Softwareentwicklung für den Mikrocontroller M68HC11 unter Linux: Assembler, C, Java

ingo beckmann
auf dem hirschberg 19
53225 bonn-beuel
ingo.beckmann@fernuni-hagen.de

Einleitung

Eingebettete Systeme sind in aller Munde. Linux wird immer erfolgreicher als Betriebssystem für eingebettete Systeme mit 32-Bit CPUs. Wenn Speicher und Rechenpower hingegen minimalistisch anmuten, darf die Hardware aber auch schon mal unter der Regie eines anderen OS laufen. Linux bietet für solche Fälle geeignete Entwicklungsumgebungen für eine Anzahl von Zielplattformen. Im Linux Magazin wurde in der Vergangenheit bereits mehrfach über die Programmierung von Kleinstrechnern unter Linux mit verschiedenen Aufgabenstellungen berichtet (vgl. [9]).

Die Plattform, die im Folgenden zum Einsatz kommt, ist die M68HC11 Mikrocontrollerfamilie von Motorola. Da die Mikrocontroller (MCU) dieser Reihen seit langem bei vielen Entwicklern sehr beliebt sind, existiert eine schier unüberschaubare Flut an Internetsites, Angeboten von Hard- und Software von kommerziellen Firmen und Hobbyelektronikern, sowie Dienstleistungen. Im Internet findet man viele (z.T. frei verfügbare) Platinenlayouts und Boards, mitunter reichlich ausgestattet mit Schnittstellen und peripherem Gerät.

Der Artikel führt zunächst kurz die MCU mit ihren on-chip Funktionen ein. Dann soll es von der Programmierung in Assembler, über die Programmiersprache C zu der Verwendung von Java mittels einer lauffähigen virtuellen Maschine für eingebettete Systeme gehen. Man stelle sich vor: Java auf einem 8-Bit Mikrocontroller, dem gerade mal maximal 64 Kbytes RAM zur Verfügung stehen!

Mikrocontroller M68HC11

Mikrocontroller sind mit zusätzlichen Funktionen ausgestattete CPUs. Die on-chip Peripherie unterstützt häufig verbreitete Bussysteme (RS-232, Ethernet, CAN, I2C, SPI, ...), AD-Wandler, LCD- und DRAM-Controller, usw. MCUs sind dazu ausgelegt, zuverlässig im Verborgenen zu werkeln, sparsam im Stromverbrauch und kostengünstig. Mit Aufgaben aus dem Regelbereich oder der Messdatenerfassung reicht ihnen in vielen Fällen eine mäßige Datenverarbeitungsgeschwindigkeit. Der 68hc11 von Motorola ist

ein in CMOS-Technologie gefertigter 8-Bit Mikrocontroller dessen statisches Design einen Bustakt von bis zu 2 MHz erlaubt. Seine CPU verwaltet zwei 8-Bit Akkumulatoren A und B, die zu einem 16-Bit Register ('D') zusammengeschlossen werden können, sowie zwei Indexregister X und Y zur indizierten Adressierung. Seine Performance beträgt etwa bescheidene 0.5 MIPS.

Ausstattungsmerkmale der M68HC11 A-Reihe sind:
- 256 Bytes RAM
- bis zu 512 Bytes internes EEPROM
- bis zu 8 Kbytes internes ROM
- 64 KB linear adressierbares RAM/EPROM (extern)
- 16-Bit Timer System mit Input Capture- und Output Compare-Funktion
- 8-Bit Pulszähler
- 21 Interruptquellen, u.a. Echtzeitinterrupt (RTI)
- synchrone serielle Schnittstelle (SPI)
- asynchrone serielle Schnittstelle (SCI, i.e. RS-232 ohne Flusskontrolle)
- 8-Kanal 8-Bit Analog-zu-Digital Wandler
- mehrere freie Ein- und Ausgabeleitungen

Die Funktionsblöcke und Ports der MCU können in Abbildung [blockDiagrmm] identifiziert werden. Tiefergehende Beschreibungen und interessante Links findet man auf der Homepage von Motorola [1] und auf den Seiten des 68hc11-Webrings [1]. Die vom Autor verwendete Platine (s. Abb. [platine]) liegt dem Buch von Michael Rose [9] bei. Diese kann jeder mit ein wenig handwerklichem Geschick selber bestücken und in Betrieb nehmen. Die Platine sieht einen Chip namens M68HC24 vor. Diese 'Port Replacement Unit' (PRU) hat zur Aufgabe, 16 I/O-Leitungen der zwei Ports B und C, die bei Einsatz eines externen Speichers (das ist der auf dem Board zusätzlich vorhandene Speicher von z.B. 32 KB) im sogenannten 'expanded mode' durch den gemultiplexten Daten/Adressbus belegt sind, wiederherzustellen.
Der 68hc24 ist seit einiger Zeit von Motorola abgekündigt, d.h. er ist nur noch sehr schwer aus Restbeständen zu besorgen. Lässt man deshalb den 68hc24 auf der Platine weg, hat das den Vorteil, dass man das System mit wenig Aufwand auf volle 64 KB (= maximal adressierbarer Raum) 'aufbohren' kann: Ein 32 KB-Speicherchip (statisches RAM 32K*8 Typ '62256' oder '43256', bzw. EEPROM Typ '28C256'; nicht jedoch EPROM wegen der abweichenden Pinbelegung!) wird vom vorhandenen 32 KB-Chip Huckepack genommen, und alle Beinchen - bis auf eines - werden an die Beinchen des unteren angelötet. Pin 20 (Chip Enable, active low) hingegen wird direkt über einen dünnen Draht mit der Adressleitung A15 irgendwo auf der Platine verbunden; der untere Chip erhält die invertierte Leitung A15 per Voreinstellung, so dass immer nur einer der beiden Chips aktiviert ist. Der untere Speicherchip liegt dann im Adressraum hexadezimal 0x8000-0xFFFF, der obere deckt den Bereich 0x0000-0x7FFF ab.

Prinzipiell eignet sich für die in diesem Artikel vorgestellte Software jeder Entwurf mit mindestens 48 Kbyte Speicher. Einige Bauteillieferanten sind im Infokasten aufgelistet [7].

Die Betriebsmodi und das Bootkonzept des 68hc11:
Nach dem Einschalten der Betriebsspannung von üblicherweise 5 Volt, bzw. einem Reset und dem Stabilisieren des Oszillators entscheiden zwei Inputs (MODA, MODB über Schalter oder Jumper) darüber, in welchem Betriebsmodus sich die MCU befindet. Von den prinzipiell vier Möglichkeiten sind vor allem zwei interessant: Durch MODA und MODB auf Massepotenzial während des Resets selektiert man 'Special

Bootstrap', durch MODA=1 und MODB=1 wird die MCU in den Zustand 'Normal Expanded' geschaltet. Was bedeutet das ?

Special Bootstrap:
Die Ausführungseinheit der MCU wartet auf die Übertragung eines Bootprogramms ('Bootloader', 256 Bytes) über die serielle Schnittstelle (1200 Baud), lädt nach der Übertragung den Programmzähler (PC) mit der Adresse hexadezimal 0x0000 (hier liegt dann das eben übertragene Programm) und beginnt mit der Abarbeitung des Codes an eben dieser Adresse.

Normal Expanded:
Die MCU springt an die Adresse, auf die der Resetvektor (0xFFFE-0xFFFF) zeigt, und beginnt mit der Ausführung des dort gespeicherten Programms. Entweder ist dieses Programm zuvor vom Bootloader dorthin geladen worden, oder es befindet sich in einem E(E)PROM, welches durch ein externes EPROM-Programmiergerät dauerhaft beschrieben wurde.

Der typische Zyklus während des Entwickelns wird in Abbildung [booting] verdeutlicht. Programme zur seriellen Kommunikation zwischen PC und 68hc11, sowie dazu passende Bootloader findet man im Internet, z.B. auf [6; 1]. Es wird noch ein einfaches Kabel mit der in Abbildung [kabel] dargestellten Pinbelegung benötigt.

Stellt sich die Frage, welche Programme sinnvollerweise in den Mikrocontroller geladen werden können. Damit befassen sich die kommenden Abschnitte über Assembler-, C- und Java-Programmierung.

Assembler

Die Beschäftigung mit Mikrocontrollern zwingt den Entwickler zwangsläufig dazu, den Befehlssatz etwas genauer unter die Lupe zu nehmen. Erste Schritte gelingen einem am einfachsten in Assembler. Ein beliebter und stabil laufender Assembler/Linker für den 68hc11 und eine Reihe weiterer MCUs ist der ASXXXX Makroassembler von Alan R. Baldwin [3]. Der Linker produziert wahlweise Intel hex-Dateien oder ASCII-Dateien im Motorola S-Record-, bzw. s19-Format. Am Beispiel einer S-Record Datei (Erweiterung *.s19) sei kurz dieses Format erklärt. Abbildung [s1record.gif] zeigt ein aus einer einzigen Zeile bestehendes 'Programmlisting': Nach dem Identifier S1 (=> 2-Byte Adresse), folgt die Anzahl der Bytes (5) dieser Zeile, die Adresse, an die das Programm geladen wird (0xFFFE), das eigentliche Programm (90 00), sowie eine Checksumme (invertierte Summe der Einträge). Nach dem Download in den Controller zeigt dessen Reset-Vektor also auf die Speicherzelle hexadezimal 0x9000, die Programmausführung im Normal Expanded Modus würde nach einem Neustart demnach dort beginnen. S-Records sind detaillierter unter [3] erklärt.

GCC – GNU Compiler Collection

Die Verwendung von Cross-Compilern wurde bereits in früheren Ausgaben des Linux-Magazins (z.B. 01/2000) diskutiert. Damals ging es um 32-Bit Mikrocontroller der Reihe Motorola 683xx, die zum Beispiel auch den Dragonball einschließt, der im PalmPilot verbaut ist. Vereinfacht gesagt geht es bei der Crosstarget-Entwicklung darum, auf einer bestimmten Host-Plattform Object-Code herzustellen, der unter einer anderen CPU-Architektur ausführbar ist. Soweit ist dies nichts anderes, als das, was wir bereits bei der Assemblerprogrammierung gemacht haben.

Die GNU Compiler Collection (GCC) unterstützt nun mehrere Programmierhochsprachen (C, C++, Fortran, Java, etc.; inline assembler ist möglich). Der 'standard GCC' kann durch Patches am Quellcode und Neukompilieren zum Cross-Compiler für andere Zielplattformen erweitert werden. Voraussetzung ist ein funktionierender host-GCC auf dem eigenen PC, der ausnahmslos allen Distributionen beiliegen dürfte. Der Franzose Stephane Carrez stellt auf seiner Homepage [4] einen Port des GCC für den 68hc11 (und den 68hc12) zur Verfügung, der die folgende GCC-Funktionalität integriert:

- GNU C, C++ Compiler (GCC 2.95.2)
- GNU Binutils (as, ld, ar, objcopy, ..., Binutils Version 2.10)
- GNU Debugger (GDB 5.0)
- NEWLIB Bibliotheken (libc, libgloss, libm)

Auf der oben genannten Website stehen ausführliche Download- und Installationsanleitungen zur Verfügung. Es müssen neben den 68hc11-Patches auch die originalen (ungepatchten) GCC-2.92.2, Binutils-2.10 und GDB-5.0 Quellenpakete vorhanden sein, denn bevor wir Cross-Kompilieren können, müssen wir uns zuvor erst den Cross-Compiler übersetzen (Urgh!). Der grobe Ablauf der Installation sieht diese Schritte für GCC, Binutils und GDB vor:
- Entpacken der tar-Quellen
- Anwendung des m68hc1x.diffs-Patches
- Aufruf von `./configure`, `make` und `make install`

Dem Aufruf von configure können optional einige Parameter mit auf den Weg gegeben werden. Über die Option `--prefix=/dir` entscheidet man beispielsweise über das spätere Installationsverzeichnis. Beispiel:

```
./configure --target=m6811-elf --program-prefix=m6811-elf- --prefix=/opt/gcc_68hc11
```

Damit landen Compiler, Linker usw. nach Installation im Verzeichnis `/opt/gcc_68hc11`. Schaut man im Pfad `/opt/gcc_68hc11/bin` (den man in seine PATH-Umgebungsvariable aufnehmen sollte) nach, so fällt einem neben dem `m6811-elf-gcc` und `m6811-elf-ld` auf, dass wir nun über einen weiteren Assembler verfügen, den `m6811-elf-as`. Die ausführbare Datei `m6811-elf-objcopy` besorgt übrigens die Übersetzung des Object-Codes ins S-Record Format, in dem die Programme über seriellen Download in den Mikrocontroller gelangen.

Empfehlenswert ist danach in jedem Fall der Download der Beispiele (Menüpunkt 'Examples'), schon allein, weil praktische Header- und Makefiles beiliegen, auf die man zur eigenen Softwareentwicklung zurückgreifen kann. Die mitgelieferten Beispiele liefern unter anderem einen Taschenrechner und eine Uhr, die über die serielle Schnittstelle bedient werden können. Nach dem Studium der Quelltexte steht der eigenen Softwareentwicklung nichts mehr im Wege. Es sei noch darauf hingewiesen, dass im Verzeichnis des Makefiles der Inhalt einer Datei namens `memory.x`, Teil eines (größeren) Linkerskripts, die Verteilung von Code und Daten auf bestimmte Speicherbereiche, sowie die Position des Stack steuert. Als Anlaufstelle für Fragen und dringende Notfälle existiert eine Mailing-Liste [4].

embedded Java auf acht Bits

Die australische Firma Real Time Computing um den Entwickler Peter Gasparik hat eine Java Virtual Machine entwickelt, die aufgrund ihres geringen Speicherbedarfs für den Einsatz auf Mikrocontrollern prädestiniert ist. SimpleRealTimeJava (simpleRTJ) ist in ANSI C geschrieben und kann mit ausführlicher HTML-Dokumentation von der Homepage der Firma [6] im Quellcode heruntergeladen werden.
Der Einsatz im privaten und im Bildungsbereich ist kostenlos, für gewerbliche Zwecke muss eine Lizenz käuflich erworben werden.

Welche Ausrüstung, neben der Hardware (68hc11-Platine, Netzteil, Kabel), ist zusätzlich gefordert, um mit simpleRTJ herumzuexperimentieren ?

- Host PC, mit einer Java-Entwicklungsumgebung und Runtime Engine (mindestens Java 1.2) [6]

- ANSI C Cross-Compiler für den Mikrocontroller (s.o.)

- Software, um die hc11'er Programme über die serielle Schnittstelle in den µC zu laden (s.o.)

- evtl. ein Oszilloskop; hilft ungemein beim Basteln und Debuggen.

SimpleRealTimeJava ersetzt ein Multitasking-Betriebssystem auf dem Mikrocontroller. Sinn macht das deshalb, weil Multithreading und Thread-Synchronisation fester Bestandteil des Java-Sprachstandards sind (zu Thread-Programmierung in Java vgl. [6]). Die Laufzeitumgebung ist so konzipiert, dass sie verschiedene Java-Threads verwaltet und preemptives Scheduling betreibt. Dazu nutzt sie einen periodischen Interrupt, beim 68hc11 den 'Real Time Interrupt' (RTI) des Prozessors. Bei Auftreten eines RT-Interrupts wird über eine Interrupt Service Routine (ISR) die Funktion vmTimeSlice aufgerufen, die für das Thread Switching verantwortlich ist. Der 68hc11 kann bei Verwendung eines 8 MHz-Quarzes RT-Interrupts im zeitlichen Abstand von ca. 4, 8, 16 oder 32 Millisekunden erzeugen. SimpleRTJ unterstützt die gewohnte Synchronisation mittels Objektmonitoren, durch die ein Thread alleinigen Zugriff auf ein Objekt erhält, d.h. konkurrierenden Threads den Zugriff auf synchronisierte Methoden und Blöcke versperrt. Die Object-Methoden wait() und notify() sind ebenfalls im API enthalten.
Ferner lassen sich von Java aus Event Handler für asynchrone Ereignisse (serielle I/O, Tastatur, ...) einrichten, die durch native Event Handler über Interrupts der MCU informiert werden. Die Latenzzeiten des Systems bewegen sich in der Größenordnung der oben angegebenen RT-Interruptzeiten, also mehrerer (zehn) Millisekunden.

Weitere Highlights sind die Unterstützung von Interfaces, Exceptions, mehrdimensionalen Feldern, Strings/StringBuffer, Fliesskommarechnungen, Software Timern, ein vereinfachtes Java Native Interface (JNI) und nicht zuletzt der Garbage Collector. Die mitgelieferten Pakete `java.lang`, `java.util`, `javax.events` und `javax.memory` decken zwar (natürlich) nicht den gewohnten Java2-Funktionsumfang ab. Sie bilden jedoch einen sinnvollen Grundstock zur ernsthaften Softwareentwicklung für ein eingebettetes System.

SimpleRTJ ist bereits für eine Reihe von Zielprozessoren kompiliert worden. Mit dem 68hc11 benötigt die VM inklusive Threadsupport, Garbage Collector und Startcode, jedoch ohne Fliesskommaunterstützung ca. 20 KB. Die restlichen 44 KB (abzüglich des Stack der VM, der sinnvollerweise unterhalb der Register, von

hexadezimal 0x0FFF an abwärts, angesiedelt ist, und der Register selbst (beim 68hc11A1 memory-mapped von 0x1000 bis 0x103F)) müssen sich der Java-Bytecode der Anwendung und der Heap teilen. Auf dem Heap lagern unter anderem die Thread- und Objekt-Referenztabellen und die dynamisch von der Anwendung erzeugten Felder und Instanzen.

Als Leitfaden für die bevorstehenden Schritte zur Inbetriebnahme von simpleRTJ sollen folgende fünf Punkte dienen. Sie fassen für den 68hc11 zusammen, was ausführlicher in der guten englischen Dokumentation allgemein beschrieben ist.

i) Erzeugen der simpleRTJ-VM als Bibliothek
ii) Schreiben eines geeigneten Startup-Codes für den 68hc11
iii) ClassLinker und Control-Datei
iv) Java Anwendung schreiben, Kompilieren
v) Booten, Upload und Reset

Ein Drama in fünf Akten ? Ran an die Arbeit!

i) In einem erstem Schritt werden wir nun die virtuelle Maschine als Bibliothek im 68hc11-Binärformat generieren. Nach dem Auspacken des zip-Archivs (z.B. nach `/opt/simpleRTJ-1.2.2/simplertj-1.2.2`) müssen dazu einige der Anpassungen in den Header-Dateien an die vorliegende Architektur vorgenommen werden.
In den nun vorhandenen Unterverzeichnissen `simplertj-1.2.2/source` muss man eventuell mittels `mv` die weiteren Unterverzeichnisse (JAVA/LANG, JAVA/UTIL, JAVAX/EVENTS, JAVAX/MEMORY) in kleingeschriebene Unterverzeichnisse gleichen Namens umbenennen. Das gleiche gilt für die Dateien `lib/JAVA.JAR`, `lib/JAVAX.JAR`, sowie für alle `*.h` und `*.c` Dateien im Verzeichnis `vm-l64k`; (Vielen Dank M$!). Am besten kopiert man sich den Ordner vm-l64k als Unterverzeichnis nach `m68hc11-examples-1.0`, wo die C-Beispiele des GCC liegen (s.o.), da man sich dort durch nur kleinere Änderungen die vorhandenen Beispiel-Makefiles leichter zu nutze machen kann. Die Abkürzung l64k steht übrigens für 'linear 64 KB', was unserer Konfiguration entspricht (Es sind auch noch die Speichermodelle 'banked 64 KB' und 'linear 16 MB' vorbereitet.). Nach `cd vm-l64k` ändern wir folgende `#define`-Konstanten in den Dateien `javavm.h` und `jvm.h`:

in javavm.h:
```
#define ENABLE_PRAGMA_PACK  false   /* !!! */
#define ENABLE_UNICODE      false
#define ENABLE_WDT          false
#define LSB_FIRST           false

#if ENABLE_PRAGMA_PACK
  #define _PACKED_
#else
  #define _PACKED_
#endif
```

Der _PACKED_-Modifizierer für Strukturen und Unions kann in unseren Fall ungesetzt bleiben, da der 68hc11 auch auf Integer-Variablen an ungeraden Adressen zugreifen kann.

in jvm.h:

```
#define ENABLE_THREADS            true
#define ENABLE_GC                 true
#define ENABLE_FLOAT              false
#define ENABLE_MULTIANEWARRAY     true
#define ENABLE_STRING_EXCEPTIONS  true
#define ENABLE_TEXT_NOTIFY        false
#define ENABLE_ODDADR_READ        true   /* !!! */
#define ENABLE_SHORT_TIMERS       true
```

Die acht C-Dateien im Verzeichnis können anschließend mit folgendem Kommando kompiliert werden, also z.B. für `j_vm.c`:

```
m6811-elf-gcc -m68hc11 -mshort -msoft-reg-count=12 -fsigned-char -Wall -Os -I. -
c -o j_vm.o j_vm.c
```

Der Parameter `-mshort` bewirkt, dass Integer-Variablen 16 Bit lang sind; die virtuelle Maschine von simpleRTJ erwartet dies. Deshalb wiederum werden während des Kompilierens haufenweise Warnings erzeugt, die von den PUT_REF und GET_REF Präprozessormakros in `jvm.h` ausgelöst werden. Uns kümmert das nicht weiter.

Schließlich werden die object-Dateien zu einer Bibliothek namens simpleRTJ verpackt:

```
m6811-elf-ar r simpleRTJlib.a j_vm.o j_float.o j_gc.o j_lang.o j_mem.o
j_thread.o j_vars.o j_bcode.o
```

ii) Der Startup-Code (ebenfalls in C) hat zur Aufgabe, den Mikrocontroller auf die Ausführung der virtuellen Laufzeitmaschine vorzubereiten. Er richtet den Handler für den RTI ein, Interrupts bleiben aber noch maskiert. Es wird Speicher für die structure `vm_config_t` zugewiesen. Mittels dieser Struktur übergibt der Programmierer wichtige Parameter des Systems an die VM. Neben Informationen über das zeitliche Scheduling, die Speicheradressen von Java-Applikation (s.u.) und Heap, der Referenztabelle, die die Zeiger auf die nativen Funktionen enthält, kann optional ein Zeiger auf die Funktion `usrNotify` übergeben werden. Diese Funktion dient der VM zu Benachrichtigung über aufgetretene Fehler und Ausnahmen zur Laufzeit. Die Funktion wird aber auch aufgerufen, wenn die VM ihrerseits einen Initialisierungsprozess abgeschlossen hat und die Java-Anwendung startet. Es empfiehlt sich auf einer 'langsamen' MCU wie dem 68hc11 die maskierbaren Interrupts (auch RTI) erst freizugeben nach Aufruf von `usrNotify` mit Argument subcode=`NOTIFY_VM_START`. Näheres dazu findet man in der Dokumentation. Komplette (Make-)Files können von der Homepage des Autors [1] downgeloaded werden.

Eine Datei mit dem verheißungsvollen Namen vm.c (und die Datei lookup.c, s.u.) werden mit den Befehlen:

```
m6811-elf-gcc -m68hc11  -mshort -msoft-reg-count=12 -fsigned-char -Wall -Os -I.
-I./../include -I../vm-l64k -c -o lookup.o lookup.c
```

```
m6811-elf-gcc -m68hc11  -mshort -msoft-reg-count=12 -fsigned-char -Wall -Os -I.
-I./../include -I../vm-l64k -c -o vm.o vm.c
```

```
m6811-elf-gcc -m68hc11 -mshort -msoft-reg-count=12 -fsigned-char -Wl,-m,m68hc11elfb -o vm.elf lookup.o vm.o ./../lib/libutil.a ./../vm-l64k/simpleRTJlib.a

m6811-elf-objcopy --output-target=srec --only-section=.text --only-section=.rodata --only-section=.vectors vm.elf vm.s19
```

übersetzt, mit der in i) erzeugten Bibliothek `simpleRTJlib.a` gelinkt, und das Resultat schließlich in das S-Record Format konvertiert. Das Ergebnis ist die Datei `vm.s19`, die fertige VM für den 68hc11.
Der schwierigste (?) Teil ist damit überstanden.

iii) Bevor wir uns endlich daran machen eine Java-Anwendung zu schreiben, sind einige Erläuterungen bzgl. der Handhabung eines Java-Bytecodes durch die VM angebracht. Vom Desktop-PC ist man es gewohnt, eine Java-Anwendung nach dem Kompilieren durch den Aufruf '`java ClassName`' zu starten, wobei die Klasse ClassName die main-Methode enthält. Das geht bei der Mikrocontroller-VM nicht. Das API beinhaltet weder die Klasse `java.lang.Class` noch `java.lang.ClassLoader`, noch könnten Klassen anderweitig dynamisch erzeugt und nachgeladen werden. Der gesamten Java-Applikation, d.h. der *.class-Files, nimmt sich stattdessen auf der Entwicklungsplattform der ClassLinker (`ClassLinker.jar`) an, welcher neben der VM und dem API dritte tragende Säule des simpleRTJ-Paketes ist. Der ClassLinker durchsucht alle Klassen, die zu einer Anwendung gehören und ersetzt darin die symbolischen Referenzen durch indizierte Referenzen; er fasst alle am Programm beteiligten Klassen zu einer Einheit zusammen und macht ein dynamisches und aufwändiges Linken zur Laufzeit überflüssig. Der ClassLinker gibt folgende Dateien aus: Die gelinkte Anwendung im Motorola S-Record oder Intel hex-Format, Map- und Listingdateien, C- und Assemblerdateien mit Funktionsrümpfen für die nativen Methoden. Die Assemblerdatei (`native/lookup.asm`) entspricht praktischerweise der vom as6811 verarbeiteten Syntax. Die Arbeit des ClassLinkers wird gesteuert durch die Datei `ClassName.control` (vgl. `simplertj-1.2.2/MyProject/MyApp.control`). In ihr werden eine Reihe von Optionen gesetzt. Ohne hier allzu sehr auf die Details eingehen zu wollen, seien repräsentativ die Einträge für die geplante Beispielanwendnug aufgelistet:

```
LinkClassPath=.:/opt/simpleRTJ-1.2.2/simplertj-1.2.2/lib/java.jar:/opt/simpleRTJ-1.2.2/simplertj-1.2.2/lib/javax.jar
MapFile=Y
MapFileHtml=N
ListFile=
ListFileHtml=Y
QuietMode=N
NativeLookupFile=native/lookup.c
MemoryModel=LINEAR_64K
AppID=MyApp.01
AppDesc=My Application
OutputFormat=Motorola,s19
BaseAddr=0x1040
ByteOrdering=MSB
```

Wichtig ist in erster Linie, dass die Einstellung 'ByteOrdering' vom Vorgabewert LSB auf MSB umgestellt wird, wegen der Verwendung von BigEndian auf dem Motorola Prozessor. Zweitens ist darauf zu achten, 'BaseAddr' auf die Adresse zu stellen, unter der die VM die Java-Anwendung erwartet. Für die geplante erste Java-Anwendung richtet man sich wieder auf der Ebene der C-Beispiele einen Ordner ein, z.B. `myProj01`, und erstellt dort `JToast.control`.

iv) Zu einer heißen Tasse guten Kaffees (*wink*) gehört morgens beim Frühstück natürlich eine Scheibe wohlgebräunten Toasts. Allein, man bekommt die Augen kaum auf und eh man sich versieht ist der Toast leider angebrannt. Wie schön wäre es, wenn der Toaster dem User mitteilen würde, wie weit der Bräunungsprozess der eben versenkten Toastscheibe schon gediehen ist. Im Bedarfsfall sollte geplagter Benutzer eingreifen und den Toast vorzeitig, d.h. vor Erreichen der maximalen Backzeit auswerfen können. Soweit die Forderungen. Abbildung [JToast.java im Anhang] zeigt das zugehörige Programmlisting.

Die beiden folgenden Schritte aus der ausführbaren Datei `makeJ` übersetzen den Code und aktivieren den ClassLinker:

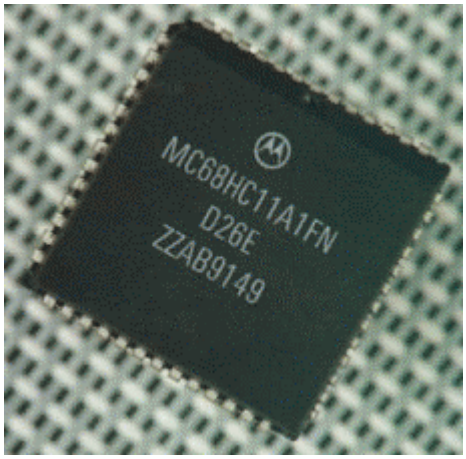
```
#! /bin/sh

echo Compiling...
# use IBM jikes
jikes                -classpath                .:/opt/simpleRTJ-1.2.2/simplertj-
1.2.2/lib/java.jar:/opt/simpleRTJ-1.2.2/simplertj-1.2.2/lib/javax.jar *.java

echo Linking...
java                  -classpath                  /opt/simpleRTJ-1.2.2/simplertj-
1.2.2/lib/ClassLinker.jar:/opt/simpleRTJ-1.2.2/simplertj-1.2.2/lib ClassLinker -
f JToast
```

Nach erfolgreichem Übersetzen und *Linken* sollte die Datei JToast.s19 im Verzeichnis vorhanden sein. `Jikes` (V1.11) hat sich als zuverlässige Alternative zum Sun Compiler javac erwiesen.

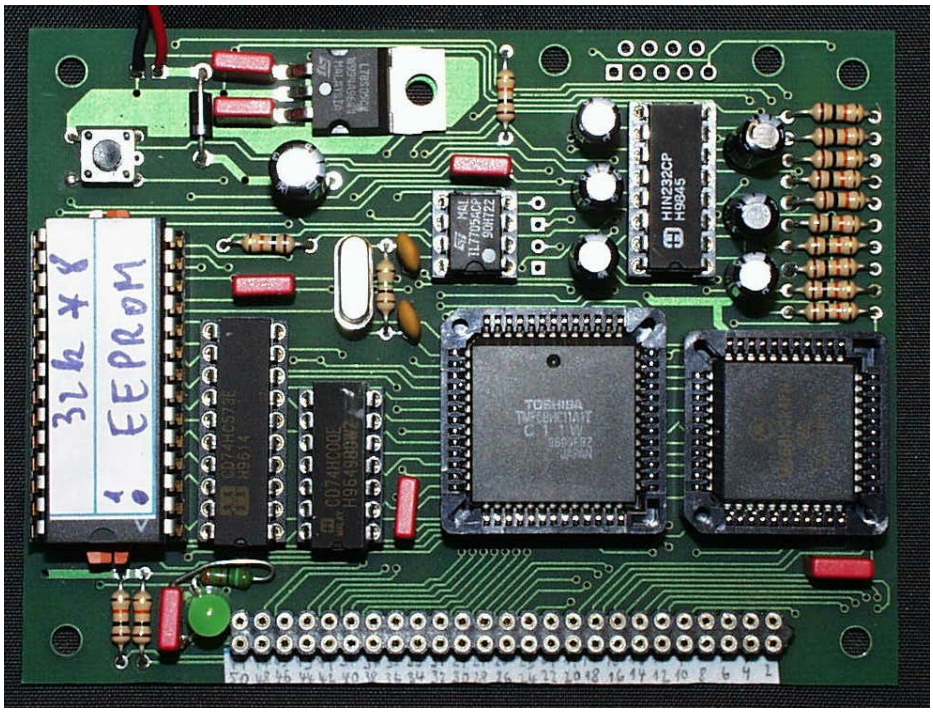
v) Belohnung dieser Prozeduren sind die Dateien `JToast.s19`, `vm.s19` und eine 'Reset-Datei' nach dem im Abschnitt 1 vorgestellten Schema, `reset.s19`. Diese Dateien werden in den Mikrocontroller geladen und nach einem Reset der MCU, der die Programmausführung an die Startadresse des Startup-Codes springen lässt, sollte die VM hochfahren und mit der Abarbeitung des Bytecodes beginnen.
Als Beleg dafür, dass das oben vorgestellte Programm tatsächlich das tut, was man von ihm erwartet, zeigen die Abbildungen [lcd1.jpg, lcd2.jpg, lcdFertig.jpg] einige Screenshots des angeschlossenen LC-Displays. Guten Appetit!

5) Dies mag nach viel Spielerei aussehen. Wenn man jedoch bedenkt, welche Zukunft z.B. den SmartCards und JavaCards vorausgesagt wird, in denen oft auch 'nur kleine' 8-Bit Mikrocontroller tuckern, wird die Sache schon interessanter. Oder wie wäre es mit einem kleinen Webserver, der irgendwo im Netz Messungen vornimmt, und in HTML formatiert ausgibt (WuT, [7]) ?
Ansonsten macht es einfach Spaß, sich mit einer 'überschaubaren VM' zu beschäftigen, die zudem noch im Sourcecode vorliegt und recht einfach auf andere Mikrocontroller portierbar sein dürfte.

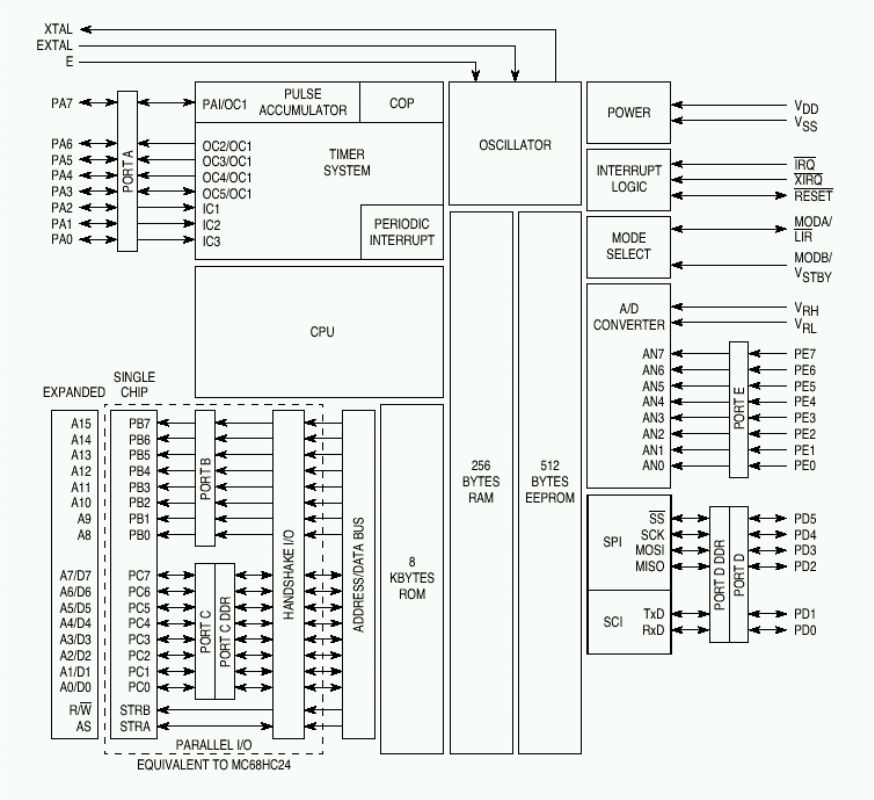
Abbildungen:



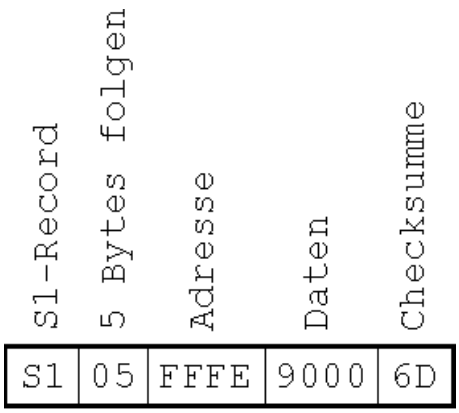
µC: (chip.gif)



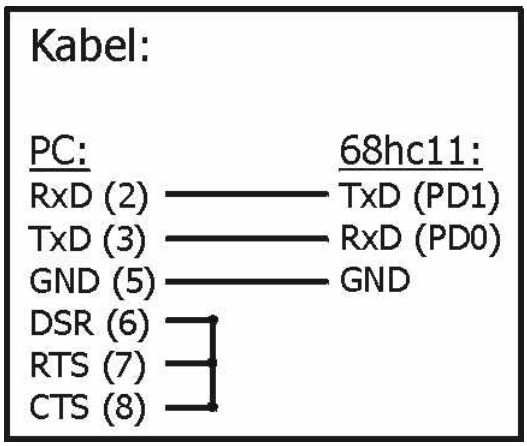
µC-Platine: Abmessungen ½Europakarte 8 cm x 10 cm (platine.jpg)



µC-Funktionsblöcke: (blockDiagramm.gif)

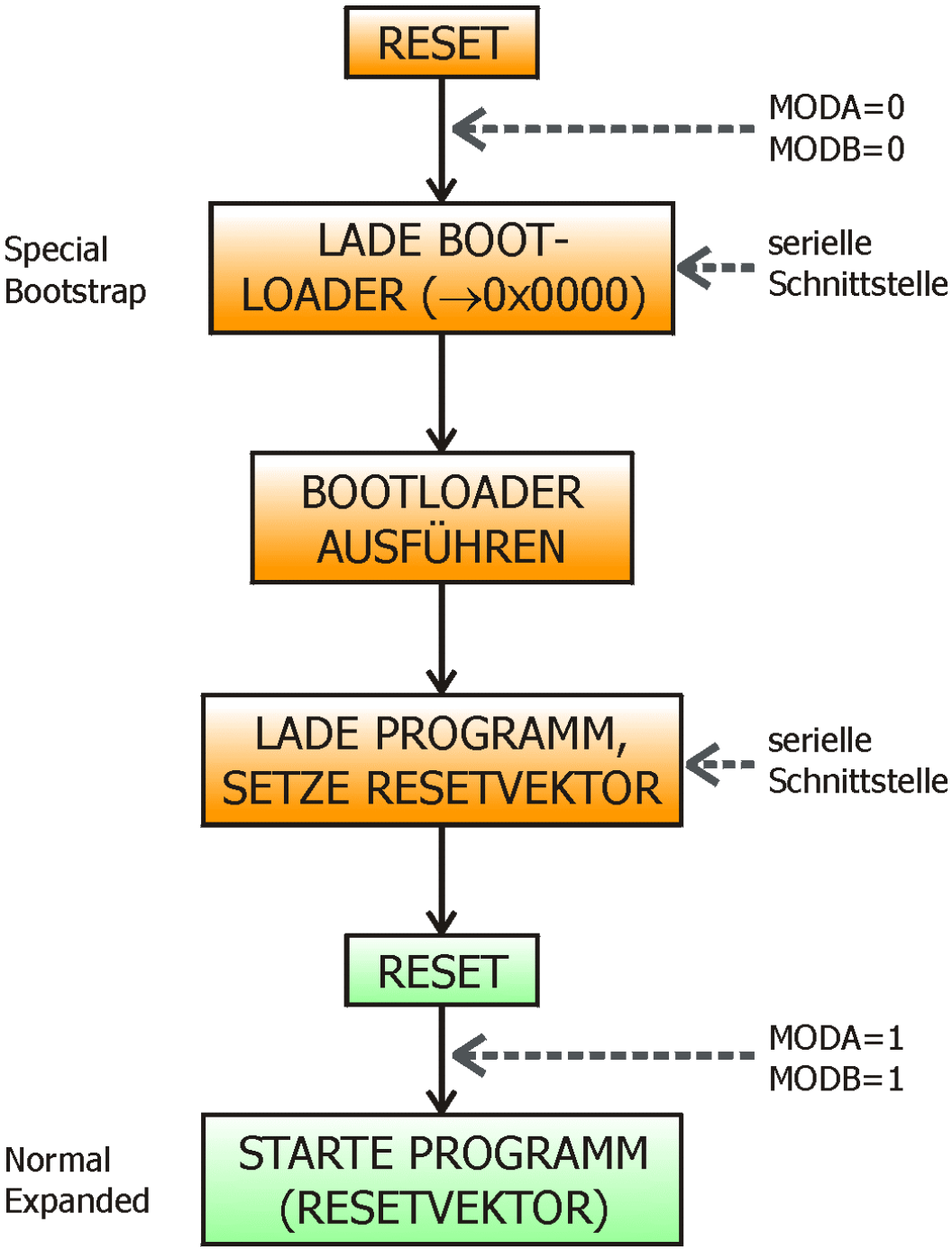


S-Record: (s1record.gif)

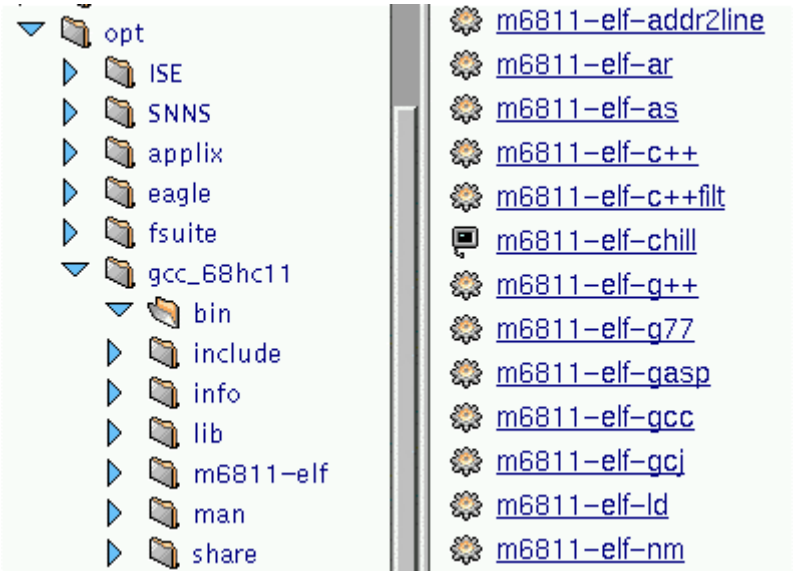


Kabel: (kabel.gif)

WebRing: (webRing.jpg)

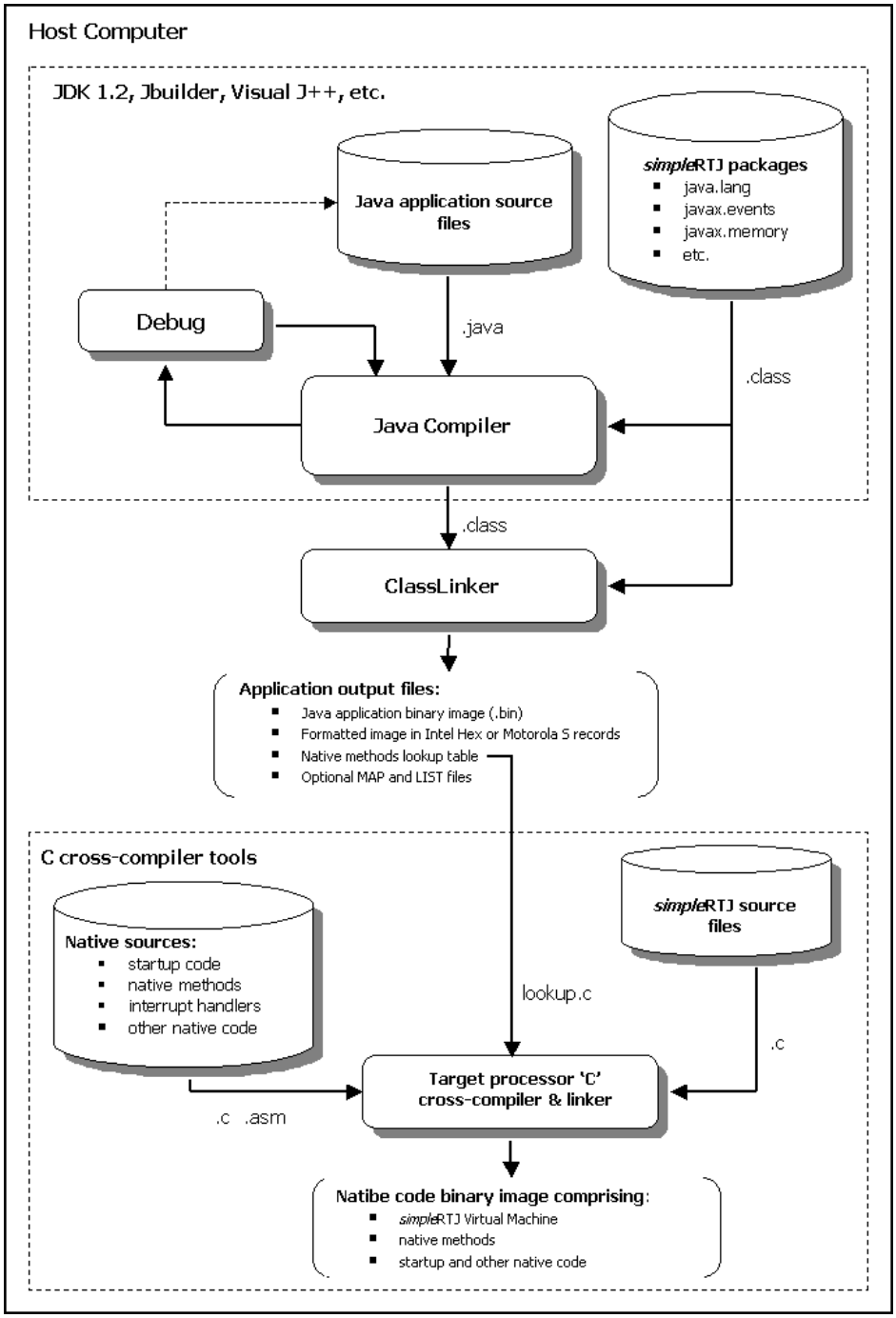


BootKonzept: (booting.gif)

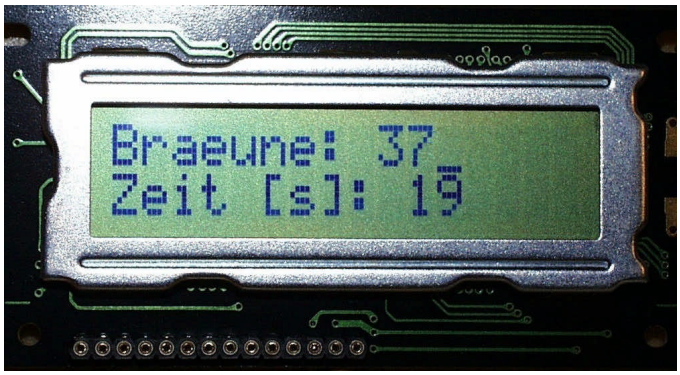


gccDir: (fileStrukturGCC.gif)

projDir: (fileStrukturProj.gif)



simpleRTJ: building (develop.gif, develop-target.gif)



LCD: (lcd1.jpg, lcd2.jpg, lcdFertig.jpg)

Links:

[1] M68HC11:
Motorola Homepage, Links zu Doku und Mailing List:
http://www.mot-sps.com, bzw. http://www.mcu.motsps.com/documentation/index.html
Web-Ring: http://www.spacegroove.com/telic/WebRing/hc11.htm
Homepage des Autors (Bootloader 'hcterm', VM-Startcode, Beispiele): http://www.nnTec.de

[2] Newsgroups (Posting des 68hc11 microcontroller FAQ):
news://comp.realtime
news://comp.robotics
news://comp.sys.m68k

[3] Assembler:
ASXXXX: http://shop-pdp.kent.edu/ashtml/asxxxx.htm
GNU AS: siehe GCC
DASM von Matt Dillon: http://members.home.com/rcolbert1/dasm.htm
AS, µC-Programmierung: Linux Magazin 10/97,
http://www.linux-magazin.de/ausgabe/1997/10/Miccon/miccon.html
Beschreibung des s19-Formats: http://www.cs.net/lucid/moto.htm

[4] GCC et al:
GCC: http://gcc.gnu.org
68hc11-Port (patch für gcc, binutils, gdb): http://home.worldnet.fr/~stcarrez/m68hc11_port.html
Mailing-List: http://www.egrougs.com/group/gnu-m68hc11
GNU Cross-Compiler zum Bestellen: http://www.microcross.com

[5] Real Time OS:
Überblick: http://www.ednmag.com/ednmag/extras/embeddedtools/rtosdisplay.asp
µC/OS-I/II: http://www.micrium.com, http://www.ucos-ii.com, lesenswert: Application Note 'The 10-Minute Guide to RTOS'
Real Time Java: http://www.rtj.org

[6] Java, simpleRealTimeJava:
Blackdown Linux JDK: http://www.blackdown.org
Kaffe VM: http://www.transvirtual.com
Sun JDK: http://www.sun.com
Jikes: http://www.alphaworks.ibm.com, IBM JDK: http://www.ibm.com
Visual Age Micro Edition, J9 VM: http://www.ibm.com/software/ad/embedded
Threads: Linux Magazin 12/97, http://www.linux-magazin.de/ausgabe/1997/12/Threads/threads.html
RealTimeComputing (simpleRealTimeJava, Bootloader 'Appload11'): http://www.rtjcom.com

[7] Boards und IC's:
Elektronikladen: http://elektronikladen.de/mops11.html

MIT Handy Board: http://el.www.media.mit.edu/projects/handyboard, http://www.handyboard.com
Nierhauve: http://www.nierhauve.de
Diverse Links: http://www.ezl.com/~rsch, http://www.multimania.com/ybnet/
CadSoft (Platinen Layout Eagle): http://www.cadsoft.de
Simons Electronic: http://www.simons-electronic.de
Segor Elektronik: http://www.segor.de
WuT (Com-Server): http://www.wut.de

[8] Simulatoren (MS-Windows):
Wookie: http://www.msoe.edu/eecs/ce/ceb/resources
THRSIM11: http://www.thrijswijk.nl/~bd/thrsim11/thrsim11.htm

[9] Literatur:
M68HC11RM/AD ('pink book') bei Motorola, diverse technical data books
Michael Rose, 'Mikroprozessor 68HC11' (inkl. Platine), Hüthig Verlag Heidelberg (1994), ISBN 3-7785-2277-9
Joseph L. Jones, Anita M. Flynn, 'Mobile Roboter', Addison-Wesley Verlag Bonn (1996), ISBN 3-89319-855-5
Michael Barr 'Programming Embedded Systems in C and C++', O'Reilly (1999); ISBN: 1565923545
Jean Labrosse (Micrium) 'MicroC/OS-II', CMP Books (1998), ISBN 0879305436
Linux Magazin 8/2000: LEGO Mindstorms Roboter Programmierung
Linux Magazin 2/2001: serielles LC-Display

JToast - Quellcode einer Beispielanwendung:

```
/**
 * Title:       JToast - a Java based Toaster<br>
 * Description: Test for simpleRealTimeJava, http://www.rtjcom.com<br>
 * Copyright:   Copyright (c) 2001, Ingo Beckmann. This is free software!<br>
 * Company:     nnTec, http://www.nnTec.de<br>
 * @author      Ingo Beckmann
 * @version     0.1
 */

import javax.events.*;

interface DisplayDevice {
    public void print(int line, String s);
}

public class JToast {

    public static int adcChannel = 1;
    public static boolean userEject = false;

    static void main() {
        LCD lcd = new LCD();    // get the LCD initialized, lcd can now be locked on
        lcd.print(1, "LCD OK");
        adcInit(adcChannel);              // analog-to-digital converter
        Thread braeunungsThread = new BraeunungsThread(1, lcd); // thread 1, printing on line 1
        // prepare manual eject event
        AsyncEvent ejectEvent = new AsyncEvent();
        try {
            ejectEvent.addHandler(new EjectHandler(lcd, braeunungsThread));
            Events events = new Events();
            events.addEvent(ejectEvent, 1);  // index zero is reserved for Timer asynch. event
            events.start();
        } catch (Exception x) { /* ignore */ }
        // start toasting:
        braeunungsThread.start(); // thread 1, printing on line 1
        new StoppUhrThread(2, lcd, 1000).start(); // thread 2, printing on line 2
    }

    /**
     * native methods
     */
    public static native void adcInit(int channel);
    public static native synchronized int adcGetResult(int channel);
    public static native void eject();
} // end class



/**
 * Eject Eventhandler
 */
class EjectHandler extends AsyncEventHandler {
    private DisplayDevice out;
    private Thread threadToWaitFor;

    EjectHandler (DisplayDevice out, Thread thread) {
```

```
        this.out = out;
        this.threadToWaitFor = thread;
    }
    public void handleAsyncEvent() {
        Thread dt = new Thread(new DummyThread());
        dt.setDaemon(false);
        dt.start();
    }
    // inner class for a dummy Thread
    class DummyThread implements Runnable {
        public void run() {
            JToast.userEject = true;
            // wait for BraeunungsThread to die...
            try {
                threadToWaitFor.join();
            } catch (InterruptedException iX) { /* do nothing */ }
            synchronized (out) {
                out.print(1, "OK, raus damit.");
            }
            JToast.eject();
        }
    }
}

/**
 * LCD
 */
class LCD implements DisplayDevice {
    LCD () {
        init();
    }
    /**
     * LCD native methods
     */
    public native void init();
    public native synchronized void print(int line, String s);
} // class

/**
 * Messung: Braeunung des Toasts
 */
class BraeunungsThread extends Thread {
    private DisplayDevice out;
    private int number, braeune = -1, newBraeune;
    private final int maxBraeune = 42;

    BraeunungsThread (int number, DisplayDevice out) {
        this.number = number;
        this.out = out;
    }
    public void run () {
        while( !JToast.userEject ) {
            newBraeune = JToast.adcGetResult(JToast.adcChannel);
            if (newBraeune > braeune) {
                braeune = newBraeune;
                synchronized (out) {
                    out.print(number, "Braeune: "+Integer.toString(braeune));
                }
            }
            if (braeune >= maxBraeune) {
                synchronized (out) {
```

```
                    out.print(number, "Toast fertig !!!");
                }
                JToast.eject();
                try {
                    Thread.sleep(10000);
                } catch (InterruptedException iX) {
                    out.print(number, "X:Interrupted"); // !!!: out not synch'd
                }
                break;
            }
            yield();
        } // while
    } // run
} // class

/**
 * Stoppuhr
 */
class StoppUhrThread extends Thread {
    private DisplayDevice out;
    private int number, sleep, time;

    StoppUhrThread (int number, DisplayDevice out, int sleep) {
        this.number = number;
        this.out = out;
        this.sleep = sleep;
        this.setDaemon(true);
    }
    public void run () {
        while(true) {
            time = System.currentTimeMillis();
            synchronized (out) {
                out.print(number, "Zeit [s]: "+Integer.toString(time/1000));
            }
            try {
                Thread.sleep(sleep);
            } catch (InterruptedException iX) {
                out.print(number, "X:Interrupted"); // !!!: out not synch'd
            }
        } // while
    } // run
} // class
```