P-Seminar

Schiller-Gymnasium Hof Schuljahr 2011-12 Version 2011-12-05

Manuel Friedrich © 2011

Nachbau einer Word-Clock – Kommentare zum Quellcode von Martin Steppuhns Pong-Word-Clock

1. Vorbemerkung

Im Rahmen des P-Seminars bauen wir eine Word-Clock, also eine Uhr, die die aktuelle Uhrzeit durch Ausgabe von Text auf einer Anzeige mitteilt. Viele verschiedene Nachbauten der QLOCKTWO. Die Anzeige wird von einem Mikrocontroller geregelt, einem Atmeg8 der Firma Atmel.

Von der Fa. Franzis-Verlag bzw. der Fa. Conrad Elektronik gibt es einen Bausatz für ein Retro Pong-Spiel. Die Platine, die mit dem Bausatz ausgeliefert wird enthält 120 LEDs in einer Matrix angeordnet und neben einigen anderen Bauteilen den Atmega8.

Um die Programmierung zu verstehen und Änderungen vornehmen zu können, analysieren wir den Quellcode eines im Internet frei zugänglichen Programms von Martin Steppuhn, mit dem die Wordclock-Anzeige auf dieser Platine realisiert wurde. Auf der Platine muss neben der Stromversorgung lediglich ein Uhrenquarz und ein Taster angelötet werden.

Der sich in der Datei main.c befindliche Quellcode soll in diesem Paper sehr ausführlich für Schülerinnen und Schüler und Lehrer, die von Mikrocontroller-Programmierung keinerlei Ahnung haben, erklärt werden. Schon an dieser Stelle möchte ich betonen, dass ich von der Programmierung eines Mikrocontrollers leider auch nicht viel Ahnung besitze und mir all mein dürftiges Wissen gerade erst aus dem Internet zusammengesucht habe!

Wir verfügen über Kenntnisse in der Programmiersprache Java, sind als mit C-ähnlichen Sprachen insoweit vertraut, dass wir Variablen, Arrays, Wertzuweisungen und einfache Kontrollstrukturen gut kennen. Diese muss ich hier nicht wiederholen.

2. Lizenzhinweis

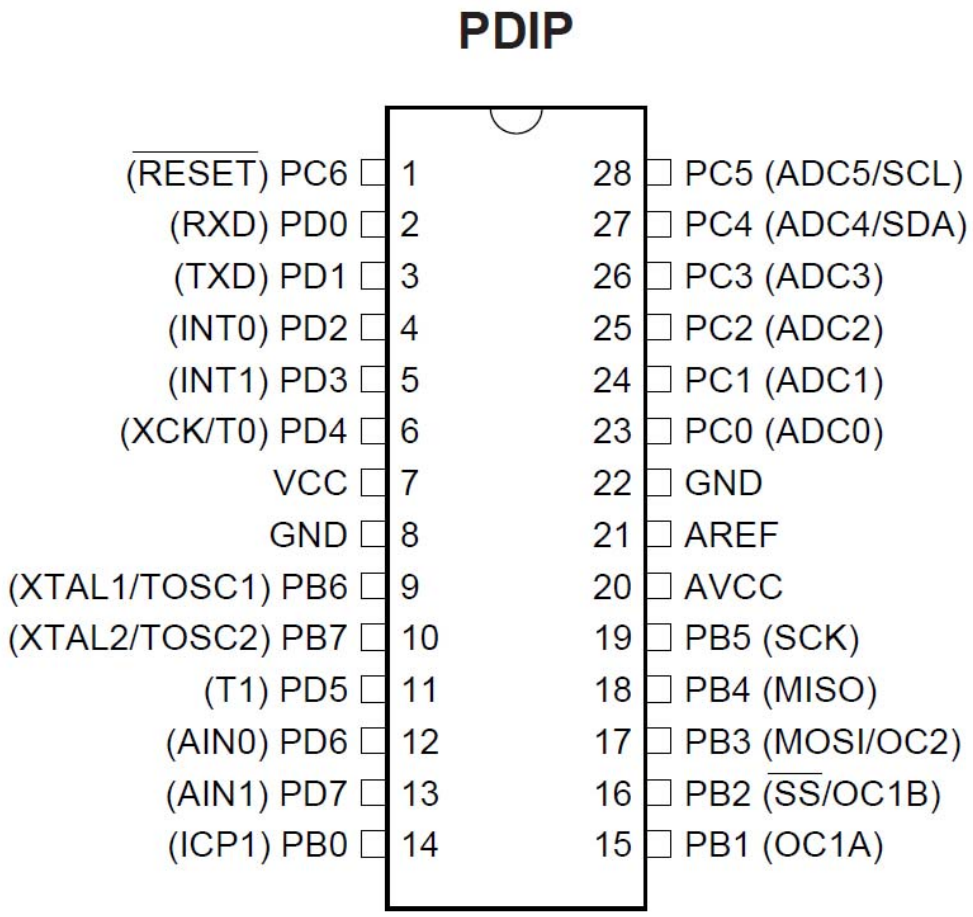
Das Programm darf frei verwendet werden, wenn der folgende Copyright-Hinweis mit weitergegeben wird, was wir hiermit tun:

```
//------------------------------------------------------------------------------
//                                    _           _
//                                   | |         | |
//      ___ _ __ ___  ___ _   _ ___| |_ ___  ___| |__
//     / _ \ '_ ` _ \/ __| | | / __| __/ _ \/ __| '_ \.
//    |  __/ | | | | \__ \ |_| \__ \ ||  __/ (__| | | |
//     \___|_| |_| |_|___/\__, |___/\__\___|\___|_| |_|
//                         __/ |
//                        |___/    Engineering
//
// Filename:    main.c
// Description: LED-Paint
//
//
// Copyright (c) Martin Steppuhn (www.emsystech.de)
//
// Nur für den privaten Gebrauch / NON-COMMERCIAL USE ONLY
//
// Die Nutzung (auch auszugsweise) ist für den privaten und nichtkommerziellen
// Gebrauch erlaubt. Eine Veröffentlichung und Weiterverwendung des Quellcodes
// ist möglich wenn diese Nutzungsbedingungen incl. Copyright beiliegen
// und die Quelle verlinkt ist. (www.emsystech.de)
//
// Bei kommerziellen Absichten nehmen Sie bitte Kontakt mit uns auf
// (info@emsystech.de)
//
// Keine Gewähr auf Fehlerfreiheit, Vollständigkeit oder Funktion. Benutzung
// auf eigene Gefahr. Es wird keinerlei Haftung für direkte oder indirekte
// Personen- oder Sachschäden übernommen.
//
// Author:      Martin Steppuhn
// History:     18.12.2009 Initial version
```

3. Was ist ein Mikrocontroller

Zunächst einmal programmieren wir einen Computerchip, an diesen sind keine Tastatur, kein Bildschirm usw. angeschlossen. Der Chip hat einfach nur 28 (bzw. 32 Beinchen), die zum Einlesen und Ausgeben von Informationen dienen. Dabei wird das Beinchen (auch Pin genannt) mit 5 Volt Strom versorgt (Ausgang) oder vom Mikrocontroller gelesen, ob an einem Pin Strom anliegt (Eingang).

Wir verwenden den Atmega8 und programmieren in C. Ein Blick in das Datenblatt des Atmega8 zeigt uns, welches Beinchen für welche Aufgabe steht.



Wir sehen, dass die Pins durchnummeriert sind von 1 bis 28. Zunächst erkennen wir, dass verschiedene Beinchen zu Ports zusammengefasst werden: Pin 1 und die Pins 23 bis Pin 28 bilden PORT-C, die Pins 2 bis 6 und Pin 11 bis Pin 13 bilden den Port-D und die Pins 14 bis 19 mit Pin 9 und 10 den PORT-B. Ports dienen zur Ausgabe und Eingabe von Informationen. Sie können aber weitere Funktionen übernehmen. Dazu später mehr.

4. Der Quellcode

Nun werden wir den Quellcode Zeile für Zeile durchgehen und kommentieren. Wir halten uns fast an die Reihenfolge, aber nicht immer.

4.1 #define-Anweisungen

In der Programmiersprache C kann man bestimmten Befehlen einen eigenen Namen geben, dies erfolgt am Anfang mit der Direktive `#define`.

Die folgenden Anweisungen dienen dazu, die Schieberegister anzusteuern (siehe Schaltplan). Ein Schieberegister hat drei Anschlüsse. Die drei Anschüsse erhalten Strom und werden wieder vom Strom getrennt. Wir brauchen also sechs Befehle, um bei den drei Pins 5 Volt Spannung anzulegen und wieder wegzunehmen.

```
#define          DATA_SET    PORTB |=  (1<<4)
#define          DATA_CLR    PORTB &= ~(1<<4)
#define          CLK_SET     PORTB |=  (1<<3)
#define          CLK_CLR     PORTB &= ~(1<<3)
#define          STROBE_SET  PORTB |=  (1<<2)
#define          STROBE_CLR  PORTB &= ~(1<<2)
```

Mit der Anweisung `PORTB |= (1<<4)` wird erreicht, dass im PORT-B der Pin PB4 Strom bekommt. In der Klammer steht vorne die 1 für „Strom an“ und die 4 für PB4. Mit der Anweisung PORTB &= ~(1<<4) wird der Strom an diesem PIN wieder ausgeschalten. Das „~“-Zeichen bedeutet eine Umkehrung (Negation).

(Hinweis: Die Zeichenfolge `|=` ist eine Kurzschreibweise für `PORTB= PORTB | (1<<4)`. Die Pin-Belegung von PORT-B erhält also einen neuen Wert. Das „|“-Zeichen ist das logische ODER. Alles Pins des PORT-B bleiben an und das vierte Bit kommt hinzu. Bei dem „&“-Zeichen handelt es sich um das logische UND. `PORT-B &= ~(1<<4)` ist eine Kurzschreibweise für `PORTB=PORTB & ~(1<<4)`. PORT B erhält dadurch einen neuen Wert, alle PINS behalten ihren Wert, nur PB4 wird ausgeschalten.)

Anstelle diese einzelnen Befehle `PORTB=PORTB | (1<<4)` eingeben zu müssen, können wir einfach die definierten Namen verwenden, z. B. `DATA_SET` schreiben bzw. `DATA_CLR`. Das macht den Quellcode sehr viel übersichtlicher. Die anderen vier Definitionen machen das gleiche, nur am Pin PB3 und am Pin PB2.

Das Schieberegister ist eine Möglichkeit, ein serielles Signal parallel auszugeben. Es arbeitet so, dass eine Folge von Bits gesendet wird, z. B. 100011011111. Bei unserer Wordclock werden dadurch die Spalten abgebildet. Es wird also angegeben, ob am unteren Rand der Matrix Strom abfließen darf oder nicht. Um die elf Bits auf die elf Anschlüsse der Matrix zu senden, brauchen wir nur drei Datenleitungen und damit auch nur drei Beinchen des Atmega8, weil wir eben die beiden Schieberegister verwenden, die zusammengeschalten (kaskadiert) sind. Um elf Bits zu übertragen müssen wir elf Mal CLK_SET aufgerufen und danach mit CLK_CLR beenden. Mit DATA_SET schalten wir bei jedem Bit zusätzlich das Beinchen PB4 auf „an“, wenn eine „1“ gesendet werden soll oder mit DATA_CLR die Spannung ab, wenn eine „0“ gesendet werden soll. Mit dem Befehl STROBE_SET wird die Änderung am Schieberegister, die zunächst nur innen gespeichert ist, nach außen freigegeben, mit STROBE_CLR gesperrt. Das Schieberegister arbeitet so, dass pro CLK_SET die aktuelle Bitfolge im Register um eines nach rechts geschoben wird und das neue Bit auf der linken Seite hinzukommt. Ziemlich kompliziert, oder?

```
#define          BUTTON                 (!(PIND & (1<<2)))
```

Mit BUTTON wird eine Bedingung definiert: PD2 wird als Eingang betrieben und es wird geprüft, ob das Beinchen 2 an PORT-D keinen Strom hat. Dies ist der Fall, wenn der Taster gedrückt wird! Mit dem „!“ wird die Anweisung PIND & (1<<2) umgedreht (negiert). Während PIND & (1<<2) bedeutet, dass auf PD2 Strom sein soll. Es mag hier seltsam erscheinen, dass beim Drücken des Tasters kein Strom am Pin PD2 anliegt und beim loslassen des Tasters schon. Dies ist aber tatsächlich so, da durch

einen internen Widerstand Pin PD2 auf VCC (die Spannungsquelle) gezogen wird, wobei nur eine sehr geringe Strommenge fließt. Dennoch ist das Beinchen auf „an“, wenn der Taster nicht gedrückt ist. Wird der Taster gedrückt, so wird der Strom gegen GND abgeleitet und der Zustand des Eingangspins wechselt auf 0.

Um die LEDs zum Leuchten zu bringen, werden nacheinander die Spalten und die Zeilen angegeben, die leuchten sollen. Die Spalten werden mit dem Array `led[0]` bis `led[11]` angesprochen. Die Zeilen mit der Bitfolge 00000001 für die erste Zeile, 00000010 für die zweite usw. Bei Binärzahlen schreibt man für Gewöhnlich ein „0b“ vorne dran, also 0b00000001 und 0b00000010 usw.

Eine Binärzahl kann aber auch als Hexzahl angegeben werden, das ist nicht so schön zu lesen, ist aber üblich. Dafür verwendet man das Präfix „0x“. Anstelle 0b00000001 kann man 0x01 schreiben, anstelle 0b00000010 0x02, für die dritte Zeile 0x04, für die fünfte 0x10, für die sechste 0x20, für die siebte 0x40, für die achte 0x80, für die neunte 0x100, für die zehnte 0x200. Nimm einen Taschenrechner und verwandle die Hex-Zahl in eine Binärzahl und du kannst das nachprüfen.

```
#define       CLEAR_ALL      led[0]=0; led[1]=0; led[2]=0; led[3]=0; led[4]=0; led[5]=0;
                             led[6]=0; led[7]=0; led[8]=0; led[9]=0; led[10]=0; led[11]=0
#define       WORD_ES_IST    led[0]|=1; led[1]|=1; led[4]|=1; led[5]|=1; led[6]|=1
#define       WORD_ZEHN      led[8]|=1; led[9]|=1; led[10]|=1; led[11]|=1
#define       WORD_FUENF     led[1]|=0x02; led[2]|=0x02; led[3]|=0x02; led[4]|=0x02
#define       WORD_VOR_1     led[6]|=0x02; led[7]|=0x02; led[8]|=0x02
#define       WORD_VIERTEL   led[5]|=0x04; led[6]|=0x04; led[7]|=0x04; led[8]|=0x04;
                             led[9]|=0x04; led[10]|=0x04; led[11]|=0x04
#define       WORD_NACH      led[0]|=0x08; led[1]|=0x08; led[2]|=0x08; led[3]|=0x08
#define       WORD_VOR_2     led[4]|=0x08; led[5]|=0x08; led[6]|=0x08
#define       WORD_HALB      led[8]|=0x08; led[9]|=0x08; led[10]|=0x08; led[11]|=0x08

#define       WORD_EMSYSTECH led[1]|=0x10; led[2]|=0x10; led[3]|=0x10; led[4]|=0x10;
                             led[5]|=0x10; led[6]|=0x10; led[7]|=0x10; led[8]|=0x10;
                             led[9]|=0x10
#define       WORD_5         led[0]|=0x20; led[1]|=0x20; led[2]|=0x20; led[3]|=0x20
#define       WORD_2         led[6]|=0x20; led[7]|=0x20; led[8]|=0x20; led[9]|=0x20
#define       WORD_1         led[8]|=0x20; led[9]|=0x20; led[10]|=0x20; led[11]|=0x20
#define       WORD_7         led[0]|=0x40; led[1]|=0x40; led[2]|=0x40; led[3]|=0x40;
                             led[4]|=0x40; led[5]|=0x40
#define       WORD_6         led[7]|=0x40; led[8]|=0x40; led[9]|=0x40; led[10]|=0x40;
                             led[11]|=0x40
#define       WORD_10        led[0]|=0x80; led[1]|=0x80; led[2]|=0x80; led[3]|=0x80
#define       WORD_9         led[3]|=0x80; led[4]|=0x80; led[5]|=0x80; led[6]|=0x80
#define       WORD_4         led[8]|=0x80; led[9]|=0x80; led[10]|=0x80; led[11]|=0x80
#define       WORD_3         led[0]|=0x100; led[1]|=0x100; led[2]|=0x100; led[3]|=0x100
#define       WORD_11        led[4]|=0x100; led[5]|=0x100; led[6]|=0x100
#define       WORD_8         led[8]|=0x100; led[9]|=0x100; led[10]|=0x100; led[11]|=0x100
#define       WORD_12        led[1]|=0x200; led[2]|=0x200; led[3]|=0x200; led[4]|=0x200;
                             led[5]|=0x200
#define       WORD_UHR       led[9]|=0x200; led[10]|=0x200; led[11]|=0x200
#define       WORD_FUNKUHR   led[5]|=0x200; led[6]|=0x200; led[7]|=0x200; led[8]|=0x200;
                             led[9]|=0x200; led[10]|=0x200; led[11]|=0x200
#define       PIC_FRAME      led[0] = 0x3FF; led[1] = 0x201; led[2]=0x201; led[3]=0x201;
                             led[4]=0x201;  led[5]=0x201; led[6]=0x201; led[7]=0x201;
                             led[8]=0x201; led[9]=0x201; led[10]=0x201; led[11]=0x3FF
```

Mit diesen Define-Anweisungen wird also bereits festgelegt, welche LEDs bei einer bestimmten Uhrzeit leuchten müssen. Die Anweisungen

CLEAR_ALL
WORD_ES_IST
WORD_12

WORD_UHR
genügen also, um die Anzeige auf 12 Uhr anzuzeigen.

Im Einstell-Modus werden die aktuellen Stunden und Minuten in Form einer zweistelligen Zahlenangabe angegeben. Dies erfolgt, indem in einem Array `font[40]` die Zahlen 0 bis 9 dargestellt werden:

```
const uint8  font[40] =             // Digits
{
       0xFE,0x82,0x82,0xFE,         // 0
       0x08,0x04,0xFE,0x00,         // 1
       0xC4,0xA2,0x92,0x8C,         // 2
       0x82,0x92,0x92,0x7C,         // 3
       0x1E,0x10,0xF8,0x10,         // 4
       0x9E,0x92,0x92,0x62,         // 5
       0x78,0x94,0x92,0x60,         // 6
       0x02,0xF2,0x0A,0x06,         // 7
       0xFE,0x92,0x92,0xFE,         // 8
       0x9E,0x92,0x92,0xFE          // 9
```

Jeder Hex-Code ist als Bitmuster einer von vier Zeilen zu verstehen, mit der die Ziffer dargestellt wird. Beispiel für die Ziffer 0:

```
0xFE,0x82,0x82,0xFE,          //    11111110
                              //    10000010
                              //    10000010
                              //    11111110
```

Das Bitmuster um 90° gedreht ergibt die Ziffer 0, wenn die 1er die LEDs darstellen, die an sind.

4.2 Variablen

Im Quelltext folgt jetzt die Definition von „lokalen“ Variablen, naja es sind eher globale Variablen:

```
/**** Local variables **********************************************************/
uint8          c;
uint16         led[12];
uint8          col_cnt;
uint16         col;
uint8          button_mem;
uint8          hour,minute,second;
uint8          time_setup;
uint8          time_setup_cnt0;
uint8          time_setup_cnt1;
uint16         loop_cnt;
uint8          sec_flag,key_flag;
```

Einige davon werde ich erst später kommentieren. Aber die für den Ablauf des Programms Wesentlichen schon hier. Zunächst einmal zum Datentyp. Der Datentyp `unit8` beschreibt eine Ganze Zahl, die in 8 Bit gespeichert wird, was fast immer ausreichend ist. Bei dem Array led[12] wollen wir aber zwölf Bitmuster mit einer Länge von 10 Bit speichern, dies entspricht den LEDs in einer Zeile. Dafür reichen 8 Bit nicht aus. Mit einer 8-Bit-Variable können nur Bitmuster der Länge 8 bzw. Werte bis 255 gespeichert werden, brauchen wir größere Zahlen, dann nehmen wir unit16, damit können Bitmuster bis zu einer Länge von 16 und Zahlenwerte bis 2^{16} gespeichert werden.

Zur Klarstellung: Manchmal nehmen wir Variablen ganz normal zum zählen, manchmal interessiert uns ihr Bitmuster, um die Variable dafür herzunehmen, die Beinchen des Mikrocontrollers an- und auszuschalten.

In den Variablen `hour`, `minute` und `second` speichern wir die aktuelle Uhrzeit. Die Variable `time_setup` verwenden wir, den aktuellen Anzeigezustand der Uhr abzuspeichern. Ist der Wert der Variable `time_setup` 0, dann wird ganz normal die Uhrzeit angezeigt. Hat sie hingegen den Wert 1 so befinden wir uns im Modus „Minuten einstellen“. Bei einem Wert von 2 befinden wir uns im Modus „Stunden einstellen“.

4.3 Mit Methode draw_time(void)

Die erste Methode ist die Methode `draw_time(void)`. Sie zerfällt in drei Teile, je nachdem welchen Wert die Variable `time_setup` besitzt.

Falls der Wert der Variable `time_setup` 0 ist, soll die normale Uhrzeit angezeigt werden.

```
else  // die normale Uhrzeit anzeigen
       {
              CLEAR_ALL;
              WORD_ES_IST;
              WORD_UHR;

              h = 0;
              if(     minute <   5) { h = hour; }
              else if(minute <  10) { WORD_FUENF;   WORD_NACH; h = hour; }
              else if(minute <  15) { WORD_ZEHN;    WORD_NACH; h = hour; }
              else if(minute <  20) { WORD_VIERTEL; WORD_NACH; h = hour; }
              else if(minute <  25) { WORD_ZEHN;    WORD_VOR_1;   WORD_HALB; h = hour + 1; }
              else if(minute <  30) { WORD_FUENF;   WORD_VOR_1; WORD_HALB; h = hour + 1; }
              else if(minute <  35) { WORD_HALB;  h = hour + 1; }
              else if(minute <  40) { WORD_FUENF;  WORD_NACH; WORD_HALB; h = hour + 1; }
              else if(minute <  45) { WORD_ZEHN; WORD_NACH; WORD_HALB; h = hour + 1; }
              else if(minute <  50) { WORD_VIERTEL; WORD_VOR_2; h = hour + 1; }
              else if(minute <  55) { WORD_ZEHN; WORD_VOR_1; h = hour + 1; }
              else if(minute <  60) { WORD_FUENF; WORD_VOR_1; h = hour + 1; }

              if     (h ==  0) { WORD_12; }
              else if(h ==  1) { WORD_1;  }
              else if(h ==  2) { WORD_2;  }
              else if(h ==  3) { WORD_3;  }
              else if(h ==  4) { WORD_4;  }
              else if(h ==  5) { WORD_5;  }
              else if(h ==  6) { WORD_6;  }
              else if(h ==  7) { WORD_7;  }
              else if(h ==  8) { WORD_8;  }
              else if(h ==  9) { WORD_9;  }
              else if(h == 10) { WORD_10; }
              else if(h == 11) { WORD_11; }
              else if(h == 12) { WORD_12; }
       }
}
```

Der Quelltext ist nicht schwer. Je nachdem welche Werte die Variablen `hour` und `minute` haben, wird die Uhrzeit angezeigt. Ab einer Minute von größer oder gleich 20 sprechen wir bei einer Zeitangabe von der nächsten Stunde. Bei 7:10 Uhr ist es ZEHN NACH SIEBEN, bei 7:25 Uhr ist es aber bereits FÜNF VOR HALB ACHT. Daher muss für Minutenwerte größer gleich 20 die nächste Stunde angegeben werden. Dazu dient die Variable `h`.

Sehen wir uns nun den Einstellmodus an. Wird der Taster mehr als 2 Sekunden gedrückt gelangen wir in den „Einstellmodus Minute“. Der Modus wird in der Variable `time_setup` gespeichert, die dann den Wert 1 besitzt.

```
else if(time_setup == 1){
  if (second & 1) { led[0] = 0x44; led[1] = 0;}
  else { led[0] = 0; led[1] = 0x44;}

  led[2] = 0;
  i = (minute / 10) * 4;
  led[3] = font[i]; led[4] = font[i+1]; led[5] = font[i+2]; led[6] = font[i+3]; led[7] = 0;
  i = (minute % 10) * 4;
```

```
  led[8] = font[i]; led[9] = font[i+1]; led[10] = font[i+2]; led[11] = font[i+3];
}
```

Während der Anzeige der Minuten sollen 4 LEDs in der Spalte 1 `led[0]` und der Spalte 2 `led[1]` abwechselnd blinken. Um in der ersten Spalte die dritte und siebte LED zum Leuchten zu bringen verwenden wir wiederum den Befehl `led[0]=0x44;` Die Hexadezimalzahl `0x44` steht für die Binärzahl `0b1000100`. Die zweite Spalte wird mit dem Befehl `led[1]=0;` ganz ausgeschalten.
In der `else`-Anweisung wird nun die erste Spalte ausgeschalten und in der zweiten Spalte die dritte und siebte LED eingeschalten.

Mit der Anweisung `(second  & 1)` können wir im Sekunden-Takt zwischen den beiden Anweisungen hin- und herschalten. Bitte beachtet: `second` ist eine Variable. Der Operator `&` ist das logische „UND“ und wird immer bitweise angewendet. Die Sekunden zählen binär von `0b00000000`, `0b00000001`, `0b00000010`, `0b00000011` usw. bis 60, dann beginnt es wieder bei 0.
In der Binärschreibweise ändert sich also das letzte Bit jede Sekunde von `1` auf `0` und dann wieder auf `1`. Mit dem UND-Vergleich `(second & 1)` prüfen wir also, ob das letzte Bit der Sekunde gleich 1 ist. Das ist alle zwei Sekunden der Fall.

Die ersten beiden Spalten blinken also im Sekundentakt. Die dritte Spalte wird ausgeschalten mit `led[2]=0`. Die Spalten 4 bis 7 `led[3]` bis `led[6]` dienen zur Anzeige der ersten Ziffer der Minutenanzeige. Die achte Spalte ist wieder ausgeschalten. Die Spalten 9 bis 12 zur Anzeige der zweiten Ziffer der Minutenanzeige. Dazu verwenden wir die Werte der Varible `font`, die oben so definiert wurde:

```
const uint8  font[40] =             // Digits
{
       0xFE,0x82,0x82,0xFE,         // 0
       0x08,0x04,0xFE,0x00,         // 1
       0xC4,0xA2,0x92,0x8C,         // 2
       0x82,0x92,0x92,0x7C,         // 3
       0x1E,0x10,0xF8,0x10,         // 4
       0x9E,0x92,0x92,0x62,         // 5
       0x78,0x94,0x92,0x60,         // 6
       0x02,0xF2,0x0A,0x06,         // 7
       0xFE,0x92,0x92,0xFE,         // 8
       0x9E,0x92,0x92,0xFE          // 9
}
```

Immer vier Werte geben eine Ziffer und sagen uns, welche Zeilen anzuschalten sind. Die Abfrage geschieht nun folgendermaßen: Nehmen wir an, wir wollen die Ziffer 5 anzeigen. Wir sehen sofort, dass wir die Werte `0x9E`, `0x92`, `0x92`, `0x62` benötigen. In unserem Array `font` sind das aber die Werte `font[20]`, `font[21]`, `font[22]` und `font[23]`.

Mit der Anweisung `i = (minute / 10) * 4;` teilen wir den aktuellen Wert für Minuten durch 10. Der Rest der Division wird verworfen und nicht weiter beachtet. Als Ergebnis erhalten wir die erste Ziffer unserer Variablen `minute`. Jetzt multiplizieren wir dieses Zwischenergebnis mit 4. Denn jede Ziffer beginnt im Array entweder am Index 0 oder einem vielfachen Wert von 4.

Für Minuten-Werte kleiner als 10 ergibt sich hier eine 0. Die Anzeige soll ja auch für einstellige Minuten-Werte zweistellig sein. Für alle Werte z. B. von 50 bis 59 ergibt sich ein Endergebnis von 20, das ist der Beginn der Ziffer 5. Dies funktioniert für alle Ziffern zwischen 0 und 59 und ergibt ein Ergebnis von 0 bis 20, aber nur in 4er Schritten, den Beginn der Ziffern 0, 1, 2, 3, 4, 5 ist im Array `font` am Index 0, 4, 8, 12, 16, 20.

Die Werte der `led[3]` bis `led[5]`, das ist die Anzeige der ersten Ziffer unserer Minutenanzeige lässt sich so bequem einstellen.

Bei der zweiten Ziffer ist es ähnlich, aber hier wird nicht geteilt, sondern der Modulo-Operator `%` verwendet. Die Berechnung ergibt den Rest einer Ganzzahlen-Division, die wir aus der Grundschule kennen. Das Ergebnis ist immer der Teil, der sich nicht teilen lässt. An einem Beispiel will ich das erklären: `(10 % 3)` ergibt den Wert 1, denn 10 geteilt durch 3 ist „3 Rest 1“, und mit dem Modulo-Operator wird immer nur der Rest beachtet. `(11 % 3)` ist dann 2, `(12 % 3)` ergibt 0, `(13 % 3)` wieder 1 usw.

Mit der Anweisung `i = (minute % 10) * 4` ermitteln wir einfach die zweite Ziffer der Variable `minute`. Der Beginn in unserem Array `font` ist dann wieder der vierfache Wert dieser Ziffer.

Ganz ähnlich erfolgt auch das Einstellen der Stunden. Wir erkennen diesen Modus daran, dass die Variable `time_setup` den Wert 2 hat.

```
if(time_setup == 2)
{
  i = (hour / 10) * 4;
  led[0] = font[i];
  led[1] = font[i+1];
  led[2] = font[i+2];
  led[3] = font[i+3];
  led[4] = 0;  // Eine Spalte ausschalten

  i = (hour % 10) * 4;
  led[5] = font[i];
  led[6] = font[i+1];
  led[7] = font[i+2];
  led[8] = font[i+3];
  led[9] = 0;
  if (second & 1){ led[10] = 0x44;   led[11] = 0;   }
  else  { led[10] = 0; led[11] = 0x44;}
}
```

Im Unterschied beginnt hier die Anzeige schon in der ersten Spalte, die blinkenden LEDs sind dafür an der rechten Seite der LED-Anzeige in den Spalten 11 und 12 `(led[10] und led[11])`.

Wenden wir uns nun dem Timer `ISR(TIMER2_OVF_vect)` zu, die sich unten im Quelltext befindet.

4.4 Timer2 ISR(TIMER2_OVF_vect)

Mit dem Atmega8 können wir Interrupts auslösen. Ein Interrupt ist eine Unterbrechung des aktuellen Programms, wenn ein Ereignis eintritt. In unserem Fall soll es ein Timer-Ereignis sein, also regelmäßig nach Ablauf einer bestimmten Zeitspanne eintreten. Wie wir den Timer einstellen sehen wir später in der `main`-Methode. Wir können uns dann aber darauf verlassen, dass in festgelegten Abständen die Methode aufgerufen wird. Die folgende Methode wird jede Sekunde automatisch aufgerufen.

```
ISR(TIMER2_OVF_vect)
{
	second++;
	if(second > 59)
	{
		second=0;
		minute++;
		if(minute > 59)
		{
			minute = 0;
			hour++;
			if(hour > 11) hour = 0;
		}
	}
	sec_flag = true;
}
```

Die Uhrzeit wird um eine Sekunde weiter gezählt. Überläufe bei Werten von 60 für Sekunden und Minuten und bei Werten von 12 bei Stunden werden korrigiert.

Zusätzlich erhält die Variable `sec_flag` den Wert `true`. Damit kann auch in der Endlosschleife der `main`-Methode auf das Ereignis reagiert werden. (Hinweis für Java-Programmierer: Alle oben vereinbarten Variablen sind globale Variablen und überall gültig!)

4.5 Der Timer 1

Ein weiteres Timer-Interrupt-Ereignis wird alle 0,83 Millisekunden ausgelöst und dient der Steuerung der LED-Anzeige.

```
ISR(TIMER1_COMPA_vect)
{
  col_cnt++;
  if(col_cnt > 11){
    key_flag = true; // zum sampeln/entprellen der Taste
    col_cnt=0;
    DATA_CLR;
  }
  else {
    DATA_SET;
  }
  CLK_SET;
  CLK_CLR;

  PORTC &= ~0x0F; // Ports löschen
  PORTD &= ~0xF0;
  PORTB &= ~0x03;

  STROBE_SET;
  STROBE_CLR;

  col = led[col_cnt];

  PORTC |= col & 0x0F;
  PORTD |= col & 0xF0;
  PORTB |= (col >> 8) & 0x03;
}
```

Die Variable `col_cnt` zählt jeweils von 0 bis 12, es wird jeweils nur eine Spalte nacheinander zum Leuchten gebracht, dann wird der Wert der Variable `col_cnt` um eins erhöht und die nächste Spalte angezeigt. Und dies alle 0,83 Millisekunden. Da unser Prozessor mit 8 MHz arbeitet dauert es 6667 Computertakte, bis der Interrupt erneut aufgerufen wird. Insgesamt dauert es ca. 10 Millisekunden, bis alle Spalten einmal angezeigt wurden. Das Auge sieht das Flimmern normalerweise nicht, weil es so schnell hintereinander passiert, dass das Auge dem Wechsel nicht folgen kann. Wir erreichen damit eine Bildwiederholungsfrequenz von 100 Hz.

Bei jedem Interrupt wird das `key_flag` gesetzt. Damit können wir im Hauptprogramm (`main`-Methode) auch auf das Ereignis reagieren.

Bei jedem Interrupt setzten wir die aktuellen Daten, die sich im Array `led[]` befinden, mit `DATA_SET` in unser Schieberegister. Nur wenn die Anzeige neu aufgebaut wird (col_cnt den Wert 12 hat), löschen wir die Daten mit `DATA_CLR`. Danach setzen wir das Clock-Signal für das Schieberegister mit `CLK_SET` und löschen es sofort danach wieder mit `CLK_CLR`, sonst würde es ja immer an bleiben.

Alle unsere Signale für die zehn Zeilen werden abgeschalten mit `PORTC &= ~0x0F; PORTD &= ~0xF0; PORTB &= ~0x03;`

Wir geben die Daten am Schieberegister frei mit `STROBE_SET` und sperren es gleich wieder mit `STROBE_CLR`. Wie Werte bleiben aber erhalten.

Die Variable `col` erhält den Wert von `led[col_cnt]` und damit das Bitmuster der Spalte, die angezeigt werden soll.

Jetzt werden die Ports wieder auf „an“ geschalten. Dazu werden die Bits von `col` mit allen Ausgängen durch ein logisches UND verknüpft. `PORTC |= col & 0x0F;  PORTD |= col & 0xF0;`
`PORTB |= (col >> 8) & 0x03;`

Die LEDs leuten also.

4.6 Die main-Methode

Nun können wir uns endlich der wichtigsten Methode zuwenden, der `main`-Methode. Die beinhaltet alle wesentlichen Einstellungen zu Beginn des Programms und mündet dann in einer Endlosschleife, weil die Uhr ja unendlich laufen soll.

Daneben ist es die Aufgabe der `main`-Methode, jederzeit zu prüfen, ob die Taste gedrückt wurde und dann die Tasteneingaben auszuwerten, um vom „Uhr-Anzeige-Modus“ in den „Minuten-Einstell-Modus“ und weiter in den „Stunden-Einstell-Modus“ zu gelangen und die Zeit-Einstellungen vorzunehmen.

Vor Beginn der Endlosschleife werden grundlegende Einstellungen vorgenommen:

```
int main(void)
{
  DDRC = 0x0F;
  DDRD = 0xF0;
  DDRB = 0x03 + (1<<2) + (1<<3) + (1<<4);
  PORTD |= (1<<2); // für Pullup
  SFIOR &= (1<<PUD);


  // Timer 2 mit externem 32kHz Quarz betreiben
  ASSR |= (1<<AS2);
  TCCR2 = (1<<CS22) + (1<<CS20);

  // Timer 1 für LED INT mit ca. 1,2kHz
  TCCR1A = 0;
  TCCR1B = (1<<WGM12) + (1<<CS10);
  OCR1A = 6667;

  // Timer Interrupts
  TIMSK  = (1<<OCIE1A) + (1<<TOIE2);   // set interrupt mask


  hour = 0;
  minute = 0;
  second = 0;
  time_setup = 0;

  sei();       // Interrupt ein
```

Am Beginn eines Programmes muss festgelegt werden, welche Beinchen als Ausgang verwendet werden sollen. Dies erfolgt mit der Anweisung DDRx und einem Wert, der die Pin-Belegung anzeigt. DDRx gibt die Richtung der Pins an, also ob es Eingänge sind oder Ausgänge. Eingänge erhalten den Wert 0, Ausgänge den Wert 1.

DDRC=0x0F könnte auch als DDRC=0b00001111 geschrieben werden. Die letzten vier Pins werden also als Ausgang verwendet. Mit DDRD=0xF0 werden die ersten vier Pins des Ports D als Ausgang definiert. Mit DDRB=0x03 werden die letzten beiden Pins des Ports B als Ausgang definiert. Zudem

kommen noch die Anweisungen `+ (1<<2) + (1<<3) + (1<<4)` hinzu, so dass auch die Beinchen 2, 3 und 4 als Ausgang verwendet werden. Letztere sind die drei Datenleitungen für die Schiftregister.

Mit der Bezeichnung „PORTD“ ist ein Register, also ein Speicherbereich im Atmega8 gemeint. Den Port PD2 verwenden wir als Eingang. Es wird der Taster an PD2 angeschlossen. Ein Taster schließt den Stromkreis wenn er gedrückt wird. Dies würde aber bedeuten, dass PD2, wenn der Taster nicht gedrückt ist nicht am Stromkreis angeschlossen wäre. Dies ist allerdings nicht gut bei einem Mikrocontroller. Alle Beinchen sollten an eine Datenleitung angeschlossen sein, da geringe elektromagnetische Störungen dem Mikrocontroller dazu bringen könnten, zu glauben, der Taster wäre geschlossen. Aus diesem Grund könnte man außen einen Widerstand anschließen, der – für den Fall dass der Taster nicht geschlossen ist, das Beinchen mit VCC, also dem Pluspol verbindet. Ein solcher Widerstand – man nennt ihn Pull-Up-Widerstand ist aber auch in den Atmega8 eingebaut. Man kann ihn einschalten und dann einen Taster direkt an den Port anschließen. Die andere Seite des Tasters wird gegen GND geschalten. Ist der Taster nicht gedrückt, so hat er durch den hochohmigen Widerstand eine sehr geringe – aber dennoch vorhandene Spannung, da er intern gegen VCC geschalten ist. Das bedeutet: Ist der Taster nicht gedrückt, so liegt am Pin Spannung an, also steht der Pin auf an (1). Wird der Taster gedrückt, so wird er auf GND gelegt und die Spannung sinkt ab (0). Zum Einschalten des Pull-Up-Wiederstandes benötigt man auch das Register SFIOR. Das zweite Bit des SFIOR-Registers hat die Bezeichnung PUD.

Als nächstes wird der Timer 2 eingestellt. Nach dem Datenblatt des Atmega8 muss das AS2-Bit des Registers ASSR gesetzt werden, wenn ein externer Quarz als Taktgeber verwendet werden soll. (Suche im Datenblatt nach dem Register ASSR und schaue dir die Belegung an!) Dies erfolgt durch die Anweisung ASSR |= (1<<AS2);

Nun würde der Timer, vorausgesetzt wir haben eine Quarz mit 32kHz angeschlossen, jede Sekunde das Register des Timers um 1 erhöhen, der Timer würde also alle 256 Sekunden einmal ausgelöst, wenn das Register überläuft. Es gibt aber das Register `TCCR2`. Mit drei Bits lässt sich ein Prescaler einstellen. In unserem Fall wird der Prescaler auf die Bitfolge 101 eingestellt. Nach dem Datenblatt des Atmeg8 entspricht dies einem Teiler von 128. Dies hat zur Folge, dass der Timer jede Sekunde ausgelöst wird. Damit lässt sich auch klären, warum ein Uhrenquarz so eine seltsam erscheinende Frequenz von 32768 Hz besitzt. Weil es sich gut durch 128 teilen lässt und sich damit Sekunden-Timer sehr gut und ohne gerundete Zahlen erzielen lassen.

Um die Register-Bits `CS22`, `CS21` und `CS20` des Registers `ASSR` auf die Bitfolge 101 einzustellen verwenden wir die Anweisung `TCCR2 = (1<<CS22) + (1<<CS20);` Das mittlere Bit bleibt dann bei 0.

Als nächstes wird der Timer 1 eingestellt. Dieser soll, wie oben geschrieben wurde, alle 0,83 Millisekunden ausgelöst werden, also 1200 Mal pro Sekunde. Der Timer 1 ist ein 16-Bit-Timer. Er besitzt aus diesem Grund zwei Register `TCCR1A` und `TCCR1B`. Dieser Timer hat verschiedene Modi, was im Datenblatt des Atmega8 nachzulesen ist. Wird das Bit `WGM12` auf 1 gesetzt, so bedeutet dies, dass der Counter auf 0 gesetzt wird, wenn der im Register `OCR1` festgelegte Wert erreicht ist. Mit den Bits `CS12`, `CS11` und `CS10` kann wie oben dargestellt ein Teiler (Prescaler) eingestellt werden, der das Timer-Ereignis verzögern würde. In unserem Beispiel wird der Wert auf die Bitfolge 0b001 eingestellt. Nach dem Atmega8-Datenblatt bedeutet dies, dass kein Prescaler verwendet wird.

Als Wert für den `OCR1` wird die Zahl 6667 festgelegt. Der Atmega8 arbeitet mit 8 MHz. Daraus lässt sich errechnen, wie lange es dauert, bis der Prozessor bis 6667 gezählt hat. Es ergibt sich eine eintausendzweihundertstel Sekunde, also 1200 Mal pro Sekunde zählt der Prozessor den Timer von 0 auf 6667. Die Anweisungen im Quelltext lauten:

```
TCCR1A = 0;    // erster Teil des 16-Bit-Registers wird nicht benötigt
TCCR1B = (1<<WGM12) + (1<<CS10);     // Modus auf 0 stellen, wenn OCR1A-Wert erreicht ist.
                                     // Kein Teiler.
OCR1A = 6667;
```

Damit ist der Timer festgelegt. Dass die Interrupts aktiviert werden, dazu ist die Anweisung
`sei();`
notwendig. Wenn wir, wie in unserem Beispiel genau zwei Timer als Interrupt-Ereignisse verwenden, so können wir die anderen Interrupts abschalten. Dazu gibt es ein spezielles Register. Das vierte und das sechst Bit des Registers `TIMSK` legt fest, auf welche beiden Interrupts reagiert werden soll. Es sind die Overflow-Interrupts für den Timer 1 und den Timer 2. Alle anderen möglichen Interrupts werden nicht geprüft.

```
TIMSK  = (1<<OCIE1A) + (1<<TOIE2);
```

Die Startwerte für `hour`, `minute` und `second` werden auf 0 gesetzt, ebenso der Anzeigemodus `time_setup` das bedeutet, dass die aktuelle Uhrzeit 0:00 Uhr angezeigt wird.

Nun endlich sind wir bei der Endlosschleife angelangt. Hier wird permanent geprüft, ob der Taster gedrückt wurde, um entsprechend darauf reagieren zu können.

```
// Hier beginnt die Endlosschleife unseres Programms
while(1)
{
  if(sec_flag)               {
  {
    sec_flag = false;
    if(BUTTON)
    {
      time_setup_cnt1++;
      if(time_setup_cnt1 > 3)
      {
        time_setup++;  if(time_setup > 2) time_setup = 0;
      }
    }
```

In die erste Sequenz gelangen wir jede Sekunde einmal, wenn das `sec_flag` vom Interrupt-Timer auf `true` gesetzt wird. Es wird dabei geprüft, ob der Taster gedrückt wurde. Ist dies der Fall zählt die Variable `time_setup_cnt1` um eins weiter. Ziel ist es, dass nach zwei Sekunden Drücken des Tasters der Einstell-Modus aktiviert wird. Dies erfolgt mit der bedingten Anweisung `if (time_setup_cnt1>3)`.

Ist man nach zwei Sekunden Drücken in den Minuten-Einstell-Modus gewechselt, so gelangt man in den Stunden-Einstell-Modus und schließlich in den Zeitanzeige-Modus nach jeweils nochmaligem zwei Sekunden langem Drücken des Tasters.

```
    else  // wenn kein Button gedrückt wurde
    {
      time_setup_cnt0++;
      if(time_setup_cnt0 > 5) time_setup = 0;
    }
```

Wenn der Taster losgelassen wurde zählt die Variable `time_setup_cnt0` im Sekundentakt weiter, um nach 4 Sekunden die Zeitanzeige wieder zu aktivieren `(time_setup = 0);`

Schließlich soll die Zeit aktualisiert werden, wohlgemerkt zunächst nur im Array `led[]`.

```
    draw_time();
  }
```

Neben dem gerade besprochenen Prüfen in Sekundenabständen wird auch alle 10 Millisekunden, immer wenn eine ganze LED-Ansicht neu angezeigt wurde (12 Mal 0,83 Millisekunden) die folgende Routine abgearbeitet, wenn der Timer das `key_flag` setzt.

```
  if(key_flag) //=== 10ms ===
  {
    key_flag = false;
    if(!BUTTON && button_mem)
    {
      button_mem = false;
      if(time_setup == 1) minute = (minute<59) ?minute+1 : 0;
      if(time_setup == 2) hour = (hour < 11) ? hour+1 :    0;
    }
    if(BUTTON)
    {
      button_mem = true;
      time_setup_cnt0 = 0;
    }
    else
    {
      time_setup_cnt1 = 0;
    }
    draw_time();
  }

}
}
```

Die Variable `button_mem` wird benötigt, um den Taster zu entprellen. Ein Taster ist ein mechanisches Bauteil, bei dem i.d.R. durch Drücken ein Kontakt geschlossen wird. Taster sind aber meist nicht so exakt gefertigt, dass man ausschließen könnte, dass beim Drücken mehrere kurze Impulse hintereinander gesetzt werden. Dies ist ein mechanisches Problem. Die elektrischen Impulse kommen ganz schnell hintereinander, bis der Taster einrastet. Für den Atmega8 würde dies bedeuten, dass er glaubt, der Taster wäre mehrere Male hintereinander gedrückt worden. Um dieses Problem auszuschließen – man nennt dies ein Entprellen des Tasters - wird eine Variable `button_mem` auf `true` gesetzt, sobald ein Kontakt vom Mikrocontroller erkannt wird.

```
    if(BUTTON)
    {
      button_mem = true;
      time_setup_cnt0 = 0;
    }
```

Gleichzeitig wird die Variable `time_setup_cnt0` auf 0 gesetzt. Wir erinnern uns, dies ist die Zeitdauer von 4 Sekunden, bis die Anzeige automatisch wieder in den Uhrenanzeige-Modus springt. Wird also der Taster gedrückt, so wird die Zeit bis zum automatischen Rücksprung wieder von vorne zu laufen beginnen.

Sollte die Taste losgelassen worden sein, bemerkt das der Mikrocontroller innerhalb von 10 Millisekunden und stellt die Zählvariable `time_setup_cnt1` auf 0, damit wird erreicht, dass nach dem Drücken des Tasters erneut zwei Sekunden gewartet werden muss, bis der Einstell-Modus sich verändert.

```
else
    {
      time_setup_cnt1 = 0;
    }
```

Im vorderen Teil der Sequenz wird geprüft, ob der Button losgelassen wurde, nachdem er gerade gedrückt gewesen war. Dies entspricht einem kurzen Tastendruck und sichert gleichzeitig, dass die Taste entprellt ist, da zwischenzeitlich 10 Millisekunden vergangen sind, genügend Zeit, um alle „falschen“ Impulse des Tasters zu ignorieren, kurz genug, damit wir nicht schon ein zweites Mal die Taste drücken wollen.

```
    if(!BUTTON && button_mem)
```

```
    {
      button_mem = false;
      if(time_setup == 1) minute = (minute<59) ?minute+1 : 0;
      if(time_setup == 2) hour = (hour < 11) ? hour+1 :    0;
    }
```

Abhängig davon, in welchem Einstellmodus wir uns befinden wird nun die `minute` oder die `hour` um eins erhöht, ggf. beim Überlauf von 59 bzw. 11 auf 0 gesetzt.

Ach ja, um es nicht zu vergessen: Alle 10ms wird die Uhrzeit aktualisiert:

```
draw_time();
```