SPI hat vier Betriebsarten, 0 bis 3. Diese Modi bestimmen im Wesentlichen die Kontrolle der Art, wie Daten in oder aus einem SPI-Gerät getaktet werden. Die

Konfiguration erfolgt durch zwei Bits im SPI Control Register (SPCR). Die Takt-Polarität wird durch das CPOL Kontroll-Bit ausgewählt, entweder ein High-aktiven, oder Low-aktiven Takt. Das Takt Phase (CPHA) Kontroll-Bit wählt zwischen zwei grundlegend verschiedenen Formaten aus. Für eine reibungslose Kommunikation zwischen Master und Slave müssen beide Geräte den gleichen Modus haben. Dies kann eine Rekonfiguration des Masters zur Folge haben, um den Anforderungen verschiedener peripherer Slaves gerecht zu werden.
Die Einstellungen der CPOL und CPHA für die verschiedenen SPI-Modi ist in Tabelle 2-3 angegeben. `Weil`
Da dies kein Standard ist und unterschiedlich in anderer Literatur spezifiziert wird, sollte die Konfiguration der SPI sorgfältig geschehen.

Table 2-3. SPI Mode Configuration

SPI Mode	CPOL	CPHA	Shift SCK-edge	Capture SCK-edge
0	0	0	Falling	Rising
1	0	1	Rising	Falling
2	1	0	Rising	Falling
3	1	1	Falling	Rising

Die Takt-Polarität hat keine erheblichen Auswirkungen auf das Transfer-Format. Ein Umschalten dieses Bit verursacht, dass das Taktsignal invertiert wird (High-aktiv wird Low-aktiv und Leerlauf-Low wird Leerlauf-High). Jedoch wählt die Einstellungen der Takt-Phase eine von zwei verschiedenen Übertragungs-Zeitverläufe, die in den nächsten zwei Kapiteln näher beschrieben. Wenn die MOSI- und MISO-Leitungen des Masters und des Slave direkt miteinander verbunden sind, zeigen die Diagramme den Zeitplan für beide Geräte, den Master und den Slave. Die /SS-Leitung ist der Slave Auswahl-Eingang für den Slave. Das /SS-Pin des Masters ist in den Diagrammen nicht angegeben. Es ist inaktiv durch einen High-Pegel an diesem Pin (falls konfiguriert als Input-Pin) oder durch eine Konfiguration als Ausgangs-Pin.

CPHA = 0, CPOL = 0 (Mode 0); CPHA = 0, CPOL = 1 (Mode 1)

Die Wahl des Zeitpunkts für eine SPI-Übertragung, wo CPHA=0 ist, wird in Abbildung 2-2 gezeigt. Zwei Wellen-Formen werden für das SCK-Signal gezeigt - eine für CPOL=0 und eine weitere für CPOL=1.

(Seite 6)
Figure 2-2. SPI Transfer Format with CPHA = 0

Wenn das SPI als Slave konfiguriert ist, beginnt die Übertragung mit der fallenden Flanke der /SS-Leitung. Dies aktiviert das SPI des Slaves und das MSB des Bytes, was im

Daten-Register (SPDR) gespeichert ist, wird auf der MISO-Leitung ausgegeben. Die eigentliche Übertragung wird gestartet, indem eine Software in das SPDR des Masters schreibt. Dies führt dazu, dass das Taktsignal erzeugt wird. In Fällen, in denen die CPHA=0, bleibt das SCK-Signal für die erste Hälfte des ersten SCK-Zyklus gleich Null. Dadurch wird sichergestellt, dass die Daten an den beiden Eingabe-Leitungen des Masters und des Slaves stabil sind. Die Daten an den Eingabe-Leitungen werden mit der Flanke des SCK-Leitung von seinem inaktiven in seinen aktiven Zustand gelesen (steigende Flanke, wenn CPOL=0 und fallenden Flanke, wenn CPOL=1). Die Flanke der SCK-Leitung von seiner aktiven auf seinen inaktiven Zustand (fallenden Flanke, wenn CPOL=0 und steigende Flanke, wenn CPOL=1) bewirkt, dass die Daten ein Bit weiter geschoben werden, so dass das nächste Bit Ausgabe an den MOSI- und MISO-Leitungen ist. Nach acht Taktimpulse ist die Übertragung beendet. In den beiden, Master- und Slave-Gerät, wird das SPI-Interrupt Flag (SPIF) gesetzt und das empfangene Byte wird in den Empfangspuffer transferiert.

CPHA = 1, CPOL = 0 (Mode 2); CPHA = 1, CPOL = 1 (Mode 3)

Der Zeitablauf für eine SPI-Übertragung, wo CPHA=1 ist in Abbildung 2-3 dargestellt. Zwei Wellen-Formen werden gezeigt für die SCK-Signal - eine für CPOL=0 und eine weitere für CPOL=1.

(Seite 7)

Figure 2-3. SPI Transfer Format with CPHA = 1

Wie in den früheren Fällen wird mit der fallenden Flanke der /SS-Leitung der Slave ausgewählt und aktiviert. Im Vergleich zu vorherigen Fällen, in denen CPHA=0, wird die Übertragung nicht gestartet und MSB wird in diesem Stadium vom Slave nicht ausgegeben.

Die eigentliche Übertragung wird gestartet, indem durch Software in das SPDR des Masters geschrieben wird, was bewirkt, dass das Taktsignal erzeugt wird. Die erste Flanke des SCK-Signals von seinem inaktiven in seinen aktiven Zustand (steigende Flanke, wenn CPOL=0 und fallenden Flanke, wenn CPOL=1) verursacht sowohl die Master – als auch den Slave zur Ausgabe des MSB des Byte im SPDR. Wie in Abbildung 2-3 dargestellt, gibt es keine Verzögerung von einem halben SCK-Zyklus wie in Mode 0 und 1. Die SCK-Leitung ändert ihren Pegel sofort mit Beginn des ersten SCK-Zyklus. Die Daten an den Eingangs-Leitungen werden gelesen mit der Flanke des

SCK-Leitung von seinem aktiven seinen inaktiven Zustand (fallenden Flanke, wenn CPOL=0 und steigende Flanke, wenn CPOL=1).
Nach acht Taktimpulse ist die Übertragung beendet. In den beiden, dem Master- und dem Slave-Gerät, wird das SPI-Interrupt Flag (SPIF) gesetzt und das empfangene Byte wird in den Empfangspuffer transferiert.

(Seite 9)

5.7.7. Hinweise für High-Speed-Übertragungen

Geräte, die mit höherer System-Taktfrequenzen laufen und SPI-Module haben, die in der Lage sein sollen, Geschwindigkeiten bis zur Hälfte des Systemtakts zum Laufen zu bringen, benötigen einen spezifischen Zeitablauf, nach den Bedürfnissen sowohl des Sender, als auch des Empfängers. Die folgenden zwei Diagramme zeigen den Zeitplan für die AVR in Master und im Slave-Modus für die SPI-Modi 0 und 1. Die genauen Werte der dargestellten Zeiten variieren zwischen den verschiedenen Teilen und sind nicht Inhalt dieser Application Note. Jedoch ist die Funktionalität aller Teile im Prinzip die gleiche, so dass die folgenden Überlegungen auch für alle Teile gelten.

Figure 2-4. Timing Master Mode

Der minimale Zeitrahmen des Taktsignal wird durch die Zeiten "1" und "2" vorgegeben. Der Wert "1" legt den SCK fest, während der Wert "2" die High / Low-Werte des Taktsignals angeben. Die maximale Anstiegs- und Abfall-Zeit des SCK-Signals wird durch die Zeit "3" spezifiziert. Dies sind die ersten Zeitangaben des AVR, um zu überprüfen, ob sie mit den Anforderungen des Slaves übereinstimmen.
Das Setup-Zeit "4" und die Halte-Zeit "5" sind wichtig, weil sie die Anforderungen des AVR an die Schnittstelle des Slaves bestimmen. Diese Zeiten bestimmen, wie lange der Slave vor der Takt-Flanke gültige Output-Daten bereithalten muss und wie lange nach der Takt-Flanke diese Daten noch gültig zu sein haben.
Wenn die Setup- und Hold-Zeit lange genug sind für den Slave betreffende Anforderungen des AVR, aber haben die den AVR betreffend auch die Anforderungen des Slaves?
Die Zeit "6" (Out-SCK) legt die minimale Zeit fest, die der AVR gültig Output-Daten bereithält, bevor die Taktflanke auftritt. Diese Zeit ist vergleichbar mit der Setup-Zeit "4" des Slaves.
Die Zeit "7" (SCK zu Out) gibt die maximale Zeit, nach der der AVR das nächste Daten-Bit ausgibt, während der Zeit "8" (SCK Out zu High) die minimale Zeit gibt, in dem das

letzte Daten-Bit auf der MOSI-Leitung noch gültig ist, nachdem SCK wieder auf seine Ruhezustand zurück ist.

(Seite 9)

Figure 2-5. Timing Slave Mode

Im Prinzip sind die Zeitangaben im Slave-Modus die gleichen, wie zuvor im Master-Modus beschrieben.
Wegen der Umstellung der Rollen zwischen Master und Slave werden die zeitliche Anforderungen auch umgekehrt. Die Minmal-Zeiten vom Master-Modus werden nun die Maximal-Zeiten und umgekehrt.

5.7.8. SPI Übermittlungs-Konflikte

Eine Schreib-Kollision tritt auf, wenn in das SPDR geschrieben wird, während eine Übertragung noch im Gange ist. Da dieses Register in Sende-Richtung nur einfach gepuffert ist, verursacht das Schreiben von Daten in das SPDR direktes Schreiben in das SPI-Schieberegister. Weil diese Schreib-Operation mit den Daten des aktuellen Transfers einen Fehler der Schreib-Kollision erzeugt, wird das WCOL-Bit im SPSR gesetzt. Die Schreib-Operation wird in diesem Fall nicht ausgeführt und der Transfer wird weiterhin ungestört fortgesetzt.
Ein Schreib-Kollision ist in der Regel ein Slave-Fehler, weil ein Sklave keine Kontrolle darüber hat, wenn der Master eine Übertragung beginnen will. Ein Master jedoch weiß, wann eine Übertragung im Gange ist. So sollte ein Master keine Fehler mit Schreib-Kollision generieren können, auch wenn die SPI-Logik diese Fehler im Master-, als auch in den Slave-Modus erkennen kann.

5.7.9. SPI-Emulation

Bei der Emulation der SPI mit dem ICE200 Hardware-Emulator muß man sich der Tatsache bewusst werden, dass die Peripheriegeräte auf diesem Emulator mit einem Breakpoint nicht gestoppt werden, sondern sie laufen weiter mit der Geschwindigkeit, mit der sie konfiguriert sind.
Bei der Emulation der SPI mit dem ICEPRO kann das Timing weniger genau sein, als es der Fall ist auf dem Teil selbst. Dies wird durch längere interne Signalleitungen des

ICEPRO verursacht, welches der Preis ist, der gezahlt werden muss für die Fähigkeit zum Aktualisieren und ihre Flexibilität.

5.7.10. SPI Setup

Die Konfiguration des SPI im Master-Modus wird in zwei verschiedenen Möglichkeiten aufgezeigt. Das erste Beispiel zeigt die Implementierung einer SPI-Kommunikation durch Polling der Interrupt-Flags. Das zweite Beispiel zeigt die Implementierung einer Interrupt-gesteuerte Kommunikation.
Eine Kommunikation zwischen zwei AVR-Geräten wird gezeigt, indem ein "Text String" aus dem als Master konfiguriert Teil auf den anderen, als Slave konfiguriert Teil, gesendet wird. Die empfangenen Zeichen werden mit den zu erwarteten verglichen und das Ergebnis dieser Test-Kommunikation wird auf Port D ausgegeben. Diese Beispiele sind gut geeignet zur Implementierung auf zwei Entwicklungs-Boards, wie die STK500.

(Seite 10)

In allen hier gezeigten Beispielen ist das SPI so konfiguriert, dass es in Mode 0 arbeitet und das MSB als erstes übermittelt. Dies geschieht, indem die Bits CPOL, CPHA und DORD im Register SPCR auf Null gesetzt werden. Im gleichen Register wird das SPI aktiviert, indem die SPE-Bit gesetzt wird, während der SCK-Frequenz im ersten Beispiel zu CK/4 spezifiziert ist und im Assembler-Code für das zweite Beispiel. Im C-Code des zweiten Beispiels ist sie auf CK/16 gesetzt.
Zum Vergleichen der Konfiguration des SPI in den verschiedenen Beispielen muss sich die Aufmerksamkeit auf die Einstellungen des Master/Slave Select (MSTR) Bits und des SPI-Interrupt-Enable (SPIE) Bits richten.

Anmerkungen:

1. Die beide Beispiele zeigen, dass zwischen einem einzigen Master und einem einzigen Slave es nicht erforderlich ist zu überprüfen, ob das MSTR-Bit gesetzt ist, noch bevor der Master eine neue Übertragung initiiert.
Dieser Code muss hinzugefügt werden in einer Multi-Master-Anwendung.

2. Obwohl die Einstellungen der Taktraten Auswahl-Bits keine Wirkung auf den Slave-Modus haben, ist zu dafür zu sorgen, dass der Systemtakt (CK) vom Slave mindestens 4 mal höher ist, als der SPI-Takt (SCK).

3. Anhängige SPI-Interrupts können durch eine Dummy-Operation zum SPSR und SPDR gelöscht werden.

Zwei Dateien kommen zusammen mit dieser Application-Note, die den C-Code dieser Beispiele enthalten.
Zur Ausführung des Codes werden zwei STK500 Entwicklungs-Boards benötigt, wie in Abbildung 2-6 gezeigt. Der Code ist für den ATmega162, kann aber für jeden anderen Teil mit SPI-Hardware-Unterstützung und PORTA, PORTB und PORTD verwendet werden.

Figure 2-6. Hardware setup

(Seite 11)

Beispiel 1 - SPI Kommunikation durch Polling:

Master-Seite:

Wenn keine Interrupts verwendet werden, gibt es nur das SPI-Modul und seine Pins zu konfigurieren. Wichtig in diesem Beispiel ist die Einstellung des /SS-Pins als Output-Pin. Dies ist zu tun, bevor das SPI im Master-Modus aktiviert wird. Das Aktivieren des SPI, während das /SS-Pin nach wie vor als ein Eingangs-Pin konfiguriert ist, würde dazu führen, dass das SPI sofort in den Slave-Modus umschaltet, wenn ein Low-Pegel an diesem Pin ist. Dieses PIN ist im Slave-Modus immer als ein Eingangs-Pin konfiguriert (siehe Abbildung 2-7 auf Seite 11). Die Anwendung des Polling ergibt die schnellste Kommunikation. Dies ist der Grund, warum Polling am häufigsten im Master-Modus angewendet wird.

Figure 2-7. Polled master - initialization and transmission

Slave-Seite:

Für die Konfiguration des AVR für die Ausführung im Slave-Modus gibt es keine Reihenfolge, in der die Register initialisiert werden sollen. Das MISO-Pin muss als Output-Pin definiert sein, während alle anderen Pins automatisch als Input-Pins konfiguriert sind, wenn das SPI aktiviert ist (siehe Tabelle 2-8). Zur Konfigurierung des AVR zum Laufen im Slave-Modus muss das MSTR-Bit auf Null gesetzt werden. In diesem Fall, der synchronen Übertragung, ist die Wahl der Taktrate mit den Bits SPR0 und SPR1 egal.
Alle anderen Einstellungen des SPI-Konfigurations Register (SPCR) müssen die gleichen

sein wie im Master-Modus. Dies ist eine wesentliche Voraussetzung für eine erfolgreiche Kommunikation zwischen den beiden Geräten.

(Seite 12)

Figure 2-8. Polled slave - initialization and reception

(Seite 13)

Beispiel 2 - SPI Kommunikation von Interrupts:

Interrupt-kontrolliert Kommunikation im Master-Modus macht vor allem Sinn, wenn der SCK-Takt wird durch Division des Systemtakts mit einem großen Divisions-Faktor (z.B. 64 oder 128) abgleitet wird. In diesem Fall kann der Prozessor andere Verarbeitung durchführen, anstatt nur darauf warten, das nächste Byte zu senden/empfangen. Im Slave-Modus kann das Teil nicht wissen, wann eine Mitteilung beginnt, aber durch eine Interrupt-kontrollierte Umsetzung kann sichergestellt werden, dass das Teil sofort reagieren wird, so dass Schreib-Kollisions Fehler werden vermieden.

Master-Seite:

Die Initialisierung des SPI geschieht auf einem ähnlichen Weg zum einem wie im vorherigen Beispiel. Wie vor dem Master-Modus muss des /SS-Pin als erstes als Ausgang gesetzt und dann kann das SPI aktiviert werden.
Der SPI-Interrupt wird aktiviert, indem das SPIE-Bit im SPCR gesetzt wird.

Figure 2-9. Interrupt controlled master - initialization and transmission

Slave-Seite:

Ein Slave nie weiß, wann der Master mit einer neuen Mitteilung beginnen wird. Interrupts sind eine perfekte Funktion, um auf solche unbestimmten Ereignisse zu reagieren, so dass dies einen gemeinsamen Weg zur Umsetzung der SPI im Slave-Modus darstellt.
In diesem Beispiel wird das Hauptprogramm benachrichtigt über die Übertragungs-Fehler und die Vollendung der Übertragung.

(Seite 14)

Figure 2-10. Interrupt controlled slave - initialization and transmission

5.8. Übergang in den Analog-Modus und zurück

Die programmtechnische Realisierung dieses Features erfolgt mittels des sehr mächtigen Befehls

`ijmp`

Dazu muß vorher das Doppelregister Z mit der Startadresse der anzuspringenden Routine geladen werden. Wenn dieses Doppelregister sonst nicht weiter gebraucht wird (z.B. für den Befehl `lpm`), so kann dieses auch allein als ProgrammAdress-Zeiger verwendet werden. Universeller ist hingegen, den ProgrammAdress-Zeiger im RAM unterzubringen, so dass der Registersatz ohne weitere Einschränkungen nutzbar ist.
Die betreffende Programm-Sequenz muß dann in etwa so sein:
(Initialisierung)

```
.DSEG
analogZ:        .BYTE 2      ; 2 Bytes als ProgrammAdress-Zeiger reservieren

.CSEG
analog:                      ; (Routine zur Analogwert-Erfassung und Übertragung
                             ; in den Registersatz R1…R8) am Ende steht ein
   rjmp endlos               ; damit stets die Abfrage des ProgrammAdress-Zeigers
                             ; eingebunden ist
endlos:
   ld ZL, -X                 ; den Z-Pointer für "ijmp" laden
                             ; der X-Pointer muß jedes Mal vorher dekrementiert werden,
                             ; da er von vorher noch auf dem High-Wert steht
   ld ZH, X+                 ; "Leer"-Befehl, nur X-Zeiger erhöhen
   ld ZH, X
;
   ijmp                      ; wenn in der "Tasten_ISR.asm" der ProgrammAdress-Zeiger
                             ; auf "analog" geladen wurde, springt "ijmp" dorthin,
                             ; sonst Verbleib hier in der "endlos"-Schleife
```

Hier gibt es allerdings ein Problem:
Wenn durch einen Interrupt die „`endlos`“-Schleife gerade dann unterbrochen, wenn bereits der „halbe“ Zeiger - also das Low-Byte – geladen ist und dann innerhalb der ISR die Zeiger-Adresse verändert wird, springt das Programm mit `ijmp` sonst wo hin, nur nicht dort, wo man es erwartet. Eine Behebung des Problems ist nur möglich, wenn sichergestellt wird, dass sich beim Laden des Z-Zeigers keine Veränderungen ergeben können.
Realisiert wir das mit einem, oder mehreren eingeschobenen nop-Befehl(en), wobei zuvor Interrupts freigegeben (`sei`) und gleich danach wieder gesperrt (`cli`) werden. Nur wenn gerade der, oder die `nop`-Befehle abgearbeitet werden, wird ein Interrupt angenommen und in der zugehörenden ISR bearbeitet.

Falls zeitkritische Interrupts, die nichts mit der „Verbiegung“ des Z-Zeigers zu tun haben, es erforderlich machen, kann auch die globale Sperrung aller Interrupts durch differenzierteres Ausmaskieren des betreffenden Interrupts ersetzt werden.

Da der X-Pointer für die RAM-Adresse von den ISR oder der „Analog“-Routine verändert worden sein kann, muß dieser neu geladen werden.
Die „endlos“-Schleife sieht dann so aus:

```
endlos:
sei                         ; Interrupts global freigeben
nop
 …
nop
cli                         ; Interrupts sperren
ldi XL, LOW(analogZ)        ; ProgrammAdress-Zeiger im RAM
ldi XH, HIGH(analogZ)
ld ZL, X+                   ; den Z-Pointer für "ijmp" laden, X incrementieren
ld ZH, X
;
ijmp                        ; wenn in der "Tasten_ISR.asm" der ProgrammAdress-Zeiger
                            ; auf "analog" geladen wurde, springt "ijmp" dorthin,
                            ; sonst Verbleib hier in der "endlos"-Schleife


(ISR -> Analog-Betrieb)
ldi XL, LOW(analogZ)        ; ProgrammAdress-Zeiger im RAM
ldi XH, HIGH(analogZ)
ldi YL, LOW(analog)         ; Programm-Adresse "analog"
ldi YH, HIGH(analog)
st X+, YL
st X, YH                    ; ProgrammAdress-Zeiger steht jetzt auf "Analog"
                            ; d.h. wenn ISR zu Ende, wird in der Endlosschleife
                            ; von "main-Master.asm" mit "ijmp" auf "analog" verzweigt!
;
```

Für den Übergang wieder zurück in den Normal-Betrieb ist lediglich die Adresse „analog“ gegen die von „endlos“ auszutauschen.

5.9. Hardware-Multiplikation mit dem AVR

(Übersetzung der Application Note AVR201: Using the AVR® Hardware Multiplier)

Merkmale

• 8 - und 16-Bit-Implementierungen

• Routinen für vorzeichenbehaftete Zahlen und ohne

• Multiplizieren mit gebrochenen vorzeichenbehafteten Zahlen und ohne

• Ausführbare Beispiel-Programmen

Einführung

Der megaAVR ist eine Reihe von neuen Produkten der AVR RISC Microcontroller Familie, die unter anderem neue Erweiterungen, einen Hardware-Multiplikator umfasst. Dieser Multiplikator ist der Lage, zwei 8-Bit-Zahlen zu multiplizieren, so dass ein 16-Bit-Ergebnis mit nur zwei Takt-Zyklen entsteht. Der Multiplikator kann sowohl ganze Zahlen mit und ohne Vorzeichen und gebrochene Zahlen bearbeiten ohne Geschwindigkeits- oder Code-Größe Beschränkung. Der erste Abschnitt dieses Dokuments wird einige Beispiele der Verwendung des Multiplikators für 8-Bit-Arithmetik geben.

Um diesen Multiplikator verwenden zu können, werden sechs neue Anweisungen in der AVR-Befehlsliste festgelegt. Diese sind:

- **`MUL`**, Multiplikation von ganzen Zahlen ohne Vorzeichen (VZ).
- **`MULS`**, Multiplikation von ganzen Zahlen mit VZ.
- **`MULSU`**, Multiplikation eine ganzen Zahl mit VZ mit einer ganzen Zahl ohne VZ.
- **`FMUL`**, Multiplikation von gebrochene Zahlen ohne VZ.
- **`FMULS`**, Multiplikation von gebrochene Zahlen mit VZ.
- **`FMULSU`**, Multiplikation einer gebrochenen Zahl mit VZ mit einer gebrochene Zahl ohne VZ.

Die MULSU und FMULSU Befehle beinhalten auch eine Verbesserung der Geschwindigkeit und Code-Dichte für die Multiplikation von 16-Bit-Operanden. Der zweite Abschnitt zeigt Beispiele, wie man effizient den Multiplikator für 16-Bit-Arithmetik anwendet.

Die Komponente, die einen eigenen digitalen Signalprozessor (DSP), besonders geeignet für Signalverarbeitung macht, ist die Multiply-Accumulate (MAC)-Einheit. Diese Einheit ist funktional gleichwertig mit einem Multiplikator, der direkt mit einer arithmetischen Logikeinheit (ALU) verbunden ist. Die MegaAVR-Mikrocontroller sind

so aufgebaut, dass die AVR-Familie die Möglichkeit zur wirksamen Durchführung der gleichen Multiplikations-Additions-Operation hat. Diese Application Note wird daher auch Beispiele für die Umsetzung der MAC-Betrieb geben.

Der Multiply-Accumulate Betrieb (manchmal auch als multiplizier-addier-Betrieb bezeichnet) hat einen kritischer Nachteil. Wenn mehrere Werte zu einem resultierenden Variable addiert werden, auch wenn sich positive und negative Werte bis zu einem gewissen Grad einander aufheben, wird das Risiko einer Überschreitung der Grenzwerte der resultierenden Variable deutlich, d.h. wenn ein zu einer Byte-Variablen mit Vorzeichen, die den Wert +127 hat, „1“ addiert wird, wird das Ergebnis -128, anstelle von 128. Eine oft benutzte Lösung dieses Problems ist die Einführung gebrochener Zahlen, was Zahlen, die kleiner als 1 und größer als oder gleich -1 bedeutet. Der letzte Abschnitt präsentiert einige Fragen bezüglich der Verwendung gebrochener Zahlen.

(Seite 2)

Zusätzlich zu den neuen Multiplikations-Anweisungen, gibt es ein paar andere Ergänzungen und Verbesserungen im Prozessor-Kern des MegaAVR. Eine Verbesserung, die besonders nützlich ist, ist die neue Anweisung `MOVW` - Kopieren eines Word-Registers, welche eine Kopie eines Register-Paars in ein anderes Register-Paar macht.

Die Datei "AVR201.asm" enthält den Quellcode der Application Note der 16-Bit-Multiplikations-Routinen.

Eine Auflistung aller Implementierungen mit der Key-Performance-Spezifikationen ist in Tabelle 1 aufgeführt.

Table 1. Performance Summary

8-bit x 8-bit Routines	**Word (Cycles)**
Unsigned multiply 8 x 8 = 16 bits	1 (2)
Signed multiply 8 x 8 = 16 bits	1 (2)
Fractional signed/unsigned multiply 8 x 8 = 16 bits	1 (2)
Fractional signed multiply-accumulate 8 x 8 += 16 bits	3 (4)
16-bit x 16-bit Routines	
Signed/unsigned multiply 16 x 16 = 16 bits	6 (9)
Unsigned multiply 16 x 16 = 32 bits	13 (17)
Signed multiply 16 x 16 = 32 bits	15 (19)
Signed multiply-accumulate 16 x 16 += 32 bits	19 (23)
Fractional signed multiply 16 x 16 = 32 bits	16 (20)
Fractional signed multiply-accumulate 16 x 16 += 32 bits	21 (25)
Unsigned multiply 16 x 16 = 24 bits	10 (14)
Signed multiply 16 x 16 = 24 bits	10 (14)
Signed multiply-accumulate 16 x 16 += 24 bits	12 (16)

Wie einfach eine 8-Bit-Multiplikation über den Hardware-Multiplikator ist, werden die Beispiele in diesem Abschnitt deutlich zeigen. Es ist gerade das Laden der Operanden in zwei Register (oder nur einen für die Quadrat-Multiplikation) und ausführen eine der Multiplikations-Befehle. Das Ergebnis wird im Register Paar R0:R1 abgelegt. Zu beachten ist jedoch, dass nur die MUL-Anweisung keine Nutzungsbeschränkungen der Register haben. Abbildung 1 zeigt die Gültigkeit des (Operanden-) Registers für die Nutzung jeder der Multiplikations-Anweisungen.

Das erste Beispiel zeigt einen Code-Ausschnitt, der den Wert des Port-B-Eingangs ausliest und dieser Wert mit einer Konstanten (5) multipliziert, bevor das Ergebnis im Register-Paar R17: R16 abgelegt wird.

Beispiel 1 - Basis Anwendung

```
in r16, PINB          ; Read pin values
ldi r17, 5            ; Load 5 into r17
mul r16, r17          ; r1:r0 = r17 * r16
movw r17:r16,r1:r0 ; Move the result to the r17:r16 register pair
```

Zu beachten ist die Verwendung der neuen `MOVW`-Anweisung. Dieses Beispiel gilt für alle der Multiplikations-Anweisungen.

(Seite 3)
Figure 1. Valid Register Usage

Beispiel 2 - Sonderfälle

Dieses Beispiel zeigt einige besondere Fälle der MUL Anweisung, welche gültig sind.

```
lds r0,variableA          ; Load r0 with SRAM variable A
lds r1,variableB          ; Load r1 with SRAM variable B
mul r1,r0                 ; r1:r0 = variable A * variable B
lds r0,variableA          ; Load r0 with SRAM variable A
mul r0,r0                 ; r1:r0 = square(variable A)
```

Auch wenn die Operanden in das Ergebnis-Register-Paar R1: R0 gegeben werden, ergibt die Operation ein korrektes Ergebnis, da R1 und R0 im ersten Takt-Zyklus geholt werden und das Ergebnis im zweiten Taktzyklus zurück gespeichert wird.

Beispiel 3 – Multiply-accumulate Operation

Das letzte Beispiel einer 8-Bit-Multiplikation zeigt eine Multiplikations-Additions-Operation. Als allgemeine Formel kann wie folgt geschrieben werden:

```
c(n) = a(n) x b + c(n – 1)

; r17:r16 = r18 * r19 + r17:r16

in r18,PINB                ; Get the current pin value on port B
ldi r19,b                  ; Load constant b into r19
muls r19,r18               ; r1:r0 = variable A * variable B
add r16,r0                 ; r17:r16 += r1:r0
adc r17,r1
```

Typische Anwendungen für den multiplizierenden-akkumulierenden Betrieb sind FIR (Finite Impulse Response) und IIR (Infinite Impulse Response) Filter, PID-Regler und FFT (Fast Fourier Transform). Für diese Anwendungen ist die FMULS-Anweisung besonders nützlich.

(Seite 4)

Der wesentliche Vorteil der Verwendung der FMULS-Anweisung statt der MULS-Anweisung ist, dass das 16-Bit-Ergebnis der FMULS-Operation immer einem (welldefined) 8-Bit-Format angenähert sein wird. Dies wird näher diskutiert unter dem Abschnitt "Verwendung gebrochener Zahlen".

Die neuen Multiplikations-Anweisungen sind speziell zur Verbesserung der 16-Bit-Multiplikation gemacht. Dieser Abschnitt enthält Lösungen für die Verwendung des Hardware-Multiplikators zur Multiplikation mit 16-Bit-Operanden.

Abbildung 2 zeigt schematisch den allgemeinen Algorithmus für die Multiplikation zweier 16-Bit-Zahlen mit einem 32-Bit-Ergebnis (C = A * B). AH bezeichnet das High-Byte und AL das Low-Byte des A-Operanden. CMH bezeichnet das mittlere High-Byte und CML das mittlere Low-Byte des Ergebnisses C. Entsprechende Notationen werden für die verbleibenden Bytes verwendet.

Der Algorithmus ist die Basis für alle Multiplikationen. Alle 16-Bit-Teilergebnisse werden verschoben und anschließend addiert. Die Vorzeichen-Erweiterung ist nur für vorzeichen-behaftete Zahlen notwendig, zu beachten ist aber die Carry-Erweiterung, die für Zahlen ohne Vorzeichen berücksichtigt werden muß.

Figure 2. 16-bit Multiplication, General Algorithm

AH AL **X** BH BL
(sign ext) AL * BL

+ (sign ext) AL * BH
+ (sign ext) AH * BL
+ AH * BH
= CH CMH CML CL

Dieser Vorgang ist gültig für Zahlen mit und ohne Vorzeichen, auch wenn nur die vorzeichenlose Multiplikations-Anweisung (`MUL`) erforderlich ist. Dies ist in Abbildung 3 dargestellt. Eine mathematische Erläuterung wird hier gegeben:

Wenn A und B positive Zahlen sind, oder zumindest einer von ihnen gleich Null ist, wird der Algorithmus eindeutig richtig, vorausgesetzt, dass das Produkt C = A * B kleiner als 2^{16} ist, wenn das Produkt vorzeichenlos ist, oder kleiner als 2^{15}, wenn das Produkt als eine vorzeichenbehaftete Zahl benutzt wird.

Wenn beide Faktoren negativ sind, wird die Notation des 2er-Komplements verwendet; $A = 2^{16} - |A|$ und $B = 2^{16} - |B|$:

$$C = A * B = (2^{16} - |A|) * (2^{16} - |B|)$$
$$= |A * B| + 2^{32} - 2^{16} * (|A| + |B|)$$

(Seite 5)
Hier sind nur die 16 LSBs betroffen; der letzte Teil dieser Summe wird verworfen und wir erhalten das (richtige) Ergebnis C = | A • B |.

Figure 3. 16-bit Multiplication, 16-bit Result

AH AL	**X**	BH BL	
		AL * BL	1
+		AL * BH	2
+		AH * BL	3
=		CH CL	

Wenn ein Faktor negativ ist und ein Faktor positiv, zum Beispiel, A ist negativ und B ist positiv:

$$C = A * B = (2^{16} - |A|) * |B|$$
$$= (2^{16} * |B|) - |A * B|$$
$$= (2^{16} - |A * B|) + 2^{16} * (|B| - 1)$$

Die MSBs werden verworfen und die richtige Ergebnis in der Notation des 2er-Komplements wird $C = 2^{16} - | A * B |$.

Das Produkt muss im Bereich von $0 = C = 2^{16} – 1$ sein, wenn vorzeichenlose Zahlen verwendet werden und im Bereich $-2^{15} = C = 2^{15} - 1$, wenn vorzeichenbehaftete Zahlen verwendet werden.
Wenn eine Integer-Multiplikation in C-Sprache gemacht wird, kann sie verwendet werden. Der Algorithmus kann um 32-Bit-Multiplikation mit 32-Bit-Ergebnis erweitert werden.

(Seite 6)

16-Bit x 16-Bit = 24-Bit - Operation

Die Funktionalität der Routine ist in Abbildung 4 dargestellt. Für die 24-Bit-Version der Multiplikations-Routinen ist das Ergebnis in den Registern R18:R17:R16. Der Algorithmus ergibt korrekte Ergebnisse, sofern das Produkt $C = A * B$ kleiner als 2^{24} ist bei der Anwendung vorzeichenloser Multiplikation und kleiner als $\pm 2^{23}$ bei Anwendung vorzeichenbehafteter Multiplikation.

Figure 4. 16-bit Multiplication, 24-bit Result

AH AL	X	BH BL	
	AH * BH		1
		AL * BL	2
+	AH * BL		3
+	BH * AL		4
=	CH CM CL		

16-bit x 16-bit = 32-bit Operation

Beispiel 4 - Basis Anwendung 16-Bit x 16-Bit = 32-Bit-Integer Multiplikation

Nachfolgend ist ein Beispiel dafür, wie ein 16 x 16 = 32 Multiply Unterprogramm aufgerufen wird. Dies ist auch in Abbildung 5 dargestellt.

```
ldi R23,HIGH(672)
ldi R22,LOW(672)     ; Load the number 672 into r23:r22
ldi R21,HIGH(1844)
ldi R20,LOW(184)     ; Load the number 1844 into r21:r20
call mul16x16_32     ; Call 16bits x 16bits = 32bits multiply routine
```

Figure 5. 16-bit Multiplication, 32-bit Result

AH AL	**X**	BH BL	
(sign ext)	AL * BH		3

+	(sign ext) AH * BL	4
+	AH * BH AL * BL	1+2
=	CH CMH CML CL	

Das 32-Bit-Ergebnis der vorzeichenlosen Multiplikation von 672 und 1844 wird nun in den Registern R19:R18:R17:R16 abgelegt. Wenn "muls16x16_32" anstelle von "mul16x16_32" aufgerufen wird, wird eine vorzeichenbehaftete Multiplikation ausgeführt. Wenn "mul16x16_16" aufgerufen wird, wird das Ergebnis nur 16 Bit lang und im Register-Paar R17: R16 abgelegt. In diesem Beispiel wird sich das 16-Bit-Ergebnis als nicht richtig erweisen.

Figure 6. 16-bit Multiplication, 32-bit Accumulated Result

AH AL	**X**	BH BL
	(sign ext) AL * BL	
+	(sign ext) AL * BH	
+	(sign ext) AH * BL	
+	AH * BH	
+	CH CMH CML CL (Old)	
=	CH CMH CML CL (New)	

Gebrochene Zahlen

Vorzeichenlose gebrochene 8-Bit-Zahlen verwenden ein Format, bei dem Zahlen im Bereich [0, 2> erlaubt sind. Bits 6 bis 0 repräsentieren den Bruch und Bit 7 stellt die Integer-Teil (0 oder 1) dar, d.h., ein 1,7-Format. Die `FMUL`-Anweisung führt die gleiche Funktion wie der `MUL`-Anweisung aus mit der Ausnahme, dass das Ergebnis 1-Bit nach links verschoben ist, so dass das High-Byte des 2-Byte-Ergebnisses dieselbe 1,7-Format haben wird, wie die Operanden (anstelle eines 2/6-Format). Zu beachten ist, dass das Ergebnis nicht korrekt sein wird, wenn das Produkt ist gleich oder größer als 2 ist.

Um voll und ganz das Format der gebrochene Zahlen zu verstehen, ist ein Vergleich mit den Ganzzahl-Format nützlich: Tabelle 2 zeigt die zwei 8-Bit vorzeichenlosen Zahlen-Formate.

Vorzeichenbehaftete gebrochene Zahlen verwenden, wie auch die vorzeichenbehafteten Ganzzahlen, das bekannte 2er-Komplement-Format.

Zahlen im Bereich [-1, 1> können in diesem Format benutzt werden.

Wenn das Byte "1011 0010" als eine ganze Zahl ohne Vorzeichen interpretiert wird, wird es als 128 + 32 + 16 + 2 = 178 interpretiert. Auf der anderen Seite, wenn es sich um eine

gebrochene vorzeichenlose Zahl handelt, wird es als 1 + 0,25 + 0,125 + 0,015625 = 1,390625 interpretiert. Wenn davon ausgegangen wird, dass das Byte eine vorzeichenbehaftete Zahl ist, so wird es ausgelegt werden als zwischen 178 - 256 = -122 (Ganzzahl) oder als 1,390625 - 2 = -0,609375 (gebrochene Zahl).

Table 2. Comparison of Integer and Fractional Formats

Bit Number	**7**	**6**	**5**	**4**
Unsigned integer bit Significance	$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$
Unsigned fractional number bit significance	$2^0 = 1$	$2^{-1} = 0,5$	$2^{-2} = 0,25$	$2^{-3} = 0,125$
Bit Number	**3**	**2**	**1**	**0**
Unsigned integer bit Significance	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
Unsigned fractional number bit significance	$2^{-4} = 0,0625$	$2^{-5} = 0,3125$	$2^{-6} = 0,015625$	$2^{-7} = 0,0078125$

Die Anwendung den FMUL-, FMULS- und FMULSU-Anweisungen sollte nicht komplexer sein, als die MUL-, MULS- und MULSU-Anweisungen. Allerdings ist ein potentielles Problem die Zuweisung gebrochener Variablen mit richtigen Werten auf einfachem Weg. Der Anteil 0,75 (= 0,5 + 0,25) wird für das Beispiel "0110 0000", also wenn acht Bits verwendet werden.

Zur Konvertierung einer positiven gebrochene Zahl im Bereich [0, 2> (zum Beispiel 1,8125) in dem im AVR benutzten Format, sollte der folgenden Algorithmus, illustriert an einem Beispiel, verwendet werden:

Gibt es eine "1" in die Reihe?

Ja, 1,8125 ist größer als oder gleich 1.
Byte ist jetzt "1xxx xxxx"
Gibt es ein "0,5" im Rest?
0,8125/0,5 = 1,625
Ja, 1,625 ist größer oder gleich 1.
Byte ist jetzt "11xx xxxx"
Gibt es eine "0,25" im Rest?
0,625/0,5 = 1,25
Ja, 1,25 ist größer oder gleich 1.
Byte ist jetzt "111x xxxx"

(Seite 9)

Gibt es eine "0,125" im Rest?
0,25/0,5 = 0,5
Nein, 0,5 ist kleiner als 1.
Byte ist nun "1110 xxxx"
Gibt es eine "0,0625" im Rest?
0,5/0,5 = 1
Ja, 1 ist größer oder gleich 1.

Byte ist nun "1110 1xxx"
Da wir keinen Rest haben, sind die restlichen drei Bits Null, und das Endergebnis ist "1110 1000", das ist 1 + 0,5 + 0,25 + 0,0625 = 1,8125.

Zum Konvertieren einer negativen gebrochenen Zahl wird zunächst eine „2“ zu der Zahl addiert und dann den gleichen Algorithmus wie bereits gezeigt verwendet.

16-Bit-gebrochene Zahlen verwendet ein ähnliches Format, wie das der 8-Bit-gebrochenen Zahlen, die acht Bits High-Teil haben das gleiche Format wie das 8-Bit-Format. Die acht Bits Low-Teil sind nur ein Erhöhung der Genauigkeit des 8-Bit-Formats; Während das 8-Bit-Format eine Genauigkeit von $\pm 2^{-8}$ hat, hat das 16-Bit-Format eine Genauigkeit von $\pm 2^{-16}$. Dann wieder sind die 32-Bit-gebrochenen Zahlen eine Erhöhung der Genauigkeit der 16-Bit-gebrochenen Zahlen.
Beachte den wichtigen Unterschied zwischen Ganzzahlen und gebrochene Zahlen, wenn zusätzliche Byte(s) verwendet werden zum Abspeichern der Zahl:
damit die Genauigkeit der Zahlen erhöht wird, wenn gebrochene Zahlen verwendet werden, der Zahlen-Bereich, der vertreten werden kann ist überschritten, wenn ganze Zahlen verwendet werden.

Wie bereits erwähnt, hat die Anwendung vorzeichenbehafteter gebrochene Zahlen im Bereich [-1, 1> den Haupt-Vorteil gegenüber Ganzzahlen:
Wenn zwei Zahlen im Bereich [-1, 1>, multipliziert werden, wird das Ergebnis im Bereich [-1, 1] sein und eine Annäherung (das/die höchste(n) Byte(s)) über das Ergebnis werden in der gleiche Anzahl von Bytes abgespeichert, wie die Faktoren, mit einer Ausnahme:
Wenn beide Faktoren „-1“ sind, sollte das Produkt eigentlich „1“ sein, aber da die Zahl „1“ nicht mit diesem Zahlen Format dargestellt werden kann, wird die `FMULS`-Anweisung stattdessen die Zahl „-1“ in R1:R0 schreiben. Der Anwender sollte sich daher versichern, dass mindestens einer der Operanden bei Verwendung der `FMULS`-Anweisung nicht „-1“ ist. Die 16-Bit x 16-Bit-gebrochene Multiplikation hat auch dieses Einschränkung.

Beispiel 5 - Basic Usage 8-Bit x 8-Bit = 16-Bit- vorzeichenbehaftete gebrochene Multiplikation

Dieses Beispiel zeigt einen Code-Ausschnitt, der den Eingang Wert von Port B einliest

und diesen mit einem konstanten gebrochene (-0,625) Wert multipliziert, vor dem Abspeichern des Ergebnis in Register-Paar R17:R16.

```
in r16,PINB          ; Read pin values
ldi r17,$B0          ; Load -0.625 into r17
fmuls r16,r17        ; r1:r0 = r17 * r16
movw r17:r16,r1:r0 ; Move the result to the r17:r16 register pair
```

Beachte, dass die Nutzung der FMULS- (und FMUL-) Anweisungen sehr ähnlich ist, wie die Verwendung der MULS- und MUL-Anweisungen.

(Seite 10)

Das nachstehende Beispiel verwendet die Daten aus dem ADC. Das Produkt sollte so konfiguriert werden, dass das Format des ADC-Ergebnisses kompatibel ist mit dem 2er-Komplement für gebrochene Zahlen. Für die ATmega83/163 bedeutet dies, dass das ADLAR-Bit im ADMUX I/O-Register gesetzt ist und ein Differenz-Kanal verwendet wird. (Die ADC-Ergebnis ist dann auf „1“ normiert.)

```
ldi r23,$62          ; Load highbyte of fraction 0.771484375
ldi r22,$C0          ; Load lowbyte of fraction 0.771484375
in r20,ADCL          ; Get lowbyte of ADC conversion
in r21,ADCH          ; Get highbyte of ADC conversion
call fmac16x16_32    ;Call routine for signed fractional multiply accumulate
```

Die Register R19:R18:R17:R16 werden das Ergebnis der Multiplikation von 0,771484375 mit der Ergebnis der ADC-Umsetzung enthalten. In diesem Beispiel wird der ADC-Ergebnis als eine vorzeichenbehaftete gebrochene Zahl behandelt. Wir könnten sie auch wie eine vorzeichenbehaftete Ganzzahl behandeln und "mac16x16_32" anstelle von "fmac16x16_32" aufrufen. In diesem Fall sollte der 0,771484375-Wert durch eine Ganzzahl ersetzt werden.

Kommentar zu Implementierungen

Alle hier implementierten 16-Bit x 16-Bit = 32-Bit-Funktionen starten durch Löschen des Register R2, welches nur als "Dummy"-Register für "add mit carry" (ADC) und "subtrahieren mit carry "(SBC) Operationen verwendet wird. Diese Operationen verändern nicht den Inhalt des Registers R2.

Wenn das Register R2 nicht an anderer Stelle im Code gebraucht wird, ist es nicht notwendig das Register R2 zu löschen jedes Mal, wenn diese Funktionen aufgerufen werden, aber nur einmal vor dem ersten Aufruf von einer der Funktionen.

6. Projektübersicht

LfdNr	Datei	Version	Erstellungs-datum	OK-getestet	Bemerkung
6.0	m128def.inc				Definitionsdatei für den ATMega128 – muß hier vorhanden und mit eingebunden sein
6.1.	main.asm	1.1	10.07.2008	17.07.2008	erster Test der Grundroutine für eine 8-Kanal-PWM - wird hier ***nicht*** verwendet
6.2.	main2.asm	1.2	30.07.2008	02.08.2008	Grundroutine des 8-Kanal-PWM, RAM füllen usw. Start-Datei mit Einsprungpunkt
6.3.	main_init.asm	1.1, 1.2	23.06.2008	27.07.2008	Erste/zweite Version der Initialisierungssequenzen für das AVR128-Board, incl. statischer Zeichenausgabe für 3 Zeilen auf dem LCD - wird hier ***nicht*** verwendet
6.4.	main_init3.asm	1.3	27.07.2008	30.07.2008	dritte Variante der Initialisierungssequenzen für das AVR128-Board, incl. Zeichenausgabe von 4 Zeilen aus dem RAM auf das LCD
6.5.	lcd4_init.asm	1.3	29.07.2008	30.07.2008	LCD-Anzeige im 4-Bit-Modus initialisieren, incl. der Übertragung von 8 benutzerdefinierten Zeichen in den LCD-Zeichensp.
6.6.	lcd4_com.asm	1.2	18.07.2008	25.07.2008	UP zur Übertragung von Befehlen an das LCD
6.7.	lcd4_put.asm	1.1	20.07.2008	25.07.2008	UP zur Datenausgabe an das LCD
6.8.	lcd4_displ.asm	1.1	30.07.2008	31.07.2008	UP zur LCD-Anzeige im 4bit-Modus, Anzeige von 80 Byte BWS
6.9.	DGcounter_init.asm	1.1	02.07.2008	06.07.2008	Programmteil (kein UP) zur Initialisierung des Drehgeber-Interfaces
6.10.	DGcounter_ISR.asm	1.1	06.07.2008	19.08.2008	ISR für das Drehgeber-Interface zur Impuls- und Schaltererfassung und
6.11.	UP_warte1ms.asm	1.1	05.07.2008	06.07.2008	UP für eine 1ms-

					Warteschleife
6.12.	UP_bin2asci	1.1	05.08.2008	07.08.2008	UP zur Wandlung eines bin/dez-Wertes in Asci zur Anzeige
6.13.	UP_Chr2RAM	1.1	05.08.2008	06.08.2008	UP zur Anzeige u.a. des akt. Channels
6.14.	UP_mul8bit.asm	1.1	11.08.2008	12.08.2008	UP zur 8x8Bit Multiplikation
6.15.	UP_div16-8bit.asm	1.1	11.08.2008	12.08.2008	UP für 16 / 8Bit Division
6.16.	UP_div8bit.asm	1.1	05.08.2008	07.08.2008	UP für 8Bit-Division
6.17.	UPs_div8bit.asm				UP-Quellen für 8Bit-Division – wird hier ***nicht*** verwendet
6.18.	LCD4_balken.asm	1.1	20.08.2008	25.08.2008	UP zur Balkenanzeige
6.19.	UPs_EEPROM.asm eeprd (reprom)	1.1	06.08.2008	06.08.2008	UP zur Datenwiederherstellung aus dem EEPROM
6.20.	UPs_EEPROM.asm eep_ISR (weprom)	1.1	06.08.2008		Datensicherung im EEPROM
6.21	LCD4_dipl%.asm	1.1	19.08.2008	19.08.2008	Refresh der %-Anzeige im LCD, nur 4 Zeichen
6.22	Taster_ISR.asm	1.1	06.08.2008	18.08.2008	ISR zur Bedienung des Steuerknopfes
Master-Slave Variante					
6.25	main.asm	1.1			Hauptprogramm Slave
6.26	main_init.asm	1.1			Initialisierung Slave
6.27	main_Master.asm	1.1	10.09.2008		Hauptprogramm Master
		1.2	20.10.2008		+Analog, +Sleep, +Leistung
6.28	main-Master_init.asm	1.1	12.09.2008		Initialisierung Master
		1.2.	18.10.2008		+Analog, +Sleep
6.29	SPI_ISR.asm	1.1	01.09.2008	09.09.2008	ISR für Slave-SPI
	SPI_ISR_2.asm	1.2	08.11.2008		+Analog, +Sleep
	SPI_ISR_3.asm	1.3	03.12.2008		neue Version, alle 5 Sequenzen als Block
6.30	SPI-Master_ISR.asm	1.1			ISR für Master-SPI (Analogausgabe „xxW“)
6.31	PD_ISR.asm	1.2			ISR für Drehgeber (modifiziert für SPI)
6.32	LCD4_balken.asm	1.1	20.08.2008	25.08.2008	Balkenanzeige

6.33	LCD4_com.asm	1.2	18.07.2008	25.07.2008	UP zur Übertragung von Befehlen an das LCD
6.34	LCD4_displ.asm	1.1	30.07.2008	31.07.2008	UP zur LCD-Anzeige im 4bit-Modus, Anzeige von 80 Byte BWS
6.35	LCD4_init.asm	1.3	29.07.2008	30.07.2008	LCD-Anzeige im 4-Bit-Modus initialisieren, incl. der Übertragung von 8 benutzerdefinierten Zeichen in den LCD-Zeichensp.
6.36	LCD4_put.asm	1.1	20.07.2008	25.07.2008	UP zur Datenausgabe an das LCD
6.37	Taster_ISR.asm	1.2	17.09.2008		ISR zur Bedienung des Steuerknopfes, Master-Variante (ohne PWM)
		1.3	22.10.2008		+Sleep, +Analog, +Leistung
6.38	DGcounter_init.asm	1.1	02.07.2008	06.07.2008	Programmteil (kein UP) zur Initialisierung des Drehgeber-Interfaces
6.39	DGcounter_ISR.asm	1.2	17.09.2008		ISR für das Drehgeber-Interface zur Impulserfassung Master-Variante
		1.3	06.12.2008		SPI-Block-Übertragung, +Leistung
6.40	UP_warte1ms.asm	1.1	05.07.2008	06.07.2008	UP für eine 1ms-Warteschleife
6.41	UP_bin2asci	1.1	05.08.2008	07.08.2008	UP zur Wandlung eines bin/dez-Wertes in Asci zur Anzeige
6.42	UP_Chr2RAM	1.1	05.08.2008	06.08.2008	UP zur Anzeige u.a. des akt. Channels
6.43	UP_mul8bit.asm	1.1	11.08.2008	12.08.2008	UP zur 8x8Bit Multiplikation
6.44	UP_div16-8bit.asm	1.1	11.08.2008	12.08.2008	UP für 16 / 8Bit Division
6.45	UP_div8bit.asm	1.1	05.08.2008	07.08.2008	UP für 8Bit-Division
6.46	UPs_EEPROM.asm eeprd (reprom)	1.1	06.08.2008	06.08.2008	UP zur Datenwiederherstellung aus dem EEPROM
6.47	UPs_EEPROM.asm eep_ISR (weprom)	1.1	06.08.2008		Datensicherung im EEPROM
6.48	LCD4_dipl%.asm	1.1	19.08.2008	19.08.2008	Refresh der %-Anzeige im LCD, nur 4 Zeichen

	DGcounter_test.asm	1.1	07.07.2008	08.07.2008	DG-Counter auf ATMega-Board, **Testversion** – wird hier ***nicht*** verwendet
	lcd4_cur.asm				Cursorkontrolle für LCD-Anzeige mit 4 Zeilen - wird hier ***nicht*** verwendet
	lcd4_func.asm				Funktionen zur Cursor-kontrolle - wird hier ***nicht*** verwendet
	lcd4_putStr.asm				Stringausgabe an das LCD - wird hier ***nicht*** verwendet
	LCD-Controller KS0066U.asm				LCD-DISPLAY ROUTINES (engl.) Beispiele mit EEPROM-Einbindung - wird hier ***nicht*** verwendet
	LCD-Routinen AVR-Tut.asm				Beispielroutinen zur LCD-Ansteuerung - wird hier ***nicht*** verwendet
	TC_init.asm	1.1	02.07.2008		Initialisierung des Timer/Counters zum Zählen der Drehgeber-Impulse und Richtungsauswertung - wird hier ***nicht*** verwendet
	TC_ISR.asm	1.2	23.06.2008		Interrupt-Service-Routinen, wandeln zwei Digitalwerte des AD-Wandlers in je eine Impulsfrequenz - wird hier ***nicht*** verwendet
	AVR-SPI Master.ASM				Atmel-SPI-Grundroutinen für AVR Master-Slave
	SPI-Schnittstelle.asm				weitere SPI-Beispiele
	ATMega SPI Performance Tuning.mht				Optimierung der SPI-Übertragungsgeschwindigkeit