10

legOS[1]

legOS ist die mächtigste Programmierumgebung für den RCX. Kein anderes System setzt so tief auf der Hardware auf und bietet doch so vielfältige Möglichkeiten. legOS übertrifft die Fähigkeiten anderer Umgebungen in fast allen Bereichen. Einige wichtige Funktionen, wie die Synchronisation von Prozessen und leistungsfähige Vernetzung mehrerer Geräte, stehen sogar ausschließlich unter legOS zur Verfügung.

In diesem Kapitel beschreibe ich zunächst die Entwicklungsumgebung für legOS und Anwendungsprogramme. Dann erkläre ich die Systemaufrufe und zeige anhand von Beispielen, wie sie verwendet werden. Anschließend bespreche ich detailliert ausgewählte Bereiche des Systems.

Natürlich sind mächtige Systeme nicht einfach. Wer sich bereits mit Linux und den Werkzeugen des GNU-Projekts auskennt, wird wenig Probleme haben, sich in legOS zurechtzufinden. Alle anderen sollten sich darauf einstellen, etwas Zeit mit der Einrichtung der Entwicklungsumgebung und dem Erlernen des Systems zu verbringen.

Programme unter legOS werden in C oder C++ geschrieben. Sie zählen zu den gängigsten Programmiersprachen überhaupt und sind unter nahezu jedem Betriebssystem verfügbar. Wer mit ihnen nicht vertraut ist, sollte zuvor eine der vielen Einführungen zur Hand nehmen. Falls Sie überhaupt keine Erfahrung im Programmieren haben, würde ich Ihnen sogar empfehlen, legOS beiseite zu legen und sich erst einmal NQC anzusehen.

---

1 Dieses Kapitel wurde für die deutsche Ausgabe dieses Buchs von Markus L. Noga, dem Entwickler von legOS, neugeschrieben. Behandelt wird die bei Drucklegung aktuelle Version 0.2.4.

Was ist legOS?

legOS ist ein Betriebssystem für den RCX, das sich weitgehend an Unix und POSIX orientiert. Wie pbFORTH ersetzt es die LEGO-Firmware vollständig. Anwendungen für legOS können in C, C++ und Assembler geschrieben werden. Nach ihrer Übersetzung in Maschinencode auf einem PC werden sie an den RCX übertragen und dort direkt auf dem Prozessor ausgeführt.

Abbildung 10-1 zeigt die Architektur von legOS und seiner Entwicklungsumgebung im Überblick. Der legOS-Kernel wird mit einem speziellen Übersetzer auf dem PC erstellt und mittels eines Firmware-Laders auf dem RCX installiert. Dazu kann zum Beispiel der beigefügte `firmdl3` oder auch `nqc` verwendet werden. Eigene Programme erstellt man in derselben Entwicklungsumgebung, ihre Übertragung geschieht dann mit dem legOS-Programmlader `dll`.

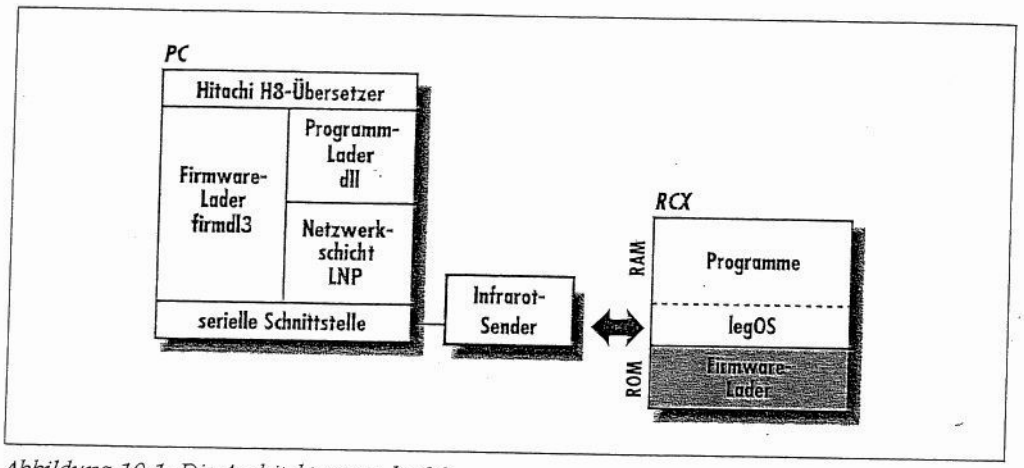

Abbildung 10-1: Die Architektur von legOS

Umfang und Komplexität einer legOS-Anwendung werden nur durch den Hauptspeicher und die Rechenleistung des RCX beschränkt. Es gibt keine Obergrenze von 32 Variablen wie in RCX Code und NQC. Anders als in NQC steht auch der volle Sprachumfang von C zur Verfügung: Unter legOS sind Pointer, Arrays und Strukturen kein Problem. Mechanismen von C++ wie Klassen und Vererbung können ebenso eingesetzt werden.

Die Programmierung unter legOS ähnelt der eines normalen Unix-Systems. Jeder, der bereits unter Linux ein C-Programm geschrieben hat, wird sich sehr schnell zu Hause fühlen. Anwendungen können auf eine Vielzahl von Kernelaufrufen zugreifen, darunter dynamische Speicherverwaltung, Multithreading und Synchronisation, außerdem auf eine Netzwerkschicht für den Infrarot-Kanal sowie auf Treiber für alle übrigen Geräte des RCX. Weitere häufig benötigte Funktionen wie Fließkomma-Arithmetik oder die Erzeugung von Zufallszahlen werden in Bibliotheken zur Verfügung gestellt.

Ist mehr als ein RCX vorhanden, so führt an legOS kaum ein Weg vorbei. RCX Code und NQC können lediglich ein Byte lange Nachrichten über Infrarot verschicken. Schlimmer noch, ein RCX ist nicht in der Lage, von sich aus einen PC anzusprechen. Das *legOS Networking Protocol*, ein schnelles und flexibles Paketnetzwerk für mehrere Geräte, behebt alle diese Probleme und stellt sogar auf jedem Gerät mehrere Zugangsports bereit.

Die Entwicklungsumgebung

Der RCX ist ein kleiner Rechner mit wenig Speicher und beschränkten Ein-/Ausgabemöglichkeiten. Daher ist es nicht möglich, legOS oder seine Anwendungen direkt auf dem RCX zu übersetzen – ein brauchbarer Übersetzer für C würde schlicht zu viel Speicher benötigen, ganz zu schweigen von dem Problem, ein Programm direkt auf dem RCX einzugeben und zu bearbeiten.

Dieses Problem teilt die überwiegende Mehrheit aller eingebetteten Systeme. Zur Entwicklung von Programmen für Mikrocontroller ist man stets auf Unterstützung durch einen leistungsstärkeren Rechner, den sogenannten Host, angewiesen. Progamme werden auf dem Host bearbeitet und übersetzt. Zur Ausführung werden sie auf das eingebettete System übertragen – im Falle des RCX per Infrarotschnittstelle.

Um so zu arbeiten, ist ein spezieller Übersetzer notwendig, ein sogenannter Cross-Compiler. Diese Spielart läuft auf einem Rechner und erzeugt Programme für einen anderen Rechner und/oder ein anderes Betriebssystem – zum Beispiel ein Übersetzer, der auf x86-Rechnern unter Windows läuft und Code für PowerMac G4 unter MacOS generiert. Wie sich früh herausstellte, wird der im RCX eingesetzte Hitachi H8-Prozessor durch den in der Unix-Welt weit verbreiteten GNU C-Compiler `gcc` unterstützt. Ich empfehle die Verwendung der Version `egcs-1.1.2` des `gcc`.

Der Übersetzer selbst benötigt einige Hilfsprogramme, hauptsächlich Assembler und Linker für die Zielplattform. Diese sind im Paket GNU `binutils` zusammengefaßt. Um in Projekten sinnvoll zu arbeiten, ist außerdem GNU `make` unerläßlich (siehe auch *Managing Projects with make* von Andrew Oram und Steve Talbott, erschienen bei O'Reilly). Auf Unix-Rechnern ist `make` meist bereits vorhanden.

Linux

Für Linux ist auf der legOS-Website (siehe Abschnitt »Online-Ressourcen«) im Bereich Files ein fertig übersetztes Paket mit `egcs-1.1.2` und `binutils` verfügbar. Es wurde unter RedHat 6.1 und SuSE 6.3 getestet und sollte auf jedem glibc-System lauffähig sein. Das Paket ist 1,7 MB groß und wird wie folgt installiert:

```
su
cd /
tar -xzf ~benutzername/rcx-tools-glibc.tar.gz
exit
```

Auf manchen Systemen ist es eventuell notwendig, */usr/local/lib* zum Suchpfad des dynamischen Linkers hinzuzufügen:[2]

```
su
vi /etc/ld.so.conf
ldconfig
exit
```

legOS selbst kann nach Belieben von jedem Benutzer unter jedem Verzeichnis installiert werden. Zur Bequemlichkeit sollte *legOS/util* dem Suchpfad `$PATH` hinzugefügt werden. Ist der Infrarotsender nicht an den ersten seriellen Port angeschlossen, ist es zusätzlich notwendig, den entsprechenden Port in `$RCXTTY` einzustellen.

Andere Unix-Systeme

Wer andere Unix-Varianten benutzt oder seine Software unter Linux gerne selbst übersetzt, muß sich die Quellen zu `binutils-2.9.1` und `egcs-1.1.2` aus dem Netz herunterladen. Zusätzlich sollte der Patch für `egcs-1.1.2` aus dem Files-Bereich der legOS-Website eingespielt werden.

Zunächst übersetzt und installiert man `binutils-2.9.1`:

```
cd binutils-2.9.1/
./configure --target=hitachi-h8300-hms --prefix=/usr/local LANGUAGES="C C++"
make all
su
make install
exit
```

Für `egcs-1.1.2` benötigt man ein temporäres Verzeichnis:

```
mkdir egcs
cd egcs/
../egcs-1.1.2/configure --target=hitachi-h8300-hms
                        --prefix=/usr/local LANGUAGES="C C++"
make cross
su
make install
exit
```

Die weiteren Schritte laufen wie unter Linux ab.

Windows

Für Windows-Benutzer liegt seit kurzem Winlegos von Rossz Vámos-Wentworth vor. Diese 5,5 MB große Distribution vereint legOS und seine Entwicklungsumgebung in einem praktischen InstallShield. Winlegos ist selbstentpackend und bietet eine dialoggeführte Installation im Verzeichnis Ihrer Wahl. Sie finden Winlegos sowie Hinweise zu anderen Distributionen für Windows auf den am Ende des Kapitels aufgeführten Webseiten.

2 Nein, ich benutze nicht ausschließlich vi. Aber vi ist überall verfügbar.

Den Übersetzer unter Windows selbst zu erstellen setzt einiges an Erfahrung voraus. Ich rate normalerweise davon ab. Wer es dennoch versuchen möchte, wird Cygwin benötigen, ein hervorragendes Paket von Cygnus Solutions, das Unix-Shells, Hilfsprogramme und Übersetzer für Win32-Systeme zur Verfügung stellt. Die aktuelle Version umfaßt 13 MB und wird jedem unter Windows gestrandeten Unix-Fan viel Freude bereiten. Der entsprechende URL befindet sich ebenfalls am Ende des Kapitels.

Erste Schritte

Ist auf dem Host alles richtig installiert, kann es losgehen!

```
make
firmdl3 kernel/legOS.srec
dll demo/helloworld.lx
```

Was bewirken diese Zeilen? `make` übersetzt den Kernel, die Demos und die notwendigen Hilfsprogramme. `firmdl3`[3] installiert legOS auf dem RCX. Und `dll`, der dynamische Lader und Linker, überträgt Anwenderprogramme an einen legOS-Rechner. Die Dateiendung .lx steht übrigens für *legOS executable*.

Es ist nicht möglich, zwei verschiedene Firmwares auf einmal zu verwenden, daher können die LEGO Software, legOS und pbForth nur abwechselnd installiert werden. Anders sieht es mit Anwendungen aus – mit `dll` können mehrere von ihnen geladen, ausgeführt und wieder gelöscht werden.

Wenn alles funktioniert, sollten nach einem Druck auf die rote Run-Taste auf dem Display des RCX nacheinander »Hello« und »World« erscheinen.

Programme und Tasten

Im Demo-Verzeichnis liegen noch weitere Programme. Diese können einfach mit `dll` geladen werden, wenn legOS schon auf dem RCX installiert ist. Der Schalter `-px` legt fest, welcher Programmplatz auf dem RCX angesprochen wird – die Voreinstellung ist `-p0`.

`dll demo/robots.lx` überschreibt also die erste Demo, und `dll -p1 demo/robots.lx` lädt sie auf einen neuen Platz. Auf dem RCX können die geladenen Programme mit der schwarzen Prgm-Taste ausgewählt werden. Das aktuelle Programm wird außen rechts auf dem Display angezeigt. Ist kein Programm geladen, steht dort ein Strich. Die Run-Taste startet und stoppt das aktuelle Programm.

legOS schaltet den RCX automatisch in einen Energiesparmodus, wenn wenig Leistung benötigt wird. Je höher die Prozessorauslastung des RCX ist, desto schneller bewegt sich die Figur auf dem Display – für die Demos und das System an sich läuft sie extrem

3 `firmdl3` überträgt normalerweise mit vierfacher Geschwindigkeit, da Firmware im Vergleich zu Anwenderprogrammen recht umfangreich ist. Wenn der Infrarotsender auf große Reichweite eingestellt oder der RCX weit entfernt ist, ist dies unmöglich. Notfalls läßt sich die Geschwindigkeit mit dem Schalter `--slow` drosseln.

langsam. Mit der grünen On-Off-Taste kann man den RCX ganz ausschalten. Laufende Programme werden dann abgebrochen.

Sollte ein Programm oder ein Teil von legOS so fehlerhaft oder böswillig sein, daß der legOS-Kernel überschrieben wird, kann es vorkommen, daß der RCX auf keine Tastendrücke mehr reagiert. Dagegen gibt es keinen Schutz, denn die H8-Prozessoren kennen keinen »Protected Mode« wie die Intel x86-Reihe.

In diesem Fall hilft nur eins: die Batterien entfernen und einige Minuten abwarten. Der Arbeitsspeicher wird schneller gelöscht, wenn man dabei alle Tasten gedrückt hält.

Ein solcher Absturz ist lästig, aber nicht gefährlich – der Firmware-Lader des RCX befindet sich im ROM und kann nicht überschrieben werden. Man kann stets neue Firmware laden, sei es ein neues legOS oder die Original-Software. legOS läßt sich übrigens auch auf friedlichem Weg entfernen, indem man On-Off und Prgm gleichzeitig drückt.

Ein eigenes Programm

Genug Demos, wie schreibt man ein Programm für legOS?

Am besten legt man ein eigenes Verzeichnis an, zum Beispiel *legOS/myproj*, und kopiert das Makefile aus *legOS/demo* dorthin. Dann schreibt man beispielsweise folgendes in *myproj/test.c*:

```
#include <conio.h>

int main(int argc, char **argv) {
  cputs("Hello");
  sleep(1);
  cputs("again");
  sleep(1);

  return 0;
}
```

Im Verzeichnis *myproj* übersetzt man das Programm mit `make test.lx` und überträgt es mit `dll test.lx` an den RCX.

Systemaufrufe

In diesem Abschnitt werden die Systemaufrufe von legOS beschrieben. An einigen Stellen beleuchte ich auch die Vorgänge unter der Haube näher. Weitere Informationen zu internen Bestandteilen des Systems finden Sie auch im Abschnitt »Programme«. Und natürlich im Quelltext, denn legOS ist Open Source.[4]

4 Ein Gedanke hinter Open Source ist dieser: Die maßgebliche Beschreibung des Quelltextes ist der Quelltext. Das ist ein Grund, warum viele Programmierer Open Source so lieben. Böse Zungen behaupten, sie würden es nur hassen, Dokumentationen zu schreiben.

Die legOS-Header sind gut dokumentiert, und sie liefern die gewünschte Information schneller als jedes Buch, wenn man das System bereits kennt. (Ein Editor sollte in der Lage sein, auf Tastendruck in die entsprechenden Header zu springen.)

Zum Einstieg ist es aber zweckmäßiger, Module, Aufrufe und Schnittstellen in linear geordneter Form vorzustellen. Für legOS Version 0.2.4 geschieht das hier zum erstenmal.

Anzeige

legOS bietet sehr viele Möglichkeiten zur Ansteuerung des LC-Displays. Fünf Ziffern und einige Symbole bieten natürlich nicht denselben Spielraum wie der Bildschirm eines normalen PCs, können aber wertvolle Hinweise über den Ablauf eines Programms liefern. Sei es die Darstellung von Sensordaten, Programmzuständen oder Fehlercodes – das Display ist unentbehrlich.

Der Kernel selbst verwendet das laufende Männchen und die Sensorindikatoren zur Zustandsanzeige. Die Motorindikatoren sind als reserviert zu betrachten. Alle anderen Symbole und die Ziffern stehen dem Benutzer zur freien Verfügung.

Text und Hexadezimalzahlen (conio.h)

Auf einer Siebensegmentanzeige kann ASCII-Text natürlich nur näherungsweise dargestellt werden. Die verwendete Schriftart stellt letztlich immer einen Kompromiß dar – aber einen, der gut genug ist für »Start«, »Stop«, »Los«, »Halt« usw.

*void cputs(const char *s)*

Gibt die Annäherung des ASCII-Strings *s* von links nach rechts auf dem Display aus. Strings, die fünf Buchstaben überschreiten, werden entsprechend abgeschnitten, ein Scrolling erfolgt nicht. Kürzere Strings werden in der Darstellung um die notwendige Anzahl Leerzeichen erweitert.

void cputw(unsigned w)

Zeigt das Wort *w* in Hexadezimaldarstellung auf den linken vier Segmenten des Displays an.

void cls(void)

Löscht den Benutzeranteil des Displays.

void cputc(char c, int pos)
void cputc_hex(char c, int pos)
void cputc_native(char c, int pos)

Gibt ein einzelnes Zeichen *c* an der gewünschten Position *pos* aus. Auf dem Display werden Positionen von rechts nach links gezählt: 0 entspricht der Ziffer rechts vom Männchen, 1 der Ziffer links vom Männchen usw. Das Zeichen *c* wird entsprechend der aufgerufenen Funktion als ASCII-Symbol, als Hexadezimalziffer von 0 bis 15 oder als Segmentmaske interpretiert. In Segmentmasken stehen gesetzte Bits für schwarze Segmente. Bit 0 steuert den mittleren Strich an, die weiteren Segmente sind gegen den Uhrzeigersinn numeriert.

Dezimalzahlen und ROM (rom/lcd.h)

Im ROM des RCX ist eine Funktion zur Darstellung von Dezimalzahlen vorhanden. Diese wird auch von der LEGO-Firmware benutzt. Für die gängigsten Verwendungen der Funktion stellt legOS kurze Makros zur Verfügung, sie direkt aufzurufen ist natürlich auch möglich.

lcd_int(i)
: Gibt eine vorzeichenbehaftete Dezimalzahl aus. Die Anzeige arbeitet mit Sättigung, Werte über 9999 erscheinen als 9999.

lcd_unsigned(u)
: Gibt eine vorzeichenlose Dezimalzahl aus. Führende Stellen werden bei Bedarf mit Nullen aufgefüllt – zum Beispiel erscheint 123 als 0123.

lcd_clock(i)
: Gibt eine vorzeichenlose Dezimalzahl mit Punkt zwischen der Zehner- und Hunderterstelle aus – zum Beispiel wird 1234 als 12.34 dargestellt.

void lcd_number(int i, lcd_number_style n, lcd_comma_style c)
: Die eigentliche Ausgaberoutine im ROM. Mögliche Werte für *n* sind `sign`, `unsign` oder `digit`, um vorzeichenbehaftete und vorzeichenlose Zahlen auszugeben bzw. die einzelne Ziffer rechts außen anzusteuern. *c* legt die Position des Dezimalpunkts fest: Er kann fehlen (`e0`), Zehntel (`e_1`), Hundertstel (`e_2`) oder Tausendstel (`e_3`) anzeigen. Zusammen mit `digit` kann ausschließlich `digit_comma` verwendet werden.

void lcd_show(lcd_segment segment)
void lcd_hide(lcd_segment segment)
: Führt eine Anzeigeoperation aus. Die Operationen sind manchmal nützlich, zum Beispiel zeigt `lcd_show(man_run)` das laufende Männchen an. Manchmal verwenden sie interne Zustände und sind verwirrend. Im allgemeinen wird von der Verwendung abgeraten.

void lcd_clear(void)
: Löscht das gesamte Display. Von der Verwendung wird abgeraten, da auch vom Kernel genutzte Bereiche gelöscht werden.

Symbole (dlcd.h)

Dies sind die Grundbausteine der Anzeige unter legOS – sie setzen oder löschen jeweils genau ein Bit. Die ASCII- und Hexadezimal-Funktionen verwenden diese Bausteine. Zur Darstellung von Symbolen werden sie auch direkt aufgerufen. Die Liste der Symbole entnimmt man dem Header.

dlcd_show(s)
: Zeigt das Symbol *s*. Zum Beispiel wird mit `dlcd_show(LCD_IR_UPPER)` der obere Teil des IR-Symbols angezeigt.

dlcd_hide(s)
Löscht das Symbol *s*.

Eingabe (dkey.h)

Die vier Tasten des RCX bieten nur wenig Spielraum. Der legOS-Kernel verwendet Run zum Starten und Stoppen von Programmen sowie On-Off zum Ein- und Ausschalten des RCX.

Ein Benutzerprogramm kann sinnvollerweise nur View und Prgm abfragen. Der ältere Header *dbutton.h* liefert nichtentprellte Eingangssignale, die nur in Ausnahmefällen nützlich sind.

KEY_ONOFF
KEY_RUN
KEY_VIEW
KEY_PRGM
Konstanten für die Tasten des RCX.

volatile unsigned char dkey
Die momentan aktive Taste oder 0. Hier ist maximal eine Taste zulässig. Werden mehrere Tasten gedrückt gehalten, so werden die vorangehenden verworfen.

volatile unsigned char dkey_multi
Momentan gedrückte Tasten. Dies ist ein Bitmuster; Tastenkombinationen sind zulässig. Die Abfrage sollte im Stil von `if(dkey_multi & KEY_VIEW) { ... }` erfolgen.

int getchar(void)
Wartet auf einen Tastendruck und liefert dessen Wert zurück.

Für gezielteres Warten stehen `dkey_pressed` und `dkey_released` zur Verfügung, die mit `wait_event` eingesetzt werden können.

Töne (dsound.h)

Der Lautsprecher des RCX wird über Pulswellenmodulation angesteuert. Theoretisch ist es sogar möglich, Samples auf diesem Kanal abzuspielen. Nur leider verbrauchen diese viel zuviel Speicher, und die Ansteuerelektronik des Lautsprechers ist auch nicht gut darauf ausgelegt. Im Gegensatz zu früheren Versionen unterstützt legOS daher nur noch das einstimmige Abspielen von Noten.

Eine Note ist ein Paar aus Tonhöhe und -länge, wie zum Beispiel `{PITCH_A0, 3}`. Diese Kombination hält das tiefste A, das wiedergegeben werden kann, für eine Dauer von 3 Zeiteinheiten[5]. `{PITCH_Gs1, 1}` steht für ein Gis bzw. As eine Oktave höher, das nur eine Einheit lang gehalten wird.

5 Die Dauer einer Zeiteinheit in Millisekunden ist frei einstellbar, daher ist der Name der Einheit ziemlich belanglos. Im Quellcode ist durchgängig von 1/16-Noten die Rede.

Pausen werden vom System als Spezialfall von Noten behandelt – sie haben die Frequenz `PITCH_PAUSE`. Tonfolgen[6] sind Arrays von Noten. Als Endmarke wird dabei `{PITCH_END, 0}` verwendet.

*void dsound_play(const note_t *notes)*
: Startet das Abspielen der Tonfolge *notes* im Hintergrund. Bereits laufende Abspielvorgänge werden abgebrochen.

void dsound_system(unsigned nr)
: Spielt einen Systemklang im Hintergrund ab. Ein Systemklang ist einfach eine Tonfolge, die durch das System definiert ist. Diese ändern sich ständig, einzig `DSOUND_BEEP` ist immer zuverlässig anzutreffen.

void dsound_stop(void)
: Bricht das Abspielen der laufenden Tonfolge ab.

int dsound_playing(void)
: Liefert 0 zurück, wenn keine Tonfolge mehr abgespielt wird. Diese Funktion ist praktisch, um nebenbei andere Aufgaben zu erledigen. Wenn tatenlos auf das Ende einer Tonfolge gewartet wird, sollte besser `wait_event(dsound_finished,0)` eingesetzt werden.

void dsound_set_duration(unsigned duration)
: Stellt die Dauer einer Zeiteinheit in Millisekunden ein.

void dsound_set_internote(unsigned duration)
: Stellt die Länge der Pause zwischen zwei Noten in Millisekunden ein. Wird dieser Wert auf 0 gesetzt, werden aufeinanderfolgende Noten gebunden gespielt. Dieser Wert darf die Länge einer Zeiteinheit nicht überschreiten.

Das folgende Programm spielt die ersten vier Takte von »Alle meine Entchen«:

```
#include <dsound.h>

static const note_t entchen[] = {
  { PITCH_C4,  1 } , { PITCH_D4,  1 } , { PITCH_E4,  1 }, { PITCH_F4,  1 } ,
  { PITCH_G4,  2 } ,                    { PITCH_G4,  2 },
  { PITCH_A4,  1 } , { PITCH_A4,  1 } , { PITCH_A4,  1 }, { PITCH_A4,  1 } ,
  { PITCH_G4,  4 } , { PITCH_END, 0 }
};

int main(int argc, char *argv[]) {
  dsound_play(entchen);
  wait_event(dsound_finished,0);
  return 0;
}
```

6 Nicht jede Tonfolge ist automatisch eine Melodie. Die meisten sind bestenfalls Geräusche.

Messen (dsensor.h)

Der RCX kann eine Vielzahl externer Sensoren betreiben und auslesen. Er verfügt auch über einen inneren Sensor, der die Batteriespannung überwacht und anderen Systemen nicht ohne weiteres zugänglich ist.

Sensoren können aktiv oder passiv betrieben werden. Im passiven Betrieb mißt der RCX den Widerstand eines Sensors – beispielsweise, um den Zustand eines Berührungssensors zu bestimmen. Im aktiven Betrieb wird zusätzlich eine gepulste Versorgungsspannung auf den Sensor gelegt. Die Rotationssensoren benötigen diese Spannung für ihre optischen Encoder. Sie funktionieren nur im aktiven Betrieb.

Ein aktiv betriebener Lichtsensor strahlt Licht ab und mißt den reflektierten Anteil. So kann man einer Linie auf dem Boden folgen. Ein passiv betriebener Lichtsensor leuchtet nicht und mißt die Umgebungsbeleuchtung. Er macht nicht auf sich aufmerksam und kann genauer messen, vielleicht um eine Kerze ausfindig zu machen.

Abgesehen vom Rotationssensor, der schnelle Vorverarbeitung benötigt, unternimmt legOS keinen Versuch, Meßwerte zu skalieren oder umzurechnen. Der gesamte Meßbereich der A/D-Wandler steht dem Anwender zur Verfügung. Das bedeutet allerdings auch, daß er die Meßwerte selbst interpretieren muß – am besten mit Hilfe der vordefinierten Makros.

SENSOR_1
SENSOR_2
SENSOR_3
BATTERY

Diese Makros liefern den momentanen Meßwert von Sensor 1 bis 3 bzw. der Batterieüberwachung zurück. Ihr Wertebereich liegt bei 0x0000 - 0xffff, wobei nur die höherwertigen zehn Bit definiert sind.

Sensoren benutzen üblicherweise nicht den vollen Wertebereich der A/D-Wandler.

*void ds_active(volatile unsigned * sensor)*
*void ds_passive(volatile unsigned * sensor)*

Diese Funktionen schalten einen Sensor auf aktiven oder passiven Betrieb. Standard ist der passive Betrieb. *sensor* ist dabei ein Zeiger auf einen Sensor. Diese Zeile schaltet zum Beispiel den zweiten Sensor auf aktiven Betrieb:

```
ds_active(&SENSOR_2);
```

*void ds_rotation_on(volatile unsigned * sensor)*
*void ds_rotation_off(volatile unsigned * sensor)*

Diese Funktionen schalten die Verarbeitung von Rotationssignalen für einen Eingang an bzw. aus. *sensor* ist wieder ein Zeiger auf einen Sensor.

*void ds_rotation_set(volatile unsigned * sensor, int pos)*

Legt die absolute Position des Rotationssensors *sensor* (siehe oben) fest. Die Sensoren können nur relative Bewegung messen, ihre Ausgangsposition muß also vom Anwender festgelegt werden.

Einen Rotationssensor an Eingang 3 würde man so aktivieren:

```
ds_active(&SENSOR_3);
ds_rotation_set(&SENSOR_3,0);
ds_rotation_on(&SENSOR_3);
```

ROTATION_1
ROTATION_2
ROTATION_3

Diese Makros liefern die Position des entsprechenden Rotationssensors in 1/16-Umdrehungen.

LIGHT(x)
LIGHT_1
LIGHT_2
LIGHT_3

Skaliert den Meßwert eines Lichtsensors auf einen Bereich von ca. 0 bis 100, wobei 100 maximaler Helligkeit entspricht. `LIGHT_n` ist eine Kurzform von `LIGHT(SENSOR_n)`.

TOUCH(x)
TOUCH_1
TOUCH_2
TOUCH_3

Skaliert den Meßwert eines Berührungssensors auf 0 bis 1, wobei 1 gedrückt und 0 nicht gedrückt bedeutet. `TOUCH_n` ist eine Kurzform von `TOUCH(SENSOR_n)`.

Die Sensorfunktionen akzeptieren nur Zeiger auf SENSOR_n als Argumente. Der Versuch, Zeiger auf TOUCH_n, LIGHT_n oder ROTATION_n zu übergeben, führt zu einer Fehlermeldung oder anderen unerwünschten Effekten.

Steuern (dmotor.h)

Die Möglichkeit, seine Umgebung nicht nur über Buchstaben auf einem Schirm und Piepstöne zu beeinflussen, macht einen Großteil des Reizes des RCX aus. Obwohl man auch Lichter anschließen kann, werden die drei Ausgänge des RCX meistens mit Motoren verbunden. Ihre Ansteuerung erfolgt intern genau wie die Tonerzeugung durch Pulswellenmodulation.

legOS stellt für Motoren die auch aus NQC bekannten Drehrichtungen Vorwärtslauf, Rückwärtslauf, Leerlauf und Halten zur Verfügung. Nur ihre Namen unterscheiden sich; in legOS sind sie mit `fwd`, `rev`, `off` und `brake` benannt.

Die Drehgeschwindigkeit läßt sich in 256 Stufen von 0 bis 255 festlegen.[7] Es wird empfohlen, statt Zahlenwerten die vordefinierten Makros `MIN_SPEED` und `MAX_SPEED` sowie Bruchteile wie `MAX_SPEED/2` zu verwenden, denn Werte können sich immer ändern.

void motor_a_dir(MotorDirection dir)
void motor_b_dir(MotorDirection dir)
void motor_c_dir(MotorDirection dir)
Legt die Drehrichtung des jeweiligen Motors fest. Mögliche Richtungen sind `fwd`, `rev`, `off` und `brake`.

void motor_a_speed(unsigned char speed)
void motor_b_speed(unsigned char speed)
void motor_c_speed(unsigned char speed)
Stellt die Drehgeschwindigkeit des jeweiligen Motors zwischen `MIN_SPEED` und `MAX_SPEED` ein.

Dieses Programmstück dreht den Motor an Ausgang C schnell vorwärts:

```
motor_c_dir(fwd);
motor_c_speed(MAX_SPEED);
```

Speicher (stdlib.h)

Die C-Standardbibliothek enthält Funktionen zur Speicherverwaltung, die auch vom legOS-Kernel bereitgestellt werden. Mit ihnen kann ein laufendes Programm neue Speicherbereiche anfordern und wieder freigeben.

Auf dem RCX sind 32 KB Speicher vorhanden, von denen legOS ungefähr 6 KB belegt. Der Rest steht für eigene Programme und ihre dynamischen Daten zur Verfügung.

Speicherbereiche haben unter legOS grundsätzlich nur einen Thread als Besitzer. Wenn dieser stirbt, werden alle Speicherbereiche, die er angefordert hat, automatisch wieder freigegeben. Ein Spezialfall ist das Stack-Segment eines Threads, das vom Kernel bei der Thread-Erzeugung angelegt wird. Geladene Programme gehören dem Kernel. Ihre Speicherbereiche für Programmcode und Daten werden nur freigegeben, wenn `dll` das Programm löscht.

Unabhängig von Eigentumsfragen kann jeder Thread jede Speicheradresse lesen und beschreiben. Dies ist eine Eigenschaft der Prozessorfamilie – wie in so vielen eingebetteten Systemen gibt es schlicht keinerlei Schutzmechanismen. Es liegt am Programmierer, ordentlichen Code zu schreiben, der mit dem Betriebssystem und anderen Threads friedlich zusammenarbeitet.

7 PWM schaltet sehr schnell zwischen der gewählten Drehrichtung und einer weiteren um. Im Kernel ist einstellbar, ob es sich bei der zweiten Richtung um Leerlauf oder Bremsen handelt. Um Überraschungen zu vermeiden, sollten absichtlicher Leerlauf und absichtliches Bremsen immer mit MAX_SPEED erfolgen.

*void *malloc(size_t size)*

Reserviert einen Speicherbereich der Größe *size* für den Thread und liefert einen Zeiger darauf zurück oder 0, falls kein Block dieser Größe mehr vorhanden ist. Der Inhalt des reservierten Blocks ist undefiniert.

*void *calloc(size_t nmemb, size_t size)*

Reserviert einen Speicherbereich mit *nmemb* Elementen der Größe *size* für den Thread und liefert einen Zeiger darauf zurück oder 0, falls kein Block dieser Größe mehr vorhanden ist. Der Block wird vom System mit Nullen gefüllt.

*void free(void *ptr)*

Gibt einen mit `malloc` oder `calloc` reservierten Speicherbereich wieder frei. Alle Zeiger in den Speicherbereich sind danach ungültig.

Speicherverwaltung und Pointer gehören zu den schwierigsten Aspekten der Programmierung in C. Scheinbar unerklärliche Programmfehler und Abstürze sind häufig auf sorglosen Umgang mit der Speicherverwaltung zurückzuführen. Daher sollten angeforderte Speicherbereiche stets vor ihrer Verwendung überprüft werden – vielleicht war nicht ausreichend Speicher vorhanden. Besonders schwer zu finden sind allerdings Off-by-One-Fehler, bei denen der reservierte Bereich nur um einen Schritt verlassen wird. Ein typischer Vertreter ist `for(i=0; i<=max; i++) a[i]=...;`

Threads (unistd.h)

Umfangreiche Betriebssysteme auf Systemen mit Speicherschutzmechanismen unterscheiden meist zwischen Prozessen, die in eigenen Speicherbereichen arbeiten, und Threads, die sich zu mehreren denselben Speicherbereich teilen. Häufig werden Threads auch zu Thread-Gruppen oder Prozessen zusammengefaßt.

Diese Kategorien kennt legOS nicht. Seine Threads haben ähnliche Merkmale wie die auf Unix-Systemen bekannten POSIX Threads. Sie gleichen ebenfalls den aus Windows bekannten Win32-Threads.

Kurz gesagt, ist ein Thread unter legOS ein Registersatz mit eigenem Stack und dem Anspruch auf Rechenzeit. Die verfügbare Rechenzeit wird vom Kernel in kurze Abschnitte gegliedert und gleichmäßig unter den Threads der höchsten Priorität aufgeteilt. Threads niedrigerer Priorität werden erst ausgeführt, wenn diese beendet sind oder auf ein Ereignis warten.

Wer ruft Threads ins Leben? Zwei von ihnen werden automatisch durch den Kernel gestartet: der Programmverwalter und ein Energiespar-Thread. Laufende Programme haben mindestens einen Thread. Er wird vom Programmverwalter gestartet und führt die Funktion `main()` des Programms aus. Dort kann man beliebige weitere Threads starten.

Die Schlichtheit des Konzepts macht legOS gleich durch konfuse Namensgebung zunichte. Im Quellcode ist abwechselnd von Task, Prozeß und Thread die Rede. Zwar lehnen sich die Funktionen größtenteils an bekannte Aufrufe wie `kill` an, doch die Tücke steckt im Detail: zum Beispiel ersetzen die Aufrufe der exec-Familie unter Unix den laufenden Prozeß, unter legOS hingegen erzeugen sie einen weiteren Thread.

*pid_t execi(int (*code_start) (int, char **), int argc, char **argv, priority_t priority, size_t stack_size);*

Erzeugt einen neuen Thread, der sofort mit der Ausführung der Funktion *start* beginnt. Diese hat die Form `int start(int argc, char **argv)`. Die Argumente *argc* und *argv* werden direkt an sie übergeben.

Unter legOS ist die Priorität eines Threads während seiner Laufzeit konstant. *priority* sollte ein Wert zwischen `PRIO_LOWEST` und `PRIO_HIGHEST` zugewiesen werden, normalerweise `PRIO_NORMAL`.

Die Stack-Größe eines Threads ist ebenfalls während seiner Laufzeit konstant. Meist ist `DEFAULT_STACK_SIZE` ein ausreichender Wert für *stack_size*.

In C werden auf dem Stack Rücksprungadressen von Funktionsaufrufen und innerhalb von Funktionen definierte Variablen abgelegt. Ist die Stack-Größe eines Threads nicht ausreichend, können beliebige Speicherbereiche überschrieben werden. Dies ist eine der häufigsten Ursachen für einen Absturz des Programms oder des gesamten Systems.

void yield()

Weist den Kernel an, den Rest der aktuellen Zeitscheibe anderen Threads zur Verfügung zu stellen. Diese Funktion wird hauptsächlich im Betriebssystem eingesetzt.

void exit(int retval)

Beendet den laufenden Thread mit dem angegebenen Rückgabewert.

void kill(pid_t pid)

Beendet den Thread *pid*, als ob dieser `exit(-1)` aufgerufen hätte.

void killall(priority_t p)

Beendet alle Threads, deren Priorität kleiner als der angegebene Wert ist. Der laufende Thread wird nicht beendet, unabhängig von seiner Priorität. Diese Funktion wird hauptsächlich im Betriebssystem eingesetzt.

Ein Beispiel zu Threads wird im folgenden Abschnitt über Wartemechanismen gezeigt.

Warten (unistd.h)

Nicht alle Threads haben ständig wichtige Aktivitäten auszuführen. Viele sind darauf angewiesen, daß bestimmte Ereignisse eintreten, bevor sie mit ihrer Aufgabe fortfahren können, doch aktives Warten mit einer Schleife würde die Rechenzeit anderer Threads schmälern. Das ist nicht Sinn der Sache, es sei denn, der Thread erledigt im Rumpf der

Warteschleife sinnvolle Arbeit. legOS verfügt über einen flexiblen Mechanismus, um die Ausführung eines Threads bis zum Eintreten beliebiger Bedingungen anzuhalten.

Wie funktioniert das? Threads können beim Scheduler eine Weckfunktion anmelden und sich in den Wartezustand versetzen. Wann immer dem Thread nach dem Prioritätsschema eine Zeitscheibe zusteht, ruft der Scheduler jetzt die Funktion auf, um festzustellen, ob der Thread weiterlaufen möchte. Ist dies nicht der Fall, wird er einfach übergangen. Beansprucht kein Thread die Zeitscheibe, schaltet der Scheduler das System für ihre Dauer in den Energiesparmodus.

*wakeup_t wait_event(wakeup_t(*wakeup) (wakeup_t), wakeup_t data)*
Hält die Ausführung des laufenden Threads an, bis die Weckfunktion einen Wert ungleich 0 annimmt. Der Rückgabewert ist derjenige der Weckfunktion.

Viele Module von legOS stellen ihre eigenen Weckfunktionen bereit, unter anderem `dkey_pressed`, `dkey_released` und `dsound_finished`. Mit ihrer Hilfe kann man beispielsweise auf den Druck einer beliebigen Taste warten:

```
wait_event(dkey_pressed,KEY_ANY);
```

Das zweite Argument ist vom 32 Bit langen Datentyp `wakeup_t` und wird der Weckfunktion bei jedem Aufruf durch den Scheduler übergeben. Eigene Weckfunktionen können es nach Belieben auswerten. Das folgende Beispiel zeigt, wie man auf die Aktivierung eines Berührungssensors an Eingang 1 innerhalb eines bestimmten Zeitraums wartet:

```
wakeup_t sensor_timeout(wakeup_t data) {
  if(TOUCH_1)
    return 1;
  if(sys_time >= (time_t) data)
    return 2;
   return 0;
}
// ...
res=wait_event(sensor_timeout, sys_time+x);
```

Nach Möglichkeit sollten sich Weckfunktionen auf wenige Operationen beschränken, da sie auf dem Stack des Schedulers ausgeführt werden.

Häufig möchte man einfach die Ausführung eines Threads für einen bestimmten Zeitraum aussetzen, zum Beispiel um ein Fahrzeug für zwei Sekunden auf der Stelle zu drehen, da dies einer Drehung um ca. 180° entspricht. Dafür stehen zwei Funktionen zur Verfügung, die intern auf `wait_event` zurückgreifen:

unsigned int sleep(unsigned int sec)
Hält die Ausführung des Threads für *sec* Sekunden an.

unsigned int msleep(unsigned int msec)
Hält die Ausführung des Threads für *msec* Millisekunden an.

Beide Funktionen geben stets 0 zurück, um aufrufkompatibel zu den gleichnamigen Unix-Funktionen zu sein. Im folgenden Beispiel wird ein Kettenfahrzeug 1,5 Sekunden lang gedreht:

```
motor_a_dir(fwd)
motor_a_speed(MAX_SPEED);
motor_c_dir(rev);
motor_c_speed(MAX_SPEED);
sleep(1.5);
motor_a_speed(0);
motor_b_speed(0);
```

Das folgende Programm treibt ein Fahrzeug mit zwei Motoren an. Ein Hilfsthread sendet während der Fahrt in immer kürzeren Abständen einen Piepton aus. Er wird vom ersten Thread beendet, sobald das Fahrzeug auf ein Hindernis stößt. Als Beispiel für die Übergabe von Werten an einen Thread wird hier das Intervall zwischen den ersten beiden Tönen durchgereicht.

```
#include <unistd.h>
#include <dsound.h>
#include <dsensor.h>
#include <dmotor.h>

#define INTERVAL     1000

//! Wird vom zweiten Thread ausgeführt und piept immer schneller.
int beeper(int argc, char **argv) {
  int ms=argc;
  while(1) {
    dsound_system(DSOUND_BEEP);
    msleep(ms);
    ms=(15*ms)/16;
  }
}

//! Warte auf Berührung.
wakeup_t touch_wakeup(wakeup_t data) {
  volatile unsigned *sensor=(volatile unsigned *) ((unsigned)data);
  return TOUCH(sensor);
}

//! Hier startet das Programm.
int main(int argc, char **argv) {
  // Starte den zweiten Thread.
  pid_t beep_thread=execi(beeper, INTERVAL, (char**)0, PRIO_NORMAL,
                          DEFAULT_STACK_SIZE);

 motor_a_dir(fwd);
 motor_a_speed(MAX_SPEED/2);
 motor_c_dir(fwd);
 motor_c_speed(MAX_SPEED/2);

 wait_event(touch_wakeup, (wakeup_t) (&SENSOR_1));
```

```
    motor_a_dir(off);
    motor_c_dir(off);

    // Beende den zweiten Thread.
    kill(beep_thread);
    return 0;
}
```

Synchronisation (semaphore.h)

Threads sind erst im Team wirklich stark. Leider ist die Koordination mehrerer Threads keine einfache Aufgabe, da diese sich ständig gegenseitig unterbrechen können. Und wann genau der Scheduler von einem zum nächsten schaltet, ist nicht vorhersagbar. Vielleicht gerade dann, wenn ein Thread erst die Hälfte aller Zeiger auf ein Objekt verändert hat?

Zur Lösung dieses Problems gibt es viele Konzepte: Mutexe, Semaphoren, Kritische Abschnitte, Monitore, Read/Write-Locks und vieles mehr. Sobald eine elementare Sicherungsoperation gegeben ist, lassen sich alle anderen Varianten daraus aufbauen – beispielsweise sind Mutexe ein Spezialfall von Semaphoren.

legOS bietet zur Synchronisation von Threads Semaphoren nach POSIX an. Es handelt sich um Objekte mit einem Zähler, der erhöht und verringert werden kann. Ist der Zähler nicht positiv, müssen alle Threads warten, die ihn verringern wollen. Erreicht der Zähler wieder den Wert 1, wird genau ein wartender Thread aufgeweckt, um ihn zu verringern.

Alle Semaphoren-Funktionen liefern im Erfolgsfall 0 zurück.

*int sem_init(sem_t * sem, int pshared, unsigned int value)*
: Diese Funktion initialisiert den Semaphoren *sem* mit dem anfänglichen Zähler *value*. Das Argument *pshared* ist nur aus Kompatibilitätsgründen vorhanden. Sein Wert wird ignoriert und sollte auf 0 gesetzt werden.

*int sem_wait(sem_t * sem)*
: Hält den Thread an, bis der Zähler des Semaphoren positiv ist, und verringert ihn dann.

*int sem_trywait(sem_t * sem)*
: Sieht nach, ob der Zähler positiv ist. Wenn ja, wird er verringert, und die Funktion liefert sofort 0 zurück. Ansonsten gibt die Funktion sofort einen anderen Wert zurück. Der aktuelle Thread wird unter keinen Umständen angehalten. Diese Funktion kann in Interrupt-Handlern benutzt werden.

*int sem_post(sem_t * sem)*
: Erhöht den Zähler des Semaphoren um 1. Der aktuelle Thread wird unter keinen Umständen angehalten. Diese Funktion kann in Interrupt-Handlern benutzt werden.

*int sem_getvalue(sem_t * sem, int *sval)*

Liefert den aktuellen Zählerstand des Semaphoren *sem* in *sval* zurück. Vorsicht! Sofern sie nicht durch entsprechende Vorkehrungen abgehalten werden, können andere Threads den aktuellen Thread unterbrechen und den Zähler verändern, bevor das Ergebnis dieses Aufrufs ausgewertet wird.

*int sem_destroy(sem_t * sem)*

Zerstört einen Semaphoren. Die Auswirkungen auf Prozesse, die auf diesen Semaphoren warten, sind undefiniert.

Üblicherweise werden Semaphoren einer Datenstruktur oder einem Stück Hardware zugeordnet. So verwenden zum Beispiel alle Funktionen des Speicherverwalters denselben Semaphoren, um den Speicheraufbau nicht zu korrumpieren.

Die Anzahl der gleichzeitig möglichen Zugriffe entspricht dem Anfangswert des Semaphoren. Die häufig verwendeten Semaphoren mit Startwert 1 entsprechen genau den Mutexen – sie sorgen für exklusiven Zugriff. Mit ihnen sichert man kritische Zugriffe folgendermaßen ab:

```
int critical_operation(int critical_data) {
  int retval=0;
  sem_wait(&access_sem);
  {
    // Mittelteil, Operation ausführen
    // ...
  }
  sem_post(&access_sem);
  return retval;
}
```

Wichtig ist, daß kein Ausführungspfad aus der Funktion zurückkehrt, ohne den Semaphoren wieder freizugeben – ansonsten würde ein Deadlock entstehen, der das Programm zum Stillstand bringt. Im Mittelteil ist `return` also tabu. Statt dessen sollte an `retval` zugewiesen werden. Andere lokale Variablen sollten erst im Mittelteil vereinbart werden, damit ihre Initialisierung nicht außerhalb des gesicherten Bereichs erfolgt.

Infrarot-Netzwerk (lnp.h)

Der RCX kann Daten über die Infrarotschnittstelle austauschen – sei es mit einem PC oder einem weiteren RCX. Leider erlauben sowohl RCX Code als auch NQC nur Nachrichten von 1 Byte Länge, und das bei einer extrem niedrigen Übertragungsrate. Weder ist es möglich, den Absender einer Nachricht zu bestimmen, noch kann ein Empfänger gezielt angesprochen werden. Schlimmer noch, der RCX ist nicht in der Lage, von sich aus Kontakt mit einem PC aufzunehmen – jeder Austausch muß vom PC eingeleitet werden.

Das *legOS Networking Protocol*, kurz LNP, hebt alle diese Einschränkungen auf. Wie die Mehrzahl aller Netzwerkprotokolle ist LNP nach dem Schichtenprinzip aufgebaut (siehe Tabelle 10-1). Einer der vielen Vorteile des Schichtenmodells ist seine Portabilität: Abge-

sehen von der hardwareabhängigen Schicht 1, wird LNP für den PC aus demselben Quelltext erzeugt wie auf dem RCX. Für den Benutzer sind hauptsächlich Schicht 2 und 3 von Interesse.

Tabelle 10-1: Das Schichtenmodell des LNP

Schicht	Name	Beschreibung
3	`adressing`	Unterscheidung von Empfängern und ihren Ports
2	`integrity`	Pakete mit Blocksicherung
1	`logical`	Strom von Bytes
0	`physical`	Moduliertes Infrarot-Trägersignal

Benutzerprogramme können auf jeder Schicht Nachrichten verschicken und Handlerfunktionen beim Kernel anmelden. Trifft ein korrekt geformtes Paket ein, ruft der Kernel den dazugehörigen Handler auf. Dabei werden Pakete höherer Schichten an den Handlern der tieferen Schichten vorbeigeleitet, da sie diese nur als Transportmedium nutzen.

Schicht 2

Auf der Sicherungsschicht werden Pakete ohne Empfänger und Absender übertragen. Ihr Inhalt ist durch eine Prüfsumme gesichert. Wenn Probleme auftreten – sei es durch Störungen, überlappendes Senden mehrerer Parteien oder die begrenzte Reichweite von Sender und Empfänger –, werden Pakete stets als Ganzes verworfen.

Die Sicherungsschicht ist für zwei Anwendungsbereiche besonders interessant. Ist bekannt, daß ein Netz nur zwei Teilnehmer hat (üblicherweise einen RCX und einen PC), kann auf Adressierung verzichtet werden. Das verringert den Verwaltungsaufwand pro Paket um 40% und erhöht den Durchsatz an kleinen Paketen beträchtlich. Anwendungen, die den RCX nur als Meß- und Steuereinheit benutzen und alle Verarbeitung auf dem PC vornehmen, können davon profitieren.

Sind hingegen viele einander unbekannte Teilnehmer im Netz, wird es wichtig, Nachrichten durch sogenanntes *Broadcasting* an alle Teilnehmer zu versenden. Zum Beispiel ist es für mobile Anwendungen nützlich, fremden Geräten die eigene Adresse mitzuteilen.

*int lnp_integrity_write(const unsigned char *data,unsigned char length);*
: Verschickt ein Datenpaket über die Sicherungsschicht an alle Teilnehmer mit Ausnahme des Absenders. Konnte die Nachricht erfolgreich abgesandt werden, gibt die Funktion 0 zurück.

void lnp_integrity_set_handler(lnp_integrity_handler_t handler)
: Bestimmt die Behandlung von Paketen der Sicherungsschicht. Der Handler muß die Form `void handler(const unsigned char *data, unsigned char length)` haben. Voreingestellt ist `LNP_INTEGRITY_NONE`.

Das folgende Beispiel verwendet die Sicherungsschicht, um ungefähr alle fünf Sekunden die eigene Adresse zu übertragen. Empfangene Adressen werden auf dem Display angezeigt.

```
#include <lnp.h>
#include <conio.h>
#include <unistd.h>

#define MSG_HELLO       0               // falls weitere dazukommen

#define BASE            5000
#define RESOLV1         313
#define RESOLV2         500
#define INTERVAL        (BASE + ((RESOLV1*LNP_HOSTADDR)%RESOLV2))

//! Handler für eingehende Adressen
void i_handler(const unsigned char *data, unsigned char size) {
  if(size==2 && data[0]==MSG_HELLO)
    cputw( (unsigned) data[1] );
  else
    cputs("OTHER");
}

//! Anfang des Programms: Handler setzen, kontin. die eigene Adresse verschicken.
int main(int argc, char **argv) {
  static const unsigned char msg[]={ MSG_HELLO, LNP_HOSTADDR };

  lnp_integrity_set_handler(i_handler);

  while(1) {
    lnp_integrity_write(msg,2);
    msleep(INTERVAL);
  }
  return 0;
}
```

Die Berechnung der Konstante INTERVAL mag auf den ersten Blick unerklärlich erscheinen, sie geschieht aber aus einem wichtigen Grund. Wenn zwei Teilnehmer zur gleichen Zeit senden, überlagern sich ihre Pakete und werden unlesbar. Würden beide gleich lang warten, bevor sie erneut senden, träte das Problem immer wieder auf. Um INTERVAL für jeden Teilnehmer unterschiedlich zu gestalten, benutze ich die Tatsache, daß keine zwei Hostadressen gleich sind.

Schicht 3

Schicht 3 ermöglicht die Adressierung der Pakete an verschiedene Empfänger. Eine Empfängeradresse setzt sich aus Hostadresse und Port zusammen – so ist es möglich, verschiedene Funktionen eines einzelnen Geräts anzusprechen.

Der Adreßraum von LNP ist 1 Byte groß und kann über eine Hostmaske variabel eingeteilt werden. Voreingestellt ist die Maske 0xf0 – die oberen vier Bits einer Empfängeradresse werden als Hostadresse und die unteren vier Bits als Port aufgefaßt. Damit lassen sich 16 Geräte mit jeweils 16 Ports unterscheiden. Sind vier Teilnehmer mit je 64 Ports nötig, kann als Hostmaske 0xc0 eingestellt werden. Die Hostmaske 0xfe würde 128 Teilnehmer à 2 Ports vereinbaren.

Für den RCX werden Hostmaske und Hostadresse in *boot/config.h* eingestellt – normalerweise ist seine Adresse 0x00. Die Standardadresse des PCs von 0x80 kann man in *util/dll-src/config.h* ändern. Port 0 ist für den Programmlader reserviert, alle übrigen sind frei verfügbar.

*int lnp_addressing_write(const unsigned char *data,unsigned char length, unsigned char dest, unsigned char srcport);*

Verschickt ein Datenpaket über die Adressierungsschicht an einen Empfänger *dest*, der sich aus Hostadresse und Port zusammensetzt. Als Absender wird der Port *srcport* der eigenen Hostadresse ausgewiesen. Konnte die Nachricht erfolgreich abgesandt werden, gibt die Funktion 0 zurück.

void lnp_addressing_set_handler(unsigned char port, lnp_addressing_handler_t handler)

Bestimmt die Behandlung von Paketen der Adressierungsschicht für einen speziellen Port. Der Handler muß die Form `void handler(const unsigned char *data, unsigned char length, unsigned char src)` haben. Für alle freien Ports ist die Voreinstellung `LNP_ADDRESSING_NONE`.

Erfolgreiches Abschicken bedeutet nur, daß während der Übertragung des Pakets keine Störungen aufgetreten sind. Ist der Sender zu weit entfernt oder zeigt der Empfänger in eine andere Richtung, kann ein Paket sein Ziel trotzdem nicht erreichen. Wichtige Pakete sollten daher immer vom Empfänger bestätigt werden. So würden auch höhere Schichten von LNP vorgehen, um gesicherte Datenströme zu übertragen.

Natürlich kann man LNP auch auf dem PC benutzen – das beste Beispiel ist der Programmlader. Seine Quellen liegen im Verzeichnis *util/dll-src*. Ein eigenes Projekt sollte *lnp.c*, *rcxtty.c* und *keepalive.c* übernehmen. Das Hauptprogramm *loader.c* kann nach Herzenslust verändert werden – zum Beispiel in einen Monitor, der eingehende Debugging-Meldungen auf dem Bildschirm anzeigt. Unter Windows sind allerdings einige Patches notwendig, um die Quellen zu übersetzen.

Alternativ kann man auch `lnpd` einsetzen. Es handelt sich dabei um einen Gateway, der auf Linux-Rechnern läuft und Pakete aus dem Internet oder einem lokalen Netz auf LNP umsetzt und umgekehrt. Mit einer kleinen Bibliothek können beliebige andere Rechner auf diesen Dienst zugreifen und mit einem RCX kommunizieren, als ob sie selbst am Infrarotsender angeschlossen wären. Momentan befindet sich `lnpd` noch in der experimentellen Phase. Die Webadresse finden Sie am Ende dieses Kapitels.

Weitere Funktionen

Einige weitere nützliche Funktionen werden im folgenden kurz vorgestellt.

Zeit (time.h)

Die Variable `sys_time` zählt die Millisekunden seit dem Einschalten des RCX. Aufgrund ihrer Länge von 32 Bit erfolgt nach 49,7 Tagen ein Überlauf – ein Problem, das legOS unter anderem mit Windows teilt. Ein Roboter, der etwas Interessantes tut, wird seine Batterien auf jeden Fall deutlich vor dieser Grenze erschöpfen.

Zufallszahlen (stdlib.h)

Die linke oder doch lieber die rechte Tür? Im Leben ist man häufig gezwungen, einfach zu raten. Durch sein beschränktes Wissen über die Umgebung muß ein mobiler Roboter sogar noch häufiger eine zufällige Entscheidung treffen.

Natürlich kann kein Algorithmus wirklichen Zufall hervorbringen. Trotzdem gibt es Verfahren, die für viele Zwecke »ausreichend zufällige« Zahlenfolgen liefern – zum Beispiel um mal nach links und mal nach rechts abzubiegen oder um nach unbestimmer Zeit plötzlich wieder loszufahren.

Der verwendete Algorithmus beruht auf linearen Kongruenzen. Die bei diesem Verfahren eventuell auftretenden Korrelationen werden durch ein Permutationsarray beseitigt. Die so erzeugten Zahlenfolgen wiederholen sich erst nach über 2^{31} Aufrufen. Insgesamt ist der Generator schnell, zuverlässig und für die meisten numerischen Verfahren gut geeignet.

long int random(void)
: Gibt eine Pseudozufallszahl zurück.

void srandom(unsigned int seed)
: Initialisiert den Pseudozufallszahlen-Generator. Für gleiche Anfangswerte liefert der Generator auch gleiche Zahlenfolgen. Ein beliebter Anfangswert ist die Systemzeit: `srandom((unsigned) sys_time)`.

Strings (string.h)

Die wichtigsten Funktionen der C-Standardbibliothek zum Umgang mit Strings stehen auch unter legOS zur Verfügung.

*void memcpy(void *dest, const void *src, size_t size)*
: Kopiert *size* Bytes von *src* nach *dest*.

*void memset(void *s, int c, size_t n)*
: Überschreibt die ersten *n* Bytes von *s* mit *c*.

char strcpy(char *dest, const char *src)*
: Kopiert den String *src* nach *dest*.

*int strlen(const char *s)*
Gibt die Länge des Strings von *s* zurück (ohne das abschließende Nullzeichen).

*int strcmp(const char *s1, const char *s2)*
Vergleicht zwei Strings. Das Ergebnis ist kleiner, gleich oder größer 0, je nachdem, ob der erste String kleiner, gleich oder größer als der zweite ist.

Erweitern von Hank

In diesem Abschnitt erkläre ich ein umfangreicheres Beispielprogramm. Es steuert eine Variante von Hank, dem Roboter aus Kapitel 2, *Hank, der Panzer mit Stoßstangen*. Die einzige Veränderung am Roboter selbst ist ein zusätzlicher Lichtsensor, der vorne am Roboter angebracht und mit Eingang 2 verbunden wird. Mit diesem Sensor kann Hank hellere Bereiche des Raums suchen, während er die Berührungssensoren dazu benutzt, Hindernissen auszuweichen. Abbildung 10-2 zeigt den erweiterten Hank.

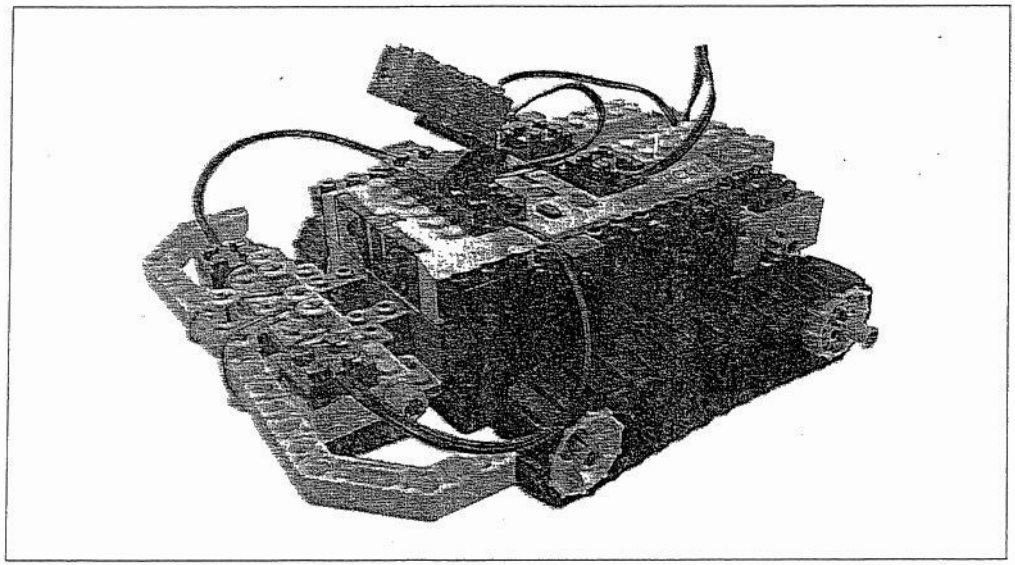

Abbildung 10-2: Hank mit einem zusätzlichen Lichtsensor

Das neue Programm für Hank bedient sich der Subsumptionsarchitektur aus Kapitel 9, *RoboTag, ein Spiel für zwei Roboter*. Zur Auffrischung: Mehrere Verhaltensmuster laufen parallel ab, wobei ein Vermittler entscheidet, welches gerade die Kontrolle über den Roboter erhält. Ursprünglich wurde diese Architektur vom Standpunkt der unterschiedlichen Verhalten aus betrachtet. Ich möchte sie aus dem Blickwinkel des Vermittlers beleuchten.

Ein Vermittler nimmt Anfragen verschiedener Prioritäten an und bestimmt, welche von ihnen tatsächlich ausgeführt werden. Woher die Anfragen kommen, ist dabei unerheblich – es zählt einzig ihre Priorität. Dem Vermittler ist es egal, wie der innere Aufbau einer Anfrage aussieht und wie sie letztlich ausgeführt wird. Ihm muß nur die Funktion zur Ausführung einer Anfrage bekannt sein. Aus objektorientierter Sicht würde man sagen: Das Entwurfsmuster Vermittler kontrolliert den Zugang zu den Methoden eines Objekts.

Ein Vorteil der Betrachtung vom Vermittler aus ist, daß die Prioritäten aller Verhalten explizit ausgeschrieben werden. Sie stehen nicht als Folge von if-Anweisungen irgendwo im Programm, sondern sind direkt als Zahlen sichtbar. Aus dieser Sicht ist es auch nicht ein Thread, der ein Verhalten ausmacht, sondern eine Priorität. Ein Verhalten, das sich nie ändert, kann mit einem einzigen Befehl an den Vermittler eingerichtet werden.

Für einen Vermittler selbst ist kein Thread notwendig. Bei genauer Betrachtung kann sich die ausgeführte Anfrage nämlich nur ändern, wenn eine neue Anfrage eingeht. Im Zeitraum zwischen zwei Anfragen muß der Vermittler keine Arbeit leisten. Also implementiert man ihn am besten als Funktion, der Anfrage und Priorität übergeben werden.

Die Suche nach dem Licht unter Rücksichtnahme auf den eigenen Körper wird von drei verschiedenen Verhalten (bzw. Prioritäten) bestimmt:

- Liegen keine anderen Anweisungen vor, steuert das Verhalten MP_CRUISE Hank stets geradeaus. Hierfür ist kein Thread notwendig.
- MP_SEEK überwacht den Lichtsensor. Sobald ein Schwellwert (engl. Threshold) unterschritten wird, versucht dieses Verhalten, Hank wieder auf die Lichtquelle auszurichten. Die notwendige Verarbeitung läuft im Thread `seeker()`.
- MP_AVOID hat die höchste Priorität. Dieses Verhalten wird von den Berührungssensoren der Stoßstangen ausgelöst und versucht Hindernissen auszuweichen. Der verantwortliche Thread ist `avoider()`.

Die im Programm verwendete Methode, heller beleuchtete Gegenden zu suchen, ist natürlich nur eine von vielen möglichen. Man kann leicht andere Verfahren ausprobieren, indem man `seeker()` anpaßt. Wegen des Vermittlers ist es nicht mehr notwendig, Hindernisse gesondert zu behandeln – MP_AVOID übernimmt bei Bedarf die Kontrolle und weicht ihnen aus.

Hier nun das vollständige Programm, *lightseeker.c*:

```
#include <unistd.h>
#include <stdlib.h>
#include <semaphore.h>
#include <conio.h>
#include <dkey.h>
#include <dmotor.h>
#include <dsensor.h>
```

```
// Verzoegerungen [ms]
//
#define BACK_TIME  500      // Zuruecksetzen
#define TURN_TIME  800      // Drehen
#define SEEK_TIME 1000      // Schwenken, um Licht zu suchen
#define WAIT_TIME   40      // Verzoegerung der Suche

// Kalibrierung des passiven Lichtsensors
//
#define RAW_DARK  0x7c00    // Minimale Helligkeit
#define RAW_LIGHT 0x6000    // Maximale Helligkeit
#define RAW_DIFF     (RAW_DARK-RAW_LIGHT)
#define HYSTERESIS   5      // Hysterese des Schwellwerts

// Prioritaeten der einzelnen Verhalten
//
#define MP_VISUAL    "csa"
#define MP_CRUISE    0
#define MP_SEEK      1
#define MP_AVOID     2
#define MP_MAX       3


//! Anforderungen an den Vermittler
typedef enum {
  COMMAND_NONE, COMMAND_FORWARD, COMMAND_REVERSE,
  COMMAND_LEFT, COMMAND_RIGHT,   COMMAND_STOP
} motor_cmd_t;

//! Anforderungen der einzelnen Verhalten
static motor_cmd_t motor_cmds[MP_MAX];

//! Darstellung des aktiven Verhaltens
static const char *motor_cmd_vis=MP_VISUAL;

//! Semaphore, der den Vermittler schützt
sem_t motor_cmd_sem;

//! Vermittler einrichten
void motor_cmd_init() {
  unsigned i;

  sem_init(&motor_cmd_sem,0,1);
  for(i=1; i<MP_MAX; i++)
    motor_cmds[i]=COMMAND_NONE;
}

//! Interne Funktion, die einen Befehl ausfuehrt.
static void motor_cmd_execute(motor_cmd_t cmd) {
  switch (cmd) {
    case COMMAND_FORWARD:
      motor_a_dir(fwd);
      motor_c_dir(fwd);
      break;
```

```
        case COMMAND_REVERSE:
          motor_a_dir(rev);
          motor_c_dir(rev);
          break;
        case COMMAND_LEFT:
          motor_a_dir(rev);
          motor_c_dir(fwd);
          break;
        case COMMAND_RIGHT:
          motor_a_dir(fwd);
          motor_c_dir(rev);
          break;
        case COMMAND_STOP:
          motor_a_dir(brake);
          motor_c_dir(brake);
          break;
        default:
      }
    }

    //! Teilt dem Vermittler die Anforderung eines Verhaltens mit.
    void motor_cmd(unsigned priority,motor_cmd_t cmd) {
      sem_wait(&motor_cmd_sem);
      if(priority<MP_MAX) {
        unsigned i;

        motor_cmds[priority]=cmd;
        for(i=MP_MAX-1; i<MP_MAX; i--)
          if(motor_cmds[i]!=COMMAND_NONE) {
            cputc(motor_cmd_vis[i],0);
            motor_cmd_execute(motor_cmds[i]);
            break;
          }
      }
      sem_post(&motor_cmd_sem);
    }

    //! Weckt bei Kontakt.
    static wakeup_t touch_event(wakeup_t data) {
      if(TOUCH_1)
        return 1;
      else if(TOUCH_3)
        return 3;
      return 0;
    }

    //! Thread, der Kontakten ausweicht.
    int avoider(int argc, char **argv) {
      while(1) {
        wakeup_t res=wait_event(touch_event,0);

        motor_cmd(MP_AVOID, COMMAND_REVERSE);
        msleep(BACK_TIME);
```

```
      motor_cmd(MP_AVOID, res==1 ? COMMAND_RIGHT : COMMAND_LEFT);
      msleep(TURN_TIME);
      motor_cmd(MP_AVOID, COMMAND_NONE);
    }
    return 0;
  }

  //! Spezielle Verarbeitung des passiven Lichtsensors
  static int process_light(int raw) {
    int cooked=100 - (int)( (100*(long)(raw-RAW_LIGHT)) / RAW_DIFF );
    if(cooked<0)
      return 0;
    if(cooked>99)
      return 99;
    return cooked;
  }

  //! Sucht in Richtung dir fuer delta Millisekunden bessere Beleuchtung als
  // threshold.
  int seek_better(int dir,int threshold,int delta) {
    static const motor_cmd_t dirs[2]={COMMAND_RIGHT, COMMAND_LEFT};
    long timeout=sys_time + delta;

    motor_cmd(MP_SEEK, dirs[dir ? 1 : 0]);
    do {
      int current=process_light(SENSOR_2);
      lcd_unsigned(current*100 + threshold);
      if(current >= threshold+HYSTERESIS)
        return current;

      msleep(WAIT_TIME);
    } while(sys_time < timeout);

    return -1;
  }

  //! Thread, der bessere Beleuchtung sucht.
  int seeker(int argc, char **argv) {
    int threshold=process_light(SENSOR_2),i;

    for(i=0; ; i++) {
      // Helligkeit messen.
      int current = process_light(SENSOR_2);
      lcd_unsigned(current*100 + threshold);

      // Schwellwert langsam erhoehen.
      if(threshold<100 && !(i%5))
        threshold++;

      // Zu dunkel -> Suche in zufaellige Richtung.
      //
      if(current < threshold-HYSTERESIS) {
        unsigned dir=random() & 1;
        threshold=seek_better(dir,current,SEEK_TIME);
```

```
        if(threshold==-1) {
          threshold=seek_better(1-dir,current,2*SEEK_TIME);
          if(threshold==-1)
            threshold=current;
        }
        motor_cmd(MP_SEEK, COMMAND_NONE);
      }

      msleep(WAIT_TIME);
    }
    return 0;
  }

  //! Hier startet das Programm.
  int main(int argc, char **argv) {
    srandom((unsigned)sys_time);

    // Vermittler und Standardverhalten einrichten.
    motor_cmd_init();
    motor_cmd(MP_CRUISE,COMMAND_FORWARD);

    // Motoren starten.
    motor_a_speed(MAX_SPEED);
    motor_c_speed(MAX_SPEED);

    // Threads starten.
    execi(avoider, 0, NULL, PRIO_NORMAL, DEFAULT_STACK_SIZE);
    return seeker( 0, NULL);
  }
```

lightseeker.c ist ein ziemlich langes Programm. Ich erkläre zunächst die Implementierung des Vermittlers, dann die Funktion `main()` und schließlich die dort gestarteten Threads.

Die Namen aller Bestandteile des Vermittlers beginnen mit `motor_cmd`, da dieser spezielle Vermittler den Zugang zu den Motoren kontrolliert. Der Datentyp `motor_cmd_t` definiert die möglichen Anfragen an den Vermittler.

Die Funktion `motor_cmd_init()` richtet den Vermittler ein. Dazu wird ein Semaphore initialisiert, der den Zugang absichert. Weiter wird das Array `motor_cmds` initialisiert, in dem der Vermittler Buch über die jeweils letzte Anfrage jeder Prioritätsstufe führt. Diese Daten werden benötigt, wenn eine höhere Prioritätsstufe die Kontrolle an eine niedrigere abgibt.

`motor_cmd()` leistet die eigentliche Vermittlungsarbeit. Semaphoren-Aufrufe stellen sicher, daß jeweils nur ein Thread zur Zeit diese Funktion ausführen kann. Zunächst trägt `motor_cmd()` die neue Anfrage entsprechend ihrer Priorität in das Array `motor_cmds` ein. Ausgehend von der höchsten Priorität, wird dann die auszuführende Anfrage bestimmt und an `motor_cmd_execute()` übergeben. Zusätzlich wird das aktive Verhalten rechts auf dem Display durch einen Buchstaben aus `motor_cmd_vis` angezeigt.

Die Prioritätsstufen und ihre Darstellung auf dem Display sind durch die Konstanten MP_XXX am Anfang des Quelltextes festgelegt. So kann der Vermittler einfacher in eigenen Projekten wiederverwendet werden. Eingestellt sind drei verschiedene Stufen: MP_CRUISE mit Priorität 0 und Darstellung »C«, MP_SEEK mit 1 und »S« sowie MP_AVOID mit 2 und »a«. Höhere numerische Werte entsprechen höherer Priorität.

Wird das Programm gestartet, richtet main() den Vermittler und das konstante Verhalten MP_CRUISE ein, das immer nur geradeaus fährt. Dann werden die Motoren eingeschaltet. Die weiteren Verhalten benötigen zwei Threads. Für MP_AVOID wird ein neuer Thread zur Ausführung von avoider() gestartet. MP_SEEK kann einfach den aktuellen Thread für seeker() verwenden.

avoider() wartet mit der Weckfunktion touch_event() darauf, daß eine Stoßstange angestoßen wird, und weicht Hindernissen auf die übliche Art und Weise aus: Hank wird zurückgesetzt und in die der aktivierten Stoßstange entgegengesetzte Richtung gedreht. Die entsprechenden Verzögerungen BACK_TIME und TURN_TIME sind am Anfang des Quelltextes definiert.

Die Funktion seeker() ist etwas komplexer. Sie unterhält einen Schwellwert der Helligkeit, threshold. Der Schwellwert wird zunächst durch eine Messung bestimmt und dann langsam erhöht, damit Hank nicht in einer dunklen Ecke hängenbleibt. Sobald die gemessene Helligkeit den Schwellwert um einen gewissen Betrag unterschreitet, wird zufällig eine Richtung ausgewählt und dort eine Weile mit seek_better() nach besserer Beleuchtung gesucht. Ist dies nicht erfolgreich, wird doppelt so lange in der entgegengesetzten Richtung gesucht. Wenn alle Stricke reißen, wird der letzte Meßwert als neuer Schwellwert benutzt.

seek_better() erhält Richtung, Schwellwert und Dauer der Suche als Eingangswerte. Die Funktion dreht Hank in die entsprechende Richtung und überwacht die Helligkeit. Sobald diese den Schwellwert um einen gewissen Betrag überschreitet, wird der Meßwert zurückgeliefert. Ist die Suche erfolglos, wird -1 zurückgegeben.

Beide Suchfunktionen zeigen Meßwert und Schwellwert nebeneinander auf dem Display an. Der Wertebereich ist jeweils 0 bis 99. Die Umrechnung der rohen Sensorwerte erfolgt in der Funktion process_light(). Im Einzelfall kann es nützlich sein, die dabei verwendeten Konstanten RAW_DARK und RAW_LIGHT speziell für den eigenen Sensor einzustellen.

Programme

Seit Version 0.2.0 kann der legOS-Kernel mehrere Programme unabhängig voneinander in den Speicher laden, verwalten und ausführen. Ihre Übertragung erfolgt mit doppelter, wahlweise sogar mit vierfacher Geschwindigkeit. Im Vergleich zu älteren Versionen des Kernels, die – wie viele Betriebssysteme für eingebettete Systeme – statisch mit einer Anwendung verknüpft und mit einfacher Geschwindigkeit immer neu geladen wurden, hat sich der Entwicklungszyklus dadurch enorm beschleunigt.

Im folgenden wird erläutert, was Programme unter legOS ausmacht und wie der Kernel, der Programmverwalter und der dynamische Linker und Lader dll auf dem Host mit ihnen umgehen.

Was sind Programme?

In drei von vier Fällen bekommt man auf die Frage »Was ist ein Programm?« zur Antwort: »Etwas, das man ausführen kann.« Diese Beschreibung vermittelt leider kein besonders klares Bild davon, was Programme ausmacht.

Unter legOS besteht ein Programm aus ausführbarem Hitachi-H8-Maschinencode und verschiedenen Arten von Daten, auf denen der Code arbeitet. Eine detaillierte Aufschlüsselung der Abschnitte eines Programms befindet sich in Tabelle 10-2.

Tabelle 10-2: Abschnitte eines Programms

Name	Beschreibung
.text	Programmcode (enthält initialisierte, konstante Daten .rodata)
.data	Initialisierte, veränderbare Daten
.bss	Nichtinitialisierte, veränderbare Daten
.stack	Laufzeit-Stack für lokale Variablen, Rücksprungadressen usw.

Innerhalb des Programmcodes ist eine Adresse besonders gekennzeichnet – diejenige, bei der die Ausführung des Programms beginnt. Wie in der Programmiersprache C üblich, entspricht die Einsprungadresse unter legOS der Funktion main() des Quelltextes.

Was passiert, wenn ein Programm ausgeführt wird? Der Programmverwalter auf dem RCX erzeugt einfach einen neuen Thread mit der main()-Funktion des Programms als Startadresse. Der Verwalter selbst ist ebenfalls ein Thread, der vom legOS-Kernel bei jedem Einschalten gestartet wird. Er ist auch für das Laden von Programmen über das Netzwerk zuständig.

Veränderbare Daten

Die Programmiersprache C bietet die Möglichkeit, Variablen einen anfänglichen Wert zuzuweisen. Handelt es sich um globale oder statische Variablen, erzeugt der Übersetzer hierfür keinen Programmcode – er schreibt den anfänglichen Wert direkt in den der Variable zugeordneten Speicherbereich. Alle diese Variablen sind im Abschnitt .data in Tabelle 10-2 zusammengefaßt.

Wird das Programm erneut ausgeführt, hat der erste Durchlauf diese Anfangswerte möglicherweise überschrieben. Auf Rechnern mit Dateisystem ist dies kein Problem – schließlich wird das Programm vor jeder Ausführung erneut geladen. Unter legOS liegt das Programm hingegen direkt im Speicher. Die geänderten Daten können ohne Hilfe des Hosts nicht erneut geladen werden.

Vielleicht würde das Programm sich im zweiten Lauf komplett anders verhalten, zum Beispiel im Fall eines Automaten, dessen Startzustand sich verändert hat. Vielleicht wäre auch kaum ein Unterschied bemerkbar. Auf jeden Fall wäre es nicht dasselbe Programm.

Um die korrekte Ausführung zu gewährleisten, muß der Programmverwalter konservativ handeln und eine komplette Kopie von `.data` anlegen. Vor jedem Programmstart wird `.data` aus dieser Kopie wiederhergestellt. Eine Kopie von `.bss` ist nicht notwendig – der nichtinitialisierte Datenbereich wird einfach mit Nullen gefüllt.

Dieses Vorgehen klingt extrem aufwendig. De facto benötigt es jedoch bei sauberer Programmierung meist sehr wenig zusätzlichen Speicher. Umfangreiche initialisierte Variablen wie Tabellen von Zustandsübergängen ändern sich nur sehr selten. Sofern der Programmierer sie als `const` deklariert, werden sie vom Compiler im Abschnitt `.rodata` bzw. `.text` untergebracht. Ebensowenig blähen große Datenstrukturen für Berechnungen zur Laufzeit oder Arrays zur Sammlung von Daten den Speicherbedarf der Kopie auf. Wenn ihnen kein anfänglicher Inhalt zugewiesen wird, weist ihnen der Compiler Speicher in `.bss` zu.

Mehrere Programme

Wenn Programme auf einem Prozessor ausgeführt werden, kann dieser nicht mehr auf Variablennamen, Sprungmarken oder ähnliches zurückgreifen. Programme in Maschinensprache arbeiten mit Registern und Speicheradressen. Der Übersetzer einer Hochsprache wie C bemüht sich, die am häufigsten verwendeten Variablen des Quellprogramms auf verfügbare Register abzubilden. Den verbleibenden Variablen werden Speicheradressen zugewiesen. Funktionen und Sprungmarken werden im Maschinencode ausschließlich über ihre Adressen angesprochen, entweder absolut (springe an Adresse 0x1234) oder relativ (springe 0x56 Adressen vorwärts).

Was passiert in einer Umgebung, die mehrere Programme in den Speicher lädt? Zwei Programme können nicht an dieselbe Adresse geladen werden, eines von ihnen muß also an eine andere Ursprungsadresse verschoben werden. Bezüge auf relative Adressen bleiben davon unberührt, Bezüge auf absolute Adressen müssen hingegen korrigiert werden.[8]

Was ist mit Aufrufen an das Betriebssystem? Abgesehen von Sonderfällen, werden sie im Maschinencode ebenfalls zu Sprüngen an absolute Adressen. Wird ein Programm im Speicher verschoben, bleiben die Adressen des Betriebssystems unverändert. Es wäre also verheerend, bei einer Verschiebung alle Verweise auf absolute Adressen zu ändern. Es ist notwendig, genau die Verweise zu finden und zu korrigieren, die sich auf Symbole des Programms selbst beziehen.

8 Einige Prozessorarchitekturen führen alle Speicherzugriffe relativ zu einem Basisregister aus. Systeme mit virtuellem Speicher teilen jedem Programm einen eigenen, gleichartigen Adreßraum zu. Auf solchen Architekturen können Programme problemlos ohne Adreßkorrektur verschoben werden.

Wie aus Tabelle 10-2 ersichtlich, wäre es auch möglich, nicht das Programm als Ganzes, sondern seine unterschiedlichen Abschnitte unabhängig voneinander im Speicher zu verschieben. Bezüge auf `.text`, `.data` und `.bss` müßten dann getrennt korrigiert werden. Bei diesem Vorgehen würden umfangreiche Anwendungen besser in den Hauptspeicher passen, wenn der freie Speicher in getrennte Bereiche aufgeteilt ist.

Momentan unternimmt legOS dazu keine Anstrengungen. Im Speicher folgen alle Programmabschnitte in der Reihenfolge von Tabelle 10-2 lückenlos aufeinander. Die einzige Ausnahme ist der Laufzeit-Stack – er wird vom Kernel bei der Erzeugung des Threads zur Programmausführung zugewiesen.

Das Dateiformat

Das Format *legOS Executable* (Dateiendung .lx) enthält außer dem ausführbaren Programmcode Informationen über Lage und Länge der Programmbereiche aus Tabelle 10-2 sowie die Anzahl und Position der bei einer Verschiebung zu korrigierenden Adressen.

Im .lx-Format liegen die höherwertigen Bytes eines Datenwortes vor den niederwertigen. Beispielsweise wird 0x1234 in der Datei als 0x12, gefolgt von 0x34, abgespeichert. Dieses Format wird auch vom Hitachi H8-Prozessor des RCX verwendet, nicht aber von x86-PCs. Es ist als *Big Endian Format* oder *Network Byte Order* bekannt.

Tabelle 10-3 beschreibt die Felder des Datei-Headers. Unmittelbar auf den Header folgen `text_size` Bytes Programmcode, dann `data_size` Bytes Daten. Der Inhalt von `.bss` wird nicht gespeichert, da es sich um Nullen handelt. Den Abschluß der Datei bilden `num_relocs` Relokationseinträge von jeweils 2 Bytes. Es handelt sich bei ihnen um Offsets relativ zum Beginn des Programmcodes.

Tabelle 10-3: Das Format legOS Executable (.lx), Version 0

Eintrag	Bytes	Beschreibung
`id`	6	Der String »`legOS\0`«
`version`	2	Version des Dateiformats, 0
`base`	2	Basisadresse des Programmcodes
`text_size`	2	Länge des Programmcodes
`data_size`	2	Länge der initialisierten, veränderlichen Daten
`bss_size`	2	Länge der nichtinitialisierten Daten
`stack_size`	2	Größe des Stacks für den Thread `main()`
`offset`	2	Offset der Routine `main()` von `base`
`num_relocs`	2	Anzahl der notwendigen Relokationen

Der Ladevorgang

Wenn der Benutzer auf dem Host `dll -p3 program.lx` aufruft, um ein Programm auf einen RCX zu übertragen, beginnt ein Zusammenspiel von `dll` und dem Programmverwalter des Zielrechners. Abbildung 10-3 zeigt den genauen Ablauf.

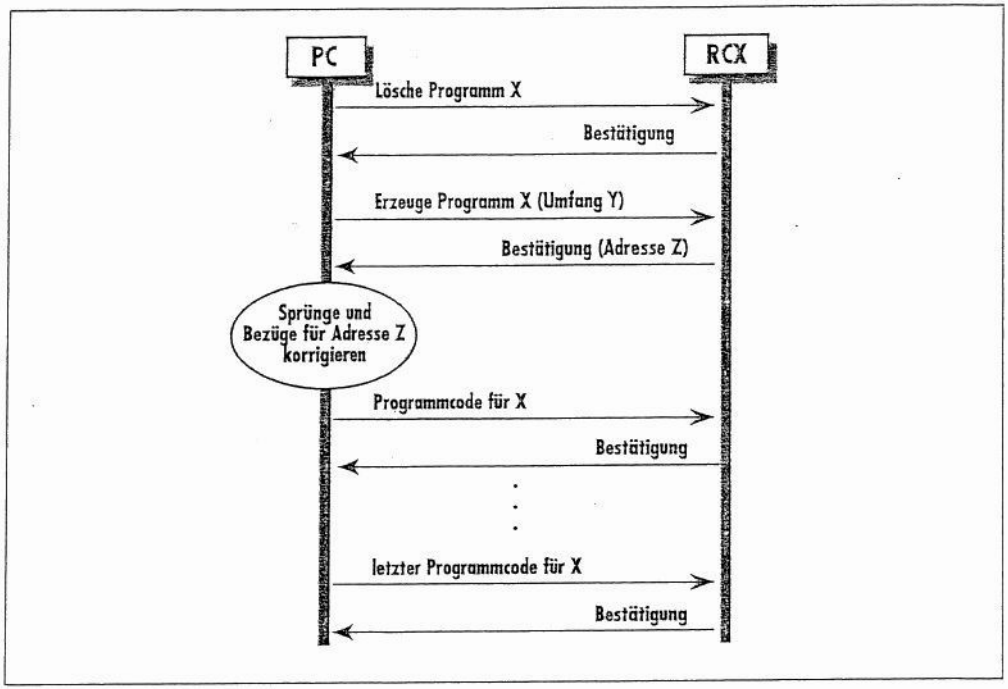

Abbildung 10-3: Das Zusammenspiel von dll und Programmverwalter

`dll` erteilt zunächst den Befehl, das entsprechende Programm zu löschen. Dies wird vom Programmverwalter ausgeführt und bestätigt. Geschieht dies nicht, bricht der Ladevorgang mit der Fehlermeldung »error deleting program« ab.

Dann teilt `dll` dem Programmverwalter mit, wieviel Platz das neue Programm für seinen Maschinencode, die verschiedenen Datenabschnitte und den Stack benötigt. Der Programmverwalter fordert den notwendigen Speicherplatz vom legOS-Kernel. Im Erfolgsfall setzt er `dll` über die Adressen der reservierten Bereiche in Kenntnis, ansonsten bricht der Ladevorgang an dieser Stelle mit einem »error creating program« ab.

`dll` benutzt jetzt die Adressen der Zielbereiche, um alle absoluten Speicherbezüge innerhalb des Programms zu korrigieren. Das fertig relozierte Programm und die initialisierten Daten werden dann an den RCX übertragen. Eine Unterbrechung in dieser Phase führt zur Meldung »error downloading program«, ansonsten steht das Programm zur Ausführung bereit.

Die eigentliche Relokation des Programms an die neue Basisadresse erfordert nur minimale Rechenleistung. Tatsächlich sieht sie folgendermaßen aus:

```
for(i=0; i<num_relocs; i++) {
  unsigned short *ptr=(unsigned short*) (text+reloc[i]);
  *ptr+=new_base - base;
}
```

Wenn die Tabelle mit den zu korrigierenden Offsets an den RCX übermittelt würde, könnte diese Korrektur ohne weiteres auch dort erfolgen. Dann wäre sogar eine Übertragung von Programmen von einem RCX zum nächsten à la Palm Pilot möglich. Man könnte mit der begrenzten Bandbreite des Infrarotkanals für die vorliegende Variante argumentieren, aber letztlich ist sie in der historischen Entwicklung begründet.

Tips und Tricks

Es ist schwer, alle Möglichkeiten von legOS auf Anhieb auszuschöpfen. Es gibt viel mehr Optionen als bei anderen Umgebungen für den RCX. Natürlich kann auch viel mehr schiefgehen – insbesondere, wenn man mit Entwicklung unter Unix wenig vertraut ist. Dieser Abschnitt soll dabei helfen, die lästigsten Klippen zu umschiffen.

Makefiles

Wer immer nur ein Programm aus einer Datei übersetzt, kann einfach das Makefile aus *legOS/demo* verwenden und `make name.lx` aufrufen. Ansonsten lohnt es sich fast immer, das Makefile den eigenen Bedürfnissen anzupassen.

Programme, die aus nur einer Datei bestehen, trägt man einfach auf dieser Zeile ein:

```
PROGRAMS=eins.lx zwei.lx drei.lx
```

Durch den Eintrag werden Argumente an `make` überflüssig – es sucht automatisch die zu jedem Programm gehörige C- oder C++-Quelldatei (.c bzw. .cpp) und erzeugt daraus Objekte und schließlich Programme.

Größere Programme werden schnell unhandlich, wenn man sie nicht auf mehrere Dateien verteilt. Weitere Objektdateien können den Programmen folgendermaßen hinzugefügt werden:

```
DOBJECTS=anneal.o map.o
```

Manchmal benötigen die Programme einen größeren Laufzeit-Stack. Für alle Threads, die das Programm erzeugt, kann man den notwendigen Platz im Programmtext selbst anfordern. Die Stack-Größe des ersten Threads kann man im Makefile einstellen:

```
STACKSIZE=-sXXXX
```

Kernel-Einstellungen

legOS ist modular aufgebaut. Viele Bestandteile des Kernels lassen sich nach Belieben einbinden oder ausschließen. Seit es den Programmverwalter gibt, spielt dies allerdings keine so große Rolle wie zuvor – der Kernel wird nicht mehr so häufig neu geladen, und ein Großteil der Funktionen wird vom Programmverwalter benötigt.

Die meisten Einstellungen können in der Datei *boot/config.h* vorgenommen werden. Tabelle 10-4 zeigt eine Liste der wichtigsten Einträge dieser Datei.

Tabelle 10-4: Wichtige Kernel-Einstellungen in config.h

Eintrag	Beschreibung
CONF_LNP_FAST	Wählt die Übertragungsrate (2400 oder 4800 bps).
CONF_LNP_HOSTADDR	Legt die Netzwerkadresse fest. Der Wertebereich ist 0x00-0xff.
CONF_LNP_HOSTMASK	Stellt die Netzwerkmaske ein. Pakete an x erreichen einen RCX, wenn (x & MASK) == ADDR.
CONF_DSENSOR_ROTATION	Bindet Unterstützung für den Rotationssensor ein.

Wer mehrere verschiedene Kernel benötigt, um zwei oder mehr RCX ihre eindeutigen Netzwerkadressen zuzuweisen, kann neue Verzeichnisse (*boot2/*, *boot3/* usw.) anlegen und *boot/config.h* sowie *boot/Makefile* dorthin kopieren und anpassen. Ein `make` in diesen Verzeichnissen hat dann den gewünschten Effekt.

Ein Teil der Funktionalität von legOS befindet sich in Bibliotheken. Eine wichtige Möglichkeit, die Größe des Kernels zu beeinflussen, liegt in der Art ihrer Einbindung. Der Standard-Kernel bindet nur diejenigen Teile der C- und Ganzzahlroutinen ein, die er selbst benötigt. Wenn Programme andere Teile dieser Bibliotheken oder Funktionen der Fließkomma- und C++-Bibliotheken ansprechen, werden diese mit dem jeweiligen Programm übertragen. Dies stellt normalerweise einen guten Kompromiß zwischen der Ladezeit des Kernels und den Ladezeiten von Programmen dar.

Es kann sich unter Umständen lohnen, Bibliotheken komplett an den Kernel zu binden. Das vergrößert den Kernel und seine Ladezeit, beschleunigt aber das Laden von Programmen. Wenn mehrere Programme die gleichen Funktionen benutzen, verringert es sogar ihren gemeinsamen Speicherbedarf. Um zum Beispiel die C-, Ganzzahl- und Fließkomma-Bibliotheken fest an den Kernel zu binden, ist folgender Eintrag in *Makefile.kernel* notwendig:

```
LIBS=--whole-archive -lc -lmint -lfloat
```

C++

Aus Tabelle 10-5 ist ersichtlich, welche Bestandteile von C++ unter legOS-0.2.4 unterstützt werden. Abgesehen von Exceptions sind mittlerweile alle Grundbestandteile der Sprache verfügbar.

Tabelle 10-5: legOS und C++

Unterstützt	Nicht unterstützt
Klassen	Exceptions
Vererbung	RTTI
new / delete	
virtual	

Leider erzeugt der C++-Compiler *g++* keinen besonders guten Code für H8-Prozessoren – im Vergleich zu *gcc* werden ähnliche Programme bisweilen mehr als dreimal so groß. Bei den beschränkten Speicherverhältnissen des RCX ist dies noch ein echtes Problem.

Die Abwesenheit von Klassenbibliotheken ist eher als Herausforderung zu sehen. Die meisten Systemobjekte und ihre Header-Dateien können leicht auf ein Klassenmodell abgebildet werden. Ich würde mich über alle Anstrengungen freuen, die in diese Richtung unternommen werden.

Online-Ressourcen

legOS

legOS
http://www.noga.de/legOS/

Meine offizielle Seite zu legOS. Hier findet man die jeweils neueste Distribution und notwendige Software wie den Übersetzer ebenso wie Diskussionsforen und Anleitungen. Die Mehrzahl der übrigen Seiten ist von hier aus erreichbar.

Luis' legOS Page
http://arthurdent.dorm.duke.edu/legos/

Luis Villas Seite hat zunächst durch das legOS-HOWTO auf sich aufmerksam gemacht. Windows-Benutzer finden hier Winlegos von Rossz Vámos-Wentworth sowie Hinweise zur Installation. Der LNP-Dämon ist im Verzeichnis */legos/archives/LNP* abgelegt.

legOS at sourceforge
http://legOS.sourceforge.net/

Auf dieser Seite kann man mit der Versionsverwaltung CVS die aktuellsten legOS-Dateien laden. Änderungen finden schneller Eingang in CVS als in eine Distribution – manchmal bedeutet dies mehr, manchmal weniger Stabilität.

Robotics / RCX / legOS
http://www.lugnet.com/robotics/rcx/legos/

LUGNET ist die zentrale Anlaufstelle, um im Internet über LEGO zu diskutieren. Die legOS-Gemeinde hört gern von Erfahrungen und Erfolgen und hilft bei Problemen weiter.

Entwicklungsumgebung

egcs Releases
ftp://egcs.cygnus.com/pub/egcs/releases/

Hier findet sich der zur Übersetzung von legOS benötigte Compiler egcs. Die vollständigen Quellen umfassen ca. 11 MB.

Binutils – GNU Project – Free Software Foundation (FSF)
http://www.gnu.org/software/binutils/binutils.html

Die Homepage der `binutils`. Zu diesem Paket gehören unter anderem Linker und Assembler, die man ebenfalls für legOS braucht. Die Quellen sind gepackt ca. 5 MB groß.

Cygwin
http://sourceware.cygnus.com/cygwin/

Cygwin ist eine freie Unix-Umgebung für Windows von Cygnus Solutions. Teile davon werden von legOS unter Windows benötigt, andere vermitteln einfach mehr Lebensqualität.

Perl.com – Acquiring Perl Software
http://www.perl.com/pub/language/info/software.html

Perl ist ab Version 0.2.4 nicht mehr Teil der Entwicklungsumgebung. Wer sich für die Sprache interessiert, findet hier Anleitungen und Pakete zur Installation von Perl auf allen unterstützten Plattformen.

Firmware-Lader

RCX Tools
http://graphics.stanford.edu/~kekoa/rcx/tools.html

Der populäre Firmware-Lader `firmdl3` findet sich hier in seiner jeweils neuesten Version. Er wurde von RCX-Pionier Kekoa Proudfoot entwickelt, der unter legOS entstandene Neuerungen wie das Laden mit vierfacher Geschwindigkeit sofort integriert hat.

Von dieser Seite aus sind auch Kekoas überragende Dokumentation der Interna des RCX und seine Low-Level-Bibliothek `librcx` erreichbar.

NQC – Not Quite C
http://www.enteract.com/~dbaum/nqc/index.html

Auch `nqc` kann Firmware auf den RCX übertragen, daher ist die entsprechende Homepage hier nochmals aufgeführt.