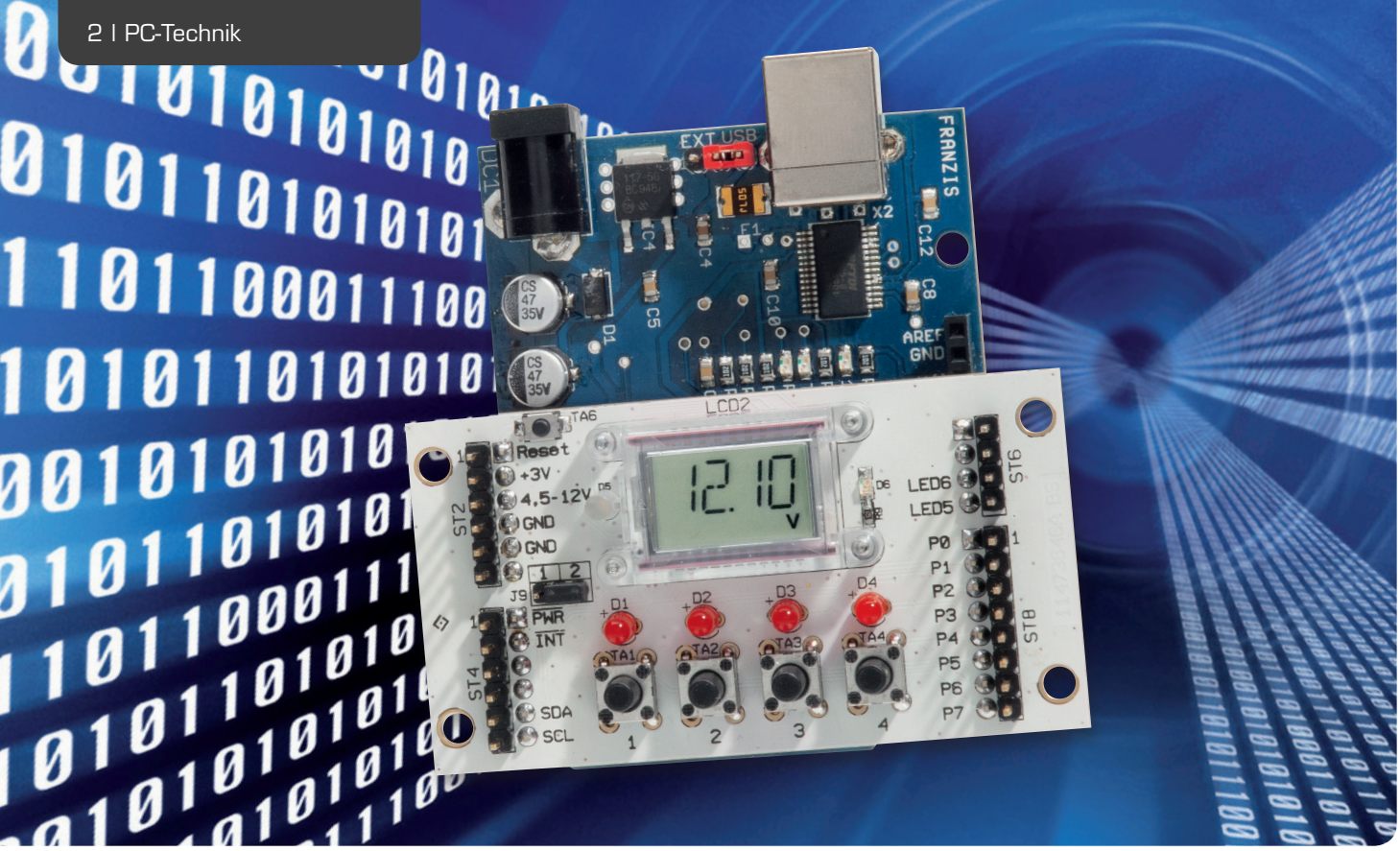


Programmierung des I²C-LCD

Das I²C-Displaymodul bietet die Möglichkeit, auf einem 4-stelligen Display mithilfe von 14-Segmentanzeigen Zahlen, Buchstaben und andere Zeichen darzustellen. Neben den 4 Segmenten ist es außerdem möglich, einige andere Symbole wie z. B. °C, %, A oder V im unteren Bereich des Displays anzuzeigen, sodass das LCD z. B. zur Anzeige einer Temperatur, einer Uhrzeit oder einer Spannung genutzt werden kann. Aufgrund der Anordnung der 4 Stiftleisten ist die Platine auch als Arduino-Shield verwendbar. Den Weg der Programmierung des Displaymoduls vom Schaltplan und Datenblatt des LCD-Treibers zur Darstellung auf dem Display zeigt im Folgenden dieser Artikel anhand einer Library für die Arduino-Entwicklungsumgebung.

Einleitung

Damit wir mithilfe der Arduino-Umgebung Zahlen, Buchstaben und andere Symbole auf dem Display des I²C-LCD darstellen und die zusätzlich montierten LEDs und Taster ansprechen können, bietet sich die Erstellung einer Library an. Dieses hat den Vorteil, dass wir nach Erstellung der Library bei der Entwicklung von neuen Projekten direkt auf die hierin implementierten Methoden zurückgreifen können, ohne jedes Mal das Rad neu erfinden zu müssen. Durch die Wiederverwendung einer Library wird zudem die Wahrscheinlichkeit verringert, dass sich in einem neuen Projekt ein Fehler einschleicht. Warum sollte eine Methode, die bereits bei 10 Projekten gute Dienste geleistet und keine Fehler aufgeworfen hat, beim 11. Projekt plötzlich Fehler machen? Möglich ist es, aber die Wahrscheinlichkeit, dass ein Fehler auftritt, ist beim Neuschreiben dieser Routinen natürlich um einiges höher, als wenn wir die Routinen der Library immer wieder verwenden. Zum anderen hat eine Library den Vorteil, dass alle Methoden, die zur Kommunikation mit dem I²C-LCD benötigt werden, zusammengefasst sind und nicht über viele Dateien eines Projektes verteilt liegen. Eine Library macht die Verwendung der Methoden daher sehr übersichtlich.

Im Folgenden wird nun Schritt für Schritt die Implementierung der Displayroutinen innerhalb der Library TwoWireLCD erklärt. Da das I²C-Displaymodul neben dem LCD auch über einige LEDs und Taster verfügt, wird im hinteren Teil dieses Artikels auch eine Möglichkeit aufgezeigt, wie mit diesen beiden Komponenten mithilfe der Library gearbeitet werden kann. Des Weiteren werden abschließend insgesamt vier einfache Beispiele vorgestellt, die praktisch die Verwendung der hier entwickelten Library aufzeigen sollen.

Bevor wir nun damit beginnen können, eine solche Library zu schreiben, müssen wir uns zunächst einmal einige Fragen stellen:

- Wie kommunizieren wir mit dem I²C-Displaymodul?
- Welche Einstellungen müssen wir beim Einschalten des Moduls machen?
- In welcher Form müssen die Zeichen vorliegen, die wir darstellen möchten, und wie speichern wir diese am besten?
- Wie stellen wir die Zeichen auf dem Display dar?

Kommunikation mit dem I²C-Displaymodul

Schauen wir uns zunächst einmal das Schaltbild des I²C-Displaymoduls (Bild 1) an, so fällt uns auf, dass der LCD-Treiberbaustein CP2401 die zentrale Komponente des Moduls ist. Wollen wir also auf dem Display etwas anzeigen, müssen wir es diesem IC mitteilen. Die eingangs gestellte Frage „Wie kommunizieren wir mit dem I²C-Displaymodul?“ müssen wir also in die Frage „Wie kommunizieren wir mit dem LCD-Treiberbaustein CP2401?“ ändern. Ein Blick ins Datenblatt des CP2401 [1] verrät uns, dass dieser Baustein über die I²C-Schnittstelle bzw. SMBus angesprochen wird. Mithilfe des I²C-Busses lassen sich also die internen Register des CP2401 ansprechen und verändern, die für alle Einstellungen und Ausgaben zuständig sind und somit auch für die Daten, die wir auf dem Display darstellen wollen. Den genauen Ablauf einer solchen Kommunikation zeigt uns Kapitel 6.2 [1]. Hier wird beschrieben, wie die Register ausgelesen bzw. geschrieben werden können. Im ersten Schritt müssen wir also in unserer Library diese Kommunikation (Lesen und Schreiben) in Quellcode umsetzen.

Beginnen wir zunächst mit dem Schreiben von Registern. Da die Arduino-Programmierumgebung bereits eine Library zur Kommunikation über den I²C-Bus bietet (Library „Wire“), brauchen wir uns nicht mehr um die einzelnen I²C-Parameter bzw. Register des ATMegas zu kümmern, sondern müssen nur noch die Daten in entsprechender Reihenfolge an den CP2401 mithilfe dieser Library senden. Um nun Daten an den Baustein zu senden, ist in unserer TwoWireLCD-Library die Methode writeRegister (Bild 2) implementiert. Zur Initialisierung der Wire-Library wird zu Beginn ein Aufruf der Methode Wire.begin() gefordert. Da dieser Aufruf jedoch nur ein einziges Mal im gesamten Programmablauf stattfinden muss und ein Mehrfachaufruf somit verhindert werden soll, ist diese Initialisierung nicht innerhalb der writeRegister-Methode implementiert, sondern sie muss vom Anwender der Library übergeordnet (z. B. innerhalb der setup-Methode) aufgerufen werden. In welcher Reihenfolge nun die Daten beim Schreiben der Register an den CP2401 gesendet werden müssen, zeigt uns die Abbildung 6.3 im Datenblatt [1]. Nach dem Start der Kommunikation wird zunächst die I²C-Adresse des anzusprechenden Bausteins übermittelt. Die I²C-Adresse des CP2401 kann je nach Beschaltung des SMBA0-Pins die Werte 0x74 (SMBA0-Pin an GND) und 0x76 (SMBA0-Pin an VDD) annehmen (Kapitel 15.3.5 [1]). Da der SMBA0-Pin in unserer Schaltung an VDD (+3V) liegt, müssen wir uns die Adresse 0x76 merken. Innerhalb der Methode writeRegister wird die Kommunikation mithilfe der Zeile Wire.beginTransmission(address) gestartet. Die I²C-Adresse wird der Methode writeRegister als Parameter-Address übergeben, sodass die Methode universell einsetzbar ist. Nachdem die Adresse eingestellt wurde, muss dem CP2401 als Nächstes mitgeteilt werden, wie die Daten geschrieben werden sollen. Beim Schreiben gibt es laut Tabelle 6.2 [1] zwei Möglichkeiten (REGSET und REGWRITE). Mithilfe von REGSET (0x03) werden alle übermittelten Daten an die gleiche

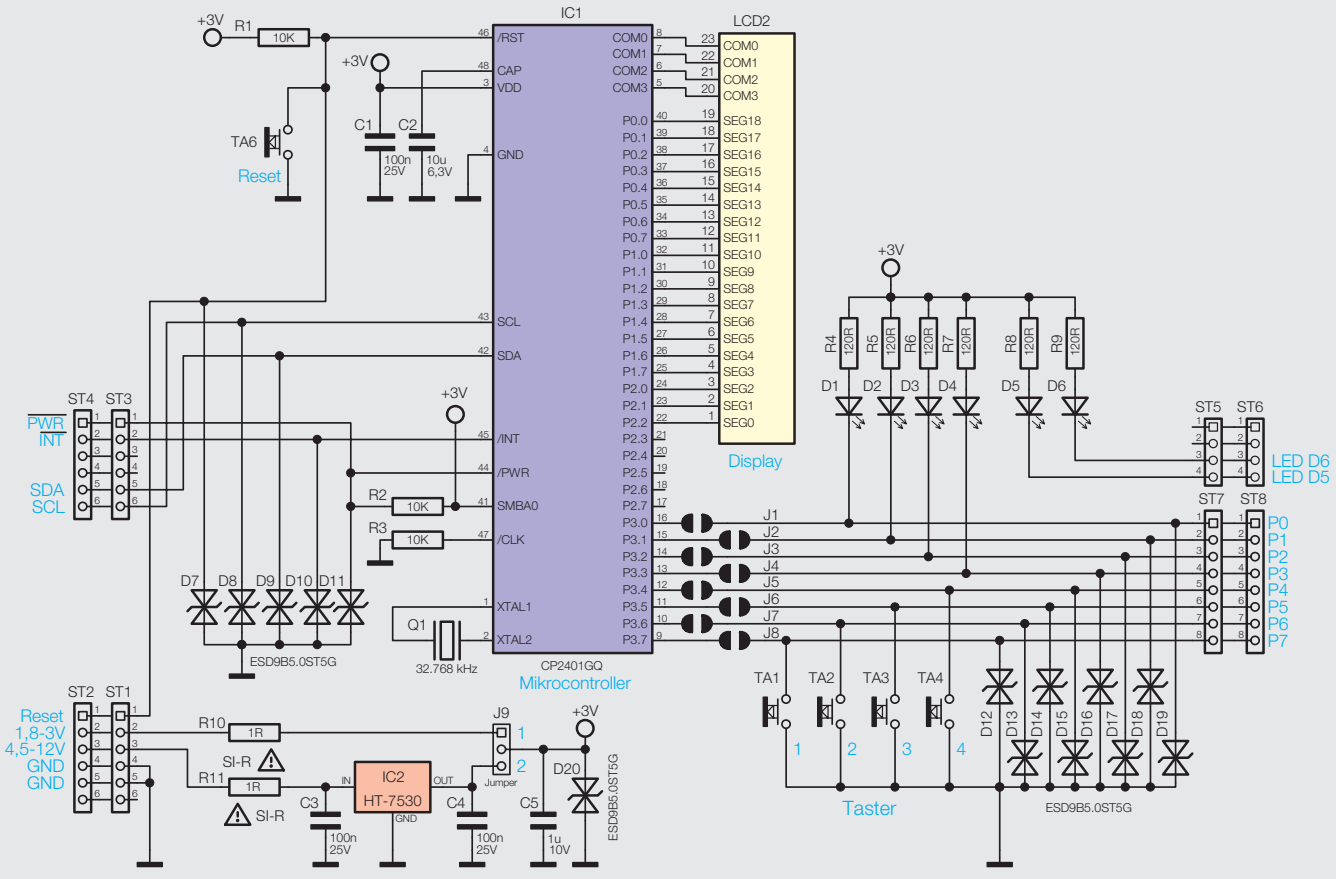


Bild 1: Schaltbild des I²C -Displaymoduls

```
void TwoWireLCD::writeRegister(unsigned char address, unsigned char command,
unsigned char regAddress, unsigned char *data, unsigned char datalength)
{
   Wire.beginTransmission(address);
   Wire.send(command);
   Wire.send(0x00);
   Wire.send(regAddress);
   Wire.send(data, datalength);
   Wire.endTransmission();
}


unsigned char TwoWireLCD::readRegister(unsigned char address, unsigned char
command, unsigned char regAddress, unsigned char length, unsigned char *data)
{
   unsigned char readBytes = 0;

   Wire.beginTransmission(address);
   Wire.send(command);
   Wire.send(0x00);
   Wire.send(regAddress);
   Wire.endTransmission();

   delay(10);

   length += 4;

   Wire.requestFrom(address, length);

   delay(10);

   Wire.receive();
   Wire.receive();
   Wire.receive();
   Wire.receive();

   while(Wire.available())
   {
      data[readBytes] = Wire.receive();
      readBytes++;
   }

   return readBytes;
}
```

Bild 2: LCD-Kommunikationsroutinen writeRegister und readRegister

```
void TwoWireLCD::begin(void)
{
   unsigned char i = 0;

   while(CP240xInitSequence[i] != 0)
   {
      writeRegister(I2C_LCD_ADDRESS, CP240xInitSequence[i+1],
         CP240xInitSequence[i+2], &CP240xInitSequence[i+3],
         CP240xInitSequence[i]);

      i += (3 + CP240xInitSequence[i]);
   }
}
```

Bild 3: LCD-Initialisierungsroutine begin

Registeradresse geschrieben, mit REGWRITE (0x04) wird ein sequenzielles Schreiben der übermittelten Daten ab der angegebenen Registeradresse durchgeführt. Nach diesem Befehl folgt die Registeradresse (ADDRH und ADDRL), ab der die folgenden Daten geschrieben werden sollen. Da alle Register im Adressbereich zwischen 0x0000 und 0x00FF (Abbildung 6.1 [1]) liegen, können wir den Wert für ADDRH mit 0x00 festsetzen und müssen der Methode writeRegister nur noch das untere Byte der Registeradresse übergeben (regAddress). Abschließend folgt die Übermittlung der Daten. Alle folgenden Bytes, die an den CP2401 gesendet werden, werden als Datenbytes interpretiert und in die entsprechenden Register geschrieben. Da die Anzahl der Daten variabel sein kann, übergeben wir der Methode writeRegister ein Array mit den zu schreibenden Daten (data) und die Länge dieses Datenarrays (datalength). Mithilfe des Befehls Wire.send(data, datalength) werden diese Daten an den CP2401 gesendet und dort verarbeitet. Abschließend müssen wir die Kommunikation mit dem Befehl Wire.endTransmission beenden. Dieser Befehl sorgt für das Senden der Stopp-Bedingung des I²C-Busses, welcher somit wieder zur Verfügung steht und von anderen Bausteinen genutzt bzw. eine neue Kommunikation zum CP2401 gestartet werden kann.

Neben dem Schreiben von Registern des CP2401 brauchen wir auch die Möglichkeit, Register lesen zu können. Innerhalb der Library ist dieser Lesevorgang in der Methode readRegister implementiert (Bild 2). Das Lesen von Registern teilt sich in zwei Schritte auf. Im ersten Schritt müssen wir dem Baustein mitteilen, wie gelesen und ab welcher Registeradresse gelesen werden soll. Wie beim Schreiben haben wir auch beim Lesen zwei Möglichkeiten, wie die Register ausgelesen werden (REGPOLL und REGREAD, Tabelle 6.2 [1]). Mit REGPOLL (0x01) wird lediglich ein Register bei jeder Anfrage neu ausgelesen. Im Gegensatz dazu wird bei REGREAD (0x02) nach jedem Lesen der Adresszähler inkrementiert, sodass bei der nächsten Anfrage der Inhalt des folgenden Registers zurückgegeben wird (sequenzielles Lesen). Wie bei der Methode writeRegister wird dieser Befehl wie auch die Adresse an die Methode readRegister übergeben und an den CP2401 übermittelt. Nachdem der Baustein nun zum Lesen der Daten vorbereitet wurde, folgt im zweiten Schritt das Auslesen. Mithilfe des Befehls Wire.requestFrom(address, length) wird nun die entsprechende Anzahl Daten (length) vom CP2401 abgeholt. Nachdem die Daten vom LCD-Treiber gelesen wurden, können diese einzeln mithilfe der Methode Wire.receive() aus dem internen Speicher der Wire-Library abgeholt werden. Hierbei ist zu beachten, dass die ersten 4 Bytes, die der CP2401 zurücksendet, keine Nutzdaten enthalten, sodass diese verworfen werden müssen. Daher wird vor dem Lesevorgang die Anzahl der zu lesenden Daten um 4 erhöht, um weiterhin die richtige Menge an Daten zu erhalten. Abschließend werden die Daten in das übergebene Array (data) kopiert. Da die Anzahl der gelesenen Daten unter Umständen nicht der Anzahl der ursprünglich zu lesenden Daten entspricht, werden nur die Daten übernommen, die auch gelesen wurden, und diese Anzahl von der Methode readRegister zurückgegeben.

Initialisierung des I²C-Displaymoduls

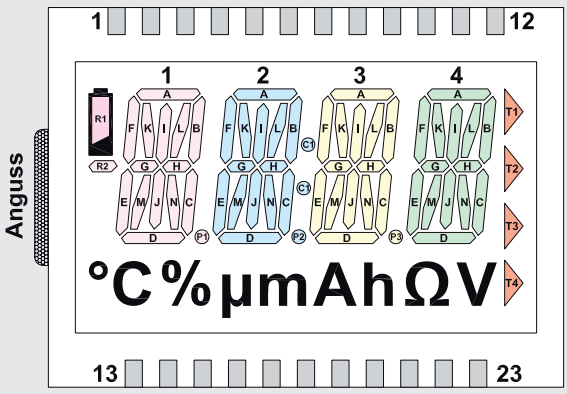
Da wir nun mit dem I²C-Displaymodul oder besser gesagt mit dem LCD-Treiber CP2401 kommunizieren können und somit in der Lage sind, die Register zu schreiben bzw. zu lesen, geht es im nächsten Schritt darum, den LCD-Treiber zu initialisieren. Bei der Initialisierung wird dem CP2401 mitgeteilt mit welchem Clock er arbeiten soll, ob die I/Os als Eingang oder Ausgang geschaltet sind, wie viele Segmente und wie viele COM-Leitungen das angeschlossene LCD hat. Hinzu kommen die Einstellungen zur Kontrastspannung, Aktualisierungsrate usw.

Innerhalb der TwoWireLCD-Library wird die Initialisierung mithilfe der Methode begin durchgeführt (Bild 3). Da es bei der Initialisierung des CP2401 darum geht, die unterschiedlichen Register mit Daten zu füllen, sind diese Daten innerhalb eines Arrays ab-

gelegt, welches innerhalb der begin-Methode ausgelesen wird, und die Daten werden mithilfe der vorher beschriebenen Methode writeRegister an den CP2401 gesendet. Die Initialisierungsdaten sind hierbei in der Datei CP240x_lcd.h im Array CP240xInitSequence gespeichert (Bild 4).

```
unsigned char CP240xInitSequence[] =
{
   // Port 0,1,2,4 als analoge I/Os und Port 3 als digitalen I/O
   5, REGWRITE, CP240X_P0MDI, 0x00, 0x00, 0x00, 0xFF, 0x00,

   // Port 3 als open-drain betreiben
   1, REGWRITE, CP240X_P3MDO, 0x00,

   // Port 3 auf High setzen
   1, REGWRITE, CP240X_P3OUT, 0xFF,

   // Schreibschutz für den RTClock entfernen
   2, REGSET, CP240X_RTCKEY, 0xA5, 0xF1,

   // RTClock deaktivieren
   2, REGWRITE, CP240X_RTCADR, (0x10 | RTC0CN), 0x00,

   // kein «AGC» verwenden, externen Quarz verwenden, kein «Bias Double»
   // verwenden
   2, REGWRITE, CP240X_RTCADR, (0x10 | RTC0XCN), 0x04,

   // kein «Automatic Load Capacitance Stepping» verwenden,
   // «Load Capacitance» einstellen
   2, REGWRITE, CP240X_RTCADR, (0x10 | RTC0XCF), 0x0C,

   // RTClock wieder aktivieren
   2, REGWRITE, CP240X_RTCADR, (0x10 | RTC0CN), 0x80,

   // LCD0 Control Register: 32 Segmente, 4-mux-Modus, 1/3 Bias
   // Contrast Adjustment: 3,02 V
   // LCD Configuration: 0x9F
   // LCD Refresh Rate: 64 Hz
   // Toggle Rate Divider: 2048
   // Charge Pump Clock: 1 MHz
   7, REGWRITE, CP240X_LCD0CN, (0x08 | 0x06 | 0x00), 0x08, 0x9F, 0x1F, 0x00,
0x0C, 0x00,

   // LCD einschalten
   1, REGWRITE, CP240X_MSCN, 0x01,

   // Ende der Initialisierung
   0
};
```

Bild 4: Initialisierungsarray CP240xInitSequence (CP240x_lcd.h)



PIN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
COM0	2F	2A	2L	2B	3F	3A	3L	3B	4F	4A	4L	4B	1B	1L	1A	1F	m	A	T1				COM0
COM1	2E	2K	2I	2H	3E	3K	3I	3H	4E	4K	4I	4H	1H	1I	1K	1E	µ	h	T2			COM1	
COM2	P1	2G	2J	2N	C1	3G	3J	3N	P3	4G	4J	4N	1N	1J	1G	R2	%	Ω	T3		COM2		
COM3		2M	2D	2C	P2	3M	3D	3C		4M	4D	4C	1C	1D	1M	R1	°C	V	T4	COM3			
CP 2401	LCD18	LCD17	LCD16	LCD15	LCD14	LCD13	LCD12	LCD11	LCD10	LCD9	LCD8	LCD7	LCD6	LCD5	LCD4	LCD3	LCD2	LCD1	LCD0	COM3	COM2	COM1	COM0

Bild 5: Die Segmente des LC-Displays mit ihrer Pin-Zuordnung

Durch diese Anordnung der Daten bleibt die Methode begin sehr übersichtlich und klein. Des Weiteren muss bei einer Anpassung der Initialisierungsdaten lediglich das Datenarray angepasst werden, die Methode begin bleibt unangetastet. Das Array CP240xInitSequence ist so aufgebaut, dass jede Zeile einer Kommunikation mit dem CP2401 entspricht. Der erste Wert einer jeden Zeile gibt die Anzahl der zu sendenden Daten an, der zweite Wert die Art des Schreibvorgangs und der dritte Wert die Registeradresse, ab der geschrieben werden soll. Danach folgen die Datenbytes entsprechend der Anzahl am Anfang der Zeile. Bei der Art des Schreibvorgangs und der Registeradresse wird auf Definitionen zurückgegriffen. Diese finden sich in der Datei CP240x_reg.h wieder. Vorteil dieser Definitionen ist eine sofortige Zuordnung der Adressen, ohne kryptische Werte kennen zu müssen. Auf die Beschreibung der einzelnen Zeilen des Initialisierungsarrays wird hier jedoch verzichtet, da diese wenig zur eigentlichen Programmierung der Library beiträgt und außerdem eine Beschreibung nur einer Wiedergabe der Daten im Datenblatt entsprechen würde. Wer sich hiermit genauer beschäftigen oder eventuell Änderungen vornehmen möchte, dem sei ein Blick ins Datenblatt [1] nahegelegt.

Berechnung der Zeichen mit dem Segmentanzeigenrechner

Nachdem der CP2401 nun von uns initialisiert wurde, geht es im nächsten Schritt darum, Daten auf dem Display darzustellen bzw. wie diese Daten auszusehen haben, damit wir die gewünschten Ergebnisse erhalten. In Kapitel 12 des CP2401-Datenblatts [1] erfahren wir, dass die sogenannten ULP-Register die Daten für die Darstellung auf dem Display enthalten und vom LCD-Segmenttreiber zyklisch ausgelesen werden. Dabei stellt sich uns aber nun die Frage, welche Bits wir innerhalb dieser Register setzen müssen, um z. B. die Zahl 6 im Display an der ersten Stelle anzeigen zu lassen. Die Zuordnungstabelle des LC-Displays (Bild 5) zeigt uns den Zusammenhang zwischen den einzelnen Segmenten und den Pins des Displays, den Anschluss des LCDs an den CP2401 zeigt uns der Schaltplan (Bild 1) und das Datenblatt des CP2401 [1] liefert die Informationen, welches ULP-Register für welchen Port-Pin zuständig ist. Mithilfe dieser 3 Angaben lässt sich somit die Zuordnung der ULP-Register zu den Segmenten des LC-Displays herstellen.

Ein Beispiel: Das Segment T1 wird über den Pin 19 des LCDs angesteuert. Der Pin 19 wiederum ist mit dem Portpin P0.0 des CP2401 verbunden. Für den Portpin P0.0 wiederum ist das Register ULPMEM00 zuständig. Hieraus ergibt sich also, dass das Segment T1 des LCDs mithilfe des Bit 0 des Register ULPMEM00 angesteuert wird. Gehen wir nun alle Segmente des LCDs durch, erhalten wir für die ULP-Register die Zuordnung nach Bild 6.

Da uns die Zuordnung nun bekannt ist, können wir uns wieder der Ursprungsfrage widmen, welche Bits gesetzt werden müssen, um ein bestimmtes Zeichen auf dem Display sichtbar zu machen. In Bild 6 sehen wir, dass sich die Bits für ein Zeichen im-

Bit: 7	6	5	4	3	2	1	0	
								ULPMEM 15 (Pins: LCD 31, LCD 30)
								ULPMEM 14 (Pins: LCD 29, LCD 28)
								ULPMEM 13 (Pins: LCD 27, LCD 26)
								ULPMEM 12 (Pins: LCD 25, LCD 24)
								ULPMEM 11 (Pins: LCD 23, LCD 22)
								ULPMEM 10 (Pins: LCD 21, LCD 20)
					P1	2E	2F	ULPMEM 09 (Pins: LCD 19, LCD 18)
2M	2G	2K	2A	2D	2J	2I	2L	ULPMEM 08 (Pins: LCD 17, LCD 16)
2C	2N	2H	2B	P2	C1	3E	3F	ULPMEM 07 (Pins: LCD 15, LCD 14)
3M	3G	3K	3A	3D	3J	3I	3L	ULPMEM 06 (Pins: LCD 13, LCD 12)
3C	3N	3H	3B		P3	4E	4F	ULPMEM 05 (Pins: LCD 11, LCD 10)
4M	4G	4K	4A	4D	4J	4I	4L	ULPMEM 04 (Pins: LCD 9, LCD 8)
4C	4N	4H	4B	1C	1N	1H	1B	ULPMEM 03 (Pins: LCD 7, LCD 6)
1D	1J	1I	1L	1M	1G	1K	1A	ULPMEM 02 (Pins: LCD 5, LCD 4)
R1	R2	1E	1F	°C	%	µ	m	ULPMEM 01 (Pins: LCD 3, LCD 2)
V	Ω	h	A	T4	T3	T2	T1	ULPMEM 00 (Pins: LCD 1, LCD 0)
COM3	COM2	COM1	COM0	COM3	COM2	COM1	COM0	

Bild 6: ULPMEM-Register des CP2401

mer über 3 ULP-Register verteilen. Dabei sind die Bits für das 2., 3. und 4. Segment (absteigend: xxEF MGKA DJIL CNHB) gleich aufgebaut. Nur beim 1. Segment sind jeweils 4 Bits spiegelverkehrt in den ULP-Registern angeordnet (absteigend: CNHB DJIL MGKA xxEF). Die Tatsache, dass beim 1. Zeichen die Bits spiegelverkehrt sind, lässt sich aber später in der Firmware mit wenig Aufwand korrigieren, sodass wir das Bitmuster für z. B. die Zahl 6 nur ein einziges Mal im Speicher benötigen und einfach entsprechend der darzustellenden Position in die ULP-Register kopieren müssen.

Auf Grundlage der ermittelten Bit-Reihenfolge im ULP-Register können wir uns nun ein Array aufbauen, in dem alle Bitmuster der darzustellenden Zeichen gespeichert werden. Um z. B. die Zahl 6 darstellen zu können, müssen die Segmente A, C, D, E, F, G und H eingeschaltet werden, alle anderen Segmente bleiben aus. Fügt man diese Information nun in die Bit-Reihenfolge im ULP-Register ein, ergibt sich das Bitmuster 0011 0101 1000 1010 (hexadezimal 0x358A).

Wir könnten nun für alle Zahlen, Buchstaben und andere Zeichen diese Bitmuster mit Zettel und Stift ermitteln, jedoch ist diese Vorgehensweise zum einen sehr zeitintensiv, zum anderen aber auch sehr fehleranfällig, da es schnell dazu kommen kann, dass man zwei Segmente miteinander vertauscht, sodass ein Zeichen später auf dem Display nicht so aussieht, wie wir es uns erwünscht haben. Daher empfiehlt es sich, an dieser Stelle auf den ELV-Segmentanzeigenrechner [2] zurückzugreifen, der es uns ermöglicht, die Bitmuster auf Grundlage eines auf dem Bildschirm dargestellten Zeichens zu generieren.

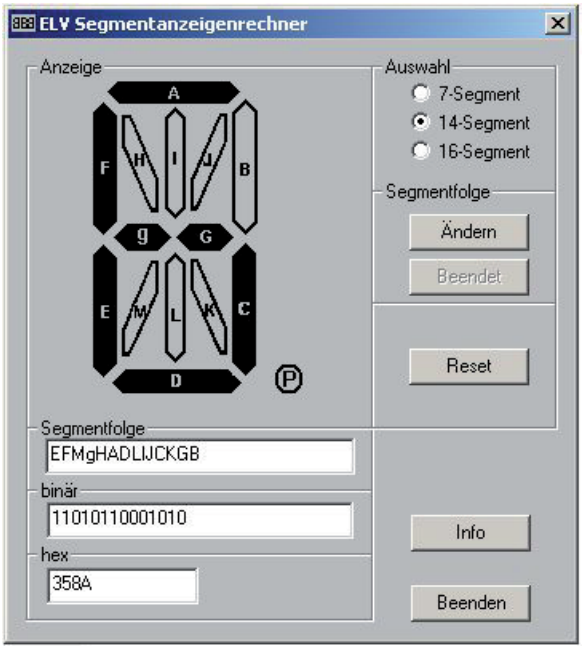


Bild 7: ELV-Segmentanzeigenrechner [2]

Nach dem Start des Segmentanzeigenrechners (Bild 7) müssen wir zunächst die Auswahl auf „14-Segment“ ändern, wodurch auch das angezeigte Zeichen sich verändert. Klicken wir nun auf die einzelnen Segmente, verfärben sich diese schwarz und im unteren Bereich des Programms wird der zu diesem Zeichen gehörende Binär- und Hexcode dargestellt. Bevor wir jedoch damit beginnen können, für alle Zeichen die entsprechenden Codes zu generieren, müssen wir zunächst die Segmentreihenfolge an die Bit-Reihenfolge in den ULP-Registern anpassen. Hierzu müssen wir als Erstes den Button „Ändern“ auf der rechten Seite des Programms anklicken. Im Anschluss wählen wir nacheinander die einzelnen Bits entsprechend ihrer Reihenfolge (beginnend beim niederwertigsten Bit). Aus der Reihenfolge im ULP-Register EFMGKADJILCNHB entsteht somit die einzustellende Segmentreihenfolge EFMgHADLIJCKGB (die Bezeichnung der einzelnen Segmente weicht leicht voneinander ab). Durch Anklicken des Buttons „Beendet“ wird die eingegebene Segmentreihenfolge übernommen, und wir können mit der Eingabe der gewünschten Zeichen beginnen. Zur Kontrolle können wir einmal die Zahl 6 einstellen, wobei wieder der von uns oben berechnete Hexcode 0x358A herauskommen muss.

Umsetzung der Zeichen in der Library

Nachdem nun alle Bitmuster der gewünschten Zeichen von uns erfasst worden sind, müssen wir diese als Array implementieren. Außerdem empfiehlt sich eine Methode, der das darzustellende Bitmuster und die Position übergeben werden kann und welche selbstständig die Umrechnung des Bitmusters für das erste Segment übernimmt.

Auf den ersten Blick erscheint es am einfachsten, die Bitmuster als ein Array mit dem Datentyp unsigned int (16 Bit) zu speichern. Ein Problem hierbei ist jedoch, dass die ULP-Register byteweise (8 Bit) geschrieben werden und die 14 Bits (entsprechend der 14 Einzelsegmente) über 3 Register verteilt liegen. Es

```
unsigned char lcdSymbols[] =
{
   0x18, 0x93,   // 0  0x3189
   0x00, 0x90,   // 1  0x0009
   0x58, 0x32,   // 2  0x2583
   0x58, 0xB0,   // 3  0x058B
   0x40, 0xB1,   // 4  0x140B
   0x58, 0xA1,   // 5  0x158A
   0x58, 0xA3,   // 6  0x358A
   0x10, 0x90,   // 7  0x0109
   0x58, 0xB3,   // 8  0x358B
   0x58, 0xB1,   // 9  0x158B
   0x50, 0xB3,   // A  0x350B
   0x1E, 0xB0,   // B  0x01EB
   0x18, 0x03,   // C  0x3180
   0x1E, 0x90,   // D  0x01E9
   0x58, 0x23,   // E  0x3582
   0x50, 0x23,   // F  0x3502
   0x18, 0xA3,   // G  0x318A
   0x40, 0xB3,   // H  0x340B
   0x06, 0x00,   // I  0x0060
   0x08, 0x92,   // J  0x2089
   0x41, 0x43,   // K  0x3414
   0x08, 0x03,   // L  0x3080
   0x21, 0x93,   // M  0x3219
   0x20, 0xD3,   // N  0x320D
   0x18, 0x93,   // O  0x3189
   0x50, 0x33,   // P  0x3503
   0x18, 0xD3,   // Q  0x318D
   0x50, 0x73,   // R  0x3507
   0x58, 0xA1,   // S  0x158A
   0x16, 0x00,   // T  0x0160
   0x08, 0x93,   // U  0x3089
   0x81, 0x03,   // V  0x3810
   0x80, 0xD3,   // W  0x380D
   0xA1, 0x40,   // X  0x0A14
   0x25, 0x00,   // Y  0x0250
   0x99, 0x00,   // Z  0x0990
   0x46, 0x20,   // +  0x0462
   0x40, 0x20,   // -  0x0402
   0x81, 0x00,   // /  0x0810
   0x20, 0x40,   // \  0x0204
   0xE7, 0x60,   // *  0x0E76
   0x00, 0x00   // Leerzeichen
};
```

Bild 8: Bitmuster für die darstellbaren Zeichen auf dem Display (lcd.h)

wäre also eine gewisse Menge an Bitoperationen und Typumwandlungen nötig, wenn man die Bitmuster so speichern würde. Betrachten wir noch einmal die Bitreihenfolge im ULP-Register, so fällt uns auf, dass das mittlere Byte einer Position komplett für die Darstellung eines Zeichens benötigt wird (Einzelsegmente MGKA DJIL). Speichern wir also das Bitmuster dieser 8 Bits direkt in einem Byte, können wir dieses Byte später einfach in das entsprechende ULP-Register kopieren, ohne eine einzige Bitoperation ausführen zu müssen (Ausnahme ist die 1. Position aufgrund ihrer spiegelverkehrten Reihenfolge). Die Speicherung der noch fehlenden 6 Bits (EF und CNHB) kann auf verschiedene Arten realisiert werden. Eine Methode ist, die Bits für die Zeichen EF und CNHB in zwei verschiedenen Bytes zu speichern. Dadurch werden für jedes darzustellende Zeichen insgesamt 3 Bytes benötigt. Vorteil dieser Möglichkeit ist, dass die gespeicherten Bytes direkt verwendet werden können, ohne dass vorher Bits innerhalb dieses Bytes ausgeblendet werden müssen. Eine andere Methode der Speicherung ist die Bits für die Zeichen EF und CNHB in einem Byte zusammenzufassen. Vorteil dieser Methode ist, dass wir für jedes darzustellende Zeichen nur 2 Bytes benötigen, jedoch müssen wir z. B. die Bits für CNHB ausblenden, wenn wir die Bits für EF in das ULP-Register schreiben wollen. Jede dieser beiden Methoden hat also ihre Vor- und Nachteile.

In der Library ist die zweite Methode innerhalb des Arrays lcdSymbols realisiert (Bild 8). Jedes Zeichen wird durch 2 Bytes dargestellt, wobei die erste Spalte die Einzelsegmente MGKA DJIL und die zweite Spalte die Segmente CNHB xxEF darstellt. Als Kommentierung sind dahinter das entsprechende Zeichen und die Bitreihenfolge in hexadezimaler Darstellung aufgeführt, wie sie vom Segmentanzeigenrechner berechnet wurden.

Um nun die einzelnen Positionen des LCDs mit den Zeichen bzw. die ULP-Register mit den entsprechenden Daten füllen zu können, empfiehlt sich die Implementierung einer Methode, welche unter Angabe des Zeichens in der Arrayliste und der Position, an dem dieses Zeichen darzustellen ist, die ULP-Register befüllt. Die Library enthält hierzu die Methode setSegment (Bild 9).

Innerhalb dieser Methode wird zunächst aus dem übergebenen Symbolindex (indexLcdSymbol) die Startposition dieses Symbols im Array lcdSymbols ermittelt. Da für jedes Symbol 2 Bytes benötigt werden, ergibt sich somit eine Multiplikation mit 2. Anschließend wird bei der weiteren Bearbeitung unterschieden, ob die 1. Position (segment == 1) oder eine der anderen mit neuen Daten befüllt werden soll. Damit die Daten des ULP-Registers im CP2401 nicht ständig ausgelesen und wieder beschrieben werden müssen, wird der 10 Byte große, globale Zwischenspeicher lcdData verwendet, sodass die Daten zunächst alle neu ermittelt werden können und anschließend von uns nur ein Schreibvorgang an das CP2401 benötigt wird (siehe „Aktualisierung des Displays“). Für die Positionen 2 bis 4 wird zunächst innerhalb der Variable temp die zu diesem Segment gehörende Position im ULP-Register bestimmt. Für das 2. Segment ergibt sich z. B. der Wert 8, entsprechend dem Register ULPMEM08. Dieses Register enthält ausschließlich Daten für das 2. Segment, sodass keine Bitoperationen nötig sind und die 8 Bits des Bitmusters für das anzuzeigende Zeichen einfach aus dem Array lcdSymbols herauskopiert werden können. Für z. B. die Zahl 6 wäre dieses das Byte 0x58. Um nun auch die Bits innerhalb des darüber- und darunterliegenden Registers schreiben zu können (Einzelsegmente EF im übergeordneten, Einzelsegmente CNHB im untergeordneten Register), werden einige Bitoperationen benötigt. Zunächst werden im untergeordneten Register die oberen 4 Bits gelöscht (hier liegen die Einzelsegmente CNHB) und anschließend die oberen 4 Bits des Bitmusters des darzustellenden Zeichens aus dem Array lcdSymbols gesetzt. Für die Zahl 6 liefert das Array lcdSymbols den Wert 0xA3 zurück, hiervon sind zunächst jedoch nur die oberen 4 Bits von Bedeutung (Segmente CNHB), daher werden zunächst mit dem &-Operator und dem Wert 0xF0 die unteren 4 Bits von 0xA3 gelöscht, sodass sich der Wert 0xA0 ergibt. Dieser wiederum wird mit dem Wert des ULP-Registers durch eine Oder-Operation verknüpft, sodass die oberen 4 Bits des Registers mit den richtigen Werten zur Darstellung einer 6 gefüllt sind. Ebenso wird mit dem übergeordneten Register vorgegangen, wobei hier die unteren 2 Bits erneuert werden. Natürlich stellt sich hier nun die Frage, warum

```
void TwoWireLCD::setSegment(unsigned char segment, unsigned
char indexLcdSymbol)
{
   // temporäre Variable mit unterschiedlicher Funktion
   // für Segment 1:
   // Zwischenspeicher für die anzuzeigenden Daten
   // für Segement 2-4:
   // Zwischenspeicher für die Indizes der neu zu
   // beschreibenden Einträge im Ausgabespeicher
   unsigned char temp;

   indexLcdSymbol *= 2;

   // Segment 1 ist ein Sonderfall, da die Register für dieses
   // Segment anders angeordnet sind als bei den Segmenten 2-4
   if(segment == 1)
   {
      temp = lcdSymbols[indexLcdSymbol];

      lcdData[2] = ((temp & 0x0F)<<4) | ((temp & 0xF0)>>4);

      lcdData[1] &= 0xCF;
      lcdData[1] |= (lcdSymbols[indexLcdSymbol+1]<<4);

      lcdData[3] &= 0xF0;
      lcdData[3] |= (lcdSymbols[indexLcdSymbol+1]>>4);
   }
   else
   {
      temp = 12 - 2 * segment;

      lcdData[temp] = lcdSymbols[indexLcdSymbol];

      lcdData[temp-1] &= 0x0F;
      lcdData[temp-1] |= lcdSymbols[indexLcdSymbol+1] & 0xF0;

      lcdData[temp+1] &= 0xFC;
      lcdData[temp+1] |= lcdSymbols[indexLcdSymbol+1] & 0x03;
   }
}
```

Bild 9: Segmentroutine setSegment

kopieren wir die Werte für diese beiden Register nicht einfach so, wie wir es beim mittleren der 3 Register eines Segmentes auch gemacht haben, sondern machen uns die Arbeit mit dem Löschen und Setzen der Bits? Diese Frage lässt sich einfach anhand eines Beispiels beantworten. Betrachten wir die Zuordnung der einzelnen Segmente im ULP-Register (Bild 6) fällt uns sofort auf, dass z. B. das Register ULPMEM05 sowohl Daten für das 3. als auch für das 4. Segment enthält. Würden wir nun das 3. Segment mit neuen Daten füllen wollen und kopieren wir einfach die Daten im Array an diese Stelle, würden die Informationen für das 4. Segment verloren gehen. Daher werden nur die Bits gelöscht bzw. gesetzt, die Informationen für das zu schreibende Segment enthalten.

Wie bereits oben beschrieben, ist das 1. Segment ein Sonderfall, da hier jeweils 4 Bits (Nibble) der ULP-Register vertauscht sind. Daher wird dieser Fall innerhalb der Methode setSegment gesondert betrachtet. Im ersten Schritt werden die Daten für das mittlere der 3 Register aus dem Array lcdSymbols gelesen. Da die beiden Nibbles des Registers vertauscht sind (MGKA DJIL gegenüber DJIL MGKA bei den anderen Segmenten), muss der aus dem Array ausgelesene Wert ebenfalls gedreht werden. Mit ((temp & 0x0F) << 4) werden die hinteren 4 Bits um 4 Positionen nach vorne und mit ((temp & 0xF0) >> 4) entsprechend die vorderen 4 Bits um 4 Positionen nach hinten verschoben. Durch die anschließende Veroderung dieser beiden Werte sind somit die Nibbles vertauscht und können an die entsprechende Position im ULP-Register geschrieben werden. Gleiches gilt für das unter- bzw. übergeordnete Register des 1. Segments. Auch hier werden die Daten aus dem Array lcdSymbols jeweils um 4 Bits nach vorne bzw. hinten geshiftet, um die vertauschten Nibbles bei diesem Segment wieder auszugleichen.

Darstellung von Texten und Zahlen mithilfe der print-Methoden

Mithilfe der Methode setSegment ist es uns nun also möglich, den einzelnen Segmenten Werte aus dem Array lcdSymbols, in dem die Bitmuster aller Zeichen gespeichert sind, zuzuweisen. Wenn wir nun aber z. B. die Zahl 123 auf dem Display darstellen möchten, müssten wir nun dreimal die Methode setSegment mit den Werten 1, 2 und 3 aufrufen. Noch unangenehmer wird die Darstellung von Texten. Um z. B. ELV auf dem Display darzustellen, müsste die Methode setSegment mit den Werten 14, 21 und 31 aufgerufen werden. Was also an dieser Stelle noch fehlt, ist zum einen eine Methode, die es ermöglicht, Zahlen darzustellen, zum anderen aber auch eine Methode, der Texte übergeben werden und die diese zur Darstellung bringt. Innerhalb der Library ist die Darstellung von Zahlen und Texten mithilfe der print-Methoden realisiert.

Die erste print-Methode dient zur Darstellung von ganzzahligen Werten (Bild 10). Hierbei wird neben dem darzustellenden Wert (value) auch übergeben, ob führende Nullen dargestellt werden sollen oder nicht (showLeadingZeros). Innerhalb der Methode wird zunächst entschieden, ob der übergebene Wert negativ ist und gegebenenfalls das Minus-Symbol eingeblendet. Anschließend wird zur einfacheren Auswertung nur mit positiven Werten weitergearbeitet. Da das Display nur 4 Zeichen darstellen kann, wird der darzustellende Wert auf 9999 begrenzt. An dieser Stelle könnten wir aber auch andere Maßnahmen treffen, um zu große bzw. zu kleine zu signalisieren (z. B. mithilfe der Pfeile T1 bis T4). Um nun die Zahl zur Darstellung bringen zu können, wird die Zahl in ihre einzelnen Stellen zerlegt, denn einzelne Werte können wir ja mit der vorher besprochenen setSegment-Methode auf dem Display anzeigen. Soll z. B. die Zahl 1234 auf dem Display erscheinen, müssen wir zunächst die Zahl 4 isolieren. Dieses geschieht durch Division und Restwertbildung. Mithilfe der Zeile div = value / 10 erhalten wir zunächst den Wert 123 für div. Die folgende Zeile rest = value – 10 * div ergibt anschließend den Wert 4 (1234 – 10 * 123 = 4). Diesen Wert schreiben wir mithilfe der setSegment-Methode nun an die 4. Position des Displays. Da die Berechnung der weiteren Stellen von value genauso abläuft, wird innerhalb einer For-Schleife Position für Position berechnet, sodass am Ende die 4 Segmente mit den Werten 1, 2, 3 und 4 gefüllt wurden. Während der For-Schleife wird außerdem kontrolliert, ob führende Nullen dargestellt werden sollen oder nicht. Ist dieses der Fall (showLeadingZeros != 0), wird die For-Schleife komplett für alle Segmente durchlaufen. Sollen führende Nullen nicht dargestellt werden (showLeadingZeros == 0), bricht die For-Schleife in dem Moment ab, in dem die Division durch 10 nur noch null ergibt (div == 0) oder alle Segmente be-

```
unsigned char TwoWireLCD::print(signed int value,
    unsigned char showLeadingZeros)
{
    char i;
    unsigned int div;
    unsigned char rest;

    // ist der Wert kleiner 0?
    // falls ja:
    // Minuszeichen darstellen und für weitere
    // Verarbeitung den absoluten Wert der Zahl
    // berechnen.
    // falls nein:
    // Minuszeichen ausschalten
    if(value < 0)
    {
        showMinus(1);
        value = -value;
    }
    else
    {
        showMinus(0);
    }

    // Wert auf 9999 begrenzen, da die Anzeige nicht mehr
    // darstellen kann.
    if(value > 9999)
        value = 9999;

    // Berechnung und Darstellung der einzelnen Stellen
    div = value / 10;
    rest = value - 10 * div;

    for(i = 4; i > 0; i--)
    {
        setSegment(i, rest);

        if((showLeadingZeros == 0) && (div == 0) || (i == 1))
            break;

        value = div;
        div = value / 10;
        rest = value - 10 * div;
    }

    // Anzahl der dargestellten Zeichen zurückgeben
    return (5-i);
}


void TwoWireLCD::print(signed int value,
    unsigned char showLeadingZeros,
    unsigned char decimalPlace)
{
    // Wert ausgeben
    unsigned char showingCharacters = print(value,
        showLeadingZeros);

    // bei Werten mit weniger Stellen als die Anzahl der
    // Nachkommastellen mit Nullen auffüllen.
    while(showingCharacters <= decimalPlace)
    {
        setSegment(4-showingCharacters, 0);
        showingCharacters++;
    }

    // Punkt darstellen
    showPoints(1<<(3-decimalPlace));
}


void TwoWireLCD::print(char* text, unsigned char count)
{
    unsigned char i;
    unsigned char index;

    // Begrenzung auf 4 Zeichen
    if(count > 4)
        count = 4;

    // Index der Zeichen innerhalb der Lookup-Table ermitteln
    // und darstellen
    for(i = 0; i < 4; i++)
    {
        if(i >= count)
            index = INDEX_SYMBOL_BLANK;
        else if(text[i] >= ‹A› && text[i] <= ‹Z›)
            index = 10 + (text[i] - ‹A›);
        else if(text[i] >= ‹a› && text[i] <= ‹z›)
            index = 10 + (text[i] - ‹a›);
        else if(text[i] == ‹+›)
            index = INDEX_SYMBOL_PLUS;
        else if(text[i] == ‹-›)
            index = INDEX_SYMBOL_MINUS;
        else if(text[i] == ‹/›)
            index = INDEX_SYMBOL_SLASH;
        else if(text[i] == 0x5C)  //»\»
            index = INDEX_SYMBOL_BACKSLASH;
        else if(text[i] == ‹*›)
            index = INDEX_SYMBOL_ASTERISK;
        else    // alles andere wird als Leerzeichen dargestellt
            index = INDEX_SYMBOL_BLANK;

        setSegment(i+1, index);
    }
}
```

Bild 10: LCD-Ausgaberoutinen print

schrieben wurden (i == 1).

Die zweite print-Methode dient zur Darstellung von Festkommawerten. Zusätzlich zur ersten print-Methode wird hier noch die Anzahl der Nachkommastellen mit angegeben (decimalPlace). Innerhalb dieser Methode werden zunächst mithilfe der ersten print-Methode die Zahlen ins Ausgaberegister geschrieben. Die print-Methode gibt uns hierbei die Anzahl der Stellen zurück, die bereits dargestellt werden. Sollte die Anzahl der führenden Nullen nicht ausreichen für die Festkommazahl, wird im nächsten Schritt die benötigte Anzahl Nullen vorangestellt. Wichtig ist dies z. B. bei der Darstellung des Wertes 0.01, welche durch den Aufruf print(1, NO_LEADING_ZEROS, 2) realisiert wird. Die erste print-Methode sorgt ausschließlich dafür, dass im 4. Segment eine 1 erscheint. Die beiden Nullen zur Darstellung der richtigen Zahl müssen also noch vorangestellt werden. Im letzten Schritt wird mithilfe der Methode showPoints der Dezimalpunkt zur Darstellung gebracht (siehe „Darstellung der sonstigen Symbole“).

Nachdem wir nun in der Lage sind, sowohl ganzzahlige als auch Festkommawerte darzustellen, fehlt uns noch die Möglichkeit, Buchstaben bzw. Texte auf dem Display darzustellen. Hierzu dient die letzte print-Methode, der ein Pointer auf die Zeichenkette (text) und die Anzahl der darzustellenden Zeichen (count) übergeben wird. Da das Display nur 4 Segmente besitzt, wird die Anzahl der darzustellenden Zeichen im ersten Schritt auf diesen Wert begrenzt. Anschließend wird wie bei der Darstellung der Zahlen jedes Zeichen einzeln mithilfe der Methode setSegment in das Ausgaberegister geschrieben. Was uns jedoch noch fehlt, ist eine Umrechnung der übergebenen ASCII-Werte der Zeichenkette in die Indizes unseres Arrays lcdSymbols, welches die Bitmuster der einzelnen Zeichen enthält. Wir könnten nun natürlich anfangen, mit unzähligen if-Abfragen jeden einzelnen Buchstaben in den entsprechenden Index umzuwandeln. Schauen wir uns jedoch einmal eine ASCII-Tabelle an, fällt uns auf, dass die Buchstaben von A bis Z alle direkt hintereinander liegen. Genauso liegen die Bitmuster dieser Buchsta-

```
void TwoWireLCD::showDegree(unsigned char visible)
{
    if(visible)
        lcdData[1] |= (1<<3);
    else
        lcdData[1] &= ~(1<<3);
}
```

Bild 11: Anzeige der einfachen Symbole am Beispiel des „°C“-Symbols

```
void TwoWireLCD::showArrows(unsigned char arrows)
{
    lcdData[0] &= 0xF0;
    lcdData[0] |= (arrows & 0x0F);
}
```

Bild 12: Anzeige der seitlichen Pfeile T1 bis T4 mithilfe der showArrows-Routine

```
void TwoWireLCD::showPoints(unsigned char points)
{
    if(points & POINT_1)
        lcdData[9] |= (1<<2);
    else
        lcdData[9] &= ~(1<<2);

    if(points & POINT_2)
        lcdData[7] |= (1<<3);
    else
        lcdData[7] &= ~(1<<3);

    if(points & POINT_3)
        lcdData[5] |= (1<<2);
    else
        lcdData[5] &= ~(1<<2);
}
```

Bild 13: Anzeige der Dezimalpunkte mithilfe der showPoints-Routine

ben in unserem Array lcdSymbols (siehe Bild 8) alle hintereinander. Nach den 10 Zahlen steht das Bitmuster für den Buchstaben „A“ an der 11. Position (Index 10). Zu diesem Index müssen wir also nur noch die Position des Buchstabens innerhalb des Alphabets hinzufügen (angefangen bei 0 für „A“). Daher wird bei jedem Buchstaben, der dargestellt werden soll, der Wert des ASCII-Zeichens „A“ abgezogen. Somit ergibt sich für die Berechnung des Index die Codezeile index = 10 + (text[i] – ‘A’). Damit wir auch Texte mit Kleinbuchstaben übergeben können, welche als Großbuchstaben dargestellt werden, wird für die Buchstaben a bis z ebenfalls diese Berechnung durchgeführt, wobei in diesem Fall der Wert des ASCII-Zeichens „a“ abgezogen wird. Für alle anderen Zeichen (+, -, /, \, *) wird der Index innerhalb des Arrays lcdSymbols mithilfe von einzelnen if-Abfragen realisiert. Alle undefinierten Zeichen werden hierbei als Leerzeichen dargestellt. Nach diesem Prinzip werden alle 4 darzustellenden Zeichen mithilfe einer For-Schleife durchlaufen und die Ausgaberegister entsprechend gefüllt.

Darstellung der sonstigen Symbole

Neben den 4 Segmenten, welche über die print-Methoden angesprochen werden, bietet das Display noch weitere Symbole (z. B. °C oder %), die eingeblendet werden können (Bild 5). In Bild 6 sieht man, wie diese Symbole auf die ULP-Register verteilt sind. So wird z. B. das „°C“-Symbol mithilfe des 3. Bits im Register ULPMEM01 angesprochen.

Innerhalb der Library werden diese Symbole mithilfe der show-Methoden ein- bzw. ausgeblendet. Für beispielsweise das „°C“-Symbol ist dieses die Methode showDegree (Bild 11), innerhalb der je nach dem übergebenen Parameter visible das Symbol ein- oder ausgeschaltet wird. Wird visible mit 0 übergeben, so wird das 3. Bit in lcdData[1] (Daten für das Register ULPMEM01) gelöscht. Entsprechend wird bei einem übergebenen Wert ungleich 0 das Bit innerhalb von lcdData[1] gesetzt.

Neben diesen einfachen Symbolen gibt es noch die 4 Pfeile T1 bis T4 (Bild 5) auf der rechten Seite und die 3 Dezimalpunkte zwischen den einzelnen Segmenten. Um die 4 Pfeile T1 bis T4 anzeigen zu können, ist die Methode showArrows in der Library enthalten (Bild 12). Der Zustand der einzelnen Pfeile wird hierbei mit dem Parameter arrows in bitcodierter Form mitgeteilt. Sollen z. B. der 2. und 4. Pfeil eingeschaltet werden, so muss der Methode der binäre Wert 0000 1010 (0x0A) übergeben werden. Um nur den 1. Pfeil darzustellen, entsprechend der binäre Wert 0000 0001 (0x01). Da die 4 Pfeile innerhalb des Registers ULPMEM00 (gleichbedeutend mit dem Zwischenspeicher lcdData[0]) wie beim übergebenen Wert innerhalb der 4 unteren Bits liegen (Bild 6), werden bei der Methode zunächst die 4 unteren Bits gelöscht und anschließend die 4 unteren Bits des bitcodierten Wertes arrows in dieses Register geschrieben.

Zu guter Letzt fehlt lediglich noch die Darstellung der 3 Dezimalpunkte zwischen den 4 Segmenten. Hierzu enthält die Library die Methode showPoints (Bild 13), der wie der Methode showArrows die Zustände der Punkte in bitcodierter Form übergeben wird. Das Bit 0 steht für den Punkt zwischen dem Segment 1 und 2 (P1), das Bit 1 für den Punkt zwischen Segment 2 und 3 (P2) usw. Damit z. B. die Punkte P1 und P2 angezeigt werden, muss der Methode der binäre Wert 0000 0011 (0x03) übergeben werden. Da die 3 Punkte innerhalb der ULP-Register verteilt liegen (Bild 6), wird innerhalb der Methode jeder Punkt einzeln geprüft und das entsprechende Bit innerhalb des Zwischenspeichers lcdData gesetzt.

Aktualisierung des Displays

Bisher wurde mit den Methoden show und print lediglich dafür gesorgt, dass der interne Zwischenspeicher lcdData mit den entsprechenden Werten gefüllt wurde. Jedoch wurde bisher das lcdData nicht in die ULP-Register des CP2401 geschrieben, sodass die Daten noch gar nicht auf dem Display dargestellt werden konnten. Hier kommt die Methode update ins Spiel (Bild 14). Hierin werden mithilfe der Methode writeRegister die 10 Bytes innerhalb des Arrays lcdData an den CP2401 gesendet und in den ULP-Registern gespeichert.

Warum aber aktualisieren wir die Register nicht sofort bei jeder Änderung eines Zeichens? Grundsätzlich wäre dieses möglich, jedoch würde dieses zum einen

```
void TwoWireLCD::update(void)
{
   writeRegister(I2C_LCD_ADDRESS, REGWRITE,
      CP240X_ULPMEM00, lcdData, 10);
}
```

Bild 14: Schreiben der ULP-Register mithilfe der update-Routine

```
void TwoWireLCD::setLeds(unsigned char ledOutput)
{
   // obere 4 Bits immer high, da diese als
   // Eingang genutzt werden
   ledOutput = ledOutput | 0xF0;

   writeRegister(I2C_LCD_ADDRESS, REGWRITE,
      CP240X_P3OUT, &ledOutput, 1);
}
```

Bild 15: Ansteuerung der LEDs D1 bis D4

den Quellcode vergrößern, da bei jeder Methode, die eine Veränderung der Anzeigedaten zur Folge hat, die Daten an den CP2401 gesendet werden würden, zum anderen hat es aber auch einen visuellen Nachteil, da so alle Veränderungen nicht in einem Rutsch angezeigt werden können. Ändert man z. B. die Anzeige von % auf V und den dargestellten Wert auf 1.23, dann benötigt dieses den Aufruf von insgesamt 3 Methoden (showPercent, showVolt und print). Würde man in jeder dieser Methoden das Display direkt aktualisieren, entstünden hierbei Zustände, die eigentlich nicht gewollt sind, auch wenn diese vom menschlichen Auge eventuell nicht erfasst werden können. Nach Aufruf der Methode showPercent(0) würde das %-Symbol verschwinden, danach mit showVolt (1) das V-Symbol erscheinen und anschließend mit print(123, NO_LEADING_ZEROS, 2) der Wert 1.23 dargestellt werden. Effektiver ist es daher, zunächst alle geänderten Zeichen innerhalb des Arrays lcdData zu erfassen und anschließend diese Daten in einem Rutsch an den CP2401 zu senden.

Ansteuerung der LEDs D1 bis D4

Neben einem LCD bietet uns das I²C-Displaymodul außerdem die Möglichkeit, insgesamt 6 LEDs anzusteuern. Alle LEDs können hierbei direkt über die Stiftleiste ein- bzw. angesteuert werden (Bild 1). Zusätzlich ist es möglich, die LEDs D1 bis D4 über den Displaytreiber CP2401 anzusteuern. Hierzu müssen jedoch die Lötbrücken J1 bis J4 geschlossen werden. Die Kathode der LEDs wird dadurch mit den unteren 4 Pins des Port 3 verbunden. Aufgrund dieser Anordnung sind die LEDs Low active, was heißt, dass die entsprechende LED leuchtet, wenn der CP2401 ein Low-Signal (GND) auf diesen Pins ausgibt. Gibt der CP2401 entsprechend ein High-Signal (+3V) aus, bleiben die LEDs aus.

Innerhalb der Library lassen sich die LEDs D1 bis D4 mithilfe der Methode setLeds ansteuern (Bild 15). Wie bereits bei den Methoden showArrows und showPoints verwendet, wird auch der Zustand der LEDs in bitcodierter Form an die Methode übergeben. Da nur die unteren 4 Bits für die LEDs genutzt werden, werden die oberen 4 Bits grundsätzlich auf High gelegt. Der Grund hierfür ist, dass die oberen 4 Bits des Port 3 als Eingänge zur Auswertung der Taster (siehe „Tasterauswertung“) genutzt werden können. Anschließend wird mithilfe der Methode writeRegister der neue Zustand der LEDs an das Register P3OUT des CP2401 geschrieben, welches für die Ausgangszustände des Port 3 zuständig ist.

Tasterauswertung

Zusätzlich zu den 4 LEDs hängen am Port 3 des LCD-Treibers noch 4 Taster. Die Frage, die sich uns nun stellt, ist, wie wir einen Tastendruck über den CP2401 erfassen. Der erste Gedanke wäre, das Register mit den entsprechenden Tasterzuständen kontinuierlich mit kurzem Zeitabstand abzufragen. Dieses wäre natürlich sehr unschön, da wir in den allermeisten Fällen als Antwort bekämen, dass kein Taster gedrückt ist. Außerdem würden wir den I²C-Bus mit vielen unnötigen Kommunikationsvorgängen belasten. Ein Blick ins Datenblatt des CP2401 [1] verrät uns in Kapitel 7, dass es einen Port-Match-Interrupt gibt. Dieser Interrupt löst bei Aktivierung aus, wenn die Pegel an den entsprechenden Ports sich von einem innerhalb der Register gesetzten Wert unterscheiden, und zieht den /INT-Pin des CP2401 auf GND runter. Somit besteht die Möglichkeit, dass wir nicht zyklisch nachschauen müssen, wie nun der Zustand der Taster ist, sondern der CP2401 löst einen Interrupt aus und gibt uns damit die Meldung, dass ein Taster gedrückt wurde. Nach dem der Interrupt eingegangen ist, müssen wir also nur noch einmal den aktuellen Zustand der Taster abfragen.

In der Library ist dieser Interruptbetrieb innerhalb der Methode enableKeyInterrupt realisiert (Bild 16). Darin wird zunächst festgelegt, für welche Portpins dieser Port-Match-Interrupt ausgeführt werden soll (Maske). Da die Taster an den oberen 4 Pins des Port 3 hängen, wird die Maske (P3MSK) mit 0xF0 initialisiert. Im nächsten Schritt müssen wir den Vergleichswert festlegen, mit dem die Kontrolle stattfindet bzw. der den Normalzustand darstellt. Da der Port 3 während der Initialisierung als Open Drain parametriert wurde, liegt intern an den Pins ein High-Pegel an, wenn kein Taster gedrückt wird. Somit müssen wir den Vergleichswert für diese 4 Pins auf High (1) setzen. Dieses geschieht durch Schreiben des Registers P3MATCH mit dem Wert 0xF0. Nachdem nun die Einstellungen für den Port-Match gemacht wurden, wird im letzten Schritt der entsprechende Interrupt im Register INT0EN aktiviert.

Neben der Aktivierung des Interrupts bietet die Library durch die Methode disableKeyInterrupt (Bild 16) auch dessen Abschaltung. Hierbei wird einfach das entsprechende Bit innerhalb des Registers INT0EN wieder auf 0 gesetzt, wodurch keine weiteren Interrupts ausgelöst werden.

Mithilfe des Interrupts wird uns bisher jedoch nur mitgeteilt, dass ein Tastendruck stattgefunden hat, jedoch wissen wir noch nicht, welcher Taster genau gedrückt wurde. Dazu ist innerhalb der Library die Methode readKeys implementiert (Bild 16). Diese liest das Register P3IN des CP2401 aus, in dem der aktuelle Zustand des Port 3 vorhanden ist. Aufgrund dieser

```
void TwoWireLCD::enableKeyInterrupt(void)
{
   unsigned char registerValue;

   // Maske für P3 festlegen
   registerValue = 0xF0;
   writeRegister(I2C_LCD_ADDRESS, REGWRITE,
      CP240X_P3MSK, &registerValue, 1);

   // Vergleichswerte für P3 festlegen
   registerValue = 0xF0;
   writeRegister(I2C_LCD_ADDRESS, REGWRITE,
      CP240X_P3MATCH, &registerValue, 1);

   // Match Interrupt einschalten
   registerValue = 0x01;
   writeRegister(I2C_LCD_ADDRESS, REGWRITE,
      CP240X_INT0EN, &registerValue, 1);
}


void TwoWireLCD::disableKeyInterrupt(void)
{
   unsigned char registerValue;

   // Match Interrupt ausschalten
   registerValue = 0x00;
   writeRegister(I2C_LCD_ADDRESS, REGWRITE,
      CP240X_INT0EN, &registerValue, 1);
}


void TwoWireLCD::resetKeyInterrupt(void)
{
   unsigned char int0;

   readRegister(I2C_LCD_ADDRESS, REGREAD,
      CP240X_INT0, 1, &int0);
}


unsigned char TwoWireLCD::readKeys(void)
{
   unsigned char keys;

   readRegister(I2C_LCD_ADDRESS, REGREAD,
      CP240X_P3IN, 1, &keys);

   return keys;
}
```

Bild 16: Routinen zur Auswertung der Tasten TA1 bis TA4

```
/************************************************
Beispiel: simpleVoltmeter

Beschreibung: Dieses Beispiel zeigt die Anzeige einer Zahl auf
dem Display des I²C-LCD-Modul anhand eines einfachen Volt-
meters.

Hinweis: Es wird die Spannung am analogen Eingang A2
gemessen.
************************************************/
#include «Wire.h»
#include «TwoWireLCD.h»

/************************************************
globale Variablen
************************************************/
// Pin an dem die Spannung gemessen wird
unsigned char sensorPin = A2;

// eingelesener Sensorwert
unsigned long sensorValue = 0;

/************************************************
Funktionen
************************************************/
void setup()
{
   // Initialisierung des I²C und des I²C-LCD-Moduls
   Wire.begin();
   LCD.begin();

   // das Volt-Symbol auf dem Display anzeigen
   LCD.showVolt(1);
}

void loop()
{
   // die Spannung am analogen Eingang A2 messen
   sensorValue = analogRead(sensorPin);
   sensorValue = sensorValue * 500 / 1024;

   // die gemessene Spannung mit 2 Nachkommastellen auf
   // dem Display anzeigen
   LCD.print((int)sensorValue, NO_LEADING_ZEROS, 2);
   LCD.update();

   delay(500);
}
```

Bild 17: Beispiel „simpleVoltmeter“

Tatsache muss nach einem Interrupt die Abfrage der Tasterzustände zeitnah erfolgen.

Würden wir nun mit den beiden Methoden enableKeyInterrupt und readKeys eine Tasterauswertung starten, würde es uns passieren, dass immer nur der allererste Tastendruck detektiert und an den ATMega gemeldet wird. Danach erfolgt kein weiterer Interrupt. Dieses liegt daran, dass der CP2401 seinen Interrupt nicht selbstständig zurücksetzt, sondern dieses von außen geschehen muss. Das Datenblatt des CP2401 [1] liefert uns in Kapitel 7 dazu die Information, dass zum Zurücksetzen der Interrupts einfach das entsprechende INT-Register ausgelesen werden muss. Dieses ist in der Library innerhalb der Methode resetKeyInterrupt realisiert (Bild 16), in der das Register INT0 ausgelesen wird. Der ausgelesene Wert ist dabei ohne Bedeutung. Nachdem ein Interrupt ausgelöst wurde, wird dieser also wieder durch den Aufruf der Methode resetKeyInterrupt „scharf geschaltet“.

Beispiele

Um die Funktionalität der Library TwoWireLCD zu verdeutlichen, liegen dieser insgesamt vier einfache Beispiele bei.

Das Beispiel simpleVoltmeter (Bild 17) zeigt die Darstellung einer Zahl auf dem Display und das Zuschalten eines außen liegenden Symbols anhand eines einfachen Voltmeters. Für die Messung der Spannung wird hierbei der analoge Eingang A2 des Arduino-Boards verwendet. Bevor zyklisch mit der Messung und Darstellung der Spannung begonnen werden kann, werden zunächst der I²C-Bus (Wire.begin()) und das I²C-LCD (LCD.begin) initialisiert (setup()). Außerdem wird mithilfe der Methode LCD.showVoltage(1) das Volt-Symbol im Zwischenspeicher aktiviert (die Darstellung auf dem Display erfolgt mit dem ersten Aufruf der Methode LCD.update()). Innerhalb der Dauerschleife loop() wird nun zyklisch der AD-Wert des Pins A2 eingelesen und in eine Spannung umgerechnet. Hierbei wird mit Festkommazahlen mit 2 Nachkommastellen gearbeitet, sodass durch den Aufruf der Methode LCD.print der Spannungswert und der Angabe der beiden Nachkommastellen auf dem Display ein Wert zwischen 0.00 und 4.99 erscheint. Abschließend werden der berechnete Spannungswert und das Volt-Symbol mithilfe der Methode LCD.update() zur Darstellung gebracht. Danach folgt eine Pause von 500 ms, bis die Messung und Ausgabe wiederholt wird.

Im zweiten Beispiel runningText (Bild 18) wird die Darstellung eines Textes anhand einer Laufschrift verdeutlicht. Innerhalb des Arrays text sind dabei alle darstellbaren Zeichen der Library aufgeführt. Wie beim Beispiel simpleVoltmeter werden auch hier zunächst in der setup-Methode der I²C-Bus und das I²C-LCD initialisiert. Anschließend erfolgt in der Dauerschleife die Ausgabe von jeweils 4 Zeichen mithilfe der print-Methode. Der Parameter pos gibt dabei die aktuelle Position innerhalb des Arrays an und wird in jedem Durchlauf um eins erhöht, sodass eine Laufschrift entsteht. Bei Erreichen der letzten Position innerhalb der Zeichenkette wird wiederum an den Anfang zurückgesprungen. Die Länge der Zeichenkette wird hierbei durch die Berechnung sizeof(text)/sizeof(text[0]) realisiert. Dadurch kann der darzustellende Text einfach verändert werden, ohne an dieser Stelle eine Anpassung durchführen zu müssen. Für Texte, die weniger als 5 Zeichen enthalten, müssen die Zeilen unterhalb des delay(500) auskommentiert werden, da für diesen Fall eine Laufschrift nicht benötigt wird und die Auswertung einen Fehler verursachen würde.

Das dritte Beispiel runningLED (Bild 19) zeigt die Verwendung der LEDs D1 bis D4 mithilfe der Library anhand eines Lauflichtes. Die setup-Methode enthält wie bei den vorangegangenen Beispielen wiederum die Initialisierung des I²C-Busses und des I²C-LCDs. Innerhalb der Dauerschleife erfolgt zunächst die Ausgabe des aktuellen Zustands. Da die LEDs wie bereits oben beschrieben Low active sind, werden die einzelnen Bits der Variable leds mithilfe der vorangestellten Tilde (~) invertiert (aus dem binären Wert 0000 0001 wird somit 1111 1110). Anschließend wird das jeweils gesetzte Bit innerhalb von leds um eine Position nach rechts bzw. links geshiftet und bei Erreichen der äußeren LEDs die Richtung umgekehrt, sodass ein hin- und herlaufendes Lauflicht entsteht.

Das letzte Beispiel simpleClock (Bild 20) zeigt die Verwendung der Tasten des I²C-Displaymoduls anhand einer einfachen Uhr, welche nach dem Start eingestellt werden kann. Um das Beispiel verwenden zu können, wird für die externen Interrupts eine weitere Library benötigt. Im Beispiel wird hierbei die Library PinChangeInt verwendet, welche aus dem Internet heruntergeladen werden kann (Link steht innerhalb des Beispiels) und im Library-Ordner der Arduino-Entwicklungsumgebung zu entpacken ist. Die Bedienung des Beispiels ist folgendermaßen realisiert: Nachdem die Schaltung mit Spannung versorgt wurde, erscheint auf dem Display der Text ST und die Zahl 0. Mithilfe der Tasten 1 und 2 kann dieser Wert auf die aktuelle Stunde eingestellt und mit der 3. Taste bestätigt werden. Es folgt die Anzeige MI und 0, bei der die aktuellen Minuten entsprechend der Eingabe der Stunden einzustellen sind. Nach Bestätigung des Wertes wird die aktuelle Uhrzeit angezeigt und auch weitergezählt. An dieser Stelle sei darauf hingewiesen, dass dieses Beispiel im Wesentlichen die Tasterauswertung des I²C-Displaymoduls mithilfe der Library verdeutlichen soll. Das Fortlaufen der Uhr ist mit einem einfachen Aufruf der Methode delay realisiert worden, was zum einen sehr ungenau ist und zum anderen auch den Programmfluss blockiert, da währenddessen keine anderweitige Berechnung stattfinden kann. Die Verwendung eines Timers ist an dieser Stelle auf jeden Fall einem Aufruf von delay vorzuziehen, hätte an dieser Stelle jedoch das Programm vergrößert und dadurch unübersichtlicher gemacht, sodass die Verwendung der Tasterauswertung untergegangen wäre. Innerhalb des Programmcodes wird in der setup-Routine zunächst der Interrupt-Pin des CP2401 (Pin 15 am Arduino-Board) als Eingang definiert und mithilfe der Methode PCattachInterrupt dafür gesorgt, dass bei einer fallenden Flanke an diesem Pin die Funktion keyPress angesprungen wird. Innerhalb der Funktion keyPress setzen wir uns einfach ein Flag (keyPressDetected) und verlassen die Methode sofort wieder. Hintergrund ist, dass diese Methode im Hintergrund von einer Interrupt-Service-Routine aufgerufen wird. Das bedeutet, dass während der Ausführung der keyPress-Methode keine weiteren Interrupts abgearbeitet werden können. Daher ist es wichtig, dass wir Funktionen, die aufgrund von Interrupts aufgerufen werden, schnell wieder verlassen. Wir merken uns also einfach nur, dass ein Interrupt stattgefunden hat, und arbeiten dieses im normalen Programmfluss ab. Innerhalb der setup-Routine werden nun wie bereits aus den anderen Beispielen bekannt der I²C-Bus und das I²C-LCD initialisiert. Außerdem wird der Interrupt zur Signalisierung eines Tastendrucks eingeschaltet (resetKeyInterrupt und enableKeyInterrupt). An-

```
/************************************************
Beispiel: runningText

Beschreibung: Dieses Beispiel zeigt die Anzeige eines Textes
auf dem Display des I²C-LCD-Modul anhand eines Lauftextes
************************************************/
#include «Wire.h»
#include «TwoWireLCD.h»

/************************************************
globale Variablen
************************************************/
// darzustellender Text
char text[] =»ABCDEFGHIJKLMNOPQRSTUVWXYZ+-*/\\»;

// aktuelle Position im Text
unsigned char pos = 0;

/************************************************
Funktionen
************************************************/
void setup()
{
   // Initialisierung des I²C und des I²C-LCD-Moduls
   Wire.begin();
   LCD.begin();
}

void loop()
{
   // Text auf dem Display ausgeben
   LCD.print(&text[pos], 4);
   LCD.update();

   delay(500);

   // eine Position weiterspringen und am Ende der
   // Zeichenkette wieder an den Anfang springen
   pos++;

   if(pos >= (sizeof(text)/sizeof(text[0])) - 4)
      pos = 0;
}
```

Bild 18: Beispiel „runningText“

```
/************************************************
Beispiel: runningLED

Beschreibung: Dieses Beispiel zeigt die Ansteuerung der LEDs auf
dem I²C-LCD-Modul anhand eines Lauflichts
************************************************/
#include «Wire.h»
#include «TwoWireLCD.h»

/************************************************
globale Variablen
************************************************/
// eingeschaltete LEDs
unsigned char leds = 0x01;

// Richtung des Lauflichts
unsigned char dir = 1;

/************************************************
Funktionen
************************************************/
void setup()
{
   // Initialisierung des I²C und des I²C-LCD-Moduls
   Wire.begin();
   LCD.begin();
}

void loop()
{
   // aktuelle LED einschalten (negative Logik: LED leuchtet
   // bei einer 0)
   LCD.setLeds(~leds);

   delay(100);

   // die eingeschaltete LED um eine Position nach links bzw.
   // rechts verschieben.
   if(dir)
   {
      leds = leds << 1;

      if(leds == LED_4)
         dir = 0;
   }
   else
   {
      leds = leds >> 1;

      if(leds == LED_1)
         dir = 1;
   }
}
```

Bild 19: Beispiel „runningLED“

schließend wird innerhalb einer Schleife die Tastenauswertung durchgeführt, bis die Stunden und Minuten eingestellt wurden. Zunächst wird daher geprüft, ob überhaupt ein Tastendruck stattgefunden hat (keyPressDetected). Falls ja, wird der aktuelle Zustand der Tasten abgefragt und der Interrupt wieder zurückgesetzt. Falls kein Interrupt stattgefunden hat, wird die Variable keys mit dem aktuellen Zustand der Tasten zurückgesetzt. Danach wird geprüft, ob die einzelnen Tasten gedrückt wurden. Dazu wird geschaut, ob die Variable keys z. B. an der Bitposition des 1. Tasters eine Null stehen hat, was gleichbedeutend mit einer gedrückten Taste 1 ist. Je nach dem aktuell einzustellenden Wert wird so mit den Tasten 1 und 2 die Stunde oder Minute um eins erhöht oder vermindert. Durch Drücken der Taste 3 wird der Modus um eins vermindert, sodass nach Eingabe der Stunden (actMode == SETTING_HOUR) die Eingabe der Minuten folgt (actMode = SETTING_MINUTE) und danach in den laufenden Modus (actModus == CLOCK_RUN) gewechselt wird. Zur Darstellung der aktuellen Werte je nach Modus werden wie bei den anderen Beispielen auch die print-Methoden verwendet. Nachdem in den laufenden Modus gewechselt wurde, wird die Schleife while(actModus) verlassen und das Programm in der Dauerschleife loop() weitergeführt. Hierin wird nun lediglich noch die Uhrzeit dargestellt und jede Sekunde um eins erhöht, wobei ein Überlauf der Sekunden, Minuten und Stunden berücksichtigt wird.

Verwendung der Library ohne Arduino-Board

Die implementierte Library TwoWireLCD ist in erster Linie an Nutzer des Arduino-Boards gerichtet, jedoch kann sie auch mit wenigen Änderungen für andere Umgebungen verwendet werden. Da auf die Wire-Library der Arduino-Entwicklungsumgebung zurückgegriffen wurde, muss die Verwendung dieser Bibliothek in diesem Fall durch die eigenen I²C-Routinen ersetzt werden. Dieses betrifft die beiden Methoden writeRegister und readRegister innerhalb der TwoWireLCD-Library, die dementsprechend angepasst werden müssen. Alle anderen Methoden der Bibliothek sollten ohne Probleme übernommen werden können, wobei natürlich je nach Compiler auch an diesen Stellen eventuell Änderungen nötig sind. Hierzu ist jedoch die Beschreibung des jeweiligen Compilers zurate zu ziehen. Angemerkt sei jedoch noch, dass die Library als C++-Klasse implementiert wurde. Wer also mit einem reinen C-Compiler arbeitet, muss die klassenspezifische Verwaltung der Library entfernen. ELV

Weitere Infos:

[1] Datenblatt CP2401, Webcode #1175
[2] Download ELV-Segmentanzeigenrechner: Webcode #1177

```
/*****************************************************************
Beispiel: simpleClock

Beschreibung: Dieses Beispiel soll anhand einer Uhr die Auswertung der Tasten auf dem
I²C-LCD-Modul verdeutlichen. Nach dem Einschalten wird ist zunächst die Uhrzeit einzu-
stellen. Im Display erscheint die Anzeige «ST 0» für die Einstellung der Stunden. Mit Hilfe
der Tasten 1 und 2 kann der Wert verändert werden. Durch Betätigen der Taste 3 wird zur
Einstellung der Minuten gewechselt. Nachdem wiederum mit den Tasten 1 und 2 der Wert
eingestellt wurde, beginnt die Uhr nach Drücken der Taste 3 zu laufen.

Hinweis: Für die Auswertung des externen Interrupts wird die Library «PinChangeInt» ver-
wendet. Diese ist im Internet auf folgender Seite zu finden:
http://www.arduino.cc/playground/Main/PinChangeInt

Achtung: Dieses Beispiel soll im Wesentlichen die Tastenauswertung über das I²C-LCD-Modul
verdeutlichen. Für das Sekundensignal der Uhr wird die delay-Funktion verwendet. Dieses ist
zum einen sehr ungenau und zum anderen sehr ineffizient, jedoch hätte eine Verwendung
der Timer das Beispiel unnötig vergrößert.
*****************************************************************/

#include «Wire.h»
#include «TwoWireLCD.h»
#include «PinChangeInt.h»

/*****************************************************************
globale Variablen
*****************************************************************/
// Texte während der Zeiteinstellung (ST: Stunde, MI: Minute)
char text[] = «STMI»;

// Uhrzeit
unsigned char second = 0;
unsigned char minute = 0;
unsigned char hour = 0;

// aktueller Modus der Anzeige
enum modes
{
   CLOCK_RUN = 0,
   SETTING_MINUTE,
   SETTING_HOUR
};
unsigned char actMode = SETTING_HOUR;

// Zustand der Tasten
unsigned char keys;

// Anzeige, ob eine Taste gedrückt wurde
volatile unsigned char keyPressDetected = 0;

/*****************************************************************
Funktionen
*****************************************************************/
// die Funktion «keyPress» wird ausgeführt sobald ein
// Tastendruck detektiert wurde (externer Interrupt)
void keyPress(void)
{
   keyPressDetected = 1;
}

void setup()
{
   // der Interrupt-Pin des I²C-LCD-Moduls liegt auf dem Pin
   // D15 des Arduino-Boards und wird daher als Eingang
   // initialisiert. Bei einer fallenden Flanke an diesem Pin
   // wird die Funktion «keyPress» ausgeführt.
   pinMode(15, INPUT);
   PCattachInterrupt(15, keyPress, FALLING);

   // Initialisierung des I²C und des I²C-LCD-Moduls
   Wire.begin();
   LCD.begin();

   // den Interrupt des I²C-LCD-Moduls zurücksetzen und
   // anschließend aktiveren.
   LCD.resetKeyInterrupt();
   LCD.enableKeyInterrupt();

   // Dauerschleife bis die Stunden und Minuten eingestellt
   // wurden
   while(actMode)
   {
      // Auswertung der Tasten
      if(keyPressDetected)
      {
         keys = LCD.readKeys();

         LCD.resetKeyInterrupt();
         keyPressDetected = 0;
      }
      else
      {
         keys = NO_KEY;
      }

      // Taste 1 wurde gedrückt:
      // je nach aktuellem Zustand die Stunden bzw. Minuten
      // hochzählen
      if((keys & KEY_1) == 0)
      {
         if(actMode == SETTING_HOUR)
         {
            if(hour < 23)
               hour++;
         }
         else if(actMode == SETTING_MINUTE)
         {
            if(minute < 59)
               minute++;
         }
      }

      // Taste 2 wurde gedrückt:
      // je nach aktuellem Zustand die Stunden bzw. Minuten
      // herunterzählen
      if((keys & KEY_2) == 0)
      {
         if(actMode == 2)
         {
            if(hour > 0)
               hour--;
         }
         else if(actMode == SETTING_HOUR)
         {
            if(minute > 0)
               minute--;
         }
      }

      // Taste 3 wurde gedrückt:
      // in den nächsten Zustand wechseln
      if((keys & KEY_3) == 0)
         --actMode;

      // Ausgabe zyklisch aktualisieren
      if(actMode == SETTING_HOUR)
      {
         LCD.print(&text[0], 2);
         LCD.print(hour, NO_LEADING_ZEROS);
         LCD.update();
      }
      else if(actMode == SETTING_MINUTE)
      {
         LCD.print(&text[2], 2);
         LCD.print(minute, NO_LEADING_ZEROS);
         LCD.update();
      }
   }
}

void loop()
{
   // Stunden und Minuten anzeigen (Anzeige der Sekunde durch
   // Blinken des Doppelpunkts)
   LCD.clearSegments();
   LCD.print(hour*100 + minute, LEADING_ZEROS);
   LCD.showColon(second & 0x01);
   LCD.update();

   // eine Sekunde warten (!!! siehe Beschreibung oben !!!)
   delay(1000);

   second++;

   // Überläufe der Uhrzeit auswerten
   if(second > 59)
   {
      second = 0;
      minute++;

      if(minute > 59)
      {
         minute = 0;
         hour++;

         if(hour > 23)
            hour = 0;
      }
   }
}
```

Bild 20: Beispiel „simpleClock“