Julian von Mendel, 2008/03/07

Beschreibung unseres Roboking-Projekts

Das primäre Ziel war es natürlich, die unter http://roboking.de erklärte Aufgabenstellung zu lösen. Grob zusammengefasst ist es das Ziel, einen Roboter zu bauen, der 20 Tennisbälle auf einem 6m²-Spielfeld so schnell wie möglich findet und an den Spielfeldrand transportiert. (Weitere Regeln und der gegnerische Roboter nehmen keinen wesentlichen Einfluss auf den Aufbau des Roboters, siehe hierzu die Roboking-Internetseite.) Um die Bälle zu sammeln und zum Spielfeldrand zu transportieren zogen wir mehrere grundsätzliche Lösungsansätze in Betracht:

1. Zufälliges Abfahren vom Spielfeld, dann alle dabei getroffenen Bälle zum Rand schieben.
2. Gezieltes Suchen der Bälle mittels drehbarer Entfernungssensoren, Unterscheidung von feindlichen Robotern/Wänden/Ringhalterungen (s. Aufgabenstellung)/Tennisbällen anhand der Objektbreite.
3. Gezieltes Suchen mittels einer Kamera.

Die erste Möglichkeiten halten wir für ineffizient. Die zweite Möglichkeit hielten wir für mittelmäßig schnell (gängige Sharp-Sensoren schaffen nur etwa 25 Messungen pro Sekunde, die verwendeten Servos 180° in 1,5s) und gut umsetzbar. Die dritte Lösung wäre ideal, ein Kamera-System (welches wir mit einer in Kombination mit Mikroprozessoren leicht verwendbaren CMUcam3 realisiert hätten) erschien aufgrund von wechselnden Umgebungseinflüssen riskant, ich werde trotzdem noch in diesem Jahr anfangen, mit einer solchen Lösung zu experimentieren, da ich die CMUcam auch für andere Projekte als interessant erachte.

Wir haben uns für die zweite Lösung entschieden. Wir verwenden zwei drehbare Sharp-Sensoren, die zusammen einen Bereich von etwa 270° abdecken.
Der Aufbau vom Roboter sieht wie folgt aus:
Der Roboter ist rund, das Fahrwerk besteht aus drei "OmniWheels", d. h. Allseitenrädern, mit denen man in beliebige Richtungen starten und während der Fahrt drehen kann.
Auf einer Seite sind die beiden drehbaren Sensoren angebracht, die Software versucht, beim Ball-Suchen diese in Fahrtrichtung zu halten.
Auf der gegenüberliegenden Seite ist die Grundplatte mit einem kreisförmigen "Ausschnitt" versehen, in den ein Ball hineinrollen kann. Eine Hebemechanik kann bis zu drei Bälle dann auf einer zweiten Ebene des Roboters verstauen. Um den Ausschnitt sind drei Entfernungssensoren so angeordnet, dass man Bälle damit exakt "einparken" kann, um sie anschließend zu sammeln.
Insgesamt verwenden wir fünf Sharp-Entfernungssensoren (drei verschiedene Typen), vier Servos, drei Gleichstrommotoren, drei Radencoder, einen Spannungsteiler zur Messung der Akkuspannung und sechs an der Unterseite angebrachte Reflexlichtschranken, um Bodenmarkierungen zu erkennen.

Die Elektronik besteht aus drei ATmega168-Mikroprozessoren, die per SPI untereinander kommunizieren können. Sie kann bis zu neun Entfernungssensoren lesen, neun Lichtschranken lesen, sieben Servos ansteuern, 6 Gleichstrommotoren steuern und stellt zahlreiche digitale Ein- und Ausgänge bereit. Weiterhin stellt sie zwei serielle Schnittstellen und ein paralleles Programmierinterface zur Verfügung. Eine der beiden Schnittstellen ist für den Anschluss an einen PDA oder einen PC gedacht, die andere für ein kleines TFT oder die oben erwähnte Kamera.

Aufgrund der benötigten IO-Ports und der gewünschten Kommunikationsmöglichkeiten hatten wir die Wahl zwischen massivem Multiplexing, einem riesigem Prozessor oder mehreren kleinen Prozessoren. Die Verarbeitung der Sensordaten und die Ansteuerung des Allseitenradantriebs erfordert für einen Prozessor eine vergleichsweise hohe Rechenleistung und eine kurze Reaktionszeit, weshalb Multiplexing unattraktiv wird. Aufgrund früherer Erfahrungen mit Atmel-Prozessoren und dem Umstand, dass selbst bei einem größeren ARM-Prozessor noch Multiplexing nötig gewesen wäre, um genügend ADC-Ports zu haben, fanden wir eine Mehrprozessorlösung naheliegend. Dazu kam, dass ich persönlich die Koordination und Interaktion eines Mehrprozessorsystems für interessant hielt, und als derjenige, der die Software-Infrastruktur geschrieben hat, auch viele grundlegende Programmieraufgaben erledigen könnte, mit denen man bei typischer Desktop-Entwicklung sonst nichts zu tun hat. Durch mehrere Prozessoren entfällt gleichzeitig der Overhead für Multitasking, da wir sozusagen auf jedem Prozessor einen "Thread" haben, und echte Parallelität erreichen. Jeder Prozessor hat eine Taktrate von 18,432 Mhz, d. h. insg. sind wir damit sogar schneller

(echt-parallele Ausführung lässt sich in der Anwendung gut ausnutzen) als kleinere ARM-Prozessoren, die Daten nur seriell verarbeiten können.

Die Software besteht im Moment aus über 5.000 Zeilen C-Programmcode, über 100 Funktionen, verteilt über zehn Module und vier Hauptprogramme (drei Prozessoren, ein Bootloader, siehe unten). Wir verwenden zum Kompilieren GCC (Version 4.2). Alle drei Prozessoren verwenden die gleichen Module, über Präprozessormakros wird lediglich unterschieden, ob ein Befehl ausgeführt oder an einen anderen Prozessor zur Ausführung weitergeleitet werden soll. Die Software kann über die parallele Schnittstelle übertragen werden.
Ein kleiner C-Bootloader ermöglicht auch die Programmierung über die serielle Schnittstelle. Ich hatte daran Interesse, weil ich an die serielle Schnittstelle der Elektronik einen "Sharp Zaurus" (Linux-WLAN-PDA) angeschlossen habe, mit dem ich drahtlos neue Software aufspielen oder Sensor-Werte beobachten kann. Der an den PDA angeschlossene Prozessor überträgt das Programm ggf. zu einem anderen Prozessor weiter, wenn nicht er selbst programmiert werden soll.

Das Hauptprogramm auf jedem Prozessor initialisiert die Module und trägt die von einem einfachen Scheduler (nicht-verdrängend) verwalteten Prozesse ein. Im Folgenden eines der drei Hauptprogramme:

```
#include "main.h"
#include "time.h"
#include "ports.h"
#include "com.h"
#include "ios.h"
#include "boot_support.h"
#include "task.h"
#include "navigation.h"
#include <avr/io.h>
#include <avr/interrupt.h>

int main(void) {
    time_init();     /* Ein Interrupt ueberwacht die Uhrzeit */
    ios_init();      /* IOs initialisieren */
    drive_init();    /* Fahrwerk initialisieren */
    com_init();      /* Kommunikation zwischen den Prozessoren und ueber die serielle
Schnittstelle initialisieren */
    task_init();     /* Prozessverwaltung initialisieren */
    sei();           /* Interrupts aktivieren */

    /* Prozesse starten */
    task_add(TASKID_COM);
    task_add(TASKID_STATUS);
    task_add(TASKID_WAITSTART);

    task_main_loop();/* Kontrolle an Scheduler uebergeben */
    return 0;
}
```

Die Kommunikation zwischen den Prozessoren erfolgt selten bewusst, d. h. direkt im Anwendungsprogramm. Kann ein Befehl auf dem aktuellen Prozessor nicht ausgeführt werden, leitet die Bibliothek diesen automatisch zum richtigen Prozessor weiter. Steuert man beispielsweise per *ioset(SERVO_SENSORL, SERVO_OFF);* einen Servo an, überträgt *ioset* die Information an den Prozessor, an den der Servo zum Bewegen des linken Sensors angeschlossen ist. Ruft man *moveabs(100, 200, 15);* auf einem beliebigen Prozessor auf, wird eine Bewegung zur Position (100, 200) veranlasst, wobei als Zielwinkel 15° übergeben wird.

Die Funktion *ioset* beginnt mit folgenden Zeilen:

```
void ioset(uint8_t ionr, uint8_t status) {
    /* Ist der gewuenschte Port an diesen Prozessor angeschlossen? */
    if (RECEIVER(ionr) != ID_ME) {
        com_sendtwobytes(RECEIVER(ionr), MSG_IOSET, ionr, status);
        return;
    }

    [..] /* Tatsächliches Setzen des IOs */
```

Die komplette Definition der IOs, den Prozessoren an die sie angeschlossen sind, Sensortypen etc. erfolgt in umfangreichen Header-Dateien. Man kann an der Elektronik die Peripherie zwischen den Prozessoren beliebig umstecken, man muss nur in den Header-Dateien die richtigen Werte eintragen. Die Ansteuerung in der Software ändert sich nicht. Wird ein bestimmter Servo beispielsweise sehr häufig von einem Prozessor angesteuert, ist es sinnvoll, ihn direkt an diesen anzuschliessen, damit die SPI-Kommunikiation entlastet wird. Fällt einem das erst mitten bei der Entwicklung des Programms auf, sind kaum Änderungen an Hard- und Software notwendig. (Auf der Elektronik lässt sich die gesamte Peripherie über Jumper von den Prozessoren trennen und mit einem anderen Prozessor verkabeln.)

Den drei Prozessoren ist jeweils eine Aufgabe zugeordnet:

- Der Strategieprozessor verarbeitet die Bodensensoren und überwacht die Spiellogik.
- Der Fahrprozessor steuert den Allseitenradantrieb unter Beachtung der Messwerte der Encoder.
- Der Navigationsprozessor verarbeitet die Sensordaten und steuert die Servos.

Zur Kommunikation untereinander wird ein selbstentwickeltes, einfaches Protokoll verwendet, das Kommunikationsmodul kümmert sich automatisch um die Übertragung von Zeichen, Zahlen oder Strings.

Die bisherigen Erfahrungen mit der Platine zeigen, dass das Mehrprozessorsystem die Software erheblich komplexer macht. Der Austausch von Sensorergebnissen, erreichten Zielpositionen, Änderungen von Modi, das Synchronisieren von wichtigen Informationen, etc. ist nicht leicht. Ein einzelner, leistungsfähiger, großer Prozessor wäre erheblich einfacher zu programmieren. Im Rahmen von Modularisierung und Generalisierung lässt sich hier sicherlich noch einiges vereinfachen, z. B. möchte ich das Synchronisieren von Variablen in Zukunft vereinfachen. Gleichzeitig vereinfacht ein Mehrprozessorsystem aber auch das effiziente Parallelisieren von Fahrwerksansteuerung, Sensorauswertung, Servobewegungen, etc.
Das aktuelle Sensorikprinzip ist meiner Meinung nach, abgesehen von dem erwähnten Kamerasystem, das bestmögliche, die bisherigen Erfahrungen sind vielversprechend. Die Sharp-Sensoren sind dennoch weit von der Perfektion entfernt, die Meßbereiche, für die sie geeignet sind führen zu vielen Problemen. Die drehbaren Sensoren sind beispielsweise nicht geeignet, um Bälle, die weniger als 30cm entfernt sind zu erkennen. Das verwendete Fahrwerk ist sehr interessant, nett anzusehen und schnell, gleichzeitig macht es gerade die Positionsüberwachung sehr viel komplexer. Die von Matthias Bär entwickelte Mechanik des Roboters funktioniert einwandfrei und hat zu keinen Problemen geführt, ich finde, dass sie sehr sauber aufgebaut ist. Die selbstentwickelte Elektronik enthält einige, kleinere Layoutfehler, gefällt mir insgesamt jedoch gut. Das Mehrprozessorkonzept hat meine beiden Teampartner kaum betroffen, da die entwickelte Infrastruktur die Abstraktion sicherstellen sollte.

Die Verbindung mit einem PDA, einem TFT und einer Kamera oder anderen Geräten mit serieller Schnittstelle würde an sich natürlich eine Vielzahl von Anwendungsmöglichkeiten eröffnen. Die Platine ist so allgemein gehalten, dass sie für praktisch jeden Roboter einsetzbar ist (die Platine enthält auch ein Lötfeld für weitere Ergänzungen). Die Software enthält insbesondere mit dem Kommunikations- und Input-Output-Modul eine grundlegende Infrastruktur, die man in Kombination mit der Platine zwingend benötigt. Das Fahrwerks-Modul ist für jeden Allseitenradantrieb mit drei Rädern geeignet. Durch die vielen Kommunikationsmöglichkeiten (SPI, RS232, I²C) und die vielen Ports kann man praktisch jede zusätzliche Hardware anschließen. Die echte Parallelität ist für manche Anwendungen gut geeignet, bei anderen Anwendungen wäre der

Kommunikations-Overhead so groß, dass ein einzelner, leistungsfähigerer Prozessor bevorzugt werden sollte. Was wie gesagt für mich die interessanteste Ergänzungsmöglichkeit ist, ist die Kombination mit einer Kamera und Bildverarbeitung. Die CMUcam3 kommt mit einem ARM-Prozessor daher, auf dem man Bilder verarbeiten kann, die Ergebnisse könnte man per RS232 an die drei anderen Prozessoren übermitteln.
Soll mit der bestehenden Mechanik und Elektronik gearbeitet werden, schränkt das momentane Sensorkonzept etwas ein: Es ist nur für die Entfernungs- und Breitenbestimmung von umliegenden Objekten geeignet. Eine Kamera wäre hier sicherlich flexibler. Bei vielen verschiedenen Aufgaben dürfte die Elektronik und das Fahrwerk universell einsetzbar sein. Für die nächstjährige Aufgabenstellung von Roboking wäre die einzige nötige Änderung am Roboter das Ersetzen der Ballsammelmechanik -- ansonsten müsste man hauptsächlich das Programm des Strategie-Prozessors anpassen.