

Anwendungsentwicklung mit Ecos

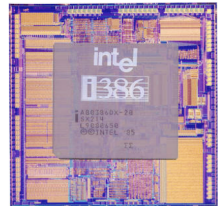
Übungen zur Vorlesung

Florian Franzmann, Martin Hoffmann, Tobias Klaus

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
`www4.informatik.uni-erlangen.de`

28. Oktober 2013

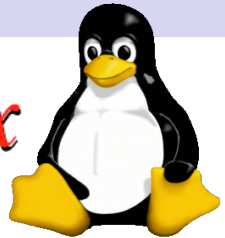
Prozessorvielfalt in der Echtzeitwelt



Noch mehr Betriebssysteme

free **RTOS**

μClinux



ecos

QNX



μC/OS-II™
The Real-Time Kernel

HIGH TEC

eCos

Embedded Configurable Operating System

“eCos is an embedded, highly configurable, open-source, royalty-free, real-time operating system”

- Ursprünglich von der Fa. Cygnus Solutions entwickelt (1997)
- Primäres Entwurfsziel:
 - “deeply embedded systems”
 - “high-volume application”
 - “consumer electronics, telecommunications, automotive, ...”
- Zusammenarbeit mit Redhat (1999)
- Seit 2002 quelloffen (GPL)

 <http://ecos.sourceware.org>



Unterstützte Plattformen

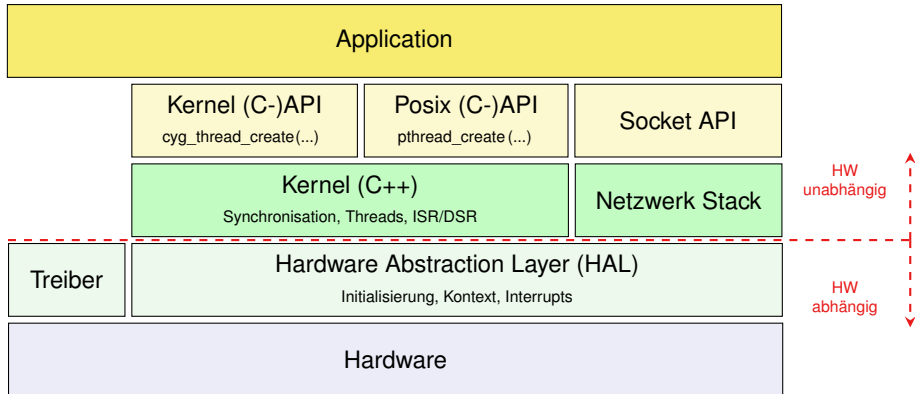
<http://www.ecoscentric.com/ecos/examples.shtml>

- Fujitsu SPARCite
- Matsushita MN10300
- Motorola PowerPC
- Advanced RISC Machines (ARM)
 - Toshiba TX39
 - Intel Strong ARM
 - Infineon TriCore
 - Hitachi SH3
 - NEC VR4300
 - MB8683X
 - Intel x86
 - ...



eCos Systemarchitektur

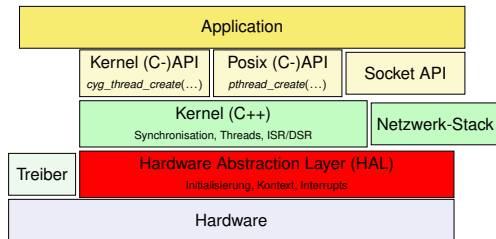
Überblick



eCos Systemarchitektur

Hardware Abstraction Layer

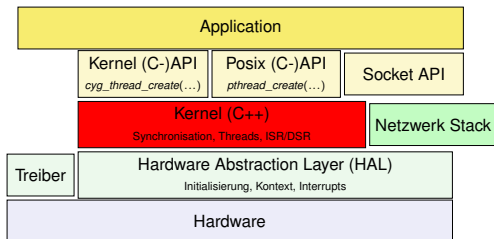
- Abstrahiert CPU- und plattformspezifische Eigenschaften
 - Kontextwechsel
 - Interruptverwaltung
 - CPU-Erkennung, Startup
 - Zeitgeber, I/O-Registerzugriffe



eCos Systemarchitektur

Kernel

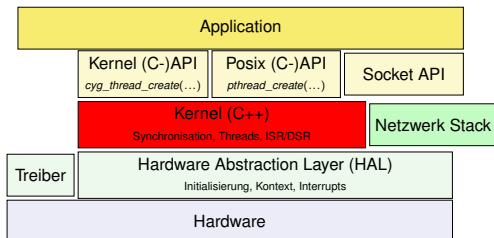
- Implementiert in C++
- Feingranular konfigurierbar
 - Verschiedene Schedulingstrategien (Bitmap/Multilevel Queue)
 - Zeitscheibenbasiert, präemptiv, prioritätenbasiert
- Verschiedene Synchronisationsstrategien
 - Mutexe, Semaphore, Bedingungsvariablen
 - Messages Boxes



eCos Systemarchitektur

Kernel – Interruptbehandlung

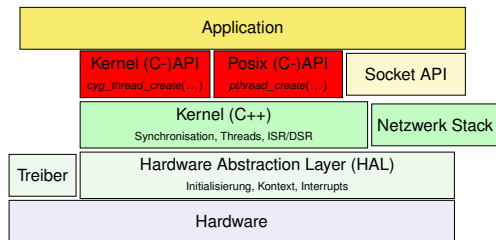
- Aufteilung wie in OoStubs
- Interrupt Service Routine (ISR)
 - Unverzögliche Ausführung
 - Asynchron
 - Kann DSR anfordern
- Deferred Service Routine (DSR)
 - Verzögerte Ausführung (beim Verlassen des Kernels)
 - Synchron



eCos Systemarchitektur

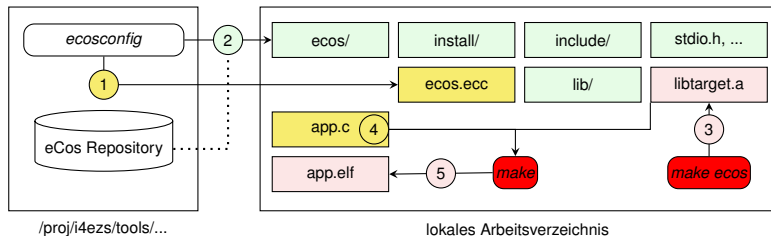
Application Programming Interface (API)

- Kernel API
 - vollständige C-Schnittstelle
 - siehe Dokumentation
- (Optionale) Posix Kompatibilitätsschicht
 - Scheduling Konfiguration, *pthread_**
 - Timer, Semaphore, Message Queues, Signale, ...



eCos Entwicklungszyklus

- 1 Erstellen einer Konfiguration (configtool/ecosconfig)
- 2 Kopieren ausgewählter Komponenten (configtool/ecosconfig)
- 3 Erstellen einer Betriebssystem**bibliothek**
- 4 Entwicklung der eigentlichen Anwendung
- 5 Kompilieren des Gesamtsystems

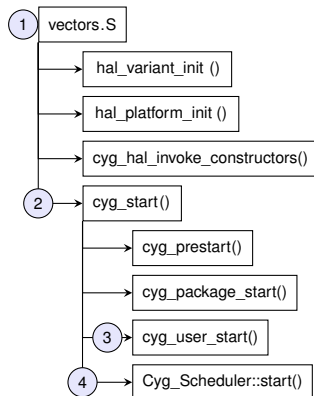


Wichtig!

Für jede Übung wird eine Konfiguration vorgegeben (Schritte 1–3)

eCos Systemstart

- 1 **vectors.S**
 - Hardwareinitialisierung
 - Globale Konstruktoren
- 2 **cyg_start()**
 - Hardwareunabhängige Vorbereitungen
- 3 **cyg_user_start()**
 - Einsprungpunkt für Anwendungscode!
 - Erzeugen von Threads
- 4 **Starten des Schedulers**



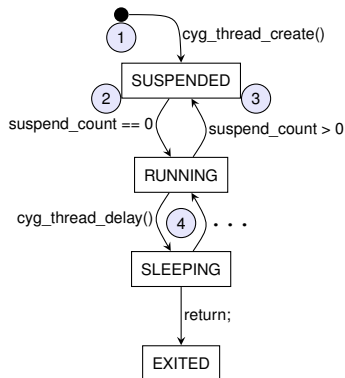
Wichtig!

In allen Übungsaufgaben muss man `cyg_user_start()` überschreiben und dort alle Threads anlegen. Die Funktion muss zurückkehren!

eCos-Threads

Threadzustände und Übergänge

- 1 Thread wird im Zustand *suspended* erzeugt.
 - `suspend_count = 1`
- 2 `cyg_thread_resume()` aktiviert
 - `suspend_count--`
- 3 `cyg_thread_suspend()` suspendiert
 - `suspend_count++`
- 4 delay, mutex, semaphore wait
- 5 Threadfunktion kehrt zurück



eCos Beispielanwendung

Threads erzeugen

```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

- Einbinden der nötigen Headerdatei
- Anlegen eines neuen eCos-Threads

Ausführliche Dokumentation

<http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html>

eCos Beispielanwendung

Threads erzeugen

```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

- Scheduling Informationen
- z. B. MLQ-Scheduler
 - Threadpriorität
 - Datentyp `cyg_uint8`

Ausführliche Dokumentation

<http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html>

eCos Beispielanwendung

Threads erzeugen

```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info ,
    cyg_thread_entry_t* entry ,
    cyg_addrword_t entry_data ,
    char* name ,
    void* stack_base ,
    cyg_ucount32 stack_size ,
    cyg_handle_t* handle ,
    cyg_thread* thread
);
```

- Thread Einsprungpunkt
- Funktionszeiger
- Signatur:
`void (*) (cyg_addrword_t)`

Ausführliche Dokumentation

<http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html>

eCos Beispielanwendung

Threads erzeugen

```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info ,
    cyg_thread_entry_t* entry ,
    cyg_addrword_t entry_data ,
    char* name,
    void* stack_base ,
    cyg_ccount32 stack_size ,
    cyg_handle_t* handle ,
    cyg_thread* thread
);
```

- Thread Parameter
- Beliebige Übergabeparameter
 - z. B. Zeiger auf threadlokale Daten

Ausführliche Dokumentation

<http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html>

eCos Beispielanwendung

Threads erzeugen

```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info ,
    cyg_thread_entry_t* entry ,
    cyg_addrword_t entry_data ,
    char* name,
    void* stack_base ,
    cyg_ucount32 stack_size ,
    cyg_handle_t* handle ,
    cyg_thread* thread
);
```

- Beliebiger Threadname
- Tipp: (gdb) info threads

Ausführliche Dokumentation

<http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html>

eCos Beispielanwendung

Threads erzeugen

```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

- Basisadresse des Threadstacks (→ `&stack[0]`)
- `cyg_uint8`-Array
- Global definieren
→ Datensegment!

Ausführliche Dokumentation

<http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html>

eCos Beispielanwendung

Threads erzeugen

```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info ,
    cyg_thread_entry_t* entry ,
    cyg_addrword_t entry_data ,
    char* name ,
    void* stack_base ,
    cyg_ccount32 stack_size ,
    cyg_handle_t* handle ,
    cyg_thread* thread
);
```

- Stackgröße in Bytes

Ausführliche Dokumentation

<http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html>

eCos Beispielanwendung

Threads erzeugen

```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

- Eindeutiger Identifikator

- zur “Steuerung” z. B.:

→

`cyg_thread_resume(handle)`

Ausführliche Dokumentation

<http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html>

eCos Beispielanwendung

Threads erzeugen

```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

- Speicher für interne Fadeninformationen
 - Fadenzustand u. a.
suspend_count
- Vermeidung von dynamischer Speicherallokation im Kernel

Ausführliche Dokumentation

<http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html>

eCos Beispielanwendung

Programmieren, Kompilieren, Debuggen

Wichtig!

Zu jeder Übungsaufgabe wird eine eCos Konfiguration bereitgestellt. Makefiles werden mit “*cmake*” generiert.

- 1 Vorgabe herunterladen, entpacken, Verzeichnis betreten
- 2 **Nötige Umgebungsvariablen setzen:** `source ecosenv.sh`
- 3 Eigene Quelldateien (nur) in `CMakeLists.txt` eintragen
- 4 `build` Verzeichnis betreten → out-of-source build¹
- 5 Makefiles erzeugen: `cmake ..` (*cmake PUNKT PUNKT!*)
- 6 Alles kompilieren: `make`
- 7 Im Simulator ausführen (debuggen):
`make sim` (`make run`, `make debug`)

¹www.cmake.org/Wiki/CMake_FAQ#Out-of-source_build_trees

eCos Beispielanwendung

Hallo Welt!

```
#include <cyg/hal/hal_arch.h>
#include <cyg/kernel/kapi.h>
#include <stdio.h>

#define MY_PRIORITY      11
#define STACKSIZE      (CYGNUM_HAL_STACK_SIZE_MINIMUM+4096)
static cyg_uint8 my_stack[STACKSIZE];
static cyg_handle_t my_handle;
static cyg_thread my_thread;

static void my_entry(cyg_addrword_t data) {
    int message = (int) data;
    printf("Beginning execution: thread data is %d\n", message);
    for (;;) {
        printf("Hello World!\n"); // \n flushes output (alternative fflush(0))
        cyg_thread_delay(1000); // Delay for 1000 * 1ms = 1 second
    }
}

void cyg_user_start(void) {
    printf("Entering cyg_user_start() function\n");
    cyg_thread_create(MY_PRIORITY, &my_entry, 0, "thread 1", my_stack, STACKSIZE,
                     &my_handle, &my_thread);
    cyg_thread_resume(my_handle);
}
```


Übungssystem - Qemu Emulation

- ❶ Qemu: $x86$ Emulator (make run, make {gdb,ddd,cgdb})
 - i386 Prozessor
 - Hauptspeicher
 - Serielle Schnittstelle (RS-232)
 - VGA-Bildschirm
 - ~ Keine Zeitdeterministische Ausführung ☹
 - ~ Dafür Debugging möglich ☺
- ❷ FailBochs $x86$ Emulator (Bochs mit Fehlerinjektionserweiterung)
 - Emuliert Hardware wie oben, zusätzlich
 - Genauer Zeitgeber
 - Digital-Analog-Umsetzer (DAC), Analog-Digital-Umsetzer (ADC)
 - ~ Zeitdeterministische Ausführung ☺
 - ~ Graphische Anzeige der Taskausführung ☺
 - ~ Kein Debugging (ausser `printf`-Debugging) ☹