PC &
ELEKTRONIK

Prof. Dr. Ernst Forgber

Multitasking mit AVR RISC-Controllern

Lösungsansätze und praktische Beispiele für Multitasking-Programme

· Entwicklung eines echtzeitfähigen, prioritätsgesteuerten, präemptiven Tasking-Systems
· Von der einfachen Programmschleife zu Multitasking- und Echtzeitkonzepten
· Anhand praktischer Beispiele für die Atmel-8-Bit-Controller

FRANZIS

Prof. Dr. Ernst Forgber

Multitasking mit AVR RISC-Controllern

PC &
ELEKTRONIK

Prof. Dr. Ernst Forgber

Multitasking mit AVR RISC-Controllern

Lösungsansätze und praktische Beispiele für Multitasking-Programme

· Entwicklung eines echtzeitfähigen, prioritätsgesteuerten, präemptiven Tasking-Systems
· Von der einfachen Programmschleife zu Multitasking- und Echtzeitkonzepten
· Anhand praktischer Beispiele für die Atmel-8-Bit-Controller

FRANZIS

Bibliografische Information der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte Daten sind im Internet über http://dnb.ddb.de abrufbar.

Herausgeber: Ulrich Dorn
Satz: DTP-Satz A. Kugge, München
art & design: www.ideehoch2.de
Druck: C.H. Beck, Nördlingen
Printed in Germany

ISBN 978-3-645-65270-4

Worum geht es hier?

Mittlerweile gibt es für praktisch alle Mikrocontroller einen leistungsfähigen C-Compiler mit ansprechender Entwicklungsumgebung, mit deren Hilfe man relativ schnell eigene Anwendungen erstellen kann. Aber selbst kleine Programme haben oft mehrere Dinge gleichzeitig zu erledigen: einen Motor steuern, eine Eingangsspannung messen, Taster und Schalter auslesen, LEDs schalten, Ausgaben über die serielle Schnittstelle an einen PC senden und vieles mehr.

Hat ein Programm mehrere Aufgaben praktisch gleichzeitig zu erledigen, lässt sich dieses Multitasking-Problem auf unterschiedliche Arten realisieren. Dieses Buch stellt anhand praktischer Beispiele für die Atmel-8-Bit-Controller mehrere Ansätze für Multitasking-Programme vor. Ausgehend von einer Lösung mit einer einfachen Programmschleife werden Multitasking- und Echtzeitkonzepte erläutert – bis hin zur Entwicklung eines echtzeitfähigen, prioritätsgesteuerten, präemptiven Tasking-Systems.

Aufbau des Buchs

Das Buch ist wie folgt aufgebaut: Im Kapitel »Multitasking, Kontext, Real-Time & Co.« werden einige Begriffe und grundlegende Mechanismen zum Thema Multitasking erklärt. Das Kapitel »Experimentierumgebung« beschreibt Hardware und Software, mit denen die Beispielprogramme entwickelt und getestet wurden.

Einige Besonderheiten der Programmierung speziell auf AVR-Controllern, wie etwa die Verwendung der seriellen Schnittstelle, der Analogwerteingabe und der Hardware-PWM (Pulse-Width Modulation) sowie die Verwendung von Interrupts, werden im Kapitel »AVR-Controller in C programmieren« behandelt.

Das Kapitel »Multitasking, die Erste: Die Minimalversion« beschäftigt sich mit dem »klassischen« Ansatz zum Multitasking, bei dem man auf einen Scheduler komplett verzichtet und alles über Interrupts und eine große Programmschleife zu realisieren versucht. Daran schließt sich die Betrachtung verschiedener Scheduler mit wachsender Komplexität in Theorie und Praxis an.

Ein einfacher prioritätsgesteuerter Scheduler für zyklische Funktionen wird im Kapitel »Multitasking, die Zweite: Ein Scheduler im Eigenbau« vorgestellt. Das Kapitel »Multitasking, die Dritte: Kooperation ist gefragt« beschäftigt sich mit einem kooperativen Scheduler auf der Basis von Dunkels' »Protothreads«. Im Kapitel »Multitasking, die Vierte: Präemptives Tasking« wird schließlich ein vollständiger präemptiver Scheduler mit Prioritätssteuerung vorgestellt.

Die Ausführungen zu den verschiedenen Konzepten erfolgen anhand durchgehender Beispiele. Dabei wird zunächst immer die grundlegende Idee vorgestellt, die dann anhand von praktischen Beispielen konkretisiert und in einem anschließenden Blick auf die Details vertieft wird.

Die Beispielprogramme können unter *www.buch.cd* heruntergeladen werden. Eine Übersicht der verwendeten Softwaremodule und Programme findet sich in Anhang A.3.

Inhaltsverzeichnis

4 Multitasking, die Erste: Die Minimalversion

Dieser erste Ansatz für ein Multitasking kommt ohne ein Tasking-System aus und kann praktisch auf jedem Rechnersystem implementiert werden, auf dem eine Uhr vorhanden ist.

4.1 Die Grundidee

Auf dem AVR-Controller läuft ein C-Programm in einer Schleife, die nur von Interrupts unterbrochen wird. In dieser Hauptschleife werden zyklisch einfache C-Funktionen aufgerufen, die die Tasks realisieren. Eine solche Task-Funktion kann eine Zustandsmaschine enthalten, durch die gesteuert wird, was im aktuellen Aufruf der Task-Funktion zu tun ist. Die Reihenfolge der Task-Aufrufe ist deterministisch durch die Hauptschleife vorgegeben.

Damit alle Tasks arbeiten können, ist es zwingend erforderlich, dass jede Task-Funktion möglichst schnell bis zum normalen Funktionsende durchlaufen wird. Keinesfalls darf eine Task-Funktion blockierend auf z. B. eine Eingabe warten oder etwa eine gewollte Wartezeit durch eine Verzögerungsschleife realisieren. Dies würde auch alle anderen Tasks ausbremsen, die ja erst nach Beendigung der laufenden Task-Funktion starten können.

Der prinzipielle Programmaufbau ist im folgenden (unvollständigen) Listing eines fiktiven Robotersystems dargestellt. Die Software ist in Tasks gegliedert. Dies sind Funktionen, die zyklisch in einer Endlosschleife aufgerufen werden und bei jedem Aufruf komplett durchlaufen. Im einfachsten Fall haben die Funktionen keine Parameter und liefern auch keinen Ergebniswert. Es gibt noch keinen Echtzeitkern, der die Tasks unterbrechen könnte, Interrupts sind aber in gewohnter Weise nutzbar.

```
Listing Anfang
...
static void InitRoboterSystem(void);
static void Task_RS232Radio(void);
static void Task_pwm(void);
static void Task_Navigation(void);
static void Task_command_decoder(void);
...
int main(void)
{   InitRoboterSystem();
    sei();  // enable interrupts
```

```
    while(1)
    {
       Task_RS232Radio();
       Task_pwm();
       Task_Navigation();
       Task_command_decoder();
       ...
    }
    return 0;
}

static void InitRoboterSystem(void)
{
    ...
    InitSystemTime();
    initRS232Radio();
    init_pwm();
    init_Navigation();
    init_command_decoder();
    ...
}
...
Listing Ende
```

Eine Task soll möglichst wenig Rechenzeit verbrauchen, damit die anderen Tasks nicht lange warten müssen. Wenn eine Task eine längere Aufgabe bearbeiten muss, ist diese in kleine Teilaufgaben zu zerlegen. Die Task enthält dann eine Zustandsmaschine, über die gesteuert wird, welche Teilaufgabe im aktuellen Aufruf zu erledigen ist.

Angenommen, die Task `Task_Navigation` habe eine gewaltige Aufgabe zu erledigen, die in drei Teile A, B, C zerlegt werden kann. In Teil A muss auf ein bestimmtes Ereignis gewartet werden, danach werden Teil B und C durchgeführt. Wenn alle drei Teile abgearbeitet sind, beginnt die Task mit Teil A wieder von vorn. Natürlich darf man in Teil A nicht etwa in einer Schleife auf das Ereignis warten, dies würde alle anderen Tasks blockieren. Der (Pseudo-)Quellcode dazu könnte im Prinzip so aussehen:

```
Listing Anfang
void Task_Navigation(void)
{  static enum State_t {part_A, part_B, part_C}
          state = part_A;
   /* store persistent data in static variables..*/
   switch (state)
   { case part_A:
          if(...) /* event occurred? */
          { /* next call shall execute part B... */
            state = part_B;
          }
          else /* still waiting for event...*/
          { state = part_A;
          }
```

```
          break;
     case part_B: /* execute part B... */
          ...
          /* next call will execute part C */
          state = part_C;
          break;
     case part_C: /* execute part C...*/
          ...
          /* next call will execute A */
          state = part_A;
          break;
     default:break;
   }  // switch
   return;
}
Listing Ende
```

Die Zustandsmaschine speichert ihren aktuellen Zustand in der `static`-Variablen `state`. Der Zusatz `static` ist erforderlich, da die Task-Funktion nach dem Verlassen sonst ihren Zustand vergessen würde. Alle Variablen, die den aktuellen Task-Aufruf überdauern sollen, sind mit der Speicherklasse `static` zu definieren.

Bei jedem Aufruf der Task läuft die Task-Funktion bis zu ihrem normalen Ende (Run to Completion, RTC). Im obigen Beispiel wird bei jedem Aufruf ein Zweig der `switch`-Anweisung durchlaufen.

Die Task-Funktion sollte möglichst schnell durchlaufen werden, damit auch andere Tasks Rechenzeit erhalten. Warteschleifen sind verboten, sie sind durch einen Wartezustand zu ersetzen.

4.2 Der praktische Einsatz

Im Folgenden wird Timer0 des AVR-Controllers als Interrupt-Quelle für die Systemuhr verwendet, und es kommt das schon bekannte Modul `avr_systime` zum Einsatz. Dabei wird die Zeit in Ticks gemessen, wobei mit den gewählten Einstellungen gilt: 1 Tick entspricht etwa 1 ms.

4.2.1 Drehzahlregler mit Tachogenerator und Hardware-PWM

Als Beispiel für den Einsatz von Multitasking in der Minimalversion soll hier ein Programm zur Drehzahlregelung eines Gleichstrommotors dienen. Der Sollwert wird über einen Poti vorgegeben, wie im Abschnitt »Analoge Werte messen« beschrieben. Den Istwert liefert ein zweiter Motor, der im Generatorbetrieb als Tachomaschine arbeitet und der mechanisch an den Antrieb gekoppelt ist, wie in der folgenden Abbildung dargestellt.

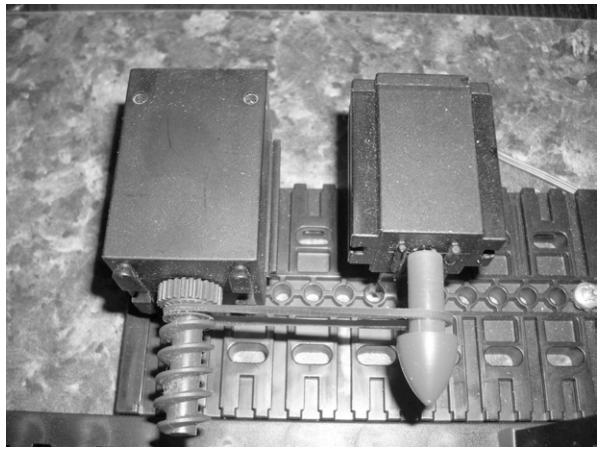

Bild 4.1: Antrieb mit zweitem Motor als Tachogenerator.

Die elektrische Beschaltung des Tachogenerators mit einem einfachen RC-Tiefpassfilter zeigt die untere Abbildung. Die Ansteuerung des Motors erfolgt über die PWM des ATmega8, wie im Abschnitt »PWM« angegeben.

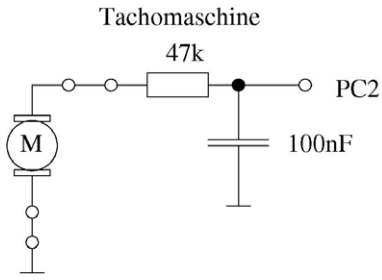

Bild 4.2: Motor in Generatorbetrieb als Tachomaschine.

Das Datenflussdiagramm des Programms in nachfolgender Abbildung zeigt die drei verwendeten Tasks. Die `ADC_Task` liest mit einer Abtastfrequenz von 10 Hz den analogen Sollwert (`setpoint`) und den analogen Istwert (`measurement`) ein. Der Istwert wird zusätzlich über einen digitalen Filter geglättet. Beide gemessenen Werte gelangen in einen globalen Speicher, aus dem die `Controller_Task` und die `Message_Task` lesen.

Die `Message_Task` gibt den Soll- und Istwert zeitgesteuert einmal pro Sekunde am Terminal aus.

Die `Controller_Task` arbeitet synchron mit der `ADC_Task`, gesteuert durch die beiden globalen Hilfsvariablen `newDataAvailable` und `DataIsProcessed`. Beide Tasks bilden ein »Producer-Consumer-Paar«, in dem der Producer Daten erzeugt, die genau einmal vom Consumer verarbeitet werden. Dabei legt der Producer (`ADC_Task`) erst dann neue Daten im Speicher für `setpoint` und `measurement` ab, wenn der Consumer (`Control-`

ler_Task) die Hilfsvariable DataIsProcessed auf 1 gesetzt hat. Umgekehrt liest der Consumer erst dann Daten aus dem Speicher, wenn der Producer die Hilfsvariable newDataAvailable auf 1 gesetzt hat.

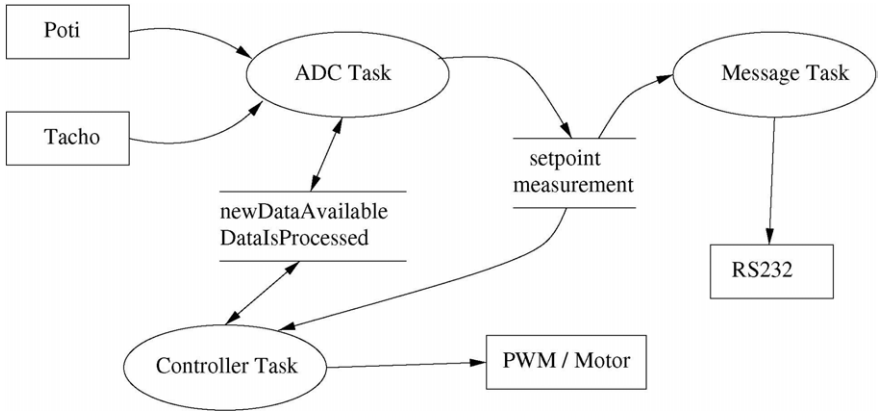

Bild 4.3: Datenflussdiagramm der Drehzahlregelung mit Tachomaschine, Minimalversion.

Es kommen die schon bekannten Module avr_ser für die serielle Schnittstelle, avr_systime für die Systemuhr, avr_adc für die Erfassung analoger Werte und avr_pwm für die Pulsweitenmodulation zum Einsatz (siehe folgende Abbildung). Der Programmcode wird in mehreren Teilen vorgestellt.

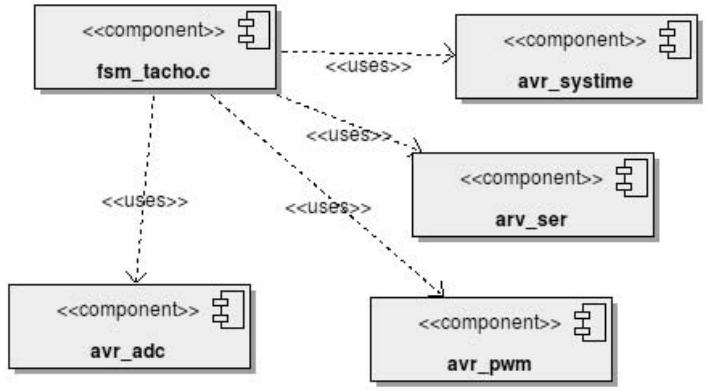

Bild 4.4: Komponentendiagramm der benutzten Module, Drehzahlregler mit Tachomaschine in der Minimalversion.

Der vollständige Quellcode des Programms ist als »Code::Blocks-Projekt« unter fsm_tacho.cbp zu finden.

Das folgende Listing zeigt die globalen Definitionen und das Hauptprogramm. Es gliedert sich in folgende Abschnitte, die auch durch Nummern im Kommentar des Codes gekennzeichnet sind:

(1) Deklaration der Funktionsprototypen für die Initialisierung sowie der Prototypen für die Task-Funktionen.

(2) Definition der globalen Speicher, die für alle Tasks erreichbar sein sollen.

(3) Einmaliger Aufruf von Funktionen zur Initialisierung des Programms.

(4) Hauptschleife, in der zyklisch in fester Reihenfolge die Task-Funktionen aufgerufen werden. Die Hauptschleife läuft mit maximaler Geschwindigkeit, um die Einhaltung der Abtastperiode müssen sich die Tasks selbst kümmern.

```
Listing Anfang
#include <avr/io.h>
#include <avr/interrupt.h>
#include "avr_ser.h"
#include "avr_systime.h"
#include "avr_adc.h"
#include "avr_pwm.h"

#define BAUD 4800UL             /*!< Baudrate RS232 */
#define CPU_CLOCK 16000000UL    /*!< externer 16-MHz-Quartz */
/*-------------------------------------------------*/
// **(1)** static function prototypes
/*-------------------------------------------------*/
static void    _initPortDirections(void);
static void    _initPorts(void);
static void    _startupMessageRS232(void);
static void    _initADC_Task(void);
static void    _ADC_Task(void);
static void    _initController_Task(void);
static void    _Controller_Task(void);
static void    _Message_Task(void);
static uint8_t Filter(uint8_t a);
/*-------------------------------------------------*/
// **(2)** shared variables:
/*-------------------------------------------------*/
static uint8_t setpoint_g=0, measurement_g=0;
static uint8_t newDataAvailable_g=0, DataIsProcessed_g=1;
/*-------------------------------------------------*/
int main(void)
{   _initPortDirections(); // **(3)** Initialisierungen
    _initPorts();
    _initSystemTime();
    _initController_Task();
    sei(); // alle INT freigeben

    serInit(CPU_CLOCK, BAUD);
```

```
    _startupMessageRS232();
    _initADC_Task();

    while (1)                // **(4)** Hauptschleife
    {   _ADC_Task();
        _Controller_Task();
        _Message_Task();
    }
    return 0;
}
Listing Ende
```

Der nächste Teil des Programms umfasst den Quellcode der Funktionen zur Initialisierung der Port-Pins und der Ausgabe einer Startmeldung am Terminal.

An das genutzte Experimentierboard sind zwei Motoren angeschlossen, von denen hier nur einer zum Einsatz kommt. Die Funktion `_initPortDirections()` setzt die entsprechenden Bits der Ausgabe-Pins in den Datenrichtungsregistern auf `1`, während die Funktion `_initPorts()` alle Ausgabe-Pins auf sinnvolle Startwerte einstellt. Die Funktion `_startupMessageRS232()` gibt eine Startmeldung am Terminal aus.

```
Listing Anfang
static void _initPortDirections(void)
{   //DDRA = 0x00;            // Port A  nicht Atmega8
    /* PORTB
    11  pb0  icp1        dir motor0, digital out
    12  pb1  oc1a        enable motor0 (sw PWM), digital out
    13  pb2  /ss, oc1b   dir motor1, digital out
    14  pb3  mosi, oc2   enable motor1 (hw-PWM), digital out
    */
    DDRB |= (1<<3)|(1<<2)|(1<<1)|(1<<0);
    DDRC = 0;
    /* PORTD
    27  pd1  txd         UART, RS232, digital out
    31  pd5  t1          LED1, JP6 gesteckt, digital out
    32  pd6  ain0        LED2, JP7 gesteckt, digital out
    33  pd7  ain1        debugging Ausgabe: System Tick
    */
    DDRD |= (1<<1)|(1<<5)|(1<<6)|(1<<7);
}
/*-------------------------------------------------*/
static void _initPorts(void)
{
    // Motor0 wird hier nicht benutzt, daher abschalten...
    clear_bit(PORTB, 0); // direction motor0
    set_bit(PORTB, 1);   // enable motor0: aus

    clear_bit(PORTB, 2); // direction motor1
    clear_bit(PORTB, 3); // enable motor1: an
```

```
    clear_bit(PORTD, 5); // LED1 aus
    clear_bit(PORTB, 6); // LED2 aus
}
/*-------------------------------------------------*/
static void _startupMessageRS232(void)
{
   serPuts("\n\r\n\r");
   serPuts("************************************\n\r");
   serPuts("* Pollin-Board Atmega8, 16MHz       *\n\r");
   serPuts("************************************\n\r");
   serPuts("FSM01: Task als State Machine\n\r");
}
Listing Ende
```

Im nächsten Teil des Listings geht es um die Initialisierung des Analog-Digital-Wandlers (ADC) mithilfe des Moduls `avr_adc`. Die Funktion `_initADC_Task()` initialisiert den ADC des ATmega8:

(5) Diese Funktion setzt den ADC-Taktteiler. Der Takt des ADC wird aus dem CPU-Takt (hier 16 MHz) abgeleitet. Der ADC-Takt sollte laut Datenblatt zwischen 50 kHz und 200 kHz liegen, hier wurde 125 kHz eingestellt.

(6) Als Referenzspannung wird die an Pin AVCC liegende interne Spannung gewählt.

(7) Diese Funktion stellt den Analogmultiplexer des ADC auf den Kanal 0 ein, an den der Poti angeschlossen ist.

(8) Der ADC wird aktiviert.

(9) Laut Datenblatt dauert die erste Wandlung nach der Aktivierung des ADC etwas länger als normal, daher wird sie bereits in der Startphase des Programms durchgeführt.

(10) Diese Schleife wartet auf das Ende der ersten Wandlung. Solche Warteschleifen sind nur während der Initialisierung erlaubt, eine Task-Funktion darf mit Warten keine Zeit verschwenden.

(11) Die globale Variable `newDataAvailable_g` dient der Synchronisation mit der Controller-Task. Der Wert 0 signalisiert, dass noch keine Messdaten vorliegen.

```
Listing Anfang
static void _initADC_Task(void)
{   adcSetPrescaler(128);                // **(5)**
    adcSelectInternalReferenceVoltage(); // **(6)**
    adcSelectMuxChannel(ADC_MUX_ADC0);   // **(7)**
    adcEnableADC();                      // **(8)**
    adcStartConversion();                // **(9)**
    while(!adcConversionIsDone());       // **(10)**
    newDataAvailable_g=0;                // **(11)**
}
Listing Ende
```

Die Task-Funktion des ADC ist im nächsten Teil des Listings angegeben. Sie wird in der Hauptschleife des Programms aufgerufen, misst die Spannung am Poti (Sollwert) und an der Tachomaschine (Istwert) und rechnet die Spannungen in ein Maß für die Drehgeschwindigkeit um. Diese Task ist zeitgesteuert und liefert mit einer definierten Abtastperiode neue Soll- und Istwerte. Sie synchronisiert sich über zwei globale Hilfsvariablen mit der Controller-Task (Producer-Consumer).

(12) Diese Variable speichert den Zeitpunkt der letzten Messung (in Ticks, 1 Tick: 1 ms) über das Funktionsende hinaus, damit die gewünschte Abtastfrequenz eingehalten werden kann.

(13) Eine Zustandsmaschine steuert die Aktion der Task im aktuellen Aufruf. Diese Variable speichert den inneren Zustand über das Funktionsende hinaus, der Startzustand ist `IDLE`.

(14) Die `switch`-Anweisung realisiert die Zustandsmaschine.

(15) Im Zustand `IDLE` wird anhand der aktuellen Zeit überprüft, ob seit der letzten Messung bereits mehr als die gewählte Abtastperiode von 100 Ticks (entspricht 10 Hz Abtastfrequenz) vergangen ist.

(16) Die Task überprüft zusätzlich, ob die Controller-Task die alten Daten bereits verarbeitet hat (Task-Synchronisation).

(17) Die Messung des nächsten Sollwerts wird vorbereitet, indem der entsprechende Analogkanal gewählt und die Wandlung gestartet wird. Die Task verschwendet aber keine Zeit mit dem Warten auf das Messergebnis, sondern schaltet für den nächsten Aufruf in den Folgezustand `WAIT_POTI`.

(18) Im Zustand `WAIT_POTI` überprüft die Task, ob die Wandlung abgeschlossen ist. Falls nicht, bleibt sie in diesem Zustand und kehrt unverzüglich zur Hauptschleife zurück.

(19) Ist die Messung des Sollwerts abgeschlossen, holt die Task den Messwert ab, speichert ihn in der globalen Variablen `setpoint_g`, startet die Messung des Istwerts an der Tachomaschine und geht in den Zustand `WAIT_TACHO` über.

(20) Im Zustand `WAIT_TACHO` überprüft die Task, ob die Messung an der Tachomaschine abgeschlossen ist.

(21) Der Messwert wird über ein Tiefpassfilter geglättet und so skaliert, dass Soll- und Istwert im Intervall `0..255` liegen.

(22) Die nächste Messung des Sollwerts wird gestartet.

(23) Der Zeitpunkt der letzten Messung wird gespeichert, damit im Folgezustand `IDLE` die Abtastperiode eingehalten werden kann.

(24) Die globale Variable `newDataAvailable_g` dient der Synchronisation mit der Controller-Task. Der Wert `1` zeigt an, dass neue Messdaten vorliegen.

```
Listing Anfang
static void _ADC_Task(void)
{
```

```
    static uint16_t t_1=0;                              //**(12)**
    uint16_t t;
    static enum States {IDLE, WAIT_POTI, WAIT_TACHO}
      state=IDLE;                                       //**(13)**
    uint8_t x;

    switch(state)                                       //**(14)**
    {
        case IDLE:
                t = _gettime_Ticks();
                if((t - t_1) > (uint16_t)100)           //**(15)**
                {
                   if(DataIsProcessed_g)               //**(16)**
                   {  DataIsProcessed_g = 0;
                      adcSelectMuxChannel(ADC_MUX_ADC0);//**(17)**
                      adcStartConversion();
                      state = WAIT_POTI;
                   }
                }
                break;
        case WAIT_POTI:
                if(adcConversionIsDone())               //**(18)**
                {  x=adcGetValue8Bit();
                   setpoint_g=x;
                   adcSelectMuxChannel(ADC_MUX_ADC2); //**(19)**
                   adcStartConversion();
                   state = WAIT_TACHO;
                }
                break;
        case WAIT_TACHO:
                if(adcConversionIsDone())               //**(20)**
                {  x=adcGetValue8Bit();
                   x = Filter (x);                      //**(21)**
                   measurement_g=x;
                   adcSelectMuxChannel(ADC_MUX_ADC0); //**(22)**
                   adcStartConversion();
                   t_1 = _gettime_Ticks();              //**(23)**
                   newDataAvailable_g = 1;              //**(24)**
                   state = IDLE;
                }
                break;
        default: break;
    }

}
Listing Ende
```

Die Spannung an der Tachomaschine enthält einen drehzahlabhängigen Wechselanteil, der auszufiltern ist, bevor der Regler den Istwert verarbeitet. Ferner ist die Tachospannung so zu skalieren, dass sich bei maximaler Antriebsdrehzahl ein Istwert von 255 ergibt. Diese beiden Aufgaben übernimmt die Funktion `Filter`, in der ein digitaler Tiefpassfilter mit Gesamtverstärkung 2 realisiert ist. Ohne auf die Einzelheiten des Filterentwurfs einzugehen, sei hier nur die Differenzengleichung angegeben:

$$y(n) = \frac{1}{2}x(n) + \frac{3}{4}y(n-1)$$

Der aktuelle Filterausgangswert y(n) ergibt sich aus der gewichteten Summe des aktuellen Filtereingangswerts x(n) und des Filterausgangs im letzten Aufruf y(n–1). Um Rechnungen mit Gleitkommazahlen zu vermeiden, arbeitet die Funktion mit Brüchen und Skalierungsfaktoren.

```
Listing Anfang
static uint8_t Filter(uint8_t a)
{  int16_t x;
   static int16_t y_1=0;
   int16_t y;
   int16_t b=1, c=4;

   x = a;   // Typumwandlung Skalierung für Fixpunktarithmetik
   x = x << 4; // Skalierung fuer Fixpunktarithmetik
   y = (b*x + (c-b)*y_1)/c;
   y_1 = y;
   // Skalierung fuer Fixpunktarithmetik rueckgaengig machen und
   y = y >> 3; // umskalieren auf Mass fuer Drehzahl (Faktor 2)
   if (y>255) return 255;  // Saettigung
   else       return (uint8_t) (y&0xFF);  //nur die unteren 8 Bit
}
Listing Ende
```

Im folgenden Teil des Listings sind die Funktion zur Initialisierung des Reglers und die zugehörige Task-Funktion angegeben. Als Regler wird ein einfacher PI-Algorithmus benutzt:

$$y(n) = k_p e(n) + k_i \sum_{j=0}^{n} e(j)$$

Der Regler berechnet aus Soll- und Istwert die Stellgröße und steuert damit über die PWM den Motor an. Diese Task synchronisiert sich mit der ADC-Task: Eine neue Stellgröße wird erst berechnet, wenn neue Messdaten vorliegen. Sind die Messdaten verarbeitet und wurde die neue Stellgröße an die PWM gegeben, erfolgt eine Bestätigung an die ADC-Task.

Auch hier wird zur Vermeidung von Gleitkommazahlen mit Brüchen und Skalierungsfaktoren gerechnet.

(25) Diese Routine initialisiert Timer2 (8 Bit) für den Betrieb im »Fast PWM«-Modus. Der Takt ist der CPU-Takt, geteilt durch einen »Prescaler«, hier ergibt sich ein Takt von 15.625 Hz. Damit läuft die PWM mit einer Periode von etwa 61 Hz. Das Vergleichsregister bekommt den Startwert 255, was maximale Motordrehzahl bedeutet.

(26) Synchronisation mit der ADC-Task: Nur falls neue Messwerte vorliegen, werden diese einmalig verarbeitet.

(27) Berechnung der neuen Stellgröße in Fixpunktarithmetik.

(28) Begrenzung der Stellgröße auf die Werte `0..255`.

(29) Synchronisation mit der ADC-Task: Die Messdaten wurden verarbeitet.

```
Listing Anfang
static void _initController_Task(void)
{
    pwmInitTimer2FastPWM(1024);              //**(25)**
    pwmSetpointOCR2(255);
}
/*-------------------------------------------------*/
static void _Controller_Task(void)
{  int16_t x,w,y,e;
   static int16_t e_integral=0;
   int16_t F=6, Kp_F=12, Ki_F=1;

    if(newDataAvailable_g)                  //**(26)**
    {
        newDataAvailable_g = 0;
        // neue Stellgroesse berechnen...
        x = measurement_g; // Typumwandlung
        x = x << 3; // Skalierung für Fixpunkt Arithmetik
        w = setpoint_g;
        w = w << 3;
        e = w - x;
        e_integral += e;
        y = (Kp_F*e + Ki_F*e_integral)/F;  //**(27)**
        y = y >> 3; // Skalierung  rueckgaengig
        if (y>255) pwmSetpointOCR2(255);   //**(28)**
        else       pwmSetpointOCR2((uint8_t) (y&0xFF));
        DataIsProcessed_g = 1;             //**(29)**
    }
}
Listing Ende
```

Der letzte Teil des Listings zeigt die Message-Task, die einmal pro Sekunde den aktuellen Soll- und Istwert am Terminal ausgibt.

(30) Auch diese Task wird durch eine Zustandsmaschine gesteuert. Die Variable `state` speichert den Zustand über das Funktionsende hinaus für den nächsten Aufruf der Task-Funktion.

(31) Die Task bleibt im Zustand `IDLE`, bis die Task-Periode von einer Sekunde (1000 Ticks) seit der letzten Terminalausgabe verstrichen ist.

(32) Im Zustand `SEND_POTI` erfolgt die Ausgabe des Sollwerts. Die Ausgaben werden auf verschiedene Zustände verteilt, um die Durchlaufzeit der Task-Funktion kurz zu halten.

(33) Ausgabe des Istwerts in einem separaten Zustand.

(34) Speichern des Zeitpunkts der letzten Ausgabe, damit die Task-Periode eingehalten werden kann.

```
Listing Anfang
static void _Message_Task(void)
{   static uint16_t t=0, t_1=0;
    static enum States {IDLE, SEND_POTI, SEND_TACHO}
      state=IDLE;                                    //**(30)**

    switch(state)
    {
        case IDLE:
               t = _gettime_Ticks();
               // Zeit seit letzter Aktivierung...
               if((t-t_1) > (uint16_t)1000)          //**(31)**
               {  state = SEND_POTI;
               }
               break;
        case SEND_POTI:
               serOutUint16Dec(setpoint_g);          //**(32)**
               serPutc(' ');
               state = SEND_TACHO;
               break;
        case SEND_TACHO:
               serOutUint16Dec(measurement_g);       //**(33)**
               serPutc('\r');
               serPutc('\n');
               state = IDLE;
               t_1 = _gettime_Ticks();               //**(34)**
               break;
        default : break;
    }
}

Listing Ende
```

4.3 Zusammenfassung

In dem vorgestellten Ansatz zum Multitasking kommt man ohne einen zusätzlichen Kernel aus, was insbesondere bei knappem Speicher wichtig ist. Tasks sind normale Funktionen, die bei jedem Durchlauf der Hauptschleife aufgerufen werden und bis zu ihrem normalen Funktionsende laufen. Was die Task-Funktion im jeweiligen Aufruf tut, regelt z. B. eine Zustandsmaschine. Dies führt bei periodischen Tasks dazu, dass man in jeder Task-Funktion ähnlichen Code für die Zeitsteuerung benötigt.

Man hat als Programmierer alle Freiheiten, muss aber auch alles selbst implementieren. So ist es etwa über globale Hilfsvariablen möglich, eine Task-Synchronisation (Producer-Consumer) zu realisieren oder über die Abfrage der Uhr verschiedene Task-Perioden zu erreichen.

Ein echter Kontextwechsel findet nicht statt, Task-Funktionen unterbrechen sich nicht gegenseitig. Aufgaben, die eine längere Bearbeitungszeit benötigen, sind manuell in Teile zu zerlegen, damit die Task-Funktion schnell terminiert.

Eine Task wird durch genau eine Task-Funktion repräsentiert. Es gibt nicht die Möglichkeit, mehrere Instanzen einer Task zu verwenden.

Die Minimalversion hat sicherlich einige Vorteile:

- Es gibt keinen Kernel, der zusätzliche Rechenleistung oder Speicher benötigt.
- Das Verfahren ist einfach zu implementieren.
- Es ergibt sich ein deterministisches Zeitverhalten.
- Alle Interrupts (außer einem für die Uhr) stehen zur Verfügung.

Dem stehen aber auch einige Einschränkungen gegenüber:

- Es gibt keine unterschiedlichen Prioritäten für Tasks.
- Alle Tasks werden in der Hauptschleife gleich oft aufgerufen, individuelle Perioden sind für jede Task neu zu programmieren.
- Umfangreiche Aufgaben sind von Hand zu zerlegen.
- Nur eine Instanz einer Task ist möglich.
- Persistente lokale Daten sind nur über die `static`-Variable realisierbar.
- Die Task-Liste ist fix, man kann keine Task löschen oder neu erzeugen.

A Anhang

A.1 Interface-Schaltung für zwei Gleichstrommotoren mit Drehgeber

Die Beispielprogramme sind auf dem »Atmel Evaluation-Board Version 2.0.1« der Firma Pollin und einem ATmega8-Controller mit 16 MHz Takt lauffähig. Die Interface-Schaltung verwendet das IC L293 für die Ansteuerung zweier Gleichstrommotoren. Jeder Motor treibt eine Welle mit einem digitalen Drehgeber an, der aus einer Zahnscheibe und zwei Gabellichtschranken besteht, wie Bild A.1 zeigt.

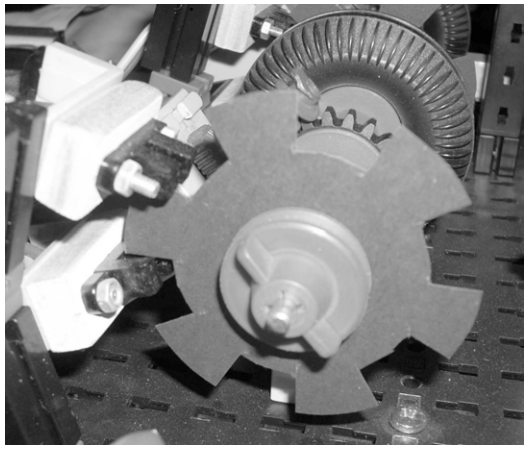

Bild A.1: Digitaler Drehgeber.

Das benutzte Interface besteht aus der Ansteuerelektronik für die Motoren über einen L293 und der Beschaltung der Gabellichtschranken TCST 2103, die die Drehgeber bilden. Sie wird in Bild A.2 dargestellt. Die Ausgangssignale der Gabellichtschranken gelangen über einen Schmitt-Trigger 74HCT14 an den AVR-Controller. Die Schaltung benötigt eine Versorgungsspannung von $Uv = 8V$, aus der die 5-V-Versorgung der Logik-ICs abgeleitet wird.

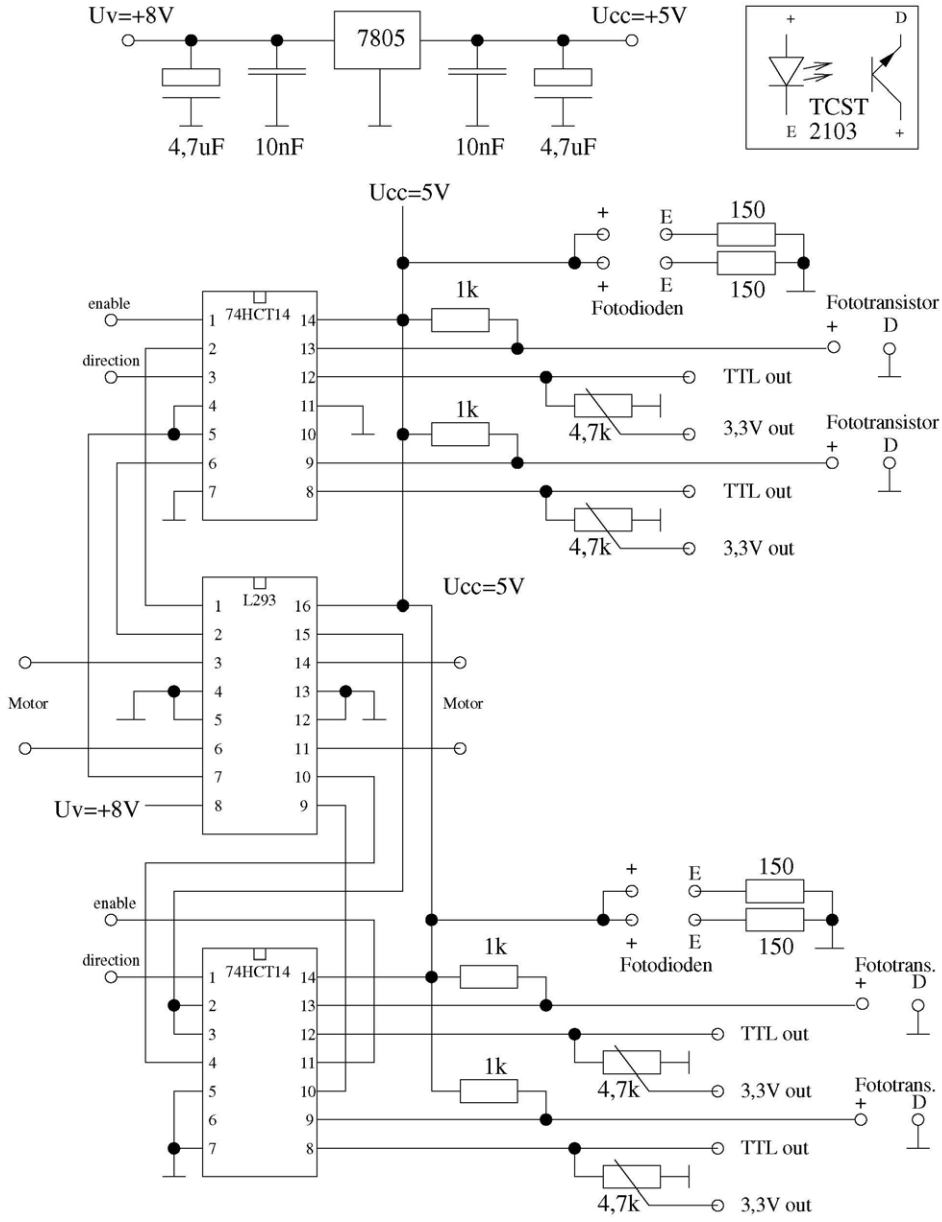

Bild A.2: Interface-Schaltung für die Motoren und die Drehgeber.

A.2 Pin-Belegung des benutzten Evaluation-Boards

Die Interface-Schaltung ist mit dem 40-poligen Wannenstecker J4 des Evaluation-Boards verbunden. Die Tabelle listet die Pin-Belegung auf.

Pin	*Name*	*Signal*	*Verwendung*
1	pc0	adc0	Poti 0, analog in
2	pc1	adc1	Poti 1, analog in
3	pc2	adc2	Tacho, analog in
4	pc3	adc3	LDR, analog in
5	pc4	adc4, sda	Induktionsschleife, analog in
6	pc5	adc5, scl	spare
7			
8			
9			
10			
11	pb0	icp1	dir motor0, digital out
12	pb1	oc1a	enable motor0 (SW-PWM), digital out
13	pb2	/ss, oc1b	dir motor1, digital out
14	pb3	mosi, oc2	enable motor1 (HW-PWM), digital out
15	pb4	miso	Gabellichtschranke2, digital in
16	pb5	sck	Gabellichtschranke3, digital in
17			
18			
19			
20			
21			
22			
23			
24			
25			
26	pd0	rxd	UART, RS232
27	pd1	txd	UART, RS232
28	pd2	int0	Gabellichtschranke0, JP3 offen, digital in
29	pd3	int1	Gabellichtschranke1, JP4 offen, digital in

Pin	Name	Signal	Verwendung
30	pd4	xck,t0	Taster3, JP5 gesteckt, digital in
31	pd5	t1	LED1, JP6 gesteckt, digital out
32	pd6	ain0	LED2, JP7 gesteckt, digital out
33	pd7	ain1	debugging: System Tick, JP8 offen, digital out
34	aref		
35	gnd		Ground
36	vcc	avcc	
37	gnd		
38	vcc	avcc	
39	gnd		
40	vcc	avcc	

Pin-Belegung J4, Pollin-Evaluation-Board, ATmega8.

A.3 Softwaremodule und Beispielprogramme

Die benutzten Softwaremodule und Beispielprogramme sind in jeweils eigenen Verzeichnissen organisiert, siehe auch die unten aufgeführte Tabelle. Zu jedem Softwaremodul enthält das Verzeichnis eine HTML-Dokumentation, die mit dem freien Tool »Doxygen« (*http://www.stack.nl/~dimitri/doxygen*) erzeugt wurde. Den Einstiegspunkt in die Dokumentation bildet jeweils die Datei *index.html*.

Verzeichnis	Inhalt	Kurzbeschreibung
`avr_adc`	Modul	Analog-Digital-Wandler
`avr_cos`	Modul	COS: Scheduler, FIFO, Task-Liste, Semaphoren, Tabellentext
`avr_pwm`	Modul	Timer2 als PWM nutzen
`avr_rtc`	Modul	RTC: Scheduler, FIFO, Task-Liste
`avr_rtos`	Modul	RTOS: Scheduler mit Systemuhr, Semaphoren, Messagebox
`avr_ser`	Modul	RS-232-Schnittstelle
`avr_systime`	Modul	Systemuhr für RTC, COS
`cos_dg_int_pwm`	Prog.	COS, Interrupt, Hardware-PWM

Verzeichnis	*Inhalt*	*Kurzbeschreibung*
cos_fifo_speed_test	Prog.	COS, Geschwindigkeitstest
cos_led_blink	Prog.	COS, blinkende LED
cos_producer_consumer_fifo	Prog.	COS, FIFO, synchrone Tasks
cos_tacho	Prog.	COS, Motor mit Tachomaschine, HW-PWM
cos_test_atmega8	Prog.	COS, Demoprogramm
cos_two_motors	Prog.	COS, 2 Motoren, Software-PWM
fsm_tacho	Prog.	Zustandsmaschine, Motor mit Tacho
pollin_hw_test_adc_potis	Prog.	Test Analogwerteingabe
pollin_hw_test_int0_drehgeber	Prog.	Test ext. Interrupt
pollin_hw_test_pwm	Prog.	Test Timer2 als PWM
pollin_hw_test_RS232	Prog.	Test serielle Schnittstelle
pollin_hw_test_systime	Prog.	Test Systemuhr
rtc01	Prog.	RTC, Motor, Drehgeber, Software-PWM
rtc_dg_int_pwm	Prog.	RTC, ext. Interrupt, HW-PWM
rtc_led_blink	Prog.	RTC, blinkende LED
rtc_tacho	Prog.	RTC, Motor mit Tachomaschine, HW-PWM
rtc_two_motors	Prog.	RTC, 2 Motoren, Software-PWM
rtos_dg_int_hw_pwm	Prog.	RTOS, ext. Interrupt, HW-PWM
rtos_led_blink	Prog.	RTOS, blinkende LED
rtos_speed_test	Prog.	RTOS, Geschwindigkeitstest
rtos_tacho_mb_sram	Prog.	RTOS, Motor mit Tacho, HW-PWM
rtos_test_atmega8	Prog.	RTOS, Demoprogramm
rtos_two_motors	Prog.	RTOS 2 Motoren, Software-PWM

Verzeichnisse mit Softwaremodulen und Beispielprogrammen.

Literaturverzeichnis

- Florian Schäffer: »AVR Hardware und C-Programmierung in der Praxis«, Elektor-Verlag Aachen, 2008.
- Günter Schmitt: »Mikrocomputertechnik mit Controllern der Atmel AVR-Risc-Familie«, Oldenbourg Verlag München, 2008.
- Dieter Werner et. al.: »Taschenbuch der Informatik«, Fachbuchverlag Leipzig, 1995.
- Tom DeMarco: »Structured Analysis and Systems Specification«, Englewood Cliffs, NJ, Prentice-Hall, 1979.
- David Bellin, Susan Suchman: »The Structured Systems Developement Manual«, Englewood Cliffs, NJ, Prentice-Hall, 1990.
- Richard Barry (2004): »Multitasking on an AVR«, *http://www.avrfreaks.net*
- »FreeRTOS«, *http://www.freertos.org*
- Neil Klingensmith: »Helium«, *http://helium.sourceforge.net/index.html*
- Michael Dorin (2008): »Building 'instant-up' real-time operating systems«, *http://www.embedded.com*
- Adam Dunkels et al. (2006): »Protothreads: Simplifying Event-Driven Programming of Memory-Constraint Embedded Systems«, *http://www.sics.se/~adam/dunkels06protothreads.pdf*
- Dimitri van Heesch: »Doxygen«, *http://www.stack.nl/~dimitri/doxygen/*
- »Code::Blocks«, *http://www.codeblocks.org/*
- »WinAVR«, *http://sourceforge.net/projects/winavr/*

Stichwortverzeichnis

Prof. Dr. Ernst Forgber

Multitasking mit AVR RISC-Controllern

Lösungsansätze und praktische Beispiele für Multitasking-Programme

Das Buch stellt anhand praktischer Beispiele für die Atmel-8-Bit-Controller mehrere Ansätze für Multitasking-Programme vor. Ausgehend von der Lösung mit einer einfachen Programmschleife werden Multitasking- und Echtzeitkonzepte erläutert – bis hin zur Entwicklung eines echtzeitfähigen, prioritätsgesteuerten, präemptiven Tasking-Systems.

Die Ausführungen der verschiedenen Konzepte erfolgen anhand durchgehender Beispiele. Dabei wird zunächst immer die grundlegende Idee vorgestellt, die dann mithilfe der praktischen Beispielen konkretisiert und in einem anschließenden Blick auf die Details vertieft wird.

Aufbau des Buchs:

Multitasking, Kontext, Real-Time & Co.
Begriffliche Erklärungen und grundlegende Mechanismen.

Experimentierumgebung
Beschreibung der Hard- und Software.

AVR-Controller in C programmieren
Besonderheiten bei der Programmierung von AVR-Controllern.

Multitasking, die Erste: Die Minimalversion
Klassischer Ansatz zum Multitasking unter komplettem Verzicht auf einen Scheduler und dem Versuch, alles über Interrupts und eine große Programmschleife zu realisieren. Betrachtung verschiedener Scheduler mit wachsender Komplexität in Theorie und Praxis.

Multitasking, die Zweite: Ein Scheduler im Eigenbau
Darstellung der Funktionsweise eines einfachen prioritätsgesteuerten Schedulers für zyklische Funktionen.

Multitasking, die Dritte: Kooperation ist gefragt
Zeigt die Funktionsweise eines kooperativen Schedulers auf der Basis von Dunkels' Protothreads.

Multitasking, die Vierte: Präemptives Tasking
Vorstellung eines vollständig prioritätsgesteuerten RTOS-Schedulers mit Präemption.

Aus dem Inhalt:

Über den Autor:

Prof. Dr. Ernst Forgber lehrt an der Hochschule Hannover an der Fakultät Elektro- und Informationstechnik.
Er arbeitet auf den Gebieten Software-Technik, Echtzeitsysteme und Entwurf digitaler Schaltungen mit VHDL.

9 783645 652704