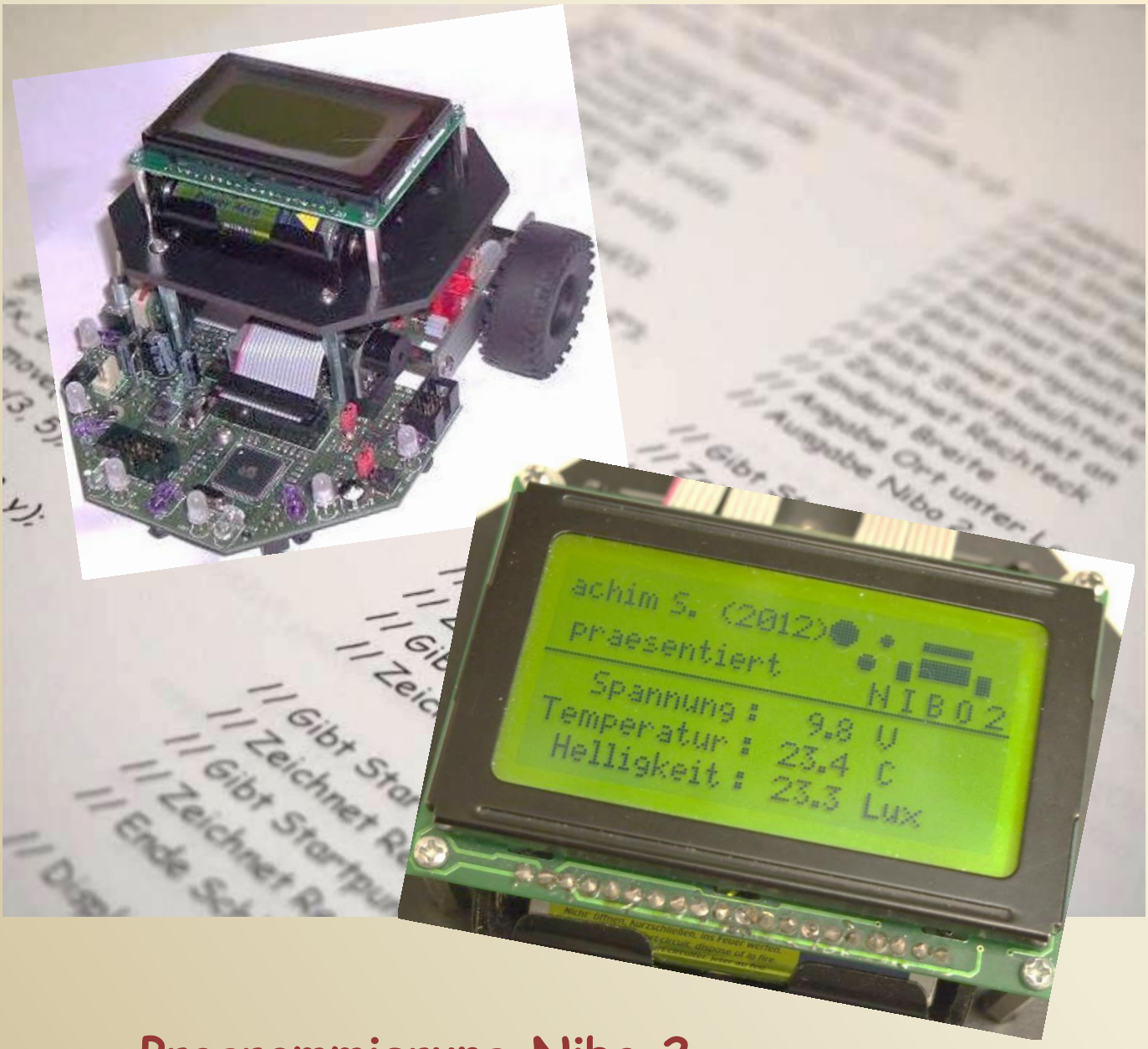
NIBO 2*

PROGRAMMIERUNG



Programmierung Nibo 2
Teil 8 – Multitasking

* by nicai-systems

Nibo 2

Programmierung
Teil 8 – Multitasking

(Multitasking bedeutet ein quasi paralleles Ausführen von mehreren Prozessen auf einem Prozessor)

Notwendige Programme:

- AVR Studio 6 (in der aktuellen Version)
- Nibo Library (in der aktuellen Version)

Bitte diese Programme nach Anweisung des Herstellers oder des Tutorials von Nicai installieren. Ich arbeite mit Windows 7 und den angegebenen Programmen. Es können auch andere Programme, z.B. Linux, verwendet werden. Welches Programm ihr nutzt, müsst ihr selber entscheiden.
Alle Fotos und Programme sind von hjs und nicai.
Die Beispielprogramme findet ihr bei Roboter.cc. Die Nutzung erfolgt auf eigenes Risiko.
Mein besonderer Dank gilt Dieter, Falk und Marco für die Hilfe.

by H.J.Seeger

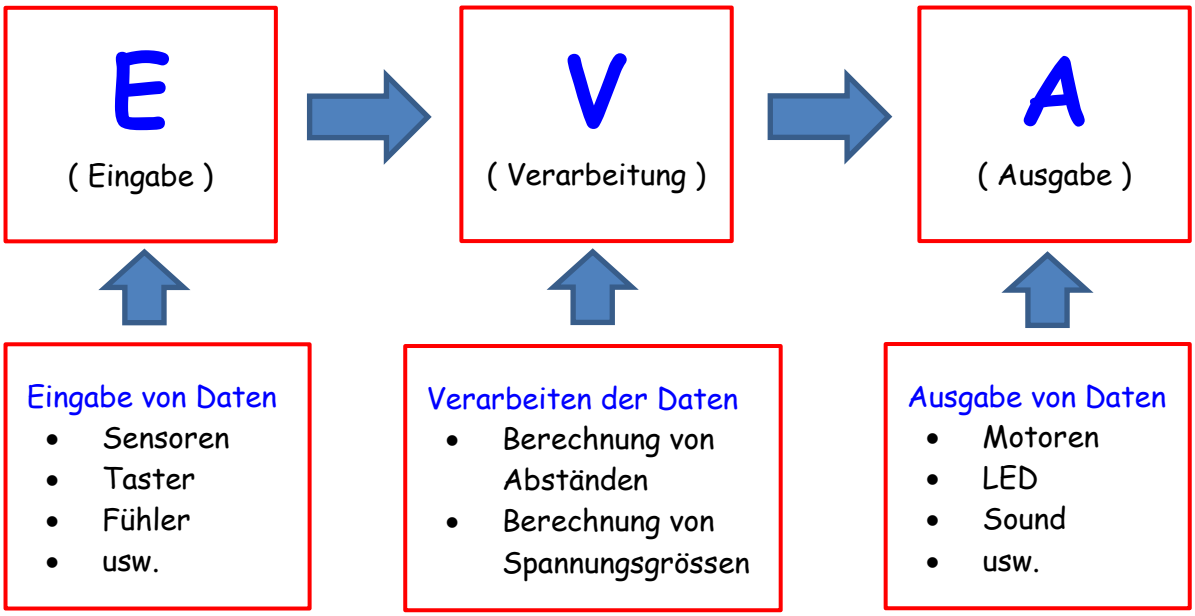
Diese Tutorial entstand unter Verwendung von Beiträgen von Falk
http://www.mikrocontroller.net/articles/Multitasking
Dabei werden Textstellen wörtlich zitiert (*). Alle Beispiele wurden dem Nibo2 angepasst.

1. Einleitung *

Da eine echte parallele Ausführung von mehreren Prozessen (Programmen, Funktionen) auf einem einzelnen CPU-Kern nicht möglich ist, wird ein "Trick" verwendet. Dabei werden die einzelnen Prozesse jeweils nur für kurze Zeit (1..50 ms) bearbeitet und danach auf einen anderen Prozess umgeschaltet. Man spricht auch von einer verschachtelten Bearbeitung (**interleaving**). Das Herz jedes Multitasking-Systems ist der Scheduler. Dieses Programm beinhaltet einen Algorithmus, der überprüft, welcher Prozess als nächstes die CPU (also Rechenzeit) zugeteilt bekommt. Leider ist das auf unserem kleinen ATmega 128 nicht so vorgesehen.
An Hand von Beispielen möchte ich euch zeigen, was möglich ist.

2. Das EVA – Prinzip

Sehen wir uns einmal die Verarbeitung von Daten auf einem Rechner bzw. dem Nibo2 an:



Das EVA-Prinzip bedeutet also, die Eingabe, Verarbeitung und Ausgabe der Daten. Da wir nur einen CPU-Kern haben, können wir auch Eingabe, Verarbeitung oder Ausgabe nicht gleichzeitig durchführen, sondern nur streng sequentiell. Sehen wir uns mal ein kleines Stück Programm vom Nibo2 - LED 1 dazu an:

```
int main()                              // Start des Programmes
 {
  leds_init();                          // LEDs initiiert
  while(1==1)                           // Endlosschleife
    {                                   // Klammer Anfang
      delay(500);                       // Pause 500 ms
      leds_set_status(LEDS_RED, 5);     // setzt LED 5 auf rot
      delay(500);                       // Pause 500 ms
      leds_set_status(LEDS_GREEN, 5); // setzt LED 5 auf grün
    }                                   // Klammer Ende
  return 0;                             // Zurück, Programmschleife von vorn
 }                                      // Ende Programm
```

Zur Erzeugung einer Blinkfrequenz von 1 Sekunde benutzen wir den Befehl delay(500). Damit erfolgt eine Verzögerung von jeweils 0,5 s. Nach dem Ablauf dieser Zeit werden die LED ein- oder ausgeschaltet. Was macht der Prozessor eigentlich während dieser Zeit?
Die Antwort ist sehr einfach – eigentlich gar nichts. Ist nicht ganz korrekt, er zählt fleissig vor sich hin und führt „Nichts-Befehle" (nop) aus. Er tritt sozusagen auf der Stelle, mit Blick auf die Uhr und wartet bis die Zeit um ist.

3. Warum kann „delay (..)" gefährlich sein?

Nehmen wir doch einfach mal die folgende Situation an. Nibo 2 ist gerade dabei die Zeit zum Einschalten einer LED zu berechnen oder eine Textausgabe zu machen. Während dieser Zeit drehen sich die Motore, gesteuert durch den Co-Prozessor, so langsam vor sich hin und Nibo schiebt sich dabei über die Tischplatte. Da die Zeit noch nicht abgelaufen ist, macht er nichts anderes. Er misst nicht den Abstand der Bodensensoren, er registriert nicht den durch den Co-Prozessor gemessenen Abstand von Hindernissen in Fahrtrichtung, er berechnet nur die Zeit oder besser gesagt, er wartet nur. Ist die Ausgabe des Textes beendet, kann er sich anderen Aufgaben zuwenden. Und plötzlich kommt das Problem in Form einer Kante und ca. 80 cm nach unten. Ist er mit der Textausgabe fertig, kann er den Abgrund erkennen und die Motore stoppen. Erkennt er den Abgrund nicht, besteht die Gefahr eines Absturzes an der Kante und schwerer technischer Schäden.

4. Wie können wir Anweisungen fast gleichzeitig ausführen?

Wollen wir mehrere Sachen fast gleichzeitig ausführen, gibt es Probleme mit der Zeit. Nehmen wir als Beispiel:

- **Eine Taste abfragen und damit eine LED schalten**
- **Eine LED mit ca. 2 s blinken lassen**
- **Eine LED mit ca. 0,6 s blinken lassen**

4.1. Einfacher Ansatz (*)

Unten sehen wir einen Ausschnitt aus dem Programmes **PRG_MTK_2**. Wie man sieht, werden

innerhalb der Hauptschleife die einzelnen Unterprogramme aufgerufen. (Ohne Unterprg)

```
while(1)
 {
  taster = taste_lesen ();          // Aufruf UPrg taste lesen und Rückgabe Wert
  led_taster (taster);              // Aufruf UPrg led taster mit Wert
  led_blinken1();                   // Aufruf UPrg led blinken 1 mit 2 s
  led_blinken2();                   // Aufruf UPrg led blinken 2 mit 0,6 s
 }
```

Wenn man das Programm nun laufen lässt, wird man feststellen dass (bitte mit **MTK 2** testen)

- es sehr langsam reagiert und eine „Ewigkeit" vergeht bis der Taster reagiert
- die Abarbeitung der Timer für das blinken nacheinander erfolgt und die LED damit sehr komisch laufen

Dieser Ansatz ist also untauglich für uns. Egal wie schnell unser AVR auch ist, es reagiert sehr langsam. Sehen wir uns einmal den Grund für unser Problem an. Es werden innerhalb der **while** Schleife immer wieder die folgenden Unterprogramme ausgeführt:

```
taster = taste_lesen ();          →   verzögert so lange Taste gedrückt
led_taster (taster);              →   schaltet LED direkt um
led_blinken1();                   →   LED blinkt, verzögert 2 x 2 s
led_blinken2();                   →   LED blinkt, verzögert 2 x 0,6 s
```

Damit haben wir beiden jedem Durchlauf des Programmes eine Verzögerung von ca. 5 bis 6 s (vielleicht auch mehr, durch den Taster). Diese Zeit ist für den Prozessor eine Ewigkeit.

Sehen wir uns einmal die betreffende Stelle im Programm an.

```
void led_blinken1 ()                // UPrg led blinken 1
 {                                  // Klammer Anfang
  delay(2000);                      // Pause 2 s
  leds_set_status(1, 5);            // setzt LED 5 auf rot
  delay(2000);                      // Pause 2 s
  leds_set_status(2, 5);            // setzt LED 5 auf grün
 }                                  // Klammer Ende
```

In diesem Unterprogramm finden wir unsere grössten Bremsen, 2 x delay (2000). Das sind allein rund 4 Sekunden. In den anderen Teilen finden wir noch mehr. Damit ist diese Version für uns nicht geeignet.

Warten verboten!

Die Anweisung delay sollten wir nur selten verwenden !

In Abhängigkeit des Programmes kann man die Anweisung „delay" nutzen. Dabei ist es aber notwendig genau abzuwägen, wo diese sinnvoll ist. Solange ich keine Zeitkritischen Prozesse habe oder Stromsparen muss, kann ich diese Funktion verwenden.

4.2. Verbesserter Ansatz (*)

Will man mehrere Dinge gleichzeitig bearbeiten, muss man die Aufgaben in kleinste Häppchen

zerteilen. Diese kleinsten Häppchen werden dann verschachtelt abgearbeitet, also ein Häppchen von Aufgabe A, ein Häppchen von Aufgabe B, ein Häppchen von Aufgabe C, usw.
Sehen wir uns einmal das Programm **PRG_MTK_3** an. Eigentlich macht es dasselbe wie das Programm **PRG_MTK_2**, eigentlich …
Richtig gesehen, läuft einiges anders. Fangen wir wieder mit dem Aufruf der Unterprg. an.

```
while(1)
 {
     taster = taste_lesen ();          // Aufruf UPrg taste lesen und Rückgabe Wert
     led_taster (taster);              // Aufruf UPrg led taster mit Wert
     led_blinken1();                   // Aufruf UPrg led blinken 1 mit 2 s
     led_blinken2();                   // Aufruf UPrg led blinken 2 mit 0,6 s
     led_blinken3();                   // Aufruf UPrg led blinken 3 unterschied
     delay(1);                         // einziges delay im Programm
 }
```

Innerhalb der while Schleife ist die Anweisung delay (1) dazu gekommen. Ich kann schon einige Leute hören, was die so sagen. Habe doch gesagt: **Warten verboten !** Stimmt genau, hab ich gesagt. Es ist aber notwendig, sich das gesamte Programm anzusehen, bevor man etwas sagt. Eines vorweg dazu. Das Programm **PRG_MTK_3** läuft ganz hervorragend. Alle 3 LEDs werden entsprechend den gewählten Zeiten ein- und ausgeschaltet. Der Taster reagiert sofort und auch diese LED schaltet sofort. Warum eigentlich?
Fangen wir mit der **while** Schleife an. In ihr werden die einzelnen Unterprogramme aufgerufen. **Taste lesen ( )** bis **led_blinken3 ( )**. Zu **delay (1)** kommen wir später. Es erfolgt dadurch ein Aufruf von Unterprogrammen, ein Rücksprung zum Hauptprogramm und wieder der Aufruf eines Unterprogrammes usw. Sehen wir uns mal ein Unterprogramm an z.B. **led blinken1()**

```
( 1 )  // *** led blinken 1 ***
( 2 )  void led_blinken1 ()                  // UPrg led blinken 1
( 3 )   {                                    // Klammer Anfang
( 4 )      led1++;                           // Zähler led 1
( 5 )      if (led1==1999)                   // Abfrage Zähler
( 6 )      leds_set_status(1, 5);            // setzt LED 5 auf grün an
( 7 )        else
( 8 )          {
( 9 )            if (led1==3999)             // Abfrage Zähler
( 10 )            {
( 11 )              leds_set_status(0, 5);   // setzt LED 5 auf grün aus
( 12 )              led1=0;                  // setzt Zähler led 1 auf 0
                  }
               }
         }                                   // Klammer Ende
```

In der **Zeile 2** wird das Unterprogramm gestartet. Bei jedem Durchlauf wird in der **Zeile 4** zum Zähler **led1** ein Betrag von 1 addiert. In der **Zeile 5** wird die Höhe des Zählers **led1** mit dem eingestellten Wert von 1999 verglichen. Entspricht der Zähler dem eingestellten Wert, wird **Zeile 6** ausgeführt und die **LED 5** eingeschaltet. Wenn nicht wird durch die **Zeile 7** die **Zeile 9** ausgeführt. Wieder wird der Zähler **led1** mit dem eingestellten Wert von 3999 verglichen. Stimmen Zähler und Wert überein, wird **Zeile 11** ausgeführt, die LED 5 ausgeschal-

-tet und der Zähler **led1** auf 0 gesetzt. Durch die beiden (1999 + 3999) Zahlen wird die Länge der Zeit bestimmt.

Und damit kommen wir zu **delay(1)**. Mit dieser kurzen Pause wird der gesamte Durchlauf eines Programm Zyklus auf mindestens 1 ms gesetzt. Innerhalb einer Zeit von 1 ms und der notwendigen Zeit für die Anweisungen wird das gesamte Programm durchlaufen. Während dieser Zeit erfolgt das schalten der Zähler (+/-), das Einlesen der Eingänge und das setzen der Ausgänge. Damit kann ich, bei einer eingestellten Zeit von 1999, von einer Laufzeit von z.B. 1999 ms ausgehen. Das entspricht ca. 2 Sekunden und damit meiner gewünschten Blinkzeit.
Bei jedem Durchlauf kann ich mehrere Zähler setzen und damit verschiedene Aufgaben (fast) gleichzeitig ausführen. Man könnte **delay** auch ganz weglassen. Dann hat man aber andere Probleme. Der Durchlauf erfolgt so schnell, das bei den eingestellten Zeiten, die LED Dauerleuchten, da kein Unterschied mehr zwischen ein und aus zu sehen ist.
Die einzelnen kleinen Häppchen sind verdaulicher als die grossen. Die maximale Durchlaufzeit der einzelnen Funktionen ist drastisch reduziert. Anstatt in der LED-Ausgabe einmal 2000 ms zu warten wird nun 2000 x 1ms gewartet. Zwischendurch werden aber 2000 mal die anderen Prozesse bearbeitet. Echte Demokratie sozusagen.
Das ist eigentlich der ganze "Trick" eines kooperativen Multitaskings. Auch wenn die Verwendung von delay(1) noch ein kleiner Schönheitsfehler ist, den die Profis lieber mit einem **Timer** erledigen, so wird das Prinzip klar.

- **Prozesse eines kooperativen Multitaskingsystems warten nicht auf das Eintreten von Ereignissen, sondern bearbeiten nur bereits eingetretene Ereignisse.**
- **Grössere Aufgaben werden in kleine Teilaufgaben zerlegt, welche nur durch mehrfaches Aufrufen der Funktion abgearbeitet werden.**
- **Prozesse eines kooperativen Multitaskings haben eine garantierte, maximale Durchlaufzeit, welche möglichst klein ist.**

Damit ähneln die Prozesse einem **Interrupt,** auch wenn sie als ganz normale Funktionen ausserhalb eines Interrupts ausgeführt werden. An diesem Beispiel erkennt man die Vor- und Nachteile des kooperativen Multitaskings

Vorteile

- einfacher Scheduler mit geringster CPU Belastung und Speicher
- Deterministische Arbeitsweise, damit einfach prüfbar und strenges Timing möglich

Nachteile

- eine andere Programmierweise zur Zerlegung größerer Aufgaben in kleine Teilaufgaben muss manuell vorgenommen werden
- Alle Häppchen werden gleich oft aufgerufen und nicht nur bei Bedarf

Wichtigster Grundsatz ist die Herangehensweise! Viele Programmieranfänger haben damit Schwierigkeiten (ich auch), was u.a. an den schlecht vermittelten (gelernten) Grundlagen liegt.

4.3. Laufschrift

Kommen wir zu einer neuen Aufgabe. Wie kann ich eine Laufschrift programmieren?
Zu Anfang hat mir diese Frage viele Probleme gemacht. Dabei ist die Aufgabe relativ leicht.
Sehen wir uns einmal als Beispiel das Unterprogramm **text_lauf** aus **PRG_MTK_4** an:

```
( 1 )   void text_lauf ()                          // Text unten von links nach rechts
( 2 )    {
( 3 )      lauf1++;                                // Zähler lauf 1
( 4 )      if (lauf1 ==250)                        // Abfrage Zähler lauf1 / Zeit einstellung
( 5 )        {
( 6 )          lauf1=0;                            // setzt lauf1 auf 0
( 7 )          textre++;                           // Zähler textre
( 8 )          gfx_move(textre, 54);               // Angabe Ort unter Logo
( 9 )          gfx_print_text(" N I B O  2");      // Ausgabe Nibo 2
( 10 )         if (textre == 125)                  // wenn text 125 ist geht auf 0
( 11 )           {
( 12 )         textre = 0;                         // setzt textre auf 0
( 13 )           }
( 14 )       }
( 15 )   }
```

Diese 15 Zeilen Programm reichen dazu aus. Sehen wir uns die Funktion einmal genauer an. In der **Zeile 1** erfolgt der Start des Unterprogrammes. In der **Zeile 3** wird die Variable **lauf1** um jeweils 1 bei jedem Durchlauf hochgezählt. Erreicht sie den Wert von 250 (**Zeile 4**), erfolgt der Start des Programmes ab der **Zeile 5** und **lauf1** wird in der **Zeile 6** auf **0** gesetzt. Der Aufruf dieses Programmteils erfolgt ca. alle 250ms. Bei jedem Aufruf wird der Zähler **textre** um jeweils 1 erhöht (**Zeile 7**). Dieser Wert wird in der **Zeile 8** genutzt um eine Position für meine Schrift anzugeben. Der Wert 54 ist dabei der vertikale Wert der Position. In der **Zeile 9** steht der Text, der ausgegeben werden soll. Dabei ist neben dem N von Nibo eine freie Stelle in der Schrift. Dadurch wird links von der Schrift jeweils eine Stelle gelöscht und **Nibo 2** wandert scheinbar nach rechts. Dabei wird in Wirklichkeit der Text überschrieben und dabei jeweils eine Stelle nach rechts verrückt. Durch die freie Stelle im Namen Nibo, wird auch das linke Zeichen gelöscht. Erreicht die Variable **textre** den Wert von 125 (**Zeile 10**), wird das Programm ab der **Zeile 11** gestartet. Dadurch wird die Variable **textre** in **Zeile 12** auf 0 gesetzt und das ganze beginnt von vorn.

Im **PRG_MTK_5** habe ich zusätzlich noch eine Laufschrift eingebaut, wobei der Zähler rückwärts läuft. Zusammen mit einer „Verriegelung" und einer andern Positionsangabe läuft die Laufschrift dabei von links nach rechts und wieder zurück oder umgekehrt. Eine Umstellung der Laufrichtung kann dabei einfach durch Tausch von 0 und 1 am Anfang des Programmes erfolgen. Damit sind wir mit diesem einfachen Multitasking am Ende. Sehen wir uns als nächste einmal einen Timer und Interrupt an.

5. Timer

Timer sind selbstständige Zähler im Prozessor. Man braucht sie dort, wo Zeitkritische und genaue Aufgaben gefordert werden, z.B. Zeitmessung.
Ein Timer ist ein Zähler, der anfängt von Null an hochzuzählen, bis er seinen Höchstwert oder einen eingestellten Vergleichswert erreicht hat. Dieser Höchstwert oder Vergleichswert ist abhängig von der Menge der sogenannten Bits, die für die Anzahl der Speicherstellen stehen. Wir haben in unserem ATmega 128 8- und 16- Bit Timer.

- **Timer mit 8 Bit – gehen von 0 bis 255 (256)**
- **Timer mit 16 Bit – gehen von 0 bis 65535 (65536)**

Bei jedem Überschreiten des Höchstwertes oder Erreichen eines Vergleichswertes hat man die Möglichkeit einen sogenanntens Interrupt auszulösen. Man kann ihn also anweisen in ein Unterprogramm zu springen und dieses abzuarbeiten.
Es gibt verschiedene Betriebsarten. So kann er z.B. zwei Werte vergleichen, oder hoch zählen bis zu einem bestimmten Wert und dann wieder rückwärts zählen. Die Art (Mode) wie der Timer zählt, wird ihm durch ein Befehl mitgeteilt.
Der Prozessor wird durch einen Quarz mit 16 MHz getaktet. Das sind 16 000 000 Takte in der Sekunde. Bei dieser Frequenz würde der Timer sehr schnell zum Ende kommen und einen Interrupt auslösen.
Damit der Timer bei einer Taktfrequenz von 16 MHz nicht zu schnell einen Interrupt auslöst, haben wir „Prescaler". Das sind Vorteiler, die den Systemtakt durch festgelegte Werte teilen und den Timer langsamer laufen lassen. Es sind z.B. Teilungen von 8, 64, 256 und 1024 möglich.

5.1. Welche Timer habe ich im Nibo 2 ?

Im ATmega 128 habe ich die folgenden Timer:

- **Timer 0 – 8 Bit – frei verwendbar**
- **Timer 1 – 16 Bit – verwendet für PWM und Clock**
- **Timer 2 – 8 Bit – frei verwendbar**
- **Timer 3 – 16 Bit – frei verwendbar**

Ich kann also die Timer 0, 2 und 3 verwenden. Der Timer 1 wird bereits durch interne Programmierung (PWM) benutzt.

5.2. Betriebsmodi

Die AVR-Timer können in unterschiedlichen Betriebsmodi betrieben werden. Diese sind:

- **Normaler Modus**
- **CTC Modus**
- PWM (gehen wir nicht drauf ein)

5.3. Mode 0 (Normal Modus)

In diesem Mode zählt der Timer immer von Null an aufwärts bis 255 beim 8-Bit Timer oder 65535 beim 16-Bit Timer. Nachdem er seinen Höchstwert erreicht hat fängt er wieder bei null an. Bei Bedarf kann bei jedem Überlauf ein Interrupt ausgelöst werden. Das heißt, dem Timer kann gesagt werden, das er bei jedem Überlauf eine Routine anspringen soll um diese dann abzuarbeiten. Das hat Priorität vor dem Ablauf der eigentlichen Programmschleife.

5.4. Mode 2 (CTC – Modus)

Hier zählt der von null an aufwärts. Es besteht die Möglichkeit einen Vergleichswert vorzugeben, bis zu dem der Timer zählen soll. Dann stellt er sich selbstständig wieder auf null und beginnt wieder von null an zu zählen. Er muss nicht zwangsläufig bis 255 oder 65535 zählen.

5.5. Änderungen im RegisterTCCR0

Bit	7	6	5	4	3	2	1	0
Name	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
Wert	0	0	0	0	1	0	1	1

Wenn wir die Bits 0, 1 und 3 verändern, indem wir die 1 eintragen, so werden die 16 MHz durch 64 geteilt und CTC ausgeführt.

5.6. Timer 2

Um Timer 2 einzustellen bedienen wir uns der sogenannten Register. Das sind Speicherplätze in denen wir die Art und Funktion der Timer konfigurieren können. Wir haben die folgenden Register zur Verfügung:

- **TCNT2** – Timer/Counter Register (Zählt von Null aufwärts)
- **OCR2** – Output Compare Register (Obergrenze für den Zähler TCNT2)
- **TCCR2** – Timer/Counter Control Register (Gibt die Funktionsweise des Zählers an)
- **TIMSK** – Timer/Counter Interrupt Mask (Schaltet einzelne Interrupts ein/aus)

5.7. Unser Timer

Mit der Erklärung können wir nun die einzelnen Funktionen unseres Timers ermitteln.

```
void nibo_timer2()                                  // Timer 1ms
  {
      TCNT2 = 0;                                    // zählt von 0 aufwärts
      OCR2=249;                                     // Obergrenze für den Zähler
      TCCR2=(1<<WGM21)|(1<<CS21)|(1<<CS20);         // Angabe der Funktionsweise
      TIMSK |= (1<<OCIE2);                          // schaltet einzelne Interrups
  }
```

Zeitrechnung: Gesuchte Frequenz / Zeit = **1000** Hz entspricht **1** ms

$$\frac{16\ MHz\ (\ 16\ 000\ 000\ Hz\ )}{64\ (\ Prescaler\ )} = 250\ 000$$

Bei einer Quarzfrequenz von 16 MHz (16 000 000 Hz) und einem eingestellten Prescaler (Vorteiler) von 64 ergeben sich 250 000.
Als nächsten wird der errechnete Wert durch unsere Frequenz geteilt.

$$\frac{250\ 000}{1000\ Hz} = 250$$

Damit ergibt sich ein Wert von 250. Eingestellt wird 249, denn der Zähler läuft von 0 bis 249, das sind 250 Takte. Durch unseren Timer wird jede Millisekunde ein Interrupt ausgelöst

6. ISR –Was ist das ?

Bei Mikrocontrollern werden Interrupts z. B. ausgelöst wenn:

- sich der an einem bestimmten Eingangs-Pin anliegende Pegel ändert
- eine vorher festgelegte Zeitspanne abgelaufen ist (Timer)
- eine serielle Übertragung abgeschlossen ist
- eine Messung des Analog-Digital-Wandlers abgeschlossen ist

Bei der Behandlung des Interrupts wird das Anwendungsprogramm unterbrochen, das auslösende Interruptflag gelöscht und ein Unterprogramm, die sogenannte **I**nterrupt **S**ervice **R**outine (**ISR**), aufgerufen. Wenn dieses beendet ist, läuft das Anwendungsprogramm ganz normal weiter. Für die Auswertung des Interrups nutze ich die folgenden Zeilen

```
ISR (TIMER2_COMP_vect)                    // wait1=1ms,
  {
      if ( wait<=5)                       // Takt-Zeiteinstellung
        {   wait++;   }                   // erhöht
      else                                // wenn dann ...
        {
            wait=0;                       // setzt wait auf 0
            wait1=1;                      // Signal          usw. ...
```

Um den Takt einstellen zu können, verwende ich die Variable wait. Bei jedem Aufruf wird wait um 1 erhöht. Hat wait den Wert von 5 erreicht, setze ich wait1 auf 1 und wait wieder auf 0. Wait1 kann ich jetzt im Programm weiter nutzen. Dadurch kann man die Geschwindigkeit der Verarbeitung anpassen.
Innerhalb der ISR befindet sich auch die Routine zum Entprellen der Taster. Ich habe diese von Peter Dannegger übernommen. (http://www.mikrocontroller.net/topic/tasten-entprellen-bulletproof) Was mir besonders daran gefällt, ich kann damit bis zu 4 Tasten entprellen und Auslesen. Dabei sind auch verschiedene Arten des Auslesens möglich. z.B. ob eine Taste kurz oder lang gedrückt wird.
Damit kann ich in meinem Programm **PRG_MTK_7** den Taster S3 verschieden nutzen. Beim ersten drücken (**kurz**) wird eine LED eingeschaltet. Beim zweiten drücken (**lang**) wird diese LED wieder ausgeschaltet.

In den folgenden Zeilen erfolgt die Auswertung des Tasters S3. In Abhängigkeit der Auswertung erfolgt dabei das Ein- oder Ausschalten einer LED.

```
if(get_key_short(1<<KEY_1))               // Abfrage Taster S3
  {     leds_set_status(1,3);   }         // Schaltet LED ein

if(get_key_long(1<<KEY_1))                // Abfrage Taster S3
  {     leds_set_status(0,3);   }         // Schaltet LED aus
```

Es kann durch eine Kombination der unterschiedlichen Abfragen und nachfolgenden Schaltungen auch verschiedene Funktionen ausgeführt werden.
Durch die Einstellung Takt kann ich die Geschwindigkeit bei der Auswertung verändern. Mache ich es noch schneller als 5, habe ich Probleme mit dem drücken des Tasters, geht zu schnell. Mache ich es langsamer als 9, dauert es zu lang.

7. Menuesteuerung

Ein besonderes Thema ist eine Menuesteuerung. Ich hatte es in einem früheren Beitrag bereits veröffentlicht. Damals allerdings mit zusätzlichen Tasten. Versuchen wir es einmal an einem anderen Beispiel.
Diesmal verwende ich nur den Taster **S3**. Dieser Taster hat im Programm 3 verschiedene Aufgaben. Im Startbildschirm soll er einfach nur das Menueprogramm starten. Dazu reicht ein einfaches **press** aus. Im nächsten Bild soll er durch ein unterschiedliches drücken den Cursor verändern (**hoch und runter**) und zusätzlich noch meine Auswahl bestätigen. Dazu verwende ich die Funktion **kurz / lang**. Bei einem kurzen Druck wandert der Cursor weiter. Ist er an der Unterkante angekommen, springt er wieder nach oben.
Möchte ich meine Auswahl bestätigen, muss ich länger drücken. Das Programm springt sofort auf die vorgesehen Seite. Dort kann ich dann mit meinem eigenen Programm weitermachen.

Die **LED 7** blinkt während der gesamten Zeit. Das ist die Kontrolle zur Funktion des Multitasking. So lange sie blinkt, ist alles ok. Wenn nicht, gibt es ein Problem.
Das gesamte Programm habe ich über Unterprogramme verbunden und viele Kommentare drin. Ihr findet es unter Roboter.cc bei **PRG_MTK_8**. Sehen wir uns den Ablauf mal genauer an.

Mit diesen zwei Zeilen rufe ich die Unterprogramme für das Titelbild und das Logo auf.

```
nibo_titelbild();                    // Aufruf titelbild ohne Position
nibo_logo(60,5);                     // Aufruf logo mit Position
```

Innerhalb der while Schleife erfolgt als erstes das setzen des wait_merkers. Beim Testen des Programmes, habe ich festgestellt, das manchmal die Impulse verschluckt werden. Damit das nicht vorkommt, nutze ich wait_merker. Am Ende des Programms setze ich ihn wieder auf 0.

```
if (wait1 == 0xFF)                   // Abfrage wai1 - Merker
   {     wait_merker=1;              // Beginn der Zeitschleife
         wait1=0;    }               // wait1 auf 0
```

Danach erfolgt die Abfrage vom Taster **S3** mit press (Gedrückt ?) Ist **S3** gedrückt erfolgt der Aufruf von Menue 1 und 2. Damit erfolgt der Grundaufbau und Aufbau Auswahl.

```
if(get_key_press(1<<KEY_1))          // Taster auf Nibo S3 - Key 1
   {     nibo_TasterS3();    }       // Aufruf UPrg

nibo_menu1();                        // Menue Grund
nibo_menu2();                        // Menue Auswahl
```

Den Aufruf der Taster, mit press und kurz / lang habe ich mit flaga verriegelt. Beim ersten Einschalten ist nur die Taste press möglich. Ist man innerhalb der Auswahl, ist nur noch kurz / lang möglich. Durch das Betätigen von **S3 kurz** wandert der Cursor von oben nach unten. Ist er unten, springt er nach oben. Dazu nutze ich das folgende Stück Code:

```
if(get_key_short(1<<KEY_1))                  // Taster S3 kurz
                {
                        nibo_zeiger();       // Aufruf UPrg nibo_zeiger
                        zeiger=zeiger+10;    // erhöht zeiger um 10
                        if (zeiger>42)       // wird zeiger grösse als 42
                        zeiger=12;           // dann auf 12
                        gfx_move(3,zeiger);  // Ort zeiger
                        gfx_print_text("==>"); // Bild Zeiger
                }
```

Hierbei wird die Position des Zeigers als erstes auf den Wert von 12 gesetzt. Durch drücken von **S3 kurz** erhöhe ich jeweils um 10 (12,22,32,42). Ist der Wert grösser 42, wird er wieder auf 12 gesetzt.
Kommen wir zur Auswertung. Drücke ich den Taster **S3 lang** erfolgt das Enter (Bestätigung) meiner Auswahl und der Aufruf eines Unterprogrammes. Dort steht noch nicht viel drin, ausser Auswahl A (B,C,D). Welche Auswahl ausgewählt wurde erfolgt durch die Auswertung der Variablen zeiger.

```
if(get_key_long(1<<KEY_1))                   // Taster S3 lang
                {
                        nibo_auswahl(zeiger);// Aufruf UPrg nibo_auswahl
                }
```

Bleibt noch das Blinklicht. Das nutze ich zur Kontrolle der Funktion des Programmes. In dem nachfolgenden Teil verwende ich wieder die bekannte Blinkschaltung. Bei jedem **wait_merker** wird der Zähler **led7** erhöht. Beim Erreichen von 35 wird LED 7 ein- oder ausgeschaltet. Dabei erfolgt die Umschaltung durch die Anweisungen if und else. In Abhängigkeit vom aktuellen Zustand („**PINC&(1<<PC7)"**) wird ein- oder ausgeschaltet. Das nennt man „**toggeln**".

```
// LED Ansteuerung 7 grün Blinklicht, Kontrolle ob Zeit ok
        if (wait_merker == 1)
         {
           led7++;                          // Zähler led 7
           if (led7 == 35 )                 // Einstellung Zeit
            {
              led7 = 0;                     // Led 7 auf 0
              if(!(PINC&(1<<PC7)))          // Abfrage PortC PC5 LED 5 Grün aus?
              leds_set_status(1,7);         // schaltet LED 5 Grün auf ein
              else                          // oder …
              leds_set_status(0,7);         // schaltet LED 5 Grün auf aus
            }
         }
        wait_merker = 0;                    // Wait_merker auf 0 setzen
```

8. Programmliste

PRG_MTK_1 – LED blinken mit delay. Es kann jeder versuchen, es zu ändern.
PRG_MTK_2 – erster Versuch eines Multitasking, sehr langsam, ungeeignet
PRG_MTK_3 – erstes Programm mit Multitasking, LED blinken korrekt
PRG_MTK_4 – Multitasking mit LED und Laufschrift
PRG_MTK_5 – Multitasking mit LED und mehrfache Laufschrift
PRG_MTK_6 – mit Timer werden LED angesteuert
PRG_MTK_7 - mit Timer und Tastenauswertung S3 kurz und lang
PRG_MTK_8 – Menuesteuerung mit Timer und Tastenentprellung
PRG_MTK_9 – LED blinken mit delay 1ms, Lösung1 zu MTK 1
PRG_MTK_10 – LED blinken mit Timer 1ms, Lösung2 zu MTK 1

In diesem kleinen Tutorial sind wir am Ende. An einigen Beispielen habe ich euch gezeigt, wie man es machen kann. Sicherlich gibt es auch andere Wege zum Ziel. Jeder kann seinen Weg nutzen. Es sind auch andere Sachen möglich, z.B. uart und die Übertragung von Daten über XBEE, doch das ist alles ein anderes Thema.
Bei unseren einfachen Programmen, aber auch der Linienverfolgung kann ich damit eine erhebliche Steigerung der Geschwindigkeit und Sicherheit erreichen. Probiert es einmal aus.

Die Programme könnt ihr auf Roboter.cc laden und aus-probieren. Ich weise darauf hin, dass die Nutzung der Programme auf eigenes Risiko erfolgt.

Viel Spass beim lesen und Programmieren

Achim