Studienarbeit

Konzeption von I/O-Erweiterungen für Lego Mindstorms

Dietmar Cordes & Gunther Lemm

März 2003, HAW Hamburg

Inhaltsverzeichnis

Vorwort

Eines der interessantesten Themengebiete, der an der HAW-Hamburg angebotenen Wahlpflichtfächer, ist der Bau mobiler Roboter. Teilnehmer dieses WPP benutzen als Basis für ihre Konstruktionen üblicherweise das vom MIT entwickelte 6.270-Prozessorboard oder das von Lego stammende „Robotics Invention System“ (kurz RIS).

Die MIT- und Lego-Plattformen wurden für sehr unterschiedliche Zielgruppen entwickelt. Das Lego-RIS ist dabei zwar nahezu idiotensicher, aber leider auch entsprechend unflexibel. Für anspruchsvolle Aufgaben fehlen hier sowohl Anschlußmöglichkeiten als auch spezielle Sensorik.

Allerdings hat das Lego-System den enormen Vorteil, daß alle Komponenten mechanisch mit den bekannten Legosteinen zusammenpassen. Die Überlegung, die Lego-Plattform deshalb auch für komplexe Aufgabenstellungen nutzbar zu machen, führte zu verschiedenen Ansätzen für Eigenbauerweiterungen.

Im Rahmen dieser Studienarbeit soll erörtert werden, ob sich die Ideen für einen Sensor-Multiplexer (Gunther Lemm) und einen Beacon-Detektor (Dietmar Cordes) mit vertretbarem Aufwand umsetzen lassen und was dabei zu beachten ist.

1 Lego-Mindstorms

Mittlerweile findet man Computer in allen Bereichen unseres Lebens. So verwundert es nicht weiter, daß auch die Firma LEGO ihre Spielwaren seit einigen Jahren mit Mikroelektronik versieht. Allerdings bleiben die Dänen ihrem gewohnten Konzept treu und produzieren keineswegs fertiges Mikroprozessor-Spielzeug in Tamagochi-Manier. Stattdessen beschränkten sie sich darauf, eine programmierbare Steuereinheit zu entwickeln, die als universelle Basis für „intelligente“ Lego-Basteleien dient.

Dieser sogenannte RCX[1] ist ansich nichts anderes als ein etwas größer geratener Batteriekasten, worin zusätzlich ein Hitachi-H8-Prozessor untergebracht wurde. Drei Sensoreingänge und drei Motorausgänge bilden die Schnittstellen zur Lego-Außenwelt.

Über einen integrierten Infrarot-Transceiver kommuniziert der RCX mit dem sogenannten "Tower". Dies ist die IR-Gegenstelle, die mit einem Rechner verbunden wird.

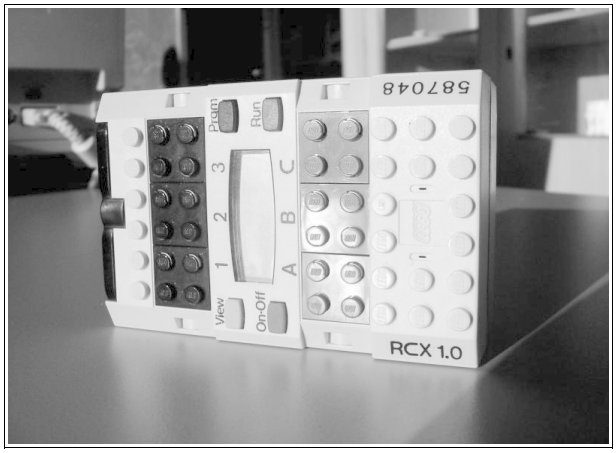

Abbildung 1.1: LEGO Mindstorm RCX

Mit der mitgelieferten Software ist es nun möglich, einfache Programme über eine grafische Oberfläche zusammenzustellen und per Infrarotübertragung in den RCX zu laden.

Alternativ kann über den Infrarot-Link auch direkt auf das Verhalten der Steuereinheit Einfluß genommen werden. Dazu besitzt der darin eingebaute Prozessor eine Art BIOS-ROM, mit dem er in der Lage ist, Bytecode-Befehle zu interpretieren und direkt umzusetzen.

Obwohl dieses System eigentlich für Kinder zwischen 10 und 16 Jahren konzipiert

1 Der RCX ist die Steuereinheit in dem von Lego unter dem Namen „Lego Mindstorms Robotics Invention System“ vertriebenen Bausatz für mobile Roboter. Siehe *http://mindstorms.lego.com*.

ist, wird es zunehmend von Schulen und Universitäten zum Bau von mobilen Robotern eingesetzt.

1.1 Anschlußproblematik

Wer mit dem oben genannten Bausatz ernsthaft versucht Roboter zu bauen, wird schnell feststellen, daß der RCX mit seinen drei Ein- und drei Ausgängen nicht gerade üppig bestückt ist. Auch in der Version 2.0 hat sich an dieser Tatsache nichts geändert.

Der Mangel an Ein- und Ausgängen ist kein neues Problem. Demnach ist es auch nicht weiter verwunderlich, daß sich schon diverse Leute Gedanken gemacht haben, wie man für Abhilfe sorgen kann. Durchstöbert man das Internet, finden sich unzählige Bauanleitungen für Eigenbausensorik oder Port-Erweiterungen.

Das Ziel der meisten Bastler ist dabei, spezielle Sensoren zu konstruieren, die nicht im Lego-Sortiment enthalten sind oder die Anzahl der Eingänge zu erhöhen, um mehr als drei Sensoren betreiben zu können.

Leider besitzen die Ein- und Ausgänge des RCX einige Eigenschaften, die den Bau solcher Erweiterungen nicht ganz einfach machen.

1.2 Ansätze für Sensor-Multiplexer

Ein naheliegender Ansatz, um mehrere Sensoren zu verschiedenen Zeitpunkten auf einen der Eingänge zu schalten, wäre eine Steuerung dieses sogenannten Multiplexers per Motorausgang. Damit würde dieser Ausgang aber nicht mehr für andere Zwecke zur Verfügung stehen. Da das ein hoher Preis für eine so simple Steuerungsaufgabe wäre, bedient man sich eines Tricks, der durch die Innenbeschaltung der Sensoreingänge ermöglicht wird:

Jeder Eingang des RCX ist intern mit einem der 10-Bit-A/D-Wandler-Eingänge des darin verwendeten Hitachi H8-Prozessors[2] verbunden. Zusätzlich liegt dieser

2 Der H8 ist ein Low-Power 16Bit-Microcontroller von Hitachi

Wandlereingang über einen 10k-PullUp-Widerstand an 5 Volt. Der zweite Anschluß der RCX-Eingänge ist mit Masse beschaltet. Über einen entsprechenden Widerstandswert zum Massepotential ergibt sich der jeweils zugehörige Sensorwert. Diese Beschaltung gilt für den passiven Modus.

Zusätzlich lassen sich die RCX-Eingänge aber auch in einen aktiven Modus umschalten, bei dem die Batteriespannung auf den Eingang gelegt wird, um aktive Sensoren mit Strom versorgen zu können. Zur Messung des Sensorwertes wird die Batteriespannung periodisch abgeschaltet.

Diesen Unterschied zwischen Batterie- und Meßspannung kann man sich als Schaltimpuls zunutze machen und damit beispielsweise einen Multiplexer ansteuern[3]. Üblicherweise wird dieser Impuls als Takt in einen Zählerbaustein gespeist. Um den gewünschten Sensor zum RCX durchschalten zu können, benötigt man demnach entsprechend viele aktiv/passiv-Wechsel zur Einstellung des Zählers.

1.3 Gasperi-Multiplexer

Eine Fundgrube für selbstgebaute Lego-Sensoren und Eingangserweiterungen ist die Webseite von Micheal Gasperi[4]. Dort findet sich eine Schaltung (siehe Abbildung 1.2), die sich den oben genannten Trick zunutze macht, um damit einen Eingangsmultiplexer für drei oder mehr Kanäle zu realisieren. Dabei verhält sich dieser Multiplexer wie ein aktiver Sensor, der seine Versorgungsspannung direkt aus dem Sensoreingang des RCX bezieht.

1.3.1 Schaltungsanalyse

3 Diesen Umschaltimpuls macht sich auch Wim Huiskamp zunutze, um ein I²C-Interface für den RCX zu bauen, das über zwei der Sensoreingänge betrieben wird. Leider ist die Datenübertragung ohne spezielle Anpassungen des RCX-Betriebssystems extrem langsam. Für die Verwendung in mobilen Robotern ist dieser Ansatz damit leider nutzlos. Siehe Elektor 4/2002, Seite 28.

4 Die Lego-Gemeinde verdankt Michael Gasperi diverse Sensorerweiterungen. Auf seiner Webseite *http://www.plazaearth.com/usr/gasperi/lego.htm* finden sich unzählige Links und Anleitungen zum Eigenbau von Lego-Sensoren und anderen Erweiterungen. Er ist außerdem einer der Autoren des Buchs „Extreme Mindstorms“, das sich mit der gesamten Thematik befaßt.

Die Schaltung[5] besteht im wesentlichen aus einem Zähler U1 und einer Handvoll bilateraler Schalter (U2), die als Multiplexer dienen. Der Umschaltimpuls wird dabei über ein RC-Glied (C2 + R2) ausgewertet. Es ist so dimensioniert, daß der Zähler

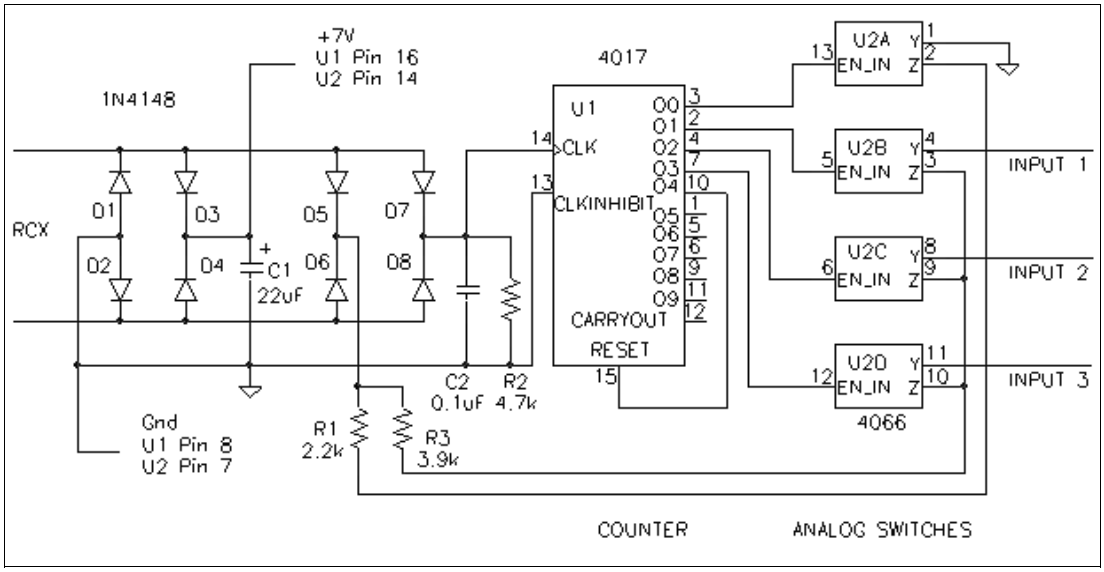

Abbildung 1.2: Schaltung des Multiplexer von Michael Gasperi

keine Clockimpulse durch das Abschalten der Batteriespannung während der Meßperioden erhält. Jede Umschaltung des Sensormodus erzeugt dagegen einen Flankenwechsel am Clock-Eingang des Zählers und sorgt damit für die Auswahl des nächsten Sensoreingangs am Multiplexer.

Um erkennen zu können, wann der Zähler wieder die Nullstellung erreicht hat, wird nach einem Zählerreset ein Sensorwert an den RCX geliefert, der von keinem der echten Sensoren erreicht werden kann. Dies geschieht, indem von U2A die Schaltungs-Masse über einen 2,2 kOhm-Widerstand (R1) an den RCX gelegt wird. Die übrigen Sensoren werden dagegen über einen 3,9 kOhm-Widerstand (R3) geschaltet und liegen somit in einem anderen Wertebereich. Neben der Verschiebung

5 Die Schaltung, auf die hier Bezug genommen wird, entstand offenbar aus verschiedenen Ideen von Sven Horstman, Paul Haas und Michael Gasperi. Zu finden ist sie auf *http://www.plazaearth.com/usr/gasperi/MUX.HTM*.

des Wertebereichs dienen diese Widerstände zur Strombegrenzung und verhindern, daß die Spannung am RCX-Eingang beim Schalten gegen Masse komplett zusammenbricht.
Die Stromversorgung geschieht verpolungssicher über einen Gleichrichter (D1-D4), über den dann ein 22µF-Kondensator geladen wird. Diese Kondensatorladung reicht aus, um das Abschalten der Battriespannung während der Messungen und Modusumschaltungen puffern zu können. Anderenfalls würde der Zählerstand verlorengehen.

1.3.2 Vor- und Nachteile

Die Schaltung hat den Vorteil, daß sie sehr leicht und kostengünsig aus Standardkomponenten aufgebaut werden kann. Leider hat sie aber auch einen großen Nachteil:
Die Umschaltung des Modus dauert jeweils ca. 10ms. Damit ergibt sich eine Umschaltzeit von etwa 20ms pro Eingangskanal und eine Gesamtumlaufzeit von über 80ms bei drei Kanälen (Zählerstand Null dient der Synchronisation). Hinzu kommt, daß sich die Kanäle nicht direkt adressieren lassen. Es ist also zwangsläufig nötig, sich durch den gesamten Zählerbereich zu takten, um einen bestimmten Sensor abfragen zu können.
Ein weiteres Manko ist, daß an diesem Multiplexer nur passive Sensoren betrieben werden können, da die Schaltung selbst schon zuviel Strom aus dem RCX-Eingang benötigt. Eine Versorgung weiterer aktiver Sensoren über den Sensorport ist somit nicht möglich.
Diese Lösung ist also nur dort empfehlenswert, wo wenige Kanäle in relativ großzügig bemessenen Zeiträumen ausgelesen werden sollen.

1.4 Sensor-Erweiterungen

Lego bietet für die Mindstorms einige Sensoren an, die für die meisten Standardanwendungen geeignet sind. Neben simplen Tastern finden sich hier auch etwas komplexere Sensoren, mit denen sich Temperaturen, Helligkeitsunterschiede oder Drehwinkel messen lassen. Für viele Anwendungsfälle ist diese Auswahl ausreichend, aber je nach Aufgabenstellung ergibt sich der Wunsch nach spezieller Sensorik.

Im Internet finden sich unzählige, meist sehr extravagante Basteleien, die dem RCX zu weiteren Augen und Ohren verhelfen. Hier reicht die Palette von Kompaß- und Sonar-Sensoren bis hin zu Druck- und Beschleunigungsmessern. Leider sind dies in den meisten Fällen Sonderlösungen, die nur auf sehr spezielle Problemlösungen zugeschnitten sind und sich nicht universell einsetzen lassen.

1.4.1 Sharp-Multiplexer

Eine typische Aufgabe beim Bau von mobilen Robotern ist das berührungslose Erkennen von Hindernissen. Die Realisierung eines solchen Sensors auf der Basis modulierter IR-Signale ist ohne Spezialbauteile kein einfaches Unterfangen und führt meist nicht zum gewünschten Ergebnis. Glücklicherweise bietet die Firma SHARP

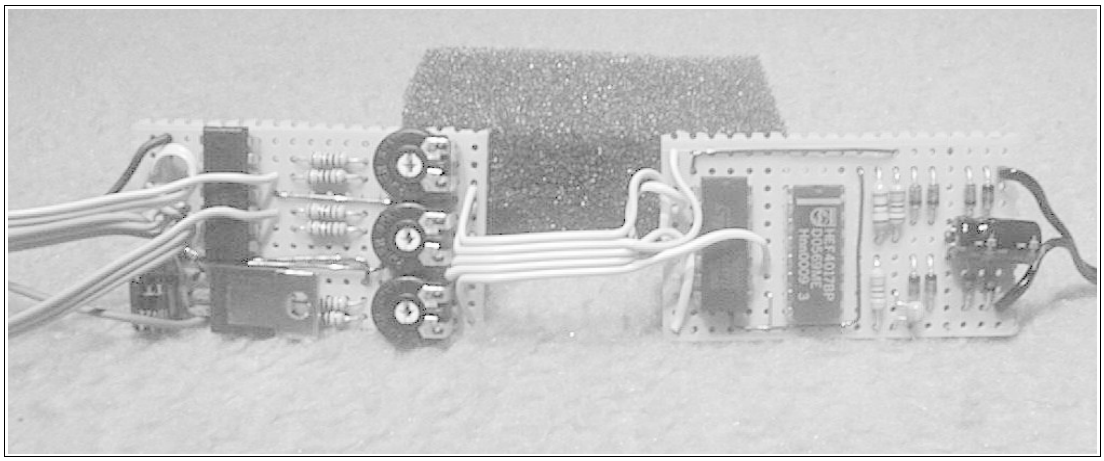

Abbildung 1.3: Schaltunsaufbau von Rainer Balzerowski

erschwingliche IR-Distanzmesser an, die den Abstand zu einem Objekt als Spannung zwischen null und drei Volt ausgeben.
Da normalerweise mehr als einer dieser Sensoren benötigt wird, hat Rainer Balzerowski die in Kapitel 1.3 vorgestellte Schaltung als Grundlage für einen Dreifach-Distanzsensor (siehe Abbildung 1.3) benutzt. Die Originalschaltung wurde dabei um einen Operationsverstärker zur Aufbereitung der Sharp-Signale und einen Spannungsregler ergänzt.

1.5 Anforderungen an Neuentwicklungen

Einzelne Sensoren für spezielle Aufgaben zu entwickeln hilft oft nicht weiter, da letztenendes die drei Sensoreingänge des RCX der Flaschenhals bleiben. Einziger Ausweg ist hier, einen Eingang an mehrere Sensoren zu verteilen. Leider sind die bisherigen Lösungen üblicherweise nicht universell einsetzbar.
Natürlich gelten für eine Neuentwicklung zusätzlich die üblichen Vorgaben wie geringe Abmaße, niedriger Stromverbrauch, geringe Herstellungskosten und die Verwendung von Standardbausteinen.

2 Beacon-Detektor

IR-Beacons sind Infrarot-Leuchtfeuer, die zur Orientierung bzw. Markierung von Punkten innerhalb eines Parcours verwendet werden. Anhand solcher Markierungen kann ein Roboter eine Ziel leicht auffinden und ansteuern.

Die hier verwendeten IR-Beacons (siehe Abbildung 2.1) wurden von Prof. Dr. Klemke entwickelt und sind bereits seit geraumer Zeit im RoboLab der der HAW-Hamburg im Einsatz. Allerdings wurden sie für die Verwendung mit Robotern konzipiert, die auf dem 6.270-Board[6] basieren.

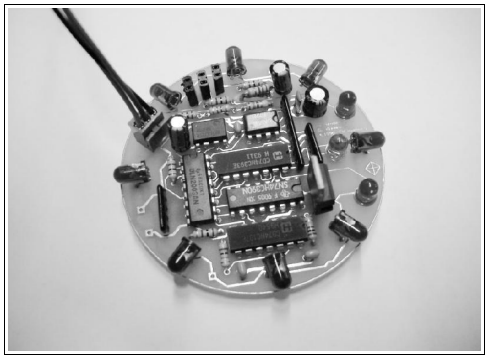

Abbildung 2.1: IR-Beacon

2.1 Die Eigenschaften von IR-Beacons

Die hier verwendeten IR-Beacons senden Pulse mit einer Frequenz von 100Hz bzw. 125Hz aus. Um das Signal unempfindlich gegenüber Umgebungslicht zu machen, wird es einer Trägerfrequenz von 40 kHz aufmoduliert.

Da das Prinzip dem einer IR-Fernbedienung entspricht, hat man auf der Empfängerseite eine große Auswahl von Bausteinen, die das Filtern der Trägerfrequenz übernehmen und die Nutzdaten als TTL-Signal liefern.[7]

Die bisher verwendeten 6.270-Boards können TTL-Signale direkt verarbeiten und

6 Das 6.270-Board ist eine Steuerplatine für mobile Roboter, die vom MIT entwickelt wurde und weltweit von vielen Universitäten als Schulungsplattform benutzt wird. Sie basiert auf einem 68HC11-Controller von Motorola und besitzt im Gegensatz zum RCX eine Vielzahl von Schnittstellen.

7 Sehr gebräuchlich sind in diesem Zusammenhang die IR-Empfänger des Typs SFH-506-xx von Siemens. Da Siemens offenbar die Fertigung dieser Bausteine eingestellt hat, bleiben als Vergleichstyp die baugleichen Empfänger TSOP17xx von Vishay (das xx steht jeweils für die zu filternde Trägerfrequenz). Für unseren Anwendungsfall empfehlen sich aber die Bausteine TSOP1840, da sie wesentlich kleiner sind als die Bauform des SFH-506. In größeren Stückzahlen kostet ein TSOP1840 weniger als ein Euro.

bieten außerdem die Möglichkeit einer interruptgesteuerten Auswertung ihrer Sensoreingänge. Somit läßt sich eine direkte Frequenzauswertung recht leicht realisieren.

Um die Frequenzen der IR-Beacons auch mit dem RCX auswerten zu können, fehlt genau diese Interruptfähigkeit[8]. Es wird also ein Sensor benötigt, der die Beacon-Frequenzen direkt in einer auf den RCX zugeschnittenen Form liefert.

2.2 Blickwinkel-Problematik im mobilen Einsatz

Ein Beacon sendet sein Signal in alle Richtungen einer Ebene aus. Es ist demnach ein 360°-Leuchtfeuer. Je nach Bauart haben die Empfänger einen Blickwinkel von etwa 120°. Wenn also ein Signal erkannt wird, weiß man nur, daß der Sender irgendwo in einem Bereich von 120° vor einem steht. Somit ist ein genaues Anpeilen eines Beacons nicht möglich. Umgehen kann man dieses Problem, indem man sich einen Sensor baut, der einen eingeschränkten Sichtbereich hat.

Abbildung 2.2: 3-fach IR-Empfänger

Die Abbildung 2.2 zeigt eine Konstruktion, die drei IR-Sensoren benutzt. Der mittlere Empfänger sieht nur in einem sehr engen Bereich und ist somit für die Feinjustierung gedacht. Die beiden äußeren Sensoren dienen mit ihrem größeren Blickwinkel der groben Anpeilung. Außerdem hat diese Konstruktion den Vorteil, daß sich erkennen läßt, in welcher Richtung der Beacon zu finden ist, wenn man sein Signal aus dem mittleren Sensor verloren hat.

8 Hier wäre es seitens der Lego-Entwickler geschickt gewesen, die Sensoreingänge nicht nur auf einen A/D-Wandler-Eingang, sondern parallel auch auf einen der unbenutzen interruptfähigen I/O-Pins zu legen.

2.3 Ansatz für einen Sensor

Die Sensoreingänge des RCX sind mit einem 10-Bit-A/D-Wandler verbunden, dessen Eingang über einen PullUp-Widerstand von 10 Kiloohm an 5 Volt liegt (siehe Kapitel 1.2). Die Spannung an diesem Eingang entspricht dem per Software ermittelten Sensorwert.

Bei kurzgeschlossenem Sensoreingang ergibt sich ein Meßwert von 0, da der Wandler-Eingang über den zweiten Anschlußpin des Sensoreingangs mit Masse verbunden wird. Im offenen Zustand liegen hingegen 5 Volt über den PullUp-Widerstand an, was zu einem Meßwert von 1023 führt.

Um mit einer möglichst einfachen Schaltung ans Ziel zu kommen, war die Überlegung, die vom Infrarotempfänger gelieferte Frequenz direkt in eine passende Spannung zu wandeln.

Praktischerweise gibt es Bausteine, die genau für diese Aufgabe entwickelt wurden. In unserem Fall kommt hierfür der LM331[9] zum Einsatz.

Da der RCX nur die besagten drei Eingänge besitzt und jede Abfrage Zeit kostet, wäre es von Vorteil, alle drei IR-Empfänger in einen Sensor zu integrieren.

Pro Empfänger können folgende Zustände auftreten:

1) kein Beacon zu sehen
2) 100Hz erkannt
3) 125Hz erkannt
4) mehr als 125 Hz erkannt (optional)

Um diese vier Zustände zu codieren, werden zwei Bit benötigt, womit sich insgesamt sechs Bit ergeben, die an den RCX übermittelt werden müssen.

9 Der LM 331 ist ein Baustein, der Frequenzen in Spannungen und umgekehrt wandeln kann. In dieser Anwendung wird ausschließlich eine Wandlung von Frequenz nach Spannung benötigt. Weitere Informationen finden sich im Datenblatt (siehe Literaturliste).

Aus elektrischer Sicht bedeutet das ein Aufsplitten des Sensor-Spannungsbereichs von 5 Volt in 64 Schritte à 78,125 mV. Hieraus ergibt sich wiederum die folgende Spannungs-Verteilung:

78,125mV	für Bit 0 (LSB)
156,25mV	für Bit 1
312,5mV	für Bit 2
625mV	für Bit 3
1,25V	für Bit 4
2,5V	für Bit 5 (MSB)

Bit 0 und 1 werden vom IR0 verwendet, Bit 2 und 3 vom IR1 und Bit 4 und 5 vom IR2. Durch eine Addition dieser Spannungen lassen sich die sechs Bit unabhängig voneinander transportieren. Über die Spannungsabstufung ergibt sich, welche Werte auf welchen IR-Empfänger entfallen. Allerdings ist die softwareseitige Auswertung relativ Aufwendig.

2.4 Testaufbau

Die Schaltung in Abbildung 2.3 entspricht einer Frequenzauswertung auf einem der drei Kanäle. Sie enthält noch keine Normierung auf unterschiedliche Wertebereiche, die bei gleichzeitiger Verwendung von drei IR-Empfängern nötig wäre.

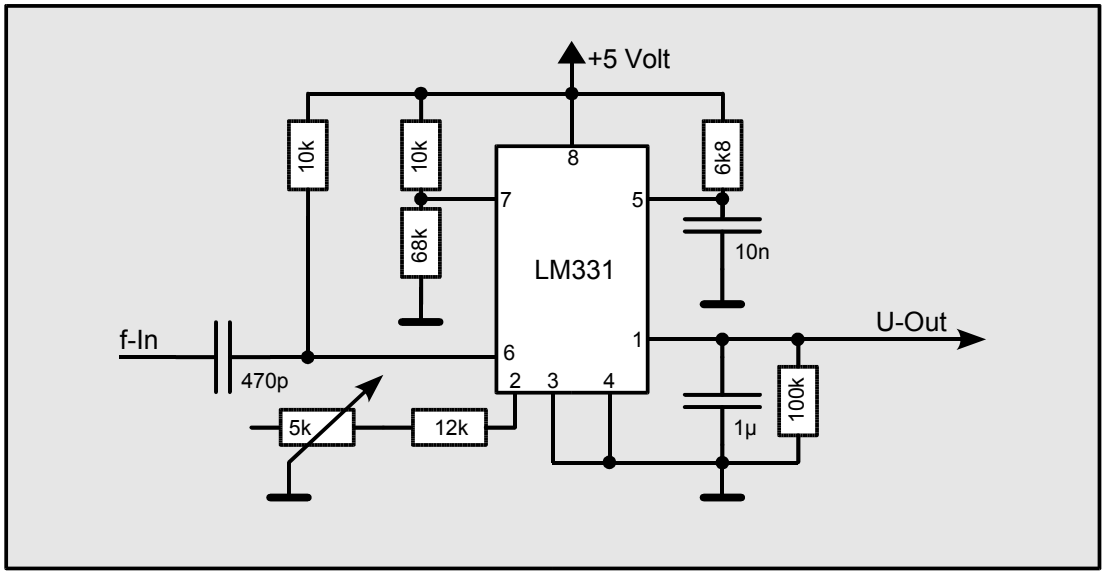

Abbildung 2.3: Schaltung des Frequenz-Spannungs Wandler

2.5 Schaltungsproblematik

Wie sich beim Aufbau der Schaltung herausstellte, wandelt der LM331 die Frequenz nicht direkt in eine Gleichspannung um, sondern erzeugt eine Pulsweitenmodulation. Diese wird durch den Kondensator 1µF-Kondensator am Schaltungsausgang in eine Gleichspannung geglättet.

Das Auf- beziehungsweise Enladen dieses Kondensators je nach anliegender Frequenz, verlangsamt den Erkennungsvorgang ganz erheblich. Somit werden kurze Impulse eines Signals, zum Beispiel beim Suchen eines Beacons mit einer Drehbewegung, eventuell gar nicht erkannt. Ein weiteres Problem stellt die Addition der einzelnen

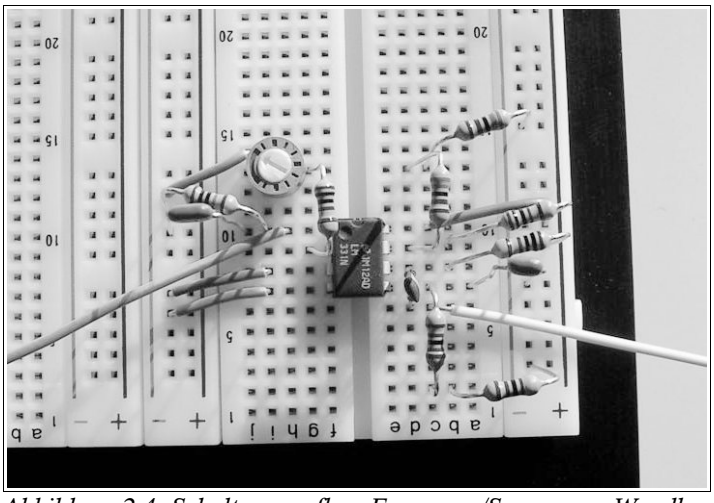

Abbildung 2.4: Schaltungsaufbau Frequenz-/Spannungs-Wandler

Spannungen dar. Jeder der drei Sensoren müßte präzise auf einen bestimmten Spannungsbereich skaliert werden. Der dazu zusätzlich nötige Aufwand macht den hier verfolgten Schaltungsansatz unbrauchbar.

2.6 Alternativen

Alternativ zu dem oben beschriebenen Verfahren, läßt sich eine Frequenz auch durch das Messen der Zeit zwischen den Impulsen ermitteln. Ein ähnlicher Ansatz wäre das Zählen von Impulsen innerhalb eines bestimmten Zeitraums. Allerdings erfordert dies einen erheblichen Schaltungsaufwand, sofern nicht wie auf den 6.270-Boards ein Prozessor zur Verfügung steht, der diese Auswertung per Interrupt bewältigen kann.[10]

10 Da der ursprüngliche Schaltungsansatz keine brauchbaren Resultate lieferte und der in Kapitel 3 beschriebene Multiplexer sowieso schon einen Prozessor beherbergt, ergab sich die Überlegung, die IR-Erkennung direkt in den Multiplexer-Entwurf zu integrieren. Siehe Kapitel 3.2.6.

3 Sensor-Multiplexer

Wie schon in Kapitel 1.5 angesprochen, ist es sinnvoll, einen universellen Multiplexer für Sensoreingänge zu entwickeln, der den RCX-seitigen Flaschenhals beseitigt.

3.1 Anforderungen

Der wesentliche Nachteil der bisherigen Multiplexer-Lösungen, die mit einer Ansteuerung per Modus-Umschaltung arbeiten, ist die inakzeptable Umschaltgeschwindigkeit bei mehr als drei Sensoren.

Aus diesem Grund sollte auf jeden Fall die Steuerung des Multiplexers über eine echte protokollbasierte Datenübertragung geschehen. Hierzu muß allerdings einer der Motorausgänge geopfert werden, um darüber Daten seriell ausgeben zu können. Seitens der Schaltung sollte also dafür gesorgt werden, daß dieser Motorport in irgendeiner Weise kompensiert wird.

Um eine robuste Übertragung zu gewährleisten, wird außerdem einer der Sensoreingänge für die Rückmeldung benötigt. Hierüber sollten sich außer dem Acknowledgement-Signal noch weitere Daten transportieren lassen.

Die Multiplexersektion muß für Standard-Lego-Sensoren geeignet sein und sollte sich so transparent wie möglich verhalten. Zudem soll jeder Eingang direkt adressierbar sein.

Wünschenswert wäre auch eine permanente Stromversorgung aktiver Sensoren. Im Normalfall erfordert die Verwendung mehrerer aktiver Sensoren nämlich ein entsprechend schnelles Umschalten.

Zusammenfassend ergibt sich die folgende Anforderungsliste:

- geringer Platzbedarf
- möglichst ohne eigene Stromversorgung
- für aktive und passive Standardsensoren geeignet
- schnelle Umschaltung unabhängig von der Anzahl der Eingänge
- (permanente) Stromversorgung für aktive Sensoren
- Kompensation für verbrauchte RCX-Eingänge

3.2 Ansatz

Zur Realisierung der Sensorumschaltung stehen zwei Optionen zur Verfügung: Man mißbraucht die aktiv/passiv-Umschaltung wie in Kapitel 1 beschrieben, um durch die Pegelunterschiede den gewünschten Sensor auszuwählen oder benutzt hierzu einen der Motorausgänge.

Da eine der Anforderungen ist, auch aktive Sensoren am Mulitplexer betreiben zu können, scheidet die erste Möglichkeit aus. Dort hätte man das Problem, daß die Steuerschaltung des Multiplexers eine für einen aktiven Sensor nötige Modusumschaltung als Zählimpuls fehlinterpretieren würde. Steuert man den Multiplexer dagegen über einen der Motorports an, hätte man keinerlei funktionelle Überschneidungen.

Die Forderung nach einer weitgehenden Kompensation der durch die Schaltung verbrauchten RCX-Ports bedeutet, daß die Steuerschaltung nicht allzu simpel ausfallen kann. Es muß demnach ermöglicht werden, verschiedenartige Hardware-Erweiterungen (Motorports, Multiplexer, Sensoren) ansprechen zu können. Sinnvoll wäre somit ein universelles serielles Übertragungsprotokoll, das unabhängig von der zu steuernden Hardware ist.

Aus dieser Vorüberlegung ergeben sich die im Folgenden beschriebenen Schaltungskomponenten.

3.2.1 Prozessor

Da der Motorausgang nur binäre Werte liefert und nur einer der Motorports benutzt werden soll, müssen Takt und Daten in einem Signal zusammengefasst werden.

Ein halbwegs robustes serielles Protokoll für diesen Anwendungsfall mit Standard-Logik in Hardware zu implementieren, ist ein relativ hoffnungsloses Unterfangen. Mit zunehmender Komplexität des Protokolls steigt die Anzahl der Bauteile und damit auch Platzbedarf und Stromaufnahme explosionsartig an.

Ein Ausweg aus diesem Dilemma sind programmierbare Logikbausteine. Leider verbrauchen diese viel zuviel Strom, als daß sie in unserem Fall sinnvoll eingesetzt werden könnten.

Somit bleibt als letzter Lösungsansatz die Verwendung eines Microcontrollers. Allerdings ist hier sehr darauf zu achten, daß man nicht mit Kanonen auf Spatzen schießt. Die meisten gebräuchlichen Microcontroller sind so leistungsstark, daß sie eine ähnliche Rechenleistung haben wie der im RCX verwendete H8-Controller. Es wäre demnach unsinnig, der Peripherie mehr Rechenleistung zu verschaffen als der RCX selbst besitzt.

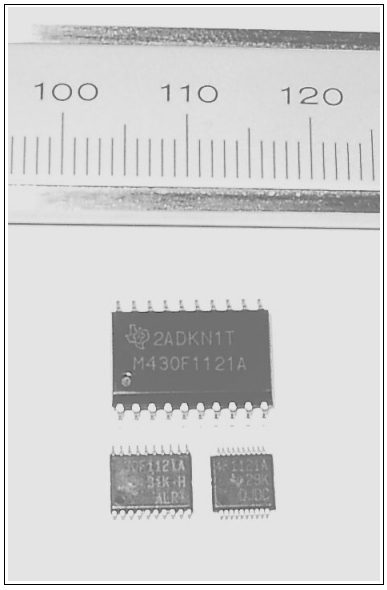

Abbildung 3.1: MSP430F1121A in verschiedenen Bauformen

Eine fast schon maßgeschneiderte Lösung für dieses Problem ist der MSP430[11] von Texas Instruments (siehe Abbildung 3.1). Dieser Controller ist für kleine batteriebetriebene Geräte entwickelt worden und benötigt bis auf die Stromversorgung keine weiteren externen

11 Der MSP430F1121A ist ein sehr kostengünstiger Microcontroller, der speziell für den Einsatz in batteriebetriebenen Geräten entwickelt wurde. Er beinhaltet eine 16-Bit RISC-CPU und stellt insgesamt 14 I/O-Pins zur Verfügung. Selbst bei einer Taktung mit 8 Mhz liegt die Stromaufnahme nur bei verschwindend geringen 350µA. Bei Abnahme von 1000 Stück liegt der Preis bei weniger als 2 Euro.

Bauelemente.

3.2.2 Dateneingang

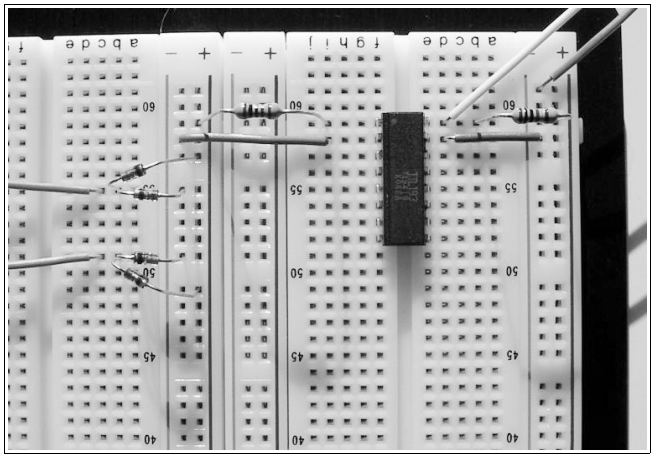

Abbildung 3.2: Schaltungsaufbau des Dateneingangs

Um die aus dem Motorport versendeten Daten mit dem Microcontroller auswerten zu können, ist eine Interface-Schaltung (siehe Abbildung 3.2) nötig, die für die Pegelanpassung sorgt. Am Motorausgang liegt jeweils die volle Batteriespannung des RCX an. Um sich gegen Verpolung bzw. Eine Drehrichtungsumkehr des Motorausgangs abzusichern, muß ein Gleichrichter vorgesehen werden.

Denkt man in diesem Zusammenhang über die Stromversorgung der Schaltung nach, würde es sich anbieten, aus dem Datensignal gleichzeitig die Betriebsspannung zu gewinnen. Bei einer Stromaufnahme des Prozessors von weniger als 1mA und einem ähnlich niedrigen Stromverbrauch der Analog-Multiplexer würde das problemlos möglich sein. Da der gesamte Multiplexer aber noch weitere Komponenten[12] beinhalten soll, die zum Teil einen wesentlich größeren Strombedarf haben, bleibt dieser Ansatz einem Low-Budget-Mux vorbehalten, der ausschließlich Sensoren umschaltet.

3.2.3 Acknowledge-Ausgang

Da die Datenübertragung über den Motorausgang unidirektional in Richtung des

12 Hier war der Gedanke, die Resourcen des Prozessors möglichst gut auszunutzen, um seinen Einsatz nicht nur mit der Decodierung des seriellen Protokolls zu rechtfertigen. Somit ergeben sich weitere Komponenten, die zwingend auf eine zusätzliche Stromversorgung angewiesen sind (z.b. Motorausgänge und die IR-Detektoren).

Multiplexers läuft, benötigt man für eine entsprechend gute Störfestigkeit auf jeden Fall eine Rückmeldung an den RCX.
Zwangsläufig bleibt hier nur der Ausweg, einen der Sensoreingänge für diesen Zweck zu opfern. Da diese Eingänge aber Analogwerte von 10 Bit auflösen können, wäre es pure Verschwendung, dort nur ein binäres Signal als Empfangsbestätigung zu übermitteln. Es soll demnach unbedingt möglich sein, parallel zur Bestätigung weitere Daten zu senden. Dies geschieht durch ein mindestens drei Bit tiefes Auffächern des Wertebereichs. Schaltungstechnisch läßt sich das am einfachsten über ein entsprechend geschaltetes Widerstandsnetzwerk erreichen. Alternativ wäre auch die Verwendung eines (vergleichsweise teuren, aber präziseren) Digital-Potentiometers denkbar.

3.2.4 Multiplexer

Beim Schalten mehrerer Sensoren auf einen weiteren Sensoreingang ist aus elektrischer Sicht zu beachten, daß die Schalter so niederohmig wie möglich sein müssen. Schalter mit einem hohen On-Widerstand würden zu einer Verfälschung der Sensorwerte und zu einer schlechten Spannungsversorgung aktiver Sensoren führen. Letzeres wäre gerade bei mehreren aktiven Sensoren ein Problem, da zuviel Zeit vergehen würde, um den sensorinternen Pufferkondensator zu laden. Da in unserem Fall wegen der Polungsunabhängigkeit beide Anschlüsse eines Sensors geschaltet werden sollen, verdoppelt sich auch der gesamte On-Widerstand.
Am einfachsten und kostengünstigsten läßt sich der Sensormultiplexer mit Bausteinen der 4000er CMOS-Logikfamilie[13] aufbauen.

3.2.5 Motorausgänge

13 Diese Logikfamilie ist schon recht betagt, aber selbst heute noch wegen ihres niedrigen Stromverbrauchs und des großen Betriebsspannungsbereichs beliebt. Wie auch im Gasperi-Mux empfiehlt sich die Verwendung bilateralter Schalter. Diese gibt es als einzelne Schalter oder auch in Form eines 1:2, 1:4 oder 1:8 Multiplexers. Die Bauteilkosten bewegen sich im Bereich weniger Cent.

Als Ersatz für den als Datenausgang mißbrauchten Motorport des RCX soll der Multiplexer auch zwei Lego-kompatible Motorausgänge bereitstellen. Der Prozessor sorgt dabei für eine Geschwindigkeitsregelung per Pulsweitenmodulation.
Aus elektrischer Sicht empfiehlt es sich, möglichst nah am Original zu bleiben. Zwar sind die im RCX verwendeten Motortreiber[14] relativ teuer, dafür bieten sie aber diverse Schutzmechanismen und sind in SMD-Bauform erhältlich. Letzeres ist unerläßlich für eine kompakte Bauweise des gesamten Multiplexers.

3.2.6 Integration der Beacon-Erkennung

Der in Kapitel 2 verfolgte Ansatz einer Beacon-Erkennung mit Hilfe eines Frequenz-/Spannungswandlers bringt uns im Zusammenhang mit einem Microcontroller nicht weiter. Statt dessen bietet es sich an, die Signale der IR-Empfänger direkt mit dem Prozessor zu verarbeiten.

3.3 Protokoll

Das hier verwendete Protokoll ist so beschaffen, daß es ohne getrennte Takt- und Datenleitungen auskommt. Dazu wird vor jedem Datenwort eine fünf Bit lange Präambel gesendet, anhand derer der Empfänger die Bit-Zeit messen kann. Danach

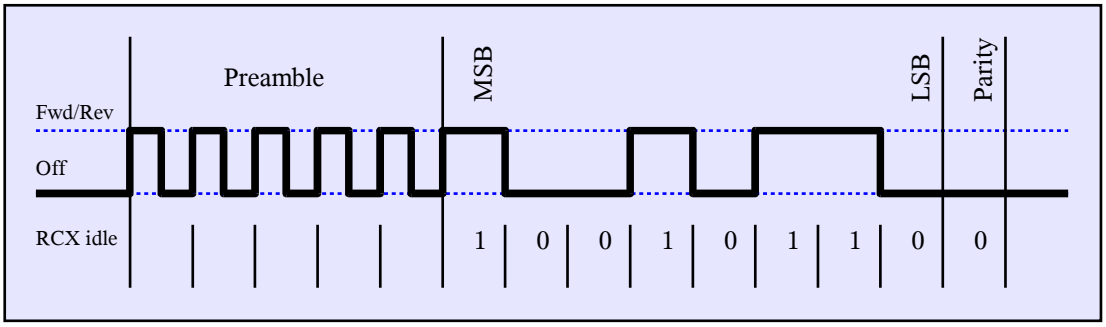

Abbildung 3.3: Übertragungsprotokoll

14 Die Firma Melexis (siehe *http://www.melexis.com*) baut speziell auf kleine Motoren ausgelegte H-Brücken-Treiber des Typs MLX10402, die direkt über TTL-Eingänge angesteuert werden. Durch die eingebaute Logik ist es über eine 2-Bit-Ansteuerung möglich, die H-Brücke in die Zustände 'vorwärts', 'rückwärts', 'aus' und 'bremsen' zu schalten. Selbst bei relativ großen Stückzahlen bewegt sich der Preis pro Treiber bei ca. 2,50 Euro.

folgen acht Datenbits beginnend mit dem Most-Significant-Bit (MSB) und einem abschließendem Parity-Bit. Die Parität wird anhand von XOR-Verknüpfungen der Datenbits berechnet. Es ergibt sich also eine gerade Parität.

Die Präambel ist so beschaffen, daß innerhalb einer Bit-Zeit zwei Flankenwechsel auftreten. Dadurch ergeben sich neun Flankenwechsel, zwischen denen jeweils eine halbe Bitzeit vergeht. Mißt man nun mit einem Microcontroller die Zeiten zwischen diesen Flanken und bildet darüber den Mittelwert, läßt sich auch bei leichten Schwankungen der Flankenabstände eine passende Bit-Zeit errechnen.

Das letzte Bit der Präambel dient der Fehlererkennung. Hier sollte sinnvollerweise der Leitungspegel innerhalb beider Bithälften einmal überprüft werden. Tritt hier kein High-Pegel gefolgt von einem Low-Pegel auf, mißt man nicht synchron zum Anfang des Datenworts. In diesem Fall muß der Rest der Daten verworfen und für mindestens neun Bitzeiten gewartet werden, bevor die nächste Präambel eingelesen werden kann.

Jedes gesendete Datenwort wird vom Empfänger über den Acknowledge-Ausgang bestätigt. Nach einer erfolgreichen Übertragung können direkt weitere Daten versandt werden. Im Fehlerfall empfiehlt sich eine Wartezeit seitens des Senders von ca. fünf Bitzeiten, damit sich der Empfänger wieder synchronisieren kann.

3.4 Interface-Testschaltung

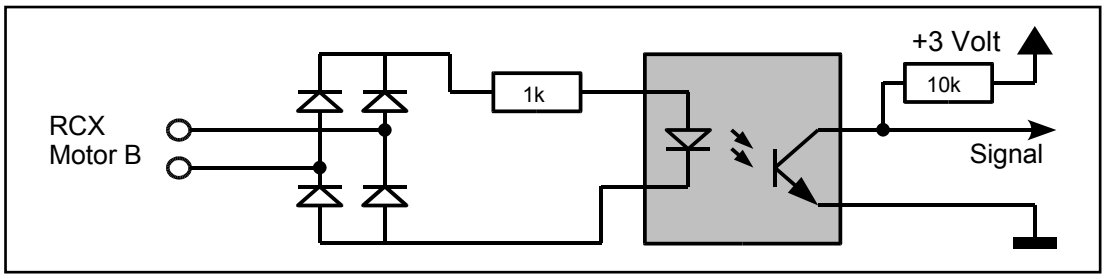

Abbildung 3.4: Schaltplan des Dateneingangsinterface

Die ursprüngliche Testschaltung bestand aus einem Gleichrichter und einem

Transistor zur Pegelanpassung. Das hatte allerdings den Nachteil, daß hierbei das Massepotential vom Gleichrichter abgegriffen wurde. Will man gleichzeitig den Acknowledge-Ausgang an den RCX anschließen, so liegt das Massepotential um zwei Diodenspannungen höher als die RCX-Masse. Der Grund dafür ist die im Motortreiber verwendete H-Brücke. Diese Differenz würde den Wertebereich des Acknowledge-Ausgangs verfälschen und eine negative Versorgungsspannung für die Analog-Multiplexer erfoderlich machen.

Eine einfache Lösung für dieses Problem ist die Verwendungen eines Optokopplers an Stelle des Transistors (siehe Abbildung 3.4). Das Massepotential wird dann über einen Gleichrichter[15] aus dem Acknowledge-Sensoreingang gewonnen.

15 Für diesen Gleichrichter sollten unbedingt Schottky-Dioden verwendet werden, da dort der Spannungsabfall nicht so hoch ist wie bei Silizium-Dioden. Mit den typischen 0,7 Volt der Si-Dioden würde sich am Analog-Mux eine Eingangsspannung von mehr als -0,5 Volt gegenüber der Mux-Betriebsspannung ergeben, was wiederum gegen die Spezifikationen des Chipherstellers verstoßen würde.

4 BrickOS

Die von Lego mitgelieferte Software zur Programmerstellung wurde für Schulkinder konzipiert. Sie ist demnach sehr einfach gehalten und taugt nicht zur Erstellung komplexer Programme.

Wie schon bei dem Problem der knapp bemessenen Ein- und Ausgänge, hat auch hier die Lego-Fangemeinde viele Lösungsansätze zu bieten.

Die Firmware des RCX läßt es zu, daß das System teilweise oder auch komplett ohne die BIOS-Funktionen betrieben wird. Darauf basierend gibt es verschiedene Ansätze, neue Betriebssyteme zu implementieren.

Eines dieser Betriebssysteme nennt sich BrickOS[16] und stellt eine komplette Multitasking-Umgebung zur Verfügung.

Der BrickOS-Kernel selbst und auch die zugehörigen Programme sind in C geschrieben und lassen sich mit einer speziellen Version des GCC übersetzen. Die dabei entstandenen Binärdateien werden über den IR-Tower[17] in den RCX geladen. Dabei bleibt ein einmal geladener BrickOS-Kernel so lange im RCX, bis er entweder manuell gelöscht wird oder die Batterien entnommen werden.

4.1 Timing-Problematik

BrickOS wird mit einem "Systemtakt" von einem Kilohertz betrieben. Das bedeutet, daß einmal pro Millisekunde der Kernel innerhalb seines Systeminterrupts sämtliche Komponenten des RCX mit neuen Daten versorgt bzw. Meßwerte von den Sensoreingängen abfragt. Dazu zählen unter anderem auch der Display-Refresh, die Wiedergabe von Tönen und das Task-Scheduling.

Versucht man nun, einen Motorausgang zur Datenübertragung zweckzuentfremden, stellt sich schnell heraus, daß einem solch ein Multitasking-Kernel das Leben nicht

16 BrickOS hatte ursprünglichen den Namen LegOS, was aber zu namensrechtlichen Problemen führte. Die offizielle Webseite dieses Projekts findet sich unter *http://brickos.sourceforge.net*.

17 Dies ist ein Infrarot-Sender, der über eine serielle Schnittstelle (bzw. USB) an einen Rechner angeschlossen wird. Hierüber ist eine bidirektionale Kommunikation mit dem RCX möglich.

gerade leicht macht.

Deutlich wird das bei der Ausgabe eines Signals, das bei jedem System-Interrupt (also jede Millisekunde) den Pegel wechselt. Dies funktioniert ansich gut, aber sowie mehrere Tasks benutzt werden oder das System andere zeitraubende Dinge zu erledigen hat, kommt es häufig vor, daß das Signal für bis zu zwei Millisekunden den Pegel beibehält. Für unser Protokoll hat das eine Menge an Übertragungsfehlern zur Folge.

4.2 Struktur des BrickOS-Systems

Um zu verstehen, woher die Timingprobleme kommen, bedarf es einer genauen Untersuchung des BrickOS-Kernels.

Obwohl der H8-Prozessor des RCX eine ganze Reihe von Timern bietet, benutzt BrickOS davon nur den ORCA[18]. Dies ist ein 16-Bit-Timer, der so programmiert wird, daß er jede Millisekunde einen Interrupt auslöst.

Innerhalb dieser Interrupt-Routine kümmert sich BrickOS um die Systemresourcen. Neben dem Task-Scheduling erfolgt hier unter anderem das Auslesen der Sensoren und Schalten der Motorausgänge. Zur Sound-Ausgabe wird eine ROM-interne Routine benutzt, die auch innerhalb dieses Interrupts aufgerufen wird.

Da der Kernel verschiedene Sensortypen direkt unterstützt und die Sensorwerte gleich passend umrechnet, kann das Zeitverhalten des System-Interrupts stark variieren.

Zur Ansteuerung der Motoren berechnet BrickOS eine Pulsweitenmodulation anhand eines 8-Bit-Counters. Auf diesen Counter wird jeweils der aktuelle Speed-Wert addiert. Bei einem Überlauf wird der zugehörige Motor eingeschaltet. Selbst bei einem Speed-Wert von 255 bedeutet das aber nicht, daß der Motor permanent

18 Der H8 besitzt zwei Timer-Sektionen. Eine davon beherbergt vier 8-Bit-Timer (hier nicht näher erläutert) und die andere einen 16-Bit-Free-Running-Counter (FRC), an den Output-Compare- (OC) und Input-Capture-Register (IC) angelagert sind. Es existieren zwei OC-Register (OCRA und OCRB), über die sich Interrupts auslösen lassen, sofern ihr Inhalt mit dem FRC übereinstimmt.

eingeschaltet ist.[19]

Weder die Motoransteuerung, noch die Struktur des Systeminterrupts gestatten eine zeitkritische Datenausgabe über den Motorausgang. Um dennoch die Ausgabe des oben beschriebenen Protokolls zu ermöglichen, ist es nötig, den Kernel zu modifizieren.

4.3 Änderungen des Interrupt-Systems

Zur Entkopplung der Motoransteuerung von den restlichen Aufgaben des Systeminterrupts, empfiehlt es sich, einen weiteren Interrupt zu benutzen. Leider hat der OCRA-Interrupt bereits die höchste Priorität. Demnach würde eine Motoransteuerung über OCRB trotzdem vom Systeminterrupt blockiert werden können. Aus diesem Grund muß der Systeminterrupt auf OCRB verlegt werden.[20]

Benutzt man nun OCRA ausschließlich zur Motoransteuerung, verbessert sich das Zeitverhalten deutlich.

Einziger Pferdefuß ist die ROM-Routine zur Sound-Ausgabe, da sie fest mit dem OCRA-Interrupt verdrahtet ist. Diese Kopplung verbietet auch eine Beschleunigung des OCRA auf z.B. 500 Mikrosekunden[21], da sich dadurch die Frequenzen der Sound-Ausgabe verschieben würden.

19 Ist der aktuelle Wert des Counters gleich null, wird selbst bei einer Addition des maximalen Speed-Wertes von 255 kein Überlauf generiert. Der Motorausgang wird also trotz der maximalen Geschwindigkeitseinstellung für eine Millisekunde ausgeschaltet.

20 Die nötigen Modifikationen werden in der Datei systime.c des BrickOS-Kernels vorgenommen. Dort wird ein weiterer Interrupthandler für OCRB implementiert, der die bisherige Systeminterrupt-Routine aufruft. Aus dieser Routine wird der Aufruf der Motoransteuerung auskommentiert und in einen getrennten Interrupthandler verlegt.

21 Eine Beschleunigung der Motoransteuerung würde eine höhere Datentransferrate ermöglichen.

4.4 Kernelerweiterung

Um innerhalb des Motor-Interrupts tatsächlich Daten ausgeben zu können, muß die zugehörige Routine entsprechend umgeschrieben werden.

Die schon vorhandenen Funktionen zum Setzen von Drehrichtung und Geschwindigkeit werden für den Datentransport zweckentfremdet. Mit einem fünften Motor-State „data“ läßt sich Motorausgang „B“ in den Übertragungsmodus schalten. Der Speed-Wert repräsentiert das zu versendende Datenbyte. Sowie dieser Wert größer als null ist, beginnt der Kernel automatisch mit dem Versand der Daten. Der PWM-Counter dient in diesem Fall als Bitzähler. Nach einer kompletten Übertragung wird der Speed-Wert wieder auf null gesetzt.[22]

Da der Motor-Interrupt einmal pro Millisekunde erfolgt, ergibt sich daraus eine Bitzeit von zwei Millisekunden, denn innerhalb der Präambel wird ein Flankenwechsel pro Halbbit benötigt. Insgesamt dauert damit die Übertragung eines Datenworts 28 Millisekunden. Zusätzlich müssen hier aber noch einige Millisekunden für das Warten auf das Acknowledge-Signal eingerechnet werden, sodaß man im Endeffekt eine Übertragungsleistung von ca. 30 Byte/s erreicht.

22 Die Änderungen betreffen den Assembler-Code innerhalb der Dateien dmotor.c und dmotor.h des Kernels. Um den Kernel nicht unnötig aufzublähen, wird die Datenausgabe ausschließlich für Motor B implementiert.

Schlußwort

Ziel der Studienarbeit war es, einen Überblick über die Funktionsweise der am RCX vorhandenen Schnittstellen zu gewinnen, um für dieses System neue Erweiterungen entwickeln zu können. Konkret ging es dabei um Ansätze für einen universellen Sensormultiplexer und für einen IR-Beacon-Detektor. Die nähere Untersuchung hat gezeigt, daß der Bau von Lego-kompatiblen Erweiterungen in den meisten Fällen keineswegs trivial ist.

Im Falle des Sensormultiplexers bestand das grundlegende Problem in der Übertragung der Steuerdaten. Hier half erst eine Anpassung der RCX-internen Software weiter. Durch diesen zwangsläufig nötigen Kernel-Patch ist der Multiplexer nur in Verbindung mit BrickOS benutzbar. Eine ähnliche Kernel-Anpassung für andere RCX-Betriebssystem wäre prinzipiell zwar möglich, ist aber sehr aufwendig.

Der Ansatz für die Schaltung des IR-Beacon-Detektors wurde durch einige elektrotechnische Details zunichte gemacht. Um hier für Abhilfe zu sorgen, hätte man den Aufbau komplett verändern müssen, was aber zu einer wesentlich aufwendigeren Schaltung geführt hätte. Als Ausweg bot sich an, beide Entwürfe miteinander zu verschmelzen, da die Rechenleistung des im Multiplexer verwendeten Microcontrollers auch zur Auswertung der Beacon-Signale ausreichen sollte.

Zusammenfassend läßt sich sagen, daß es einen recht großen Aufwand erfordert, aus den Lego Mindstorms eine Plattform für anspruchsvolle Lego-Roboter zu machen. Grund hierfür ist das interne Design des RCX. Man merkt sehr deutlich, daß die Lego-Entwickler dieses System nicht als Ersatz für das beliebte 6.270-Board des MIT entwickelt haben.

Bei der Umsetzung des hier beschriebenen Multiplexers sollte auf jeden Fall darauf geachtet werden, daß die bisher bestehenden Abhängigkeiten von dem verwendeten RCX-Betriebssystem so weit wie möglich beseitigt werden. Damit hätte man im besten Fall eine universelle Erweiterung, die sich mit beliebigen Betriebssytemen

benutzen ließe.

Außerdem wird sich dabei zeigen, ob es sinnvoll ist, den RCX mit hohem Aufwand zu einer universellen Plattform für mobile Roboterprojekte zu machen oder ob man lieber den umgekehrten Weg geht und ein Prozessorboard entwickelt, das auch Schnittstellen zu Lego-Sensoren bietet.

Literaturverzeichnis

BrickOS : The BrickOS Homepage
http://brickos.sourceforge.net

CD4051 : Single 8 Ch. Analog Multiplexer
http://www.fairchildsemi.com/ds/CD/CD4051BC.pdf

Elektor 4/2002 : Seite 28, I²C Bus für den RCX

Gasperi : MindStorms RCX Sensor Input Page
http://www.plazaearth.com/usr/gasperi/lego.htm

Gasperi Mux : The RCX Input Mux
http://www.plazaearth.com/usr/gasperi/mux.htm

Hi-Technic : HiTechnic LEGO RCX Sensors
http://www.hitechnic.com

Hitachi : Hitachi Semiconductor Homepage
http://www.hitachisemiconductor.com

Hitachi H8 : Hitachi H8/300 Serie
http://www.hitachisemiconductor.com/sic/jsp/japan/eng/products/mpumcu/816bit/h8300/3297

LM331 : Precision Voltage-to-Frequency Converter
http://www.national.com/pf/LM/LM331.html

MELEXIS : Microelectronic integrated systems
http://www.melexis.com

MLX10402 : MLX 10402 Motor Driver IC online
http://www.melexis.com/prodfiles/mlx10402.pdf

Mindsensors : Mindsensors Robotics Homepage
http://www.mindsensors.com

Mindstorms : LEGO Mindstorms
http://mindstorms.lego.com

MSP430 : MSP430 Ultra-Low Power Microcontrollers : MSP430 Home
http://www.ti.com/msp430

Sharp : Sharp Microelectronics Europe
http://www.sharp-sme.com

Sharp Sensors : Sharp GP2D12 Optical Sensor
http://sharp-world.com/products/device/ctlg/esite22_10/table/pdf/osd/optical_sd/gp2d120_e.pdf

TSOP17xx : TSOP17.. Photo Modules for PCM Remote Control Systems
http://www.vishay.com/docs/tsop17.pdf

TSOP18xx : TSOP18.. Photo Modules for PCM Remote Control Systems
http://www.vishay.com/docs/82047/82047.pdf

Anhang

Nachfolgend sind die in Kapitel 4 angesprochenen Kerneländerungen aufgeführt.

```
/* WARNING! This file was modified to support datatransfer to lepomux devices.
   These modifications are not part of the original BrickOS distribution.
   Use it at your own risk!
   This file was taken from version 0.2.6.07nmChg of BrickOS.
   2003-01-12 - Gunther Lemm <lemm@lepomux.org>
*/

/*! \file   dmotor.c
    \brief  direct motor access
    \author Markus L. Noga <markus@noga.de>
*/


/*
 *  The contents of this file are subject to the Mozilla Public License
 *  Version 1.0 (the "License"); you may not use this file except in
 *  compliance with the License. You may obtain a copy of the License at
 *  http://www.mozilla.org/MPL/
 *
 *  Software distributed under the License is distributed on an "AS IS"
 *  basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the
 *  License for the specific language governing rights and limitations
 *  under the License.
 *
 *  The Original Code is legOS code, released October 17, 1999.
 *
 *  The Initial Developer of the Original Code is Markus L. Noga.
 *  Portions created by Markus L. Noga are Copyright (C) 1999
 *  Markus L. Noga. All Rights Reserved.
 *
 *  Contributor(s): Markus L. Noga <markus@noga.de>
 *                  Lou Sortman <lou (at) sunsite (dot) unc (dot) edu>
 */

#include <sys/dmotor.h>
#include <dlcd.h>

#ifdef CONF_DMOTOR

#include <sys/h8.h>
#include <sys/irq.h>

////////////////////////////////////////////////////////////////////////////
//
// Variables
//
////////////////////////////////////////////////////////////////////////////

//! motor drive patterns
/*! to be indexed with MotorDirections
    \sa MotorDirections
*/

#ifdef CONF_DMOTOR_HOLD
const unsigned char dm_a_pattern[]={0xc0,0x40,0x80,0x00,0xc0},
                    dm_b_pattern[]={0x0c,0x04,0x08,0x00,0x10},
                    dm_c_pattern[]={0x03,0x01,0x02,0x00,0x03};
```

```
#else

const unsigned char dm_a_pattern[]={0x00,0x80,0x40,0xc0,0x00},
                    dm_b_pattern[]={0x00,0x08,0x04,0x0c,0x10},
                    dm_c_pattern[]={0x00,0x02,0x01,0x03,0x00};
#endif

MotorState dm_a,                  //!< motor A state
           dm_b,                  //!< motor B state
           dm_c;                  //!< motor C state

////////////////////////////////////////////////////////////////////////////////
//
// Functions
//
////////////////////////////////////////////////////////////////////////////////

//! direct motor output handler
/*! called by system timer in the 16bit timer OCIA irq
*/
extern void dm_handler(void);

__asm__("
.text
.align 1
.global _dm_handler
_dm_handler:
               ; r6 saved by ROM
                ; r0 saved by systime_handler
"
#ifdef CONF_DMOTOR_HOLD
"              mov.b #0xcf,r6l                      ; r6l is output\n"
#else
"              sub.w r6,r6                   ; r6l is output\n"
#endif
"               ; we simultaneously load delta (r0h) and sum (r0l)
               ; this depends on byte order, but the H8 will stay MSB
               ; and the resulting code is efficient and compact.

               ; r6l = motor output byte
               ; r6h = drivepattern output tempvar

               ; @_dm_x+0 = delta (speed)
               ; @_dm_x+1 = sum   (counter)
               ; @_dm_x+2 = direction (off,fwd,rev,brake,data)

               ; motor a - fwd: 10000000
               ; motor b - fwd: 00001000
               ; motor c - fwd: 00000010
               ; motor a - rev: 01000000
               ; motor b - rev: 00000100
               ; motor c - rev: 00000001




                ; motor A

                mov.w   @_dm_a,r0
                add.b   r0h,r0l                 ; add delta to sum
                bcc     dm0                     ; sum overflow?
               mov.b   @_dm_a+2,r6h            ; -> output drive pattern
```

```
   xor.b   r6h,r6l
dm0:mov.b   r0l,@_dm_a+1            ; save sum

                                    ; (clears overflow flag)

   ; motor B

   mov.b   @_dm_b+2,r0l           ; data mode?
   cmp.b   #0x10,r0l
   beq     dat                    ; motor b = data output

mob:mov.w   @_dm_b,r0             ; motor b = normal output
    add.b   r0h,r0l                 ; add delta to sum
    bcc     dm1                     ; sum overflow?
   mov.b   @_dm_b+2,r6h            ; -> output drive pattern
   xor.b   r6h,r6l
dm1:mov.b   r0l,@_dm_b+1            ; save sum
    bra    moc                     ; (clears overflow flag)
```

```
; ================ data output on motorport 2 ===============
; @_dm_b   (speed)        data to send
; @_dm_b+1 (sum)          internal halfbit counter
; @_dm_b+2 (direction)    data mode if set to 'data'

; -------------------- bitcounter init ----------------------
dat:mov.b  @_dm_b,r0l            ; send or idle?
    beq    moc
    mov.b  @_dm_b+1,r0l          ; load counter
    bne    da2                   ; start or continue?
    mov.b  #30,r0l                     ; init halfbit counter
da2:dec    r0l
    mov.b  r0l,@_dm_b+1          ; save counter
    cmp.b  #2,r0l                ; counter > 2 ?
    bmi    da7                   ; send parity
    cmp.b  #18,r0l                     ; counter > 16 ?
    bmi    da3                   ; send databyte

; ---------------------- send preamble ----------------------
    and.b  #1,r0l                ; preamb: odd or even?
    beq    da4
    bset   #3,r6l                ; even: output preamb hi
    bra    da6
da4:bclr   #3,r6l                ; odd: output preamb low
    bra    da6

; ---------------------- parity check -----------------------
da7:mov.b  #8,r6h                ; init bitcounter
    mov.b  @_dm_b,r0l            ; fetch data
    mov.b  #0,r0h                ; init xor-register
da8:xor.b  r0l,r0h                     ; xor (LSB is relevant)
    shlr.b r0l                   ; shift to next bit
    dec    r6h                   ; dec bitcounter
    bne    da8                   ; repeat
    btst   #0,r0h                ; test parity bit
    bra    da9

; ---------------------- data output ------------------------
da3:mov.b  @_dm_b+1,r0l          ; counter -> r0l
    mov.b  @_dm_b,r0h            ; data -> r0h
    mov.b  #2,r6h
    sub.b  r6h,r0l                     ; counter-2
    shlr.b r0l                   ; counter/2
    btst   r0l,r0h                     ; send data bit # r0l
da9:bne    da5
    bclr   #3,r6l                ; output hi
    bra    da6
da5:bset   #3,r6l                ; output low

da6:mov.b  @_dm_b+1,r0l
    bne    moc                   ; everything sent?
    mov.b  #0,r0l
    mov.b  r0l,@_dm_b            ; done: clear data register
; ============================================================

     ; motor C

moc:mov.w   @_dm_c,r0
     add.b   r0h,r0l               ; add delta to sum
     bcc     dm2                   ; sum overflow?
    mov.b  @_dm_c+2,r6h           ; -> output drive pattern
    xor.b  r6h,r6l
```

```
            dm2:mov.b   r0l,@_dm_c+1            ; save sum

               ; driver chip

               mov.b   r6l,@0xf000:16        ; output motor waveform

               rts
       ");



//! initialize motors
//
void dm_init(void) {
  dm_shutdown();                            // shutdown hardware
}


//! shutdown motors
//
void dm_shutdown(void) {
  motor_controller=0x00;             // shutdown hardware

  motor_a_dir(off);                  // initialize driver data
  motor_b_dir(off);
  motor_c_dir(off);

  motor_a_speed(MAX_SPEED);
  motor_b_speed(MAX_SPEED);
  motor_c_speed(MAX_SPEED);
}

#ifdef CONF_VIS
/*
** Define non-inline versions to display arrows
*/

void motor_a_dir(MotorDirection dir)
{
  dm_a.dir = dm_a_pattern[dir];
  dlcd_hide(LCD_A_LEFT);
  dlcd_hide(LCD_A_RIGHT);
  if (dir == fwd || dir == brake)
    dlcd_show(LCD_A_RIGHT);
  else if (dir == rev || dir == brake)
    dlcd_show(LCD_A_LEFT);
}

void motor_b_dir(MotorDirection dir)
{
  dm_b.dir = dm_b_pattern[dir];
  dlcd_hide(LCD_B_LEFT);
  dlcd_hide(LCD_B_RIGHT);
  if (dir == fwd || dir == brake)
    dlcd_show(LCD_B_RIGHT);
  else if (dir == rev || dir == brake)
    dlcd_show(LCD_B_LEFT);
}

void motor_c_dir(MotorDirection dir)
{
```

```
  dm_c.dir = dm_c_pattern[dir];
  dlcd_hide(LCD_C_LEFT);
  dlcd_hide(LCD_C_RIGHT);
  if (dir == fwd || dir == brake)
    dlcd_show(LCD_C_RIGHT);
  else if (dir == rev || dir == brake)
    dlcd_show(LCD_C_LEFT);
}

#endif // ifdef CONF_VIS

#endif // CONF_DMOTOR
```

```
/* WARNING! This file was modified to support datatransfer to lepomux devices.
   These modifications are not part of the original BrickOS distribution.
   Use it at your own risk!
   This file was taken from version 0.2.6.07nmChg of BrickOS.
   2003-01-12 - Gunther Lemm <lemm@lepomux.org>
*/

/*! \file   include/dmotor.h
    \brief  direct motor access
    \author Markus L. Noga <markus@noga.de>
 */

/*
 *  The contents of this file are subject to the Mozilla Public License
 *  Version 1.0 (the "License"); you may not use this file except in
 *  compliance with the License. You may obtain a copy of the License
 *  at http://www.mozilla.org/MPL/
 *
 *  Software distributed under the License is distributed on an "AS IS"
 *  basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
 *  the License for the specific language governing rights and
 *  limitations under the License.
 *
 *  The Original Code is legOS code, released October 17, 1999.
 *
 *  The Initial Developer of the Original Code is Markus L. Noga.
 *  Portions created by Markus L. Noga are Copyright (C) 1999
 *  Markus L. Noga. All Rights Reserved.
 *
 *  Contributor(s): Markus L. Noga <markus@noga.de>
 */

#ifndef __dmotor_h__
#define __dmotor_h__

#ifdef  __cplusplus
extern "C" {
#endif

#include <config.h>

#ifdef CONF_DMOTOR

////////////////////////////////////////////////////////////////////////
//
// Definitions
//
////////////////////////////////////////////////////////////////////////

//! the motor directions
typedef enum {
  off = 0,                  //!< freewheel
  fwd = 1,                  //!< forward
  rev = 2,                  //!< reverse
  brake = 3,                //!< hold current position
  data = 4                  //!< motor b = data output

} MotorDirection;

#ifndef DOXYGEN_SHOULD_SKIP_INTERNALS
//! the motor status type.
typedef struct {
```

```
  union {
    unsigned assembler;              //!< assures word alignment for assembler

    struct {
      unsigned char delta;    //!< the speed setting

      volatile unsigned char sum;     //!< running sum

    } c;
  } access;                   //!< provides access from C and assembler


  unsigned char dir;          //!< output pattern when sum overflows

} MotorState;
#endif  // DOXYGEN_SHOULD_SKIP_INTERNALS

#define  MIN_SPEED     0       //!< minimum motor speed
#define  MAX_SPEED     255     //!< maximum motor speed

////////////////////////////////////////////////////////////////////////
//
// Variables
//
////////////////////////////////////////////////////////////////////////

//! motor drive patterns
/*! to be indexed with MotorDirections
   \sa MotorDirections
 */
extern const unsigned char    dm_a_pattern[5],
                              dm_b_pattern[5],
                              dm_c_pattern[5];

extern MotorState      dm_a,   //!< motor A state
                       dm_b,   //!< motor B state
                       dm_c;   //!< motor C state

////////////////////////////////////////////////////////////////////////
//
// Functions
//
////////////////////////////////////////////////////////////////////////

#ifdef CONF_VIS

/*
** motor_*_dir() functions will display direction arrows, so
** define them in kernel/dmotor.c
*/
extern void motor_a_dir(MotorDirection dir);
extern void motor_b_dir(MotorDirection dir);
extern void motor_c_dir(MotorDirection dir);

#else

/*
** No display, so make these functions inline
*/
//! set motor A direction
/*! \param dir the direction
 */
```

```
extern inline void motor_a_dir(MotorDirection dir)
{
  dm_a.dir = dm_a_pattern[dir];
}

//! set motor B direction
/*! \param dir the direction
 */
extern inline void motor_b_dir(MotorDirection dir)
{
  dm_b.dir = dm_b_pattern[dir];
}

//! set motor C direction
/*! \param dir the direction
 */
extern inline void motor_c_dir(MotorDirection dir)
{
  dm_c.dir = dm_c_pattern[dir];
}

#endif // ifdef CONF_VIS


//! set motor A speed
/*! \param speed the speed
 */
extern inline void motor_a_speed(unsigned char speed)
{
  dm_a.access.c.delta = speed;
}

//! set motor B speed
/*! \param speed the speed
 */
extern inline void motor_b_speed(unsigned char speed)
{
  dm_b.access.c.delta = speed;
}

//! set motor C speed
/*! \param speed the speed
 */
extern inline void motor_c_speed(unsigned char speed)
{
  dm_c.access.c.delta = speed;
}

#endif // CONF_DMOTOR

#ifdef  __cplusplus
}
#endif

#endif // __dmotor_h__
```

```
/* WARNING! This file was modified to support datatransfer to lepomux devices.
   These modifications are not part of the original BrickOS distribution.
   Use it at your own risk!
   This file was taken from version 0.2.6.07nmChg of BrickOS.
   2003-01-12 - Gunther Lemm <lemm@lepomux.org>
*/
/*! \file systime.c
    \brief  system time services
    \author Markus L. Noga <markus@noga.de>
*/
/*
 *  The contents of this file are subject to the Mozilla Public License
 *  Version 1.0 (the "License"); you may not use this file except in
 *  compliance with the License. You may obtain a copy of the License at
 *  http://www.mozilla.org/MPL/
 *
 *  Software distributed under the License is distributed on an "AS IS"
 *  basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the
 *  License for the specific language governing rights and limitations
 *  under the License.
 *
 *  The Original Code is legOS code, released October 17, 1999.
 *
 *  The Initial Developer of the Original Code is Markus L. Noga.
 *  Portions created by Markus L. Noga are Copyright (C) 1999
 *  Markus L. Noga. All Rights Reserved.
 *
 *  Contributor(s): Markus L. Noga <markus@noga.de>
 *                  David Van Wagner <davevw@alumni.cse.ucsc.edu>
 */
/*
 *  2000.05.01 - Paolo Masetti <paolo.masetti@itlug.org>
 *
 * - Added battery indicator handler
 *
 *  2000.08.12 - Rossz Vámos-Wentworth <rossw@jps.net>
 *
 * - Added idle shutdown handler
 *
 */

#include <config.h>

#ifdef CONF_TIME

#include <sys/time.h>
#include <sys/h8.h>
#include <sys/irq.h>
#include <sys/dmotor.h>
#include <sys/dsound.h>
#include <sys/battery.h>
#ifdef CONF_AUTOSHUTOFF
#include <sys/timeout.h>
#endif

////////////////////////////////////////////////////////////////////////////
//
// Global Variables
//
////////////////////////////////////////////////////////////////////////////

//! current system time in ms
```

```
/*! \warning This is a 32 bit value which will overflow after 49.7 days
             of continuous operation.
*/
volatile time_t sys_time;


////////////////////////////////////////////////////////////////////////////
//
// Internal Variables
//
////////////////////////////////////////////////////////////////////////////

#ifdef CONF_TM
volatile unsigned char tm_timeslice;            //!< task time slice
volatile unsigned char tm_current_slice;        //!< current time remaining

void* tm_switcher_vector;                       //!< pointer to task switcher
#endif


////////////////////////////////////////////////////////////////////////////
//
// Functions
//
////////////////////////////////////////////////////////////////////////////

//! system time handler for the 16bit timer OCIA irq
/*! this is the pulse of the system.
    task switcher and motor driver calls are initiated here.
*/
extern void systime_handler(void);    /* system interrupt handler running on OCRB */

__asm__("
.text
.align 1
.global _systime_handler
.global _systime_tm_return
_systime_handler:
               ; r6 saved by ROM

                push r0                         ; both motors & task
                                                ; switcher need this reg.

                ; increment system timer

                mov.w @_sys_time+2,r6           ; LSW -> r6
                add.b #0x1,r6l                  ; 16 bit: add 1
                addx  #0x0,r6h
                mov.w r6,@_sys_time+2
                bcc sys_nohigh                  ; speedup for 65535 cases

                  mov.w @_sys_time,r6           ; MSW -> r6
                  add.b #0x1,r6l
                  addx  #0x0,r6h
                  mov.w r6,@_sys_time
              sys_nohigh:
        "
#ifdef CONF_DMOTOR
        "
                ;this is the original call – we don't use it here
                ;jsr _dm_handler                 ; call motor driver
"
```

```
#endif

#ifdef CONF_DSOUND
        "
                jsr _dsound_handler             ; call sound handler
        "
#endif

#ifdef CONF_LNP
        "
                mov.w @_lnp_timeout_counter,r6  ; check LNP timeout counter
                subs #0x1,r6
                mov.w r6,r6                     ; subs doesn't change flags!
                bne sys_noreset

                  jsr _lnp_integrity_reset
                  mov.w @_lnp_timeout,r6        ; reset timeout

              sys_noreset:
                mov.w r6,@_lnp_timeout_counter
        "
#endif


#ifdef CONF_DKEY
        "
                jsr _dkey_handler
        "
#endif

#ifndef CONF_TM
#ifdef CONF_BATTERY_INDICATOR
        "
                mov.w @_battery_refresh_counter,r6
                subs #0x1,r6
                bne batt_norefresh

                  jsr _battery_refresh
                  mov.w @_battery_refresh_period,r6

              batt_norefresh:
                mov.w r6,@_battery_refresh_counter
        "
#endif
#endif

#ifdef CONF_AUTOSHUTOFF
        "
                mov.w @_auto_shutoff_counter,r6
                subs  #0x1,r6
                bne auto_notshutoff

                  jsr _autoshutoff_check
                  mov.w @_auto_shutoff_period,r6

              auto_notshutoff:
                  mov.w r6,@_auto_shutoff_counter
        "
#endif

#ifdef CONF_VIS
        "
```

```
                mov.b @_vis_refresh_counter,r6l
                dec r6l
                bne vis_norefresh

                  jsr _vis_handler
                  mov.b @_vis_refresh_period,r6l

              vis_norefresh:
                mov.b r6l,@_vis_refresh_counter
        "
#endif

#ifdef CONF_LCD_REFRESH
        "
                mov.b @_lcd_refresh_counter,r6l
                dec r6l
                bne lcd_norefresh

                jsr _lcd_refresh_next_byte
                mov.b @_lcd_refresh_period,r6l

              lcd_norefresh:
                mov.b r6l,@_lcd_refresh_counter
        "
#endif

#ifdef CONF_TM
        "
                mov.b @_tm_current_slice,r6l
                dec r6l
                bne sys_noswitch                ; timeslice elapsed?

                  mov.w @_tm_switcher_vector,r6
                  jsr @r6                       ; call task switcher

              _systime_tm_return:
                mov.b @_tm_timeslice,r6l        ; new timeslice

              sys_noswitch:
                mov.b r6l,@_tm_current_slice
        "
#endif
        "
                bset    #4,@0x97:8              ; select OCRB
                mov.b   @0x92:8,r0h
                mov.b   @0x93:8,r0l
                push    r1
                mov.w   #500,r1                 ; 1ms system interrupt
                add     r1,r0
                mov.b   r0h,@0x94:8
                mov.b   r0l,@0x95:8
                pop     r1
                pop     r0
                bclr    #2,@0x91:8              ; reset compare B IRQ flag
                rts
        "
);

extern void motor_handler(void);      /* the new motor handler running on OCRA */

__asm__("
```

```
.text
.align 1
.global _motor_handler
_motor_handler:
               ; r6 saved by ROM

               push    r0
               push    r1
               jsr _dm_handler                     ; call motor driver
               bclr    #4,@0x97:8                  ; select OCRA
               mov.b   @0x92:8,r0h
               mov.b   @0x93:8,r0l
               mov.w   #500,r1                     ; add the value for 1ms
               add.w   r1,r0                       ; to the FRC value
               mov.b   r0h,@0x94:8
               mov.b   r0l,@0x95:8
               pop     r1
               pop     r0
               bclr    #3,@0x91:8                  ; reset compare A IRQ flag
               rts
        "
);

//! initialize system timer
/*! task switcher initialized to empty handler
    motors turned off
*/
void systime_init(void) {
  systime_shutdown();                          // shutdown hardware

  sys_time=0l;                                 // init timer

#ifdef CONF_TM
  tm_current_slice=tm_timeslice=TM_DEFAULT_SLICE;
  tm_switcher_vector=&rom_dummy_handler;       // empty handler
#endif

#ifdef CONF_DMOTOR
  dm_shutdown();
#endif
```

```
  // configure 16-bit timer compare A IRQ
  // to occur every 1 ms, hook and enable it.
  //
  T_CSR =TCSR_OCA | TCSR_OCB; // | TCSR_RESET_ON_A;
  T_CR  =TCR_CLOCK_32;
  T_OCR&=~TOCR_OCRB;
  T_OCRA=500;
  T_OCR|=TOCR_OCRB;
  T_OCRB=500;

  ocia_vector=&motor_handler;
  T_IER|=TIER_ENABLE_OCA;
  ocib_vector=&systime_handler;
  T_IER|=TIER_ENABLE_OCB;

}

//! shutdown system timer
/*! will also stop task switching and motors.
*/
void systime_shutdown(void) {
  T_IER&=~TIER_ENABLE_OCB;    // unhook compare B IRQ
  T_IER&=~TIER_ENABLE_OCA;    // unhook compare A IRQ
}

#ifdef CONF_TM
//! set task switcher vector
/*! \param switcher the switcher
*/
void systime_set_switcher(void* switcher) {
  tm_switcher_vector=switcher;
}

//! set multitasking timeslice in ms
/*! \param slice the timeslice. must be at least 5ms.
*/
void systime_set_timeslice(unsigned char slice) {
  if(slice>5) {                    // some minimum value
    tm_timeslice=slice;
    if(tm_current_slice>tm_timeslice)
      tm_current_slice=tm_timeslice;
  }
}

#endif
#endif // CONF_TIME
```