Graphic-LCD am AVR

Dipl.-Ing. (FH) Peter Salomon

Inhaltsverzeichnis

XXX

1. Vorbemerkungen

Im Gegensatz zu einfachen LCDs zur Darstellung von ausschließlich Zeichen (Buchstaben, Zahlen, Sonderzeichen) mit 1, 2 oder 4 Zeilen und max. 20 Zeichen gibt es bei Graphic-LCDs die unterschiedlichsten Bauformen und damit Auflösungen in horizontaler und vertikaler Dimension. Hinzu kommt noch, daß es zwar einige quasi-Standard-Controller gibt, z.B.

T6863.pdf

HD61202.pdf,

KS0108b.pdf,

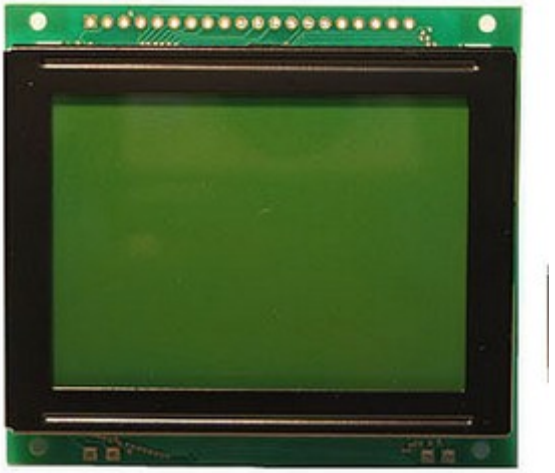
ST7565R.pdf

darüber hinaus aber noch jede Menge exotische Spezial-Controller, auch für ausgefallene Pixel-Anzahl in horizontaler und vertikaler Richtung.

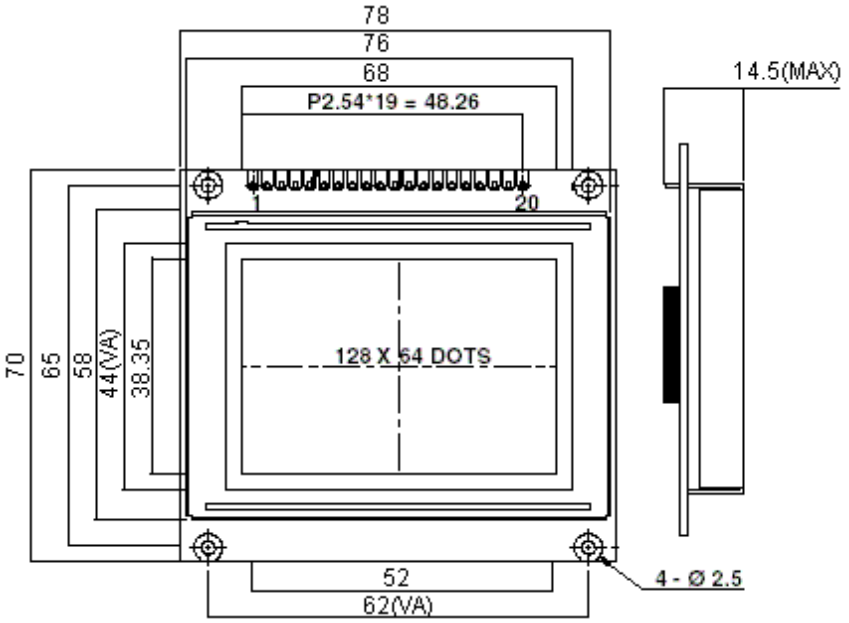
2. Produkt-Informationen

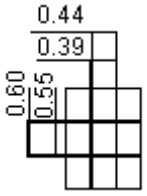
2.1. **Preiswertes GLCD mit T6963-Controller** ($22,90)

(http://www.futurlec.com/LED/LCD128X64.shtml)



Features
Built-in Controller (T6963C) +5V Power Supply 1/64 Duty Cycle Built-in Backlight


78
76
68
P2.54*19 = 48.26
14.5(MAX)
1
20
70
65
58
44(VA)
38.35
128 X 64 DOTS
52
62(VA)
4 - Ø 2.5


0.44
0.39
0.60
0.55

Module Abmessung	78 x 70 x 12 mm
Bildgrösse	65 x 44 mm
Pixel Grösse	0,39 x 0,55 mm
Pixel Abstand	0,44 x 0,6 mm

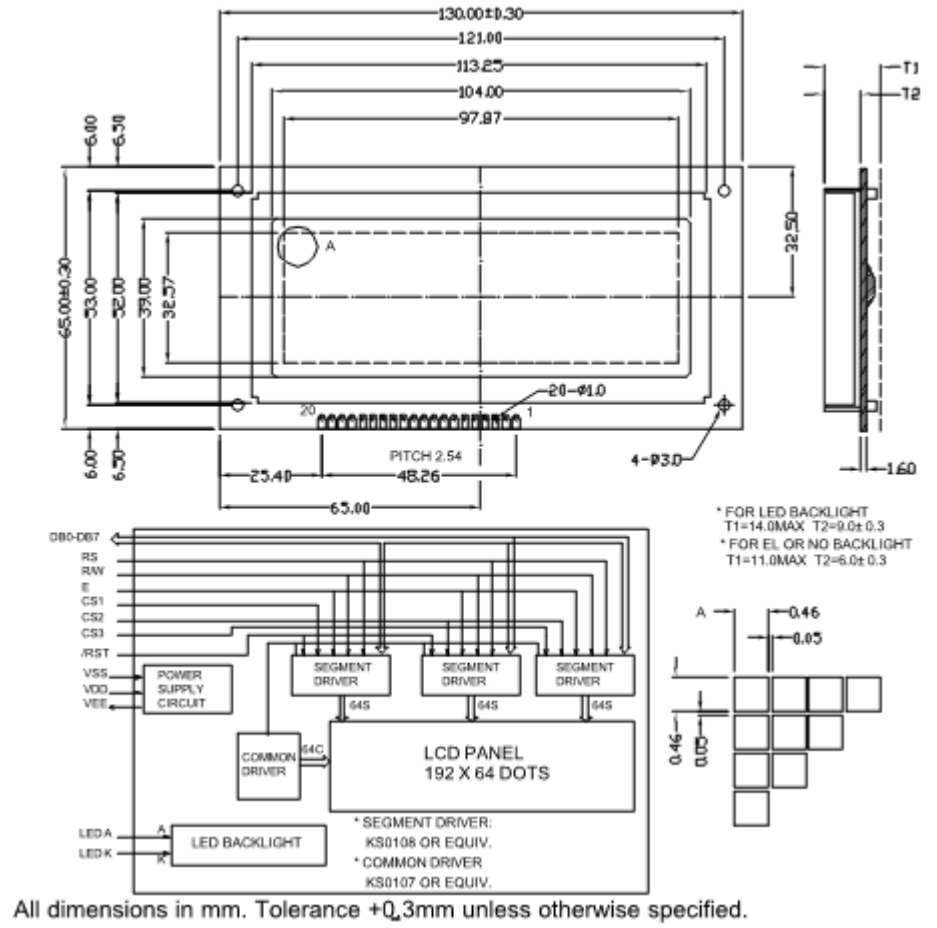
2.2. **Preiswertes GLCD mit KS0108-Controller** (6,95 Euro)

(http://www.pollin.de/shop/dt/NzgyOTc4OTk-/Bauelemente_Bauteile/Aktive_Bauelemente/Optoelektronik/LCD_Modul_DM19264A_02_192x64_Pixel.html)

Grafisches LCD mit gelber LED-Hintergrundbeleuchtung und integriertem Controller (KS0108B).

Technische Daten:

- Betriebsspannung 5 V-
- 192x64 Pixel
- integrierter Controller KS0108B
- LED-Hintergrundbeleuchtung
- 20-poliger Anschluss
- sichtbarer Bereich 104x39 mm
- Außenmaße (LxBxH): 130x65x12,5 mm



All dimensions in mm. Tolerance +0.3mm unless otherwise specified.

Die Kontrastspannung an Pin 3 wird mittels Poti eingestellt – angeschlossen zwischen Pin 19 (-5V/-10V) und Vdd (Pin 2). Es ist möglich, daß Pin 19 nicht angeschlossen ist (Leiterbahnunterbrechung), ggf. reichen die -5V mit einem integrierten 7660 nicht aus, dann kann ein zweiter nachgerüstet werden, um -10V zu erhalten. Die CSx-Signale sind LOW-aktiv. Mit der fallenden Flanke von E werden die Daten übernommen, d.h. die Daten müssen auch nach der fallenden Flanke von E noch stabil anliegen.

Bei diesem Pollin-Sonderangebot fehlt die Verbindung zwischen dem Anschluß-Pin 19 und Pin 5 des 7660, d.h. die -5V liegen nicht am Anschluß 19 an. Diese Verbindung muß nachträglich hergestellt werden, wenn mit dem internen -5/-10V-Generator gearbeitet werden soll.

Die Anschlüsse der Beleuchtungs-LEDs sind separat seitlich herausgeführt. Nach Datenblatt sollen hier 4,2V anliegen, wobei $I_{LED} < 300mA$ sein muß, was mit einem Vorwiderstand von 4,7 Ohm, geschaltet von der +5V-Betriebsspannung realisierbar ist.

3. Hardware-Vorrausetzungen

Vergleich der Anschaltung:

KNr	Chr-LCD	GLCD (T6963)	GLCD (KS0108)
1	GND	FG/Vee (0V)	Vss - GND
2	Vcc	Vss (0V)	Vdd - +5V
3	LCD-Kontrast	Vdd (+5V)	Vo (Vdd-15V…Vdd+0,3V)
4	RS	Vo (0 … 10V)	RS
5	RW	WR	R/W
6	E	RD	E
7	DB0 (GND)	CE	DB0
8	DB1 /GND)	C/D	DB1
9	DB2 (GND)	Reset	DB2
10	DB3 (GND)	DB0	DB3
11	DB4	DB1	DB4
12	DB5	DB2	DB5
13	DB6	DB3	DB6
14	DB7	DB4	DB7
15	NC	DB5	CS1
16	NC	DB6	/RST
17	---	DB7	CS2
18	---	FS	CS3
19	---	K (LED +5V)	VEE (neg. Outp. -5V/-10V)
20	---	A (LED 0V)	A (LED-Backlight 4,2V)

4. Hardware - Initialisierung

4.1. Variante mit T6963-Controller

Bevor irgendwelche Aktionen mit dem `GrController` unternommen werden können, muß dieser (wie auch die `CharController`) initialisiert werden. In der *ApplicationNote.pdf* wird dazu folgendes geschrieben:

Zunächst muß das GrLCD incl. Controller nach dem Zuschalten von Vcc mindestens für 6 Takte (oder 1ms) an /RES auf LOW (Reset) gehalten werden. Das kann mittels 10kOhm gegen Vcc und 0,1µF gegen Masse erreicht werden (siehe Appl.-Schaltung S7 der *ApplicationNote.pdf*).

Da Steuer-Controller (AVR) und `GrController` asynchron arbeiten, muß bevor

irgendwelche Befehle oder Daten an den `GrController` gesendet werden, der Statuszustand von `GrController` geprüft werden.

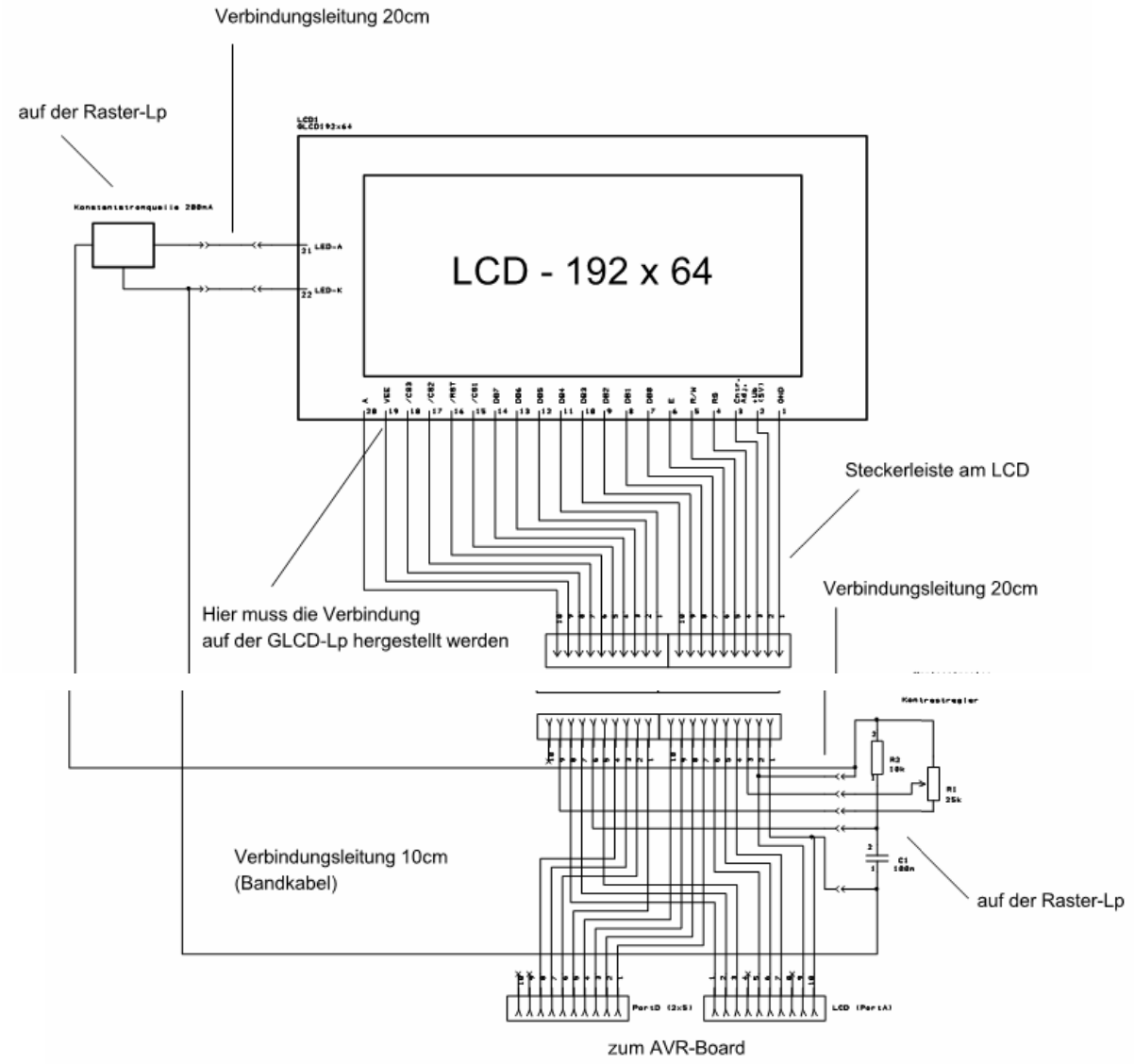
In Anlehnung an `T6963c_main_v1.asm` bzw. `TEST2 - T6963.ASM` wird die Initialisierungs-Routine so aussehen:

(*weiteres wird zunächst ausgesetzt, wegen Orientierung auf das KS0180-Controller-Display*)

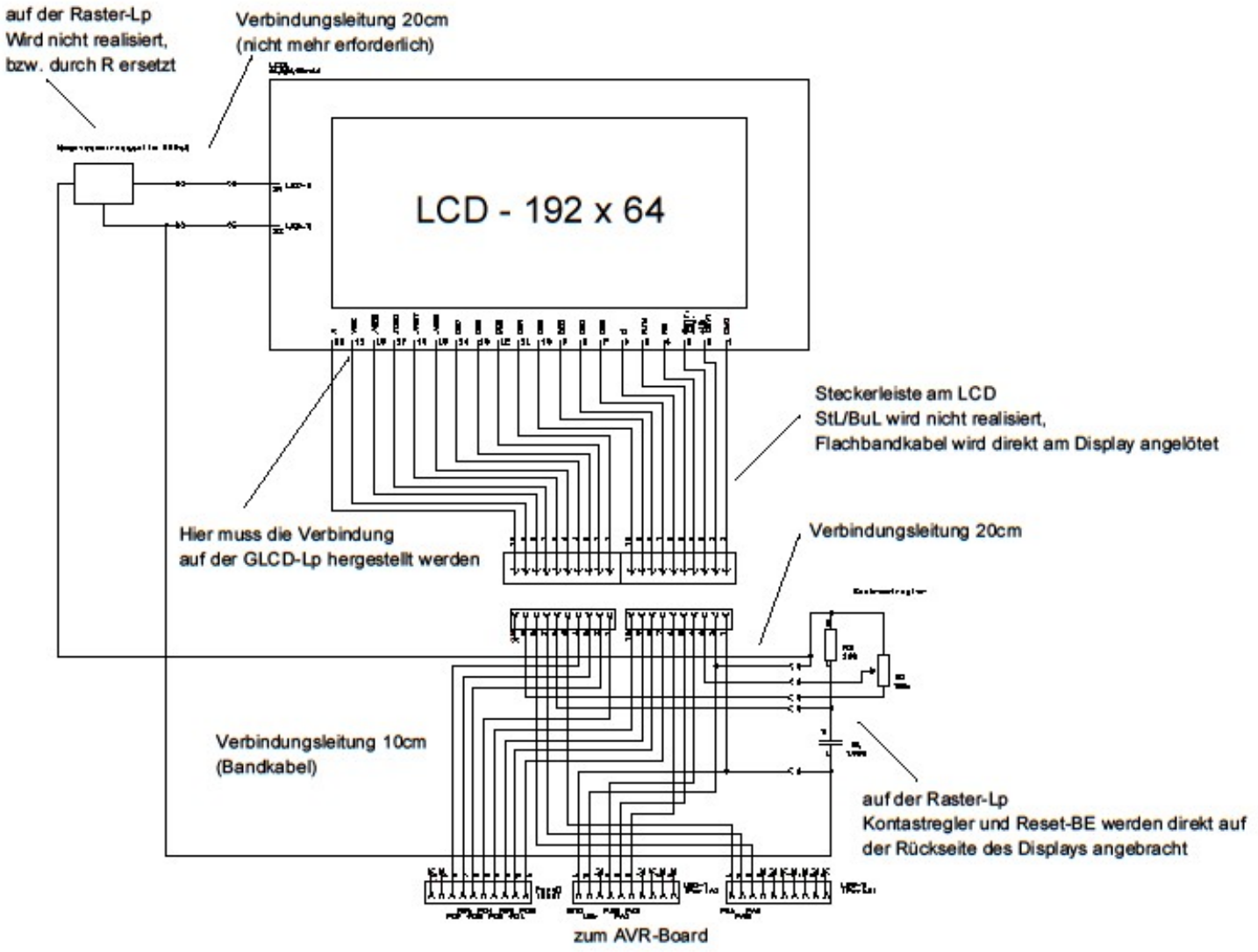
4.2. Variante mit KS0108-Controller

Ein Test ist erst nach der Hardware-Implementierung möglich.

Dazu wird zunächst eine Anschaltung wie folgt vorgenommen:



Die Anschaltung wurde noch mal korrigiert:


auf der Raster-Lp
Wird nicht realisiert,
bzw. durch R ersetzt
Verbindungsleitung 20cm
(nicht mehr erforderlich)
LCD - 192 x 64
Steckerleiste am LCD
StL/BuL wird nicht realisiert,
Flachbandkabel wird direkt am Display angelötet
Hier muss die Verbindung
auf der GLCD-Lp hergestellt werden
Verbindungsleitung 20cm
Verbindungsleitung 10cm
(Bandkabel)
auf der Raster-Lp
Kontastregler und Reset-BE werden direkt auf
der Rückseite des Displays angebracht
zum AVR-Board

Das Daten-Interface wird adaptiert in Anlehnung an die Originalschaltung des „ATMegaEvoBoard“ wie folgt:

LCD-Connector	Pin 4 (RS)	->	PortA0
	Pin 5 (R/W)	->	PortA1
	Pin 6 (E)	->	PortA2
	Pin 12 (/CS0)	->	PortA5
	Pin 13 (/CS1)	->	PortA6
	Pin 14 (/CS2)	->	PortA7
Daten	Pin 1 (DB0)	->	PortD0
	Pin 2 (DB1)	->	PortD1
	Pin 3 (DB2)	->	PortD2
	Pin 4 (DB3)	->	PortD3
	Pin 5 (DB4)	->	PortD4
	Pin 6 (DB5)	->	PortD5
	Pin 7 (DB6)	->	PortD6
	Pin 8 (DB7)	->	PortD7

Sind die o.g. aufgeführten Hardware-mäßigen Verbindungen hergestellt, sollte zunächst die LED-Beleuchtung geprüft werden, des Weiteren die LCD-Kontrasteinstellung mittels Poti R1.
R2 in Verbindung mit C1 bewirkt ein `Reset` des GLCD beim Einschalten.
Entsprechend der Zeitkonstante:

$T = 2\pi * R2 * C1 = 6{,}28 * 10^{4}\ V/A * 10^{-7}\ As/V = 6{,}28 * 10^{-3}\ s$ bzw. $\approx$ 6ms

ergibt sich, daß erst nach ca. 10ms an das GLCD Daten oder Befehlen gesendet werden können. Das ist bei der Init-Routine des AVR zu berücksichtigen.

Der Einschalt-Test erfolgte positiv, d.h. die Hintergrundbeleuchtung ließ sich ein- und ausschalten (ist im SP nicht enthalten).
Ansonsten bedarf es zunächst einige Test-Routinen, um auf dem Display etwas sehen zu können, siehe dazu Abschnitt 6 - Grund-Funktionen.

5. Befehlsliste des KS0180

Nach Datenblatt *KS0108b.pdf* und in Ergänzung zum Äquivalent-Chip *S6B0108-KS0108.pdf* von SAMSUNG sind folgende 7 Befehle verfügbar:

(1) **DISPLAY ON/OFF**

Das Display ON/OFF-Flip-Flop schaltet die Flüssigkristall-Anzeige ON/OFF. Wenn das Flip-Flop zurückgesetzt wird (logisch), wird die Spannung an den Segment-Ausgängen selektiert oder nicht. Wenn das Flip-Flop gesetzt ist (logisch hoch), erscheint die nicht selektierte Spannung auf den Segment Ausgangsklemmen unabhängig von Display-RAM Daten. Der Status des Display ON/OFF-Flip-Flops kann geändert werden durch einen Befehl. Die Anzeige von Daten auf allen Segmenten verschwindet, während RSTB LOW ist. Der Status der Flip-Flop-Ausgang wird ausgegeben an DB5 vom Status Lesebefehl. Das Display ON/OFF-Flip-Flop wird vom CL-Signal synchronisiert.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	1	1	1	1	D

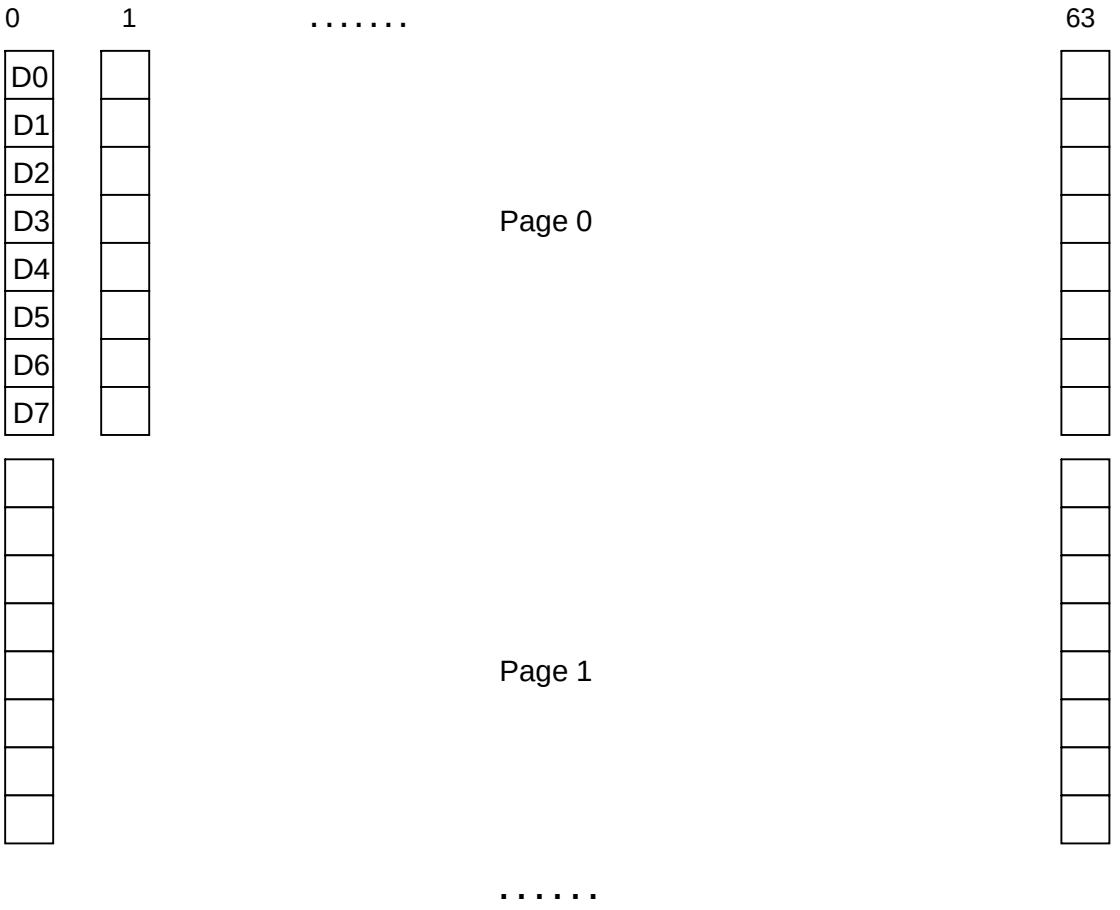
Die Daten-Anzeige wird aktiviert, wenn D = 1 ist, und verschwindet, wenn D 0 ist. Obwohl die Daten nicht auf dem Bildschirm mit D = 0, es bleiben die Daten im Anzeige-RAM. Daher können diese angezeigt werden, indem D = 0 in D = 1 geändert wird.

(2) **SetAdress (Y)**

Y Adreßzähler bezeichnet die Adresse des internen Display Daten-RAMs. Eine Adresse wird vom Befehlssatz gesetzt und wird automatisch um 1 erhöht von Lese- oder Schreibvorgängen der Display-Daten.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0

Y-Adresse (AC0 - AC5) des Daten-RAMs der Anzeige wird im Y-Adress-Zähler gesetzt. Eine Adresse wird durch den Befehlssatz gesetzt und um 1 automatisch nach jedem Lese- oder Schreibvorgänge von Display-Daten erhöht.



Die Bezeichnung der X- /Y-Adresse ist widersprüchlich. Der X-/Y-Bezeichnung zur Folge müssten die Datenbytes waagerecht in einer Zeile abgebildet werden und es demzufolge 8 Datenbytes auf einer Zeile bei 64 Zeilen untereinander geben.
Die o.g. strukturelle Definition ist aus der „*ApplNote - KS0108 (AN2147 - CYPRESS).pdf*“ entnommen worden – Ob aber ein Daten-Byte wirklich 8 Pixel senkrecht untereinander beschreibt, muß aber noch in der Realität überprüft werden.

(3) **SetPage (X)**

Das X-Seiten Register bezeichnet die Seiten des internen Display Daten-RAMs. Eine Count-Funktion ist nicht verfügbar. Die Adresse wird gesetzt durch den Befehl:

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	0	1	1	1	AC2	AC1	AC0

X-Adresse (AC0 - AC2) des Displays-RAM wird im X-Adress-Register gesetzt. Schreiben oder Lesen von oder nach MPU wird in dieser angegebenen Seite ausgeführt, bis die nächste Seite gesetzt ist.

(4) **Display StartLine**

Das Display-Startlinie-Register zeigt auf die oberste Zeile des Display-RAMs des LCDs. Die Bit-Daten (DB0 bis DB5) des Displays Startlinie werden im Anzeige-Startlinie-Register gehalten. Die gespeicherten Daten werden an den Z-Adreßzähler zur Voreinstellung des Z-Adress-Zählers übertragen, während FRM HIGH ist. Das wird angewendet zum Scrollen der verwendeten LCDs.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	1	AC5	AC4	AC3	AC2	AC1	AC0

Z-Adresse (AC0 - AC5) des Anzeige-Daten-RAMs wird im Anzeige-Startlinie-Register eingestellt und zeigt auf den Anfang des Bildschirms. Wenn das Anzeige-Tastverhältnis 1 / 64 oder andere (1 / 32 - 1 / 64) ist, werden die Daten der gesamten Zeilenanzahl des LCD-Bildschirm angezeigt, die von der Zeile der Display-Startlinie-Anweisung angegeben wurde.

(5) **Status Read**

Dieser Befehl sollte vor allen Schreib- oder Leseaktionen ausgeführt werden, um z.B. sicherzustellen, daß der LCD-Controller bereits ist Daten zu empfangen, bzw. zu senden und nicht mit internen Operationen beschäftigt ist.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	BUSY	0	ON/OFF	RESET	0	0	0	0

· BUSY

Wenn BUSY = 1 ist, wird der Chip interne Operationen ausführen und keine Anweisungen akzeptieren.

Wenn BUSY = 0 ist, ist der Chip bereit alle Anweisungen zu akzeptieren.

· ON / OFF

Wenn ON / OFF = 1 ist das Display ausgeschaltet.

Wenn ON / OFF = 0 ist das Display eingeschaltet.

· RESET

Wenn RESET = 1 ist, wird das System initialisiert.

In diesem Zustand kann keine Anweisungen außer dem Status gelesen und akzeptiert werden.

Wenn RESET = 0 ist, ist die Initialisierung abgeschlossen und das System ist in den üblichen Betriebsbedingungen.

(6) **Write Display Data**

Das Display-Daten-RAM speichert eine Anzeige für das LCD. Um einen Punkt der Matrix des LCDs anzuzeigen, ist ein Datenwert = 1 zu schreiben. Für den anderen Fall, den Aus-Zustand, schreibt man eine „0“. Der Display-Daten-RAM-Adress- und Segment-Ausgang kann durch das ADC-Signal angesteuert werden.

- ADC = H ® Y-Adresse 0: S1 - Y-Adresse 63: S64
- ADC = L ® Y-Adresse 0: S64 - Y-Adresse 63: S1

ADC-Terminal ist zu verbinden mit VDD oder VSS (wird im GLCD nicht verwendet!).

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	0	D7	D6	D5	D4	D3	D2	D1	D0

Schreibt Daten (D0 - D7) in das Display-Daten-RAM. Nach dem Schreiben der Anweisung wird die Y-Adresse automatisch um 1 erhöht.

(7) **Read Display Data**

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	1	D7	D6	D5	D4	D3	D2	D1	D0

Liest Daten (D0 - D7) vom Display Daten-RAM. Nach der Lese-Instruktion wird die Y-Adresse automatisch um 1 erhöht.

6. Grund-Funktionen

Damit die Programmierung in übersichtlicher Weise erfolgen kann, sollen einige Grundfunktionen als modulare Routinen implementiert werden (Quellen-Vorgabe).

`GLCD_init`	Init MCU-Interface
`GLCD_res`	Reset the LCD
`GLCD_rdSt`	Read current status of LCD
`GLCD_wrI`	Write an instruction to LCD
`GLCD_rdD`	Read a data byte from LCD
`GLCD_wrD`	Write a data byte to LCD

Des weiteren

`GLCD_Busy`	Check if LCD is busy
`GLCD_Displ`	Check if LCD display is on
`GLCD_DisplI`	Turn ON/OFF LCD display
`GLCD_SetY`	Set Y address of LCD
`GLCD_SetX`	Set X page of LCD
`GLCD_StL`	Set LCD display's start line

"Primitive" Graphic-Routinen

`GLCD_SetP`	Set a LCD-Pixel
`GLCD_ClrP`	Clear a LCD-Pixel
`GLCD_LineA`	Draw a general Line
`GLCD_LineH`	Draw a horizontally Line
`GLCD_LineV`	Draw a vertical Line
`GLCD_Rect`	Draw a rectangle (H/V)
`GLCD_Circ`	Draw a circle

Im Laufe der Bearbeitung wird sich ergeben, daß von den o.g. Routinen einige anders strukturiert und ebenso auch anders benannt werden. Des Weiteren ist eine Text-Ausgabe notwendig. Der dazu notwenige Zeichen-Font wird in einer Tabelle fest im Programmcode verankert.

6.1. `GLCD_init` – Routine

Für die Kommunikation mit dem Mikrocontroller müssen die von dem LCD genutzten Ports eine geeignete Anfangs-Initalisierung erhalten. Das betrifft:

PortA0	->	Output (RS)	->	LOW (Befehle);		HIGH: Daten
PortA1	->	Output (R/W)	->	HIGH (READ);		LOW: WRITE
PortA2	->	Output (E)	->	LOW	-> HIGH	-> LOW (CLOCK)
PortA5	->	Output (/CS0)	->	HIGH	)	
PortA6	->	Output (/CS1)	->	HIGH	) kein Chip ausgewählt	
PortA7	->	Output (/CS2)	->	HIGH	)	

PortD0	->	Input (DB0)
PortD1	->	Input (DB1)
PortD2	->	Input (DB2)
PortD3	->	Input (DB3)
PortD4	->	Input (DB4)
PortD5	->	Input (DB5)
PortD6	->	Input (DB6)
PortD7	->	Input (DB7)

Das ergibt folgende Code-Sequenz:

```
GLCD_init:    ldi R16, 0b11100111          ; PA0 – PA2, PA5 – PA7 -> Output
              out DDRA, R16
              ldi R16, 0b11100010          ; /CS0 - /CS2 deaktiviert
              out PORTA, R16               ; RS = LOW, R/W = HIGH
              ;
              clr R16                      ; PD0 – PD7 -> Input
              out DDRD, R16          ;
```

Die `PORTD`-Initialisierung ist eigentlich nicht notwendig, da diese beim MC-Start automatisch auf `Input` gesetzt wird.

Bevor irgendwelche Kommunikation mit dem LCD erfolgreich stattfinden kann, muß die Reset-Zeit der LCD-Controller abgewartet werden. Dazu kann pauschal eine Warte-Sequenz (ca. 10ms) eingefügt werden (siehe 4.2.), oder es wird das `Busy`-Flag der LCD-Controller ausgewertet. Damit könnte dann gleichzeitig die Betriebsbereitschaft des GLCDs festgestellt und ggf. mittels LED signalisiert werden (wobei hier vielleicht noch

die Betriebsbereitschaft anderer Komponenten mit berücksichtigt werden sollte) – ist hier (noch) nicht implementiert!

```
Init1:  cbi PORTA, 5            ; setze PORTA5 (/CS0) auf LOW, Contr1 aktiviert
        sbic PORTD, 7           ; überspringe nächsten Befehl, wenn PD7 = LOW
        rjmp Init2              ; bei HIGH = interne Op.
        ;
        sbi PORTA, 5            ; setze PORTA5 (/CS0) wieder auf HIGH, Contr1 deaktiv.
        cbi PORTA, 6            ; setze PORTA6 (/CS1) auf LOW, Contr2 aktiviert
        sbic PORTD, 7           ; überspringe nächsten Befehl, wenn PD7 = LOW
        rjmp Init2              ; bei HIGH = interne Op.
        ;
        sbi PORTA, 6            ; setze PORTA6 (/CS1) wieder auf HIGH, Contr2 deaktiv.
        cbi PORTA, 7            ; setze PORTA7 (/CS2) auf LOW, Contr3 aktiviert
        sbic PORTD, 7           ; überspringe nächsten Befehl, wenn PD7 = LOW
        rjmp init2              ; bei HIGH = interne Op.
        ;
        sbi PORTA, 7            ; setze PORTA7 (/CS2) wieder auf HIGH, Contr3 deaktiv.
        rjmp weiter
        ;
Init2:  rjmp init1              ; wieder von vorn
```

Mit der Controller-Aktivierung durch `/CSx` wir nicht automatisch der Status gelesen und auf den Datenbus an `PORTD` ausgegeben. Hierzu ist `RS` auf LOW und `R/W` auf HIGH zu legen. Erst nach wenigstens 150ns darf der Enable- (Clock-) Impuls angelegt werden, d.h. `E` wird auf HIGH geschaltet und nach einer Wartezeit von mindestens 500ns wieder auf LOW zurückgesetzt. Erst dann liegt das Status-Byte auf dem Datenbus und demzufolge am `PORTD` an (siehe Datenblatt *KS0108b.pdf*).
Da die Prüf-Sequenz `sbic PORTD, 7` / `rjmp Init2` mehrfach verwendet wird, kann diese und das `E`-Timing als UP ausgegliedert werden – vorausgesetzt `RS` und `R/W` sind vorher entsprechend o.g. gesetzt:

```
BusyCl:     rcall delay150
            sbi PORTA, 2            ; E → HIGH
            rcall delay500          ; Impulsbreite
            cbi PORTA, 2            ; E → LOW
                                    ; mit der H/L-Flanke wird das Statusbyte
                                    ; auf den Datenbus gelegt
            sbic PORTD, 7           ; überspringe nächsten Befehl, wenn PD7 = LOW
            rjmp BusyCl             ; sonst bei HIGH = interne Op -> warten bis fertig
            ret
            ;
```

Dadurch vereinfacht sich `Init1` und wird ergänzt mit:

```
Init1:  cbi PORTA, 0            ; RS → LOW
        sbi PORTA, 1            ; R/W → HIGH
        cbi PORTA, 5            ; setze PORTA5 (/CS0) auf LOW
        rcall BusyCl            ; Aufruf UP Busy
        ;
        sbi PORTA, 5            ; setze PORTA5 (/CS0) wieder auf HIGH
        cbi PORTA, 6            ; setze PORTA6 (/CS1) auf LOW
        rcall BusyCl            ; Aufruf UP Busy
        ;
        sbi PORTA, 6            ; setze PORTA6 (/CS1) wieder auf HIGH
        cbi PORTA, 7            ; setze PORTA7 (/CS2) auf LOW
        rcall BusyCl            ; Aufruf UP Busy
        ;
        sbi PORTA, 7            ; setze PORTA7 (/CS2) wieder auf HIGH
        ret                     ; hier weiter
        ;
```

Diese Code-Sequenz ist als UP angelegt, so daß sie problemlos in das Gesamt-Programm integriert werden kann. Sollte die Initialisierung nicht erfolgreich verlaufen – Endlos-Schleife `BusyCl -> rjmp busyCl` – so könnte (sollte) das mit einer LED-Anzeige signalisiert werden – ist (noch nicht) implementiert!

Da die einzelnen Sequenzen der `GLCD_Init`-Routine auch noch in anderen Routinen vorkommen können, wird hier noch weiter verallgemeinert, gleichzeitig erhalten die diesbezüglichen UPs entsprechende Namen:

(letzte gültige Version - Ausnahme BusyCl)

```
GLCD_init:    ldi R16, 0b11100111     ; PA0 – PA2, PA5 – PA7 -> Output
              out DDRA, R16
              ldi R16, 0b11100010     ; /CS0 - /CS2 deaktiviert –
              out PORTA, R16          ; W/R = HIGH, RS = LOW
              ;
              clr R16                 ; PD0 – PD7 -> Input
              out DDRD, R16
              rcall Comm1on           ; Kommunikation mit dem 1. Chip an
              ;
              rcall comm1off          ; Kommunikation mit dem 1. Chip aus
              ;
              rcall Comm2on           ; Kommunikation mit dem 2. Chip an
              ;
              rcall Comm2off          ; Kommunikation mit dem 2. Chip aus
              ;
              rcall Comm3on           ; Kommunikation mit dem 3. Chip an
              ;
              rcall Comm3off          ; Kommunikation mit dem 3. Chip aus
              ;
              ret                     ; Rücksprung aus der Routine (Ende)
              ;
Comm1on:      cbi PORTA, 5            ; setze PORTA5 (/CS0) auf LOW
              rcall BusyCl            ; Aufruf UP BusyCl
              ret
              ;
```

```
Comm1off:     sbi PORTA, 5            ; setze PORTA5 (/CS0) wieder auf HIGH
              ret
              ;
Comm2on:      cbi PORTA, 6            ; setze PORTA6 (/CS1) auf LOW
              rcall BusyCl            ; Aufruf UP BusyCl
              ret
              ;
Comm2off:     sbi PORTA, 6            ; setze PORTA6 (/CS1) wieder auf HIGH
              ret
              ;
Comm3on:      cbi PORTA, 7            ; setze PORTA7 (/CS2) auf LOW
              rcall BusyCl            ; Aufruf UP BusyCl
              ret
              ;
Comm3off:     sbi PORTA, 7            ; setze PORTA7 (/CS2) wieder auf HIGH
              ret
              ;
BusyCl:       rcall delay150
              sbi PORTA, 2            ; E → HIGH
              rcall delay500          ; Impulsanfang, Impulsbreite >500ns
              cbi PORTA, 2            ; E → LOW, Impulsende
                                      ; mit der H/L-Flanke wird das Statusbyte
                                      ; auf den Datenbus gelegt

              sbic PIND, 7            ; überspringe nächsten Befehl, wenn PD7 = LOW
              rjmp BusyCl             ; sonst bei HIGH = interne Op -> warten bis fertig
              ret
              ;
; nachfolgende UPs werden z.Zt. nicht benötigt
delay150:     ldi R20, 255            ; Zeitschleife ca. 150µs
              ldi R21, 8
              rcall time1
              ret
              ;
delay500:     ldi R20, 255            ; Zeitschleife ca. 500µs
              ldi R21, 26
              rcall time1
              ret
              ;
time1:        dec R20                 ;Zeitschleifengenerator
              brne time1
              ;
              dec R21
              brne time1
              ;
              ret
```

Obwohl es so aussieht, als ob durch die vielen `rcall`-Aufrufe der Code sehr aufgebläht wurde, können doch nun vorteilhaft im Weiteren die einzelnen UPs auch separat eingesetzt werden.

Im Erfolgsfall wäre als Nächstes der Befehl zum Einschalten des GLCDs zu geben.

BusyCL (Read) ist nicht korrekt! Siehe Timing…

In dem Dokument *KS0108.pdf* gibt es zwei Impulsdiagramme (2x Fig.3 - MPU write timing), die für die nachfolgenden Betrachtungen relevant sind, wobei zu beachten ist,

daß es sich beim 1. Bild um den Write- und beim 2. um den Read-Vorgang handelt – erkennbar an dem R/W-Verlauf! Die entsprechenden Zeitwerte sind in der voran stehenden Tabelle aufgeführt.

(3) MPU Interface

Chatacteristic	Symbol	Min	Typ	Max	Unit
E Cycle	t_C	1000	-	-	ns
E High Level Width	t_{WH}	450	-	-	ns
E Low Level Width	t_{WL}	450	-	-	ns
E Rise Time	t_R	-	-	25	ns
E Fall Time	t_F	-	-	25	ns
Address Set-Up Time	t_{ASU}	140	-	-	ns
Address Hold Time	t_{AH}	10	-	-	ns
Data Set-Up Time	t_{SU}	200	-	-	ns
Data Delay Time	t_D	-	-	320	ns
Data Hold Time (Write)	t_{DHW}	10	-	-	ns
Data Hold Time (Read)	t_{DHR}	20	-	-	ns

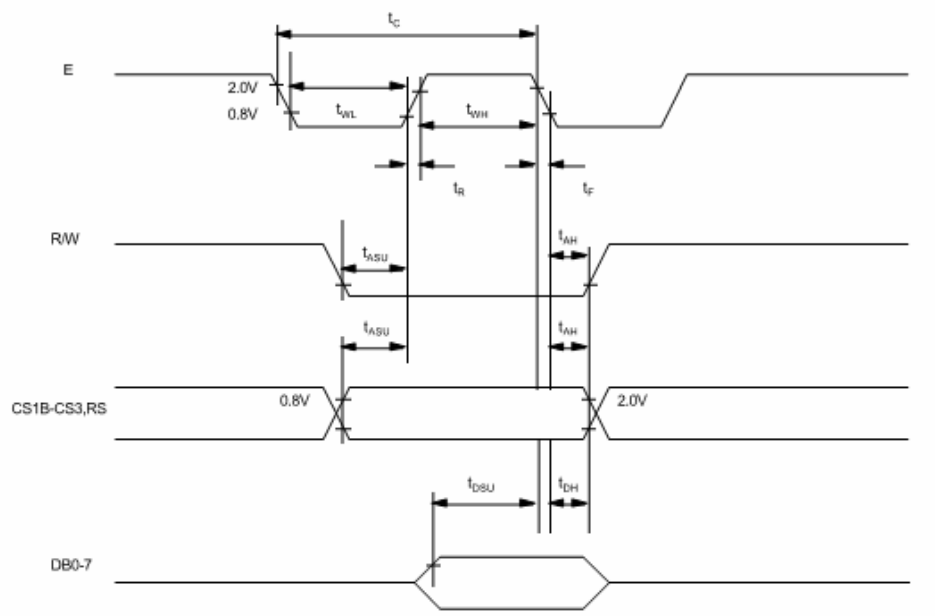


Bild : Write-Vorgang

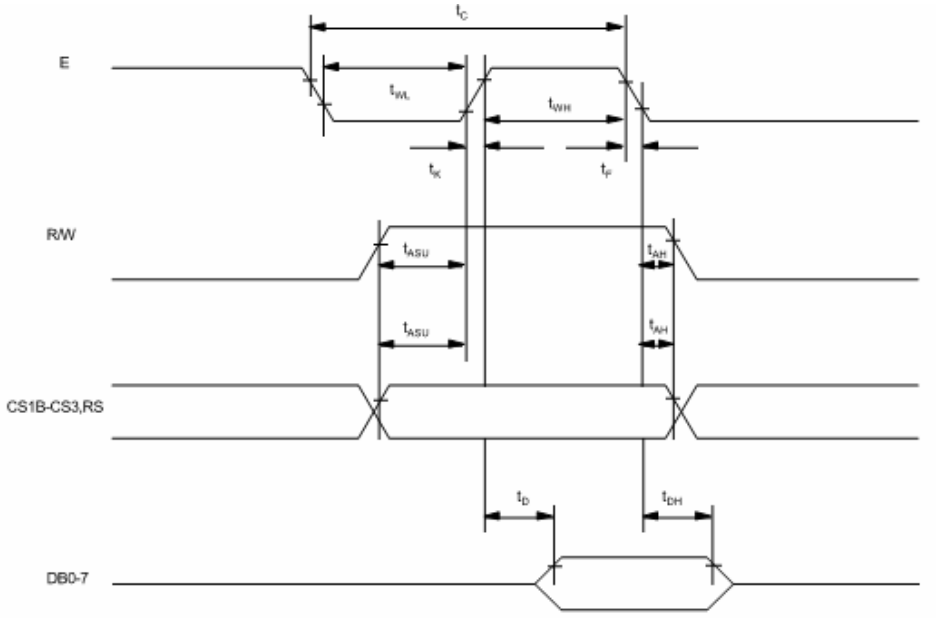


Bild : Read-Vorgang

Zunächst wurde ein Übertragungsfehler korrigiert:
Die Timing-Bedingungen sind in *ns* angegeben und nicht in *ms*! Dadurch ergeben sich ganz andere Verhältnisse bei einem Read- bzw. auch beim Write-Vorgang. Die `delay150`- bzw. `delay500`-UPs werden nicht mehr gebraucht.
Für die 150ns Vorhaltezeit R/W und /CS reichen die 3 Takte eines `rcall` = 3x125ns = 375ns völlig aus, hingegen die mind. 500ns für den HIGH-Teil des GLCD-Taktes `E` mit 4x `nop` + ein weiterer Befehl gebildet werden. Nun ergibt sich die Schwierigkeit, daß die Daten-Gültigkeit nur 20ns nach der H/L-Flanke von `E` definiert ist, was wiederum zur Folge hat, daß der nachfolgender Befehl zur Datenauswertung, z.B. `sbic PORTD, 7` nach 125ns wahrscheinlich in's Leere greift!
Wenn man von der L/H-Flanke von `E` ausgeht, stehen andererseits die Daten aber auch erst nach einer Verzögerung (Delay) von wenigstens 320ns gültig zur Verfügung. Deshalb muß die Datenauswertung vor dem H/L-Befehl eingefügt werden. Beim `sbic PORTD, 7` ist allerdings nicht zu vergessen, daß auch in dem übersprungenen Zweig der GLCD-Takt mit dem H/L-Befehl abzuschließen ist.
Die betreffende Code-Sequenz ändert sich dann wie folgt:

```
BusyCl:     ;rcall delay150         ; wird nicht benötigt, da 3x125ns für rcall BusyCl
                                    ; > 150ns
            sbi PORTA, 2            ; E → HIGH
            ;rcall delay500         ; Impulsanfang, Impulsbreite >500ns
            nop
            nop
            nop                     ; nach 320ns liegen die gelesenen Daten
                                    ; auf dem Datenbus vor
            sbic PIND, 7            ; überspringe nächsten Befehl, wenn PD7 = LOW
            rjmp BusyCle            ; sonst bei HIGH = interne Op -> warten bis fertig
            nop
            cbi PORTA, 2            ; E → LOW, Impulsende
            ret
            ;
BusyCLe:    cbi PORTA, 2            ; E → LOW, Impulsende
            nop                     ; für mind. 500ns LOW
            nop
            nop
            nop
            rjmp BusyCl
            ;
```

Diese spezielle `BusyCl`-Routine wäre demnach nicht für andere Zwecke zu gebrauchen! Da im Weiteren Read-Vorgänge auch noch für Anderes notwendig sind, ist eine direkte Auswertung am Port ungünstig. Weil die Daten kurz nach der H/L-Flanke nicht mehr zur Verfügung stehen, ist es universell besser, diese in einem Register abzulegen – hier `r17`.

Die nochmalige Korrektur sieht dann so aus:

```
BusyCl:     ;rcall delay150         ; wird nicht benötigt, da 3x125ns für rcall BusyCl
                                    ; > 150ns
            sbi PORTA, 2            ; E → HIGH
            ;rcall delay500         ; Impulsanfang, Impulsbreite >500ns
            nop
            nop
            nop                     ; nach 320ns liegen die gelesenen Daten
                                    ; auf dem Datenbus vor
            in r17, PIND            ; akt. PIND-Daten, auch noch nach E-LOW
            sbrc r17, 7             ; überspringe nächsten Befehl, wenn r17:7 = LOW
            rjmp BusyCle            ; sonst bei HIGH = interne Op -> warten bis fertig
            cbi PORTA, 2            ; E → LOW, Impulsende
            ret
            ;
BusyCLe:    cbi PORTA, 2            ; E → LOW, Impulsende
            nop                     ; für mind. 500ns LOW
            nop
            nop
            rjmp BusyCl             ; Schleife schließen
            ;
```

6.1.1. Test UP `GLCD_init`

Jedoch sollte (muss) vorab das UP `GLCD_init` unter Einbeziehung der Hardware getestet werden.

Da der ATMega128 die JTAG-Schnittstelle unterstützt, bietet sich hier an, die AVR-Programmierung und die anschließende schrittweise Ausführung des Codes im Hardware-unterstützten JTAG-Simulator durchzuführen.

Realisiert wird das durch den „AVR-JTAG-USB“ von www.olimex.com, einem JTAG-ICE-kompatiblen Clone des ATMEL-Gerätes. –

Leider funktioniert das Gerät nicht (mehr) …

Es wurde die Bestellung eines „USBProg 3.0“ von www.embedded-projects.net vorkonfektioniert als „JTAG ICE mkII“ und fertig eingespielter Firmware bestellt.

Die Lieferung ist noch nicht erfolgt …

6.1.1.1. Test mit dem DRAGON-Board

Eine Alternative sollte das DRAGON-Board von ATMEL bieten. Ein solches liegt vor und nach einigen Ansfangsschwierigkeiten konnte es auch über die JTAG-Schnittstelle mit dem GLCD-Board (ATMegaEvoBoard) angesprochen werden.

Jedoch zeigte sich, daß die im Diassembler-Fenster vonstatten gehende Simulation nicht korrekt ablief. Die Rücklesung (oder auch die Darstellung) des Hex-Codes erfolgte offensichtlich fehlerbehaftet, so daß der Simulator bis zum Ende des Programmspeichers durchlief und dann automatisch abgebrochen wurde. Irgendwelche Step-Funktionen etc. waren nicht ausführbar. Eine Erklärung konnte nicht gefunden werden. In der DRAGON-Hilfe wird vom Setzen der DWEN-Fuse gesprochen. Dies gibt es aber beim ATmega128 nicht, dafür die Fuse OCDEN (ON-Chip-Debugging ON). Das Setzen wird aber immer wieder zurückgenommen, wenn der JTAG-Simulator aktiviert wird.

6.1.1.2. Test mit dem Original JTAGICEmkII

Als nächster Versuch wurde ein Original JTAGICEmkII von ATMEL eingesetzt. Auch bei diesem Teil gab es Anfangsschwierigkeiten, d.h. es konnte zunächst keine Kommunikation via USB hergestellt werden. Im Windows-Gerätemanager wurde auch – in Gegensatz zu dem Olimex-Nachbau – kein neues enumeriertes USB-Gerät gefunden. Erst nach einem Upgrade vom ATMEL-Studio heraus konnte wenigstens die Programming-Funktion aktiviert werden. Die USB-Kommunikation läuft offensichtlich immer nur solange, wie die Befehls-/Datenübertragung läuft, ansonsten befindet sich das USB-Grät im „Idle-Mode“, d.h. es ist deaktiviert und wird somit in Gerätemanager als virtuelles COM-Port nicht angezeigt.

Damit wurde z.B. die aktuelle Fuse-Programmierung überprüft: JTAGEN und SPIEN waren enabled, d.h. "angehakt", OCDEN nicht. Das wurde nachgeholt, dann das Programmiert-Tool beendet.

Nachfolgend wurde unter "Debug/Select Platform and device..." JTAGICEmkII, ATmega128 und USB ausgewählt. Das Ergebnis ist in der Statusleiste zu sehen.

So weit so gut - wenn jetzt aber "Start Debugging" gedrückt wird, so passiert erst mal gar nichts - außer, daß offensichtlich der JTAG-Adapter in laufender Aktion ist (erkennbar am schnellen Flackern der linken grünen LED und "Running" in der Statusleiste). Wird jetzt "BREAK" gedrückt, so wird ein Diassembler-Fenster aufgemacht, der gelbe Zeilenzeiger springt auf die letzte Programmspeicher-Zelle +0000FFFF und "Stopped" wird in der Statuszeile angezeigt. Von dort läßt sich dann nichts mehr bewegen - kein Reset oder sonstige Debug-Funktionen sind wirksam, außer mit "Stopp Debugging" das ganze abzubrechen.

Wird hingegen der „AVR-Simulator“ ausgewählt, so funktioniert das Debugging einwandfrei: nach "Start Debugging" und Darstellung eines Fortschrittsbalkens unten über der Statuszeile steht dann der gelbe Zeilenzeiger ordnungsgemäß am Beginn des Programms und man kann mit den zur Verfügung stehenden Debug-Befehlen, wie z.B. "Step" den ASM-Code im einzelnen durchgehen.

6.2. `GLCD_ON` – Routine

Da nunmehr die 3 Controller des GLCDs bereit sind Befehle zu empfangen, aber auf dem angeschlossenen GLCD noch nichts zu sehen ist (-> testen! – richtig: nur kurzes Aufflackern beim Einschalten, dann „dunkel“), sollte nun der Befehl zum Einschalten des GLCDs gegeben werden.

Dabei ist folgendes zu beachten:

- es müssen immer alle 3 Controller angesprochen werden
- es ist vielleicht zweckmäßig den Einschalt-Befehl erst nach dem Löschen der Daten-RAMs zu geben, damit eine unnötige Darstellung von Zufallsdaten gar nicht erst erfolgt (wird ggf. später noch nachgeholt).
- Da die Routine als UP realisiert werden soll (-> Verwendung nicht nur beim `Init`-Prozess), kann nicht davon ausgegangen werden, daß die Controller empfangsbereit für Befehle, bzw. die vom GLCD gesendeten Daten gültig sind. Wenn die Routine außerhalb von `GLCD_Init` aufgerufen wird, kann auch nicht ausgeschlossen werden, daß das GLCD bereits eingeschaltet ist.

Deshalb muß hier außer dem `GLCDon/off`-Bit jedes Mal die „Busy“-Eigenschaft gestestet werden.

```
GLCD_on:    rcall comm1on          ; Kommunikation mit 1.Chip an – akt. Daten in r17
                                   ; Status an Bit5 prüfen
            sbrs r17, 5            ; wenn Bit 5 = 0: Display bereits eingeschaltet
                                   ; bei HIGH nächsten Befehl überspringen
            rjmp GLCD_on1          ; comm1off zum Schließen Chip1
            ;
            cbi PORTA, 1           ; R/W auf LOW setzen – Write
            cbi PORTD, 5           ; Chip 1 - LCD ein
            rcall BusyCl           ; Takt – Befehl senden
            ;
            sbi PORTA, 1           ; R/W auf HIGH setzen – Read
            sbi PORTD, 5           ; zurücksetzen
GLCD_on1:   rcall comm1off         ; Chip 1 schließen
            ;
```

```
                rcall comm2on           ; Kommunikation mit 2.Chip an – akt. Daten in r17
                                        ; Status an Bit5 prüfen
                sbrs r17, 5             ; wenn Bit 5 = 0: Display bereits eingeschaltet
                                        ; bei HIGH nächsten Befehl überspringen
                rjmp GLCD_on2           ; comm2off zum Schließen Chip2
                ;
                cbi PORTA, 1            ; R/W auf LOW setzen – Write
                cbi PORTD, 5            ; Chip 2 - LCD ein
                rcall BusyCl            ; Takt – Befehl senden
                ;
                sbi PORTA, 1            ; R/W auf HIGH setzen – Read
                sbi PORTD, 5            ; zurücksetzen
GLCD_on2:       rcall comm2off          ; Chip 2 schließen
                ;
                rcall comm3on           ; Kommunikation mit 3.Chip an – akt. Daten in r17
                                        ; Status an Bit5 prüfen
                sbrs r17, 5             ; wenn Bit 5 = 0: Display bereits eingeschaltet
                                        ; bei HIGH nächsten Befehl überspringen
                rjmp GLCD_on3           ; comm3off zum Schließen Chip3
                ;
                cbi PORTA, 1            ; R/W auf LOW setzen – Write
                cbi PORTD, 5            ; Chip 3 - LCD ein
                rcall BusyCl            ; Takt – Befehl senden
                ;
                sbi PORTA, 1            ; R/W auf HIGH setzen – Read
                sbi PORTD, 5            ; zurücksetzen
GLCD_on3:       rcall comm3off          ; Chip 3 schließen
                ;
                ret
```

Damit müsste das GLCD eingeschaltet sein, d.h. die in den internen RAM's der 3 Chips abgespeicherten Daten müssten nun als Pixel eingeschaltet zu sehen sein.

6.2.1. Test UP `GLCD_on`

Nach dem Einspielen der Firmware einschließlich der Funktionen `GLCD_init` und `GLCD_ON` ist nach *Reset* oder dem Zuschalten der Stromversorgung nichts zu sehen. Auch das anfängliche Flackern auf den Bildschirm ist nun nicht mehr vorhanden. Inwieweit die beiden o.g. Funktion korrekt laufen, kann so ohne weiteres nicht ermittelt werden.

Ein „Trockentest" mit dem AVR-Simulator im ATMEL-Studio könnte zwar in gewissem Maße Aufschluss geben, ist jedoch sehr zeitaufwändig und außerdem lassen sich damit nicht die konkreten Anschalt-Bedingungen des GLCD's prüfen. Eine „In-Circuit-Emulation" über die JTAG-Schnittstelle und eines dementsprechenden Prüfadapters sollten dann schon aufschlussreichere Ergebnisse bringen.

Die JTAG-Simulation muß derzeit ausgesetzt werden, solange kein funktionierender Prüfadapter zur Hand ist, bzw. die Debugging-Funktion im AVR-Studio nicht richtig funktioniert – siehe 6.1.1. Test UP `GLCD_init`.

Nach den Erkenntnissen unter **6.3.1.** muß die Routine `GLCD_on` noch mal überarbeitet werden (Befehl fehlt noch???) … -> ist erledigt!

6.2.1.1. Alternative Test-Instrumentarien

Da z.Zt. kein funktionierendes Equipment für einen In-Circuit-Test zur Verfügung steht, muß mit Hilfslösungen gearbeitet werden. Dazu werden in gewohnter Weise an relevanten Stellen im Code sog. Test-Sequenzen eingebaut, wo dann über einen Prüfport und LED-Anzeige bestimmte Zustände signalisiert werden können.
Das Prüf-Port PORTC muß zunächst initialisiert werden:

```
            clr r16                 ; alle C-Pins auf LOW
            out PORTC, r16
            ldi r16, 0b11111111     ; alle auf Ausgabe
            out DDRC, r16
```

So kann z.B. das zum UP `BusyCl` Erwähnte eingeführt werden in `BusyCl_new`:

```
BusyCl:     ...

BusyCLe:    cbi PORTA, 2            ; E → LOW, Impulsende
                                    ; für mind. 500ns LOW –
                                    ; wird durch nachfolgende Befehlssequenz erfüllt
            inc r18                 ; Hochzählen bis 255 -> 0
            breq BusyClo
            ;
            nop
            nop
            rjmp BusyCl             ; Schleife schließen
;
BusyClo:    sbi PORTC, 7            ; PC7 auf HIGH – LED_7 an
            rjmp BusyClo            ; Endlosschleife – raus nur über Reset!
```

Andererseits besteht auch die Möglichkeit die Abarbeitung einzelner UPs zu signalisieren. Dazu wird z.B. in `GLCD_main` folgendes eingeführt:

```
GLCD_main:  sbi PORTC, 0            ; nur zum Test: Init ok – LED_0
            ;
            rcall GLCD_init         ; UP zur AVR-Port- und LCD-Initialisierung
            sbi PORTC, 1            ; nur zum Test; GLCD_init ok – LED_1
            ;
            rcall GLCD_on           ; Graphic-LCD einschalten
            sbi PORTC, 2            ; nur zum Test: GLCD_on ok – LED_2
            ;
```

Das ist z.Zt. implementiert und die korrekte Funktion konnte soweit nachgewiesen werden. Die LEDs 0 bis 2 gehen an (ok), aber auch die Kontroll-LED_7, wobei es noch nicht geklärt ist, warum sich diese angeschaltet wird, d.h. sich der LCD-Controller „aufgehängt“ hat.

6.2.1.2. Test mit JTAGICEmkII

Das Problem mit dem JTAG-Adapter scheint nun gelöst zu sein, jedenfalls konnte bei Verwendung des Original JTAGICEmkII von ATMEL die Ursache der inkorrekten Funktionsweise der Debugger-Funktion gefunden werden:
Die Stromversorgung auf dem „ATMegaEvo-Board“ war durch einen zu klein bemessenen Ladekondensator nach dem Graetz-Gleichrichter zu unstabil, d.h. der nachfolgende Linearregler LM317 konnte das nicht mehr ausregeln und somit ging das Board alle 25ms in den RESET-Zustand. Dazu gibt es eine Diskussion im Forum von www.mikrocontroller.net/forum/mikrocontroller-elektronik/avr - Suche AVR JTAGICEmkII Debug-Probleme.

Der anschließende Testlauf – STEP, wie auch RUN über den JTAGICE verliefen nun scheinbar erfolgreich. Eine detaillierte Untersuchung wurde zunächst nicht durchgeführt, da offensichtliche Fehler nicht erkennbar waren.
Das Problem, daß sich der LCD-Controller „aufgehängt“ hat, konnte durch eine Programmänderung am Schluß beseitigt werden:

```
GLCD_end:     rjmp GLCD_end              ; Endlosschleife
```

Offensichtlich ist die vorhergehende Loop-Schleife zu `GLCD_main` unverträglich mit der Initialisierung – ohne das jetzt näher untersuchen zu wollen.

Es sieht so aus, als ob auf dem GLCD alle Pixel eingeschaltet sind.
Da von der nachfolgenden Löschung aller Pixel mit `GLCD_clr` auf dem Display nichts zu sehen war, muß evtl. doch davon ausgegangen werden, daß das Display nicht eingeschaltet worden ist. Eine genauere STEP-by-STEP-Untersuchung soll dazu Klarheit bringen.
In der Tat kam keine Einschaltung zu Stande, weil im UP `GLCD_on` der Befehl „Display-Einschalten“ falsch, bzw. überhaupt nicht vorhanden war.

Aus diesem Grunde wurde `GLCD_on` verändert:

```
GLCD_on:      rcall comm1on      ; Kommunikation mit 1.Chip an - akt. Daten in r17
                                 ; Status an Bit5 prüfen
              sbrs r17, 5        ; wenn Bit 5 = 0: Display bereits eingeschaltet
                                 ; bei HIGH nächsten Befehl überspringen
              rjmp GLCD_on1      ; Comm1off zum Schließen Chip1
              ;
              cbi PORTA, 1       ; R/W auf LOW setzen - Write
;             cbi PORTD, 5       ; Chip 1 - LCD ein :-> FALSCH!
              ;
              ldi r16, 0b00111111      ; Befehl DISPLAY_on
              out PORTD, r16
              rcall BusyCl       ; Takt
              ;
              sbi PORTA, 1       ; R/W auf HIGH setzen - Read
              ;sbi PORTD, 5      ; zurücksetzen – unnütz!
GLCD_on1:     rcall comm1off     ; Chip 1 schließen
              ;
              rcall comm2on      ; Kommunikation mit 2.Chip an - akt. Daten in r17
                                 ; Status an Bit5 prüfen
              sbrs r17, 5        ; wenn Bit 5 = 0: Display bereits eingeschaltet
                                 ; bei HIGH nächsten Befehl überspringen
              rjmp GLCD_on2      ; Comm2off zum Schließen Chip2
              ;
              cbi PORTA, 1       ; R/W auf LOW setzen - Write
;             cbi PORTD, 5       ; Chip 2 - LCD ein  :-> FALSCH!
              ;
              ldi r16, 0b00111111      ; Befehl DISPLAY_on
              out PORTD, r16
              rcall BusyCl       ; Takt
              ;
              sbi PORTA, 1       ; R/W auf HIGH setzen - Read

              ;sbi PORTD, 5      ; zurücksetzen – unnütz!
GLCD_on2:     rcall comm2off     ; Chip 2 schließen
              ;
              rcall comm3on      ; Kommunikation mit 3.Chip an - akt. Daten in r17
                                 ; Status an Bit5 prüfen
              sbrs r17, 5        ; wenn Bit 5 = 0: Display bereits eingeschaltet
                                 ; bei HIGH nächsten Befehl überspringen
              rjmp GLCD_on3      ; comm3off zum Schließen Chip3
              ;
              cbi PORTA, 1       ; R/W auf LOW setzen - Write
;             cbi PORTD, 5       ; Chip 3 - LCD ein :-> FALSCH!
              ;
              ldi r16, 0b00111111      ; Befehl DISPLAY_on
              out PORTD, r16
              rcall BusyCl       ; Takt
              ;
              sbi PORTA, 1       ; R/W auf HIGH setzen - Read

              ;sbi PORTD, 5      ; zurücksetzen – unnütz!
GLCD_on3:     rcall comm3off     ; Chip 3 schließen
              ;
              ret
```

Diesmal wurde das Einschalten des Displays = Anzeige aller Pixel beobachtet. Es erwies sich als notwendig, den Kontrast entsprechend einzuregulieren, so daß beim Display_off auch keine Pixel mehr zu sehen sind.

6.3. UP `GLCD_clr`

Die Bearbeitung dieser Funktion wird derzeit ausgesetzt, solange keine Notwendigkeit zur Anwendung besteht.

Da nun feststeht, daß alle Pixel nach der GLCD-Initialisierung zunächst eingeschaltet sind (so sieht es jedenfalls aus), besteht nun ggf. doch die Notwendigkeit diese zu löschen.

Funktional muß dazu in die 3 GLCD_RAM-Bereiche (0,0; 63,63), (64,0; 127,63), (128,0; 191,63) jeweils „0“ eingetragen werden.

Dazu muß zunächst die Startadresse bereitgestellt werden in dem UP:

```
GLCD_adrSt:   ldi r16, 0b01000000     ; Befehl “SetAdrY_0”
              out PORTD, r16
              ldi R16, 0b11100000     ; /CS0 - /CS2 deaktiviert -
              out PORTA, R16          ; R/W = LOW, RS = LOW (Befehl)
              ;
              ldi r16, 0b11111111     ; PORTD auf Ausgang
              out DDRD, r16           ; damit liegt der Befehl am GLCD an
              rcall Comm1on           ; Kommunikation mit dem 1. Chip an
              ;
              rcall Comm1off          ; Kommunikation mit dem 1. Chip aus
              ;
              rcall Comm2on           ; Kommunikation mit dem 2. Chip an
              ;
              rcall Comm2off          ; Kommunikation mit dem 2. Chip aus
              ;
              rcall Comm3on           ; Kommunikation mit dem 3. Chip an
              ;
              rcall Comm3off          ; Kommunikation mit dem 3. Chip aus
              ;
```

Da die Sequenz `rcall Comm1on/rcall comm1off` usw. schon im UP `GLCD_init` vorkommt kann diese – nach Einführung des Labels `GLCD_comm: -` vereinfacht werden:

```
              rcall GLCD_comm
```

Weiter geht es dann mit:

```
              ldi r16, 0b10111000     ; Befehl “SetAdrX_0“
              out PORTD, r16
              ;
              rcall GLCD_comm
              ;
              ret
              ;
```

Damit sind die 3 Controller-Chips auf dem GLCD für die Datenaufnahme vorbereitet.

Im UP `GLCD_clr` geht es dann weiter mit:

```
            ldi r16, 0b11100001     ; /CS0 - /CS2 deaktiviert -
            out PORTA, r16          ; R/W = LOW, RS = HIGH (Daten)
            clr r16                 ; alle Pixel aus
            ;com r16                ; oder alle Pixel an
                                    ; werden beide Befehle auskommentiert,
                                    ; wird das aktuelle Pixelmuster in r16 übertragen
            out PORTD, r16
```

Daß mit jeder Datenübertragung der interne AdresszählerX der Chips automatisch inkrementiert, ist zwar für den Übertragungsprozeß einfacher, aber trotzdem muß die Anzahl der übertragenen Bytes kontrolliert werden. Das geht am besten mit einem virtuellen externen Adresszähler `r19`, welcher jedoch im Gegensatz zu dem internen Chip-Adresszähler abwärts zählt. Damit ist ein günstigeres Abbruch-Kriterium gegeben.

```
            ldi r19, 64             ; virtuellen Chip-Adresszähler laden
GLCD_clr1:  rcall GLCD_comm1        ; interner Adresszähler + 1
            ;
            rcall GLCD_comm1off
            ;
            dec r19
            brne GLCD_clr1          ; solange bis r19 = 0
            ;
            ldi r19, 64             ; virtuellen Chip-Adresszähler laden
GLCD_clr2:  rcall GLCD_comm2        ; interner Adresszähler + 1
            ;
            rcall GLCD_comm2off
            ;
            dec r19
            brne GLCD_clr2          ; solange bis r19 = 0
            ;
            ldi r19, 64             ; virtuellen Chip-Adresszähler laden
            ;
GLCD_clr3:  rcall GLCD_comm3        ; interner Adresszähler + 1
            ;
            rcall GLCD_comm3off
            ;
            dec r19
            brne GLCD_clr3          ; solange bis r19 = 0
            ;
```

Damit sollten die ersten 8 Zeilen = 1.Page des GLCD gelöscht sein.
Für die weiteren 6 Pages = 56 Zeilen muß das gleiche Prozedere ebenfalls durchgeführt werden, wobei es kein Auto-Increment beim Page-Zähler gibt. Es muß also nach jeder Page-Sequenz die neue Page-Adresse separat übertragen werden.
Grundsätzlich wäre zu untersuchen, ob es günstiger ist, die Übertragung Page-weise für alle 3 Chips nacheinander – wie oben begonnen – oder Chip-weise die Pages vollständig abzuarbeiten.

In jedem Fall sollte aus Vereinfachungs-Gründen wiederholte Funktionalität als weitere UP ausgegliedert werden.

6.3.1. Testergebnis `GLCD_clr`

Nach dem Laden des Programms konnte keine Veränderung der GLCD-Ausgabe festgestellt werden. Offensichtlich gibt es auch Kommunikationsprobleme, denn die Busy-LED(7) leuchtet (-> „aufgehangen“ in Endlosschleife)

Der Test über JTAG kommt zu folgendem Ergebnis:

Im UP `BusyCL` kommt es zu Konflikten, da je nach Situation das Datenrichtungsregister `DDRD` von `PORTD` nicht immer auf Eingang geschaltet ist und außerdem nicht sichergestellt ist, daß `R/W` auf HIGH liegt, somit kein Status eingelesen werden kann.

Des Weiteren wird mit dem Einlesen des Status der eigentliche Befehl nicht an den GLCD-Controller übermittelt. Das UP `BusyCL` muß also in zwei Sequenzen aufgeteilt werden:

- `BusyCL` - zur Ermittlung der Befehls-Bereitschaft des Chips
- `InstrCL` - zur Übermittlung des Befehls

Das UP `BusyCL` wurde dementsprechend ergänzt:

```
BusyCl:     in r15, PORTA     ; PORTA-Status sichern
            in r14, DDRD      ; Datenrichtung PORTD sichern
            ;
            clr r16           ; PORTD auf Eingang umschalten,
                              ; - Inhalt bleibt erhalten
            out DDRD, r16
            cbi PORTA, 0      ; RS = 0: Befehl
            sbi PORTA, 1      ; R/W = HIGH (Read-Status)
            ;rcall delay150   ; wird nicht benötigt, da 3x125ns für rcall BusyCl
                              ; > 150ns
            sbi PORTA, 2      ; E -> Taktimpuls HIGH
            ;rcall delay500   ; Impulsanfang, Impulsbreite >500ns
            nop
            nop
            nop               ; nach 320ns liegen die gelesenen Daten
                              ; auf dem Datenbus vor
            in r17, PIND      ; akt. PIND-Daten sichern, auch noch nach E-LOW
            sbrc r17, 7       ; überspringe nächsten Befehl, wenn r17:7 (Busy) = LOW
            rjmp BusyCle      ; sonst bei HIGH = interne Op -> warten bis fertig
            cbi PORTA, 2      ; E -> LOW, Impulsende
            ;
InstrCL:    out PORTA, r15    ; PORTA-Status wieder herstellen
            out DDRD, r14     ; Datenrichtung PORTD wieder herstellen
            sbi PORTA, 2      ; E -> Taktimpuls HIGH
            nop
            nop
            nop
```

```
                cbi PORTA, 2        ; E -> LOW, Impulsende
                ;
                ret
                ;
BusyCLe:        cbi PORTA, 2        ; E -> LOW, Impulsende
                                    ; für mind. 500ns LOW -
                                    ; wird durch nachfolgende Befehlssequenz erfüllt
                inc r18             ; Hochzählen bis 255 -> 0
                breq BusyClo
                ;
                nop
                nop
                rjmp BusyCl         ; Schleife schließen
;
BusyClo:        sbi PORTC, 7        ; PC7 auf HIGH - LED an
                rjmp BusyClo        ; Endlosschleife - raus nur über Reset!
;
```

Ggf. kann (muß) die Routine `InstrCl` als separates UP ausgegliedert werden. Dann ist jedoch die Datensicherung in `r15, r14` entsprechend vorzubereiten.

Der weitere Test über JTAG ergab, daß nur das erste Pixel-Byte ganz links gelöscht wurde. Alles andere blieb so, wie es ist. Somit wird also auch die Information nur an den 1. GLCD-Controller übertragen und die anderen beiden „gehen leer aus“ ???

Bei einem STEP/RUN-Test mit Haltepunkt bei `brne GLCD_pg1` im UP `GLCD_page` konnte die Wirksamkeit der Routine nachgewiesen werden. Wird hingegen die Schleife `GLCD_pg1` ohne Unterbrechung abgearbeitet, so ist kein Löschen zu sehen. Das Einbringen eines 150µs-Delay gleich nach `Comm_off` (auch bei `GLCD_pg2` und `GLCD_pg3` war dann erfolgreich. Warum???

Bemerkenswert ist, daß ein Reset oder ein kurzes Unterbrechen der Stromversorgung die RAM-Zellen der GLCD-Controller nicht dazu veranlasst, ihre Informationen zu „vergessen“, d.h. wieder auf HIGH zu gehen. Erst eine längere Pause von mehr als 1 Minute ist dazu notwendig!

Ergänzung der Routine `GLCD_pages`

Wie schon weiter oben beschrieben, ist die Page-Adressierung nicht automatisch inkrementierend, so wie bei der X-Adressierung.

Deshalb wurde nachfolgende Erweiterung von `GLCD_pages` vorgenommen:

```
GLCD_pages:   ldi r20, 8              ; virtuellen Chip-Pagezähler laden
GLCD_pgs1:    rcall GLCD_page
              ;
```

```
                clz                     ; sicherheitshalber Z-Flag löschen
                dec r20
                brne GLCD_pgs2          ; solange bis r20 = 0
                ;
                ret
                ;
GLCD_pgs2:      in r16, PORTD           ; Pixeldaten sichern (wichtig!)
                push r16
                ;
                mov r16, r20            ; sichere r20
                com r16                 ; invertiere Wert
                inc r16                 ; +1
                cbr r16, 0b01000000     ; lösche Bit6, damit "SetAdrX"-Befehl ok
                out PORTD, r16
                ;
                cbi PORTA, 0            ; RS = LOW (Befehl)
                rcall Comm1on           ; Kommunikation mit dem 1. Chip an
                ;
                rcall Comm1off          ; Kommunikation mit dem 1. Chip aus
                ;
                rcall Comm2on           ; Kommunikation mit dem 2. Chip an
                ;
                rcall Comm2off          ; Kommunikation mit dem 2. Chip aus
                ;
                rcall Comm3on           ; Kommunikation mit dem 3. Chip an
                ;
                rcall Comm3off          ; Kommunikation mit dem 3. Chip aus
                ;
                pop r16                 ; Pixeldaten zurück
                out PORTD, r16
                sbi PORTA, 0            ; RS = HIGH (Daten)
                rjmp GLCD_pgs1          ; nächste Page übertragen
                ;
```

Nunmehr sind die Routinen des UP `GLCD_clr` vollständig, d.h. die Pixel-Löschung nach dem Einschalten war erfolgreich auf der ganzen GLCD-Fläche. Jedoch ist die Effizienz noch nicht so ganz befriedigend, weil trotz einer Ausführungszeit von ??? ms der ganze Vorgang zu beobachten ist. Hier gibt es sicherlich noch Optimierungsbedarf.
Ungeklärt ist vor allem noch, woraus sich die Notwendigkeit der Einschiebung einer 150ms-Verzögerung (`delay150`) in die `GLCD_pg`–Schleife nach jedem RAM-Zugriff ergibt (siehe weiter oben die Bemerkung zum STEP/RUN-Test). Im Datenblatt des GLCD-Controllers KS108b steht eine minimale Vorhaltezeit von 140ns für `R/W`, `/CS` und `RS`, bevor die steigende Taktflanke `E` anliegt. Für Daten sind es sogar mindestens 200ns.
Bei 16MHz Taktfrequenz des ATmega128 ergibt sich pro Takt 62,5ns, so daß es eigentlich mit 3 Takten getan sein müsste, auch wenn die Haltezeit von ca. 20ns noch berücksichtigt wird. Anders sieht es mit der minimalen LOW/HIGH-Zeit des GLCD-Controllers `E` aus, welche mit jeweils mindestens 450ns, bzw. mit 1000ns für die gesamt

E-Zeit vorgegeben ist. Damit ergibt sich, daß zwischen 2 Takten E mindestens 16 Takte des ATMega128 liegen müssen. Das ist in der GLCD_pgX-Schleife nicht der Fall und dort liegt ja auch das Problem.
Warum dieses offensichtlich nur beim Datentransfer in den RAM vorhanden ist – und nicht auch beim RS-, R/W-Timing – ist noch nicht geklärt!
Die Einfügung von 3x nop anstelle des delay150 löst das Problem noch nicht.
Aus Optimierungsgründen ist es erforderlich, die minimale nop-Anzahl zu bestimmen.

Des Weiteren ist es aus Performence-Gründen vielleicht möglich, den AutoIncrement-Zugriff auf das RAM zu machen, ohne jedes Mal den Controller zu deaktivieren. Dann kann allerdings nicht mit den jetzt verfügbaren allgemeinen UPs GLCD_commX gearbeitet werden.

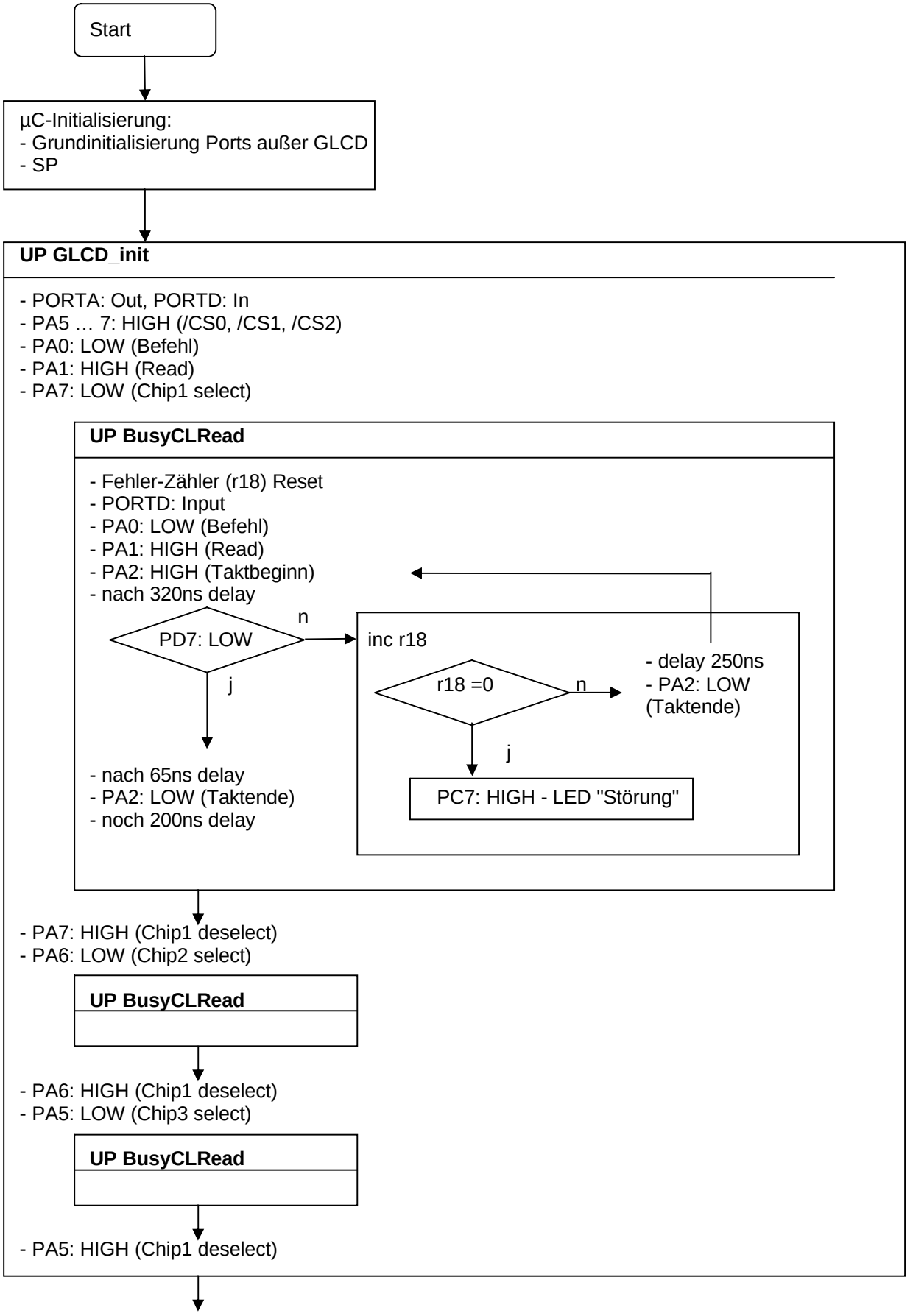
6.3.2. Schlussfolgerungen zur Weiterentwicklung

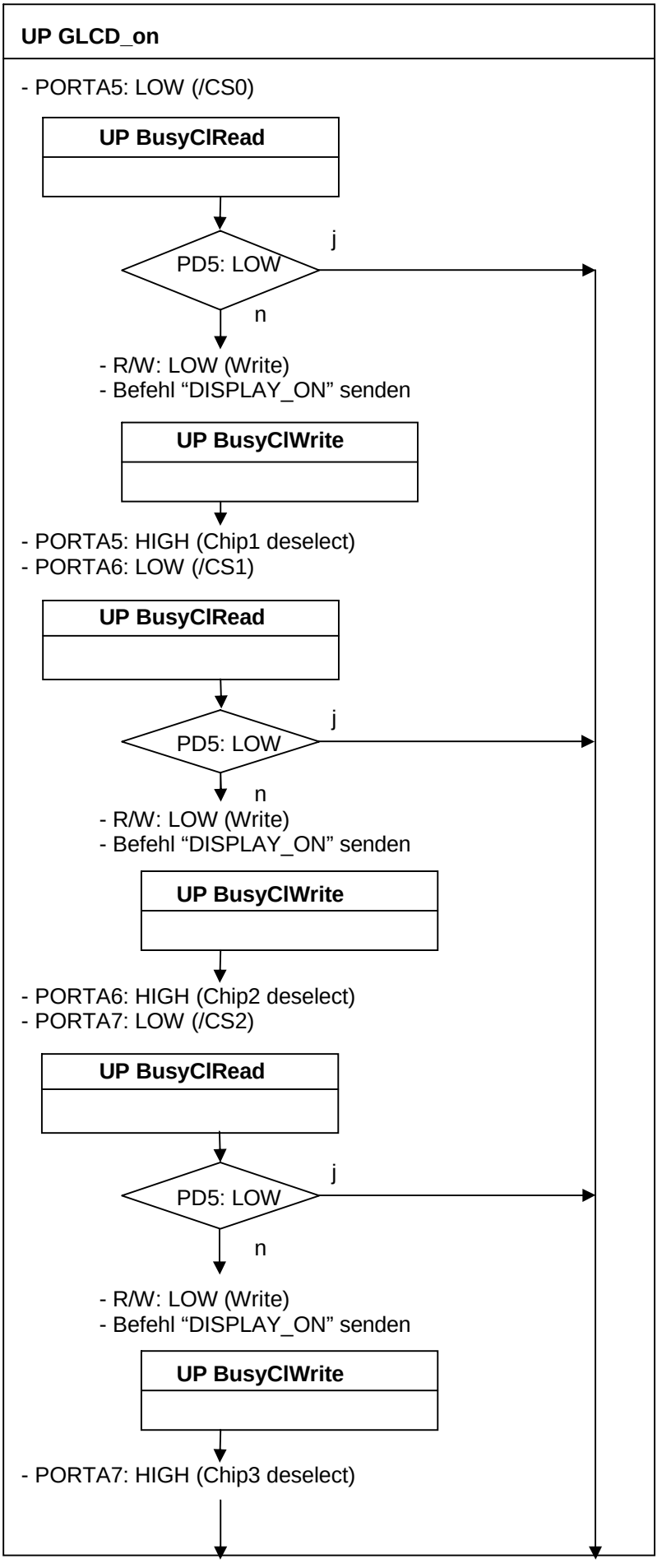
Die bisherigen Untersuchungen zeigte die prinzipielle Möglichkeit der Kommunikation des ATMega128 mit dem GLCD-ControllerSet. Einige Fragen, bzw. Unklarheiten bestehen zwar immer noch, aber aus den gewonnenen Erfahrungen sollten für die Weiterentwicklung einige Richtungs-weisende Schlussfolgerungen gezogen werden:

- Erstellung eines Flussdiagramms zur besseren Übersicht und Verständlichkeit,
- weitere Modularisierung des Gesamtpaketes der GLCD-Routinen,
- Optimierung der einzelnen UPs
- ggf. neues Projekt im ATMEL-Studio
-

(hier weiter)

6.3.2.1. Flussdiagramm




UP GLCD_on
- PORTA5: LOW (/CS0)
UP BusyClRead
PD5: LOW
j
n
- R/W: LOW (Write)
- Befehl “DISPLAY_ON” senden
UP BusyClWrite
- PORTA5: HIGH (Chip1 deselect)
- PORTA6: LOW (/CS1)
UP BusyClRead
PD5: LOW
j
n
- R/W: LOW (Write)
- Befehl “DISPLAY_ON” senden
UP BusyClWrite
- PORTA6: HIGH (Chip2 deselect)
- PORTA7: LOW (/CS2)
UP BusyClRead
PD5: LOW
j
n
- R/W: LOW (Write)
- Befehl “DISPLAY_ON” senden
UP BusyClWrite
- PORTA7: HIGH (Chip3 deselect)

UP GLCD_clr

UP GLCD_adrSt

- PORTD: 0b01000000 (SetAdrY_0)
- PORTA: 0b11100000 (/CS0 - /CS2 HIGH –
  R/W = LOW, RS = LOW (Befehl)
- PORTD: Output

Comm1_on

↓

Comm1_off

↓

Comm2_on

↓

Comm2_off

↓

Comm3_on

↓

Comm3_off

- PORTD: 0b10111000 (SetAdrX_0)

Comm1_on

↓

Comm1_off

↓

Comm2_on

↓

Comm2_off

↓

Comm3_on

↓

Comm3_off

- PORTA: 0b11100001 (/CS0 - /CS2 deaktiviert - R/W = LOW, RS = HIGH (Daten)
- PORTD: 0b00000000 (alle Pixel aus)

UP GLCD_pages

UP GLCD_pages

R20: 8 (virtuellen Pagezähler/Chip laden)

UP GLCD_page

6.4. Struktur des Display-RAMs

Zunächst soll in einem Versuch geklärt werden, wie sich ein Daten-Byte auf dem GLCD abbildet. Dazu wird in einer Test-Sequenz auf die X-Adresse 0, Page-Adresse 0 z.B. das Byte 0b10101010 ausgegeben (geschrieben).

```
ldi r16, 0b10101010
out PORTD, r16          ; Daten liegen am GLCD-Datenbus an
rcall comm1on           ; Kommunikation zum 1. Controller an
cbi PORTA, 1            ; R/W: -> Schreiben
                        ; PORTA0 liegt noch auf LOW (Befehl)
rcall busyCl            ; Takt – X-Adresse + 1
sbi PORTA, 0            ; Daten
rcall busyCl            ; Takt – Daten werden von PORTD in RAM geschrieben
;
rcall comm1off          ; Kommunikation zum 1. Controller aus
;
```

Die o.g. Sequenz wurde so nicht implementiert, weil schon durch das UP `GLCD_clr` die RAM-Struktur hinreichend geklärt wurde. Es ist tatsächlich so, daß auf der X-Achse (GLCD – Y-Adresse!) von links nach rechts gezählt wird – und zwar 64 Byte senkrecht nebeneinander. Es werden also 8 Pixel in 8 Zeilen untereinander mit der gleichen Y-Adresse angesprochen (siehe dazu in der Befehlsliste des GLCD-Controllers (2) „SetAdress (Y)“). Das betrifft den 1.Controller. Der 2.Controller zählt zwar wieder von 0 – 63, betrifft aber jetzt die Pixel 65 – 128 auf dem GLCD. Der 3. Controller verwaltet die Pixel 129 – 192.
Zu überprüfen wäre jetzt noch, ob tatsächlich die Reihenfolge im DatenByte D0 … D7 mit der Reihenfolge der Zeilen 1 … 8 von oben nach unten übereinstimmt. Dazu wird der Einfachheit halber versuchsweise im UP `GLCD-clr` für das zu schreibende Byte ein $E1 eingesetzt, welches sich aus dem aktuellen Status von PORTA ergibt. Damit müssten die Zeilen 1-3 und 8 eingeschaltet und 4-7 ausgeschaltet sein. Praktisch braucht nur im UP `GLCD_clr` der Befehl `clr r16` deaktiviert werden.

Wegen der Probleme mit dem lang andauerndem Speichererhalt (siehe unter Testergebnisse den „bemerkenswerten“ Hinweis) war auch schon ein Schreiben mit $FF erfolgreich.

Ergebnis

Die Zeilen-Reihenfolge in Abhängigkeit der Bit-Stellung im Datenbyte ist genau so wie erwartet, d.h. Bit0 liegt auf Zeile 1 usw. und Bit7 auf Zeile 8. Die Darstellung in (2) der Befehlsliste ist richtig!

Zu prüfende Funktionalität:

Nach dem ein (mehrere) Byte in das/die GLCD-RAM geschrieben wurde, sollen diese mit dem Daten-Lesebefehl wieder zurück gelesen werden.

Siehe dazu 6.7.

6.5. UP `GLCD_SetX`

Mit diesem Unterprogramm soll die aktuelle X-Adresse gesetzt werden. Diese geht bei dem hier verwendeten GLCD von 0 … 191 (3x 64 = 192 Pixel auf einer Zeile). Es ist also je nach X-Wert noch zwischen den 3 LCD-Controllern zu unterscheiden.

Das UP **`GLCD_SetX`** ist im Zusammenhang mit dem UP **`GLCD_SetY`** die Grundvoraussetzung zum Setzen, bzw. Löschen eines Pixels.

Zu beachten ist, daß der dazu betreffende Befehl „**SetAdrY**“ zu verwenden ist!

Dieser wurde bereits schon im UP **`GLCD_adrSt`** eingesetzt.

Die Parameter-Übergabe „AdrX“ (0 … 191) erfolgt in r13.

```
GLCD_SetX:    ldi r16, 0b10111111     ; 191
              sub r16, r13            ; prüfen, ob AdressWertX > 191
              brcc SetXo              ; Fehler!
              ;
              sbrs r13, 6             ; überspringe nächsten Befehl, wenn Bit6 gesetzt (>63)
              rjmp SetX1              ; AdrWertX <=63
              ;
              sbrs r13, 7             ; überspringe nächsten Befehl, wenn Bit7 gesetzt (>127)
              rjmp SetX2              ; 63 < AdrWert > 127
              ;
              mov r16, r13            ; virt. Adressähler unverändert lassen
              andi r16, 0b00011111    ; Ausmaskieren der Bits > 31
              sbr r16, 0b01000000     ; damit Befehl "SetAdrY" komplett
              ;or r16, r13            ; damit Befehl + Adresse für den betreffenden LCD-Chip
                                      ; wird nicht mehr gebraucht
              out PORTD, r16          ; an PORTD senden
              ldi R16, 0b11100000     ; /CS0 - /CS2 deaktiviert -
              out PORTA, r16          ; R/W = LOW, RS = LOW (Befehl)
              ;
              ldi r16, 0b11111111     ; PORTD auf Ausgang
              out DDRD, r16           ; damit liegt der Befehl am GLCD an
              ;
              rcall Comm3_on          ; Übertragen auf Chip3
              rcall Comm3_off
              ;
              rjmp SetXend
              ;
SetX1:        mov r16, r13
              andi r16, 0b00011111    ; Ausmaskieren der Bits > 31
              sbr r16, 0b01000000     ; damit Befehl "SetAdrY"
              ;or r16, r13            ; damit Befehl+Adresse für den betreffenden LCD-Chip
              out PORTD, r16          ; an PORTD senden
              ldi R16, 0b11100000     ; /CS0 - /CS2 deaktiviert -
              out PORTA, r16          ; R/W = LOW, RS = LOW (Befehl)
              ;
              ldi r16, 0b11111111     ; PORTD auf Ausgang
              out DDRD, r16           ; damit liegt der Befehl am GLCD  an
              ;
              rcall Comm1_on          ; Übertragen auf Chip1
              rcall Comm1_off
              ;
              rjmp SetXend
```

```
                ;
SetX2:          mov r16, r13
                andi r16, 0b00011111    ; Ausmaskieren der Bits > 31
                sbr r16, 0b01000000     ; damit Befehl "SetAdrY"
                ;or r16, r13            ; damit Befehl + Adresse für den betreffenden LCD-Chip
                out PORTD, r16          ; an PORTD senden
                ldi R16, 0b11100000     ; /CS0 - /CS2 deaktiviert -
                out PORTA, r16          ; R/W = LOW, RS = LOW (Befehl)
                ;
                ldi r16, 0b11111111     ; PORTD auf Ausgang
                out DDRD, r16           ; damit liegt der Befehl am GLCD an
                ;
                rcall Comm2_on          ; Übertragen auf Chip2
                rcall Comm2_off
                ;
SetXend:        ret
;
SetXo:          sbi PORTC, 7            ; PC7 auf HIGH - LED "Störung" ein
                rjmp SetXo              ; Endlosschleife - raus nur über Reset!

;
```

Zunächst erfolgt eine Prüfung auf Zulässigkeit des Adress-Wertes.

Die Selektierung der Adress-Werte auf die 3 LCD-Controller erfolgt über die Auswahl Bit6, Bit7 des Adress-Wertes und nach Ausmaskieren der unteren 5 Bits durch Zuweisung über die UP `CommX_on` / `CommX_off`.

6.5.1 Test

Der Test mit dem JTAGICEmkII verlief zunächst negativ, weil die rot gekennzeichneten Programm-Schritte nicht korrekt waren.

Korrektur:

```
GLCD_SetX:      ldi r16, 0b10111111     ; 191
                sub r16, r13            ; prüfen, ob AdressWertX > 191
                brcs SetXo              ; Fehler!
                ;
                ldi r16, 0b00111111     ; 63
                sub r16, r13            ; prüfen, ob AdressWertX > 63
                brcc SetX1              ; AdrWertX <=63
                ;
                ldi r16, 0b01111111     ; 127
                sub r16, r13            ; prüfen, ob AdressWertX > 127
                brcc SetX2              ; AdrWertX <=127
                ;
```

Nach Korrektur schien die X-Adressierung einwandfrei zu funktionieren.

6.6. UP `GLCD_SetY`

Mit diesem Unterprogramm soll die aktuelle „X-Adresse“ (Page-Adresse) gesetzt werden. Diese geht bei dem hier verwendeten GLCD von 0 … 7. Eine Page-Adresse beinhaltet immer 8 Bits vertikal untereinander, d.h. die Zeilen 1 … 8 haben die Page-Adresse „0“, die Zeilen 9 … 16 die Page-Adresse „1“ usw. Um ein bestimmtes Pixel innerhalb einer Page-Adresse ansprechen zu können, muß noch eine Auswahl getroffen werden, was z.B. in Form einer Maske geschehen kann, die über das Daten-Byte gelegt wird. Die dazu notwendigen 8 Masken werden dann zweckmäßigerweise als Konstanten in einer Tabelle mit im Programmcode abgelegt. Zur Adressierung der Masken wird dann mit einem Adress-Offset gearbeitet, der invers zum Pixel-Standort im Datenbyte ist, z.B. `Pix1 = D7`.

```
GLCD_SetY:    ldi r16, 0b10000111     ; 191
              sub r16, r12            ; prüfen, ob PageAdressWertY > 7
              brcc SetXo              ; Fehler!
              ;
              ldi r16, 0b10111000     ; Befehl "SetAdrX_0"
              or r16, r12             ; damit Befehl + Adresse
              out PORTD, r16          ; an PORTD an
              ;
              ldi r16, 0b11111111     ; PORTD auf Ausgang
              out DDRD, r16           ; damit liegt der Befehl am GLCD an
              ;
              rcall Comm1_on          ; alle 3 Chips
              rcall Comm1_off
              rcall Comm2_on
              rcall Comm2_off
              rcall Comm3_on
              rcall Comm3_off
              ;
              ret
              ;
SetXo:        sbi PORTC, 7            ; PC7 auf HIGH - LED "Störung" ein
              rjmp SetXo              ; Endlosschleife - raus nur über Reset!
              ;
```

Zunächst erfolgt eine Prüfung auf Zulässigkeit des Adress-Wertes.
Eine Selektierung auf einen der drei GLCD-Chips braucht hier nicht zu erfolgen, da die Y-Adresse für alle gleich ist.

Damit es zu keiner Verwechslung kommt, wurde die Bezeichnung des UP `GLCD_SetY` in `GLCD_SetP` geändert (siehe 6.6.1 Test und Korrektur). Somit wird in diesem Zusammenhang auch nicht mehr von der Y-Adresse, sondern nur noch von der **Page**-Adresse gesprochen. Weitere Auswirkungen hat das zunächst nicht – außer, daß die

eigentliche Y-Adresse für den Zugriff auf ein einzelnes Pixel innerhalb der Page-Adresse noch bestimmt werden muß.

6.6.1 Test

Auch dieser Test mit den JTAGICEmkII verlief zunächst nicht positiv, weil noch einige Programmfehler (rot markiert) vorhanden waren.

Korrektur:

```
GLCD_SetP:    ldi r16, 0b00000111     ; 7
              sub r16, r12            ; prüfen, ob PageAdressWertY > 7
              brcs SetPo              ; Fehler!
              ;
```

Nach Korrektur schien die Page-Adressierung einwandfrei zu funktionieren.

6.7. UP `GLCD_read`

Das UP `GLCD_read` ist wichtig für die Manipulation des GLCD-RAMs. Da dieser rücklesbar ist, macht es keinen Sinn im µC noch einmal extra einen Display-RAM aufzubauen. Der Aufbau des GLCD-RAMs ist unter „6.4. Struktur des Display-RAMs“ erläutert. Danach liegen die Pixel eines Daten-Bytes senkrecht untereinander. Zur Manipulation eines beliebigen Pixels ist zunächst dessen X-Adresse zu bestimmen, danach die Page-Adresse, worin sich das Pixel befindet. Mit diesen Adress-Werten wird das betreffende Daten-Byte ausgelesen.

```
GLCD_read:                      ; Gültigkeit r13 ungeprüft, da im UP GLCD_SetX
            sbrs r13, 6         ; überspringe nächsten Befehl, wenn Bit6 gesetzt (>63)
            rjmp rd1            ; AdrWertX <=63
            ;
            sbrs r13, 7         ; überspringe nächsten Befehl, wenn Bit7 gesetzt (>127)
            rjmp rd2            ; 63 < AdrWert > 127
            ;
            ldi R16, 0b11100011       ; /CS0 - /CS2 deaktiviert -
            out PORTA, r16      ; R/W = HIGH (Lesen), RS = HIGH (Daten)
            ;
            cls r16             ; PORTD auf Eingang
            out DDRD, r16       ; damit ist PORTD für den Datenempfang bereit
            ;
            ;rcall Comm3_on     ; Übertragen auf Chip3 - wird hier nicht verwendet
            cbi PORTA, 7        ; setze PORTA7 (/CS2) auf LOW
            rcall BusyClread1 ; Takt
            ;
            ;rcall Comm3_off
            sbi PORTA, 7        ; setze PORTA7 (/CS2) wieder auf HIGH
            ;
            rjmp rdEnd
            ;
rd1:        cbi PORTA, 5        ; setze PORTA5 (/CS0) auf LOW
            rcall BusyClread1 ; Takt
            ;
            sbi PORTA, 5        ; setze PORTA5 (/CS0) wieder auf HIGH
            ;
            rjmp rdEnd
            ;
rd2:        cbi PORTA, 6        ; setze PORTA6 (/CS1) auf LOW
            rcall BusyClread1 ; Takt
            ;
            sbi PORTA, 6        ; setze PORTA6 (/CS1) wieder auf HIGH
            ;
rdEnd:      ret
            ;
```

In `r17` steht nunmehr das rückgelesene Daten-Byte des Display-RAMs zur weiteren Bearbeitung zur Verfügung. Vor der Zurück-Schreibung auf das Display-RAM muß die X-Adresse neu eingeschrieben werden, da nach jedem Zugriff (Takt) auf das Display-RAM dessen X-Adresse intern automatisch inkrementiert wird.

Der Test sollte mit einem Pixel-Muster erfolgen, wie unter UP `GLCD_clr` erläutert.

6.7.1 Test

Auch dieser Test mit den JTAGICEmkII verlief zunächst nicht positiv, weil noch einige Programmfehler (rot markiert) vorhanden waren. Die Programmfehler waren durch Kopieren die gleichen, wie unter 6.5., so daß die Korrektur analog erfolgen muß.

Korrektur:

```
GLCD_read:                          ; Gültigkeit r13 ungeprüft, da im UP GLCD_SetX
            ldi r16, 0b00111111     ; 63
            sub r16, r13            ; prüfen, ob AdressWertX > 63
            brcc rd1                ; AdrWertX <=63
            ;
            ldi r16, 0b01111111     ; 127
            sub r16, r13            ; prüfen, ob AdressWertX > 127
            brcc rd2                ; 63 < AdrWert > 127
            ;
```

Das UP `BusyClread1` kann hier nicht verwendet werden, weil dieses ausschließlich für den Befehl „Status lesen“ gedacht ist (Kriterium Test `BusyFlag`).

Das kann jedoch modifiziert werden in:

```
DatClread:  sbi PORTA, 2        ; E -> Taktimpuls HIGH
                                ; Impulsanfang, Impulsbreite >500ns
            nop                 ; pro Takt 62,5ns bei fT= 16 MHz
            nop
            nop
            nop
            nop                 ; nach 320ns liegen die gelesenen Daten
                                ; auf dem Datenbus vor
            in r17, PIND        ; akt. PIND-Daten sichern, auch noch nach E-LOW
            nop
            cbi PORTA, 2        ; E -> LOW, Impulsende
            nop                 ; für mind. 500ns LOW
            nop
            nop
            ;
            ret
            ;
```

Nach diesen Korrekturen funktionierte das Programm soweit erst einmal, d.h. in `r17` konnte mit einer zufälligen X-/P-Adressierung der zurück gelesene Inhalt des betreffenden Display-RAMs nachgewiesen werden. Die zufällige Adressierung ergab sich bei X = 80H und P = 0 beim Step-Betrieb des JTAGICEmkII durch die Inhalte von `r13` und `r12` nach dem Start. Da der Inhalt „E1“ (Hex) bedingt durch die Ausschaltung der Pixel-Löschung (siehe 6.3. `GLCD_clr`) jedoch auf alle Adressen zutrifft, ist das noch kein endgültiger Beweis einer korrekten Adressierung.

6.8. UP `GLCD_write`

Auch das UP `GLCD_write` ist wichtig für Pixel-Manipulationen der/des GLCD-RAMs. Wie auch schon beim UP **`GLCD_read`** sind vorher die betreffenden Adressen – X- und Page-Adresse – an die GLCD-Chips zu senden. Bei aufeinander folgenden X-Adressen braucht man das durch die automatische Inkrement-Funktion der X-Adresse zwar nur im Ausnahmefall des Übergangs von einem Chip auf den nächsten tun, aber trotzdem ist dadurch die Implementation externer Adresszähler notwendig. Für die Page-Adresse gibt es keine interne Inkrement-Funktion, so daß hier sowie so ein Adresszähler in Form eines Page-Zählers notwendig ist. Die eigentliche Y-Adresse ist dann innerhalb der betreffenden Page-Adresse verfügbar.
Diese Betrachtungen treffen genau so auch auf das UP **`GLCD_read`** zu.

Durch die Page-Adressierung kann bei der Verwendung des UP `GLCD_write` auch auf mehrere, maximal 8 Pixel untereinander gleichzeitig zugegriffen werden, ohne jedes Mal neu adressieren zu müssen. Das ist insbesondere bei der Erzeugung senkrechter Linien von Vorteil. Entsprechende Algorithmen müssen herausfinden, wo sich das betreffende Linienstück befindet – an Anfang, in der Mitte oder an Ende der Linie. Je nach dem werden dann über Masken-Operationen die betreffenden Bits = Pixel im Datenbyte = Page gesetzt oder auch nicht.

Das UP `GLCD_write` erwartet das an die vorher definierte X- und Page-Adresse zu schreibende Byte im Register `r16`.

```
GLCD_write:   push r16                ; Datenbyte sichern
              ;
              ldi r16, 0b00111111     ; 63
              sub r16, r13            ; prüfen, ob AdressWertX > 63
              brcc wr1                ; AdrWertX <=63
              ;
              ldi r16, 0b01111111     ; 127
              sub r16, r13            ; prüfen, ob AdressWertX > 127
              brcc wr2                ; 63 < AdrWert > 127
              ;
              ldi R16, 0b11100001     ; /CS0 - /CS2 deaktiviert -
              out PORTA, r16          ; R/W = LOW, RS = HIGH (Daten)
              ;
              pop r16                 ; DatenByte zurück
              out PORTD, r16
              ldi r16, 0b11111111     ; PORTD auf Ausgang
              out DDRD, r16           ; damit liegt an PORTD das DatenByte an
              ;
              cbi PORTA, 7            ; setze PORTA7 (/CS2) auf LOW
              ;
              rcall Comm3on           ; Übertragen auf Chip3
```

```
                rcall Comm3off
                ;
                rjmp Write_end
                ;
wr1:            ldi R16, 0b11100001     ; /CS0 - /CS2 deaktiviert -
                out PORTA, r16          ; R/W = LOW, RS = HIGH (Daten)
                ;
                pop r16                 ; DatenByte zurück
                out PORTD, r16
                ldi r16, 0b11111111     ; PORTD auf Ausgang
                out DDRD, r16           ; damit liegt an PORTD das DatenByte an
                ;
                cbi PORTA, 5            ; setze PORTA5 (/CS0) auf LOW
                ;
                rcall Comm1on           ; Übertragen auf Chip1
                rcall Comm1off
                ;
                rjmp Write_end
                ;
wr2:            ldi R16, 0b11100001     ; /CS0 - /CS2 deaktiviert -
                out PORTA, r16          ; R/W = LOW, RS = HIGH (Daten)
                ;
                pop r16                 ; DatenByte zurück
                out PORTD, r16
                ldi r16, 0b11111111     ; PORTD auf Ausgang
                out DDRD, r16           ; damit liegt an PORTD das DatenByte an
                ;
                cbi PORTA, 6            ; setze PORTA6 (/CS1) auf LOW
                ;
                rcall Comm2on           ; Übertragen auf Chip2
                rcall Comm2off
                ;
Write_end:      ret
                ;
```

Da hier die Parameter-Übergabe in `r16` erfolgen soll, muß dieser Wert zunächst auf dem Stack gesichert werden, da die Chip-Auswahl ebenfalls dieses UPR benutzt.
Zur Übertragung auf die drei GLCD-Chips wird der umfangreiche UP-Satz `CommXon` / `CommXoff` benutzt, welcher in jedem Fall die Bereitschaft der Chips neue Daten zu empfangen berücksichtigt. Ggf. könnte das noch vereinfacht werden.

6.8.1 Test

Zur besseren Übersicht wurde der Test mit gelöschtem Display (`clr r16` ; alle Pixel aus – in `GLCD_clr`) und separat vorgegebener Adresse vorgenommen:

```
ldi r16, 0b00101010       ; 42
mov r13, r16              ; damit X-Adresse „42“ gesichert (1.Chip)
ldi r16, 0b00000011       ; 3
mov r12, r16              ; mit Page-Adresse „3“ - etwa in Display-Mitte
ldi r16, 0b10101010       ; DatenByte (kleine punktierte senkrechte Linie)
rcall GLCD_write
```

Ein erster Test ergab, daß zwar eine senkrechte punktierte Linie gezeichnet wird, aber nicht an der programmierten Stelle:
X = 42, Page = 3
Das ist auch klar, weil zwar die Adress-Parameter übergeben, aber nicht das UP `GLCD_SetX` und `GLCD_SetP` aufgerufen wurden.

Korrektur bzw. Ergänzung

```
ldi r16, 0b00101010       ; 42
mov r13, r16              ; damit X-Adresse „42“ gesichert (2.Chip)
ldi r16, 0b00000011       ; 3
mov r12, r16              ; mit Page-Adresse „3“ - etwa in Display-Mitte
rcall GLCD_SetX           ; X-Adresse einstellen
rcall GLCD_SetP           ; Page-Adresse einstellen
ldi r16, 0b10101010       ; DatenByte (kleine punktierte senkrechte Linie)
rcall GLCD_write          ; Datenbyte an GLCD senden
```

Der Test ergab zwar eine andere Stelle, aber noch nicht die richtige Adresse. Der Fehler lag im UP `GLCD_SetX`, hier in der Code-Zeile

```
andi r16, 0b00011111     ; Ausmaskieren der Bits > 31
```

was geändert werden muß in

```
andi r16, 0b00111111     ; Ausmaskieren der Bits > 63
```

weil die GLCD-Chips natürlich nicht nur 32, sondern 64 X-Adressen verwalten!

Damit verlief der Test nun erfolgreich.

Zur weiteren Ausgestaltung der GLCD-Programme sollten diverse Eingabemöglichkeiten – Taster, Poti o.ä. vorgesehen werden, um nicht jedes Mal zu testende Eingabe-Parameter mit einer neuen Firmware „brennen“ zu müssen. Die Anzahl der Erneuerungen der Firmware ist nach Datenblattangaben auf ca. 10.000 begrenzt.

6.9. UP `GLCD_txt`

6.9.1 Allgemeines zu „Text“

Zur Darstellung von Text auf dem GLCD bedarf es vor allem eines geeigneten Zeichen-Fonts. Eine selbstständige Erstellung ist sehr aufwändig, weshalb man günstigerweise auf bereits fertige Software zurückgreifen sollte.
Die Suche nach brauchbarer Software für den hier vorliegenden Einsatzfall erwies sich allerdings auch nicht gerade einfach. Font-Generatoren ermöglichen in der Regel nur die Neuerstellung kompletter Fonts, was ebenfalls einen erheblichen Aufwand bedeutet.
Und dann ist in Folge der komplizierten File-Struktur der Inhalt so ohne weiteres für die direkte Implementation in die Zeichentabelle des GLCDs auch nicht zu gebrauchen.

Grundsätzlich ist zu unterscheiden zwischen proportionalen Schriftarten und Schriften mit konstanter Breite. Letztere sind zwar im Handling einfacher, ergeben jedoch Zeichen, die mit unterschiedlichen Zwischenräumen nicht nur unschön aussehen, sondern auch Platz verschwendend bei der Anzeige sind.
Bei einer proportionalen Schriftart sind die Zwischenräume immer gleich – hier eine Pixel-Breite.

6.9.2 Zeichentabelle

Bedingt durch die Page-Höhe des GLCDs von einem Byte = 8 Pixel senkrecht sollte der hier verwendete Zeichensatz ebenfalls nur 8 Pixel hoch sein. Damit Unterlängen (z.B. für Buchstaben q und g) möglich sind, ist die Basislinie bei +2 von unten. Damit bleiben für Großbuchstaben und Ziffern noch 6 Pixel übrig. Die Breite ist variabel, was wiederum bedeutet, daß eine diesbezügliche Information mit in der Zeichentabelle, oder in einer weiteren Tabelle abgelegt werden muß.

Format der Zeichentabelle

```
.dB colcount, col1, col2, … coln      ; colcount = Byte-Anzahl, col1 usw. = Byte
                                      ; colln +1 = 0-Byte als Zwischenraum
```

Da eine direkte Konvertierung von vorhandenen Fonts in die hier verwendete Zeichentabelle nicht möglich war, musste das per Hand vorgenommen werden. Zur graphischen Pixel-Anzeige und ggf. Korrektur wurde das *Gtool* „Font“ verwendet. Damit lassen sich vorhanden (Windows-) Schriftarten importieren und ggf. bearbeiten. Die Export-Funktion ist hier allerdings auch nicht brauchbar, so daß die einzelnen Pixel-

Bytes per Hand übertragen werden müssen. Zur Übertragung muß von rechts geschaut werden, d.h. `Bit0` ist rechts (ganz oben).
Zur Disposition stand den Zwischenraum (0-Byte) jedes Mal programmtechnisch zu implementieren, oder diesen gleich in der Zeichentabelle mit unterzubringen. Bedingt durch die Autoinkrement-Funktion der X-Adressen der Controller-Chips des GLCDs wurde zu letzterem Verfahren entschieden. Im anderen Fall würde das zwar erheblich mehr programmtechnischen Aufwand bedeuten, wäre aber auch nicht so flexibel. Als Kompromiss muß man halt bei größeren Zwischenräumen, als es das 0-Byte bereits realisiert, weitere 0-Bytes dazwischenschieben, oder ein Leerzeichen setzen.

Zeichentabelle

```
Chr20:  .dB $08, $00, $00, $00, $00, $00, $00, $00, $00; Leerzeichen (8Col)
Chr21:  .dB $02, $2F, $00                               ;!
chr22:  .dB $03, $03, $03, $00                          ;“
chr23:  .dB $05, $3A, $FF, $3A, $FF, $00                ;#
chr24:  .dB $06, $12, $25, $7F, $29, $12, $00           ;$
chr25:  .dB $07, $02, $05, $32, $1E, $29, $10, $00      ;%
chr26:  .dB $06, $10, $2B, $2D, $13, $28, $00           ;&
chr27:  .dB $02, $03, $00                               ;’
chr28:  .dB $04, $3C, $42, $81, $00                     ;(
chr29:  .dB $05, $81, $42, $3C, $00                     ;)
chr2A:  .dB $04, $05, $02, $05, $00                     ;*
chr2B:  .dB $06, $08, $08, $3E, $08, $08, $00           ;+
chr2C:  .dB $02, $60, $00                               ;,
chr2D:  .dB $04, $08, $08, $08, $00                     ;-
chr2E:  .dB $02, $20, $00                               ;.
chr2F:  .dB $04, $30, $0C, $03, $00                     ;/
chr30:  .dB $05, $1E, $21, $21, $1E, $00                ;0
chr31:  .dB $03, $02, $3F, $00                          ;1
chr32:  .dB $05, $22, $31, $29, $26, $00                ;2
chr33:  .dB $05, $12, $21, $25, $1E, $00                ;3
chr34:  .dB $05, §1C, $12, $3F, $10, $00                ;4
chr35:  .dB $05, $17, $25, $25, $19, $00                ;5
chr36:  .dB $05, $1E, $25, $25, $18, $00                ;6
chr37:  .dB $05, $01, $31, $0D, $03, $00                ;7
chr38:  .dB $05, $1A, $25, $25, $1A, $00                ;8
chr39:  .dB $05, $06, $29, $29, $1E, $00                ;9
chr3A:  .dB $03, $12, $00                               ;:
chr3B:  .dB $04, $40, $24, $00                          ;;
chr3C:  .dB $04, $08, $14, $22, $00                     ;<
chr3D:  .dB $04, $14, $14, $14, $00                     ;=
chr3E:  .dB $04, $22, $14, $08, $00                     ;>
chr3F:  .dB $04, $01, $2D, $02, $00                     ;?
chr40:  .dB $08, $3C, $7E, $A3, $BD, $A7, $F3, $5E, $00;@
chr41:  .dB $06, $30, $0E, $09, $0E, $30, $00           ;A
chr42:  .dB $05, $3F, $25, $25, $1E, $00                ;B
chr43:  .dB $05, $1E, $21, $21, $12, $00                ;C
chr44:  .dB $05, $3F, $21, $21, $1E, $00                ;D
chr45:  .dB $05, $3F, $25, $25, $25, $00                ;E
chr46:  .dB $05, $3F, $05, $05, $01, $00                ;F
```

```
chr47:  .dB $05, $1E, $21, $29, $1A, $00              ;G
chr48:  .dB $05, $2F, $04, $04, $2F, $00              ;H
chr49:  .dB $02, $2F, $00                             ;I
chr4A:  .dB $04, $30, $20, $3F, $00                   ;J
chr4B:  .dB $05, $3F, $0C, $12, $21, $00              ;K
chr4C:  .dB $05, $3F, $20, $20, $20, $00              ;L
chr4D:  .dB $06, $3F, $06, $08, $06, $3F, $00         ;M
chr4E:  .dB $05, $3F, $06, $18, $3F, $00              ;N
chr4F:  .dB $05, $1E, $21, $21, $1E, $00              ;O
chr50:  .dB $05, $3F, $09, $09, $06, $00              ;P
chr51:  .dB $05, $1E, $21, $31, $2E, $00              ;Q
chr52:  .dB $05, $3F, $0D, $15, $27, $00              ;R
chr53:  .dB $05, $16, $25, $29, $1A, $00              ;S
chr54:  .dB $06, $01, $01, $3F, $01, $01, $00         ;T
chr55:  .dB $05, $1F, $20, $20, $1F, $00              ;U
chr56:  .dB $06, $03, $1C, $20, $1C, $03, $00         ;V
chr57:  .dB $06, $3F, $18, $04, $18, $3F, $00         ;W
chr58:  .dB $06, $21, $12, $0C, $12, $21, $00         ;X
chr59:  .dB $06, $01, $06, $38, $06, $01, $00         ;Y
chr5A:  .dB $05, $31, $29, $28, $03, $00              ;Z
chr5B:  .dB $03, $FF, $18, $00                        ;[
chr5C:  .dB $04, $03, $0C, $30, $00                   ;\
chr5D:  .dB $03, $18, $FF, $00                        ;]
chr5E:  .dB $04, $02, $01, $02, $00                   ;^
chr5F:  .dB $05, $80, $80, $80, $80, $00              ;_
chr60:  .dB $03, $01, $02, $00                        ;`
chr61:  .dB $05, $10, $2A, $2A, $34, $00              ;a
chr62:  .dB $05, $3F, $24, $24, $18, $00              ;b
chr63:  .dB $05, $1C, $22, $22, $14, $00              ;c
chr64:  .dB $05, $18, $24, $24, $3F, $00              ;d
chr65:  .dB $05, $1C, $2A, $2A, $02, $00              ;e
chr66:  .dB $04, $04, $3E, $05, $00                   ;f
chr67:  .dB $05, $98, $A4, $A4, $78, $00              ;g
chr68:  .dB $05, $3F, $04, $04, $38, $00              ;h
chr69:  .dB $02, $3D, $00                             ;i
chr6A:  .dB $03, $40, $3D, $00                        ;j
chr6B:  .dB $05, $3F, $08, $14, $22, $00              ;k
chr6C:  .dB $03, $1F, $20, $00                        ;l
chr6D:  .dB $06, $3C, $04, $3C, $04, $38, $00         ;m
chr6E:  .dB $05, $3C, $04, $04, $38, $00              ;n
chr6F:  .dB $05, $18, $24, $24, $18, $00              ;o
chr70:  .dB $05, $FC, $24, $24, $18, $00              ;p
chr71:  .dB $05, $18, $24, $24, $FC, $00              ;q
chr72:  .dB $04, $3C, $08, $04, $00                   ;r
chr73:  .dB $05, $24, $2A, $2A, $12, $00              ;s
chr74:  .dB $04, $02, $1F, $22, $00                   ;t
chr75:  .dB $05, $1C, $20, $20, $3C, $00              ;u
chr76:  .dB $04, $0C, $30, $0C, $00                   ;v
chr77:  .dB $06, $1C, $20, $1C, $20, $1C, $00         ;w
chr78:  .dB $05, $24, $18, $18, $24, $00              ;x
chr79:  .dB $04, $8C, $70, $0C, $00                   ;y
chr7A:  .dB $05, $24, $34, $2C, $24, $00              ;z
chr7B:  .dB $04, $08, $F7, $81, $00                   ;{
chr7C:  .dB $02, $FF, $00                             ;|
chr7D:  .dB $04, $81, $FF, $08, $00                   ;}
chr7E:  .dB $05, $10, $08, $10, $08                   ;~
chr7F:  .dB $07, $1E, $2D, $33, $33, $21, $1E, $00    ;©
```

```
chr80:  .dB $06, $08, $1E, $29, $21, $02, $00          ;€
chr81:  .dB $06, $08, $1C, $2A, $08, $08, $00          ;←
chr82:  .dB $06, $04, $02, $3F, $02, $04, $00          ;↑
chr83:  .dB $06, $08, $08, $2A, $1C, $08, $00          ;→
chr84:  .dB $06, $08, $10, $3F, $10, $08, $00          ;↓
chr85:  .dB $06, $24, $42, $FF, $42, $24, $00          ;↕
chr86:  .dB $09, $08, $1C, $2A, $08, $08, $2A, $1C, $08, $00 ;↔
```

Auf die deutschen Umlaute Ä, ä, Ö, ö, Ü, ü, und „ß“ wurde verzichtet, da diese sehr verstreut im ASCII-Zeichensatz weit nach `$86` vorkommen. Die englische Schreibweise dafür Ä = Ae usw., bzw. ß = ss muß somit dafür akzeptiert werden.
Da die Tabelle keine gleichmäßige Struktur hat, wurde den einzelnen Zeichensätzen pro Zeichen jeweils eine Marke für eine einfache Adressierung zugewiesen. Als Markenname sollte eigentlich das ASCII-Zeichen selbst und das Präfix `chr` verwendet werden, was aber Konflikte mit dem Assembler verursachte. Dies war z.T. schon beim Übertragen des Quelltextes ersichtlich, als Kommentartexte nicht mehr als solche kenntlich (grün) gemacht wurden. Beim Compilieren wurden dann insbesondere die mit Satz- und Sonderzeichen verwendeten Markennamen nicht akzeptiert.
Ein Ausweg bietet sich in der Verwendung der laufenden ASCII-Nummern für die Markennamen an, was aber wiederum die Identifizierung sichtlich erschwert.
Andererseits wäre es auch möglich, die Markennamen im Klartext anzugeben.

Da die diesbezüglichen Verarbeitungsalgorithmen noch nicht feststehen, ist auch noch nicht abwägbar, welche Namensgebung hier am günstigsten ist.
Vielleicht reicht die einfache ASCII-Nummer aus –> nein, Labelbezeichnungen müssen mit einem Buchstaben beginnen!
Ein weiteres Problem ist die gerade oder ungerade Anzahl bei `.dB`-Anweisungen. Eine ungerade Anzahl kommentiert der Assembler mit einer Warnmeldung, von wegen der Leerstelle im Programmspeicher. Die Eintragung der Bytes im Programmspeicher erfolgt bei der `.dB` -Anweisung immer am Beginn einer neuen Wort-Adresse.
Zu prüfen wäre noch die Reihenfolge der Byte-Ablage entsprechend der `.dB` -Anweisung. Möglicherweise muß an Stelle dessen die `.dw`-Anweisung verwendet werden. Dies ist zwar weniger übersichtlich, als die `.dB`-Anweisung, wäre aber im Fall der Umkehrung HIGH-/LOW-Byte noch zu akzeptieren. Ggf. müsste die `.dB`-Anweisung auskommentiert und darunter mit der `.dw`-Anweisung gearbeitet werden.

Um die reale Adressierungsweise ergründen zu können wurde folgende Code-Sequenz versuchsweise als `Test1` implementiert und mit dem normalen AVR-Simulator schrittweise untersucht:

```
Test1:  ldi ZH, high(chr24<<1)
        ldi ZL, low(chr24<<1)
t1:     lpm r16, Z+
        rjmp t1
```

Die Pixelbytes wurden in der richtigen Reihenfolge ausgelesen. Es scheint also tatsächlich mit der `.dB`-Anweisung der richtige Aufbau der Zeichentabelle möglich zu sein.

6.9.3 Übertragung der Tabellenwerte

Zur Übertragung der Tabellenwerte in die RAMs der GLCD-Controller zur Darstellung des Zeichens bedarf es einiger Vorbereitungen. Zunächst ist zu klären, mit welchen Parametern die Übertragung zu erfolgen hat:

- Start-Adresse des Zeichens
- das Zeichen selbst – als ASCII-Wert (laufende Nummer, beginnend bei `$20` bis `$86`)

Die Start-Adresse setzt sich aus dem X- und dem Y-Wert (Page) zusammen und wird entsprechend der gewünschten örtlichen Zuordnung festgelegt, d.h. durch die betreffenden UPs an das GLCD übertragen.

Etwas schwieriger ist die Auswahl des betreffenden Zeichensatzes. Wenn davon ausgegangen werden soll, daß zwar das darzustellende Zeichen als ASCII-Wert vorliegt, ist damit noch lange nicht der betreffende Zeichensatz adressiert.

Da die Zeichentabelle ungleichmäßig strukturiert ist, kann aus dem ASCII-Wert die betreffende Adresse in der Zeichentabelle nicht berechnet werden. Zur Lösung des Problems gibt es zwei Wege:

- in einer weiteren Tabelle `AdrTbl` die Adressen der einzelnen Zeichen von `chr20` bis `chr86` ablegen, oder
- die vorhandene `ChrTbl` solange mit 0-Bytes auffüllen, daß eine gleichmäßige Struktur herauskommt.

In beiden Fällen wird mehr Programm-Speicherplatz benötigt, wobei das erste Verfahren als flexibler und demzufolge günstiger erscheint.

Über die `AdrTbl` mit folgendem Aufbau kann dann mittels eines Offsets der laufenden ASCII-Nummer (Ausblendung der ersten 32 Zeichen = `$20`) auf die wirkliche Adresse des gesuchten Zeichensatzes zugegriffen werden.

```
AdrTbl:       .dB LOW(chr20<<1), HIGH(chr20<<1)
              .dB LOW(chr21<<1), HIGH(chr21<<1)
              .dB LOW(chr22<<1), HIGH(chr22<<1)
              .dB LOW(chr23<<1), HIGH(chr23<<1)
              .dB LOW(chr24<<1), HIGH(chr24<<1)
              .dB LOW(chr25<<1), HIGH(chr25<<1)
              .dB LOW(chr26<<1), HIGH(chr26<<1)
              .dB LOW(chr27<<1), HIGH(chr27<<1)
              .dB LOW(chr28<<1), HIGH(chr28<<1)
              .dB LOW(chr29<<1), HIGH(chr29<<1)
              .dB LOW(chr2A<<1), HIGH(chr2A<<1)
              .dB LOW(chr2B<<1), HIGH(chr2B<<1)
              .dB LOW(chr2C<<1), HIGH(chr2C<<1)
              .dB LOW(chr2D<<1), HIGH(chr2D<<1)
              .dB LOW(chr2E<<1), HIGH(chr2E<<1)
              .dB LOW(chr2F<<1), HIGH(chr2F<<1)
              .dB LOW(chr30<<1), HIGH(chr30<<1)
              .dB LOW(chr31<<1), HIGH(chr31<<1)
              .dB LOW(chr32<<1), HIGH(chr32<<1)
              .dB LOW(chr33<<1), HIGH(chr33<<1)
              .dB LOW(chr34<<1), HIGH(chr34<<1)
              .dB LOW(chr35<<1), HIGH(chr35<<1)
              .dB LOW(chr36<<1), HIGH(chr36<<1)
              .dB LOW(chr37<<1), HIGH(chr37<<1)
              .dB LOW(chr38<<1), HIGH(chr38<<1)
              .dB LOW(chr39<<1), HIGH(chr39<<1)
              .dB LOW(chr3A<<1), HIGH(chr3A<<1)
              .dB LOW(chr3B<<1), HIGH(chr3B<<1)
              .dB LOW(chr3C<<1), HIGH(chr3C<<1)
              .dB LOW(chr3D<<1), HIGH(chr3D<<1)
              .dB LOW(chr3E<<1), HIGH(chr3E<<1)
              .dB LOW(chr3F<<1), HIGH(chr3F<<1)
              .dB LOW(chr40<<1), HIGH(chr40<<1)
              .dB LOW(chr41<<1), HIGH(chr41<<1)
              .dB LOW(chr42<<1), HIGH(chr42<<1)
              .dB LOW(chr43<<1), HIGH(chr43<<1)
              .dB LOW(chr44<<1), HIGH(chr44<<1)
              .dB LOW(chr45<<1), HIGH(chr45<<1)
              .dB LOW(chr46<<1), HIGH(chr46<<1)
              .dB LOW(chr47<<1), HIGH(chr47<<1)
              .dB LOW(chr48<<1), HIGH(chr48<<1)
              .dB LOW(chr49<<1), HIGH(chr49<<1)
              .dB LOW(chr4A<<1), HIGH(chr4A<<1)
              .dB LOW(chr4B<<1), HIGH(chr4B<<1)
              .dB LOW(chr4C<<1), HIGH(chr4C<<1)
              .dB LOW(chr4D<<1), HIGH(chr4D<<1)
              .dB LOW(chr4E<<1), HIGH(chr4E<<1)
              .dB LOW(chr4F<<1), HIGH(chr4F<<1)
              .dB LOW(chr50<<1), HIGH(chr50<<1)
              .dB LOW(chr51<<1), HIGH(chr51<<1)
              .dB LOW(chr52<<1), HIGH(chr52<<1)
              .dB LOW(chr53<<1), HIGH(chr53<<1)
              .dB LOW(chr54<<1), HIGH(chr54<<1)
              .dB LOW(chr55<<1), HIGH(chr55<<1)
              .dB LOW(chr56<<1), HIGH(chr56<<1)
              .dB LOW(chr57<<1), HIGH(chr57<<1)
              .dB LOW(chr58<<1), HIGH(chr58<<1)
```

```
                .dB LOW(chr59<<1), HIGH(chr59<<1)
                .dB LOW(chr5A<<1), HIGH(chr5A<<1)
                .dB LOW(chr5B<<1), HIGH(chr5B<<1)
                .dB LOW(chr5C<<1), HIGH(chr5C<<1)
                .dB LOW(chr5D<<1), HIGH(chr5D<<1)
                .dB LOW(chr5E<<1), HIGH(chr5E<<1)
                .dB LOW(chr5F<<1), HIGH(chr5F<<1)
                .dB LOW(chr60<<1), HIGH(chr60<<1)
                .dB LOW(chr61<<1), HIGH(chr61<<1)
                .dB LOW(chr62<<1), HIGH(chr62<<1)
                .dB LOW(chr63<<1), HIGH(chr63<<1)
                .dB LOW(chr64<<1), HIGH(chr64<<1)
                .dB LOW(chr65<<1), HIGH(chr65<<1)
                .dB LOW(chr66<<1), HIGH(chr66<<1)
                .dB LOW(chr67<<1), HIGH(chr67<<1)
                .dB LOW(chr68<<1), HIGH(chr68<<1)
                .dB LOW(chr69<<1), HIGH(chr69<<1)
                .dB LOW(chr6A<<1), HIGH(chr6A<<1)
                .dB LOW(chr6B<<1), HIGH(chr6B<<1)
                .dB LOW(chr6C<<1), HIGH(chr6C<<1)
                .dB LOW(chr6D<<1), HIGH(chr6D<<1)
                .dB LOW(chr6E<<1), HIGH(chr6E<<1)
                .dB LOW(chr6F<<1), HIGH(chr6F<<1)
                .dB LOW(chr70<<1), HIGH(chr70<<1)
                .dB LOW(chr71<<1), HIGH(chr71<<1)
                .dB LOW(chr72<<1), HIGH(chr72<<1)
                .dB LOW(chr73<<1), HIGH(chr73<<1)
                .dB LOW(chr74<<1), HIGH(chr74<<1)
                .dB LOW(chr75<<1), HIGH(chr75<<1)
                .dB LOW(chr76<<1), HIGH(chr76<<1)
                .dB LOW(chr77<<1), HIGH(chr77<<1)
                .dB LOW(chr78<<1), HIGH(chr78<<1)
                .dB LOW(chr79<<1), HIGH(chr79<<1)
                .dB LOW(chr7A<<1), HIGH(chr7A<<1)
                .dB LOW(chr7B<<1), HIGH(chr7B<<1)
                .dB LOW(chr7C<<1), HIGH(chr7C<<1)
                .dB LOW(chr7D<<1), HIGH(chr7D<<1)
                .dB LOW(chr7E<<1), HIGH(chr7E<<1)
                .dB LOW(chr7F<<1), HIGH(chr7F<<1)
                .dB LOW(chr80<<1), HIGH(chr80<<1)
                .dB LOW(chr81<<1), HIGH(chr81<<1)
                .dB LOW(chr82<<1), HIGH(chr82<<1)
                .dB LOW(chr83<<1), HIGH(chr83<<1)
                .dB LOW(chr84<<1), HIGH(chr84<<1)
                .dB LOW(chr85<<1), HIGH(chr85<<1)
                .dB LOW(chr86<<1), HIGH(chr86<<1)
;
```

6.9.4. Test

Bei der Bearbeitung dieser Adress-Tabelle ist zu beachten, daß der String `chrXX` eigentlich nur einen numerischen Adress-Wert darstellt, welcher entsprechend der Befehls-Charakteristik von `lpm` eine Wort-Adresse ab Bit1 verlangt. Bit0 hat immer „0“ zu sein. Zur besseren Verständlichkeit wurde aus diesem Grunde auch die `<<1` – Schreibweise (= ein Bit nach links verschoben) verwendet.

Die Code-Sequenz für einen Test-Zugriff ergibt sich dann wie folgt:

Übergabe-Parameter: ASCII-Code in `r16`

```
test1:      ldi r16, '!'            ; nur zum Laden mit einem beliebigen Zeichen
                                    ; hier „!“
            subi r16, $20           ; die ersten Zeichen ausblenden
            lsl r16                 ; um 1 Bit nach links schieben, Bit7 -> Cy
                                    ; = Offset der Zeichentabelle
            clr r0                  ; temp. Hilfsregister löschen
            rol r0                  ; ggf. Cy in Bit0 übertragen
            ldi ZL, LOW(AdrTbl<<1)  ; Startadresse von AdrTbl laden
            ldi ZH, HIGH(AdrTbl<<1)
            ;
            add ZL, r16             ; Offset addieren
            adc ZH, r0              ; Korrektur hinzufügen
                                    ; Z = Zeiger auf den betreffenden Zeichensatz
            lpm r24, Z+             ; r24:r25 = Adresse Zeichensatz
            lpm r25, Z
            mov r30, r24            ; als Startadresse für “lpm”
            mov r31, r25
            ;
t1:         lpm r16, Z+             ; 1. Byte aus dem Zeichensatz
                                    ; = Anzahl der zu übertragenden Bytes
                                    ; automatisch Inkrement von Z,
                                    ; d.h. beim nächsten Zugriff -> 2. Byte usw.
            rjmp t1                 ; Endlos-Schleife nur zum Test!
                                    ; (Beobachtung im AVR-Simulator)
```

Für die Weiterverarbeitung nach `t1`: muß zunächst die Anzahl der zu übertragenen Bytes gesichert werden:

```
            mov r0, r16             ; Zeichenbyte-Zähler (temporär)
t2:         lpm r16, Z+             ; nächstes Byte lesen (1., 2., usw. Zeichen-Byte)
            rcall GLCD_write        ; UP Daten-Schreiben aufrufen
                                    ; Parameter- Übergabe in r16
            inc r13                 ; X-Adresszähler inkrementieren
            dec r0                  ; Zeichenbyte-Zähler dekrementieren
            brne t2                 ; bis alle Zeichen-Bytes übertragen sind
            ;
            (ret)
```

Den Zeichenbyte-Zähler dekrementieren muß zu letzt erfolgen, da sonst die Sprung-Bedingung für `brne` vom `inc`-Befehl überschrieben wird.

Bevor `GLCD_write` aufgerufen wird, muß die X- und Y-Adresse, von der aus das Zeichen geschrieben werden soll, mit den Funktionen `GLCD_SetX` und `GLCD_SetY` übergeben werden.

Um die vollständige Darstellung des Zeichens auch am Ende der Zeile bei X <= 192 noch zu gewährleisten, muß vorher überprüft werden, ob dieses noch auf die Zeile geht, oder von links mit einer neuen Zeile (Page) begonnen werden muß.

Die Routine wurde noch etwas sortiert und erweitert – nun als UP:

```
GLCD_chr:     clr r13                 ; X-Adresse auf "0"
              clr r12                 ; Y-Adresse (Page) auf "0"
              rcall GLCD_SetX         ; akt. X-Adresse an GLCD übermitteln –
                                      ; Parameter in r13
              ;
              rcall GLCD_SetP         ; akt. Y-Adresse an GLCD übermitteln –
                                      ; Parameter in r12
              ;
              ;ldi r16, '!'           ; nur zum Laden mit einem beliebigen Zeichen
                                      ; hier "!"
              ldi r16, $20            ; lade ASCII-"Leerzeichen"
              mov r10, r16            ; in temp. Zeichenspeicher
              ;
chr1:         mov r16, r10            ; aktuelles Zeichen laden
              subi r16, $20           ; die ersten Zeichen ausblenden
              lsl r16                 ; um 1 Bit nach links schieben =
                                      ; Offset der Zeichentabelle, Bit7 -> Cy
              clr r0                  ; temp. Hilfsregister löschen
              rol r0                  ; ggf. Cy in Bit0 übertragen = Korrektur
              ldi ZL, LOW(AdrTbl<<1)  ; Startadresse von AdrTbl laden
              ldi ZH, HIGH(AdrTbl<<1)
              ;
              add ZL, r16             ; Offset addieren
              adc ZH, r0              ; Korrektur hinzufügen
                                      ; Z = Zeiger auf den betreffenden Zeichensatz
              lpm r24, Z+             ; r24:r25 = Adresse Zeichensatz
              lpm r25, Z
              mov r30, r24            ; als Startadresse für "lpm"
              mov r31, r25
              ;
chr2:         lpm r16, Z+             ; 1. Byte aus dem Zeichensatz
                                      ; = Anzahl der zu übertragenden Bytes
                                      ; automatisch Inkrement von Z,
                                      ; d.h. beim nächsten Zugriff -> 2. Byte usw.
              ;
              ;rjmp t1                ; Endlos-Schleife nur zum Test!
                                      ; (Beobachtung im AVR-Simulator)
              ;
              mov r0, r16             ; Zeichenbyte-Zähler (temporär)
              add r16, r13            ; prüfen, ob X-Adresse + Zeichenbreite > 192
              clc                     ; sicherheitshalber Cy löschen
              subi r16, 191           ; d.h. nicht mehr auf die Zeile passt
              brcs chr2               ; wenn r16 <= 191
              ;
              clr r13                 ; X-Adresse zurücksetzen auf "0"
              inc r12                 ; Y-Adresse inkrementieren (nächste Zeile – Page)
              rcall GLCD_SetX         ; akt. X-Adresse an GLCD übermitteln –
```

```
                                    ; Parameter in r13
                ;
                rcall GLCD_SetP     ; akt. Y-Adresse an GLCD übermitteln –
                                    ; Parameter in r12
                ;
chr2:           lpm r16, Z+         ; nächstes Byte lesen (1., 2., usw. Zeichen-Byte)
                rcall GLCD_write    ; UP Daten-Schreiben aufrufen
                                    ; Parameter- Übergabe in r16
                inc r13             ; X-Adresszähler inkrementieren
                dec r0              ; Zeichenbyte-Zähler dekrementieren
                brne chr2           ; bis alle Zeichen-Bytes übertragen sind
                ;
                inc r10             ; nächstes Zeichen bereitstellen
                mov r16, r10
                clc                 ; sicherheitshalber Cy löschen
                subi r16, $86       ; prüfen, ob alle Zeichen geschrieben sind
                mov r16, r10        ; Wert wieder herstellen
                brcs chr1           ; wenn nicht, dann nächstes Zeichen
                ;                   ; sonst fertig
                ret
;
```

Das Programm funktioniert im Wesentlichen, bis auf ein Problem beim Übergang c -> d in der 2. Zeile. Von „c“ werden nur die ersten 3 Bytes korrekt übertragen, während das 4. Byte an die X-Adresse 191 übertragen wird. Das danach folgende 0-Byte kann leider nicht identifiziert werden. Vom nachfolgenden „d“ wird das 1- Byte an die X-Adresse 190 übertragen. Die weiteren Bytes werden dann wieder korrekt ausgegeben. Auffällig sind der nicht vorhandene Zwischenraum zwischen „c“ und „d“, sowie die verstümmelte Darstellung der Zeichen.

Vermutlich liegt das Problem beim Übergang von einem Controller-Chip auf das nächste, wenn das innerhalb eines Zyklus der Zeichenbyte-Übertragung passiert. Versuchsweise wurde deshalb die `chr3`-Schleife erweitert:

```
chr3:           lpm r16, Z+         ; nächstes Byte lesen (1., 2., usw. Zeichen-Byte)
                rcall GLCD_write    ; UP Daten-Schreiben aufrufen
                                    ; Parameter- Übergabe in r16
                inc r13             ; X-Adresszähler inkrementieren
                ;
                ldi r16, 0b01000000 ; 64
                clc                 ; sicherheitshalber Cy löschen
                cp r16, r13         ; prüfen, ob AdrWertX = 64 = 1. Byte Chip1
                brne chr5           ; AdrWertX <>64
                ;
                rcall GLCD_SetX     ; interne X-Adresse Chip1 = 0 setzen
                ;
chr4:           ldi r16, 0b10000000 ; 128
                clc                 ; sicherheitshalber Cy löschen
                cp r16, r13         ; prüfen, ob AdrWertX = 128 = 1.Byte Chip2
                brne chr5           ; AdrWertX <> 128
                ;
                rcall GLCD_SetX     ; interne X-Adresse Chip2 = 0 setzen
                ;
```

```
chr5:       dec r0                  ; Zeichenbyte-Zähler dekrementieren
            brne chr3               ; bis alle Zeichen-Bytes übertragen sind
            ;
```

Mit dieser Änderung läuft das UP `GLCD_chr` einwandfrei.

6.10 Linie

6.10.1 Allgemeines zur Linie

Linien werden im Allgemeinen durch ihren beiden Anfangs- und End-Koordinatenpunkte definiert:

- `Xanf; Yanf`
- `Xend; Yend`

Dabei sind zwei Sonderfälle festzustellen:

(1) `Xanf = Xend ->` senkrechte Linie

(2) `Yanf = Yend ->` waagerechte Linie

Der gültige Wertebereich von `X, Y` ergibt sich aus der Pixel-Auflösung des GLCD:

`Xmin ... Xmax` = 1 … 192

`Ymin ... Ymax` = 1 … 64

Für die Adressen-Zähler sind somit bei folgender Zählweise die Grenzwerte einzuhalten:

`Xadr:` 0 … 191

`PageAdr:` 0 … 7

Ein Über- oder Unterschreiten dieser Grenzwerte muß programmtechnisch abgefangen werden.

Da das UP `GLCD_write` immer ein ganzes Byte an eine `X-` und `PageAdr` schreibt, sind für diese Sonderfälle auch gesondere Algorithmen angebracht.

6.10.2 Line - Vorbereitende Maßnahmen

hier weiter:

- Linie allgemein -> waagerecht
  -> senkrecht
  -> im Winkel

6.10.3 UP `GLCD_LineH`

Bedingt durch Arbeitsweise der LCD-Controller (siehe …) ist das Zeichnen einer horizontalen Linie die einfachste Variante.

Es müssen lediglich die X- und Y-Koordinaten des Linien-Anfangs, sowie die X-Koordinate des Endpunktes übergeben werden. Da die Y-Anfangskoordinate gleich der Y-Endkoordinate ist, erübrigt sich deren Angabe.

Die eigentliche Y-Koordinate befindet sich innerhalb einer Page-Adresse und muß deshalb programmtechnisch entsprechend bearbeitet werden.

```
; UP Line horizontal (UP von GLCD_LineAllg)
;   - Übergabeparameter:        r13 -    X-AnfangsAdresse      (>=0)
;                               r16 -    X-EndAdresse          (<=191)
;                               r12 -    Y-Adresse (Page + Zeile: 0 ... 64)
;   - Rückgabeparameter:        r13 -    X-AnfangsAdresse      (>=0)
;                               r16 -    Datenbyte ("Punkt")
;                               r12 -    PageAdresse
;-----------------------------------------------------------------------
;
; Achtung – der Code ist noch fehlerbehaftet!
;
GLCD_LineH:
        rcall GLCD_SetX                 ; X-Parameter in r13 (im Simulator-Test raus)
        ;
        sub r16, r13                    ; = Länge der Linie in Pixel
        mov r13, r16                    ; r13 = Incrementanzahl Zähler
        mov r11, r12                    ; Datenbyte sichern
        lsr r12                         ; Y-Adresse / 8 = PageAdresse in r12
        lsr r12
        lsr r12
        rcall GLCD_SetP                 ; Page-Parameter in r12 (im Simulator-Test raus)
        ;
        clz                             ; ZeroFlag löschen
        ldi r16, 0b00000111             ; AND-Maske laden
        and r11, r16                    ; Y-Adresse maskieren (-> 0 ... 7)
        clr r16                         ; r16 löschen
        sec                             ; setze Cy
lineH1:
        rol r16                         ; Bit von Cy nach links einschieben
        dec r11                         ; Zeilenzähler decrementieren
        breq lineH2                     ; wenn "0" erreicht: r16 = Datenbyte
        ;
        rjmp lineH1
        ;
lineH2:
        rcall GLCD_write                ; Schreiben in den GLCD-RAM, GLCD incr. X-Adresse
        ;
        dec r13                         ; r13 = Incrementanzahl Zähler
        breq lineHend                   ; r13 = 0: -> Ende
        rjmp lineH2
        ;
lineHend:
        ret
```

Diese Variante des einfachen Schreibens in den GLCD-RAM hat den entscheidenden Nachteil, dass davon ausgegangen wird, dass sich keinerlei Informationen darin befinden, d.h. der GLCD-RAM leer ist und somit ebenso der Bildschirm. Das ist aber in der Praxis nicht immer der Fall und beim Einsatz o.g. Routine würden vorhandene Pixel-Informationen an der betreffenden Page-Adresse ausnahmslos gelöscht.
Abhilfe schafft hier eine erweiterte Variante von `GLCD_LineH`, bei der zunächst das vorhandene Datenbyte ausgelesen wird, dann eine OR-Verknüpfung mit dem zu schreibenden Datenbyte stattfindet und zuletzt das Datenbyte wieder an die alte Adresse zurückgeschrieben wird.

Allerdings ergibt sich genau damit ein Problem:
Da die GLCD-Controller immer automatisch nach jedem Zugriff (egal ob schreibend oder lesend) die X-Adresse inkrementieren, würde das zurückzuschreibende Byte an der falschen Adresse landen.
Es muß also immer wieder – bevor das Datenbyte zurückgeschrieben wird – die X-Adresse noch mal übergeben werden.
Die erweiterte `GLCD_LineH` sieht dann so aus:

```
GLCD_LineH:
        rcall GLCD_SetX                 ; X-Parameter in r13 (im Simulator-Test raus)
        ;
        sub r16, r13                    ; Länge der Linie in Pixel
        mov r11, r16                    ; r11 = Incrementanzahl Zähler
        push r12                        ; Y-Adresse sichern
        lsr r12                         ; Y-Adresse / 8 = PageAdresse in r12
        lsr r12
        lsr r12
        rcall GLCD_SetP                 ; PageAdresse in r12 (im Simulator-Test raus)
        ;
        clz                             ; ZeroFlag löschen
        ldi r16, 0b00000111             ; AND-Maske laden
        pop r12                         ; Y-Adresse zurück
        and r12, r16                    ; Y-Adresse maskieren (-> 0 ... 7):
                                                Zeilenzähler innhalb Page
        clr r16                         ; r16 löschen
        sec                             ; setze Cy
lineH1:
        rol r16                         ; Bit von Cy nach links einschieben
        dec r12                         ; Zeilenzähler decrementieren
        breq lineH2                     ; wenn "0" erreicht: r16 = Datenbyte
        ;
        rjmp lineH1
        ;
lineH2:
        rcall GLCD_read                 ; lesen aktuelle RAM-Daten von Adr(X, P)
                                        ; Rückgabe in  r17, GLCD incr. X-Adresse
```

```
        ;
        or r16, r17                     ; mit vorhandenen RAM-Daten verknüpfen
        rcall GLCD_SetX                 ; X-Adresse auf urspünglichen Wert (r13) zurücksetzen
        ;
        rcall GLCD_write                ; Schreiben in den GLCD-RAM, GLCD incr. X-Adresse
        ;
        dec r11                         ; r11 = Incrementanzahl Zähler
        breq lineHend                   ; r11 = 0: -> Ende
        ;
        inc r13                         ; nächste X-Adresse
        rjmp lineH2
        ;
lineHend:
        ret
        ;
```

6.10.3.1 Test `GLCD_LineH`

Zum Test des UP `GLCD_LineH` soll eine einfache Linie auf der Y-Position „1“ = Y-Adresse „0“ von *Xa* = 3 bis *Xe* = 189 erzeugt werden. Dazu bedarf es in `GLCD_main` folgender Sequenz zur Einstellung der Parameter:

```
        ldi r16, 0              ; Zeilenadresse:        0
        mov r12, r16
        ldi r16, 0b00000010     ; X-AnfangsAdresse:     2
        mov r13, r16
        ldi r16, 0b10111101     ; X-EndAdresse:         189
        mov r9, r16
        rcall GLCD_LineH
```

Der erste Test verlief negativ, d.h. es war nichts auf dem Bildschirm zu sehen.
Die anfangs vorgesehene Parameterübergabe zu *Xe* in `r16` mußte in `r9` geändert werden, da `r16` im `GLCD_SetX` verändert wird.
Des Weiteren hat sich herausgestellt – durch Schrittbetrieb über den *JTAGICEmkII* – daß die Sequenz:

```
lineH1:  rol r16                ; Bit von Cy nach links einschieben
         dec r12                ; Zeilenzähler decrementieren
         breq lineH2            ; wenn "0" erreicht: r16 = Datenbyte
         ;
         rjmp lineH1
```

ein Problem mit `dec r12` hat, wenn z.B. `r12` bereits auf auf „0“ steht, weil es die erste Zeile sein soll. Das Z-Flag ist dann nach dem `dec`-Befehl nicht gesetzt und somit kein Sprung-Kriterium für `breq lineH2`.

Wird die Sequenz geändert in:

```
lineH1:  rol r16        ; Bit von Cy nach links einschieben
         dec r12        ; Zeilenzähler decrementieren
         brlt lineH2    ; wenn S-Flag gesetzt (bei 0-Zeile)
```

```
          ;
          brne  lineH1        ; wenn "0" noch nicht erreicht, sonst
                              ; r16 = Datenbyte
```

kann das Problem behoben werden (noch nicht endgültig gestestet)

Ein weiteres Problem trat beim Handling mit den RAM-Daten in `r16` auf. Da `r16` in den UP `GLCD_read, GLCD_SetX` und auch in `GLCD_write` verändert wird, aber stets als Übergabe-Parameter für `GLCD_write` benötigt wird, muss dessen Wert über `push/pop`-Sequenzen gesichert werden.

Die betreffende Code-Sequenz sieht dann so aus:

```
lineH2:   push r16              ; RAM-Daten sichern
          rcall GLCD_read       ; auslesen der aktuellen RAM-Daten von Adr(Xa, P)
                                ; Rückgabe in  r17, X-Adresse wird RAM-intern incr.!
          rcall GLCD_SetX       ; X-Adresse wieder auf Wert (r13) zurücksetzen
          ;
          pop r16               ; RAM-Daten restaurieren
          push r16              ; und gleich wieder sichern
          or r16, r17           ; mit vorhandenen RAM-Daten verknüpfen
          rcall GLCD_write      ; Schreiben in den GLCD-RAM, GLCD incr. X-Adresse
          ;
          pop r16               ; RAM-Daten zurück
          dec r11               ; r11 = Incrementanzahl Zähler
          breq lineHend          r11 = 0: -> Ende
          ;
          inc r13               ; nächste X-Adresse
          rjmp lineH2
          ;
lineHend:
          ret
          ;
;
```

Damit wurde dann tatsächlich eine Linie (2,0; 189,0) gezeichnet.

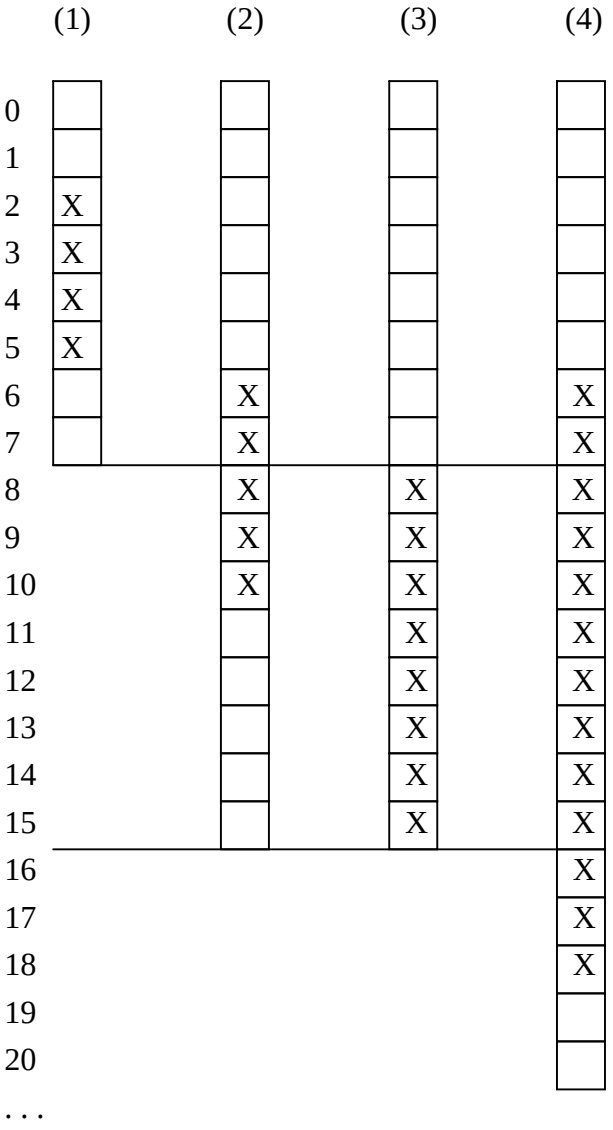
Weitere Varianten einer geraden Linien-Zeichnung sind mit dem Eingabe-Feature über Potis (Analogwert-Eingabe) geplant.

6.10.4 UP `GLCD_LineV`

Während das Zeichnen von horizontalen, d.h. waagerechten Linen noch relativ einfach war – abgesehen von einigen „Randbedingungen“ – gilt diese Aussage für vertikale (senkrechte) Linien so nicht mehr.

Bedingt durch die spezielle Page-Adressierung und die darin eigentlich enthaltenen Y-Adressierung bedarf es einiger Vorüberlegungen.

Für vertikale, d.h senkrechte Linien sind folgende Fälle zu unterscheiden:



<u>Fall (1)</u>

Die Linie befindet sich innerhalb einer PageAdresse, d.h.

Y0 * Pn = Ya = Y7 * Pn (mit Pn = PageAdresse)

Fall (3) ist nur ein Spezialfall von (1).

Fall (2)

Die Linie geht über 2 Page-Adressen, d.h.

Y0 * Pn = Ya = Y7 * Pn+1 (mit Pn = PageAdresse)

Fall (4)

Das ist eigentlich der allgemeine Fall, d.h. die Linie geht über mehrere Page-Adressen.

Y0 * Pn = Ya = Y7 * Pn+m (mit Pn = PageAdresse, m = Anzahl Page-Adressen)

Vor der programmtechnischen Umsetzung ist deshalb zu untersuchen, zu welchem Fall die betreffende Linie gehört, um somit die entsprechende Programmverzweigung vornehmen zu können.
Aus den zu übergebenden Parametern X, Ya und Ye kann der betreffende Fall ermittelt werden.

6.10.4.1 Programmtechnische Realisierung Fall 1

Für den Fall_1 ist die programmtechnische Realisierung für einen ersten Ansatz noch relativ einfach:

```
GLCD_LineV:   mov r9, r8              ; Y-Endadresse sichern
              sub r8, r12             ; Länge der Linie in Pixel (-> r8)
              mov r16, r12            ; Kopie in tempArbeitsregister
              clc                     ; sicherheitshalber Cy löschen
              cpi r16, 8              ; Vergleich mit "8"
              brcc LineV_             ; wenn r16 >= 8 -  Fall_4 oder Fall_2
              ;
; Fall_1
LineV1:       sec                     ; Cy setzen, d.h. Einsen einschieben
              rol   r0                ; Cy-Bit nach links einschieben in r0, 0
              dec r8                  ; LineLängen-Zähler
              brne lineV1
              ;
              cpi r16, 0              ; Sonderfall Y-AnfangsAdresse = 0
              breq LineV_
              ;
LineV2:       clc                     ; Cy löschen, d.h. Nullen einschieben
              rol r0                  ; Cy-Bit nach links einschieben in r0, 0
              dec r16                 ; Zähler AnfangsAdresse
              brne LineV2
              ;
```

Die Sprungadresse LineV_ ist vorläufig, damit der Simulatortest funktioniert.
Der Sonderfall Y-AnfangsAdresse = 0 ist zwar schon berücksichtigt, aber noch nicht der Sonderfall Y-EndAdresse = 7, d.h. wenn das gesamte Byte der 1. Page gesetzt soll sein.

In diesem Zusammenhang wurde auch gleich noch eine Code-Optimierung vorgenommen:

```
GLCD_LineV:   mov r16, r12          ; für Vergleich bereitstellen
              clc                   ; sicherheitshalber Cy löschen
              cpi r16, 8            ; Vergleich mit "8"
              brcc LineV_           ; wenn r16 >= 8          -> PageAdr incr.
                                    ; sonst
; Fall_1
              mov r9, r8            ; Y-EndAdresse sichern
              sub r8, r12           ; Länge der Linie in Pixel (-> r8)
              sub r9, r8            ; Ye - Länge = 0?
              breq LineVa           ; -> Sonderfall: Länge = 8


LineV1:       sec                   ; Cy setzen, d.h. Einsen einschieben -
              rol r0                ; Cy-Bit nach links einschieben in r0, 0
              dec r8                ; LineLängen-Zähler
              brne lineV1
              ;
              cpi r16, 0            ; Sonderfall: Y-AnfangsAdresse = 0
              breq LineV2           ; damit fertig, sonst
              ;
LineV0:       clc                   ; Cy löschen, d.h. Nullen einschieben
              rol r0                ; Cy-Bit nach links einschieben in r0, 0
              dec r12               ; Zähler AnfangsAdresse
              brne LineV0           ; noch nicht alle Nullen eingeschoben?
              ;
              rjmp LineV2           ; sonst fertig
              ;
LineVa:       clr r0                ; r0 löschen
              com r0                ; alle Bits in r0 setzen
              ;
LineV2:       nop                   ; (vorläufig), GLCD_read/write, EndAdresse > 7 usw

LineV_:       nop                   ; (vorläufig), AnfangsAdresse > 7 usw.
```

Damit wäre der `Fall_1` soweit bearbeitet (außer `GLCD read/write` usw.).
Allerdings gibt es noch ein Problem, weil bisher die Begrenzung auf das erste Byte, bzw. der Übergang auf die nächste PageAdresse noch nicht berücksichtigt wurde.

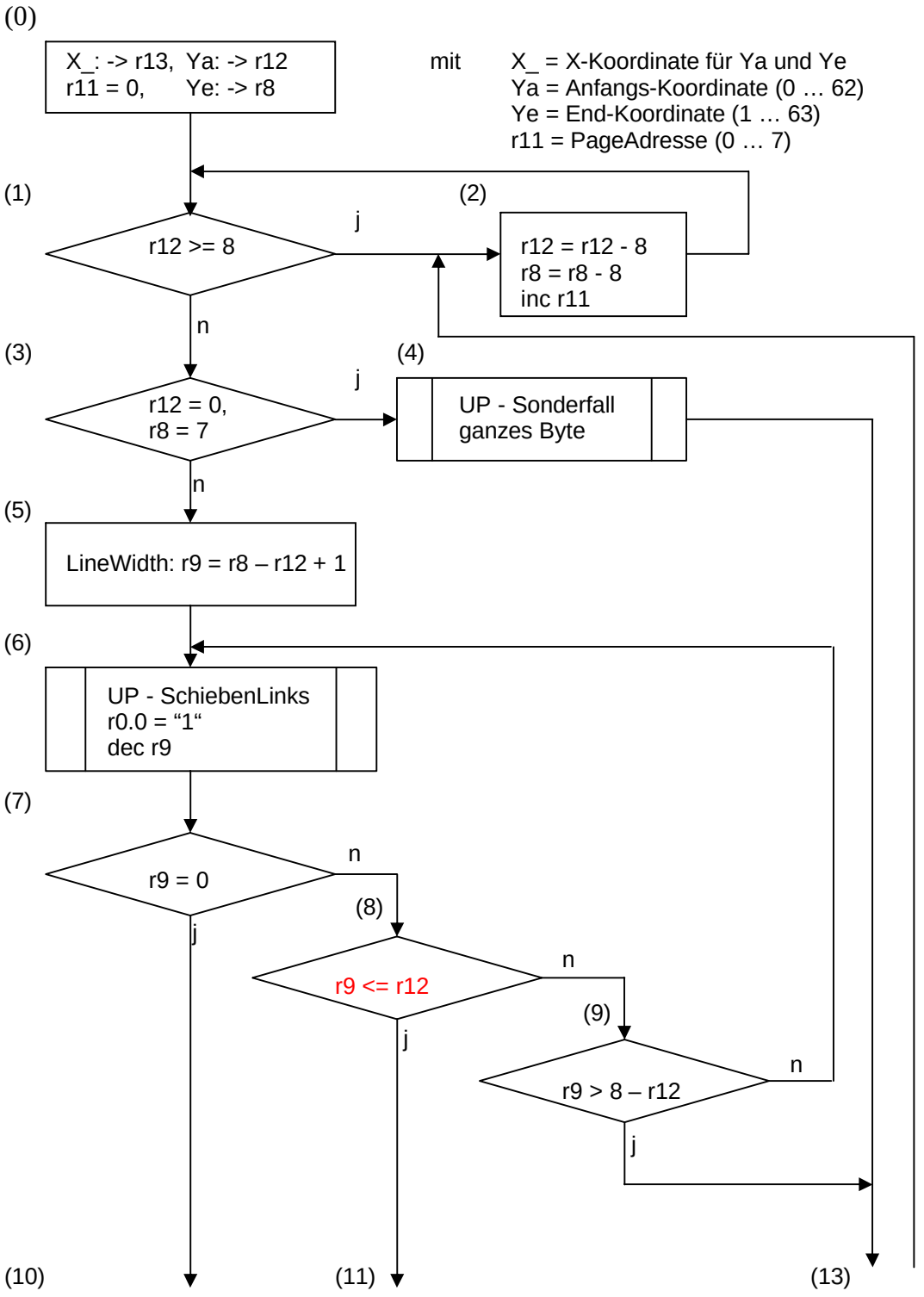
Zu prüfen wäre auch noch, ob es günstiger ist, sofort das erste Daten-Byte in den GLCD-RAM zu schreiben, oder eine Zwischenspeicherung in `r0 … r7` vorzunehmen, um dann in einer konzentrierten Aktion alle relevanten Bytes auf einmal von dort in's RAM zu kopieren. Letzteres würde sich recht komfortabel mit indirekten Ladebefehlen, z.B. `ld X+` realisieren lassen, wobei zunächst in `X` die Anfangs-Adresse von `r0` = 0 gespeichert werden müsste, aber dann durch das Autoincrement sehr einfach automatische die weiteren Register `r1 … r7` angesprochen werden.

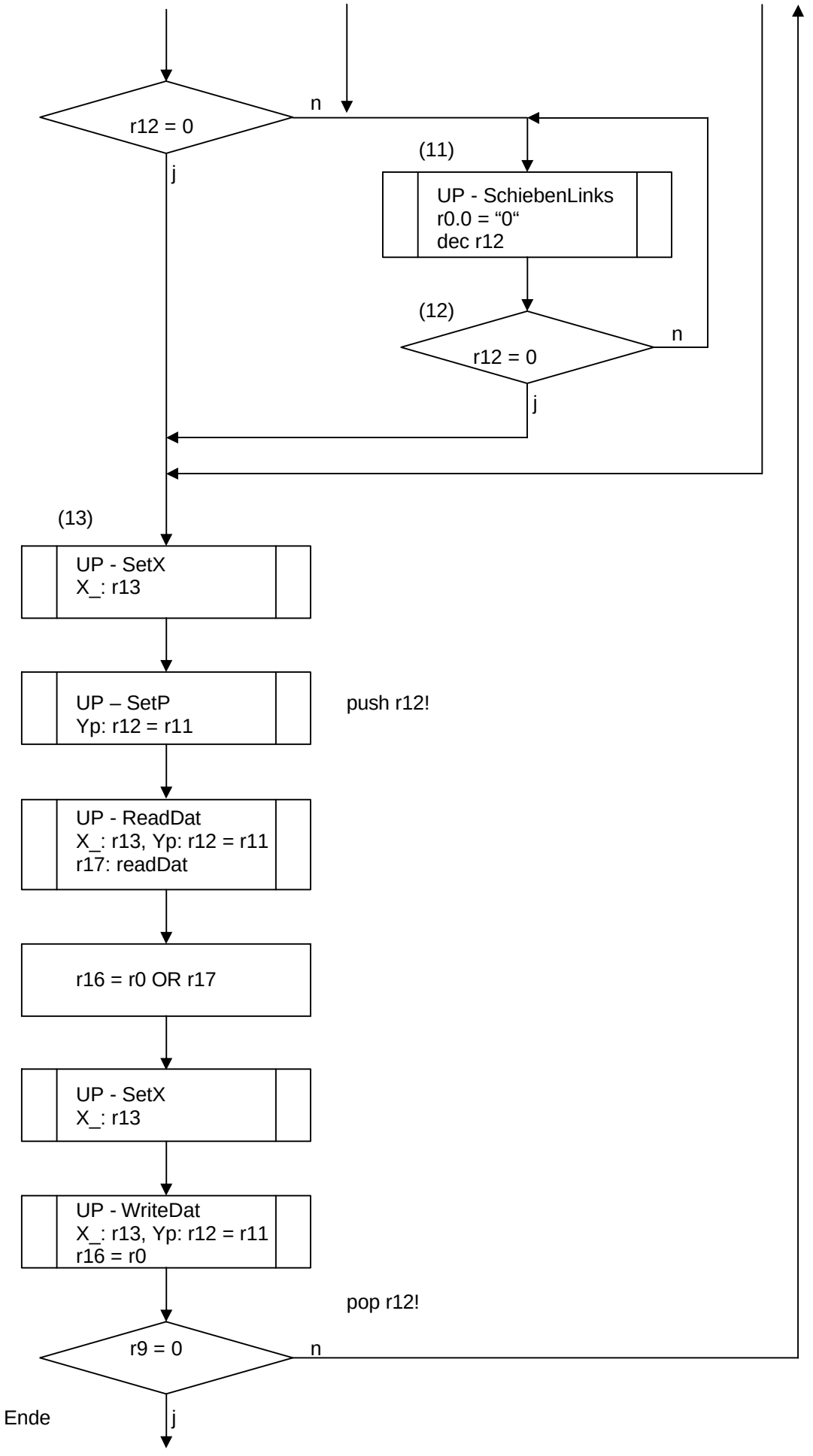
Des Weiteren wäre es zweckmäßig einige Code-Sequenzen als UP zu realisieren, weil diese dann auch in anderen Zusammenhängen (Fall_2 usw) benutzt werden können. Das betrifft z.B. `LineVa`, `LineV0` und `LineV1`.

6.10.4.2 Programmtechnische Realisierung Fall 2

Die bisherige Begrenzung auf die 1. PageAdresse (= erstes DatenByte) macht eine Erweiterung der programmtechnischen Umsetzung erforderlich. Bei der Erarbeitung der Code-Sequenzen für Fall_1 hat sich schon herausgestellt, dass eine strickte Trennung Fall_1, Fall_2 usw. eigentlich nicht erforderlich ist.
Bei der letzten Code-Überarbeitung wurde bereits der Ansatz verfolgt, dass ein nahtloser Übergang möglich ist und somit folgender ProgrammAblaufPlan entwickelt werden kann:

PAP


(0)
X_: -> r13, Ya: -> r12
r11 = 0, Ye: -> r8
mit X_ = X-Koordinate für Ya und Ye
Ya = Anfangs-Koordinate (0 … 62)
Ye = End-Koordinate (1 … 63)
r11 = PageAdresse (0 … 7)
(1)
r12 >= 8
j
(2)
r12 = r12 - 8
r8 = r8 - 8
inc r11
n
(3)
r12 = 0,
r8 = 7
j
(4)
UP - Sonderfall
ganzes Byte
n
(5)
LineWidth: r9 = r8 – r12 + 1
(6)
UP - SchiebenLinks
r0.0 = “1“
dec r9
(7)
r9 = 0
n
j
(8)
r9 <= r12
n
j
(9)
r9 > 8 – r12
n
j
(10)
(11)
(13)


r12 = 0
n
j
(11)
UP - SchiebenLinks
r0.0 = “0“
dec r12
(12)
r12 = 0
n
j
(13)
UP - SetX
X_: r13
UP – SetP
Yp: r12 = r11
push r12!
UP - ReadDat
X_: r13, Yp: r12 = r11
r17: readDat
r16 = r0 OR r17
UP - SetX
X_: r13
UP - WriteDat
X_: r13, Yp: r12 = r11
r16 = r0
pop r12!
r9 = 0
n
Ende
j

Verbale Beschreibung

Beim Aufruf des UP `GLCD_LineV` werden folgende drei Parameter übergeben:

(0)

X-Adresse (0 … 191) -> `r13`

Y-Anfangsadresse (0 … 62) -> `r12`

Y-Endadresse (1 … 63) -> `r8`

Die Einschränkungen bzgl. des Wertebereiches der Y-Anfangs- bzw. Endadresse im Vergleich zum Gesamt-Adressbereich Y des GLCDs von 0 … 63 ist aus der Definition einer Geraden mit mindestens 2 Pixel erklärbar.

Die PageAdresse wird in `r11` auf "0" gesetzt.

(1)

Zunächst wird geprüft, ob die Y-Anfangsadresse außerhalb der 1. PageAdresse ist. Ist das der Fall, wird

(2)

die aktuelle PageAdresse incrementiert, d.h. zunächst auf 1 gesetzt und sowohl von der Y-AnfangsAdresse, als auch von der Y-EndAdresse 8 subtrahiert. Somit wird erreicht, dass sich wieder die gleichen Anfangsverhältnisse einstellen, wie beim Beginn – mit dem Unterschied, dass nunmehr die PageAdresse 1 die Basis ist. Die Programmschleife wird zurückgeführt auf (1) und erneut diese Prüfung durchgeführt – solange bis sich die Y-Anfangsadresse innerhalb der aktuellen PageAdresse befindet.

Dann wird in

(3)

geklärt werden, ob der Sonderfall "ganzes Byte = alle Pixel gesetzt" vorliegt.

Ist das der Fall, wird in

(4)

das UP `LineVall` aufgerufen, wo alle Bits in `r0` gesetzt werden. Der weitere Verlauf mit `GLCD_SetX`, `GLCD_SetP` usw. ist dann in (13) der gleiche wie in den anderen Fällen.

Ist die Abfrage in (3) negativ wird in

(5)

die Länge der Linie in der Variablen `LineWidth` in `r9` gebildet. Diese kann < >8 sein. Der Sonderfall `r9 = 8` ist schon mit (4) abgefangen worden. In

(6)

wird nun das UP `LineV1` (Schieben links „1") aufgerufen, wo eine "1" nach links von Cy in `r0` eingeschoben und `r9` decrementiert wird. Danach wird in

(7)

gestestet, ob `r9 = 0` ist. Das würde bedeuten, daß alle Pixel der vertikalen Linie gesetzt sind. Ist das der Fall, geht es zu einer weiteren Abfrage

(10)

Dort wird untersucht, ob die relative Anfangsadresse in `r12 = 0` ist. Ist das der Fall, so wird der weitere Verlauf mit `GLCD_SetX`, `GLCD_SetP` usw. in (15) fortgesetzt.

Anderenfalls wird das UP `LineV0` (Schieben links „0“) in (11) aufgerufen, wo eine “0“ nach links von Cy in `r0` eingeschoben und `r12` solange decrementiert wird, bis `r12 = 0` ist.

Durch die Abfrage in (1) muß `r12 < 8` sein.

Danach wird der weitere Verlauf mit `GLCD_SetX`, `GLCD_SetP` usw. in (13) fortgesetzt.

Geht die Abfrage in (7) negativ aus, weil noch nicht alle Pixel der Linie eingeschoben sind, wird in

(8)

geprüft, ob die noch einzuschiebende Pixelzahl in `r9 = r12` ist. In diesem Fall wird mit dem UP `LineV0` (Schieben links „0“) in (11) fortgesetzt.

Anderenfalls wird in

(9)

geprüft, ob die Anzahl der noch einzuschiebenden Pixel größer ist, als in die aktuelle PageAdresse passen, d.h. `r9 > 8-r12`. Ist das der Fall, wird wird der weitere Verlauf mit `GLCD_SetX`, `GLCD_SetP` usw. in (13) fortgesetzt. Anderenfalls wird die Schleife in (6) geschlossen, das UP `LineV1` aufgerufen und somit das nächste Pixel in `r0` eingeschoben.

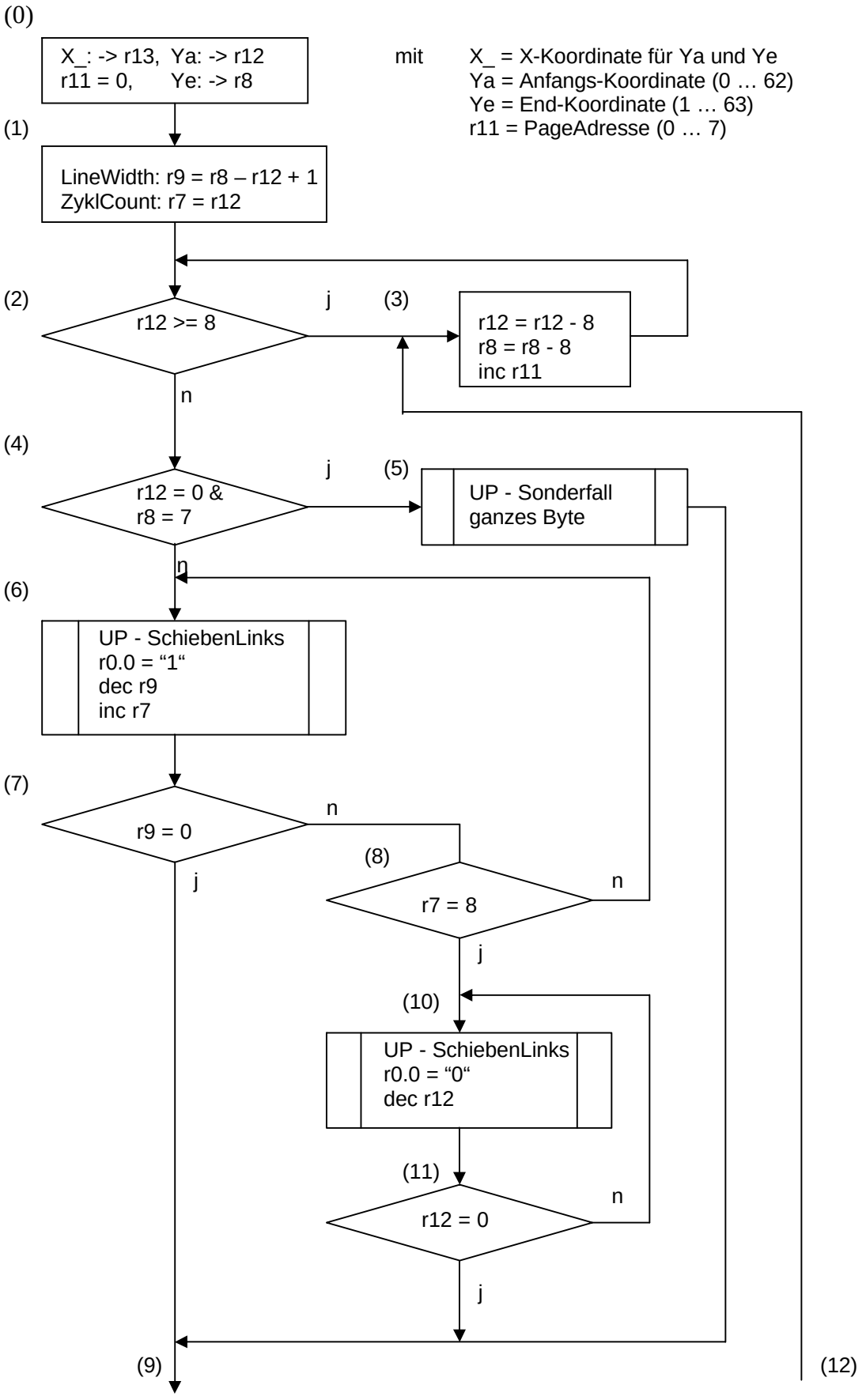
(13)

Korrektur

Bei der Erprobung des ersten Code-Teilabschnitts (0 … 13) stellte sich heraus, daß der Ablauf ab (8) nicht korrekt ist. Die rote Markierung kennzeichnet das. Probleme gibt es insbesondere bei Linien, die über die erste AdressPage hinausgehen.

Zur Abhilfe wird ein weiteres temporäres Register - Zyklus-Zähler eingeführt: `r7`. Dieser überwacht die Schiebesequenzen in einer *PageAdr* auf <= 8. Somit wird gewährleistet, daß bei vollständiger Bereitstellung eines DatenBytes nach den Schiebesequenzen zunächst die Übertragung in das GLCD stattfindet und erst danach die weitere Bearbeitung.

Somit ergibt sich folgender, korrigierter PAP:



Bis (4) ist das Programm noch identisch wie vorher. In
(5)
wird zusätzlich der Zykluszähler *ZyklCount* in `r7` eingeführt und mit der Y-*AnfangsAdr* `r12` geladen. Da `r12` wegen (3) niemals > 7 sein kann, ist auch gewährleistet, daß nur die Pixel innerhalb der aktuellen *PageAdr* bearbeitet werden, auch wenn `r9` noch nicht 0 ist.
Solange sich der Schiebeprozess noch innerhalb der aktuellen *PageAdr* befindet, d.h `r7` noch nicht 8 ist, wird die Schleife nach (6) geschlossen und ein neues Bit eingeschoben.
In dem UP `LineV1` in
(6)
wird außer dem Linksschieben einer "1" in `r0` der *LineWidth*-Wert in `r9` dekrementiert und der *ZyklCount*-Wert in `r7` inkrementiert. In
(7)
wird geprüft, ob `r9` bereits "0" ist, d.h. die Pixel der Linie vollständig in `r0` eingeschoben sind. Ist das der Fall, wird der weitere Ablauf wie gehabt mit `GLCD_SetX`, `GLCD_SetP` usw. in (9) fortgesetzt.
Anderenfalls wird in
(8)
geprüft, ob der *ZyklCount*-Wert bereits "8" erreicht hat. Ist das der Fall, wird in
(10)
das UP LineV0 aufgerufen und das Linksschieben einer "0" in `r0` vorgenommen. In
(11)
wird nun geprüft, ob `r12` bereits „"0" ist, d.h. alle „Nullen" eingeschoben sind. Ist das der Fall, wird der weitere Ablauf wie gehabt mit `GLCD_SetX`, `GLCD_SetP` usw. in (9) fortgesetzt.
Anderenfalls wird die Schleife zu (10) geschlossen.

Bevor die Abarbeitung der UP `GLCD_SetX`, `GLCD_SetP` usw. weiter geführt wird – und über (12) die große Schleife geschlossen werden kann, soll zu nächst ein Simulator-Test der bisherigen Codesequenz mit folgenden Beispielen stattfinden:

AnfangsAdr	EndAdr	Ergebnis
0	5	ok
0	7	ok
0	12	ok, d.h. mit Codeverkürzung
2	5	ok
2	7	ok
5	12	ok

Der vorläufige, d.h. ohne Hardwaretest endgültige Code für das UP `GLCD_LineV` ist dann:

```
;=====================================================================
;
; UP Line vertikal (UP von LineAllg)
;                          - Übergabeparameter:    r13 -   X-Adresse (0 ... 191)
;                                                  r12 -   Y-AnfangsAdresse (>=0)
;                                                  r8  -   Y-Endadresse   (<=63)
;                          - Rückgabeparameter:    r13 -   X-Adresse
;                                                  r16 -   Datenbyte (0 ... 255)
;                                                  r12 -   PageAdresse
;---------------------------------------------------------------------
;
GLCD_LineV:     clr r11             ; PageAdresszähler r11 auf "0" setzen   - (0)
                push r8             ; Y-Endadresse sichern
                sub r8, r12         ;
                mov r9, r8          ;
                inc r9              ; LinienLänge in Pixel: r9 = r8 - r12 +1   - (1)
                pop r8
                ;
LineV_1:        clr r0              ; temp. Schiebedatenregister auf Null setzen
                mov r16, r12        ; für Vergleich bereitstellen
                clc                 ; sicherheitshalber Cy löschen
                cpi r16, 8          ; Vergleich mit "8"                     - (2)
                brcc LineV_2        ; wenn r12 >= 8->       PageAdresse incr.
                                    ; sonst
; Sonderfall ganzes Byte
                push r12            ; Y-Anfangsadresse sichern
                ldi r16, 1
                sub r12, r16
                pop r12             ; Y-Anfangsadresse wiederherstellen
                brcc LineV_5        ; wenn r12 <> 0 war
                                    ; sonst
                ldi r16, 7
                clc                 ; sicherheitshalber Cy löschen
                cp r8, r16          ;                                       - (5)
                brcc LineV_4        ; Sonderfall r12 = 0, r8 >= 7: -> ganzes Byte
                                    ; sonst
LineV_5:        mov r7, r12         ; ZyklusZähler vorladen
                ;
LineV_6:        rcall LineV1        ; „Einsen“ nach links einschieben       - (6)
                ;
                mov r16, r9         ; für Vergleich bereitstellen
                cpi r16, 0          ; alle "Einsen" eingeschoben?
                breq LineV0e        ; dann "Nullen" prüfen
                                    ; sonst
```

```
                mov r16, r7         ; für Vergleich bereitstellen
                cpi r16, 8          ; Schleife für akt. PageAdr, incl Nullen   - (8)
                brcs LineV_6
                ;
                rjmp LineV0e        ; sonst
                ;
LineV_11:       rcall LineV0        ; „Nullen“ nach links einschieben          - (10)
                ;
LineV0e:        mov r16, r12        ; für Vergleich bereitstellen
                cpi r16, 0          ; alle „Nullen“ eingeschoben?              - (11)
                brne LineV_11       ;
                                    ; sonst fertig
;
LineVset:       rjmp LineVe         ; nur beim Simulator-Test! - Bis LineVe überspringen!
                                    ; sonst raus!
                                    ;
                rcall GLCD_SetX     ; Setzen der X-Adresse,
                                     ; Übergabeparameter: r13                  - (9)
                                    ;
                push r12
                mov r12, r11        ; PageAdresse bereitstellen
                rcall GLCD_SetP     ; Setzen der Page-Adresse,
                                    ; Übergabeparameter: r12 (0 ... 7)
                ;
;
; für Variante ohne Read/Write nachfolgende Sequenz bis LineVwr ausblenden
;
                pop r12
                rcall GLCD_read     ; aktuelles DatenByte GLCD-RAM auslesen -> r17
                                    ; X-Adresse wird RAM-intern incrementiert!
                mov r16, r0         ; das zu schreibende Byte bereitstellen
                or r16, r17         ; mit vorhandenen RAM-Daten verknüpfen
                rcall GLCD_SetX     ; erneut Setzen der X-Adresse,
                                    ; Übergabeparameter: r13
                ;
LineVwr:        rcall GLCD_write    ; Schreiben GLCD-RAM, GLCD incr. X-Adresse
                ;
LineVe:         mov r16, r9
                cpi r16, 0          ; in r9 kein Rest mehr?
                breq LineVend       ; vertikale Line fertig gezeichnet
                                    ; sonst
                ;
LineV_2:        mov r16, r12        ; AnfangsAdr bereitstellen                 - (3)
                subi r16, 8         ; AnfangsAdr > 7 usw.
                brcc LineV20
                ;
                clr r16             ; bei neg. -> r12 auf Null setzen
                ;
LineV20:        mov r12, r16        ; relative AnfAdr neu
                mov r16, r8         ; Endadr bereitstellen
                subi r16, 8         ; relative EndAdr neu
                mov r8, r16
                inc r11             ; nächste PageAdr
                rjmp LineV_1        ; Schleife schliessen
                ;
LineV_4:        rcall LineVall      ; ganzes Byte                              - (4)
                ;
                rjmp LineVset
                ;
;
```

```
LineVend:  ret
;
;=============================================================
;
; UP Schieben "0" von Cy nach links in r0
;                          Übergabeparameter:      r12     -       Y-AnfangsAdresse
;-------------------------------------------------------------------------------------------------------
;
LineV0:          clc                    ; Cy löschen, d.h. Nullen einschieben
                 rol r0                 ; Cy-Bit nach links einschieben r0,0
                 dec r12                ; Zähler AnfangsAdresse
                 ;
                 ret
;
;=====================================================================
;
; UP Schieben "1" von Cy nach links in r0
;                 Übergabeparameter:      r9      -       LinienLänge
;                                                         r7  -   temp. ZyklusZähler
;-------------------------------------------------------------------------------------------------------
;
LineV1:          sec                    ; Cy setzen, d.h. Einsen einschieben -
                 rol r0                 ; Cy-Bit nach links einschieben in r0,0
                 dec r9                 ; LinienLängen-Zähler
                 inc r7                 ; temp. Zykluszähler
                 ;
                 ret
;
;=====================================================================
;
; UP alle Bits in r0 setzen
;-------------------------------------------------------------------------------------------------------------------
;
LineVall:        clr r0                 ;
                 com r0                 ; alle Bits in r0 setzen
                 ldi r16, 8
                 sub r9, r16            ; LinienLänge - 8
                 ret
;
```