eCos Applikations-Entwicklung

Andreas Bürgel[1]

http://www.andreas-buergel.de

7. April 2003

[1] Mail: andreas@andreas-buergel.de

Version $Id: eCos-Applikations-Entwicklung.tex,v 1.2 2003/04/07 20:45:10 andreas Exp $

eCos™ ist ein eingetragenes Markenzeichen von Red Hat, Inc.
RedHat®, RedBoot™, GNUPro®, Cygwin™ sind eingetragene Markenzeichen von Red Hat, Inc.
Linux® ist ein eingetragenes Markenzeichen von Linus Torvalds
UNIX® ist ein eingetragenes Markenzeichen von The Open Group
Microsoft®, Windows®, Windows NT®, Windows 95®, Windows 98®, Windows 2000® sind eingetragene Markenzeichen der Microsoft Corporation
ARM® ist ein eingetragenes Markenzeichen von ARM Ltd.
MIPS® ist ein eingetragenes Markenzeichen von MIPS, Inc.
Intel®, StrongARM® sind eingetragene Markenzeichen der Intel Corporation
ATMEL® und AT91® sind eingetragene Markenzeichen der Atmel Corporation
Alchemy SemiConductor®, Au1000®, Au1500® sind eingetragene Markenzeichen von Alchemy Semiconductor Inc.
HyperStone® ist ein eingetragenes Markenzeichen der HyperStone AG
Alle in dieser Aufführung nicht genannten Markenzeichen sind ebenfalls eingetragene Markenzeichen ihrer Besitzer. Der Autor will sich diese auf keinen Fall zu eigen machen.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Über dieses Dokument . . . 1
1.2 Begriffsklärung . . . 1
1.3 Was ist eCos? . . . 2
1.3.1 Was eCos nicht ist . . . 2
1.3.2 Warum doch eCos? . . . 2
1.3.3 Das eCos-Applikations-Modell . . . 2

2 Die eCos-Bibliothek libtarget.a erzeugen **3**
2.1 eCos-Quelldateien besorgen . . . 3
2.2 Konfiguration der Laufzeit-Bibliothek . . . 3
2.2.1 libtarget.a unter Linux erzeugen . . . 4
2.2.2 libtarget.a unter Windows erzeugen . . . 6

3 Applikationen schreiben **7**
3.1 Grundsätzliches . . . 7
3.1.1 Applikationen kompilieren und linken . . . 7
3.1.2 Programmier-Schnittstellen/API . . . 8
3.1.3 Einschränkungen bei der Benutzung von C++ . . . 8
3.2 Hello world! . . . 8
3.3 Zwei Threads . . . 9
3.4 Speicher verwalten . . . 12
3.5 Kommunikation zwischen Threads . . . 16
3.6 Synchronisation . . . 22
3.6.1 Mutex . . . 22
3.6.2 Semaphor . . . 26
3.6.3 Condition-Variable . . . 29
3.6.4 Flags . . . 29
3.7 Zeit, Uhren und Alarme . . . 29
3.7.1 Uhrzeit abfragen . . . 29
3.7.2 Alarme benutzen . . . 29
3.8 Einen Interrupt-Handler schreiben . . . 29
3.9 Gerätetreiber benutzen . . . 35

A Das eCos Komponenten- und Konfigurations-Konzept **39**

B ecosconfig **41**
B.1 Linux . . . 41

C Probleme und Lösungen **43**
C.1 Die Applikation wird beendet, wenn alle User-Threads blockiert sind . . . 43
C.2 Wie man die seriellen Schnittstellen aktiviert . . . 43
C.3 Download vom Host- zum Ziel-Rechner bricht ab . . . 43

D Einen ROM-Monitor kompilieren **45**

E RedBoot **47**

Kapitel 1

Einleitung

1.1 Über dieses Dokument

Dieses Dokument ist ein Fragment. Es erhebt z.Z. keinen Anspruch auf Vollständigkeit und Richtigkeit. Der Autor übernimmt keinerlei Haftung für Schäden, die durch Anwendung der in diesem Dokument enthaltenen Informationen entstanden sind.

Sollte ein Leser dieses Dokuments auf die glorreiche Idee kommen, die Steuerungs-Software für die Speisewasserpumpen des Primär-Kreislaufs eines Atomkraftwerks umzuschreiben und damit ein mehr oder weniger zivilisiertes Land von der Weltkarte entfernen, so ist das nicht die Schuld des Autors. Der Autor übernimmt auf keinen Fall irgendwelche Haftung.

Dieses Dokument darf frei benutzt und weiter verbreitet werden. Die kommerzielle Weiterverbreitung ist jedoch untersagt. D.h. man darf mit Hilfe der Informationen in diesem Dokument Produkte entwickeln und diese verkaufen ohne für das Dokument zu bezahlen. Es ist jedoch nicht gestattet für die Weiterverbreitung dieses Dokuments Geld zu verlangen. Ebenso ist die Veränderung des Dokuments sowie die Verwendung des Dokuments in eigenen Werken, dies betrifft auch Auszüge des Dokuments, nur mit schriftlicher Einwilligung des Autors gestattet.

Der Autor behält sich vor die Nutzungsbedingungen ohne vorherige Bekanntmachung jederzeit zu ändern.

1.2 Begriffsklärung

Als *binär* oder als *Binärdatei* wird hier alles verstanden, was Produkt eines Übersetzer, d.h. eines Assemblers oder eines Compilers, ist. *Executable* ist eine ausführbare Applikation für eine bestimmte Hardware und letztlich auch eine Binär-Datei. Eine *Objekt-Datei* ist Ergebnis des Übersetzungs-Vorgangs einer einzelnen Quell-Datei und damit ebenfalls binär. Das, was auf .o endet ist meist eine Objekt-Datei. Eine *Bibliothek* ist eine Zusammenfassung mehrerer logisch zusammen gehörender Objekt-Dateien.

Als *Host* wird der Rechner bezeichnet, auf dem die Entwicklung stattfindet bzw. auf dem der Debugger läuft. Als *Target* oder *Ziel*-Hardware wird der Rechner bezeichnet, für den die Applikation entwickelt wird.

Als *eCos-Repository* wird der Verzeichnisbaum bezeichnet in dem die Quell-Dateien von eCos selbst, als da wären Kernel-, HAL- und Geräte-Treiber-Quellen, liegen. Als *(eCos-)Build-Verzeichnis* wird das Verzeichnis bezeichnet in dem die Konfigurationsdatei ecos.ecc und der Verzeichnisbaum der Header- und Binär-Dateien der eCos-Laufzeit-Bibliothek bzw. eines der auf eCos basierenden ROM-Monitore liegen.

Ein in spitze Klammern gefasster <BEGRIFF> ist ein Platzhalter, z.B. für einen Namen oder einen Programm-Parameter.

Alle Dateinamen und Benutzer-Eingaben werden serifenlos gedruckt. In `Schreibmaschinenschrift` werden Programm- und Skript-Dateien, sowie Funktions-Namen und teilweise auch Ausgaben von Programmen gedruckt.

1.3 Was ist eCos?

1.3.1 Was eCos nicht ist

eCos ist - auch wenn es von der Firma RedHat stammt - kein Linux und auch kein Unix. Es ist kein interaktives System und bietet *keine Benutzer-Schnittstelle* wie Shell oder grafische Benutzer-Oberfläche[1] an. Es unterstützt nicht mehr als einen *einzigen Prozess*, kennt *keinen Speicher-Schutz* zwischen Betriebssystem und Applikation und es kennt auch keine virtuelle Speicher-Verwaltung.

1.3.2 Warum doch eCos?

Der Zielmarkt von eCos sind *tief eingebettete Systeme* und *Echtzeit-Systeme*, die keine Benutzer-Interaktion benötigen, bzw. deren Benutzer-Interaktion auf dem Niveau „Mensch vs. Getränkeautomat“ liegt. Speicherschutz ist für derartige Anwendungsfälle wenig relevant, da ein stabil weiterlaufendes Betriebssystem mit abgestürzter Applikation sinnlos ist. Ebenso kommt man bei tief eingebetteten Systemen normalerweise mit einer einzelnen (Multithreaded-)Applikation aus. Der Verzicht auf eine strikte Trennung zwischen Be triebssystem und Applikation verkleinert den Overhead für den Aufruf einer Betriebssystem-Funktion drastisch, da ein einfacher Funktions-Aufruf weniger Zeit verbraucht als ein Funktions-Aufruf über Software-Interrupt und Dispatcher.
Großer Wert wurde beim Design von eCos auf die Abstraktion zwischen der Hardware und dem Betriebssystem gelegt. Sämtliche Betriebssystem-Funktionen setzen auf einer Abstraktionsschicht - HAL genannt - auf. Dies ermöglicht es sowohl eCos selbst, als auch Applikationen einfach auf andere Hardware-Plattformen zu portieren. Abhängig von der Applikation *kann* es sein, dass eine Applikation durch einfaches Rekompilieren auf eine völlig andere Hardware-Plattform übertragen werden kann.

1.3.3 Das eCos-Applikations-Modell

Eine fertig kompilierte und gelinkte eCos-Applikation ist ein monolithischer Block bestehend aus dem eigentlichen Applikations-Code, dem Betriebssystem, Gerätetreibern und optionalen zusätzlichen Daten, z.B. FPGA-Programmen.
Betriebssystem und Gerätetreiber sind hierbei zu der (statischen) Bibliothek libtarget.a zusammen gefasst, die zur Applikation gelinkt wird. Dies steht im Gegensatz zu einer z.B. Linux-Applikation, die ja selbst keinen Kernel-Code und keinen libc-Code enthält[2].

[1] Das stimmt nicht ganz. Für den iPaq-Port gibt es Microwindows Unterstützung.
[2] Im Falle, dass die Linux-Applikation statisch mit der libc gelinkt wird, enthält die Applikation natürlich auch Bibliotheks-Code.

Kapitel 2

Die eCos-Bibliothek libtarget.a erzeugen

2.1 eCos-Quelldateien besorgen

Die aktuelle eCos-Version (Release) ist auf dem RedHat FTP-Server unter ftp://sources.redhat.com/pub/ecos in diversen Installations-Paket-Formaten zu finden. Leider gibt RedHat nur recht sporadisch ein neues Release frei, so dass die empfohlene Methode einen aktuellen eCos-Stand zu bekommen die ist, die aktuelle Entwickler-Version vom RedHat-CVS-Server zu laden (Mehr zu CVS findet sich bei [6]). Leider ist bei der Entwickler-Version nicht davon auszugehen, dass sie in allen Aspekten funktioniert. Hardware-unabhängiger Code und der Basis-Code diverser Architekturen, insbesondere für ARM, haben jedoch eine hinreichende Stabilität erreicht und werden nur noch selten verändert.
Kontaktiert man den CVS-Server erstmalig, so muss man sich zunächst einmal mittels
cvs -d :pserver:anoncvs@sources.redhat.com:/cvs/ecos login
und dem Passwort *anoncvs* dort anmelden. Die eCos Quelldateien lädt man danach mittels
cvs -d :pserver:anoncvs@sources.redhat.com:/cvs/ecos -z 9 co -P ecos

Hat man erst einmal einen eCos-Quelldateibaum, so kann man diesen jederzeit mittels
cvs -z 9 update -d -P
aktualisieren indem man in das Verzeichnis, in dem der Quell-Datei-Baum seine Wurzel hat wechselt. Will man nur einen Teilbaum aktualisieren, so wechselt man ins Wurzel-Verzeichnis des Teilbaums bevor man den CVS-Update-Befehl eingibt.

2.2 Konfiguration der Laufzeit-Bibliothek

eCos ist extrem konfigurierbar und läßt sich auf die speziellen Bedürfnisse der Ziel-Applikation zurecht schneidern. Wenn z.B. eine Single-Thread-Applikation ausreicht, so baut man eine Laufzeit-Bibliothek ohne Scheduler-Code. Dies wird möglich durch die Aufteilung in Komponenten bzw. Pakete. Die Konfiguration erfolgt unter Windows durch das Programm ecosconfig. Das Ergebnis der Konfigurations-Anstrengungen ist eine Konfigurations-Datei, die ecos.ecc heißt bzw. ein Dateibaum mit dazugehörigem Makefile, aus dem libtarget.a kompiliert wird.
ecosconfig ist als fertig kompilierte Binär-Datei für Windows und Linux unter ftp://sources.redhat.com/pub/ecos/anoncvs zu finden.
Eine wichtige Konfigurations-Eigenschaft ist der Startup-Typ. Eine Applikation kann für drei verschiedene Startup-Typen kompiliert und gelinkt werden. Eine wichtige Rolle hierbei spielen Linker-Skripte, da das Speicher-Layout der Applikation abhängig vom Startup-Typ ist. Für die unterstützten Startup-Typen eines eCos-Ports gibt es auch jeweils ein eigenes Linker-Skript[8].

- **ROM-Startup**
  Die Applikation wird ab der physikalischen Adresse Null über JTAG oder ein Programmiergerät in den nicht-flüchtigen Speicher des Targets, in der Regel ist dies Flash-Speicher, programmiert. Vor

dem Start der Applikation läuft Hardware-Initialisierungs-Code ab, der entsprechend hinzu gelinkt wurde. Die Applikation läuft anschließend aus dem ROM, was unter Umständen recht langsam ist.

- **ROMRAM-Startup**
  Die Applikation wird wie beim ROM-Startup in den nicht-flüchtigen Speicher programmiert. In einer frühen Phase der Hardware-Initialisierung kopiert sie sich jedoch ins RAM, so dass sie später aus dem RAM läuft. Dieser Startup-Typ ist nicht für alle von eCos unterstüzten Plattformen implementiert.

- **RAM-Startup**
  In das ROM wurde ein ROM-Monitor wie ARM-Angel, RedHat-RedBoot oder der GDB-Stub[1] programmiert. Die Applikation wird über eine serielle Schnittstelle wie RS-232 oder eine Netzwerk-Verbindung vom Host-Rechner herunter geladen, vom ROM-Monitor empfangen und ins Target-RAM geladen. Nach der Übertragung wird die Applikation aus dem RAM heraus gestartet. Der ROM-Monitor übernimmt hierbei sämtliche Hardware-Initialisierungen.

RAM-Startup wird in der Zeit des Entwickelns und Testens verwendet, während ROMRAM-Startup - und vielleicht auch ROM-Startup - für Prototypen oder die Serie gedacht ist.

2.2.1 libtarget.a unter Linux erzeugen

Unter Linux ist ecosconfig ein Kommandozeilen-Programm, mit dem die Konfigurationsdatei ecos.ecc nicht editierbar ist - abgesehen vom Hinzufügen bzw. Entfernen von Komponenten oder dem Auflösen von Konfigurations-Konflikten. Zum Editieren von ecos.ecc kann man einen normalen Text-Editor verwenden[2]. Im folgenden Beispiel wird eine Laufzeitbibliothek libtarget.a für ein StrongARM 1110 Board erzeugt. Der eCos-Quell-Datei-Baum befindet sich unterhalb des Verzeichnisses /opt/ecos/ecos. In diesem Unterverzeichnis befinden sich die Verzeichnisse host, packages und CVS. CVS enthält Daten für die Versionsverwaltung, host enthält die Quell-Dateien für ecosconfig. Unter packages liegen alle Teile des Betriebssystems, wie Kernel, HAL und Gerätetreiber.

```
shell> cd $HOME
shell> mkdir -p develop/eCos_work/<PLATTFORM>_<STARTUP_TYP>
shell> export ECOS_REPOSITORY=/opt/ecos/ecos/packages
```

Die Plattform ist in diesem Beispiel das StrongARM 1110 DIMM. Diese ist in der eCos-Datenbank-Datei ecos.db als sa1110dimm eingetragen. Startup-Typ ist Download bzw. RAM, es ist also sinnvoll das Verzeichnis sa1110dimm_ram zu nennen.
Die Umgebungs-Variable ECOS_REPOSITORY enthält den Verzeichnis-Pfad, unter dem die eCos-Qnuell-Dateien liegen (s.o.). Diese Variable wird von ecosconfig ausgewertet, um die eCos-Quell-Dateien und die eCos-Datenbank-Datei ecos.db zu finden. Es ist sinnvoll, diese Variable in seine persönliche Shell-Konfigurations-Datei einzufügen, wenn man öfter mit eCos arbeitet.

```
shell> cd develop/eCos_work
shell> ln -s sa1110dimm_ram sa1110dimm
```

Es ist sinnvoll, für jeden Startup-Typ einer Plattform ein eigenes Verzeichnis anzulegen, also z.B. neben sa1110dimm_ram noch sa1110dimm_romram, und einen symbolischen Link mit dem Plattform-Namen, hier also sa1110dimm, anzulegen, der auf das aktuell benutzte Plattfom-Startup-Typ Verzeichnis zeigt.
Hintergrund hierfür ist, dass man in Skripten und Makefiles nicht das konkrete Verzeichnis, also z.B. sa1110dimm_ram, in Pfade usw. einträgt, sondern stattdessen den Link, hier sa1110dimm. So kann man die Applikation ohne Änderungen an Makefiles usw., nur durch ändern eines Datei-System-Links, für einen anderen Startup-Typ kompilieren. Während man zum Entwickeln die ganze Zeit sa1110dimm_ram benutzt hat, kann man für den Prototypen schnell eine Standalone-Applikation mit ROMRAM-Startup erzeugen.

[1] Der GDB-Stub ist ein minimaler ROM-Monitor, der einen Programm-Download per RS-232 vom GNU-Debugger GDB auf das Target ermöglicht. Außerdem ermöglicht der GDB-Stub Remote-Debugging.

[2] Eine zur Windows-Version äquivalente Version von ecosconfig mit grafischer Benutzeroberfläche ist für Linux in Entwicklung.

shell> cd $HOME
shell> cd develop/eCos_work/sa1110dimm_ram
shell> ecosconfig new sa1110dimm <TEMPLATE>

Ergebnis des Aufrufs von ecosconfig ist eine Konfigurations-Datei für die eCos Laufzeit-Bibliothek libtarget.a mit dem Namen ecos.ecc. In dieser Datei sind sämtliche Merkmale von libtarget.a gespeichert. ecos.ecc ist eine normale Textdatei, die mit jedem Text-Editor bearbeitet werden kann. Die Datei ist austauschbar mit der von der Windows-Version von ecosconfig erzeugten ecos.ecc.

Für verschiedene Anwendungsfälle sind bereits verschiedene Schablonen (Templates) definiert. Das Template default enthält alle Elemente, die man für die normale Applikations-Entwicklung für eCos braucht. Das Template elix bspw. enthält die Pakete, die im Template default enthalten sind plus der POSIX Kompatibilitäts-API EL/IX. Ein Template definiert also eine Menge von Paketen, die letztlich in der Laufzeit-Bibliothek libtarget.a enthalten sind.

shell> ecosconfig add <PACKAGE_0> <<PACKAGE_1> ... <PACKAGE_N>>

Der Umfang von libtarget.a läßt sich vor dem Kompilieren durch gezieltes Entfernen oder Hinzufügen von Paketen verändern. Im folgenden Beispiel werden Flash-Treiber hinzugefügt, um das Onboard-Flash des SA1110-DIMM von der Applikation aus beschreiben zu können.
shell> ecosconfig add CYGPKG_IO_FLASH CYGPKG_DEVS_FLASH_AMD_AM29XXXXX CYGPKG_DEVS_FLASH_SA1110DIMM

Der Flash-Treiber besteht aus drei Teilen. Der erste Teil ist der abstrakte Teil, der die API enthält, der zweite Teil ist der Chip-spezifische Teil, der den konkreten Programmier-Algorithmus für den eingesetzten Baustein bzw. die Baustein-Familie enthält, und der dritte Teil ist der Plattform-spezifische Treiber, der Informationen über Anzahl und Adressen der Flash-Bausteine enthält. Der dritte Teil ist vom Plattform-Hersteller zur Verfügung zu stellen.

Jetzt sollte ecos.ecc editiert und an die speziellen Bedürfnisse angepasst werden. Insbesondere die Option CYG_HAL_STARTUP sollte passend zum oben gewählten Verzeichnisnamen gesetzt werden.

shell> ecosconfig check

Mit diesem Befehl überprüft man, ob die gegenwärtige Konfiguration konfliktfrei ist. Es gibt Optionen, die sich gegenseitig bedingen oder ausschließen. Ein Check-Durchlauf meldet diese Konflikte.

shell> ecosconfig resolve

Haben sich Konflikte ergeben, so können diese evtuell automatisch durch einen Resolve-Durchlauf aufgelöst werden. Ist das nicht der Fall, so ist ecos.ecc wiederum zu editieren, bis keine Konflikte mehr auftreten.

shell> ecosconfig tree

Dieser Befehl erzeugt Makefiles und einen Dateibaum sowie Header-Dateien, die die Konfigurationsdaten aus ecos.ecc dem Compiler zur Verfügung stellen.

shell> make

Ergebnis des Compiler-Laufs sind install/lib/libtarget.a, install/lib/target.ld und ein Verzeichnis-Baum mit den Header-Dateien zu allen Paketen der Konfiguration unter install/include.

2.2.2 libtarget.a unter Windows erzeugen

Unter Windows werden die gleichen Werkzeuge, natürlich für Windows kompiliert, verwendet, wie unter Linux. Damit diese lauffähig sind, ist die Cygwin-Umgebung zu installieren, die eine Unix-Umgebung soweit möglich unter Windows emuliert. Da Cygwin nur auf den „höherwertigen“ Windows-Plattformen Windows NT/2000/XP stabil funktioniert, ist die Entwicklung von eCos-Applikationen nur unter diesen „Betriebssystemen“ möglich.
Unter Windows hat ecosconfig ein komfortables, mit der Maus bedienbares Interface. Sämtliche Eigenschaften in ecos.ecc lassen sich über Menüs auswählen.

Kapitel 3

Applikationen schreiben

3.1 Grundsätzliches

3.1.1 Applikationen kompilieren und linken

Um eine eCos-Applikation zu übersetzen und zu linken, sind bestimmte Compiler- und Linker-Optionen notwendig. Hierzu das folgende Makefile als Beispiel:

```
CC      = arm-elf-g++
INCDIR  = -I$$HOME/develop/eCos_work/sa1110dimm/install/include \
          -I$$HOME/irgendwas_anderes
LIBDIR  = -L$$HOME/develop/eCos_work/sa1110dimm/install/lib
OPT     = -O2
CCFLAGS = -ggdb -fno-rtti -fno-exceptions $(OPT)
LDFLAGS = -nostdlib -Ttarget.ld $(LIBDIR)

all: hello

hello: hello.o
@$(CC) $(LIBDIR) $(INCDIR) $(LDFLAGS) -o $@ $<

hello.o: hello.cc
@$(CC) -c $(CCFLAGS) $(INCDIR) $<
```

Der besseren Übersicht und der Bequemlichkeit halber werden alle Compiler- und Linker-Schalter und Parameter in Variablen gespeichert. Der Compiler - hier: arm-elf-g++, der GNU-C/C++-Compiler für ARM-Prozessoren - benötigt den Pfad zu den eCos-Include-Dateien, $$HOME/develop/eCos_work/sa1110dimm/install/include - sowie die Schalter no-rtti und no-exceptions. Der Include-Pfad muss das Include-Verzeichnis der kompilierten eCos-Umgebung enthalten, dies ist immer <PFAD_ZUR_ECOS_UMGEBUNG>/install/include. Dieses Verzeichnis sollte, wie im Beispiel, das erste Include-Verzeichnis im Include-Pfad des Compilers sein. Der Schalter no-rtti schaltet die Generierung von Code für das C++-Feature Runtime-Type-Information - Typ-Informationen über Objekte zur Laufzeit der Applikation - ab, da die zu eCos gehörenden Bibliotheken bzw. libtarget.a dies nicht unterstützen. Das gleiche gilt für den Schalter no-exceptions, der die Unterstützung für C++-Exceptions abschaltet. Programmiert man reines C, so sind diese Schalter nicht nötig. Der Linker - hier: arm-elf-ld, wird zur Erfüllung des Targets *hello:* implizit vom Compiler aufgerufen - benötigt den Pfad zur eCos-Laufzeit-Bibliothek libtarget.a und zum Binden das für die Ziel-Plattform spezielle Linker-Skript target.ld, das sich im gleichen Verzeichnis wie libtarget.ld befindet. Außerdem benötigt der Linker die Schalter nostdlib und Ttarget.ld. Der Schalter nostdlib sorgt dafür, dass der Linker keine implizit definierten Bibliotheken benutzt, um das Programm zu binden. Der Schalter Ttarget.ld definiert das Linker-Skript target.ld als Steuer-Skript für den Link-Vorgang. Würde dieser Schalter nicht angegeben, so würde der Linker ein implizit definiertes, für die Ziel-Hardware unpassendes Skript verwenden.

3.1.2 Programmier-Schnittstellen/API

eCos bietet diverse Programmier-Schnittstellen an, mitunter sogar parallel.

- **Native C-API**
  Dies ist die originäre eCos-API.
- **Native C++-API**
  Der eCos-Kernel ist selbst C++ implementiert. Die Header-Dateien der Kernel-Objekte sind direkt als API nutzbar. Die C-API ist eigentlich nichts anderes als ein Wrapper für die C++-API. Leider ist diese API nicht in der Qualität der C-API dokumentiert. Eine parallele Benutzung der C- und der C++-API ist problemlos möglich. Die Geräte-Treiber-Schnittstelle ist jedoch rein C-basiert.
- **μITRON API**
  Dies ist eine zur in Japan verbreiteten μITRON-API kompatible API. Die μITRON-API ist eine Komponente, die explizit zu einer eCos-Laufzeit-Umgebung hinzugefügt werden muss.
- **POSIX über EL/IX**
  RedHat stellt mit EL/IX eine API zur Verfügung, die teil-kompatibel zur POSIX-Spezifikation ist. Teil-kompatibel bedeutet in diesem Zusammenhang, dass Funktionen, die nur auf Mehr-Prozess-Systemen Sinn machen, weg gelassen wurden. Ansonsten vereinfacht diese API die Portierung einer (MultiThreaded-)POSIX/UNIX-Applikation extrem. EL/IX ist eine Komponente, die einer eCos-Laufzeit-Umgebung explizit hinzugefügt werden muss und nicht unerheblich Speicherplatz benötigt.

In den folgenden Beispielen werden sowohl die C- als auch die C++-API eingesetzt, da es zu POSIX genügend Drittliteratur, z.B. [7], gibt und die μITRON-API hier eher unbekannt ist.

3.1.3 Einschränkungen bei der Benutzung von C++

Unter eCos sind (noch) nicht sämtliche Features von C++ verwendbar:

- Streams
- STL (Standard Template Library)
- Alle anderen Funktionen der libstdc++

3.2 Hello world!

Das obligatorische „Hello, world!"-Programm sieht für eCos so aus:

```
#include <stdio.h>

int main ( void) {
  printf ( "\n Hello world!\n");
  return 0;
}
```

Die eCos-(C-)-Version unterscheidet sich nicht von derjenigen für ein beliebiges anderes Betriebssystem. Es gibt `main`, `printf` und `stdio`, das war's.
Unter eCos ist der Einstiegspunkt für eine Applikation aber normalerweise nicht `main`, sondern die Funktion `cyg_user_start`. Semantisch ändert sich allerdings nichts am Programm.

```
1  #include <stdio.h>
2
3  #ifdef __cplusplus
4  extern "C" {
5  #endif
```

```
 6  int cyg_user_start ( void) {
 7    printf ( "\nHello world.\n");
 8    return 0;
 9  }
10  #ifdef __cplusplus
11  }
12  #endif
```

Unbedingt zu beachten ist die Deklaration von `cyg_user_start` als `extern "C"`, wenn die Applikation in C++ implementiert wird, da libtarget.a als C-Programm gebunden wird (C-linkage) und `cyg_user_start` ansonsten nicht vom Linker gefunden würde und eine Default-Version von `cyg_user_start` benutzt wird. Der Effekt hiervon ist, das die Applikation nichts macht, da sie direkt nach dem Start wieder beendet wird.

3.3 Zwei Threads

Der Umgang mit Threads ist für die Entwicklung von eCos-Applikationen elementar. Das folgende Programm gibt abwechselnd „thread1“ und „thread2“ aus, indem es zwei parallel laufende Threads erzeugt, die „thread1“ bzw. „thread2“ ausgeben. In diesem, und den folgenden Beispielen, wird der Time-Slice-Scheduler verwendet, der lauffähigen höherpriorisierten Threads zu CPU zuweist und die CPU-Zeit zwischen lauffähigen Threads gleicher Priorität aufteilt.

```
 1  // Kernel C-API
 2  #include <cyg/kernel/kapi.h>
 3  // für printf()
 4  #include <stdio.h>
 5
 6
 7  // eine sinnvolle Stack-Größe definieren
 8  #define THREAD_STACK_SIZE (4096)
 9
10  // Speicher für Thread-Stacks allokieren
11  char gStack1[THREAD_STACK_SIZE];
12  char gStack2[THREAD_STACK_SIZE];
13
14  // Speicher für Thread-"Objekte" allokieren
15  cyg_thread gThread1;
16  cyg_thread gThread2;
17
18  // Speicher für Thread-Handles allokieren
19  cyg_handle_t gThreadHandle1;
20  cyg_handle_t gThreadHandle2;
21
22  // Thread-Nutzlast deklarieren (siehe auch unten)
23  cyg_thread_entry_t threadBody;
24
25
26  //
27  void cyg_user_start ( void) {
28    cyg_thread_create(
29      4,                          // Priorität
30      threadBody,                 // Nutzlast
31      (cyg_addrword_t) "thread1", // thread-spezifische Daten
32      "Thread 1",                 // Thread-Name
33      (void *) gStack1,           // Zeiger auf Thread-Stack
```

```
34      THREAD_STACK_SIZE,          // Größe des Stacks
35      &gThreadHandle1,            // Zeiger auf Handle
36      &gThread1);                 // Zeiger auf
37                                  // Thread-"Objekt"
38    cyg_thread_create(
39      4,
40      threadBody,
41      (cyg_addrword_t) "thread2",
42      "Thread 2",
43      (void *) gStack2,
44      THREAD_STACK_SIZE,
45      &gThreadHandle2,
46      &gThread2);
47
48    cyg_thread_resume ( gThreadHandle1);
49    cyg_thread_resume ( gThreadHandle2);
50  }
51
52
53  //
54  // pThreadData - thread-spezifische Daten
55  //
56  void threadBody ( cyg_addrword_t pThreadData) {
57    char* lString = (char*) pThreadData;
58
59    for ( ;;) {
60      printf ( " %s\n", lString);
61      cyg_thread_yield ();
62    }
63  }
```

Zuerst werden 4096 Bytes Speicher für jeden Thread-Stack allokiert (Zeile 11 und 12), was für die meisten Fälle ausreichend ist. Das Prinzip, den Stack-Speicher explizit in einer eigenen Anweisung und nicht implizit bei der Instanziierung des Threads zu allokieren, hat u.a. den Vorteil, dass der Stack-Speicher zeitkritischer Threads auf Maschinen mit inhomogener Speicher-Geschwindigkeit, z.B. zusätzliches SRAM neben DRAM, in den schnelleren Speicher gelegt werden kann.
In den Variablen `gThread1` (Zeile 15/36) und `gThread2` (Zeile 16/46) legt der Kernel die neu erzeugten Thread-Objekte ab. Nach der Erzeugung der Threads, sollte man diese Variablen nicht mehr anfassen. `gThreadHandle1` und `gThreadhandle2` sind Handles mit denen man nach Erzeugung der Threads mit diesen interagieren kann. `threadBody` schließlich ist die Funktion, die von den Threads ausgeführt wird. Jede Funktion mit der Signatur von `threadBody` kann „Nutzlast“ eines Threads sein. In `cyg_user_start` werden beide Threads zunächst mittels `cyg_thread_create` erzeugt und anschließend mittels `cyg_thread_resume` lauffähig gemacht. Der Scheduler startet nachdem `cyg_user_start` beendet ist. Thread-Prioritäten (Z.29/39) sind von 1 bis 31 definiert, wobei 1 die höchste mögliche Priorität und 31 die niedrigste mögliche Priorität ist.
Der Befehl `cyg_thread_yield` (Zeile 61) teilt dem Scheduler mit, dass der gegenwärtig laufende Thread die CPU frei geben will und löst damit ein Rescheduling aus.
Eine C++-Version obigen Programms sieht so aus:

```
1  #include <cyg/kernel/kapi.h>
2  // Kernel Konfiguration
3  #include <pkgconf/kernel.h>
4  // Cyg_Thread-Klasse
5  #include <cyg/kernel/thread.hxx>
6  #include <cyg/kernel/thread.inl>
```

```
 7  #include <stdio.h>
 8
 9  #define THREAD_STACK_SIZE (4096)
10
11  char gStack1[THREAD_STACK_SIZE];
12  char gStack2[THREAD_STACK_SIZE];
13  // Zeiger auf Thread-Objekte
14  Cyg_Thread* gThread1;
15  Cyg_Thread* gThread2;
16
17  cyg_thread_entry_t threadBody;
18
19
20  //
21  #ifdef __cplusplus
22  extern "C" {
23  #endif
24  void cyg_user_start ( void) {
25    gThread1 = new Cyg_Thread (
26                     4,                         // Priorität
27                     threadBody,                // Nutzlast
28                     (CYG_ADDRWORD) "thread1", // thread-spezifische Daten
29                     "Thread 1",                // Thread-Name
30                     (CYG_ADDRWORD) gStack1,   // Zeiger auf Thread-Stack
31                     THREAD_STACK_SIZE);       // Größe des Stacks
32    gThread2 = new Cyg_Thread (
33                     4,
34                     threadBody,
35                     (CYG_ADDRWORD) "thread2",
36                     "Thread 2",
37                     (CYG_ADDRWORD) gStack2,
38                     THREAD_STACK_SIZE);
39
40    gThread1->resume ();
41    gThread2->resume ();
42  }
43  #ifdef __cplusplus
44  }
45  #endif
46
47
48  //
49  void threadBody ( cyg_addrword_t pThreadData) {
50    char* lString = (char*) pThreadData;
51
52    for ( ;;) {
53      printf ( " %s\n", lString);
54      Cyg_Thread::self()->yield ();
55    }
56  }
```

Der Thread-Name muss nicht unbedingt gesetzt werden, er ist jedoch beim Debuggen hilfreich, da der Debugger Thread-Namen anzeigt.

3.4 Speicher verwalten

Speicher läßt sich in einer eCos-Applikation wie gewohnt mit `malloc` bzw. `new` anfordern und mit `free` bzw. `delete` frei geben. Bei Verwendung von `new` und `delete` ist allerdings die Option CYGFUN_INFRA_-EMPTY_DELETE_FUNCTIONS in der Konfigurationsdatei ecos.ecc abzuschalten, d.h. auf „user value 0“ zu setzen. eCos bietet mit dem Memory-Pool noch ein zusätzliches Konstrukt Speicher zu verwalten.
Ein Memory-Pool ist ein vom eCos-Kernel verwalteter Speicherblock mit eigenen Verwaltungsfunktionen. eCos unterscheidet zwei Arten von Memory-Pools. Eine Version kann nur Blöcke gleicher, fest bei der Erzeugung des Pools eingestellter Größe, die andere Version kann Blöcke variabler Größe verwalten. Ein Vorteil eines Memory-Pools ist zunächst einmal die Möglichkeit diesen an eine beliebige Stelle des Adressraums platzieren zu können, da bei seiner Erzeugung eine Basisadresse angegeben werden muss, d.h. auch hier kann man u.U. wieder spezielle Eigenschaften des Arbeitsspeichers ausnutzen. Ein weiterer Vorteil des Memory-Pools ist, dass dieser drei verschiedene Methoden zur Allokation anbietet nämlich erstens blockierend - der Speicher anfordende Thread blockiert so lange in seiner Ausführung bis die Anforderung erfüllt wird -, zweitens nicht-blockierend - die Anforderung kehrt garantiert zurück, liefert jedoch NULL wenn kein Speicher allokiert werden konnte - und drittens blockierend mit Timeout - die Anforderung kehrt nach einer einstellbaren Zeitspanne garantiert zurück, liefert jedoch NULL, wenn kein Speicher allokiert werden konnte.
Hat eine Applikation viele Speicherblöcke gleicher, fester Größe zu verwalten, so kann es von Vorteil sein einen Memory-Pool mit fester Blockgröße zu verwenden, da dieser anzunehmenderweise effizientere Verwaltungs-Algorithmen anwenden kann als `malloc` bzw. `new`.

```
 1  #include <cyg/kernel/kapi.h>
 2  // Memory-Pool
 3  #include <cyg/memalloc/kapi.h>
 4  #include <stdio.h>
 5
 6  #define THREAD_STACK_SIZE (4096)
 7
 8  char gStack1[THREAD_STACK_SIZE];
 9  cyg_thread gThread1;
10  cyg_handle_t gThreadHandle1;
11  cyg_thread_entry_t threadBody;
12
13  // Speicher für Memory-Pool-"Objekte" allokieren
14  cyg_mempool_var gMemPoolVar;
15  cyg_mempool_fix gMemPoolFix;
16  // Handles für Memory-Pool-"Objekte"
17  cyg_handle_t gMemPoolVarHandle;
18  cyg_handle_t gMemPoolFixHandle;
19
20
21  //
22  void cyg_user_start ( void) {
23    cyg_thread_create( 4,
24                       threadBody,
25                       (cyg_addrword_t) "thread1",
26                       "Thread 1",
27                       (void *) gStack1,
28                       THREAD_STACK_SIZE,
29                    &gThreadHandle1,
30                       &gThread1);
31    cyg_thread_resume ( gThreadHandle1);
32  }
33
```

```
34  //
35  void threadBody ( cyg_addrword_t pThreadData) {
36    char* lString = (char*) pThreadData;
37    void* lMem1   = NULL;
38    void* lMem2   = NULL;
39
40    printf ( "1");
41
42    cyg_mempool_var_create (
43      (void*) ( 1024 * 1024 * 16), // Basisadresse 16MB
44      32,                          // Größe 32 Byte
45      &gMemPoolVarHandle,          // Zeiger auf MemPool-Handle
46      &gMemPoolVar);               // Zeiger auf MemPool-"Objekt"
47
48    printf ( "2");
49
50    cyg_mempool_fix_create (
51      (void*) ( 1024 * 1024 * 17), // Basisadresse 17MB
52      32,                          // Größe 32 Byte
53      4,                           // Block-Größe 4 Byte
54      &gMemPoolFixHandle,          // Zeiger auf MemPool-Handle
55      &gMemPoolFix);               // Zeiger auf MemPool-"Objekt"
56
57
58    printf ( "3");
59
60    // blockierende Speicher-Anforderung von 4 Byte
61    // wenn niemals genügend Speicher zur Verfügung steht,
62    // wartet diese Anforderung _ewig_
63    lMem1 = cyg_mempool_var_alloc ( gMemPoolVarHandle, 4);
64
65    printf ( "4");
66
67    // Freigabe von 4 Byte
68    cyg_mempool_var_free ( gMemPoolVarHandle, lMem1);
69
70    printf ( "5");
71
72    // nicht blockierende Speicher-Anforderung von 4 Byte
73    // diese Anforderung kehrt sofort zurück,
74    // konnte kein Speicher allokiert werden wird NULL
75    // zurück geliefert
76    lMem1 = cyg_mempool_var_try_alloc ( gMemPoolVarHandle, 4);
77
78    if ( lMem1 == NULL) {
79      printf ( "NULL");
80    }
81
82    printf ( "6");
83
84    // Freigabe von 4 Byte
85    cyg_mempool_var_free ( gMemPoolVarHandle, lMem1);
86
87    // zeitlich begrenzt blockierende Anforderung von 4 Byte
```

```
 88     // es Wartezeit ist auf 100 Ticks der internen Uhr begrenzt
 89     // diese Anforderung kehrt sofort zurück, wenn Speicher allokiert
 90     // werden konnte und ansonsten nach dem eingestellten Timeout
 91     // dann wird NULL zurück geliefert
 92     lMem1 = cyg_mempool_var_timed_alloc ( gMemPoolVarHandle, 4, 100);
 93
 94
 95     // blockierende Speicher-Anforderung von 4Byte
 96     // aus dem Pool fester Blockgröße
 97     lMem2 = cyg_mempool_fix_alloc ( gMemPoolFixHandle);
 98
 99     // Freigabe von 4 Byte
100     cyg_mempool_fix_free ( gMemPoolFixHandle, lMem2);
101
102     // nicht blockierende Speicher-Anforderung von 4 Byte
103     lMem2 = cyg_mempool_fix_try_alloc ( gMemPoolFixHandle);
104
105     if ( lMem2 == NULL) {
106       printf ( "NULL");
107     }
108
109     // Freigabe von 4 Byte
110     cyg_mempool_fix_free ( gMemPoolFixHandle, lMem2);
111
112     // zeitlich begrenzt blockierende Anforderung von 4 Byte
113     lMem2 = cyg_mempool_fix_timed_alloc ( gMemPoolFixHandle, 100);
114
115     printf ( " all done\n");
116   }
```

Allerdings legen Memory-Pools interne Verwaltungsdaten ebenfalls in dem ihnen zugewiesenen Speicherblock ab, sodass nicht sämtliche Bytes des Memory-Pools vom Benutzer allokierbar sind. Die Größe der Verwaltungsdaten beträgt bei Pools variabler Blockgröße mindestens 28 Byte und bei Pools fester Blockgröße 4 Byte, wobei nicht auszuschließen ist, dass größere Pools mehr Byte an Verwaltungsdaten benötigen. Hat ein variabler Memory-Pool bspw. die Größe 1024 Byte, so kann man maximal 996 Byte davon benutzen.

```
 1  // Kernel C-API
 2  #include <cyg/kernel/kapi.h>
 3  #include <pkgconf/kernel.h>
 4  #include <cyg/kernel/thread.hxx>
 5  #include <cyg/kernel/thread.inl>
 6  // für Memory-Pool-Klassen
 7  #include <cyg/memalloc/memvar.hxx>
 8  #include <cyg/memalloc/memfixed.hxx>
 9  #include <stdio.h>
10
11  #define THREAD_STACK_SIZE (4096)
12
13  char gStack1[THREAD_STACK_SIZE];
14  Cyg_Thread* gThread1;
15  cyg_thread_entry_t threadBody;
16
17  // Zeiger auf Pool-Objekte definieren
18  Cyg_Mempool_Variable* gMemPoolVar;
```

```
19  Cyg_Mempool_Fixed*    gMemPoolFix;
20
21
22  //
23  #ifdef __cplusplus
24  extern "C" {
25  #endif
26  void cyg_user_start ( void) {
27    gThread1 = new Cyg_Thread (
28                     4,
29                     threadBody,
30                     (cyg_addrword_t) "thread1",
31                     "Thread 1",
32                     (cyg_addrword_t) gStack1,
33                     THREAD_STACK_SIZE);
34
35    gThread1->resume ();
36  }
37  #ifdef __cplusplus
38  }
39  #endif
40
41
42  //
43  void threadBody ( cyg_addrword_t pThreadData) {
44    char* lString = (char*) pThreadData;
45    cyg_uint8* lMem1 = NULL;
46    cyg_uint8* lMem2 = NULL;
47
48    gMemPoolVar = new Cyg_Mempool_Variable (
49                        (cyg_uint8*) ( 1024 * 1024 * 16), // Basisadresse 16MB
50                        32,                             // Größe 32 Byte
51                        8);                             // Alignment (8 ist Standard)
52
53    gMemPoolFix = new Cyg_Mempool_Fixed (
54                        (cyg_uint8*) ( 1024 * 1024 * 17), // Basisadresse 17MB
55                        32,                             // Größe 32 Byte
56                        4);                             // Block-Größe 4 Byte
57
58    printf ( "3");
59
60    // blockierende Speicher-Anforderung von 4 Byte
61    // wenn niemals genügend Speicher zur Verfügung steht,
62    // wartet diese Anforderung _ewig_
63    lMem1 = gMemPoolVar->alloc ( 4);
64
65    printf ( "4");
66
67    // Freigabe von 4 Byte
68    gMemPoolVar->free ( lMem1);
69
70    printf ( "5");
71
72    // nicht blockierende Speicher-Anforderung von 4 Byte
```

```
 73     // diese Anforderung kehrt sofort zurück,
 74     // konnte kein Speicher allokiert werden wird NULL
 75     // zurück geliefert
 76     lMem1 = gMemPoolVar->try_alloc ( 4);
 77
 78     if ( lMem1 == NULL) {
 79       printf ( "NULL");
 80     }
 81
 82     printf ( "6");
 83
 84     // Freigabe von 4 Byte
 85     gMemPoolVar->free ( lMem1);
 86
 87     // zeitlich begrenzt blockierende Anforderung von 4 Byte
 88     // es Wartezeit ist auf 100 Ticks der internen Uhr begrenzt
 89     // diese Anforderung kehrt sofort zurück, wenn Speicher allokiert
 90     // werden konnte und ansonsten nach dem eingestellten Timeout
 91     // dann wird NULL zurück geliefert
 92     lMem1 = gMemPoolVar->alloc ( 4, 100);
 93
 94
 95     // blockierende Speicher-Anforderung von 4Byte
 96     // aus dem Pool fester Blockgröße
 97     lMem2 = gMemPoolFix->alloc ();
 98
 99     // Freigabe von 4 Byte
100     gMemPoolFix->free ( lMem2);
101
102     // nicht blockierende Speicher-Anforderung von 4 Byte
103     lMem2 = gMemPoolFix->try_alloc ();
104
105     if ( lMem2 == NULL) {
106       printf ( "NULL");
107     }
108
109     // Freigabe von 4 Byte
110     gMemPoolFix->free ( lMem2);
111
112     // zeitlich begrenzt blockierende Anforderung von 4 Byte
113     lMem2 = gMemPoolFix->alloc ( 100);
114
115     printf ( " all done\n");
116   }
```

Die C++-Version des Beispiels unterscheidet sich von der C-Version lediglich dadurch, dass die C++-Pools keine explizite `*_timed_alloc()`-Funktion anbieten. Stattdessen ist die `alloc()`-Methode entsprechend überladen worden. Außerdem wird dem Konstruktor die Basisadresse nicht mit Typ `void*` übergeben, sondern mit dem Typ `cyg_uint8*`, der normalerweise identisch zum Typ `unsigned char*` ist.

3.5 Kommunikation zwischen Threads

eCos stellt mit der Message-Box einen einfachen, aber leistungsfähigen Mechanismus zum Datenaustausch zwischen Threads zu Verfügung. Eine Message-Box ist ein FIFO-Puffer in den ein oder mehrere Threads

hinein schreiben und aus dem ein oder mehrere Threads lesen können. Die Reihenfolge, in der Objekte in die Message-Box eingefügt werden, entspricht garantiert der Reichenfolge in der die Objekte aus der Message-Box ausgelesen werden. Der Zugriff auf eine Message-Box ist implizit synchronisiert. So lange noch Speicherplatz in der Message-Box verfügbar ist, kann ein schreibender Thread Objekte hinein schreiben, ansonsten blockiert er. Umgekehrt blockiert ein lesender Zugriff, so lange die Message-Box leer ist, den lesenden Thread. Wie bei den Memory-Pools gibt es jedoch auch alternative nicht-blockierende Zugriffsmethoden bzw. Methoden mit Timeout, wobei für Zugriffs-Methoden mit Timeout die Option CYGFUN_-KERNEL_THREADS_TIMER in der eCos-Konfiguration aktiviert sein muss. Die Standardgröße - d.h. die Anzahl der Objekte, die sich gleichzeitig in der Message-Box befinden können - wird durch die Option CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE bestimmt und hat den Wert zehn voreingestellt.

```
 1  // Kernel C-API enthält auch Message-Box API
 2  #include <cyg/kernel/kapi.h>
 3  #include <stdio.h>
 4
 5  #define THREAD_STACK_SIZE (4096)
 6
 7  char gStack1[THREAD_STACK_SIZE];
 8  char gStack2[THREAD_STACK_SIZE];
 9  char gStack3[THREAD_STACK_SIZE];
10
11  cyg_thread gSender;
12  cyg_thread gReceiver;
13  cyg_thread gParasite;
14  cyg_handle_t gSenderHandle;
15  cyg_handle_t gReceiverHandle;
16  cyg_handle_t gParasiteHandle;
17  cyg_thread_entry_t senderBody;
18  cyg_thread_entry_t receiverBody;
19  cyg_thread_entry_t parasiteBody;
20
21  // Speicher für Message-Box-"Objekt" allokieren
22  cyg_mbox gMBox;
23
24  // Handle für die MessageBox
25  cyg_handle_t gMBoxHandle;
26
27
28
29  //
30  void cyg_user_start ( void) {
31    cyg_thread_create ( 4,
32                        senderBody,
33                        (cyg_addrword_t) "a sender",
34                        "Sender",
35                        (void *) gStack1,
36                        THREAD_STACK_SIZE,
37                     &gSenderHandle,
38                        &gSender);
39    cyg_thread_create ( 4,
40                        receiverBody,
41                        (cyg_addrword_t) "a receiver",
42                        "Receiver",
43                        (void *) gStack2,
44                        THREAD_STACK_SIZE,
```

```
45                    &gReceiverHandle,
46                       &gReceiver);
47    cyg_thread_create ( 4,
48                       parasiteBody,
49                       (cyg_addrword_t) "a parasite",
50                       "Parasite",
51                       (void *) gStack3,
52                       THREAD_STACK_SIZE,
53                    &gParasiteHandle,
54                       &gParasite);
55
56    cyg_mbox_create ( &gMBoxHandle, &gMBox);
57
58    cyg_thread_resume ( gSenderHandle);
59    cyg_thread_resume ( gReceiverHandle);
60    // dieser Thread bringt die Ausgabe etwas durcheinander
61    // läßt man ihn abgeschaltet, so bleibt die Ausgabe korrekt
62    cyg_thread_resume ( gParasiteHandle);
63  }
64
65
66  //
67  void senderBody ( cyg_addrword_t pThreadData) {
68    char* lTestString =
69      "Sind Sie vielleicht John Wayne? Ocer bin ich das?";
70    cyg_uint32 lTestStringSize = strlen ( lTestString);
71    cyg_uint32 i = 0;
72
73    for ( ;;) {
74      // nicht-blockierendes Einfügen eines Objekt(-Zeiger)s
75      // in die Message-Box
76      // ist die Box voll, so wird false zurück geliefert
77      if ( cyg_mbox_tryput ( gMBoxHandle, (void*) &lTestStringSize)) {
78        for ( i = 0; i < lTestStringSize;) {
79          // blockierendes Einfügen eines Objekt(-Zeiger)s
80          // in die Message-Box
81          // ist die Box voll, so blockiert dieser Aufruf so lange
82          // bis wieder ein Objekt in die Box eingefügt werden kann
83          cyg_mbox_put ( gMBoxHandle, (void*) &(lTestString[i]));
84          i++;
85
86          if ( i < lTestStringSize) {
87            // zeitlich begrenzt blockierendes Einfügen eines Objekt(-Zeiger)s
88            // ist die Message-Box 100 Timer-Ticks voll, so liefert die
89            // Funktion false zurück
90            if ( cyg_mbox_timed_put ( gMBoxHandle,
91                                     (void*) &(lTestString[i]),
92                                     100))
93            {
94              i++;
95            }
96          }
97        }
98      }
```

```
 99     }
100   }
101
102
103   //
104   void receiverBody ( cyg_addrword_t pThreadData) {
105     cyg_uint32 lReadStringSize = 0;
106     cyg_uint32 i = 0;
107     void* lTryGet;
108
109     for ( ;;) {
110       // kehrt auf jeden Fall zurück, wenn kein Objekt in der Message-Box
111       // steht, wird NULL zurückgeliefert
112       lTryGet = cyg_mbox_tryget ( gMBoxHandle);
113
114       if ( lTryGet != NULL) {
115         lReadStringSize = *((cyg_uint32*) lTryGet);
116
117         for ( i = 0; i < lReadStringSize;) {
118           char* lCharPointer;
119
120           // blockierendes Auslesen eines Objekt(-Zeiger)s aus der Message-Box
121           // ist die Box leer, so blockiert dieser Aufruf bis mindestens
122           // ein Objekt in der Box ist
123           lCharPointer = (char*) cyg_mbox_get ( gMBoxHandle);
124           printf ( "%c", *lCharPointer);
125           i++;
126
127           if ( i < lReadStringSize) {
128             // zeitlich begrenzt blockierendes Auslesen eines Objekt(-Zeiger)s
129             // aus der Message-Box
130             // die Wartezeit beträgt 100 Timer-Ticks
131             lCharPointer = (char*) cyg_mbox_timed_get ( gMBoxHandle, 100);
132
133             if ( lCharPointer != NULL) {
134               printf ( "%c", *lCharPointer);
135               i++;
136             }
137           }
138         }
139         printf ( "\n");
140       }
141     }
142   }
143
144
145   //
146   void parasiteBody ( cyg_addrword_t pThreadData) {
147     for ( ;;) {
148       // liest ein Objekt aus der Message-Box aus,
149       // dieses wird jedoch nicht aus der Box entfernt
150       // sondern verbleibt darin
151       // ist die Box leer, so wird NULL zurückgeliefert
152       char* lCharPointer = (char*) cyg_mbox_peek_item ( gMBoxHandle);
```

```
153
154       if ( lCharPointer != NULL) {
155         // dies bringt die Ausgabe des Test-Strings etwas durcheinander
156         printf ( "%c", *lCharPointer);
157         cyg_thread_yield ();
158       }
159     }
160   }
```

Für C++ gibt es zwei Versionen der Message-Box, die sich aber lediglich im zu übergebenden Objekt-Typ unterscheiden. Während der einen Version auch `void*` übergeben werden, benutzt die andere Version C++-Templates.

```
 1  // Kernel C-API
 2  #include <cyg/kernel/kapi.h>
 3  #include <pkgconf/kernel.h>
 4  #include <cyg/kernel/thread.hxx>
 5  #include <cyg/kernel/thread.inl>
 6  // Message-Box-Klasse; für die Templates benutzende Message-Box
 7  // Klasse ist <cyg/kernel/mboxt.hxx> zu importieren
 8  #include <cyg/kernel/mbox.hxx>
 9  #include <stdio.h>
10
11  #define THREAD_STACK_SIZE (4096)
12
13  char gStack1[THREAD_STACK_SIZE];
14  char gStack2[THREAD_STACK_SIZE];
15
16  Cyg_Thread* gSender;
17  Cyg_Thread* gReceiver;
18  cyg_thread_entry_t senderBody;
19  cyg_thread_entry_t receiverBody;
20  // Zeiger auf Message-Box-Objekt; Template-basierte Klasse heißt Cyg_Mboxt
21  Cyg_Mbox* gMBox;
22
23  //
24  #ifdef __cplusplus
25  extern "C" {
26  #endif
27  void cyg_user_start ( void) {
28    gSender = new Cyg_Thread (
29                     4,
30                     senderBody,
31                     (cyg_addrword_t) "a sender",
32                     "Sender",
33                     (cyg_addrword_t) gStack1,
34                     THREAD_STACK_SIZE);
35    gReceiver = new Cyg_Thread (
36                     4,
37                     receiverBody,
38                     (cyg_addrword_t) "a receiver",
39                     "Receiver",
40                     (cyg_addrword_t) gStack2,
41                     THREAD_STACK_SIZE);
42
```

```
43    gMBox = new Cyg_Mbox ();
44
45    gSender->resume ();
46    gReceiver->resume ();
47  }
48  #ifdef __cplusplus
49  }
50  #endif
51
52
53  //
54  void senderBody ( cyg_addrword_t pThreadData) {
55    char* lTestString =
56      "Sind Sie vielleicht John Wayne? Ocer bin ich das?";
57    cyg_uint32 lTestStringSize = strlen ( lTestString);
58    cyg_uint32 i = 0;
59
60    for ( ;;) {
61      if ( gMBox->tryput ( (void*) &lTestStringSize)) {
62        for ( i = 0; i < lTestStringSize;) {
63          gMBox->put ( (void*) &(lTestString[i]));
64          i++;
65
66          if ( i < lTestStringSize) {
67            // put wurde für Timeout überladen
68            if ( gMBox->put ( (void*) &(lTestString[i]), 100)) {
69              i++;
70            }
71          }
72        }
73      }
74    }
75  }
76
77
78  //
79  void receiverBody ( cyg_addrword_t pThreadData) {
80    cyg_uint32 lReadStringSize = 0;
81    cyg_uint32 i = 0;
82    void* lTryGet;
83
84    for ( ;;) {
85      lTryGet = gMBox->tryget ();
86
87      if ( lTryGet != NULL) {
88        lReadStringSize = *((cyg_uint32*) lTryGet);
89
90        for ( i = 0; i < lReadStringSize;) {
91          char* lCharPointer;
92
93          lCharPointer = (char*) gMBox->get ();
94          printf ( "%c", *lCharPointer);
95          i++;
96
```

```
 97         if ( i < lReadStringSize) {
 98           // get wurde für Timeout überladen
 99           lCharPointer = (char*) gMBox->get ( 100);
100
101           if ( lCharPointer != NULL) {
102             printf ( "%c", *lCharPointer);
103             i++;
104           }
105         }
106       }
107       printf ( "\n");
108     }
109   }
110 }
```

3.6 Synchronisation

Einige eCos-Konstrukte, wie die oben behandelten Memory-Pools und Message-Boxes, enthalten implizite Synchronisationsmechanismen. eCos bietet jedoch auch mehrere explizite Konstrukte zur Synchronisation an, auf die in den folgenden Abschnitten eingegangen wird.

3.6.1 Mutex

Ein Mutex ist ein Objekt, dass zwei Zustände annehmen kann frei bzw. unlocked und gesperrt bzw. locked. Ein Thread, der einen Mutex gesperrt hat besitzt diesen und ist der einzige, der ihn wieder freigeben kann. Versucht ein anderer Thread einen bereits gesperrten Mutex zu sperren d.h. Eigentümer dieses Mutex zu werden, so blockiert dieser Thread so lange, bis der Eigentümer des Mutex diesen wieder frei gegeben hat.

```
 1 // Kernel C-API enthält auch Mutex
 2 #include <cyg/kernel/kapi.h>
 3 #include <stdio.h>
 4
 5 #define THREAD_STACK_SIZE (4096)
 6
 7 char gStack1[THREAD_STACK_SIZE];
 8 char gStack2[THREAD_STACK_SIZE];
 9 char gStack3[THREAD_STACK_SIZE];
10
11 cyg_thread gThread1;
12 cyg_thread gThread2;
13 cyg_thread gPrintThread;
14 cyg_handle_t gThreadHandle1;
15 cyg_handle_t gThreadHandle2;
16 cyg_handle_t gPrintThreadHandle;
17 cyg_thread_entry_t threadBody;
18 cyg_thread_entry_t printBody;
19 // Speicher für Mutex-"Objekt" allokieren
20 cyg_mutex_t gMutex;
21 // globale Variable mit der herum gespielt wird
22 cyg_int32  gSum = 0;
23
24
25 //
26 void cyg_user_start ( void) {
```

```
27    cyg_thread_create( 4,
28                       threadBody,
29                       (cyg_addrword_t) 1,
30                       "Thread 1",
31                       (void *) gStack1,
32                       THREAD_STACK_SIZE,
33                       &gThreadHandle1,
34                       &gThread1);
35    cyg_thread_create( 4,
36                       threadBody,
37                       (cyg_addrword_t) -1,
38                       "Thread 2",
39                       (void *) gStack2,
40                       THREAD_STACK_SIZE,
41                       &gThreadHandle2,
42                       &gThread2);
43    cyg_thread_create( 4,
44                       printBody,
45                       (cyg_addrword_t) 0,
46                       "Print Thread",
47                       (void *) gStack3,
48                       THREAD_STACK_SIZE,
49                       &gPrintThreadHandle,
50                       &gPrintThread);
51
52    // Mutex initialisieren
53    cyg_mutex_init ( &gMutex);
54
55    cyg_thread_resume ( gThreadHandle1);
56    cyg_thread_resume ( gThreadHandle2);
57    cyg_thread_resume ( gPrintThreadHandle);
58  }
59
60
61  //
62  void threadBody ( cyg_addrword_t pThreadData) {
63    cyg_int32  lAddend = (cyg_int32) pThreadData;
64    cyg_bool_t lLockSuccess = false;
65
66    for ( ;;) {
67      // Mutex sperren
68      // ist der Mutex bereits gesperrt, so blockiert der Thread
69      // hier so lange bis der Mutex wieder frei ist
70      // war cyg_mutex_lock erfolgreich, so wird true zurück
71      // geliefert; false wird nur zurück geliefert, wenn
72      // cyg_mutex_release benutzt wurde
73      lLockSuccess = cyg_mutex_lock ( &gMutex);
74
75      if ( lLockSuccess) {
76        gSum += lAddend;
77        cyg_thread_delay ( 100);
78
79        // Mutex freigeben
80        cyg_mutex_unlock ( &gMutex);
```

```
81
82        cyg_thread_yield ();
83      }
84    }
85  }
86
87
88  //
89  void printBody ( cyg_addrword_t pThreadData) {
90    for ( ;;) {
91      printf ( " %i\n", gSum);
92      cyg_thread_yield ();
93    }
94  }
95
```

Das Programm gibt mehrere Einsen, mehrere Nullen, mehrere Einsen usw. aus. Ohne den Mutex würden nur Nullen ausgegeben, da die Subtraktion bzw. Addition durch einen Thread durch den anderen sofort wieder ausgeglichen werden würde. Der Mutex blockiert jedoch immer einen Thread, sodass der Variablen-Wert für kurze Zeit auch ungleich Null sein kann.

```
 1  #include <cyg/kernel/kapi.h>
 2  #include <pkgconf/kernel.h>
 3  #include <cyg/kernel/thread.hxx>
 4  #include <cyg/kernel/thread.inl>
 5  // Mutex-Klasse
 6  #include <cyg/kernel/mutex.hxx>
 7  #include <stdio.h>
 8
 9  #define THREAD_STACK_SIZE (4096)
10
11  char gStack1[THREAD_STACK_SIZE];
12  char gStack2[THREAD_STACK_SIZE];
13  char gStack3[THREAD_STACK_SIZE];
14
15  Cyg_Thread* gThread1;
16  Cyg_Thread* gThread2;
17  Cyg_Thread* gPrintThread;
18  cyg_thread_entry_t threadBody;
19  cyg_thread_entry_t printBody;
20  // Zeiger auf Mutex-Objekt
21  Cyg_Mutex*  gMutex;
22  cyg_int32   gSum = 0;
23
24  //
25  #ifdef __cplusplus
26  extern "C" {
27  #endif
28  void cyg_user_start ( void) {
29    gThread1 = new Cyg_Thread (
30                      4,
31                      threadBody,
32                      (cyg_addrword_t) 1,
33                      "Thread 1",
34                      (cyg_addrword_t) gStack1,
```

```
35                   THREAD_STACK_SIZE);
36    gThread2 = new Cyg_Thread (
37                   4,
38                   threadBody,
39                   (cyg_addrword_t) -1,
40                   "Thread 2",
41                   (cyg_addrword_t) gStack2,
42                   THREAD_STACK_SIZE);
43    gPrintThread = new Cyg_Thread (
44                   4,
45                   printBody,
46                   (cyg_addrword_t) 0,
47                   "Print Thread",
48                   (cyg_addrword_t) gStack3,
49                   THREAD_STACK_SIZE);
50
51    // Mutex instanziieren
52    gMutex = new Cyg_Mutex ();
53
54    gThread1->resume ();
55    gThread2->resume ();
56    gPrintThread->resume ();
57  }
58  #ifdef __cplusplus
59  }
60  #endif
61
62
63  //
64  void threadBody ( cyg_addrword_t pThreadData) {
65    cyg_int32 lAddend = (cyg_int32) pThreadData;
66
67    for ( ;;) {
68      if ( gMutex->lock ()) {
69        gSum += lAddend;
70        Cyg_Thread::self()->delay ( 100);
71        gMutex->unlock ();
72        Cyg_Thread::self()->yield ();
73      }
74    }
75  }
76
77
78  //
79  void printBody ( cyg_addrword_t pThreadData) {
80    for ( ;;) {
81      printf ( " %i\n", gSum);
82      Cyg_Thread::self()->yield ();
83    }
84  }
```

Mutexe besitzen zusätzlich noch die Methoden `cyg_mutex_release` bzw. `release` und `cyg_mutex_destroy` bzw. den Destruktor der Mutex-Klasse. `cyg_mutex_destroy` zerstört einen Mutex. *Ein Mutex sollte nicht im gesperrten Zustand zerstört werden*, da dies zu nicht vorhersagbaren Effekten führen kann. `cyg_mutex_release` bzw. `release` erlöst alle auf einen Mutex wartenden - d.h. ein Thread will einen bereits gesperrten

Mutex sperren - Threads vom Warten. Diese Threads sind dann natürlich nicht Eigentümer des Mutex. Dies wird ihnen durch den Rückgabewert von `cyg_mutex_lock` bzw. `lock` mitgeteilt. Ist ein Thread Eigentümer eines Mutex geworden liefert `cyg_mutex_lock` bzw. `lock` true zurück, ansonsten false.

3.6.2 Semaphor

Ein Semaphor ist intern ein Zähler, der um eins dekrementiert wird, wenn ein Thread auf ihn zugreift und um eins erhöht wird, wenn ein Thread einen Semaphor frei gibt. Wenn der Zähler bereits gleich Null ist, so blockiert ein auf den Semaphor zugreifender Thread so lange bis der Zähler des Semaphor wieder größer als Null ist. Ein Semaphor mit einem maximalen Zählerstand von Eins entspricht in seiner Funktion also einem Mutex.
Modell für einen Semaphor ist bspw. ein Raum in den nur n Personen hinein passen, den aber m Personen betreten wollen, wobei gilt $n < m$.

```
 1  // Kernel C-API enthält auch Mutex
 2  #include <cyg/kernel/kapi.h>
 3  #include <stdio.h>
 4  // für rand()
 5  #include <stdlib.h>
 6
 7  #define THREAD_STACK_SIZE (4096)
 8  #define THREAD_NUMBER     (10)
 9  #define ROOM_SIZE         (3)
10
11  cyg_uint8 gStackArray[THREAD_NUMBER][THREAD_STACK_SIZE];
12
13  cyg_thread   gThreadArray [THREAD_NUMBER];
14  cyg_handle_t gThreadHandleArray [THREAD_NUMBER];
15  cyg_thread_entry_t threadBody;
16  // Speicher für Semaphor-"Objekt" allokieren
17  cyg_sem_t gSemaphore;
18
19  //
20  void cyg_user_start ( void) {
21    cyg_uint32 i = 0;
22
23    for ( i = 0; i < THREAD_NUMBER; i++) {
24      cyg_thread_create(
25        4,
26        threadBody,
27        (cyg_addrword_t) i,
28        NULL,
29        (void *) gStackArray[i],
30        THREAD_STACK_SIZE,
31        &gThreadHandleArray[i],
32        &gThreadArray[i]);
33    }
34
35    // Semaphor initialisieren
36    cyg_semaphore_init (
37      &gSemaphore, // Zeiger auf Semaphor-"Objekt"
38      ROOM_SIZE);  // Initialisierungswert des internen Zählers
39
40    for ( i = 0; i < THREAD_NUMBER; i++) {
41      cyg_thread_resume ( gThreadHandleArray[i]);
```

```
42     }
43   }
44
45
46   //
47   void enterRoom ( cyg_uint32 pThreadNumber) {
48     cyg_count32 lRoomLeft = 0;
49
50     // versuche den internen Zähler des Semaphor zu dekrementieren
51     // ist der Zähler bereits Null, so blockiert diese Aufruf
52     cyg_semaphore_wait ( &gSemaphore);
53
54     // liefert die Anzahl der Threads, die noch im Raum Platz haben
55     // bzw. wird der Wert des internen Zählers geliefert
56     cyg_semaphore_peek ( &gSemaphore, &lRoomLeft);
57
58     printf ( " thread %i has entered room, %i threads in room\n",
59              pThreadNumber,
60              ( ROOM_SIZE - lRoomLeft));
61   }
62
63
64   //
65   void exitRoom ( cyg_uint32 pThreadNumber) {
66     // weckt einen auf diesen Semaphor wartenden Thread auf
67     // existiert kein solcher, so wird der interne Zähler
68     // des Semaphor um eins erhöht
69     cyg_semaphore_post ( &gSemaphore);
70     printf ( " thread %i has left room\n", pThreadNumber);
71   }
72
73
74   //
75   void threadBody ( cyg_addrword_t pThreadData) {
76     cyg_uint32 lThreadNumber = (cyg_uint32) pThreadData;
77
78     for ( ;;) {
79       cyg_thread_delay ( rand () % 1000);
80       enterRoom ( lThreadNumber);
81       cyg_thread_delay ( rand () % 500);
82       exitRoom  ( lThreadNumber);
83     }
84   }
```

Zusätzlich zu den im Beispiel verwendeten Semaphor-Funktionen gibt es noch `cyg_semaphore_trywait`, `cyg_semaphore_timed_wait` und `cyg_semaphore_destroy`. Die ersteren beiden - man ahnte es bereits - sind Alternativen zur in Zeile 52 verwendeten Funktion `cyg_semaphore_wait`, wiederum zeitlich begrenzt blockierend bzw. überhaupt nicht blockierend. Beide liefern `false`, wenn der interne Semaphor-Zähler bereits Null ist während man versucht auf den Semaphor zu zugreifen.
`cyg_semaphore_destroy` zersört den Semaphor und sollte nicht aufgerufen werden solange noch Threads auf diesen Semaphor warten.

```
1   #include <cyg/kernel/kapi.h>
2   #include <pkgconf/kernel.h>
3   #include <cyg/kernel/thread.hxx>
```

```
 4  #include <cyg/kernel/thread.inl>
 5  #include <cyg/kernel/sema.hxx>
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8
 9  #define THREAD_STACK_SIZE (4096)
10  #define THREAD_NUMBER     (10)
11  #define ROOM_SIZE         (3)
12
13  cyg_uint8 gStackArray[THREAD_NUMBER][THREAD_STACK_SIZE];
14  Cyg_Thread*  gThreadArray [THREAD_NUMBER];
15  cyg_thread_entry_t threadBody;
16  // Zeiger auf Semaphor-Objekt
17  Cyg_Counting_Semaphore* gSemaphore;
18
19
20  #ifdef __cplusplus
21  extern "C" {
22  #endif
23  void cyg_user_start ( void) {
24    for ( cyg_uint32 i = 0; i < THREAD_NUMBER; i++) {
25      gThreadArray[i] = new Cyg_Thread (
26                              4,
27                              threadBody,
28                              (CYG_ADDRWORD) i,
29                              NULL,
30                              (CYG_ADDRWORD) gStackArray[i],
31                              THREAD_STACK_SIZE);
32    }
33
34    gSemaphore = new Cyg_Counting_Semaphore ( ROOM_SIZE);
35
36    for ( cyg_uint32 i = 0; i < THREAD_NUMBER; i++) {
37      gThreadArray[i]->resume ();
38    }
39  }
40  #ifdef __cplusplus
41  }
42  #endif
43
44
45  void enterRoom ( cyg_uint32 pThreadNumber) {
46    cyg_count32 lRoomLeft = 0;
47
48    gSemaphore->wait ();
49    lRoomLeft = gSemaphore->peek ();
50
51    printf ( " thread %i has entered room, %i threads in room\n",
52             pThreadNumber,
53             ( ROOM_SIZE - lRoomLeft));
54  }
55
56
57  void exitRoom ( cyg_uint32 pThreadNumber) {
```

```
58    gSemaphore->post ();
59    printf ( " thread %i has left room\n", pThreadNumber);
60  }
61
62
63  void threadBody ( cyg_addrword_t pThreadData) {
64    cyg_uint32 lThreadNumber = (cyg_uint32) pThreadData;
65
66    for ( ;;) {
67      Cyg_Thread::self()->delay ( rand () % 1000);
68      enterRoom ( lThreadNumber);
69      Cyg_Thread::self()->delay ( rand () % 500);
70      exitRoom  ( lThreadNumber);
71    }
72  }
```

Die C++-Version ist wenig spektakulär, wenn man davon absieht, dass die Semaphor-Klasse `Cyg_Counting_Semaphor` heißt.

3.6.3 Condition-Variable

3.6.4 Flags

3.7 Zeit, Uhren und Alarme

3.7.1 Uhrzeit abfragen

3.7.2 Alarme benutzen

3.8 Einen Interrupt-Handler schreiben

Die Interrupt-Behandlung zerfällt in eCos normalerweise in zwei einzelne Funktionen. Die zuerst beim Auftreten eines Interrupts aufgerufene Funktion nennt man ISR, für Interrupt-Service-Routine. Während diese Funktion abgearbeitet wird, kann das System keine weiteren Interrupts - auch nicht von anderen Quellen - verarbeiten, d.h. eine ISR sollte möglichst kompakt sein und nur das nötigste machen. Komplexere Teile einer Interrupt-Behandlung sollten in eine DSR - für Deferred-Service-Routine - ausgelagert werden. *Ob* - wenn die Interrupt-Behandlung nur sehr kurz ist kann man auch ohne DSR auskommen - eine DSR aufgerufen wird, hängt vom Rückgabewert der ISR ab, es sind spezielle Konstanten dafür definiert. *Wann* die DSR aufgerufen wird, entscheidet der Scheduler. Ist zwischendurch ein weiterer Interrupt aufgetreten, so wird erst dessen ISR aufgerufen, bevor die DSR des vorher aufgetretenen Interrupts aufgerufen wird. Insgesamt führt dieses Konzept zu sehr geringen Latenz-Zeiten und vermeidet, dass Interrupts verloren gehen.
Das folgende Beispiel-Programm ist für ein SA1110-Modul und ein passendes Entwicklungs-Board geschrieben. Der erste Tast-Schalter links neben dem Modul-Sockel ist mit dem PIO-Eingang Nummer eins der StrongARM-CPU verbunden. Jeder Druck auf den Schalter erzeugt - mindestens, denn der Schalter prellt - eine fallende Flanke an diesem Eingang. Das Programm konfiguriert die PIO- und den Interrupt-Controller der StrongARM-CPU so, dass diese fallende Flanke als Interrupt erkannt wird.

```
1  #include <cyg/kernel/kapi.h>
2  // Konstanten für StrongARM 1110
3  #include <cyg/hal/hal_sa11x0.h>
4  #include <stdio.h>
5
6  #define THREAD_STACK_SIZE (4096)
7
```

```
 8  char gStack[THREAD_STACK_SIZE];
 9
10  cyg_thread gPrintThread;
11  cyg_handle_t gPrintThreadHandle;
12  cyg_thread_entry_t printBody;
13  // Anzahl der Tastendrücke
14  cyg_uint32 gInterruptCount = 0;
15  // zeigt, ob ISR allein oder ISR+DSR ausgeführt wurden
16  cyg_int32  gSum = 0;
17  // Vorwärts-Deklaration der beiden Interrupt-Behandlungs-Funktionen
18  cyg_ISR_t isrPressButton;
19  cyg_DSR_t dsrPressButton;
20  // Speicher für Interrupt-"Objekt"
21  cyg_interrupt gIntr;
22
23
24  //
25  // Interrupt-Service-Routine für Druck auf den Taster an GPIO1
26  //
27  // pIntrVector  - Interrupt-Vektor
28  // pDataAddress - Zeiger auf Handler-spezifische Daten
29  //
30  // cyg_uint32   - Rückgabewert, zeigt an, ob DSR aufgerufen werden soll
31  //
32  cyg_uint32 isrPushButton ( cyg_vector_t   pIntrVector,
33                             cyg_addrword_t pDataAddress)
34  {
35    // aktuellen Interrupt sperren
36    cyg_interrupt_mask ( pIntrVector);
37    // aktuellen Interrupt bestätigen
38    cyg_interrupt_acknowledge ( pIntrVector);
39
40    gInterruptCount++;
41
42    // wenn gSum ungerade ist, wird die DSR aufgerufen
43    // ansonsten wird gSum inkrementiert und die
44    // Interrupt-Behandlung ist beendet
45    if ( gSum % 2) {
46      return ( CYG_ISR_HANDLED | CYG_ISR_CALL_DSR);
47    } else {
48      gSum++;
49
50      // Interrupt entsperren
51      cyg_interrupt_unmask ( pIntrVector);
52      return ( CYG_ISR_HANDLED);
53    }
54  }
55
56
57  //
58  // Deferred-Service-Routine für Taster an GPIO1
59  //
60  // pIntrVector  - Interrupt-Vektor
61  // pCounter     - Anzahl der bisher angefallenen Interrupts (unbenutzt)
```

```
 62  // pDataAddress - Zeiger auf Handler-spezifische Daten
 63  //
 64  void dsrPushButton ( cyg_vector_t   pIntrVector,
 65                       cyg_ucount32   pCounter,
 66                       cyg_addrword_t pDataAddress)
 67  {
 68    gSum += 3;
 69
 70    // diesen Interrupt wieder frei geben
 71    cyg_interrupt_unmask ( pIntrVector);
 72  }
 73
 74
 75  //
 76  void setupInterruptHandler () {
 77    cyg_handle_t   lIntrHandle;
 78    cyg_vector_t   lIntrVector;
 79    cyg_priority_t lIntrPriority;
 80
 81    lIntrPriority = 99;
 82    lIntrVector = CYGNUM_HAL_INTERRUPT_GPIO1; // Konstante aus hal_sa11x0.h
 83
 84    // ein Interrupt-"Objekt" im Kernel erzeugen
 85    cyg_interrupt_create (
 86      lIntrVector,        // Interrupt-Vektor (Interrupt-Nummer)
 87      lIntrPriority,      // Interrupt-Priorität (auf ARM-CPUs unbenutzt)
 88      (cyg_addrword_t) 0, // Zeiger auf Handler-spezifische Daten,
 89                          // z.B. auf einen Datenpuffer
 90      isrPushButton,      // ISR-Funktion
 91      dsrPushButton,      // DSR-Funktion
 92      &lIntrHandle,       // Interrupt-Handle
 93      &gIntr);            // Zeiger auf (Speicher für) Interrupt-"Objekt",
 94                          // muss global oder statisch sein
 95
 96    // Interrupt konfigurieren
 97    // der Schalter erzeugt eine fallende Flanke am I/O-Port
 98    // der Interrupt-Controller wird eingestellt auf eine fallende Flanke
 99    // zu reagieren
100    cyg_interrupt_configure (
101      lIntrVector, // Interrupt-Nummer
102      false,       // reagiere auf Flanke, nicht Level
103      false);      // reagiere auf fallend, nicht steigend
104
105    // Handler an den Interrupt binden
106    cyg_interrupt_attach ( lIntrHandle);
107
108    // Interrupt entsperren
109    cyg_interrupt_unmask ( lIntrVector);
110  }
111
112
113
114  //
115  void cyg_user_start ( void) {
```

```
116   cyg_thread_create( 4,
117                      printBody,
118                      (cyg_addrword_t) 0,
119                      "Print Thread",
120                      (void *) gStack,
121                      THREAD_STACK_SIZE,
122                      &gPrintThreadHandle,
123                      &gPrintThread);
124
125   setupInterruptHandler ();
126   cyg_thread_resume ( gPrintThreadHandle);
127 }
128
129
130 //
131 void printBody ( cyg_addrword_t pThreadData) {
132   cyg_int32 lOldSum = gSum - 1;
133   cyg_uint32 lIterationCount = 0;
134
135   for ( ;; lIterationCount++) {
136     if ( lOldSum != gSum) {
137       printf (
138         " gSum = %i, %d interrupts so far, %d main loop iterations\n",
139         gSum,
140         gInterruptCount,
141         lIterationCount);
142       lOldSum = gSum;
143     }
144   }
145 }
146
```

Die C++-Version unterscheidet sich hier nur unwesentlich von der C-Version. Insbesondere die eigentlichen Handler-Methoden `isrPushButton` und `dsrPushButton` sind fast identisch zu C-Version. Dass die meisten Methoden der Klasse `Cyg_Interrupt` statisch sind, verursacht eher mehr Tipp-Aufwand und schlechtere Übersichtlichkeit als in der C-Version.

```
 1 #include <cyg/kernel/kapi.h>
 2 #include <cyg/hal/hal_sa11x0.h>
 3 #include <pkgconf/kernel.h>
 4 #include <cyg/kernel/thread.hxx>
 5 #include <cyg/kernel/thread.inl>
 6 // Interrupt-Klasse
 7 #include <cyg/kernel/intr.hxx>
 8 #include <stdio.h>
 9
10 #define THREAD_STACK_SIZE (4096)
11
12 char gStack[THREAD_STACK_SIZE];
13
14 Cyg_Thread* gPrintThread;
15 cyg_uint32  gInterruptCount = 0;
16 cyg_int32   gSum = 0;
17 // Zeiger auf Interrupt-Objekt
18 Cyg_Interrupt* gInterrupt;
```

```
19
20
21  //
22  cyg_uint32 isrPushButton ( cyg_vector   pIntrVector,
23                             CYG_ADDRWORD pDataAddress)
24  {
25    Cyg_Interrupt::mask_interrupt ( pIntrVector);
26    Cyg_Interrupt::acknowledge_interrupt ( pIntrVector);
27
28    gInterruptCount++;
29
30    if ( gSum % 2) {
31      return ( CYG_ISR_HANDLED | CYG_ISR_CALL_DSR);
32    } else {
33      gSum++;
34      Cyg_Interrupt::unmask_interrupt ( pIntrVector);
35      return ( CYG_ISR_HANDLED);
36    }
37  }
38
39
40  //
41  void dsrPushButton ( cyg_vector   pIntrVector,
42                       cyg_ucount32 pCounter,
43                       CYG_ADDRWORD pDataAddress)
44  {
45    gSum += 3;
46
47    // diesen Interrupt wieder frei geben
48    // gInterrupt->unmask_interrupt () ist identisch zu
49    // Cyg_Interrupt::unmask_interrupt () da dies eine
50    // statische Methode ist
51    gInterrupt->unmask_interrupt ( pIntrVector);
52  }
53
54
55  //
56  void setupInterruptHandler () {
57    cyg_vector   lIntrVector;
58    cyg_priority lIntrPriority;
59
60    lIntrPriority = 99;
61    lIntrVector = CYGNUM_HAL_INTERRUPT_GPIO1; // Konstante aus hal_sa11x0.h
62
63    // Interrupt-Objekt instanziieren
64    gInterrupt = new Cyg_Interrupt (
65                       lIntrVector,
66                       lIntrPriority,
67                       (CYG_ADDRWORD) 0, // Zeiger auf Handler-spezifische Daten
68                       isrPushButton,
69                       dsrPushButton);
70
71    // configure_interrupt ist eine statische Methode
72    // alternativ wäre natürlich auch
```

```
 73    // Cyg_Interrupt::configure_interrupt (...) möglich
 74    gInterrupt->configure_interrupt ( lIntrVector, false, false);
 75    // attach() ist zur Abwechslung mal nicht statisch
 76    gInterrupt->attach ();
 77    // unmask_interrupt ist eine statische Methode
 78    gInterrupt->unmask_interrupt ( lIntrVector);
 79  }
 80
 81
 82  //
 83  void printBody ( cyg_addrword_t pThreadData) {
 84    cyg_int32 lOldSum = gSum - 1;
 85    cyg_uint32 lIterationCount = 0;
 86
 87    for ( ;; lIterationCount++) {
 88      if ( lOldSum != gSum) {
 89        printf (
 90          " gSum = %i, %d interrupts so far, %d main loop iterations\n",
 91          gSum,
 92          gInterruptCount,
 93          lIterationCount);
 94        lOldSum = gSum;
 95      }
 96    }
 97  }
 98
 99
100  //
101  #ifdef __cplusplus
102  extern "C" {
103  #endif
104  void cyg_user_start ( void) {
105    gPrintThread = new Cyg_Thread (
106                         4,
107                         printBody,
108                         (cyg_addrword_t) 0,
109                         "Print Thread",
110                         (cyg_addrword_t) gStack,
111                         THREAD_STACK_SIZE);
112
113    setupInterruptHandler ();
114
115    gPrintThread->resume ();
116  }
117  #ifdef __cplusplus
118  }
119  #endif
```

Die Verwendung „komplexer“Funktionen, wie etwa `printf` und anderer Funktionen aus `stdio.h` oder `stdlib.h`, innerhalb von ISR oder DSR ist nicht erlaubt. Diese Funktionen sind normalerweise so implementiert, diese Eigenschaft ist auch konfigurierbar, dass sie thread-safe sind, also Synchronalisations-Funktionalität des Kernels benötigen. Während der Interrupt-Verarbeitung können jedoch nur bestimmte Kernel-Funktionen aufgerufen werden (siehe hierfür auch [1]).
Außerdem benötigt z.B. `printf` erheblich Zeit, da die Ausgabe ja über eine langsame Netzwerk-Verbindung erfolgt. Das kann dazu führen, dass Interrupts verloren gehen.

3.9 Gerätetreiber benutzen

Für einige Geräte, wie z.B. die seriellen Schnittstellen, gibt es unter eCos „richtige“ Gerätetreiber, so wie man es von Unix-Systemen gewohnt ist. D.h. der Treiber bildet das Gerät aus Applikationssicht als *Datei* mit den Operationen *öffnen*, *lesen*, *schreiben* und *konfigurieren* ab.
Da die Schnittstellen-Funktionen von Gerätetreibern immer C-Funktionen sind, diesmal nur ein Beispiel (in C++). Das Programm benutzt die zweite serielle Schnittstelle und schaltet diese in den Loopback-Modus, was bedeutet, dass die über diese Schnittstelle gesendeten Daten direkt wieder von der selben Schnittstelle empfangen werden können. Der Loopback-Modus entspricht also einer Kurzschluss-Brücke zwischen Sende- und Empfangs-Leitung. Das Programm implementiert ein einfaches Echo mit einem Sender- und einem Empfänger-Thread.

```
 1  #include <pkgconf/kernel.h>
 2  #include <stdio.h>
 3  #include <cyg/kernel/thread.hxx>
 4  #include <cyg/kernel/thread.inl>
 5  #include <cyg/kernel/kapi.h>
 6  #include <cyg/hal/hal_sa11x0.h>
 7  // für cyg_io_* Funktionen
 8  #include <cyg/io/io.h>
 9  // für Konstanten für Baudrate, Parität etc.
10  #include <cyg/io/ttyio.h>
11  // für die Konfigurations-Schlüssel für cyg_io_set_config
12  #include <cyg/io/config_keys.h>
13
14  #define THREAD_STACK_SIZE (4096)
15
16
17  Cyg_Thread* gReader;
18  Cyg_Thread* gWriter;
19  cyg_thread_entry serialReader;
20  cyg_thread_entry serialWriter;
21  char gReaderStack [THREAD_STACK_SIZE];
22  char gWriterStack [THREAD_STACK_SIZE];
23  // Handle für das benutzte Gerät
24  cyg_io_handle_t gTTYDevHandle;
25
26  volatile unsigned int& PPC_PIN_ASSIGNMENT_REG =
27    *(volatile unsigned int*) SA11X0_PPC_PIN_ASSIGNMENT;
28  // Register für alternative Funktionalität von UART1
29  volatile cyg_uint8& GP_CLOCK_REG = *(volatile cyg_uint8*) 0x80020060;
30  // UART1 Control-Register 3
31  volatile cyg_uint8& UART1_CTRL3_REG =
32    *(volatile cyg_uint8*) SA11X0_UART1_CONTROL3;
33
34
35  //
36  void serialWriter ( cyg_addrword_t pDataAddress) {
37    Cyg_ErrNo  lError = 0;
38    cyg_uint32 lLength = 0;
39    cyg_uint32 lCounter = 0;
40    char* lString =
41      "I never gonna work another day in my life.\
42       The Gods told me to relax, \
43       they said I'm gonna get fixed up right.\
```

```
44       I never gonna work another day in my life.\
45       I'm way to busy powertripping,\
46       but I gonna sched you some light.";
47
48    for ( ;;) {
49      lLength = (cyg_uint32) strlen ( lString);
50
51      // blockiert so lange, bis die zu sendenden Zeichen an den
52      // Treiber übergeben werden konnten
53      lError  = cyg_io_write (
54                  gTTYDevHandle,
55                  lString,
56                  &lLength);
57
58      printf ( " error-code = %i, try = %d\n", lError, lCounter);
59      lCounter++;
60      if ( lError != ENOERR) {
61        printf ( " write failed\n");
62      }
63    }
64  }
65
66
67  //
68  void serialReader ( cyg_addrword_t aDataAddress) {
69    cyg_uint32 lBufferSize = 4;
70    char lBuffer [lBufferSize + 1];
71    cyg_uint32 lLength = lBufferSize;
72    Cyg_ErrNo lError = 0;
73    unsigned int lCounter = 0;
74
75    for ( ;;) {
76      lLength = lBufferSize;
77
78      // liest Zeichen vom Gerät, dieser Aufruf blockiert
79      // so lange bis so viele Zeichen gelesen werden
80      // konnten, wie es im Parameter lLength angegeben wurde
81      // kehrt die Funktion zurück, obwohl nicht genügend Zeichen gelesen
82      // werden konnten, so liefert die Funktion EAGAIN zurück
83      // in lLength steht danach die Anzahl der tatsächlich gelesenen
84      // Zeichen
85      lError = cyg_io_read (
86                 gTTYDevHandle, // Device-Handle
87                 lBuffer,       // Zeiger auf Puffer in dem empfangene
88                                // Daten abgelegt werden sollen
89                 &lLength);     // Eingabe: Anzahl Zeichen, die gelesen
90                                // werden sollen
91                                // Rückgabe: Anzahl Zeichen, die gelesen
92                                // wurden
93      lBuffer[lBufferSize] = 0;
94      printf ( " read %d \"%s\" %d\n", lCounter, lBuffer, lLength);
95      lCounter++;
96    }
97  }
```

```
 98
 99
100  //
101  // Diese Funktion konfiguriert die SA1110-CPU so, dass UART1
102  // im UART-Modus ist und schaltet diese UART in den Loopback-
103  // Betrieb.
104  // Diese Funktion ist nur für dieses spezielle Test-Programm
105  // wichtig.
106  //
107  void setupSerialHardware () {
108    GP_CLOCK_REG &= ( 0xFF ^ 0x1);
109    GP_CLOCK_REG &= ( 0xFF ^ ( ( 1 << 4) | ( 1 << 5)));
110    GP_CLOCK_REG |= 0x1;
111    PPC_PIN_ASSIGNMENT_REG &= ( 0xFFFFFFFF ^ ( 1 << 12));
112    UART1_CTRL3_REG |= SA11X0_UART_LOOPBACK_MODE;
113  }
114
115
116
117  //
118  // Initialisierung der UART auf Gerätetreiber-Ebene
119  //
120  void setupSerialDevice () {
121    Cyg_ErrNo lError = 0;
122    cyg_serial_info_t lSerialInfo;
123    cyg_uint32 lLength = 0;
124
125    // Gerät öffnen
126    lError = cyg_io_lookup (
127               "/dev/ser1",     // Geräte-Name
128               &gTTYDevHandle); // Zeiger auf ein Handle
129
130    if ( lError != ENOERR) {
131      printf ( " device lookup failed\n");
132      return;
133    } else {
134      printf ( " handle created\n");
135    }
136
137    lSerialInfo.baud = CYGNUM_SERIAL_BAUD_300;
138    lSerialInfo.stop = CYGNUM_SERIAL_STOP_1;
139    lSerialInfo.parity = CYGNUM_SERIAL_PARITY_NONE;
140    lSerialInfo.word_length = CYGNUM_SERIAL_WORD_LENGTH_8;
141
142    // Gerät konfigurieren
143    lError = cyg_io_set_config (
144               gTTYDevHandle,                 // Geräte-Handle
145               CYG_IO_SET_CONFIG_SERIAL_INFO, // Konfig.-Schlüssel,
146                                              // zeigt an welche Art von
147                                              // Konfig.-Daten übergeben
148                                              // werden
149               &lSerialInfo,                  // Zeiger auf Konfig.-Daten
150               &lLength);                     // für bestimmte Konfig.-
151                                              // Schlüssel, hier unbenutzt
```

```
152
153   if ( lError != ENOERR) {
154     printf ( " set_config failed\n");
155   } else {
156     printf ( " device configured\n");
157   }
158 }
159
160
161 #ifdef __cplusplus
162 extern "C" {
163 #endif
164 void cyg_user_start ( void) {
165   gReader = new Cyg_Thread ( CYG_SCHED_DEFAULT_INFO,
166                              serialReader,
167                              (cyg_addrword_t) 0,
168                              "SerialReader",
169                              (cyg_addrword_t) gReaderStack,
170                              THREAD_STACK_SIZE);
171
172   gWriter = new Cyg_Thread ( CYG_SCHED_DEFAULT_INFO,
173                              serialWriter,
174                              (cyg_addrword_t) 0,
175                              "SerialWriter",
176                              (cyg_addrword_t) gWriterStack,
177                              THREAD_STACK_SIZE);
178   setupSerialHardware ();
179   setupSerialDevice   ();
180
181   gWriter->resume ();
182   gReader->resume ();
183 }
184 #ifdef __cplusplus
185 }
186 #endif
```

Anhang A

Das eCos Komponenten- und Konfigurations-Konzept

eCos ist in Komponenten und Pakete aufgeteilt. Zusätzlich sind Parameter vieler dieser Komponenten konfigurierbar. Beispielsweise kann man zwischen verschiedenen Schedulern wählen und Parameter dieser Scheduler, z.B. die Länge einer Zeitscheibe, konfigurieren. Zu diesem Zweck hat RedHat die *Component Description Language*(CDL) geschaffen. +++ FIXME blablabla und CDL-Beispiel +++

Anhang B

ecosconfig

B.1 Linux

Der allgemeine Aufbau eines ecosconfig-Aufrufs ist:

ecosconfig <OPTION> <KOMMANDO>

ecosconfig-Kommandos:

list	Zeigt eine Liste aller Pakete und Templates, die in der Datenbank-Datei ecos.db aufgeführt sind. Für alle Pakete, die zwar in ecos.db aufgeführt, aber nicht im eCos-Repository zu finden sind, wird eine Warnung ausgegeben.
new <TARGET> <TEMPLATE> <<VERSION>>	Dieses Kommando erzeugt eine neue Konfigurationsdatei ecos.ecc für die Ziel-Hardware <TARGET> mit dem angegebenen Template. Beispiel: Der Befehl ecosconfig pc redboot erzeugt ecos.ecc den RedBoot-ROM-Monitor für einen PC
add <PAKET> <<PAKET_2>...<PAKET_N>>	Dieses Kommando fügt der Konfiguration ein oder mehrere Paket hinzu.
tree	Dieses Kommando erzeugt einen Dateibaum aus dem libtarget.a bzw. ein ROM-Monitor kompiliert werden und erzeugt einen Include-Datei-Baum, der die Schnittstellen für Applikationen für die Plattform enthält. Außerdem wird das Makefile zum Kompilieren von libtarget.a bzw. eines ROM-Monitors erzeugt.

Anhang C

Probleme und Lösungen

C.1 Die Applikation wird beendet, wenn alle User-Threads blockiert sind

Angenommen man hat eine Multithreaded-Applikation entwickelt und es tritt der Fall auf, dass alle User-Threads der Applikation blockiert sind, da sie auf irgendwelche Bedingungen, z.B. I/O, warten. In diesem Fall sollte eigentlich der Idle-Thread des Kernels die Rechenzeit „verbrauchen" bis irgendein anderer Thread wieder lauffähig wird.
Verblüffenderweise tritt bei einer Fehlkonfiguration der Effekt auf, dass die Applikation regulär endet, wenn alle User-Threads blockiert und nur der Idle-Thread lauffähig ist. Dies verhindert man, indem man die Konfigurations-Optionen CYGSEM_LIBC_STARTUP_MAIN_INITCONTEXT auf user_value 1 und CYGSEM_LIBC_STARTUP_MAIN_THREAD in ecos.ecc auf user_value 0 setzt.
Allerdings darf man in `main` dann keine libc-Funktionen aufrufen. Da man normalerweise aber `cyg_user_start` statt `main` als Einstiegspunkt in die Applikation benutzt, spielt diese Einschränkung nur eine untergeordnete Rolle.

C.2 Wie man die seriellen Schnittstellen aktiviert

In den üblichen Konfigurations-Templates default oder auch elix ist zwar die Unterstützung für die seriellen Schnittstellen enthalten, jedoch nicht aktiviert. Typischerweise meldet `cyg_io_lookup` in diesem Fall einen Fehler.
Damit man die seriellen Schnittstellen des Moduls auch nutzen kann, ist die Option CYGPKG_IO_SERIAL_DEVICES auf user_value 1 zu setzen. Außerdem ist natürlich die Unterstützung für jede einzelne von der Applikation benutzte serielle Schnittstelle zu aktivieren. Bei StrongARM 1110 basierten Systemen aktiviert man die zweite serielle Schnittstelle, indem man die Option CYGPKG_IO_SERIAL_ARM_SA11X0_SERIAL1 auf user_value 1 setzt.

C.3 Download vom Host- zum Ziel-Rechner bricht ab

Wenn der Download zur Ziel-Hardware erst normal läuft, dann „stottert" und schließlich ganz zusammenbricht, so kann das an einem unpassenden Linker-Skript für RAM-Startup liegen[1]. In diesem Fall sollte man das Skript genau prüfen, ob es wirklich zum RAM-Startup für die Ziel-Hardware passt. Ist das der Fall, so kann man ausprobieren den Beginn der Sektion .rom_vectors in Richtung größerer Adressen zu verschieben, wobei die Start-Adress 0x20000 normalerweise ausreichen sollte. Ist die Änderung erfolgreich, so sollte man neben target.ld auch die entsprechende Linker-Skript-Schablone im Quell-Datei-Baum des gegenwärtig benutzten eCos-Ports entsprechend ändern. Die entsprechenden Dateien heißen meist mlt_<CPU_ARCHITEKTUR>_<PLATTFORM>_ram.ldi und mlt_<CPU_ARCHITEKTUR>_<PLATTFORM>_ram.h.

[1] Ist die Applikation auch für RAM-Startup gelinkt?

Anhang D

Einen ROM-Monitor kompilieren

Einen ROM-Monitor zu „bauen“ ist völlig analog zur Erzeugung von libtarget.a in Abschnitt 2.2.1 bzw. 2.2.2. Man muss lediglich ein anderes Template statt default benutzen.

Um einen GDB-Stub zu erzeugen wählt man das Template stubs. Der GDB-Stub ist zwar nicht besonders leistungsfähig, reicht jedoch in der Zeit der Applikations-Entwicklung aus und hat den Vorteil nur wenig nicht-flüchtigen Speicher zu belegen. Ein guter Name für das Basis-Verzeichnis dieses Build-Verzeichnis-Baumes ist <PLATTFORM>_stub, also z.B. sa1110dimm_stub. Statt libtarget.a erzeugt der Build-Prozess hier die Datei gdb_module.bin im Verzeichnis <PLATTFORM>_stub/install/bin, die man direkt ins Boot-ROM programmieren muss.

Um den RedBoot ROM-Monitor zu erzeugen wählt man das Template redboot. Will man RedBoot benutzen um Applikations-Images zu verwalten, sollten noch die notwendigen Flash-Treiber zur Konfiguration hinzugefügt werden. Benutzt man statt der RS232-Verbindung eine Ethernet-Verbindung, so sind natürlich auch noch die Ethernet-Treiber zur Konfiguration hinzuzufügen. Ein guter Name für das Basis-Verzeichnis des Build-Verzeichnis-Baumes ist <PLATTFORM>_redboot, also z.B. sa1110dimm_redboot. Statt libtarget.a erzeugt der Build-Prozess hier die Datei redboot.bin im Verzeichnis <PLATTFORM>_redboot/install/bin, die man direkt ins Boot-ROM programmieren muss.

Anhang E

RedBoot

RedBoot ist der Standard ROM-Monitor von RedHat. Er basiert, genau wie der gdb-Stub, auf dem eCos-HAL, was den angenehmen Vorteil hat, dass man auch RedBoot hat, wenn man einen eCos-Port für die Ziel-Hardware hat.
Neben den Fähigkeiten des gdb-Stubs, nämlich ein Programm vom gdb empfangen zu können und Remote-Debugging zu ermöglichen - insofern ist der gdb-Stub eine Untermenge der RedBoot-Funktionalität, bietet RedBoot noch weitere Funktionalität. Man kann sich per Terminal-Programm, unter Linux z.B. minicom, mit einem RedBoot-Target verbinden und dort interaktiv Dinge tun. Am interessantesten ist hierbei die Verwaltung des Target-Flash-Speichers. Ist RedBoot mit Flash-Unterstützung konfiguriert, wobei auch hier die eCos-Treiber für Flash-Speicher verwendet werden, so kann man verschiedene Flash-Images per Terminal-Programm auf das Target herunterladen und in den Flash-Speicher programmieren.
Abbildung E.1 zeigt die RedBoot-Ausgaben beim Booten. Ist das System gebootet, so erscheint der Prompt RedBoot> und man kann Kommandos eingeben. help führt zu einer Ausgabe aller verfügbaren Kommandos.
RedBoot benutzt ein FIS (Flash-Image-System) genanntes Pseudo-Datei-System zur Verwaltung des Flash-Speichers. D.h., dass man die Abbilder(Images) mehrerer Applikationen im Flash-Speicher ablegen und eins davon zur automatischen Ausführung nach dem Booten auswählen kann.
Abbildung E.3 zeigt als Ergebnis von fis list den Inhalt des Flash-Speichers ohne Applikations-Images. Am Ende des Flash-Speichers legt RedBoot Verwaltungs-Informationen über den Inhalt des Speichers ab.
Man kann Images mit einem Terminal-Programm per X-, Y- oder Z-Modem-Protokoll auf das Target herunterladen und RedBoot kann diese in den Flash-Speicher programmieren. Ein Programmiergerät ist, sobald RedBoot auf einer Plattform zuverlässig läuft, nicht mehr notwendig. Ist RedBoot mit TCP/IP-Unterstützung kompiliert, so kann man Images auch per TFTP-Protokoll von einem (TFTP-)Host herunterladen.

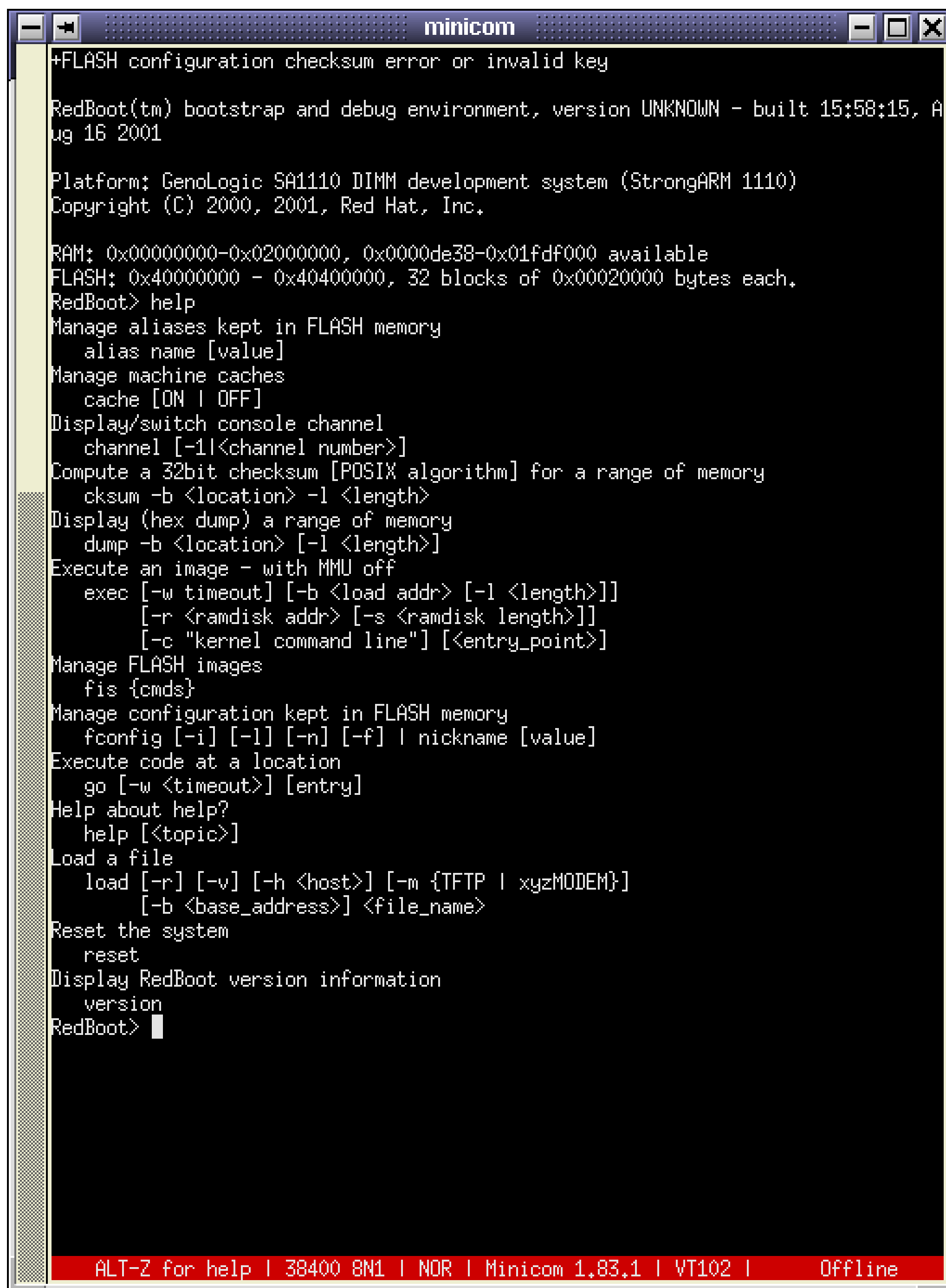

Abbildung E.1: Boot-Ausgaben/Hilfe-Übersicht zu allen RedBoot-Kommandos

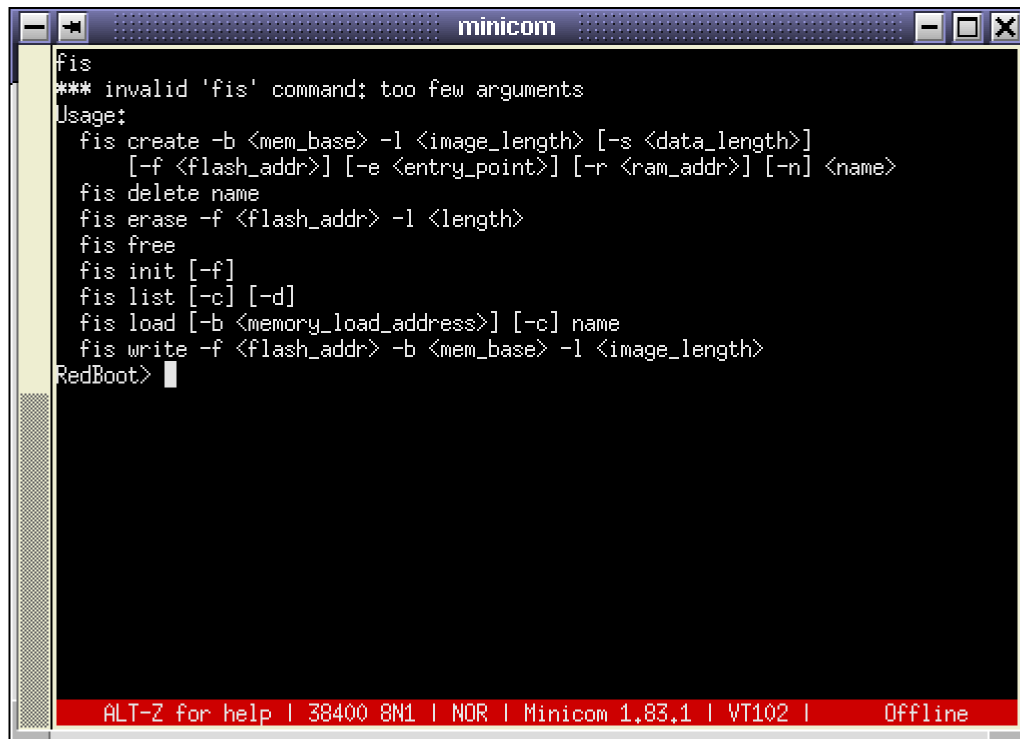

Abbildung E.2: Hilfe-Übersicht zur Flash-Image-Verwaltung fis

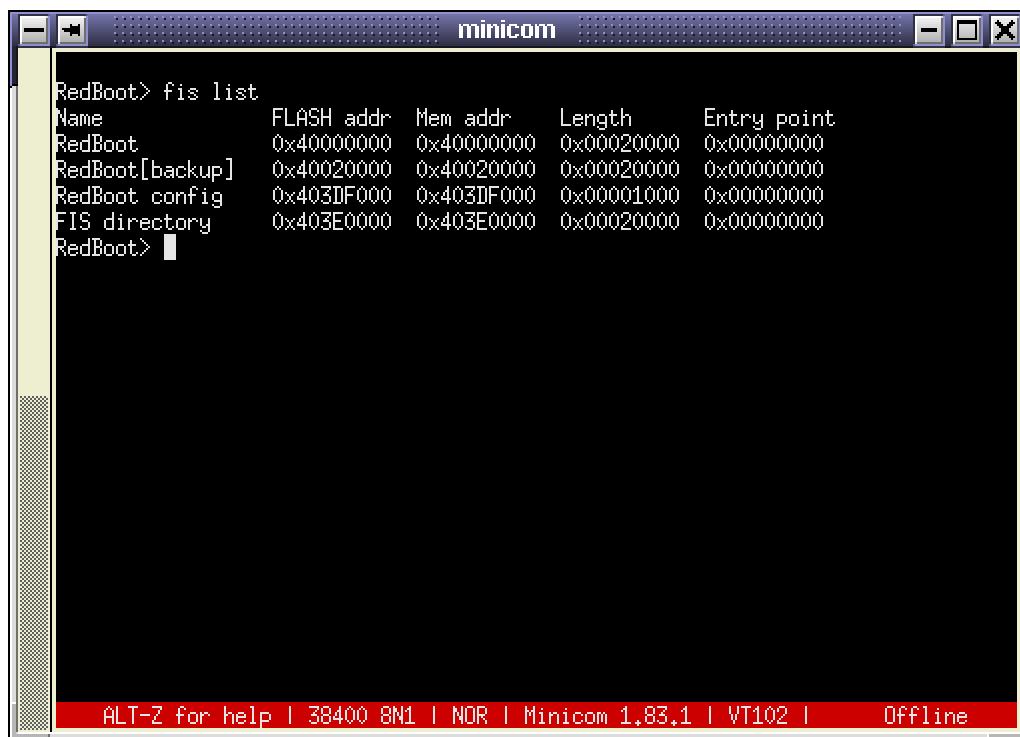

Abbildung E.3: Inhalt des von fis verwalteten Flash-Speichers

Literaturverzeichnis

[1] RedHat-eCos-Team:
eCos Reference Manual
http://sources.redhat.com/ecos/docs-latest/ref/ecos-ref.1.html
http://sources.redhat.com/ecos/docs-latest/pdf/ecos-ref.pdf

[2] RedHat-eCos-Team:
eCos User's Guide
http://sources.redhat.com/ecos/docs-latest/guides/user-guides.1.html
http://sources.redhat.com/ecos/docs-latest/pdf/user-guides.pdf

[3] Bart Veer, John Dallaway:
The eCos Component Writer's Guide
http://sources.redhat.com/ecos/docs-latest/cdl/cdl-guide.html
http://sources.redhat.com/ecos/docs-latest/pdf/cdl-guide.pdf

[4] **Archiv der eCos Mailing-Liste**
http://sources.redhat.com/ml/ecos-discuss

[5] **RedBoot User's Guide**
http://sources.redhat.com/ecos/docs-latest/redboot/redboot.html
http://sources.redhat.com/ecos/docs-latest/redboot/redboot.pdf

[6] **Die CVS-Seite**
http://www.cvshome.org

[7] W. Richard Stevens:
Programmierung in der UNIX-Umgebung
Addison-Wesley Verlag, ISBN 3-89319-814-8

[8] Andreas Bürgel:
Die GNUPro Tool-Chain
http://www.andreas-buergel.de/documents/gnupro_toolchain.pdf