



FACHHOCHSCHUL-BACHELORSTUDIENGANG

Automatisierungstechnik

Entwicklung von Hardwaretreibern für autonomen Roboter

ALS BACHELORARBEIT EINGEREICHT

zur Erlangung des akademischen Grades

Bachelor of Science in Engineering

von

Krößwang Ridler Thomas

Juni 2011

Betreuung der Bachelorarbeit durch

DI Walter Rokitansky

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt, die den benutzten Quellen entnommenen Stellen als solche kenntlich gemacht habe und dass diese Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

.....
Thomas Krößwang Ridler

Traun, 20.06.11

Kurzfassung

Ziel dieser Bachelorarbeit ist die Entwicklung von Hardwaretreibern auf Basis des Mikrocontrollers ATXmega256 für die Hauptplatine eines autonomen, mobilen Roboters, welcher für den internationalen Roboterwettbewerb „Eurobot“ entwickelt wurde. Aufgabe dieser Hardwaretreiber ist es, den höheren Programmebenen (Strategieprogramm, State-Machine) Routinen zur Verfügung zu stellen, mit deren Hilfe sie auf die Hardware des Mikrocontrollers zugreifen können, ohne direkt auf die mikrocontrollerspezifischen Register zugreifen zu müssen. Dies ermöglicht eine übersichtlichere Programmstruktur und vereinfacht programmtechnische Änderungen aufgrund geänderter Hardware erheblich.

Die Treiber sind speziell für die Hauptplatine der Eurobot 2011 entwickelt, genauere Informationen dazu können in der Bachelorarbeit „Erstellung einer Hauptplatine für autonomen Roboter“ nachgelesen werden, ebenso eine kurze Beschreibung des Roboters sowie die Regeln des Wettbewerbs.

Im ersten Teil der Arbeit sollen kurz die Gründe und Vorteile des Einsatzes von Hardwaretreibern umrissen werden.

Der zweite Teil enthält eine Übersicht über die 3 sich auf der Hauptplatine befindlichen Mikrocontroller, für die die Hardwaretreiber entwickelt werden. Darin enthalten sind eine Übersicht über die Pinbelegung, eine Timerübersicht sowie die benötigten Treiber für jeden Mikrocontroller.

Hauptteil der Arbeit ist die Behandlung der wichtigsten Hardwaretreiber des Roboters. Darin enthalten sind jeweils Funktionsbeschreibung, Einsatzgebiete, Beschreibung der wichtigsten Register sowie Beispielcode.

Schließlich wird noch ein einfaches Steuerprogramm zum Testen der Treiber vorgestellt, welches per USART Befehle vom PC ausführt und Informationen am Bildschirm ausgibt.

0. Inhaltsverzeichnis

1. Einleitung	5
1.1 Abgrenzung und Zielsetzung	5
1.2 Allgemeines zu Hardwaretreibern.....	6
2. Pinbelegung und Timerübersicht der Mikrocontroller	6
2.1 Mikrocontroller 1.....	7
2.2 Mikrocontroller 2.....	8
2.3 Mikrocontroller 3.....	9
3. Hardwaretreiber	10
3.1 Port- und Pinconfiguration	10
3.2 Überlauftimer zur Taktvorgabe	12
3.3 PWM-Timer.....	14
3.4 Analog-Digital-Wandler	17
3.5 Quadraturdecoder	22
3.6 Sensorik.....	24
4. Test der Hardwaretreiber	26
5. Fazit und Danksagungen	28
6. Abbildungsverzeichnis	29
7. Codeverzeichnis	29
8. Literaturverzeichnis	29
9. Anhang	30

1. Einleitung

1.1 Abgrenzung und Zielsetzung

Im Wintersemester 2010/11 sowie im Sommersemester 2011 wurde ein mobiler, autonomer Roboter zur Teilnahme am Robotikbewerb „Eurobot 2011“ entwickelt. Ziel dieses Roboters ist es, die beim Bewerb gestellte Aufgabe innerhalb einer Zeit von 90 Sekunden bestmöglich zu erfüllen um ein Maximum an Punkten zu erreichen. Auf die genauen Regeln des diesjährigen Bewerbs soll in dieser Arbeit nicht weiter eingegangen werden, es sei hierzu auf die Homepage www.eurobot.org verwiesen¹.

Als Inhalt meines interdisziplinären Projekts FUP5 sowie meiner ersten Bachelorarbeit² wurde für diesen Roboter eine Steuerplatine für diesen Roboter entwickelt. Aufgabe dieser Platine ist das Erfassen der Umwelt des Roboters mittels Sensoren, Berechnung von Aktionen und Strategien sowie Durchführung dieser Aktionen mittels der angeschlossenen Aktoren wie Motoren oder Pumpen. Die Rechenleistung dafür wird von 3 Mikrocontrollern des Typs ATXmega256 bereitgestellt, die in die Platine integriert sind. Inhalt dieser zweiten Bachelorarbeit ist die Entwicklung von Hardwaretreibern, mithilfe derer die Kommunikation zwischen dem Mikroprozessor mit interner Hardware (Timer, ADC) sowie externer Hardware (Sensoren, Aktoren, PC) ermöglicht wird.

Ziel dieser Arbeit ist nicht die Aufstellung aller im Roboter zum Einsatz kommenden Treibern, sie beschränkt sich auf die grundlegendsten Treiber, die in einem derartigen Robotersystem zum Einsatz kommen, etwa Routinen zum Einlesen von Sensoren, PWM-Schnittstellen oder Quadraturdecoder.

Obwohl die oben genannte Platine und somit die Hardwaretreiber speziell für den diesjährigen Bewerb entwickelt wurden und die meiste Hard- und Software davon bei Folgebewerben wahrscheinlich nicht 1 zu 1 wiederverwendet werden kann, so könnte der Inhalt dieser Arbeit trotzdem für nachfolgende Entwickler als Hilfestellung bei der Entwicklung diverser Hardwareschnittstellen dienen.

¹ [Eurobot 2011 Rules, www.eurobot.org]

² [Thomas Krößwang-Ridler: Erstellung einer Hautplatine für autonomen Roboter]

1.2 Allgemeines zu Hardwaretreibern³

Das Programm eines Mikrocontrollers stellt den „Kern“ der Softwarefunktionalität dar. Der Aufbau ist in der Regel hierarchisch in einem Schichtenmodell aufgebaut. Dabei greifen die oberen Schichten wie Anwendungstasks und Programme auf die darunterliegenden Schichten wie etwa die Hardwaretreiber zu.

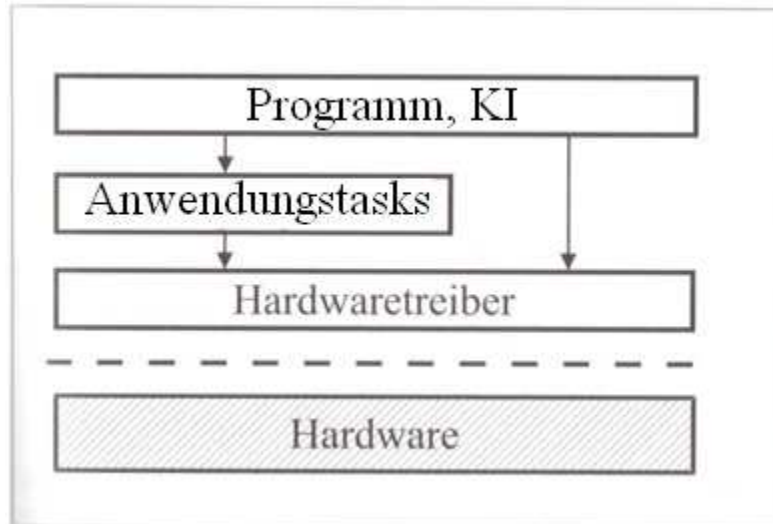


Abbildung 1: Schichtenmodell eines Embedded-Betriebssystems

Hardwaretreiber dienen als Schnittstelle zwischen der Hardware selbst (angeschlossene Sensoren, Aktoren, Timer, PWM, ...) und den höheren Programmebenen, etwa einer State Machine. Ziel ist es dabei, die Hardwareregister vor dem direkten Zugriff durch die höheren Ebenen zu schützen. Dies ermöglicht eine in gewissen Grenzen hardwareunabhängige Programmierung der höheren Ebenen, da bei Änderung der Hardware (z.B. Tausch des μC) nur die Hardwaretreiber angepasst werden müssen. Zudem kann durch Zusammenlegen von teilweise sehr aufwändigen Bitoperationen beim Zugriff auf die Hardwareregister Codelänge eingespart werden. Hardwaretreiber werden zumeist als Funktionen mit Übergabeparametern (etwa *PumpeEin(pumpeNr)*) realisiert, die in den höheren Programmebenen aufgerufen werden können.

2. Pinbelegung und Timerübersicht der Mikrocontroller

In diesem Unterpunkt soll eine kurze Übersicht über die angeschlossene Hardware (Sensoren, Aktoren, ...) sowie die Timerbelegung der einzelnen Mikrocontroller angeführt werden. Diese dient vor allem zur Konfiguration der Ports sowie der Planung benötigter Hardwaretreiber.

³ [Jörg Wiegelmann: Softwareentwicklung in C für Mikroprozessoren und Mikrocontroller]

2.1 Mikrocontroller 1

Dieser Mikrocontroller ist für die Regelung der Antriebsmotoren und des Linearantriebs sowie Ansteuerung der Pneumatikaktoren / -Sensoren zuständig. Weiters liest er über die SPI-Schnittstelle (MISO, MOSI, SCK, /CS1) die Gegnererkennungshardware aus. In Abbildung 2 ist die Pinbelegung sowie Timernutzung dargestellt.

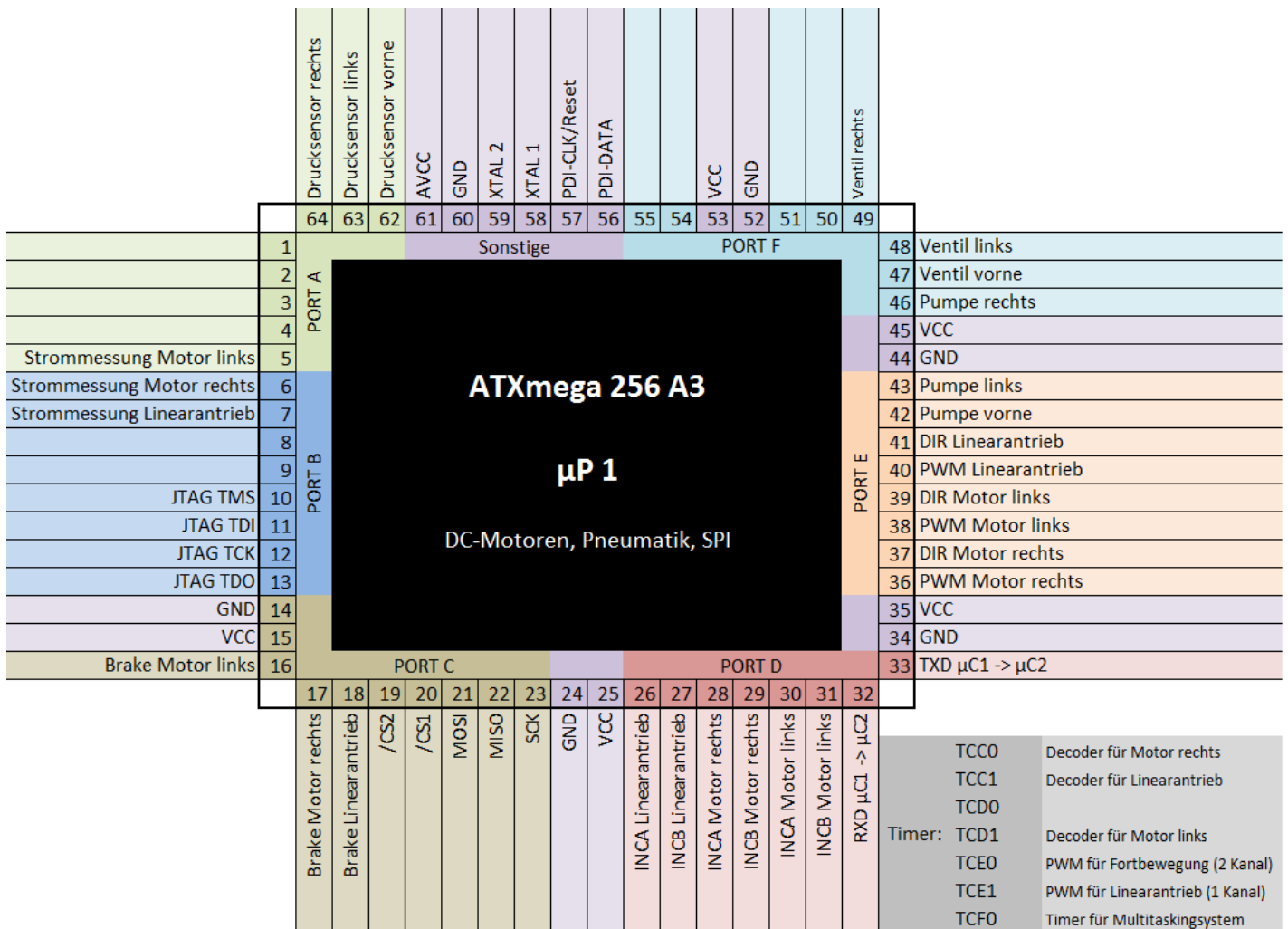


Abbildung 2: Pinbelegung Mikrocontroller 1

Benötigte Hardwaretreiber:

- Timer mit Interrupt bei Überlauf für Multitaskingsystem (siehe Punkt 3.2)
- PWM-Timer für Ansteuerung der Fortbewegungs- und des Linearmotors (siehe Punkt 3.3)
- Quadraturdecoder zur Bestimmung der Position der Antriebe (siehe Punkt 3.5)
- Analog-Digital-Wandler zur Druckmessung der Pumpen sowie Strommessung der DC-Motoren (siehe Punkt 3.4)

2.2 Mikrocontroller 2

Hauptaufgabe dieses Mikrocontrollers ist das Einlesen der Sensorik (Infrarot-Digitalsensoren, Ultraschall-Distanzsensoren). Zudem enthält er die künstliche Intelligenz des Roboters, Strategieplanung und versendet per USART Befehle an die anderen beiden Mikrocontroller bzw. erhält Regler/Sensordaten von ihnen. Durch seine PC-Anbindung über USB oder XBEE eignet er sich am besten zum Debuggen. In Abbildung 3 ist die Pinbelegung sowie Timernutzung dargestellt.

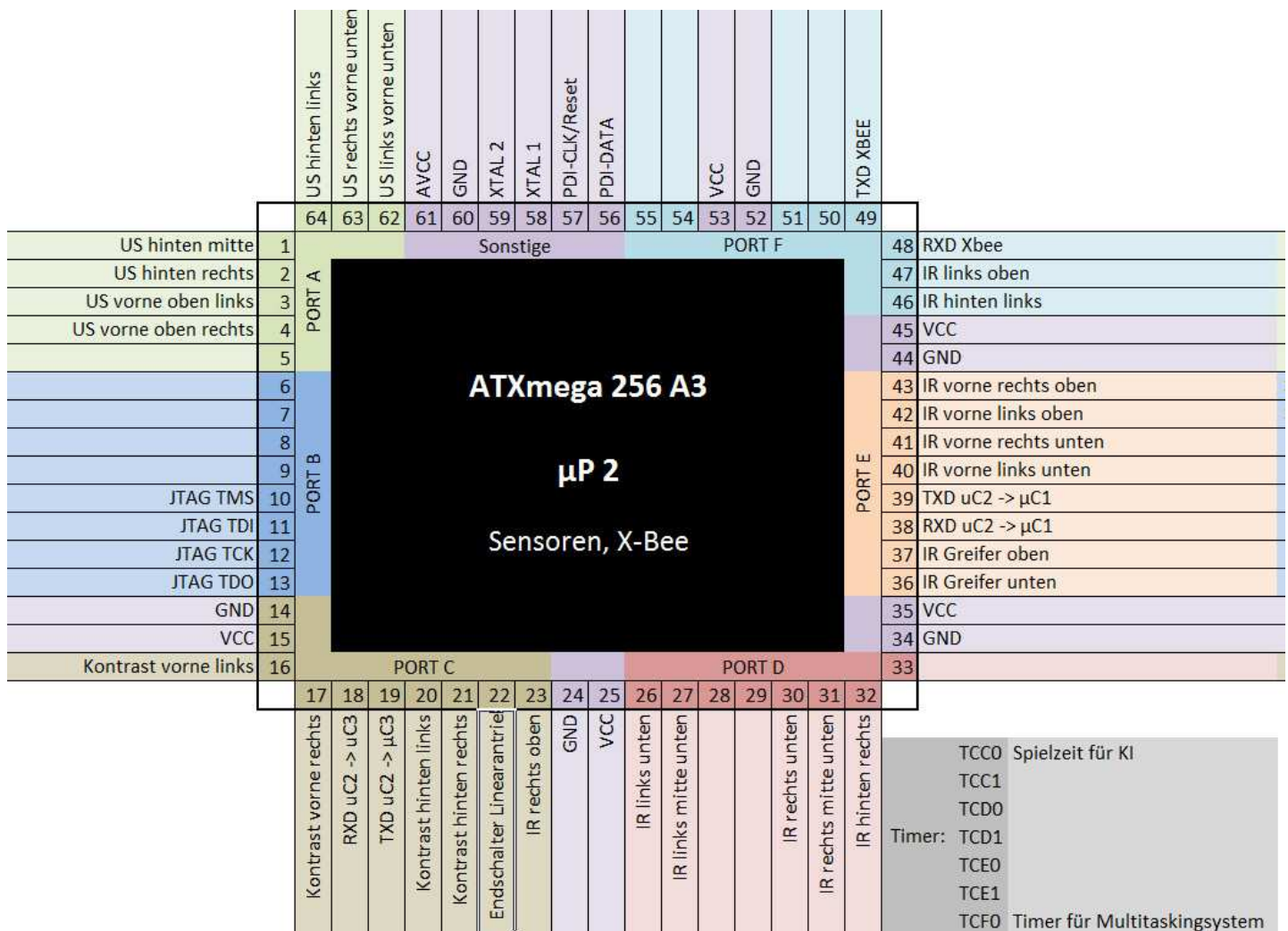


Abbildung 3: Pinbelegung Mikrocontroller 2

Benötigte Hardwaretreiber:

- Timer mit Interrupt bei Überlauf für Multitaskingsystem (siehe Punkt 3.2)
- Timer mit Interrupt bei Überlauf für spielzeitabhängige Aktionen der KI (siehe Punkt 3.2)
- Treiber zum periodischen Einlesen der Sensordaten (siehe Punkt 3.6)
- Analog-Digital-Wandler zur Druckmessung der Pumpen sowie Strommessung der DC-Motoren (siehe Punkt 3.4)

3. Hardwaretreiber

In diesem Punkt sollen die wichtigsten Hardwareinitialisierungs- und Zugriffsroutinen beschrieben und mit Codebeispielen veranschaulicht werden. Für genauere Informationen sei auf das Manual des ATXmega256 verwiesen.⁴

3.1 Port- und Pinconfiguration

Beschreibung:

Für den Anschluss unterschiedlicher Hardware müssen die Portpins des Mikrocontrollers entsprechend konfiguriert werden, um eine korrekte Kommunikation zu ermöglichen. Neben der Vorgabe der Datenrichtung (Eingangs-/Ausgangspin) bietet der ATXmega256 noch weitere Einstellmöglichkeiten, etwa zuschaltbare Pullup-/Pulldown-Widerstände oder Interrupts, die auf Flanken am Eingang reagieren.

Register:

siehe auch „Register Description Ports“ im ATXmega-Manual S.138ff

PORTx.DIR: Dient zur Wahl der Signalrichtung der einzelnen Portpins.

PORTx.OUT: Dient zur Ausgabe von High oder Low wenn der Pin auf „Ausgang“ gesetzt ist.

PORTx.PIN: Dient zum Einlesen der Anliegenden Spannung am Pin.

PORTx.INTCTRL:

Jeder Port des ATXmega256 verfügt über 2 Interrupts, die durch bestimmte Flanken oder Pegel am Pin aufgerufen werden. Im Register INTCTRL können die beiden Interrupts enabled sowie eine Interrupt-Priorität vergeben werden.

PORTx.INT0MASK, PORTx.INT1MASK:

In diesem Register kann ausgewählt werden, welche Portpins die Interrupts 0 und 1 auslösen sollen.

PORTx.PINyCTRL:

Jeder Pin eines Ports kann mit diesem Register individuell konfiguriert werden. Die Begrenzung der Slewrate sowie die Invertierung von Eingangs-/Ausgangspegel können hier aktiviert werden. Weiters können verschiedene Beschaltungen (Totempole, Pullup, Pulldown, Wired And/Or, ...) für den Pin aktiviert werden (siehe Table 13-4 im Manual Seite 142). Außerdem kann in dem Register gewählt werden,

⁴ [AVR XMEGA A3 Device Datasheet, www.atmel.com]

welche Art von Flanken (positiv, negativ, beide) oder Pegel einen Interrupt auslösen sollen (siehe Table 13-5 im Manual Seite 143).

Beispielcode:

Es soll der PORT E wie folgt konfiguriert werden:

- Pin 2 und 3 als Ausgang, alle anderen Pins als Eingang
- Pin 4 und 5 mit internem Pullup-Widerstand, alle anderen Pins unbeschaltet
- Pin 6 als Quelle für Port-Interrupt 0 durch steigende Flanke

```
void portE_init()
{
    PORTE.OUT = 0x00;
    PORTE.DIR = 0x0C;          //Pin 2 und 3 als Ausgang
    PORTE.PIN0CTRL = 0x00;
    PORTE.PIN1CTRL = 0x00;
    PORTE.PIN2CTRL = 0x00;
    PORTE.PIN3CTRL = 0x00;
    PORTE.PIN4CTRL = 0x18; //Pin 4 internen Pullup-Widerstand aktivieren
    PORTE.PIN5CTRL = 0x18; //Pin 5 internen Pullup-Widerstand aktivieren
    PORTE.PIN6CTRL = 0x01; //Pin 6 löst Interrupt aus bei steigender Flanke
    PORTE.PIN7CTRL = 0x00;
    PORTE.INTCTRL = 0x01;    //Interrupt0 enable
    PORTE.INT0MASK = 0x40; //Pin 6 als Interruptquelle für Interrupt0 wählen
    PORTE.INT1MASK = 0x00;
}
```

Quellcode 1: Initialisierung von Port E

3.2 Überlauftimer zur Taktvorgabe

Beschreibung:

Aufgabe dieser Timer ist die Erzeugung von periodischen Timer-Interrupts mit festgelegter Periodendauer. Dazu wird der Timer im „Normal Operation“-Modus betrieben. Als Taktgenerator wird der interne Takt des Mikrocontrollers verwendet, welcher über den Prescaler geteilt wird ($clk/1$ bis $clk/1024$). Über das Periodendauerregister (PER) kann die Zählweite und somit die Periodendauer des Timers definiert werden. Bei Überlauf wird der Timer/Counter Overflow/Underflow Interrupt ausgelöst, in dem die periodisch durchzuführenden Operationen abgearbeitet werden. Um den Interrupt zu aktivieren muss zusätzlich der Overflow-Interrupt enabled sein.

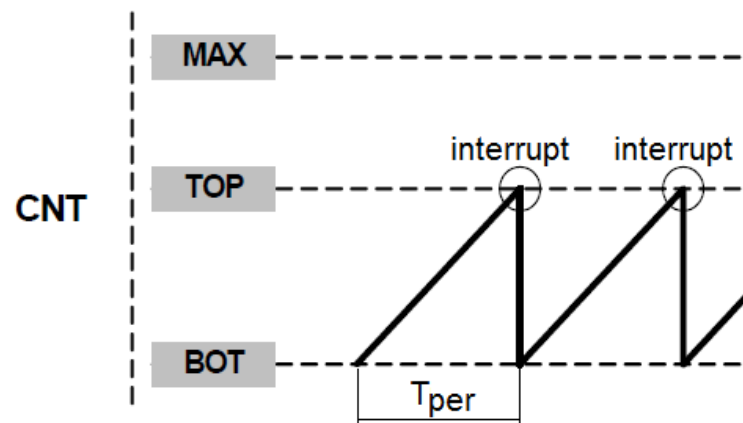


Abbildung 5: Normal Mode mit Interruptauslösung

Verwendung:

- Takttimer für Multitaskingsystem aller Mikrocontroller
- Timer zur Spielzeitanzeige für das LCD in Mikrocontroller 3
- Timer für spielzeitgesteuerte Aktionen der KI in Mikrocontroller 2

Register:

siehe auch „Timer Register Description“ im ATXmega-Manual S.165ff

CTRLA:

Dient zur Wahl des Eingangstaktes des Timers. Zusammen mit dem PER-Register bestimmt es die Periodendauer/Frequenz der auftretenden Interrupts:

$$f_{int} = \frac{1}{T_{int}} = \frac{f_{per}}{N * PER}$$

Entwicklung von Hardwaretreibern für autonomen Roboter

fint...Interruptfrequenz
fper ... interner Takt des Mikrocontrollers (peripheral clock frequency)
N ... Prescaler
PER...Period Register (=TOP-Wert) des Timers

PER:

siehe CTRLA

CTRLB:

Dient zur Wahl der Betriebsart des Timers. Für Normal Operation Mode 0x00 setzen.

INTCTRLA:

Legt fest, welche Interruptquellen aktiviert sind und welche Priorität sie haben.

Benötigt wird für den Überlauftimer der Overflow Interrupt, der durch die niederwertigsten 2 Bits definiert ist.

Beispielcode:

Für einen periodischen Timerinterrupt von „Timer F0“ von 1ms bzw. 1kHz:

$$\text{fper} = 32\text{MHz}, N = 1, \text{PER} = 32000 \rightarrow \text{fint} = \frac{32\text{MHz}}{1 \cdot 32000} = 1\text{kHz}$$

```
void tcf0_init(void)           //Timerinitialisierung
{
    TCF0.CTRLA = 0x01;         //Clocksource = fper/1
    TCF0.CTRLB = 0x00;         //Timermode = Normal Operation
    TCF0.INTCTRLA = 0x01;      //Overflow-Interrupt enable (low priority)
    TCF0.PER = 32000;          //Period-Register für Overflow
}

unsigned int spielzeit;

interrupt [TCC0_OVF_vect] void tcc0_overflow_isr() //Interruptbehandlung
{
    spielzeit++;               //Beispielcode
    CheckTime();               //Beispielcode
}
```

Quellcode 2: Initialisierung eines periodischen Überlauftimers

3.3 PWM-Timer

Beschreibung:

Zur Ansteuerung vieler Aktoren eines Roboters werden PWM-Signale benötigt. Dies sind Rechtecksignale, die durch ihre Periodendauer sowie ihr Puls/Pausen-Verhältnis (Pulsweite) definiert sind. Der ATXmega256 verfügt pro Timer vom Typ 0 über 4 PWM-Kanäle, Timer vom Typ 1 besitzen je 2 PWM-Kanäle. Die PWM-Signale eines Timers haben gleiche Periodendauer, die Pulsweite kann aber für jeden Kanal einzeln vorgegeben werden. Jeder PWM-Kanal verfügt über einen eigenen Output-Pin des s an dem das Signal abgegriffen werden kann.

Die Periodendauer wird wie im „Normal Operation“-Modus des Timers über das PER-Register definiert. Die Pulsweite wird über das CC-Register für jeden Kanal einzeln festgelegt. Bei Überschreiten des Zählstandes über den Registerwert wird der PWM-Ausgang auf Low gesetzt, beim Timerüberlauf wieder auf High. Das Verhältnis von CC-Wert zu PER entspricht dem Pulsweitenverhältnis (High/Low-Zeit).

Die zugehörigen Outputpins der Timer vom Typ 0 sind immer die Pins 0-3 des jeweiligen Ports (bei Timer C0 etwa PORTC Pin 0-3), beim Timer vom Typ 1 sind es die Pins 4 und 5 des jeweiligen Ports. Somit können mithilfe der 4 Timer Typ 0 und 3 Timer vom Typ 1 bis zu 22 PWM-Kanäle pro ATXmega256 realisiert werden.

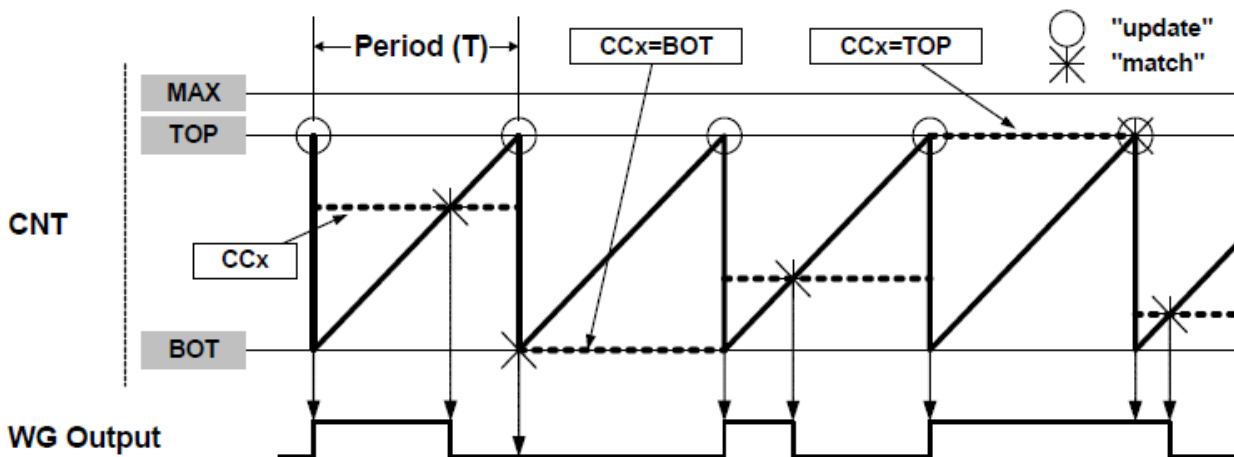


Abbildung 6: Timer im Single-Slope PWM-Mode

Verwendung:

- Ansteuerung der Servomotoren für Greifer in Mikrocontroller 3
- Ansteuerung der DC-Motoren für Fortbewegung und Linearantrieb in Mikrocontroller 1

Register:

siehe auch „Timer Register Description“ im ATxMega-Manual S.165ff

CTRLA:

Dient zur Wahl des Eingangstaktes des Timers. Zusammen mit dem PER-Register bestimmt es die Periodendauer/Frequenz der PWM:

$$f_{pwm} = \frac{f_{per}}{N * PER}$$

f_{pwm}...PWM-Frequenz

f_{per} ... interner Takt des Mikrocontroller (peripheral clock frequency)

N ... Prescaler

PER...Period Register des Timers

PER:

siehe CTRLA

CTRLB:

Dient zur Wahl der Betriebsart des Timers.

Die niederwertigsten 3 Bits bestimmen die Betriebsart des Mikrocontrollers (siehe Table 14-4 im ATxMega-Manual). Für Single-Slope-PWM müssen sie auf 3 gesetzt werden.

Die Höchstwertigsten 4 Bit bestimmen, ob die jeweiligen Port-Pins, die den PWM-Kanälen des Mikrocontrollers zugehörig sind, als PWM-Kanäle arbeiten. Wird z.B. nur eine 1-Kanal-PWM benötigt, so muss nur eines dieser 4 Bits 1 gesetzt werden, die anderen 3 bleiben auf 0, sodass diese Pins für andere Funktionen genutzt werden können.

CCx:

Dienen zur Auswahl des Capture/Compare-Wertes, der die Pulsweite der PWM bestimmt.

Timer vom Typ 0 besitzen CCA, CCB, CCC und CCD, Timer vom Typ 1 besitzen nur CCA und CCB.

Beispielcode:

Für eine 4-Kanal-PWM an Timer D0 mit einer Periodendauer von 20ms (50Hz):

$$f_{per} = 32\text{MHz}, N = 64, PER = 10000 \rightarrow f_{pwm} = \frac{32\text{MHz}}{64 \cdot 10000} = 50\text{Hz}$$

```
void tcd0_init(void)           //Timerinitialisierung
{
    TCD0.CTRLA = 0x05;         //Clocksource = fper/64
    TCD0.CTRLB = 0xF5;         //Timermode = Single Slope PWM, alle 4 PWM-Kanäle aktiv
    TCD0.PER = 10000;          //Period-Register für Overflow

    TCD0.CCA = 0;              // Pulsweitenverhältnis Kanal A = 0%
    TCD0.CCB = 2500;           // Pulsweitenverhältnis Kanal B = 25%
    TCD0.CCC = 5000;           // Pulsweitenverhältnis Kanal C = 50%
    TCD0.CCD = 10000;          // Pulsweitenverhältnis Kanal D = 100%
}
```

```
void SetPWM(unsigned char ucChannel, unsigned int uiPWMValue) //PWM einstellen
{
    if(ucChannel == 0)         //Wahl des PWM-Kanals
        TCD0.CCA = uiPWMValue; //Definition der Pulsweite
    else if(ucChannel == 1)
        TCD0.CCB = uiPWMValue;
    else if(ucChannel == 2)
        TCD0.CCC = uiPWMValue;
    else if(ucChannel == 3)
        TCD0.CCD = uiPWMValue;
}
```

Quellcode 3: Initialisierung eines PWM-Timers

3.4 Analog-Digital-Wandler

Beschreibung:

Der ATXmega256 verfügt über 2 unabhängige integrierte 12-Bit-ADCs, die jeweils 8 analoge Eingangsspannungen (Port A und B) sowie einige intern bereitgestellte Analogspannungen (z.B. Temperatursensor des Mikrocontrollers) auslesen können. Es stehen pro ADC 4 Kanäle zur Verfügung, die Unterschiedlich konfiguriert werden können, etwa für Single-Ended (Pin-Spannung gegen Ground), Differenzspannungen zwischen 2 Pins oder interne Spannungen. Die Auswahl der zu messenden Eingangsspannung(en) wird über das MUX-Register jedes Kanals getroffen.

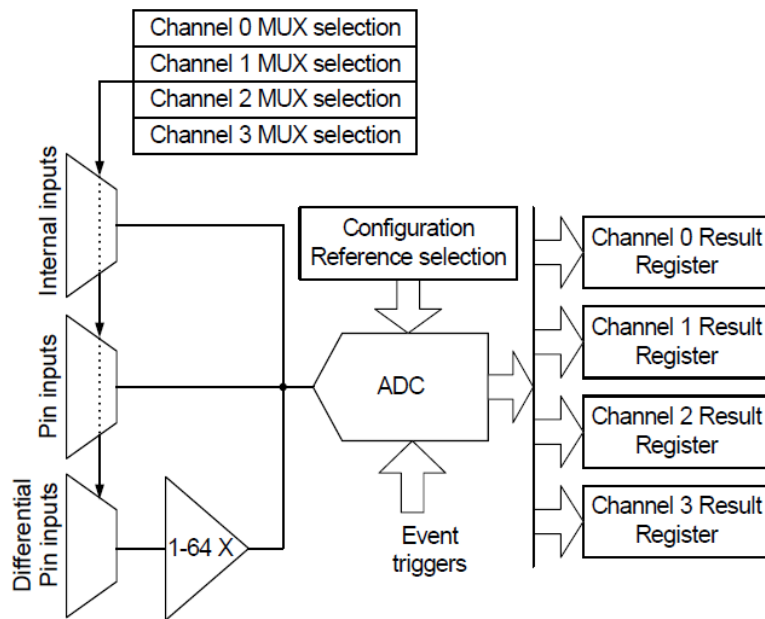


Abbildung 7: ADC-Übersicht

Verwendung:

- Einlesen der Ultraschallsensoren in Mikrocontroller 2
- Einlesen der Drucksensoren in Mikrocontroller 1
- Einlesen der motorstromproportionalen Spannungen in Mikrocontroller 1
- Einlesen der Batteriespannung in Mikrocontroller 3

Register:

siehe auch ADC Register Description“ im ATXmega-Manual S.302ff

CTRLB:

Dient zum Einstellen einiger ADC-Optionen:

Conversion Mode: Ausgabe der Ergebnisse unsigned (0) oder signed (1) Ergebnissen. Bei Single-Ended-Messungen sollte unsigned gewählt werden, da keine negativen Vorzeichen im Ergebnis auftreten können.

Free Running Mode On/Off: Im Free Running Mode läuft der ADC durchgehend, d.h. nach jeder Wandlung wird sofort eine neue gestartet.

Auflösung: Es können 8 oder 12-Bit-Wandlungen durchgeführt werden. Standardmäßig sind 12-Bit Auflösung und right-adjusted (niederwertigste 8 Bit im Low-Register des Ergebnisses) eingestellt.

REFCTRL:

Dient zur Auswahl der Referenzspannung für Wandlungen:

Table 25-3. ADC Reference Configuration

REFSEL[1:0]	Group Configuration	Description
00	INT1V	Internal 1.00V
01	INTVCC	Internal $V_{CC}/1.6$
10 ⁽¹⁾	AREFA	External reference from AREF pin on PORT A.
11 ⁽²⁾	AREFB	External reference from AREF pin on PORT B.

- Notes: 1. Only available if AREF exist on PORT A.
2. Only available if AREF exist on PORT B.

Abbildung 8: ADC Referenztafel

Weiters können in diesem Register die interne Temperaturreferenz sowie ein internes Bandgap für die ADC-Messung vorbereitet werden.

EVCTRL:

Dient zur Auswahl von Ereignissen aus dem Event System des Mikrocontrollers, die die Wandlung auslösen sollen. Sollen Wandlungen nur von der Software aus gestartet werden können, so wird dieses Register 0x00 gesetzt.

PRESCALER:

Dient zur Auswahl der Taktfrequenz des ADCs. Möglich sind $\frac{1}{4}$ bis $\frac{1}{512}$ des internen Taktes

CALL und CALH:

Dient zum Kalibrieren des ADCs, siehe Beispielcode.

Channel.CTRL:

Jeder ADC-Kanal enthält ein eigenes Control-Register zur Parametrisierung der durchzuführenden Wandlungen

Inputmode: Wahl der Wandlungsart (Single-Ended, Differenzspannung, interne Spannungsquelle)

Gain: Vorverstärkung der Eingangsspannung vor Wandlung

Start: Startet die Wandlung des Kanals, wird von der Hardware rückgesetzt

Channel.MUXCTRL:

Dient zur Auswahl der Eingangsspannung(en) für den ADC. Je nach Inputmode können entweder die 8 Eingangspins oder interne Spannungen als Eingang für die nächste Wandlung des Channels gewählt werden. Bei Differential-Input-Messungen müssen sowohl der positive als auch der negative Eingang der Messung gewählt werden.

Channel.RESH und Channel.RESL:

Dies ist das Ergebnisregister, indem der gewandelte Spannungswert abgelegt wird. Bei Standardeinstellung (12-Bit Auflösung und right-adjusted) werden die niederwertigen 8 Bit in RESL gespeichert und die höherwertigen 4 Bit in RESH.

Beispielcode:

Im folgenden Beispiel soll der ADCA initialisiert werden. Als Referenzspannung soll die am Referenzpin anliegende 3.3V-Versorgungsspannung des Mikrocontrollers dienen. Die Wandlung soll ausschließlich durch Softwareaufruf gestartet werden.

Kanal 0 soll so konfiguriert werden, dass er Single-Ended-Wandlungen von Port A Pin7 durchführt.

Zur Kalibrierung des ADCs wird folgendes Unterprogramm benötigt:

```
// Function used to read the calibration byte from the
// signature row, specified by 'index'
#pragma optsize-
unsigned char read_calibration_byte(unsigned char index)
{
    unsigned char r;
    NVM.CMD=NVM_CMD_READ_CALIB_ROW_gc;
    r*((flash unsigned char*) index);
    // Clean up NVM command register
    NVM.CMD=NVM_CMD_NO_OPERATION_gc;
    return r;
}
#pragma optsize_default
```

Quellcode 4: Auto-Kalibrierung des ADCs

```

// ADCA initialization
void adca_init(void)
{
    // ADCA is enabled
    // Resolution: 12 Bits
    // Load the calibration value for 12 Bit resolution
    // from the signature row
    ADCA.CALL=read_calibration_byte(PROD_SIGNATURES_START+ADCACAL0_offset);
    ADCA.CALH=read_calibration_byte(PROD_SIGNATURES_START+ADCACAL1_offset);

    // Free Running mode: Off
    // Conversion mode: Unsigned
    ADCA.CTRLB=(ADCA.CTRLB & (~(ADC_CONMODE_bm | ADC_FREERUN_bm |
ADC_RESOLUTION_gm))) | ADC_RESOLUTION_12BIT_gc;

    // Clock frequency: 1000 kHz
    ADCA.PRESCALER=(ADCA.PRESCALER & (~ADC_PRESCALER_gm)) | ADC_PRESCALER_DIV32_gc;

    // Reference: Internal 3.3V
    // Temperature reference: On
    ADCA.REFCTRL=(ADCA.REFCTRL & ((~(ADC_REFSEL_gm | ADC_TEMPREF_bm)) |
ADC_BANDGAP_bm)) | ADC_REFSEL_VCC_gc | ADC_TEMPREF_bm | ADC_BANDGAP_bm;

    // Initialize the ADC Compare register
    ADCA.CMPL=0x00;
    ADCA.CMPH=0x00;

    // ADC channel 0 gain: 1
    // ADC channel 0 input mode: Single-ended positive input signal
    ADCA.CH0.CTRL=(ADCA.CH0.CTRL & (~(ADC_CH_START_bm | ADC_CH_GAINFAC_gm |
ADC_CH_INPUTMODE_gm))) | ADC_CH_GAIN_1X_gc | ADC_CH_INPUTMODE_SINGLEENDED_gc;

    // ADC channel 0 positive input: ADC7 pin
    // ADC channel 0 negative input: GND
    ADCA.CH0.MUXCTRL=(ADCA.CH0.MUXCTRL & (~(ADC_CH_MUXPOS_gm | ADC_CH_MUXNEG_gm))) |
    ADC_CH_MUXPOS_PIN7_gc;

    // AD conversion is started by software
    ADCA.EVCTRL=ADC_EVACT_NONE_gc;

    // Channel 0 interrupt: Disabled
    ADCA.CH0.INTCTRL=(ADCA.CH0.INTCTRL & (~(ADC_CH_INTMODE_gm | ADC_CH_INTLVL_gm))) |
    ADC_CH_INTMODE_COMPLETE_gc | ADC_CH_INTLVL_OFF_gc;
    // Enable the ADC
    ADCA.CTRLA|=ADC_ENABLE_bm;
    // Insert a delay to allow the ADC common mode voltage to stabilize
    delay_us(2);
}

```

Quellcode 5: Initialisierung eines ADCs

Mit folgender Funktion können Wandlungen auf verschiedenen Channels durchgeführt werden, vorausgesetzt die Channels sind im `adca_init()` parametriert. Mit der Variable `Pin` kann der einzulesende Mikrocontroller-Pin gewählt werden (Port A0...A7).

```
// ADCA channel data read function using polled mode
unsigned int adca_read(unsigned char channel, unsigned char Pin)
{
    ADC_CH_t *pch=&ADCA.CH0+channel;
    unsigned int data;

    if(Pin > 7)
        return(-1);

    Pin <<= 3;
    // Pin setzen
    pch->MUXCTRL &= ~0x38;
    pch->MUXCTRL |= Pin;

    // Start the AD conversion
    pch->CTRL|=ADC_CH_START_bm;
    // Wait for the AD conversion to complete
    while ((pch->INTFLAGS & ADC_CH_CHIF_bm)==0);
    // Clear the interrupt flag
    pch->INTFLAGS=ADC_CH_CHIF_bm;
    // Read the AD conversion result
    ((unsigned char *) &data)[0]=pch->RESL;
    ((unsigned char *) &data)[1]=pch->RESH;
    return data;
}
```

Quellcode 6: Starten einer ADC-Wandlung

3.5 Quadraturdecoder

Beschreibung:

Zur Bestimmung von aktuellem Winkel, Drehzahl und Drehrichtung von Motoren werden häufig Encoder verwendet, die als Ausgangssignal Rechteckssignale verwenden. Die Drehzahl ist proportional der Frequenz und aus dem Vorzeichen der Phasendifferenz der beiden Quadratursignale QDPH0 und QDPH90 kann die Drehrichtung bestimmt werden. Zusätzlich bieten viele Encoder auch ein Indexsignal, welches eine absolute Winkel-/Positionsbestimmung durch Referenzierung am Indexpunkt ermöglicht.

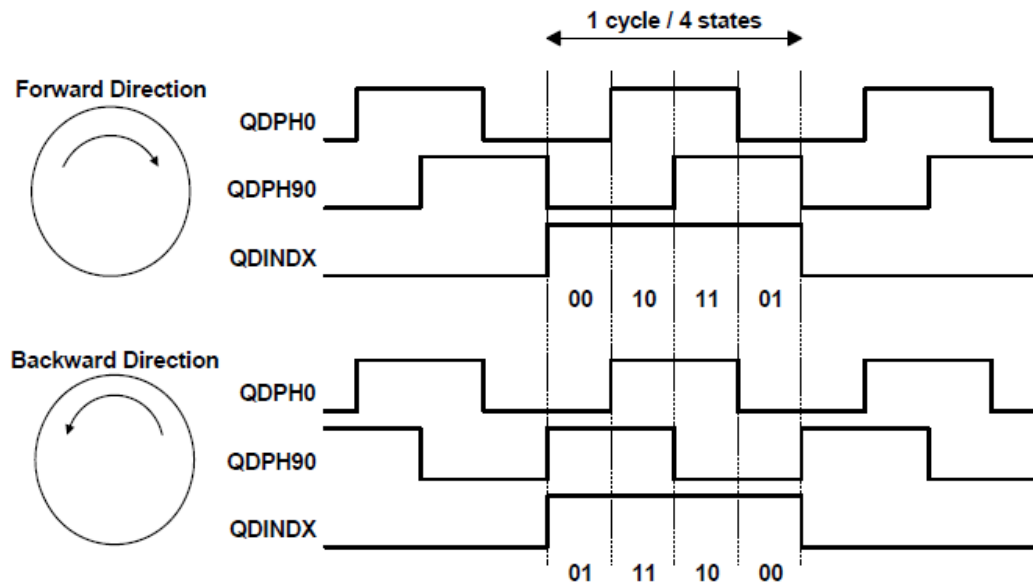


Abbildung 9: Signalverlauf eines rotierenden Encoders⁵

Zur Auswertung der Encodersignale im Mikrocontroller wird ein interner Timer verwendet, der über die Signale QDPH0 und QDPH90 getaktet wird und daraus auch die Zählrichtung ableitet.

Zur Implementierung dieser Funktion wird ein von ATMEL zur Verfügung gestellter Treiber verwendet, der die Verknüpfung der verwendeten Portpins mit dem zur Verarbeitung vorgesehenen Timer über das interne Eventsystem herstellt. Für genauere Informationen dazu sei auf die Application Note „AVR1600: Using the XMEGA Quadrature Decoder“ von ATMEL verwiesen.⁶ Quellcode des Treibers „QDEC_Total_Setup“ siehe Anhang.

Verwendung:

- Drehratenbestimmung der DC-Motoren für Fortbewegung und Linearantrieb in Mikrocontroller 1

^{5,6} [AVR 1600: Using the XMEGA Quadrature Decoder, www.atmel.com]

Treibercode:

Zur Implementierung eines Quadraturdecoders mithilfe des Treibers müssen folgende Daten bereitgestellt werden:

- Portpins für Quadratursignale und ev. Indexsignal: Die Quadratursignalpins und der Indexpin (wenn vorhanden) müssen am Mikrocontroller am selben Port direkt nacheinander liegen! Eventuell ist zur Korrektur der Zählrichtung eine Invertierung erforderlich.
- Eventsystem: Dient als „Linker“ zwischen Portpins und Timer. Es muss ein Eventkanal bereitgestellt werden, die Wahl der Eventquelle erfolgt über qPinInput, siehe dazu Table 6-3 auf Seite 72 des Manuals.
- Die Konfiguration des Timers erfolgt automatisch durch QDEC_Total_Setup, es müssen nur Timer und Zählweite (=lineCount) angegeben werden. Der aktuelle Zählstand kann dann im Count-Register des Timers abgelesen werden.

```

/*****
*** Funktionsname:  QDEC_Total_Setup
*** Erstellt von Atmel Corporation (AVR1600)
*** Beschreibung:  Diese Funktion kombiniert QDEC_Port_Setup und
***                QDEC_TC_Dec_Setup
*** Parameter: qPort    Verwendeter Port
*** Parameter: qPin     Erster Input Pin
*** Parameter: invIO    Wahr wenn I/O pins invertiert werden sollen
***
***
*** Parameter: qEvMux    Verwendeter EventChannel(nur 0,2 & 4)
*** Parameter: qPinInput Input Pin von QDPH0 zum EVSYS.CHMUX
*** Parameter: useIndex  Wahr für optionales Index Signal
*** Parameter: qIndexState Status zum triggern des Indexes
***
*** Parameter: qTimer    Verwendeter Timer
*** Parameter: qEventChannel Verwendeter EventChannel
*** Parameter: lineCount Anzahl der Signale/Umdrehung
***
*** Rückgabeparameter: bool Falsch bei Problemen des Setup
*****/
bool QDEC_Total_Setup(PORT_t * qPort,
                    uint8_t qPin,
                    bool invIO,
                    uint8_t qEvMux,
                    EVSYS_CHMUX_t qPinInput,
                    bool useIndex,
                    EVSYS_QDIRM_t qIndexState,
                    TC1_t * qTimer,
                    TC_EVSEL_t qEventChannel,
                    uint16_t lineCount)
{
    if( !QDEC_Port_Setup(qPort, qPin, useIndex, invIO) )
        return false;
    if( !QDEC_EVSYS_Setup(qEvMux, qPinInput, useIndex, qIndexState ) )
        return false;
    QDEC_TC_Dec_Setup(qTimer, qEventChannel, lineCount);

    return true;
}

```

Beispielcode:

Implementierung eines Decoders ohne Indexsignal. Die beiden Portpins für das Encodersignal sind Port D Pin 4 und 5. Timer soll TCD1 sein mit einem lineCount von 1000.

```
//QDEC1 Quadrature Decoder 1
QDEC_Total_Setup(&PORTD,      //PORTD für QDEC1
                 4,           //Pin4 als erster Pin
                 0,           //nicht Invertiert
                 0,           //Event Multiplexer 0 (2 oder 4 auch möglich)
                 0x6C,        //Input Pin für EventSystem (Manual S72)
                 0,           //kein Index Signal
                 0,           //00 als Index Status
                 &TCD1,       //Timer/Counter TCD1 für QDEC1
                 0x08,        //EventChannel für QDEC1
                 1000);       //Zählweite
```

Quellcode 8: Beispielprogramm Quadraturdecoder

3.6 Sensorik

Beschreibung:

Um den Roboter autonom und in Echtzeit agieren lassen zu können ist es unerlässlich, seine Sensordaten auf dem neuesten Stand zu halten. Zu diesem Zwecke muss ein zyklisches Einlesen der Sensordaten erfolgen.

Um dies sicherzustellen wurde eine zyklische Routine in das Multitaskingsystem des Mikrocontrollers integriert, die alle 5ms die Sensordaten einliest und in interne Register speichert, auf die von anderen Routinen zugegriffen werden kann.

Zur Erstellung dieser zyklischen Routine muss zuerst ein neuer Task für das Multitaskingsystem erstellt werden. Dies geschieht im Unterprogramm InitSensor(), welches im Hauptprogramm einmal aufgerufen werden muss:

```
void InitSensor(void)
{
    SET_CYCLE(SENSOR_TASKNBR, 5);           // Zykluszeit: 5 ms
    SET_TASK(SENSOR_TASKNBR, CYCLE);       // zyklischer Task
    SET_TASK_HANDLE(SENSOR_TASKNBR, SensorTask); // Zuweisung der Abarbeitung zum Task
}
```

Quellcode 9: Initialisierung des Sensortasks

Im eigentlichen Task kann nun das Einlesen der Sensoren erfolgen. Digitalsensordaten werden durch Bitoperationen aus des Portregistern PORTx.IN gewonnen, analoge Sensordaten über die ADC-Routine `adca_read(unsigned char channel, unsigned char Pin)`, siehe Punkt 3.4: Analog Digital Wandler. Der Rückgabewert des Tasks ist entweder CYCLE (Task soll nach Zykluszeit erneut aufgerufen werden), ENABLE (Task soll sofort wieder abgearbeitet werden) und DISABLE (Task nicht erneut aufrufen).

```

unsigned char SensorTask(void)
{
    SET_CYCLE(SENSOR_TASKNBR, 5);           // Zykluszeit: 5 ms
    SET_TASK(SENSOR_TASKNBR, CYCLE);       // zyklischer Task

    //Infrarotsensoren
    ir_links_unten = PORTD.IN & 0x01;
    ir_links_mitte_unten = (PORTD.IN & 0x02) >> 1;
    ir_hinten_links = (PORTF.IN & 0x01);
    ir_links_oben = (PORTF.IN & 0x02) >> 1;
    ir_rechts_unten = (PORTD.IN & 0x10) >> 4;
    ir_rechts_mitte_unten = (PORTD.IN & 0x20) >> 5;
    ir_hinten_rechts = (PORTD.IN & 0x40) >> 6;
    ir_rechts_oben = (PORTC.IN & 0x80) >> 7;

    ir_vorne_links_unten = (PORTE.IN & 0x10) >> 4;
    ir_vorne_rechts_unten = (PORTE.IN & 0x20) >> 5;
    ir_vorne_links_oben = (PORTE.IN & 0x40) >> 6;
    ir_vorne_rechts_oben = (PORTE.IN & 0x80) >> 7;

    ir_greifer_unten = (PORTE.IN & 0x01);
    ir_greifer_oben = (PORTE.IN & 0x02) >> 1;

    //Kontrastsensoren:
    kontrast_vorne_links = (PORTC.IN & 0x01);
    kontrast_vorne_rechts = (PORTC.IN & 0x02) >> 1;
    kontrast_hinten_links = (PORTC.IN & 0x10) >> 4;
    kontrast_hinten_rechts = (PORTC.IN & 0x20) >> 5;

    endschalter_linearantrieb = (PORTC.IN & 0x40) >> 6;

    //Ultraschallsensoren:
    us_unten_links = (float)(adca_read(ADC_CH_0, 0)) * 0.0002358 + 0.0918;
    us_unten_rechts = (float)(adca_read(ADC_CH_0, 1)) * 0.0002358 + 0.0918;
    us_hinten_links = (float)(adca_read(ADC_CH_0, 2)) * 0.0002358 + 0.0918;
    us_hinten_mitte = (float)(adca_read(ADC_CH_0, 3)) * 0.0002358 + 0.0918;
    us_hinten_rechts = (float)(adca_read(ADC_CH_0, 4)) * 0.0002358 + 0.0918;
    us_oben_links = (float)(adca_read(ADC_CH_0, 5)) * 0.0002358 + 0.0918;
    us_oben_rechts = (float)(adca_read(ADC_CH_0, 6)) * 0.0002358 + 0.0918;

    return(CYCLE);                          //Task weiterhin zyklisch aufrufen
}

```

4. Test der Hardwaretreiber

Zum Testen und Parametrisieren der Hardwareroutinen wurde ein einfaches Befehlsprogramm entwickelt, welches die Ansteuerung verschiedener Komponenten per Terminal-Programm vom PC aus ermöglicht. Zu diesem Zweck dient die USART-Schnittstelle USARTF0 am Port F von Mikrocontroller 2. Durch Eingabe von Steuerbefehlen am Terminal können Aktoren angesteuert und Sensoren ausgelesen werden. Es wurde zudem auf eine einfache Erweiterbarkeit um zusätzliche Befehlssätze geachtet.

Befehle bestehen dabei immer aus einem Steuerzeichen gefolgt von 0 bis 5 Parameterzeichen (=Ziffern). So bedeutet „c1“ etwa das schließen (Close) von Greifer 1, während der Befehl „s“ alle Sensorwerte am Terminal ausgibt. Die Eingabe eines korrekten Befehlssatzes wird dem Benutzer zusätzlich durch einen automatisch generierten Zeilenumbruch im Terminal bestätigt.

Zur Ergänzung weiterer Befehle muss das Steuerzeichen sowie die Anzahl der Parameterzeichen im UP *pcCommand* ergänzt werden und die dadurch auszuführenden Aktionen im UP *executeCommand* hinzugefügt werden.

Das Unterprogramm *pcCommand* (*char inchar*) wird direkt vom Empfangsinterrupt der USART-Schnittstelle zum PC aufgerufen, Übergabeparameter ist das empfangene Zeichen.

Quellcode:

```
unsigned char input[5];      //Parameterzeichen
unsigned char counter;      //Anzahl der folgenden Parameterzeichen
unsigned char command;      //Steuerzeichen

//Verarbeitung der übergebenen USART-Zeichen zu einem Befehl
void pcCommand (char inchar)
{
    //Steuerzeichen speichern und Anzahl der Parameterzeichen vorgeben
    if(inchar == 's')        //Steuerzeichen
    {
        counter = 0;        //Anzahl der folgenden Parameterzeichen vorgeben
        command = inchar;    //Steuerzeichen speichern für Befehlsausführung
    }
    else if(inchar == 'c')
    {
        counter = 1;
        command = inchar;
    }
    else if(inchar == 'o')
    {
        counter = 1;
        command = inchar;
    }

    //Übergebene Parameter speichern
    else if((counter != 0) && (inchar >= '0') && (inchar <= '9')) //Nur Ziffern erlaubt
    {
        counter--;
        input[counter] = inchar - '0';    //Parameterwert speichern

        if(counter == 0)                  //Wenn alle notwendigen Parameterzeichen erhalten
        {
            executeCommand(command, input); //--> Befehl ausführen
            printf("\r");                  //Bestätigung Befehl korrekt
        }
    }
}
```

```
//Ausführen eines Befehls
void executeCommand(unsigned char ucCommand, unsigned char ucInput[])
{
    if(command == 's')
    {
        PrintSensors();           //Sensorwerte Ausgeben
    }
    else if(command == 'c')
    {
        SetGripper(ucInput[0], CLOSE); //Greifer x (0...3) schließen
    }
    else if(command == 'o')
    {
        SetGripper(ucInput[0], OPEN);  //Greifer x (0...3) öffnen
    }
}
```

Quellcode 11: Einfaches Testprogramm für Hardwaretreiber

5. Fazit und Danksagungen

Mittlerweile ist die Hardwaretreiberprogrammierung am Roboter praktisch abgeschlossen und die Hauptaufgabe liegt nun bei der Programmierung der künstlichen Intelligenz des Roboters wie etwa Pfadplanung und Reaktion auf externe Ereignisse wie Kollisionen. Besonders hier zeigt sich die Wichtigkeit sauber programmierter und ausgiebig getesteter Hardwaretreiber, die dem Programmierer der höheren Programmebenen das Werkzeug zum Ansteuern externer Komponenten bieten.

Auch das selbsterstellte Terminalprogramm zur Hardwareansteuerung per PC hat bei dem Testen verschiedener Funktionen sowie bei der Parametrisierung von Hardwarechnittstellen bewährt. Auch beim Testen und Debuggen der Strategieplanung (KI) war es sehr hilfreich, da es um Befehle wie „gotoStateX“ oder einen Nothalt des Roboters erweitert werden kann.

Bedanken möchte ich mich vor allem bei unseren Betreuern BSc Michael Zauner und Dipl. Ing. (FH) Raimund Edlinger, die unserem Team durch wichtige Tipps und Hinweise geholfen haben. Durch die vielen internen Testmatches gegen ihren Roboter konnten wir wichtige Erfahrungen sammeln und Fehler erkennen, mithilfe derer wir unseren eigenen Roboter stark verbessern konnten. Besonders danke ich den anderen Teammitgliedern Stefan Meisinger, Edin Mujagic und Lukas Schlossinger für die wunderbare Zusammenarbeit im Berufspraktikum.

6. Abbildungsverzeichnis

Abbildung 1: Schichtenmodell eines Embedded-Betriebssystems	6
Abbildung 2: Pinbelegung μ C 1	7
Abbildung 3: Pinbelegung μ C 2	8
Abbildung 4: Pinbelegung μ C 3	9
Abbildung 5: Normal Mode mit Interruptauslösung	12
Abbildung 6: Timer im Single-Slope PWM-Mode	14
Abbildung 7: ADC-Übersicht	17
Abbildung 8: ADC Referenztafel	18
Abbildung 9: Signalverlauf eines rotierenden Encoders	22

7. Codeverzeichnis

Quellcode 1: Initialisierung von Port E	11
Quellcode 2: Initialisierung eines periodischen Überlauf timers	13
Quellcode 3: Initialisierung eines PWM-Timers	16
Quellcode 4: Auto-Kalibrierung des ADCs	19
Quellcode 5: Initialisierung eines ADCs	20
Quellcode 6: Starten einer ADC-Wandlung	21
Quellcode 7: Initialisierung eines Quadraturdecoders	23
Quellcode 8: Beispielprogramm Quadraturdecoder	24
Quellcode 9: Initialisierung des Sensortasks	24
Quellcode 10: Sensortask	25
Quellcode 11: Einfaches Testprogramm für Hardwaretreiber	28

8. Literaturverzeichnis

¹ Eurobot Association: Eurobot 2011 “Chess up” – Rules 2011,
http://www.eurobot.org/commonfiles/docs/2011/E2011_Rules-EN.pdf

² Thomas Krößwang-Ridler: Erstellung einer Hauptplatine für autonomen Roboter, FH Wels 2011

³ Jörg Wiegelmann: Softwareentwicklung in C für Mikroprozessoren und Mikrocontroller, 3. Auflage, 2004 Hüthig GmbH & Co. KG Heidelberg

⁴ Atmel Corporation: AVR XMEGA A3 Device Datasheet, 2010,
http://www.atmel.com/dyn/resources/prod_documents/doc8068.pdf

^{5,6} Atmel Corporation: AVR 1600: Using the XMEGA Quadrature Decoder, 2008,
http://www.atmel.com/dyn/resources/prod_documents/doc8109.pdf

9. Anhang

Quadraturdecoder Quellcode

```

/*****
*** Funktionsname:  QDEC_Total_Setup          ***
*** Erstellt von Atmel Corporation (AVR1600)          ***
*** Beschreibung:  Diese Funktion kombiniert QDEC_Port_Setup und ***
***                QDEC_TC_Dec_Setup          ***
*** Parameter: qPort    Verwendeter Port          ***
*** Parameter: qPin     Erster Input Pin          ***
*** Parameter: invIO    Wahr wenn I/O pins invertiert werden ***
***                sollen          ***
***                ***
*** Parameter: qEvMux    Verwendeter EventChannel(nur 0,2 & 4) ***
*** Parameter: qPinInput Input Pin von QDPH0 zum EVSYS.CHMUX ***
*** Parameter: useIndex  Wahr für optionales Index Signal      ***
*** Parameter: qIndexState Status zum triggern des Indexes     ***
***                ***
*** Parameter: qTimer    Verwendeter Timer          ***
*** Parameter: qEventChannel Verwendeter EventChannel      ***
*** Parameter: lineCount Anzahl der Signale/Umdrehung      ***
***                ***
*** Rückgabeparameter: bool Falsch bei Problemen des Setup   ***
*****/
bool QDEC_Total_Setup(PORT_t * qPort,
                    uint8_t qPin,
                    bool invIO,
                    uint8_t qEvMux,
                    EVSYS_CHMUX_t qPinInput,
                    bool useIndex,
                    EVSYS_QDIRM_t qIndexState,
                    TC1_t * qTimer,
                    TC_EVSEL_t qEventChannel,
                    uint16_t lineCount)
{
    if( !QDEC_Port_Setup(qPort, qPin, useIndex, invIO) )
        return false;
    if( !QDEC_EVSYS_Setup(qEvMux, qPinInput, useIndex, qIndexState) )
        return false;
    QDEC_TC_Dec_Setup(qTimer, qEventChannel, lineCount);

    return true;
}

```

```

/*****
*** Funktionsname:  QDEC_Port_Setup          ***
*** Erstellt von Atmel Corporation (AVR1600)          ***
*** Beschreibung:  Konfiguriert den Port für die Qadratur Decoder ***
***
*** Parameter: qPort    Verwendeter Port          ***
*** Parameter: qPin    Erster Input Pin          ***
*** Parameter: invIO    Wahr wenn I/O pins invertiert werden ***
***                  sollen                      ***
*** Parameter: useIndex  Wahr für optionales Index Signal ***
***
*** Rückgabeparameter: bool Falsch bei Problemen des Setup ***
*****/
bool QDEC_Port_Setup(PORT_t * qPort, uint8_t qPin, bool useIndex, bool invIO)
{
    /* Make setup depending on if Index signal is used. */
    if(useIndex){
        if(qPin > 5){
            return false;
        }
        qPort->DIRCLR = (0x07<<qPin);

        /* Configure Index signal sensing. */
        PORTCFG.MPCMASK = (0x04<<qPin);
        qPort->PIN0CTRL = (qPort->PIN0CTRL & ~PORT_ISC_gm) |
PORT_ISC_BOTHEDGES_gc
        | (invIO ? PORT_INVEN_bm : 0);

    }else{
        if(qPin > 6){
            return false;
        }
        qPort->DIRCLR = (0x03<<qPin);
    }

    /* Set QDPH0 and QDPH1 sensing level. */
    PORTCFG.MPCMASK = (0x03<<qPin);
    qPort->PIN0CTRL = (qPort->PIN0CTRL & ~PORT_ISC_gm) | PORT_ISC_LEVEL_gc
        | (invIO ? PORT_INVEN_bm : 0);

    return true;
}

```

Entwicklung von Hardwaretreibern für autonomen Roboter

```

/*****
*** Funktionsname:  QDEC_Total_Setup          ***
*** Erstellt von Atmel Corporation (AVR1600)  ***
*** Beschreibung:  Diese Funktion konfiguriert das EventSystem ***
***                zum Quadrature Decoden    ***
***                ***                        ***
*** Parameter: qEvMux    Verwendeter EventChannel(nur 0,2 & 4) ***
*** Parameter: qPinInput Input Pin von QDPH0 zum EVSYS.CHMUX ***
*** Parameter: useIndex  Wahr für optionales Index Signal      ***
*** Parameter: qIndexState Status zum triggern des Indexes     ***
***                ***                ***                ***
*** Rückgabeparameter: bool Falsch bei Problemen des Setup   ***
*****/
bool QDEC_EVSYS_Setup(uint8_t qEvMux,
                     EVSYS_CHMUX_t qPinInput,
                     bool useIndex,
                     EVSYS_QDIRM_t qIndexState )
{
    switch (qEvMux){
        case 0:

            /* Configure event channel 0 for quadrature decoding of pins. */
            EVSYS.CH0MUX = qPinInput;
            EVSYS.CH0CTRL = EVSYS_QDEN_bm | EVSYS_DIGFILT_2SAMPLES_gc;
            if(useIndex){
                /* Configure event channel 1 as index channel. Note
                 * that when enabling Index in channel n, the channel
                 * n+1 must be configured for the index signal.*/
                EVSYS.CH1MUX = qPinInput + 2;
                EVSYS.CH1CTRL = EVSYS_DIGFILT_2SAMPLES_gc;
                EVSYS.CH0CTRL |= (uint8_t) qIndexState | EVSYS_QDIEN_bm;
            }
            break;
        case 2:
            EVSYS.CH2MUX = qPinInput;
            EVSYS.CH2CTRL = EVSYS_QDEN_bm | EVSYS_DIGFILT_2SAMPLES_gc;
            if(useIndex){
                EVSYS.CH3MUX = qPinInput + 2;
                EVSYS.CH3CTRL = EVSYS_DIGFILT_2SAMPLES_gc;
                EVSYS.CH2CTRL |= (uint8_t) qIndexState | EVSYS_QDIEN_bm;
            }
            break;
        case 4:
            EVSYS.CH4MUX = qPinInput;
            EVSYS.CH4CTRL = EVSYS_QDEN_bm | EVSYS_DIGFILT_2SAMPLES_gc;
            if(useIndex){
                EVSYS.CH5MUX = qPinInput + 2;
                EVSYS.CH5CTRL = EVSYS_DIGFILT_2SAMPLES_gc;
                EVSYS.CH4CTRL |= (uint8_t) qIndexState | EVSYS_QDIEN_bm;
            }
            break;
        default:
            return false;
    }
    return true;
}

```

```

/*****
*** Funktionsname:  QDEC_Total_Setup          ***
*** Erstellt von Atmel Corporation (AVR1600)  ***
*** Beschreibung:  Konfiguriert den verwendeten Timer/Counter ***
***                                     ***
*** Parameter: qTimer    Verwendeter Timer          ***
*** Parameter: qEventChannel  Verwendeter EventChannel ***
*** Parameter: lineCount  Anzahl der Signale/Umdrehung ***
***                                     ***
*****/
void QDEC_TC_Dec_Setup(TC1_t * qTimer, TC_EVSEL_t qEventChannel, uint16_t lineCount)
{
    /* Configure TC as a quadrature counter. */
    qTimer->CTRLD = (uint8_t) TC_EVACT_QDEC_gc | qEventChannel;
    qTimer->PER = (lineCount * 4) - 1;
    qTimer->CTRLA = TC_CLKSEL_DIV1_gc;
}

```