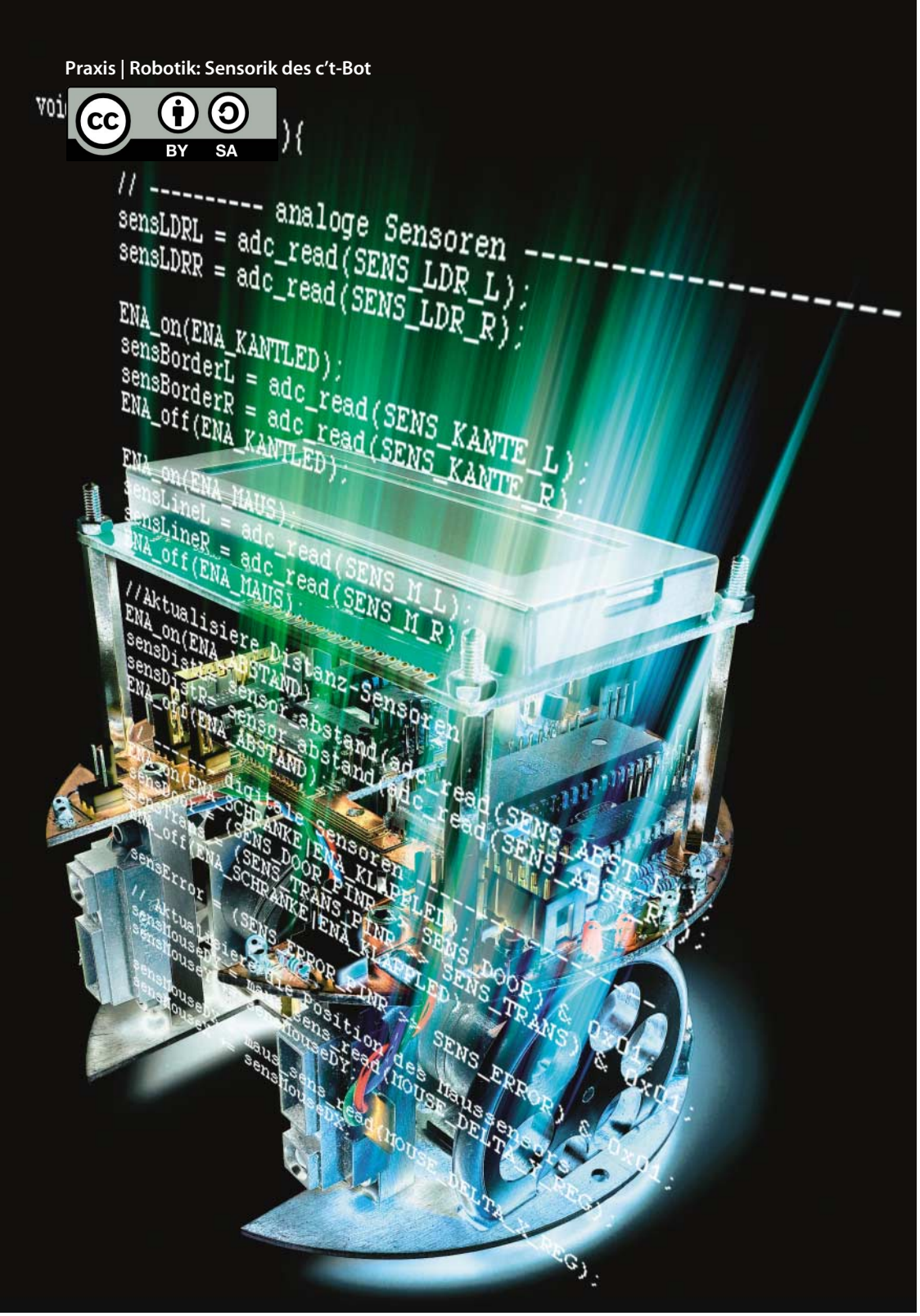


Benjamin Benz

Nervensystem

Programmierung des c't-Bot von der Pike auf

Wie halte ich meinen c't-Bot davon ab, sich permanent im Kreis zu drehen? Wie funktionieren seine Fühler und warum liefert der Maussensor so seltsame Werte? Nach der Montage des c't-Bot dürfte unweigerlich die eine oder andere dieser Fragen auftauchen. Um sie zu beantworten, muss man dem Bot etwas tiefer unter die Haube blicken.

Diese Folge der c't-Bot-Artikelserie widmet sich der Programmierung auf niedrigster Ebene. Es soll dabei aber nicht nur darum gehen, Bits über I/O-Leitungen zu schubsen, den Mikrocontroller per Interrupt zu unterbrechen und Analog-Digital-Umsetzer zu kommandieren. Auch die Aufbereitung der Sensordaten steht auf dem Programm. Wer bereit ist, seinen Roboter zu tunen, kann ihn mit diesem Wissen leicht an die jeweilige Umgebung anpassen. Wem all der C-Code hier spanisch vorkommt, findet in der Link-Liste auf der Projektseite einführende Tutorials. Damit Sie nicht alles selbst schreiben müssen, bietet unsere Code-Basis Bibliotheken für alle hier besprochenen Funktionen und lauffähigen Testprogramme.

Der Mikrocontroller (ATmega32) des c't-Bot – auf dem Gesamtschaltplan [1] rechts oben – kommuniziert über 32 Ein-/Ausgabeleitungen oder genauer Pins mit dem Rest der Welt. Er organisiert diese Pins als vier Ports zu je acht Leitungen. Jeden dieser Pins kann man als so genannten General Purpose I/O (GPIO) konfigurieren. Dann kümmert sich die selbst geschriebene Software darum, dass er zum richtigen Zeitpunkt die korrekte Richtung (Eingabe oder Ausgabe) hat und legt entweder einen Pegel an oder liest ihn aus. Alternativ dazu kann jeder einzelne Pin eine oder mehrere Spezialfunktionen übernehmen, dann kümmern sich spezielle Funktionseinheiten des Prozessors um ihn. So kann PD0 (Port D, Pin 0 alias Anschlussbeinchen 14) auch als Empfangsleitung (RXD) der seriellen Schnittstelle fungieren.

Wegen der Funktionsfülle des c't-Bot mussten wir bei der Pin-Zuordnung einige Kompromisse eingehen, denn alle 32 Port-Pins des Controllers sind belegt. Nicht jede im Datenblatt verzeichnete Spezialfunktion ist daher ohne weiteres zugänglich. Für versierte Bastler haben wir jedoch alle Ports auf Stiftleisten herausgeführt (J5, J6, J7, J8), sodass man über diese nach- und umrüsten kann – dabei sollte man sorgfältig prüfen, ob die eigene Erweiterung nicht mit bereits vorhandenen Funktionen kollidiert.

Im Folgenden wollen wir die Standardkonfiguration des c't-Bot näher erklären. Der Demo-Code kümmert sich zwar um alle

Low-Level-Belange, aber ein wenig Kenntnis der tiefer liegenden Schichten hilft bei der Fehlersuche und bei künftigen Erweiterungen.

Nach einem Reset konfiguriert der Controller alle Leitungen als Eingänge, damit definitiv kein Kurzschluss entsteht. Zu Beginn eines eigenen Programms muss man die einzelnen Pins konfigurieren. Für jeden der vier Ports (A, B, C, D) steuern jeweils drei Register (PORTx, PINx, DDRx) die Pins. Das Data Direction Register (DDRx) legt fest, ob es sich um einen Ein- oder Ausgang handelt. Die Soll-Werte für Ausgangs-Pins schreibt man in PORTx, und PINx gibt Aufschluss über die an Eingangs-Pins anliegenden Pegel. Dabei steht jedes Bit für einen einzelnen Pin, eine Eins in DDRx kennzeichnet einen Ausgang, eine Null einen Eingang. Der C-Befehl

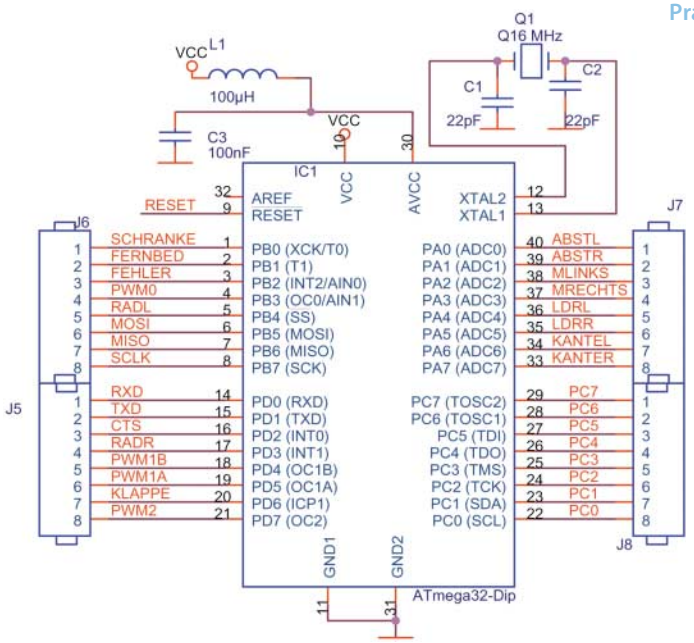
```
DDRC |= (1<<6);
```

schreibt nur eine Eins an das sechste Bit des Registers DDRC. Beim Setzen und Löschen stellen die Operatoren „|=“ und „&=“ sicher, dass keine anderen Bits verändert werden. „|=“ ist lediglich eine Kurzform der Schreibweise `PORTC = PORTC | (1<<6);`. Das macht PC6 zum Ausgang – dieser kontrolliert übrigens die Drehrichtung des linken Motors. Um nicht jedes Mal mit Pin-Nummern jonglieren zu müssen, führt unser Code ein paar Konstanten ein:

```
#define BOT_DIR_L_DDR    DDRC
#define BOT_DIR_L_PIN    (1<<6)
BOT_DIR_L_DDR |= BOT_DIR_L_PIN;
```

Die Kontrolle des Ausgangs erfolgt über das PORTx-Register. Auch hier steht jedes Bit für einen Pin. Setzt man das jeweilige Bit (`PORTC |= (1<6);`), zieht der Controller den Pin auf die stabilisierte Versorgungsspannung Vcc, löscht man es (`PORTC &= ~(1<<6);`), zieht er ihn gegen Masse.

Sowohl gegen Vcc als auch gegen Masse kann der ATmega maximal 40 mA pro Pin, aber insgesamt nicht mehr als 200 mA treiben. Handelt es sich um einen Eingang, so kann man über die Register PINx die Eingangspegel auslesen. Bei Eingängen legt das PORTx-Register fest, ob der Controller seine internen Pullup-Widerstände aktiviert (Bit gesetzt). Diese ziehen unbeschaltete Leitungen auf einen definierten Pegel (Vcc).



Die 32 Port-Pins des ATmega32 liegen alle auf Stiftleisten. So sind sie leicht für künftige Erweiterungen, aber auch zum Nachmessen erreichbar.

Damit der C-Compiler die Register adressieren kann, bindet man das Header-File io.h ein:

```
#include <avr/io.h>
```

Multiplikatoren

Da die 32 I/O-Leitungen nicht ausreichen, um alle Peripherie des c't-Bot direkt anzusprechen, hängen die LEDs, das Display sowie eine ganze Reihe von Schaltern an insgesamt drei Schieberegistern (74HC595). Nachdem man sie nacheinander über eine Leitung mit 8 Bits versorgt hat, geben sie diese auf acht getrennten Leitungen aus. Fünf Prozessor-Pins steuern so insgesamt 24 Leitungen an.

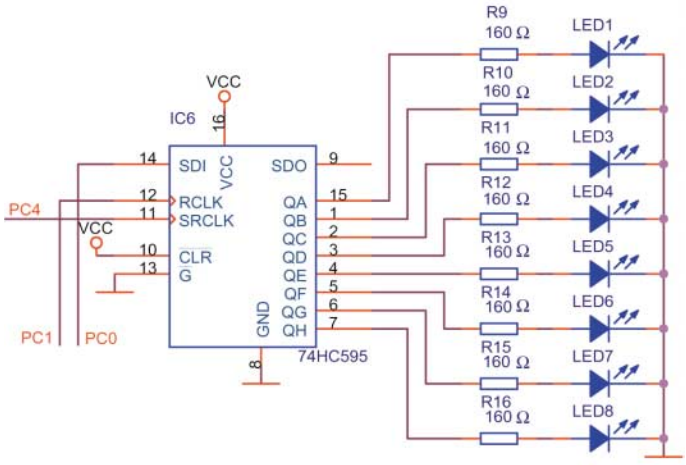
Die 74HC595-Chips besitzen vier Eingänge: Über den Pin SDI nehmen sie die Daten seriell entgegen. Sobald der Pegel an Pin SRCLK von 0 auf 1 wechselt, übernehmen sie ein Datenbit in ihr internes Register.

```
#define SHIFT_OUT 0x1F
#define SHIFT_PORT PORTC
SHIFT_PORT &= ~SHIFT_OUT;
for (i=8; i>0; i--){
  SHIFT_PORT |= ((data >> 7)& 0x01);
  SHIFT_PORT |= SRCLK ;
  data= data << 1;
  SHIFT_PORT &= ~SHIFT_OUT;
}
```

Nach der Übertragung von acht Bit veranlasst eine steigende Flanke auf Pin RCLK die Übertragung der Daten an den Ausgang.

```
SHIFT_PORT |= RCLK;
```

Die Datenleitungen aller drei Schieberegister hängen zusammen. Ohne Takt auf SRCLK ignorieren die 74HC595-Chips eingehende Daten. Der vierte Eingang



Ein Schieberegister empfängt Daten seriell und verteilt sie dann auf seine parallelen Ausgänge. So lassen sich die knappen I/O-Leitungen des Prozessors erweitern.

G legt fest, ob das Register die Ausgänge aktiv gegen Masse oder Vcc treiben soll oder sie von den Pins abkoppelt. Bei IC5 und IC6 ist kein Umschalten nötig, daher liegen ihre G-Leitungen fest auf Masse; sie sind dauerhaft aktiv. Anders sieht das bei IC4 aus, das sich um das LC-Display kümmert. Dieses aktiviert seinen Ausgang nur kurz beim Schreiben der Daten – wenn PC1 (SRCLK) Low ist. In der übrigen Zeit kann der Mikrocontroller über PC5 das Busy-Flag des Displays auslesen. Die restliche Beschaltung des Displays ist trickreich, um mit möglichst wenig I/O-Leitungen auszukommen. Daher hängt auch die SRCLK-Leitung von IC4 mit den RCLK-Leitungen der beiden anderen zusammen. Wer sich für die Details der Ansteuerung des Displays interessiert, findet sie im Quelltext (mcu/display.c). Für die meisten Aufgaben sollten aber die in display.h definierten Funktionen ausreichen. Beispielsweise geben drei Code-Zeilen:

```
display_cursor(1,1);
sprintf(display_buf,"c't-Roboter");
display_buffer();
```

einen Willkommensgruß in der ersten Zeile des Displays aus. Auch die LEDs hängen an einem Schieberegister und lassen sich mit einigen vorgefertigten Routinen (siehe led.h) steuern:

```
LED_on(LED_GRUEN);
LED_off(LED_WEISS);
```

Stromsparer

Energiesparen liegt nicht nur voll im Trend, bei einem Roboter zahlt es sich auch direkt durch längere Akkulaufzeit aus. Da für die meisten Sensoren nur vergleichsweise selten Arbeit anfällt und sie relativ viel Leistung aufnehmen, lohnt es, sie nur bei Bedarf einzuschalten. Zuständig dafür sind die acht ENA-Leitungen, die das Schieberegister IC5 zur Verfügung stellt. Die daran hängenden MOSFET-Transistoren trennen die Sensoren bei Bedarf von der Stromversorgung ab. So kontrolliert beispielsweise TR5 die Stromzufuhr zur integrierten Leuchtdiode von U1, dem Klappensensor. Je nach Vorwiderstand lassen sich so pro CNY-70-Sensor bis zu 50 mA einsparen. Auch bei jedem der beiden IR-Distanzsensoren (GP2D12) sind nochmals bis zu 50 mA zu holen.

Im C-Programm kümmern sich die Routinen ENA_on(), ENA_off() oder ENA_set() um die Steuerung der einzelnen Leitungen. Sie erwarten jeweils eine Bitmaske mit den zu schaltenden Kanälen; einige Konstanten aus ena.h erleichtern den Zugriff:

```
ENA_on(ENA_ABSTAND);
```

aktiviert beispielsweise die Distanzsensoren. Da die Enable-Leitungen ein Schieberegister benutzen, muss man bei Veränderungen an unserer Firmware aufpassen, dass man dabei nicht Display-Zugriffen in die Quere kommt. Wer mit einem Multimeter nachmessen möchte, sollte beachten, dass die Transistoren bei einem Low-Pegel durchschalten und bei High sperren.

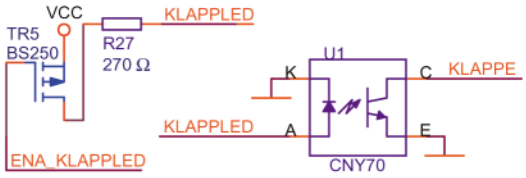
Schwarz oder Weiß

Für einige der Sensoren reicht es aus zu wissen, welchen von zwei Zuständen sie erkennen. Die Abfrage dieser binären Daten ist nicht nur einfach, sondern insbesondere auch sehr schnell. Der Mikrocontroller schafft das in einem Taktzyklus (62,5 ns). Zu den binären Sensoren gehören beim c't-Bot die Lichtschranke zur Überwachung des Transportfaches, der Klappensensor, die Rad-Encoder, die Fehlerdetektion und der IR-Empfänger für die Fernbedienungssignale. Fragt man den Klappensensor U1 ab, so steht in der Variablen sensDoor eine 1, wenn der Sensor nicht verdeckt ist und eine 0, wenn man ihm ein reflektierendes Material vorhält. Wieder erleichtern einige Konstanten die Lesbarkeit:

```
ENA_on(ENA_KLAPPLED);
sensDoor = (SENS_DOOR_PINR ⤸
        >> SENS_DOOR) & 0x01;
ENA_off(ENA_KLAPPLED);
```

Später kann man so detektieren, in welcher Position der Schwenkarm der Klappe sich befindet. Bei den Rad-Encodern sieht es ähnlich aus.

Am Pin PB2 hängt ein Signal, das der Schaltplan [1] als FEHLER beschreibt. Eine einfache Schaltung sorgt dafür, dass dieses Signal auf Low geht, sobald entweder die Batteriespannung zu weit absinkt (5,6 V, das entspricht einer Zellenspannung von 1,1 V) oder an einem der beiden Servo-Ports zu viel Strom (je nach Akkuspannung 118 bis 178 mA) gezogen wird. Die Unterscheidung, woher der Fehler kommt, obliegt der Software: Bleibt der Pin auch bei deaktiviertem Servo Low, so ist die Batterie leer. Ist er es nur bei angeschaltetem Servo, so blockiert dieser. Im Normalfall sollte der Controller hier eine Eins auslesen.



Lässt der Mikrocontroller die Leitung ENA_KLAPPLED auf Low ziehen, so versorgt der Transistor TR5 den CN70-Sensor mit Strom. Braucht man den Sensor gerade nicht, ist die Leitung auf High.

Grau in Grau

Die Distanz-, Licht-, Linien- und Abgrundsensoren liefern wesentlich detailliertere Informationen als ein schlichtes An oder Aus. Ihre analoge Ausgangsspannung korreliert mit der physikalischen Größe, die sie messen. Der Zusammenhang ist dabei meist alles andere als linear.

Der ATmega besitzt acht analoge Eingänge, deren Spannung er mit einer Auflösung von 10 Bit digitalisieren kann. Liefert der integrierte A/D-Umsetzer 0 zurück, so entspricht das 0 V, 1023 bedeuten, dass der zu messende Pegel größer oder gleich der Referenzspannung ist. Der Umsetzer verwendet dabei entweder eine interne Referenz (2,56 V) oder die Betriebsspannung von Pin AVCC (5,0 V). Welche man für die jeweilige Messung benutzt, legen die Bits REFS0 und REFS1 im Register ADMUX fest. Für AVCC als Referenz setzt man das REFS0-Bit im ADMUX-Register. Die anderen Bits von ADMUX wählen den Kanal:

```
ADMUX= _BV(REFS0);
ADMUX |= (channel & 0x07);
```

_BV(REFS0) ist nur eine vereinfachte Schreibweise von (1<<REFS0). Bevor der Umsetzer loslegt, fehlen noch ein paar Einstellungen im ADSCRA-Register, unter anderem, um den Takt auszuwählen:

```
ADCSRA= _BV(ADPS2) | _BV(ADPS1)| ⤸
   _BV(ADPS0) | _BV(ADEN)| _BV(ADSC);
```

Nun heißt es warten, bis der Umsetzer mit einem Resultat aufwartet. Das zeigt er durch Löschen des ADSC-Bits im ADSCRA-Register. Das Ergebnis der Wandlung steht im Register ADC bereit:

```
while ( (ADCSRA & (1<<ADSC)) != 0){}
result= ADC;
```

Wer keine CPU-Zeit vergeuden möchte, kann den A/D-Umsetzer auch im Interrupt-Modus betreiben. Dann unterbricht er das laufende Programm und ruft eine spezielle Funktion (Interrupt-Service-Routine, ISR) auf, sobald die A/D-Wandlung fertig ist. Da dies jedoch komplexer und schlechter durchschaubar ist sowie einige andere Teile des Codes beeinflusst, haben wir hier darauf verzichtet. Wer mag, kann uns aber gerne einen Patch schicken, der das nachrüstet – wie man solche Patches erstellt und einreicht, beschreibt die Projektseite [1].

Begradigung

Die Messergebnisse der Abstandssensoren hängen leider nicht linear mit den realen Distanzen zusammen. Die Grafik auf der nächsten Seite zeigt die Kennlinie der GP2D12-Sensoren. Ignoriert man den Nahbereich unterhalb von rund acht Zentimetern – in dem die Spannung wieder rapide sinkt – und korrigiert die Kurve um einen Offset, so führt schon eine Näherung durch eine einfache Hyperbel zu recht guten Ergebnissen:

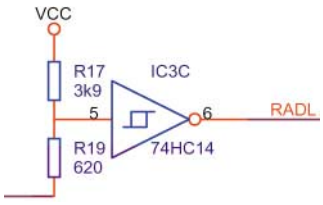
$$D = a / (x - b)$$

D ist die Distanz und x der Messwert, wie er vom A/D-Umsetzer kommt. Die Steigung a der Hyperbel und ihren Offset b ermittelt man am lebenden Objekt. Bereits zwei Messpunkte (x_1 und x_2) reichen aus, um sie eindeutig zu bestimmen. Diese sollten weder zu nah beisammen noch an den Rändern des Messbereichs liegen, sonst steigt der Linearisierungsfehler. Mit ein wenig Experimentieren sollten sich geeignete Werte finden lassen:

$$a = (x_2 - x_1) * D_2 * D_1 / (D_1 - D_2)$$
$$b = (D_2 * x_2 - D_1 * x_1) / (D_2 - D_1)$$

Diese Korrekturen übernimmt die C-Funktion sensor_abstand(), die Parameter legt man als Präprozessorkonstanten in der Datei sensor_correction.h fest. Passt man diese Datei an eigene Bedürfnisse an, so schützt ein Eintrag in der Datei .cvsignore sie bei künftigen Abgleichen mit unserem CVS-Code-Repository. Details zu diesem Mechanismus stehen auf der Projektseite in der FAQ [1].

Wem die Genauigkeit nicht ausreicht, der muss eine komplexere Näherungskurve verwenden oder eine Tabelle aufstellen, die für verschiedene Abschnitte unterschiedliche Koeffizienten definiert.

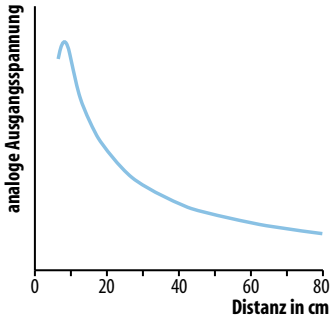


Übersteigt das analoge Signal der Rad-Encoder etwa 2,4 V, so interpretiert es der Schmitt-Trigger 74HC14 als High, unterhalb von rund 1,4 V als Low.

1, 2, 3 … Unterbrechung

Eine ganze Reihe von Vorgängen im c't-Bot soll zu genau definierten Zeitpunkten und vor allem unabhängig vom Hauptprogramm geschehen. So brauchen die Motoren ein pulsweitenmoduliertes Signal (PWM) [2], und den IR-Empfänger muss man in genau definierten Intervallen auslesen. Auch die Rad-Encoder müssen recht häufig abgefragt werden. Dazu generieren die Timer-Einheiten des ATmega32 in einstellbaren Intervallen Interrupt-Signale. Der Prozessor unterbricht daraufhin seine aktuelle Arbeit und verzweigt in die zugehörige Interrupt-Service-Routine (ISR). Hat diese ihre Arbeit erledigt – zum Beispiel einen Pegel vom IR-Sensor ausgelesen und gespeichert –, kehrt der Prozessor zum eigentlichen Code zurück.

Der ATmega32 bringt drei solcher Timer mit (zwei mit 8- und einen mit 16-Bit-Zähler), von denen jeder mehrere Interrupts auslösen kann. Der Timer2 (8 Bit) erzeugt in unserer Firmware beispielsweise alle 179 µs einen Interrupt zur Abfrage des IR-Empfängers und der Rad-En-



Die Ausgangsspannung der Distanzsensoren hängt nicht linear mit dem Abstand zusammen. Bei rund 8 cm hat sie ein Maximum und fällt von dort in beide Richtungen ab.

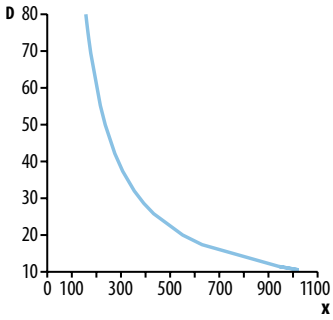
Passt man die Parameter der Ausgleichskurve gut an, so linearisiert sie die Abstandsmessergebnisse in weiten Bereichen recht gut.

coder. Er zählt dazu ein 8-Bit-Register (TCNT2) mit jedem Clock-Signal um eins hoch, vergleicht es dann mit dem Compare-Register (OCR2) und löst bei Gleichheit einen Interrupt aus. Wie schnell das Hochzählen geschieht, entscheidet der verwendete Zählertakt.

Ein Vorteiler stellt verschiedene Frequenzen zur Verfügung, die er durch Teilen aus dem Systemtakt (16 MHz) erzeugt. Dies ist nötig, da das Vergleichsregister nur 8 Bit breit ist – in 179 µs würde der Timer bei vollem Takt bis 2864 zählen. Im Timer-Control-Register (TCCR2) legen drei Bits (CS20 bis CS22) fest, welchen Takt der Zähler verwendet (Tabelle im Datenblatt). Da der Acht-Bit-Timer nur bis 255 zählen kann fällt die Wahl hier auf ein Teilerverhältnis von 1:64 und somit einen Takt von 250 kHz. Der Vergleichswert ergibt sich dann zu (250 kHz × 179 µs -1 = 44):

```
#define XTAL 16000000 // [Hz]
#define TIMER_2_CLOCK     5619 // [Hz]
...
OCR2 = ((XTAL/64/TIMER_2_CLOCK) - 1 );
TCCR2 = _BV(WGM21) | _BV(CS22);
```

Das WGM21-Bit sorgt dafür, dass der Zähler nach jedem Erreichen des Vergleichswertes wieder bei null anfängt zu zählen. Den Anfangswert schreibt man direkt in das Timer-Register und aktiviert zuletzt die Interrupts, die per Default abgeschaltet sind:




```
TCNT2 = 0;
TIMSK |= _BV(OCIE2);
sei();
```

Das OCIE2-Bit im TIMSK-Register schaltet den Vergleichs-Interrupt ein und das Makro sei(), das in avr/interrupt.h definiert ist, aktiviert systemweit alle Interrupts.

Die Interrupt-Service-Routine unterscheidet sich dank einiger Makros aus avr/signal.h kaum von einer gewöhnlichen C-Funktion. Sie beginnt mit dem Schlüsselwort SIGNAL, gefolgt vom Namen des zu behandelnden Interrupts:

```
SIGNAL (SIG_OUTPUT_COMPARE2){
  ir_isr();
  bot_encoder_isr();
}
```

Während der Ausführung der Funktion sind alle anderen Interrupts blockiert. Wer das nicht will, nutzt statt SIGNAL das Makro INTERRUPT, muss dann aber selbst dafür sorgen, dass sich überlappende Aufrufe nicht gegenseitig beeinflussen. Grundsätzlich gilt, dass ein Programm möglichst wenig Zeit in ISRs verbringen soll. Aufwendige Operationen gehören dort nicht hinein.

Die Interna der Dekodierung von RC5-Fernbedienungssignalen – um die sich die Routine ir_isr() kümmert – haben wir bereits im Projekt c't-Netz-Schalter ausführlich beschrieben. Der Artikel steht mittlerweile auf der Netz-Schalter-Projektseite [7].

Wer Lust verspürt, das modernere und leicht erweiterte Protokoll RC6 zu implementieren, findet in der Link-Liste auf der c't-

Bot-Projektseite einige Informationen.

Auch die Auswertung der Rad-Encoder ist kein Hexenwerk. Die Funktion bot_encoder_isr() kümmert sich darum und beachtet dabei auch die Drehrichtung der Motoren. Ihre Resultate stehen in den globalen Variablen sensEncL und sensEncR dem Hauptprogramm zur Verfügung.

Am Puls der Zeit

Wie die Pulsweitenmodulation grundsätzlich funktioniert und wie man damit die Geschwindigkeit der Motoren regelt, haben wir bereits in [2] ausführlich beschrieben. Für die Erzeugung der Signale zeichnet Timer1 (16 Bit) verantwortlich. In einer speziellen PWM-Betriebsart zählt er mit 8 Bit Breite ständig auf und ab. Solange sein Wert über dem Vergleichswert in OC1A liegt, gibt der PWM-Pin PD5 ein High aus, Entsprechendes gilt für PD4 und den Vergleichswert OC1B. Je größer also der Vergleichswert, desto langsamer drehen die Motoren.

Um von den Verhaltensroutinen aus das PWM-Verhältnis festzulegen, muss man lediglich die Vergleichswerte setzen. Diese Aufgabe übernimmt die Funktion bot_motor():

```
#define PWM_L OCR1A
#define PWM_R OCR1B
void bot_motor(int16 left, int16 right){
  PWM_L = 255-abs(left);
  PWM_R = 255-abs(right);
```

Der Timer kümmert sich komplett autark um die Erzeugung der PWM-Signale, Rechenzeit geht dafür nicht verloren. Die Drehrichtung der Motoren legen zwei ganz normale I/O-Pins fest:

```
if (left > 0 )
  BOT_DIR_L_PORT |= BOT_DIR_L_PIN;
else
  BOT_DIR_L_PORT &= ~BOT_DIR_L_PIN;

if (right < 0 )
  BOT_DIR_R_PORT |= BOT_DIR_R_PIN;
else
  BOT_DIR_R_PORT &= ~BOT_DIR_R_PIN;
}
```

Die beiden Motoren müssen entgegengesetzt angesteuert werden, da sie um 180 Grad gedreht eingebaut sind.

Mäuschen spielen

Der Maussensor besitzt bereits viel Eigenintelligenz. Die Bilder seiner 19×19-Pixel-Kamera wertet er intern aus. Der ATmega32 muss nur noch über eine synchrone serielle Schnittstelle (PB5 bis PB7) nach den Positionsänderungen fragen. Diese Delta-X- und Delta-Y-Werte setzt der Sensor-Chip nach dem Auslesen selbstständig zurück. Daher kumuliert der Mikrocontroller alle gelesenen Einzelwerte in den globalen Variablen sensMouseY und sensMouseX.

Der Maus-Chip konfiguriert sich weitgehend selbst, lediglich die Stromsparmodi lassen sich durch das Beschreiben eines Registers beeinflussen. So schaltet das Kommando

```
maus_sens_write(MOUSE_CONFIG_REG,
MOUSE_CFG_POWERDOWN);
```

den gesamten analogen Schaltungsteil des Sensors ab. Die Konstante MOUSE_CFG_FORCEAWAKE zwingt den Sensor, immer wach zu bleiben. MOUSE_CFG_NORMAL stellt einen Kompromiss dar und gestattet es dem Sensor, nach einer Sekunde Inaktivität einzuschlafen. Alle benötigten Funktionen und Konstanten listet mouse.h auf.

Gemachtes Nest

Alle hier besprochenen Low-Level-Funktionen sind im c't-Bot-Code bereits implementiert. Jede Funktionseinheit hat im Unterverzeichnis mcu eine eigene .c-Datei. Die dazugehörigen Header-Dateien (Endung .h) liegen im Verzeichnis include.

Nach einem Reset beginnt der Mikrocontroller, das Hauptprogramm (main()) in der Datei ct-Bot.c abzuarbeiten. Dieses initialisiert zuerst alle Einzelteile (init()). Danach laufen die Timer autonom und kümmern sich um die Erzeugung der PWM-Signale, das Dekodieren der Fernbedienungssignale (ir-rc5.c: ir_isr()) und die Abfrage der Rad-Encoder (sensor-low.c:bot_encoder_isr()). Alle anderen Sensoren fragt das Hauptprogramm mit der Routine bot_sens_isr() aus der Datei sensor-low.c immer wieder selbst ab.

Unsere Bibliotheken stellen die Resultate in globalen Variablen allen weiteren Programmteilen zur Verfügung. Die Include-Datei sensor.h listet sie alle auf. Diese Sensorvariablen liefern die Entscheidungsgrundlagen für das Verhaltenssystem, das bereits der letzte Artikel [5] ausführlich beschrieben hat.

Eine Ausnahme bilden lediglich die empfangenen IR-Codes, die erst in der Variablen RC5_Code landen, wenn die Funktion rc5_control() aus rc5.c sie bearbeitet hat. Diese Funktion holt die Resultate der Interrupt-Service-Routine ab und prüft, ob die gedrückte Taste mit einer Aktion verknüpft ist. Diese Aktion beeinflusst dann das Verhalten des Roboters. Sie ist nichts anderes als ein Zeiger auf eine C-Funktion:

```
typedef void
   (*RemCtrlFunc)(RemCtrlFuncPar *par);
```

Die Zuordnung erfolgt über eine Liste mit Einträgen vom Typ:

```
typedef struct {
  uint16 command;
  RemCtrlFunc func;
  RemCtrlFuncPar par;
} RemCtrlAction;
```

Dabei bekommen die RC5-Kommandos (command) eine Funktion zugeordnet (func), die dann mit den Parametern par aufgerufen wird. Die möglichen Tasten deklariert die Datei rc5-codes.h. Dieser etwas komplex anmutende Überbau erlaubt sehr kompakte Tastenzuordnungen in einem Array, wie die drei folgenden Beispiele zeigen:

```
static RemCtrlAction gRemCtrlAction[] = {
{RC5_CODE_PWR, rc5_bot_set_speed,
  {BOT_SPEED_STOP, BOT_SPEED_STOP}},
{RC5_CODE_UP, rc5_bot_change_speed,
  {10, 10 }},
{RC5_CODE_5, rc5_bot_goto, { 0, 0 }}
}
```

Steckverbinder des c't-Bot

Stecker	Verwendung
J1	Servo für die Transportfachverriegelung
J2	Erweiterungsservo
J3	Spannungsversorgung für Erweiterungsmodule
J4	Erweiterungsmodul, angebunden über serielle Schnittstelle
J5	Zugang zu CPU-Port D
J6	Zugang zu CPU-Port B
J7	Zugang zu CPU-Port A
J8	Zugang zu CPU-Port C[1]
ST1	Akkupack
ST2	Motor links
ST3	Motor rechts
ST4	Display
ST5	ISP-Programmieradapter
ST6	ISP-Programmieradapter
ST7	Maussensorplatine
ST8	Sensorplatine links
ST9	Sensorplatine rechts
BR1	Drahtbrücke für LCD-Beleuchtung

[1] Achtung: Belegung von J8 weicht von der der anderen Stiftleisten ab

Hilfsfunktionen wie rc5_bot_goto() teilen den Nutzerwunsch dann dem Verhaltenssystem mit. Diese strikte Trennung erlaubt es, sich auf die intelligente Steuerung des Roboters zu konzentrieren und die tieferen Schichten außer Acht zu lassen.

Diät

Der C-Cross-Compiler gcc übersetzt bereitwillig beliebig komplexen Code, man kann also theoretisch wie vom PC her gewohnt programmieren. Der Mikrocontroller arbeitet aber intern nur mit 8 Bit. Das bedeutet, dass er jede Operation mit komplexeren Datentypen in vielen Einzelschritten erledigen muss. Besonders Gleitkommaoperationen wie beispielsweise die Multiplikation:

```
float faktorLeft = 1.0;
faktorLeft  *= 0.9;
```

kosten viel Rechenzeit. Man sollte sie daher sehr sparsam einsetzen. Auch andere Ressourcen sind sehr begrenzt: Für Programmcode und Konstanten stehen insgesamt 32 KByte Flash-Speicher zur Verfügung. Alle Variablen müssen in die 2 KByte SRAM hineinpassen. Für Parameter, die einen Neustart überleben sollen, gibt es dann noch 1 KByte EEPROM. String-Operationen und damit auch Ausgaben auf das Display schlucken schnell sehr viel RAM. Damit Datenstrukturen wie der Verhaltensstack schlank bleiben, sollte man sich bei ihrer Definition auf das Wesentliche beschränken. Stößt man dennoch an die Grenzen des Controllers, so kann man in der Header-Datei ct-Bot.h unbenötigte Features deaktivieren. Kommentiert man beispielsweise

```
#define DISPLAY_AVAILABLE
```

aus, indem man davor /* und danach */ schreibt, so entfernt der Präprozessor allen Display-Code. Das spart rund 2 KByte Flash-Speicher und 100 Byte RAM. Weitere Details zu den Speichertypen des Controllers beschreibt der Artikel auf [7].

Evolution

Beim genauen Betrachten der Platine fallen einige unbelegte Steckverbinder und ein Loch in der Mitte auf. Jetzt schon über Details denkbarer Hardware-Er-

weiterungen zu berichten, wäre zwar verfrüht, aber die Begriffe Kameramodul und Funkverbindung sind gelegentlich in der Redaktion ebenso zu hören wie Speichererweiterung oder CPU-Board. Was würde sich für eine Anbindung solcher Module besser eignen als der Port J4? Welche der Optionen das Rennen machen und genaue Zeitpläne stehen bislang nicht fest. Zuerst soll eine Mechanik zur Verriegelung des Transportfaches kommen. Der dafür nötige Servo passt in das ominöse Loch und wird über den Stecker J1 angesteuert. Der zugehörige Sensor (U1) war oben schon erwähnt.

Für eigene Erweiterungen dürfte sich wohl am besten der I^2C-Bus eignen. Für diesen bietet der Handel nicht nur diverse Sensoren, sondern auch Port- und Speichererweiterungen an. Dabei kann man bis zu 128 Geräte kaskadieren, die sich dann die Brutto-Übertragungsrate des Fast-Mode von 400 kBit/s teilen. Aus lizenzrechtlichen Gründen findet man diesen seriellen Bus in den Atmel-Datenblättern nicht unter dem Namen I^2C, sondern als „Two-Wire Serial Interface".

An den beiden I^2C-Leitungen des Controllers (PC0 und PC1) hängen zwar bereits die Schieberegister und die Steuerleitungen des Displays, aber das sollte nicht stören. Passt man bei der Programmierung auf, die anderen Steuerleitungen der Schieberegister und des Displays nicht zu beeinflussen, steht einer Zweitnutzung der Pins nichts im Wege. Zugänglich sind sie über J8.

Come together

Am Samstag, den 11. März 2006, wollen wir auf der CeBIT in Hannover eine kleine Entwicklerrunde veranstalten. Ab 17 Uhr wird es am Heise-Stand (Halle 5, Stand E38) eine kurze Einführung in das Projekt und reichlich Gelegenheit geben, mit uns über mögliche Erweiterungen, Programmierung der c't-Bots, netzwerkfähige Simulatoren und alle anderen Belange rund um c't-Bot und c't-Sim zu diskutieren.

Die nächste Folge der Serie widmet sich wieder etwas höheren Ebenen der Programmierung. Damit sich die c't-Bots clever verhalten, wollen wir verschiedene Strategien vorstellen und zeigen, wie sich diese kombinieren lassen. Für den nächsten Hardware-Artikel stehen Regelkreise und die Interpretation der Maussensordaten auf dem Leseplan. (bbe)

Literatur

[1] Webseite zum c't-Bot-Projekt: www.heise.de/ct/ftp/projekte/ct-bot

[2] Benjamin Benz, Carl Thiede, Thorsten Thiele, Spielgefährten, Roboter für Löter, Simulator für Soft-Werker, c't 2/06, S. 130

[3] Benjamin Benz, Peter König, virtuelle Spielgefährten, Simulator für c't-Bots, c't 3/06, S. 186

[4] Benjamin Benz, Carl Thiede, Thorsten Thiele, Hallo Welt!, Aufbau und Inbetriebnahme des c't-Bot, c't 4/06, S. 208

[5] Benjamin Benz, Peter König, Lasse Schwarten, Drängelnde Spielgefährten, Kollisionen und Sensoren für den c't-Sim, neues Verhalten für den c't-Bot, c't 5/06, S. 224

[6] Webseite zum c't-Projekt COM-auf-LAN: www.heise.de/ct/ftp/projekte/com2lan

[7] Webseite zum c't-Netz-Schalter-Projekt: www.heise.de/ct/ftp/projekte/netz-schalter

Soft-Link 0606264