Der MSP430 - Roboter

Jens Altenburg

Jetzt wird es komplex. Aus einer Einzelapplikationen wird nun ein Ganzes. Schrittmotorsteuerung, Funk-Telemetrie und Multitasking bilden die Grundlage eines leistungsfähigen Systems. Ergänzt wird dies durch eine robuste erweiterungsfähige Mechanikplattform. Die rechte Würze steuert ein neuartiger Bildsensor bei. Und mit Bildverabeitung auf dem Mikrocontroller kommt auch die Software nicht zu kurz.

Präzise, robust und leistungsstark

Bei der Suche nach einer möglichst attraktiven Referenzapplikation fiel die Qual der Wahl auf einen Roboter. Das sieht im ersten Augenschein nicht gerade nach einer Kreativitätsoffensive aus. Roboterprojekte gibt's doch weiß Gott genug. Fischertechnik, Real Robots, von einer Web-Recherche wollen wir gar nicht reden, die Konkurrenz ist hart.
Dennoch, Roboter als Experimental-Plattform erfreuen sich permanenter Beliebheit an den diversen Forschungseinrichtungen dieser Welt. Es lässt sich darüber spekulieren, warum dies so ist (wahrscheinlich spielen auch Wissenschaftler gern). Ob und auf welche Weise von diesen Experimenten verwertbare Ergebnisse erzielt werden, steht auf einem anderen Blatt. Jedoch, die Kombination aus Mechanik, Elektronik und Software besitzt ihren eigenen Charme, und man will es kaum glauben, die Beschränkung der Ressourcen, seien es Rechenleistung oder auch nur begrenzte elektromechanische Eigenschaften, schafft besondere Entwicklungsanreize. Doch dies allein reicht kaum, um den Aufwand einer Neuentwicklung zu begründen. Obendrein, dass soll nicht unerwähnt bleiben, der MSP - Roboter ist keinesfalls ein Billigprodukt.
Betrachten wir deswegen die Features des entstandenen Gerätes. Das Antriebsprinzip beruht auf dem bekannten und für solche Experimente weit verbreiteten *differential drive*. Hierbei bilden zwei Motoren die Abtriebsquelle. Ein Stützrad verhindert ein Kippen dieses Aufbaus. Die unabhängige Steuerung der Motoren sichert höchste Manövrierbarkeit auf kleinstem Raum. Anstelle der häufig verwendeten Gleichstrommotoren kommen Schrittantriebe zum Einsatz. Ein exakter Geradeauslauf, bzw. genaue Kurvenfahrten sind damit kein Problem mehr.
Für die Kommunikation mit dem Roboter bzw. von Robotern untereinander, kommt das bekannte Funkmodul zum Einsatz. Die Steuerung übernimmt der MSP430, dessen Multitasking System um einige Echtzeitfähigkeiten ergänzt wird.
Eine robuste Mechnik hält alle Teile zusammen. Es bleiben auch genügend "mechanische" Schnittstllen, sprich Befestigungsbohrungen, für individuelle Erweiterung übrig.

Die Leiterplatte trägt alle elektronischen Komponenten mit Ausnahme des Funkmoduls und der Prozessorplatine. Speziell die steckbare Prozessorplatine verfolgt zwei Ziele. Zum ersten können alle Leser, die bisher die beschriebenen Experimente aufgebaut haben, den einmal erworbenen Baustein wiederverwenden. Und zum zweiten können somit auch ganz andere Mikrocontroller auf eine Adapterplatine montiert und eingesetzt werden.
Somit scheinen fast alle "Zutaten" des Roboters versammelt. Einzig die Frage der Sensoren ist noch offen. Die Anzahl der erhältlichen Sensoren dürfte nahezu die gesamte Bandbreite messbarer physikalischer Größen abdecken. Der MSP430 ist mit seinen Peripheriemodulen, digitale Standardeingänge, Timer für exakte Zeitmessungen oder A/D-Wandler für hochgenaue Spannungemessungen, bestens gerüstet, alle möglichen Wünsche zu erfüllen. Doch dies ist nicht genug, der Roboter bekommt einen CMOS-Bildsensor.

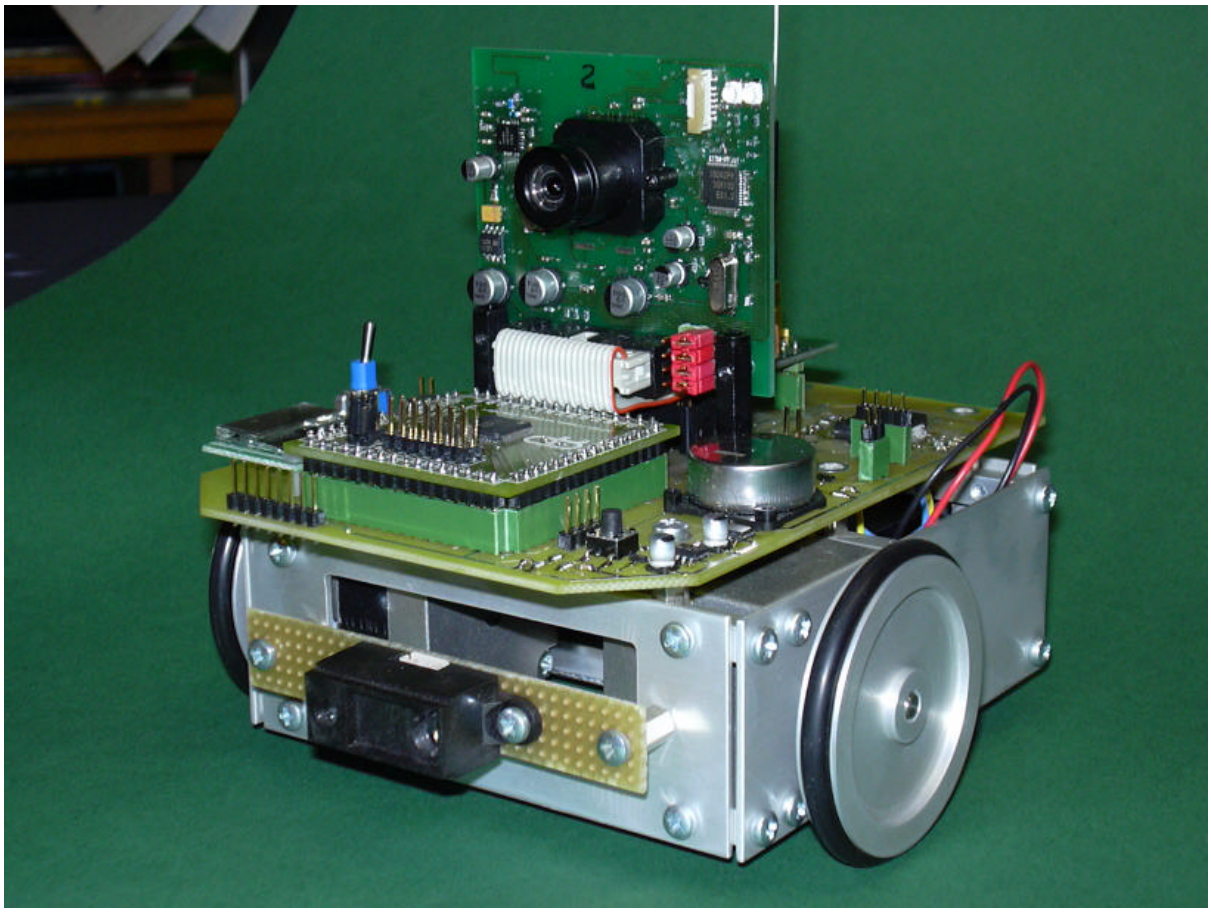


Bild 1 MSP430 - Roboter in "Vollausstattung" mit Schrittmotorantrieb, Funkmodul und Bildsensor. Vielfältige Optionen sind denkbar, z.B. ein Distanzsensor GP2D02 (vorn im Bild). Eine Solarzelle und ein Goldcap als Energiespeicher können den MSP430 und das Funkmodul betreiben.

Eine wesentliche Eigenschaft des MSP430s sind seine flexibel programmierbaren *low power* Eigenschaften. Um diese Features zu nutzen, erhält der Roboter eine Solarzelle zur Stromversorgung. Die Zelle reicht natürlich nicht, um eine völlig autarke Versorgung sicherzustellen, dazu müsste sie sehr viel größer werden. Sie wird so dimensioniert, dass die Maximalspannung 3,6 V nicht übersteigt. Zusätzlich ist ein Goldcap-Kondensator mit 1F Kapazität als Energiespeicher vorgesehen. Damit kann der Prozessor im *low power* Mode aktiv bleiben. Unter

günstigen Bestrahlungsverhältnissen reicht die gespeicherte Energie aus, um kurzzeitig das Funkmodul zu aktivieren.

Schaltungstechnik

Da das System auch für andere Mikrocontroller offen sein soll, werden einige Erläuterungen zu den Schaltungsteilen nötig.

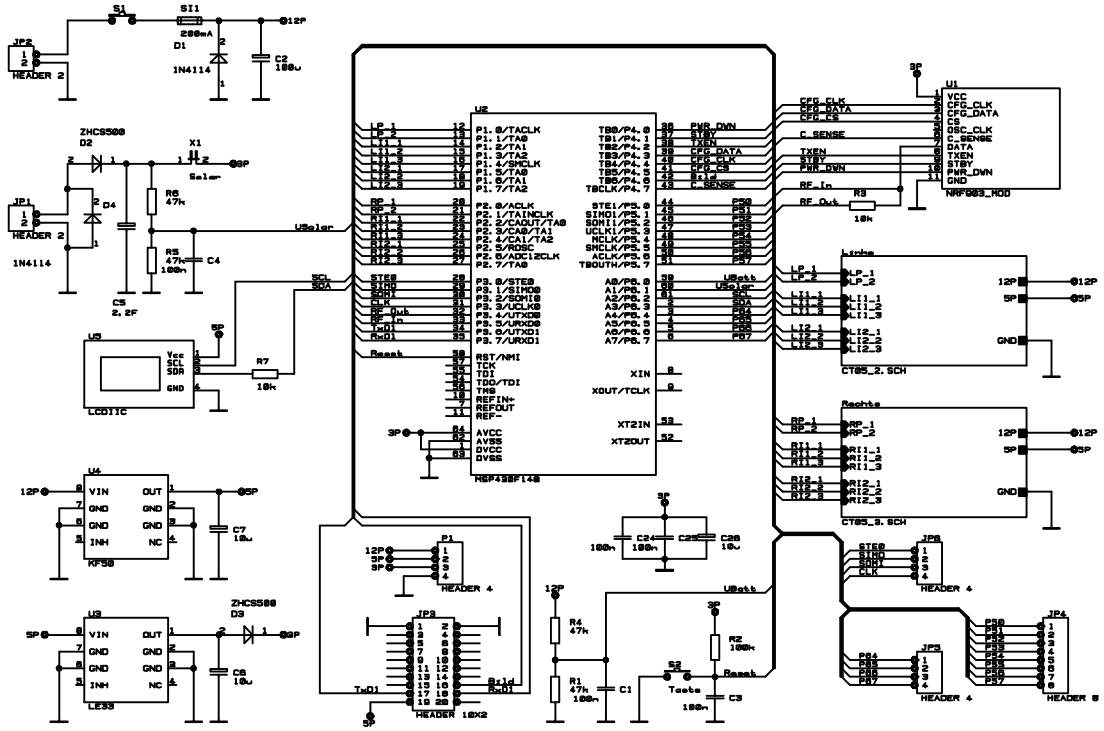


Bild 2 CPU-Baugruppe

Kern des Ganzen ist die CPU-Baugruppe mit dem MSP430. Dessen Aufgabe ist die Ansteuerung des Funkmoduls und der Schrittmotoren.

Als Schrittmotortreiber kommen die Typen L6258 der Fa. ST-Microelectronics zum Einsatz. Interessant ist auch das LCD-Display. Eigentlich nicht wirklich notwendig, ist es mehr eine Option. Der Datenaustausch zwischen Display und CPU erfolgt über den I²C Bus. Leider ist das Display nicht in einer 3 Volt Version erhältlich, es kann also nicht im extremen low-power-Mode angesteuert werden (Versorgung nur durch Solarzelle und Goldcap).

Die Stromversorgung erfolgt ganz konservativ mit Linearreglern. Der genaue Typ ist (fast) egal. Im Sinne sparsamen Stromverbrauchs kommen Mikropower-Typen zum Einsatz. Zur Stromzufuhr dienen die Anschlüsse JP1, JP2. An JP1 wird die Solarzelle angeklemmt. Die Z-Diode D4 verhindert Überspannungen. Die Hauptstromversorgung erfolgt über JP2. Es werden minimal 12 V benötigt. Der Roboter wird aus zwei 8,4 V Blöcken versorgt. Das ist ein Kompromiss zwischen Akku-Preis und Baugröße. Wer hier mehr Leistung wünscht, ersetzt den Batteriekasten durch einen Akku-Pack, z.B. aus 10 AAA-Zellen. Wer es richtig luxuriös

möchte, nimmt Li-Ionen-Akkus. Bitte jedoch die Minimalspannung von ca. 12 V für die Motortreiber berücksichtigen.
Die anderen Steckverbinder dienen zur Systemerweiterung. Der MSP430 besitzt mehr I/O-Leitungen, als in diesem Fall nötig sind.
Auf den zweiten Blick verwundert die fehlende Takterzeugung. Nirgendwo ist ein Quarz oder Keramikschwinger in der Schaltung. Doch das täuscht. Die Takterzeugung erfolgt ja auf dem Header-Board. Nimmt man einen anderen Prozessor, ist auch dessen Takt-Versorgung auf der Adapterplatine zu realisieren.

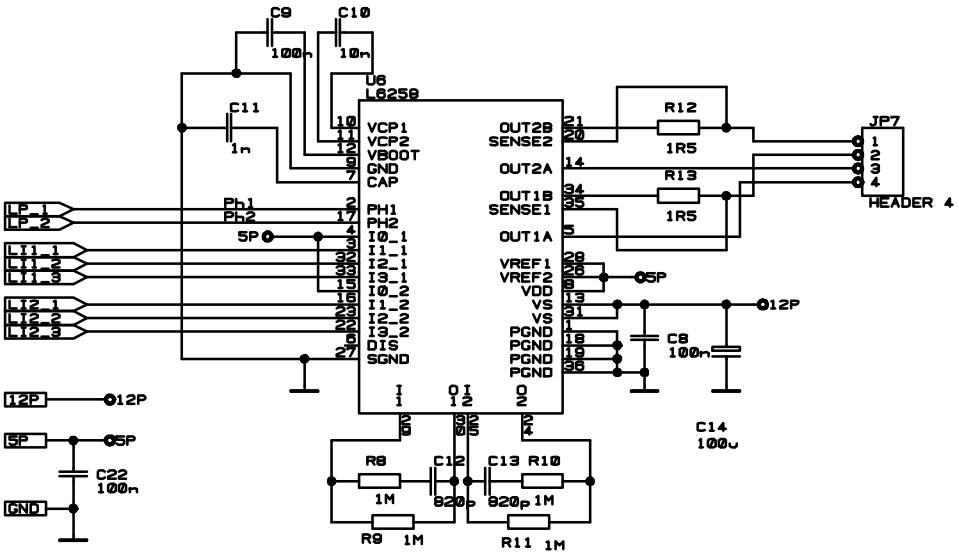


Bild 3 Ansteuerung des Schrittmotors mit dem L6258

Gänzlich modifiziert wurde auch die Schrittmotorsteuerung. Da die Schaltung zu umfangreich für einen Lochrasteraufbau ist, besteht keine zwingenden Notwendigkeit mehr, auf SMD-Bausteine zu verzichten. Hinzu kommt, dass autonome Roboter eigentlich immer mit zuwenig leistungsfähigen Stromversorgungen kämpfen müssen. Der Eigenverbrauch der in [2] verwendeten Ansteuer-ICs ist für mobile Zwecke zu groß.

Softwaredesign

Jetzt beginnt der eigentlich interessante Teil. Bis zum Punkt Hardwaredesign gibt es eine Unzahl von Aufbauvorschlägen, auch kommerzielle Systeme. Doch dannach wird es dünn. Wer soviel Zeit (und Geld) in ein ambitioniertes Projekt gesteckt hat, fühlt sich bei der Software oft allein gelassen. Ein wenig hängt dies wohl mit der Materie zusammen.
Komplexe *Embedded Systeme* brauchen zu ihrer Realisierung eigentlich perfekte "Allrounder". Der geplagte Ingenieur muss Hardwaredesigner, Softwareentwickler und versierter Mechaniker in einer Person sein. Solche Perfektionisten gibt es praktisch nicht.
Verschärfend wirkt sich auch die Vielfalt potentieller Sensoren bzw. denkbarer Einsatzfälle aus. Mit großer Wahrscheinlichkeit ist mit der Umsetzung komplexer Algorithmen zu rechnen. Klar sind auch Anforderungen, die harten Echtzeitkriterien genügen müssen.

Beides soll natürlich miteinander "verheiratet" werden, selbstverständlich auch übersichtlich und auf diverse Controller portierbar sein. Mit anderen Worten, das übliche Dilemma zwischen einfach, dafür speziell und mit geringem Ressourcenverbrauch sowie der allgemein und überall verwendbaren "Woll-Milch-Sau".
Das vorgestellte Multitasking-System für den MSP430 bildet für diese Zwecke ein gute Ausgangsbasis. Die harten Echtzeit-Forderungen betreffen hauptsächlich das Funkmodul und die Schrittantriebe.

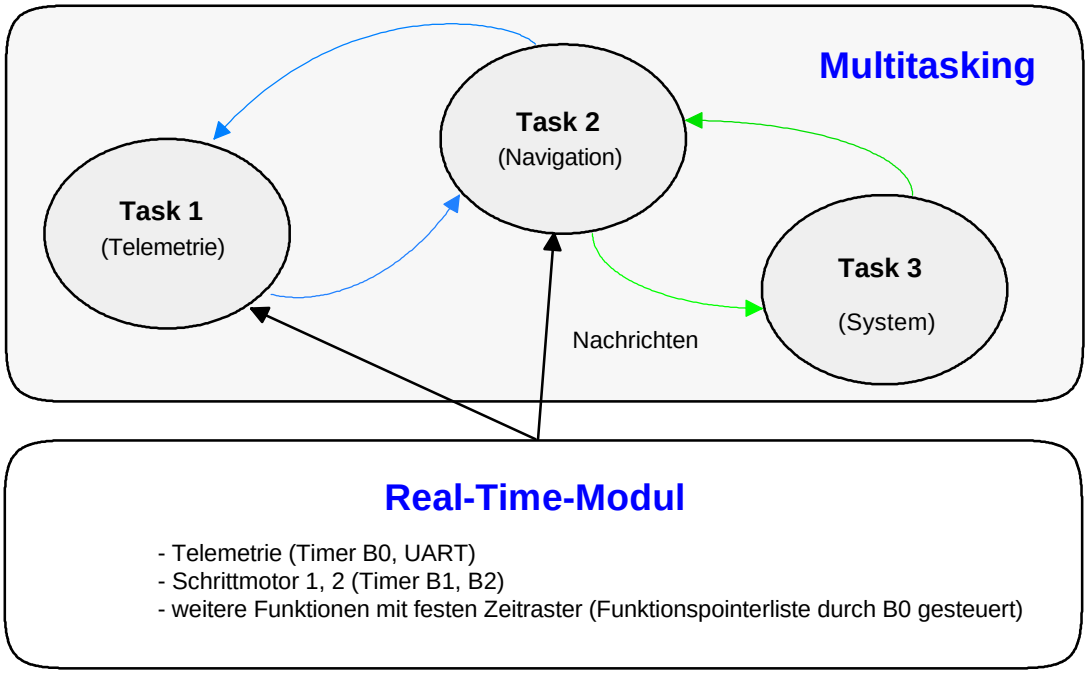


Bild 4 Zusammenhänge zwischen Echtzeit-Modul und Multitasking-System der Anwendertasks. Die Kommunikation zwischen den Modulen erfolgt über Benachrichtigungsfunktionen (Callback-Routinen)

Je nach konkreter Anwendung können weitere Funktionen mit festen Zeitraster hinzukommen. Wir starten mit der Konzeption des Echtzeitmoduls. Der MSP430 verfügt über sehr leistungsfähige Timer. Zwei Timer verfügen über vier Interruptvektoren. TimerA0 ist an den Scheduler vergeben. TimerA1 bleibt vorläufig unbenutzt. Das Real-Time-Modul stützt sich auf den Timer B. Abweichend vom Timer wird dieser im *continuos mode* betrieben. Warum?
Es werden mindestens drei verschiedene unabhängige Zeitrasterungen nötig. Ein fester Grundtakt von 2,5 ms wird für das Funkmodul benötigt. Die Schrittmotoren brauchen variable Schrittzeiten. Praktisch sinnvoll sind Werte zwischen 1...50 ms. Die kleinste Zeit (obere Schrittfrequenz) wird von den Eigenschaften des Motors begrenzt. Für den langsamsten Schritttakt ergibt sich eine willkürlich festgelegte untere Bewegungsgeschwindigkeit. Mit den vorgegebenen Raddurchmessern erreicht der Roboter damit Geschwindigkeiten zwischen 17 $mm*s^{-1}$ und 860 $mm*s^{-1}$. Beide Schritttakte müssen voneinader unabhängig steuerbar sein (Kurvenfahrten).
Im *continuos mode* bleibt das eigentliche Zählerregister des Timers unbeeinflusst. Mit Hilfe der *compare register* können daraus variable Zeiten generiert werden. Sobald der Inhalt des compare register mit dem Inhalt des Zählregister übereinstimmt, wird ein Interrupt erzeugt. In

der Interruptservice-Funktion (ISR) muss neben der eigentlichen Funktionalität der nächste Interruptzeitpunkt festgelegt werden. Im der ISR wird deshalb das *compare register* neu gesetzt. Das ist ganz einfach. Zum Inhalt des Registers beim Eintreten in den Interrupt wird die neue Zeitspanne als Vielfaches der Timertakte hinzugezählt.

```
/*************************************************************************
* Funktion  : Timer B0 arbeitet die Funktionspointerliste ab
* Autor     : Jens Altenburg
*************************************************************************/
#pragma interrupt_handler timerb_0_isr:TIMERB0_VECTOR
void timerb_0_isr(void){
    byte i;
    TBCCR0 += 20000;   /* Zeit bis zum nächsten Interrupt */
    for(i = 0; i < (sizeof(TimerB0) / sizeof(stRealTimeFunc)); i++){
        if(abTimerTick[i] == 0){
            abTimerTick[i] = TimerB0[i].bTimeStamp;
            TimerB0[i].fFunc();
            }
        else {
            abTimerTick[i]--;
            }
        }
    }
```

Im Beispiel wird zum Register TBCCR0 der Wert 20000 (2,5 ms) addiert. Ein Überlauf der Rechnung hat keine nachteiligen Folgen, da der Zähler ebenfalls "überläuft". Das klingt zuerst verwirrend, man kann sich das Prinzip aber am leichtesten mit einer Uhrzeitberechnung verdeutlichen. Es sei z.B. 23.00 Uhr und acht Stunden später soll der Wecker klingeln. In diesem Fall ergibt 23 + 8 nicht 31 sondern 7, das weiss jeder.
Die Hauptfunktion der ISR von Timer B0 ist die Bearbeitung der Funktionspointerliste *stReal-TimeFunc*. Diese Liste enthält alle in Vielfachen von 2,5 ms aufrufbare Funktionen. Jede dieser Funktionen ist so kurz wie möglich zu halten! Die Schrittmotoren können hiermit nicht gesteuert werden, da sonst nur Schritttakte in diesen Vielfachen möglich wären.
Der Timer B kann noch einen zweiten Interrupt auslösen. Als Trigger sind z.B. die *compare register* CCR1 bis CCR6 programmierbar. Da alle Triggerquellen auf den gleichen Interrupt wirken muss im Interrupt abgefragt werden, wer die ISR auslöste.

```
/*************************************************************************
* Funktion  : Schrittmotormodule
* Autor     : Jens Altenburg
*************************************************************************/
#pragma interrupt_handler timerb_1_isr:TIMERB1_VECTOR
void timerb_1_isr(void){
    switch(TBIV){
        case enTB1:
            TBCCR1 += SMD_wMotor1();
            break;
        case enTB2:
            TBCCR2 += SMD_wMotor2();
            break;
        case enTB3:
        case enTB4:
        case enTB5:
        case enTB6:
            break;
        }
    }
```

Dazu wird das Register *TBIV* ausgewertet. Die Reihenfolge der Bearbeitung richtet sich nach der Priorität des Auslösers. Haben mehrere *compare register* gleichzeitig ausgelöst, wird die ISR so oft aufgerufen, bis alle Ereignisse bearbeitet worden sind. Dies führt u.U. zu Zeitjitter bei der Bearbeitung. Deshalb gilt es auch hier die ISRs so schnell wie möglich zu machen.

Anwender-Tasks

Mit der Definition des Real-Time-Moduls sind die roboterspezifischen Funktionen festgelegt. Jetzt kommt der Anwender zum Zuge. Was der Roboter machen soll und wie dies programmiert wird, bleibt nun jedem selbst überlassen. Aha, und wieder bleibt der User genau dann im Regen stehen, wenn's interessant wird!
Stimmt nicht ganz. Ein einfaches Beispiel dient als "Regenschirm". Es werden drei Tasks definiert, Telemetrie, Navigation und System. Hinter der Telemetrie versteckt sich das Funkmodul, die Navi ist für die Schrittmotoren zuständig und der Systemtask überwacht die Batterien.
Die Anwender-Tasks können als *endless-loop* programmiert werden. Die Aufteilung der Algorithmen auf mehr oder minder viele kurze "Schnippel" entfällt. Auch eine feste Zeiteinteilung ist nicht mehr nötig. Aber Vorsicht, die zeitliche Abfolge der Inter-Task-Kommunikation wird nun immens wichtig. Konnte man beim kooperativen Multitasking noch davon ausgehen, dass die Abarbeitung bestimmter Funktionen immer sichergestellt wurde (vorher erfolgte keine Freigabe), ist der Umschaltzeitpunkt nun unbekannt.
Wir beginnen mit dem Design des Navigationstasks. Kernstück ist eine Liste von Jobs. Ein Job definiert sich durch Bewegungsmuster und Schrittzahl. Zu Anfang benötigen wir fünf Muster, Vorwärts, Rückwärts, Rechts, Links und Stopp. Später kann die Anzahl der Muster größer werden, z.B. Kurvenfahrten mit variablen Radien, etc.
Die Navi-Task überprüft den Inhalt der Liste. Entsprechend dem Inhalt der Liste werden daraus die Kommandos für die Schrittmotoren abgeleitet.

```
/************************************************************************
*    Typdefinition für ein Element der Job-Liste
************************************************************************/
typedef struct NAVI_stJob{
    NAVI_enType enType;     /* Job-Typ */
    dword       dwSteps;    /* Schrittzahl */
    }NAVI_stJob;

/************************************************************************
*    Function:  NAVI-Task
*    Author:    Jens Altenburg
************************************************************************/
void NAVI_vMain( void ){
    NAVI_stJob stJob; /* lokale Kopie des auszuführenden Jobs */
    byte bJobState;   /* Jobzustand */
    while(1){
        if(bIndexToJob != bIndexToNextJob){
            bJobState = nLeftBit + nRightBit;   /* Job aktiv */
            bIndexToJob++;                      /* diesen Job starten */
            stJob = stJobList[bIndexToJob & nJobListSize];
            bJobState = nLeftBit + nRightBit;   /* Job aktiv */
            switch(stJob.enType){
                case enRobS:                    /* Stop */
```

```
                SMD_vStartL(vCallbackLeft, nFullStep, stJob.wSpeed, 0);
                SMD_vStartR(vCallbackRight, nFullStep, stJob.wSpeed, 0);
                break;
            case enRobF:                    /* Robot Forward */
                SMD_vStartL(..., stJob.iSteps);
                SMD_vStartR(..., stJob.iSteps * -1);
                break;
            case enRobB:                    /* Robot Bachward */
                SMD_vStartL(..., stJob.iSteps * -1);
                SMD_vStartR(..., stJob.iSteps);
                break;
            case enRobL:                    /* Robot Left */
                SMD_vStartL(..., stJob.iSteps * -1);
                SMD_vStartR(..., stJob.iSteps * -1);
                break;
            case enRobR:                    /* Robot Right */
                SMD_vStartL(..., stJob.iSteps);
                SMD_vStartR(..., stJob.iSteps);
                break;
            }
        while(bJobState);                   /* ich haben fertig */
        }
    }
}
```

Der Vergleich der beiden Indices liefert eine Aussage ob neue Jobs in der Liste stehen. Ist dies der Fall, wird der auszuführende Job ausgewählt und die Parameter an die Startfunktion der Schrittmotoren übergeben. Mit einer entsprechenden Benachrichtigungsfunktion können neue Jobs in die Liste eingetragen werden.
Im Beispiel können neue Jobs nur von der Telemetrie kommen. Die Telemetrie wird ebenfalls als separater Task implementiert. Kern der Telemetrie ist das Funkmodul aus [4]. Da nun nicht jeder die Module bereits erworben hat, wird zu einer Hilfe gegriffen. Zwar wird das Funkmodul komplett in den Telemetrietask intergriert, ein Compilerschalter sorgt jedoch dafür, dass auch ohne Funk im Telemetrietask ankommende Botschaften "simuliert" werden. Damit wird die Intertask-Komunikation vollständig. Wer später wirklich funken will, löscht den Compilerschalter und kann sofort loslegen.

Sensoren für Roboter

Durchaus stiefmütterlich ist bisher die Sensorik des Roboter behandelt worden. Das hängt hauptsächlich daran, dass gerade dieses Gebiet nahezu unüberschaubar ist. Für praktisch alle physikalischen Größen, seien Temperatur, Druck, Licht oder was auch sonst, gibt es geeignete Wandler. Für die gewandelten Messgrößen, oft als Spannungswert verfügbar, bietet der MSP430 alle Voraussetzungen, daraus sinnvolle Entscheidungsstrategien abzuleiten.
Es fällt also schwer hier allgemeingültige Vorschläge zu unterbreiten. Im Bild 1 ist als "Standardsensor" lediglich ein Infrarot-Distanzsensor GP2D02 vorgesehen. Für zusätzliche Ergänzungen sind eine Anzahl von I/O-Ports auf diverse Stiftleisten herausgeführt. Die Kreativität des Einzelnen ist also ausschlaggebend.
Die aufwendige Mechanik und der leistungsfähige Steuercontroller "schreien" jedoch geradzu nach einen adäquaten Sensor. Die Fa. Parallax (Vertrieb Deutschland Elektronikladen) hat für derartige Anwendungen einen neuartigen Bildsensor herausgebracht. Kern der Baugruppe ist

ein CMOS-Kamerasensor. Damit lassen sich Farbbilder mit einer Auflösung von ca. 160x120 Pixeln (QQCIF-Format) machen. Der Clou des Ganzen, die Bilder werden über eine drahtlose Datenübertragung direkt an den PC geschickt.

Bild 5 Standbildübertragung mit dem Parallax-Bildsensor

Zum Zeitpunkt der Manuskripterstellung waren noch nicht alle technischen Parameter bekannt. In jedem Fall soll die Funkstrecke bidirektional arbeiten. Das bedeutet, dass sowohl Daten von der Kameraeinheit kommen, als auch zu ihr gesendet werden können. Zur Kommunikation verfügt die Kamera dazu über einen seriellen Port. Bis zu sechzehn Kameras können gleichzeitig betrieben werden.

Für die ersten Experimente wird der USB-Treiber installiert, das beigefügte PC-Programm gestartet und schon landen die Bilddaten des Sensors direkt auf dem Monitor.

Erfreulicherweise ist das Interface und das verwendete Datenprotokoll offen, hier kann sich also der PC-Programmierprofi austoben. Die USB-Schnittstelle verhält sich aus Sicht des PC wie ein schneller COM-Port. Man braucht also für eigene PC-Applikationen keinen seperaten USB-Treiber, o.ä..

Mit diesem Sensor ausgerüstet, mutiert der MSP - Roboter zum selbstgebauten Erkundungssystem, à la "Opportunity light". Durch den Bildsensor steht nun ein zweiter unabhängiger Telemtrie-Kanal zur Verfügung. Im Prinzip könnte auf das 433MHz Modul verzichtet werden, der Datenkanal im 2,4 GHz Band steht ja bereit. In der Praxis stolpert man schnell über die Probleme der Funkausbreitung im 2,4 GHz Band. Solange sich Sender und Empfänger im direkten Sichtkontakt befinden, ist alles gut. Sobald jedoch der Roboter mit seiner wissenschaftlichen Mission beginnt, z.B. Studium der Lebensbedingungen von Wollmäusen unter Studentenbetten, kann die Übertragungsleistung innerhalb weniger Zentimeter drastisch einbrechen. Und das Fatale daran, da die Datenübertragung rein digital ist, kündigen sich solche Zusammenbrüche vorher kaum an. Gut, wenn dann noch ein zweiter Kanal existiert, mit dem ein Notprogramm zur Rettung an den Roboter gesandt wird.

Bildverarbeitung mit Mikrocontrollern

Die Verfügbarkeit von CMOS-Bildsensoren und die Rechenleistung schneller Mikrocontroller lassen auch Experimente zur Bildverarbeitung mit Mikrocontrollern in erreichbare Nähe rücken. An der Fakultät für Informatik und Automatisierung der TU Ilmenau ist unter Leitung von Prof. Wernstedt am Institut für Automatisierungs- und Systemtechnik für diesen Einsatz das Robotersystem "MauSI" entstanden. Dem allgemeinen Abkürzungswahn folgend, steht MauSI für 'Mobiles autonomes System variabler Intelligenz".
Ziel dieser Entwicklung war u.a. auch der Entwurf geeigneter Objekterkennungsverfahren. Geeignet heißt, auf den Roboter bezogen, die Algorithmen müssen mit extrem limitierten Ressourcen auskommen. Bekanntermaßen sind jedoch Bildverarbeitung bzw. Mustererkennung rechenintensive Prozesse. Hier einen Mikrocontroller einzusetzen, scheint durchaus kühn. Unter bestimmten Vereinfachungen lässt sich jedoch zeigen, dass auch ein Mikrocontroller primitive Muster detektieren und daraus Werte bezüglich Abstand und Bewegungsrichtung gewinnen kann.
Ziel ist es, die Position eines Tischtennisballs aus einem realen Szenario zu ermitteln. Damit "reduziert" sich der notwendige Rechenaufwand auf die Suche nach dem größten zusammenhängenden Objekt in einer zweidimensionalen Punktematrix (Schwarz/Weißbild).

Bild 6 Reales Szenario zur Objektdetektion. Ein MauSI-Skelet (links) versucht das Ziel, den roten Tischtennisball zu lokalisieren.

Mathematisch gesehen läuft dies auf eine gewichtete Bildsegmentierung hinaus. Das Hilfsmittel zur Trennung der verschiedenen Bildbereiche ist der Schwellwertoperator ***S***. Dies ist letzlich das Maß, mit dem die Helligkeitswerte von Objekt und Hintergrund auseinandergehalten werden.
Die Berechnung des Schwellwertoperator zeigt die folgende Gleichung. Auf die Herleitung der Berechnungsvorschrift soll an dieser Stelle verzichtet werden.

$$S = \frac{\mu_O + \mu_H}{2} - \frac{\sigma^2}{\mu_O - \mu_H} \ln(\frac{P_O}{P_H})$$

$\mu_{O,H}$ – *Mittelwert* (O – *Objekt*, H – *Hintergrund*)

$P_{O,H}$ – *Auftrittswahrscheinlichkeit*

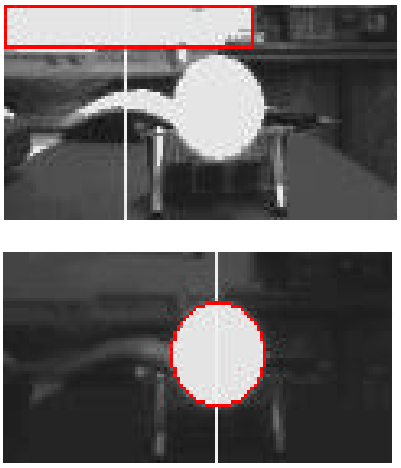
σ – *Streuung*

Diese Gleichung ist für den Controller noch nicht handhabbar. Mit einigen Tricks wird sie es aber. Der Tischtennisball wird als flächiger Lambertscher Strahler, d.h. ein Objekt mit gleichmäßiger Helligkeit, angenommen. Unter dieser Voraussetzung wird die Intensitätsstreuung $\sigma^2 << (\mu_O - \mu_H)$, somit entfällt der zweite Term der Gleichung.

```
/*************************************************************************
* Kurzbeschreibung     : Suchfunktion nach größtem Object
* Uebergabeparameter   : Farbparameter
* Return Value         : x-Koordinate
*************************************************************************/
VIDEO_stObject VIDEO_wSearchObject   ( void )
    {
    ... /* Intensitätsmittelwert, gleichzeitig suche max. Helligkeit */

    bLevel  = 0;                                          /* Spitzenwert µO */
    k  = 0;                                        /* Mittelwert über Bild µH */
    for(n = 0; n < 64; n+= 4){
        wXpos  = 0;
         for(m = 0; m < 176; m++){
            wXpos   += bMonoPix (m, n);     /* Werte aus Zeile aufsummieren */
                 if (bMonoPix (m, n) > bLevel ){       /* Spitzenwert finden */
                    bLevel   = bMonoPix (m, n);
                      }
                 }
    k  += ((byte)(wXpos  / nPixelNumber );                /* Bildmittelwert */
    /* Abbruch der Suchfunktion bei zu kontrastarmen Bildern */
    if((bLevel  - k) < 20) return  enError         /* zu schwacher Kontrast */
    /* Schwellwert aus Gleichung festlegen */
    bLevel  = k + ((bLevel  - k) >> 1);           /* Schwellwert festlegen */
```

Der Codeschnipsel deutet die reale Implementierung an. Der Wert in *bLevel* kennzeichnet den Schwellwertoperator. Mit diesem Operator wird das Bild erneut zeilenweise untersucht.
Gesucht wird nun nach dem Ball. Dieser wird als Kreisfläche auf die Pixelmatrix projiziert. Der Schwellwertoperator binarisiert die Zeile in dem Objekt zugehörige Pixel bzw. Hintergrundpixel. Jetzt wird die x-Koordinate des Mittelpunktes des größten zusammenhängenden Blockes berechnet. Dieser Wert wird zusammen mit der Anzahl der zugeordneten Pixel in einer Tabelle abgelegt. Die Tabellengröße ist begrenzt. Ein neuer Eintrag kommt nur dann hinzu, wenn er größer als ein bisheriger ist und diesen ersetzt. Nach der Untersuchung des gesamten Bildes beinhaltet diese Tabelle eine Liste der Größe nach gewichteter x-Koordinaten.
Bei einer idealen Suche nach einer einzigen vorhanden Kreisfläche wären alle x-Koordinaten identisch. Das Durchsuchen eines realen Bildes ergibt dagegen charakteristische Häufungen. Durch die Wichtung der Tabelleneinträge nach der Pixelmenge kennzeichnet der Mittelwert der *n* größten Einträge die gesuchte x-Koordinate des Objektes. Ein empirisch definierter Wert von n = 5 zeigte brauchbare Ergebnisse (Bild 7).



fehlerhafte Detektion infolge mangelhafter Separierung von Objekt und Hintergrund

exakte Detektion bei optimaler Trennung von Objekt und Hintergrund durch einen Filter

Bild 7 Lokalisierung des Tischtennisballs in einem Schwarz/Weißbild mit bzw. ohne Zusatzfilter.

Bei der Interpretation der Bilder ist zu berücksichtigen, dass der Algorithmus erkannte Objektpunkte weiß markiert (deswegen die extremen Helligkeitskontraste). Die vermeintliche x-Koordinate des detektierten Objektes zeigt der senkrechte Strich. Ohne einen Farbfilter schlägt die Objektdetektion fehl.

Zusammenfassung

Mit dem vorliegenden Artikel erreicht die Serie "Elektronische Zwerge" ihren Abschluss. In der Rückschau wird die enorme Leistungsbandbreite moderner Mikrocontroller sichtbar. Was mit einfachen LED-Spielereien begann, führt bis zur Bildverarbeitung. Speziell beim letzteren ist der MSP430 nicht unbedingt das Mittel der Wahl. Wobei auch hier mit der Ankündigung neuer Typen, zum Teil mit mehr als 10KB RAM, durchaus Experimente möglich sein sollten.
Will man Mikrocontroller für kompexe Applikationen, wie z.B. Bildverarbeitung, einsetzen, ist sehr oft ein Blick in die "Heimcomputer-Historie" angeraten. Zu Zeiten des C64 o.ä. wurde ja bereits heftig gespielt. Und Ballerspiele aller Art brauchen seit jeher einen schnellen Grafikaufbau und hantieren mit bewegten Elementen. Vergleicht man einen der neueren MSP430 mit historischen Heimcomputern, so sind da nur marginale Unterschiede (in der Rechenleistung, sowie RAM bzw. ROM Größe). Mit anderen Worten, hier können geschickte Programmierkonzepte en Masse abgekupfert werden.
Weitere noch stärkere Leistungssteigerungen lässt der Einsatz von Mikrocontrollern mit ARM-Core erwarten. Diese Bausteine, bislang zumeist für die Telekommunikation (Mobiltelefone) oder für PDA entwickelt und für den "Bastelfreak" schwer erhältlich, gibt es nun auch in Form eines Header-Boards. Der LPC2104 von Phillips wartet mit 128KB ROM, 64 KB RAM und sagenhaften 60 MHz Taktfrequenz auf. Obendrein ist der ARM ein echter 32-Bitter. Einziger Wermutstropfen, es gibt noch keine wirklich preiswerte Entwicklungsumgebung.
Der Gedanke, mit einem derartig leistungsfähigen Controller einen Roboter zu bauen, reizt immens. Die Rechengeschwindigkeit und die eingebaute Peripherie lassen hoffen, dass Steuer-

und Regelfunktion zusammen mit Echtzeit-Bildbearbeitung, eventuell sogar Bildkompression keine Träumerei sind.

Literaturverzeichnis

[1] Man, Richard; Willrich, Christina: "A Minimalist Multitasking Executive" Circuit Cellar Issue 101, 12/1998

[2] Altenburg, Jens; "Elektronische Zwerge - Einführung in die Mikrocontroller-Praxis" c't Magazin für Computertechnik, 24/2003

[3] Altenburg, J; Altenburg, U: "Mobile Roboter - Vom einfachen Experiment zur künstlichen Intelligenz, Carl Hanser Verlag 2002

[4] Altenburg, Jens; "Elektronische Zwerge - Schrittmotorsteuerung" c't Magazin für Computertechnik, 5/2004