DIY – Individual Prototyping and Systems Engineering

Embedded Entwicklung

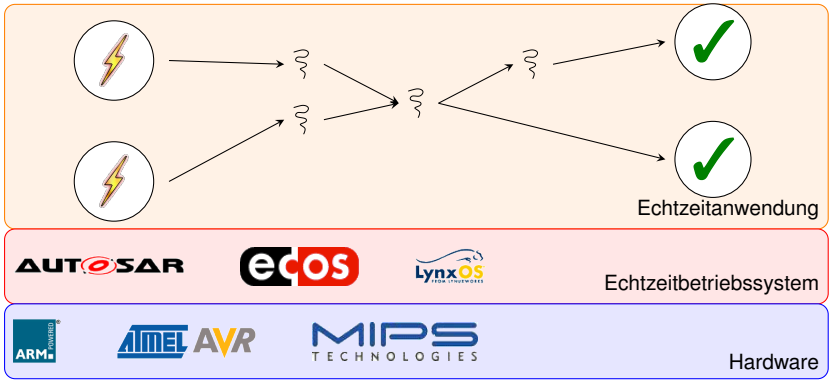
Peter Wägemann

Lehrstuhl für Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

https://www4.cs.fau.de

30. April 2018

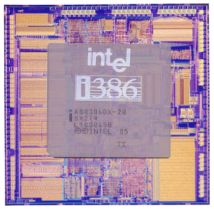
Wiederholung: Echtzeit-{Anwendung, Rechensystem}



Prozessorvielfalt in der Echtzeitwelt

freescale semiconductor

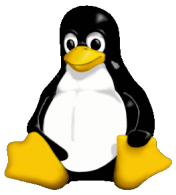
ARM Cortex
Intelligent Processors by ARM



Noch mehr Betriebssysteme

freeRTOS

µClinux



eCos

QNX

TinyOS

µC/OS-II™
The Real-Time Kernel

HIGH TEC

eCos

Embedded **C**onfigurable **O**perating **S**ystem

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`

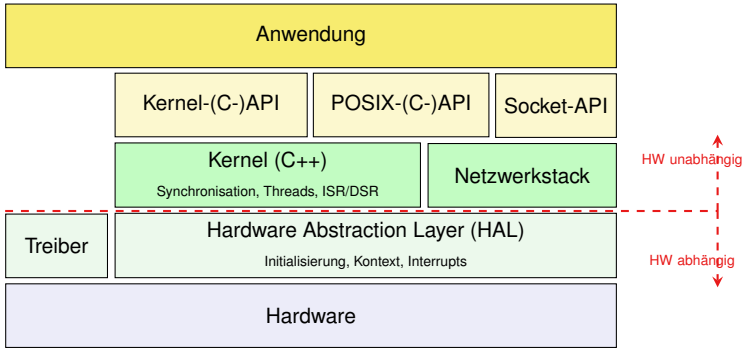
Unterstützte Plattformen

`http://www.ecoscentric.com/ecos/examples.shtml`

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

eCos-Systemarchitektur

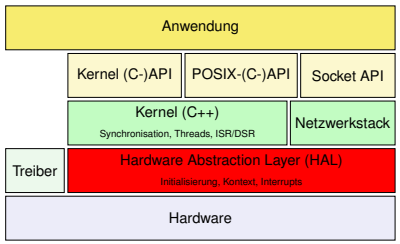
Überblick



eCos-Systemarchitektur

Hardware Abstraction Layer

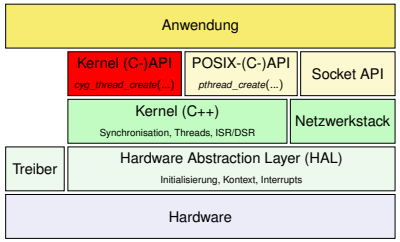
- Abstrahiert CPU- und plattformspezifische Eigenschaften
  - Kontextwechsel
  - Interruptverwaltung
  - CPU-Erkennung, Startup
  - Zeitgeber, I/O-Registerzugriffe



eCos-Systemarchitektur

Kernel

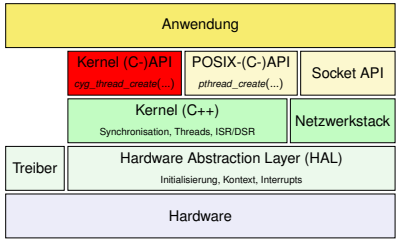
- Implementiert in C++
- Feingranular konfigurierbar
  - Verschiedene Schedulingstrategien (Bitmap/Multilevel Queue)
  - Zeitscheibenbasiert, präemptiv, prioritätenbasiert
- Verschiedene Synchronisationsstrategien
  - Mutexe, Semaphore, Bedingungsvariablen
  - Messages Boxes



eCos-Systemarchitektur

Kernel – Interruptbehandlung

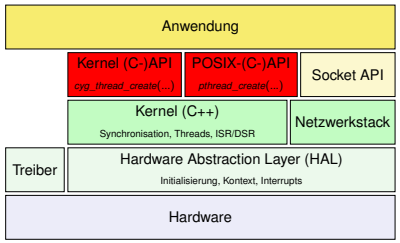
- Interrupt Service Routine (ISR)
  - Unverzügliche Ausführung
  - Asynchron
  - Kann DSR anfordern
- Deferred Service Routine (DSR)
  - Verzögerte Ausführung (beim Verlassen des Kernels)
  - Synchron



eCos-Systemarchitektur

Application Programming Interface (API)

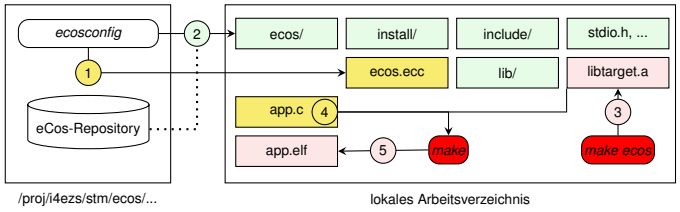
- Kernel API
  - vollständige C-Schnittstelle
  - siehe Dokumentation[1]
- (Optionale) POSIX-Kompatibilitätsschicht
  - Scheduling-Konfiguration, *pthread_*∗
  - Timer, Semaphore, Message Queues, Signale, ...



[1] `http://ecos.sourceware.org/docs-2.0/ref/ecos-ref.html`

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

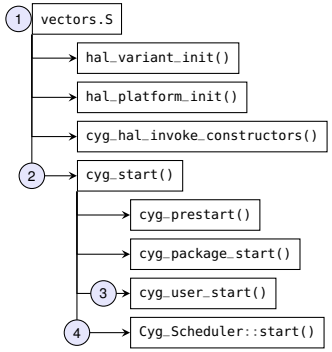


Wichtig!

Wir geben eine grundlegende Konfiguration vor!

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!

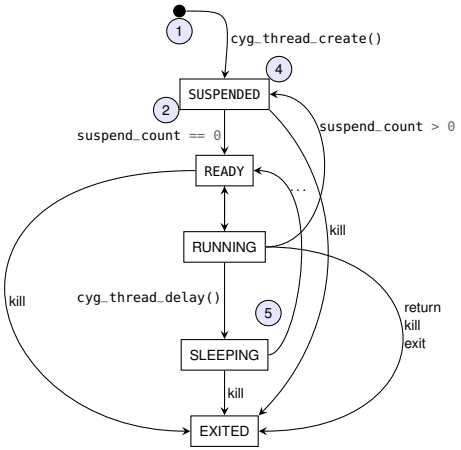
`cyg_user_start()` muss zurückkehren!

Gliederung

eCos-Threads

Threadzustände und Übergänge

1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`
2. `cyg_thread_resume()` aktiviert
   - `suspend_count--`
3. bereit <-> laufend
4. `cyg_thread_suspend()` suspendiert
   - `suspend_count++`
5. delay, mutex, semaphore wait
6. Threadterminierung



eCos-Threads

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-Threads

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-Threads

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`

pw DIY (SS 18) – Kapitel III Echtzeitbetriebssysteme 16/54
2 eCos-Threads

eCos-Threads

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

Ausführliche Dokumentation

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

pw DIY (SS 18) – Kapitel III Echtzeitbetriebssysteme 16/54
2 eCos-Threads

eCos-Threads

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`

pw DIY (SS 18) – Kapitel III Echtzeitbetriebssysteme 16/54
2 eCos-Threads

eCos-Threads

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!
- *Warum ist die notwendig?*

Ausführliche Dokumentation

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

eCos-Threads

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
);
```

- Stackgröße in Bytes

Ausführliche Dokumentation

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

pw DIY (SS 18) – Kapitel III Echtzeitbetriebssysteme 16/54
2 eCos-Threads

eCos-Threads

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`

pw DIY (SS 18) – Kapitel III Echtzeitbetriebssysteme 16/54
2 eCos-Threads

eCos-Threads

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 dynamischer Speicherallokation im Kernel

Ausführliche Dokumentation

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

pw DIY (SS 18) – Kapitel III Echtzeitbetriebssysteme 16/54
2 eCos-Threads

eCos-Threads

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
);
```

Ausführliche Dokumentation

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

eCos-Beispielanwendung

Hello World!

```
#include <cyg/hal/hal_arch.h>
#include <cyg/kernel/kapi.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;
  ezs_printf("Beginning execution: thread data is %d\n", message);
  for (;;) {
    ezs_printf("Hello World!\n"); // \n flushes output
  }
}

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

Umgang mit Zeit in eCos

- Aktuelle Aufgabe: Ausführung soll um feste Zeit *verzögert* werden
  ⇝ `cyg_thread_delay()` (bei uns `ezs_delay_us()`)
- Erwartet Parameter der Einheit *Clock-Ticks – Wieso?*
- Zeitmessung nur per Timer möglich ⇝ Timer-Zyklus kleinste Einheit

`cyg_clock_get_resolution(cyg_real_time_clock())` liefert Auflösung der Echtzeituhr:

```
typedef struct {
  cyg_uint32 dividend;
  cyg_uint32 divisor;
} cyg_resolution_t;
```

- $\frac{\texttt{dividend}}{\texttt{divisor}}$ ⇝ Zeit in ns, die ein Tick dauert (beispielsweise 1.000.000 ns)
- *Warum Aufteilung in Dividend & Divisor?*
- Umrechnung: z.B. `ms_to_cyg_ticks(uint32_t ms)`
- Vorsicht: Ganzzahl-Division, Wertebereichen, Fließkomma-Genauigkeit

Gliederung

Einschub: Schlüsselwort volatile

- Bei einem Interrupt wird `timer_event = 1` gesetzt
- Aktive Warteschleife wartet, bis `timer_event != 0`
- Flag (scheinbar) in Schleife nicht verändert ⇝ Compiler-Optimierung
  - `timer_event` wird einmalig vor der Warteschleife in Register geladen
    ☞ Endlosschleife
- `volatile` erzwingt das **Laden bei jedem Lesezugriff**

```
static uint8_t timer_event = 0;
ISR (INT0_vect) { timer_event = 1; }

void main(void) {
  while(1) {
    while(timer_event == 0) { /* warte auf Timer-Event */ }
    /* bearbeite Timer-Event */
volatile static uint8_t timer_event = 0;
ISR (INT0_vect) { timer_event = 1; }

void main(void) {
  while(1) {
    while(timer_event == 0) { /* warte auf Timer-Event */ }
    /* bearbeite Timer-Event */
```

Einschub: Lost-Update-Problematik

- Tastendruckzähler: Zählt mittels Variable `zaehler`
  - Inkrementierung in der Unterbrechungsbehandlung
  - Dekrementierung im Hauptprogramm zum Start der Verarbeitung

Hauptprogramm **H**

```
; volatile uint8_t zaehler;
; C-Anweisung: zaehler--;
lds r24, zaehler
dec r24
sts zaehler, r24
```

Interruptbehandlung **I**

```
; C-Anweisung: zaehler++
lds r25, zaehler
inc r25
sts zaehler, r25
```

Zeile	zaehler	r24	r25
-	5		
3 H	5	5	-
4 H	5	4	-
2 I	5	4	5
3 I	5	4	6
4 I	6	4	6
5 H	4	4	-

Wie behandle ich einen Interrupt?

Interrupt-Service-Routinen-Ausführung

- Unverzüglich, *asynchron*
  ⇝ auch innerhalb von Kernelfunktionen!
- Innerhalb ISR *keine Systemaufrufe* erlaubt!
  ⇒ Anmelden einer Deferrable Service Routine (DSR)

Deferrable-Service-Routinen-Ausführung

- *Synchron* zum Scheduler
- Falls Scheduler nicht verriegelt: *Unverzüglich* nach ISR
- sonst: Beim *Verlassen* des Kerns

Synonym: *Prolog-Epilog-Schema* bzw. *top/bottom half*

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#include <cyg/kernel/kapi.h>
void cyg_interrupt_create
(
  cyg_vector_t vector,
  cyg_priority_t priority,
  cyg_addrword_t data,
  cyg_ISR_t* isr,
  cyg_DSR_t* dsr,
  cyg_handle_t* handle,
  cyg_interrupt* intr
);
```

- Interruptvektornummer
- ⇝ Hardwarehandbuch

[2] `http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html`

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#include <cyg/kernel/kapi.h>
void cyg_interrupt_create
(
  cyg_vector_t vector,
  cyg_priority_t priority,
  cyg_addrword_t data,
  cyg_ISR_t* isr,
  cyg_DSR_t* dsr,
  cyg_handle_t* handle,
  cyg_interrupt* intr
);
```

- Interruptpriorität
- für unterbrechbare Unterbrechungen (hardwareabhängig)

[2] http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#include <cyg/kernel/kapi.h>
void cyg_interrupt_create
(
  cyg_vector_t vector,
  cyg_priority_t priority,
  cyg_addrword_t data,
  cyg_ISR_t* isr,
  cyg_DSR_t* dsr,
  cyg_handle_t* handle,
  cyg_interrupt* intr
);
```

- Beliebiger Übergabeparameter für ISR/DSR

Für eCos:
`build/ecos/install/include/cyg/hal/var_intr.h`

[2] http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#include <cyg/kernel/kapi.h>
void cyg_interrupt_create
(
  cyg_vector_t vector,
  cyg_priority_t priority,
  cyg_addrword_t data,
  cyg_ISR_t* isr,
  cyg_DSR_t* dsr,
  cyg_handle_t* handle,
  cyg_interrupt* intr
);
```

- Funktionszeiger auf *ISR-Implementierung*

Signatur:
`cyg_uint32 (*)(cyg_vector_t, cyg_addrword_t)`

[2] http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#include <cyg/kernel/kapi.h>
void cyg_interrupt_create
(
  cyg_vector_t vector,
  cyg_priority_t priority,
  cyg_addrword_t data,
  cyg_ISR_t* isr,
  cyg_DSR_t* dsr,
  cyg_handle_t* handle,
  cyg_interrupt* intr
);
```

- Funktionszeiger auf *DSR-Implementierung*

Signatur:
`cyg_uint32 (*)(cyg_vector_t, cyg_ucount32 count, cyg_addrword_t)`

[2] http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#include <cyg/kernel/kapi.h>
void cyg_interrupt_create
(
  cyg_vector_t vector,
  cyg_priority_t priority,
  cyg_addrword_t data,
  cyg_ISR_t* isr,
  cyg_DSR_t* dsr,
  cyg_handle_t* handle,
  cyg_interrupt* intr
);
```

- Handle und Speicher für *Interruptobjekt*

[2] http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

ISR-Implementierungsskelett

Beispiel einer minimalen ISR

```
cyg_uint32 isr(cyg_vector_t vector, cyg_addrword_t data) {
  cyg_bool_t dsr_required = 0;
  ...
  cyg_interrupt_acknowledge(vector);
  if (dsr_required) {
    return CYG_ISR_CALL_DSR;
  } else {
    return CYG_ISR_HANDLED;
  }
}
```

1. Beliebiger ISR-Code
2. Bestätigung der Interruptbehandlung
   Wozu ist das gut?
3. Anforderung einer DSR
   oder
4. Rückkehr ohne DSR

ISR-Implementierungsskelett

Beispiel einer minimalen ISR

```
cyg_uint32 isr(cyg_vector_t vector, cyg_addrword_t data) {
  cyg_bool_t dsr_required = 0;
  ...
  cyg_interrupt_acknowledge(vector);
  if (dsr_required) {
    return CYG_ISR_CALL_DSR;
  } else {
    return CYG_ISR_HANDLED;
  }
}
```

1. Beliebiger ISR-Code
2. Bestätigung der Interruptbehandlung
   Wozu ist das gut?
3. Anforderung einer DSR
   oder
4. Rückkehr ohne DSR

pw DIY (SS 18) – Kapitel III Echtzeitbetriebssysteme 24/54
3 Interruptbehandlung – 3.3 eCos-Unterbrechungsbehandlung

ISR-Implementierungsskelett

Beispiel einer minimalen ISR

```
cyg_uint32 isr(cyg_vector_t vector, cyg_addrword_t data) {
  cyg_bool_t dsr_required = 0;
  ...
  cyg_interrupt_acknowledge(vector);
  if (dsr_required) {
    return CYG_ISR_CALL_DSR;
  } else {
    return CYG_ISR_HANDLED;
  }
}
```

1. Beliebiger ISR-Code
2. Bestätigung der Interruptbehandlung
   Wozu ist das gut?
3. Anforderung einer DSR
   oder
4. Rückkehr ohne DSR

pw DIY (SS 18) – Kapitel III Echtzeitbetriebssysteme 24/54
3 Interruptbehandlung – 3.3 eCos-Unterbrechungsbehandlung

ISR-Implementierungsskelett

Beispiel einer minimalen ISR

```
cyg_uint32 isr(cyg_vector_t vector, cyg_addrword_t data) {
  cyg_bool_t dsr_required = 0;
  ...
  cyg_interrupt_acknowledge(vector);
  if (dsr_required) {
    return CYG_ISR_CALL_DSR;
  } else {
    return CYG_ISR_HANDLED;
  }
}
```

1. Beliebiger ISR-Code
2. Bestätigung der Interruptbehandlung
   Wozu ist das gut?
3. Anforderung einer DSR
   oder
4. Rückkehr ohne DSR

DSR-Implementierungskelett

Beispiel einer minimalen DSR

```
void dsr(
  cyg_vector_t vector,
  cyg_ucount32 count,
  cyg_addrword_t data)
{
  ...
}
```

1. Anzahl der ISRs, die diese DSR anforderten
   ⇝ normalerweise 1
2. Ausführung *synchron* zum Scheduler
   Was bedeutet das?

DSR-Implementierungskelett

Beispiel einer minimalen DSR

```
void dsr(
  cyg_vector_t vector,
  cyg_ucount32 count,
  cyg_addrword_t data)
{
  ...
}
```

1. Anzahl der ISRs, die diese DSR anforderten
   ⇝ normalerweise 1
2. Ausführung *synchron* zum Scheduler
   Was bedeutet das?

Gliederung

Alarme und periodische Aktivierung

eCos-Zähler-Abstraktion

eCos-*Alarme* basieren auf eCos-*Zählern* (Counter[3])

- Anlegen eines Zähler für bestimmtes Ereignis

```
void cyg_counter_create(cyg_handle_t* handle,
                        cyg_counter* counter)
```

- Inkrementieren:

```
void cyg_counter_tick(cyg_handle_t counter)
```

  - Zeitgeberunterbrechung (→ DSR)
    → eCos-interne Uhr als Zähler
  - externes Ereignis (Taster, etc.)
    → Zähler wird „von Hand“ inkrementiert (→ DSR, → Faden)

- eCos verwaltet Zählerstand intern
- Zugriff auf Zählerstand:

```
cyg_tick_count_t cyg_counter_current_value(cyg_handle_t ctr);
void cyg_counter_set_value(cyg_handle_t ctr, cyg_tick_count_t val);
```

[3] http://ecos.sourceware.org/docs-la/ref/kernel-counters.html

Alarme und periodische Aktivierung

eCos-Uhr

eCos-*Uhren* (Clocks[4]) sind spezialisierte *Zähler*

- Basierend auf *Zeitgeberunterbrechung*
- Festgelegte Zeitauflösung beim Erstellen
- Einzige vorgegebene Uhr: die eCos Uhr

```
cyg_handle_t cyg_real_time_clock(void);
```

  - Abfragen:

```
cyg_tick_count_t cyg_current_time(void)
```

  - Handle auf Uhr-internen Zähler holen

```
void cyg_clock_to_counter(cyg_handle_t clock,
                          cyg_handle_t* counter);
```

  - Inkrementieren:

```
void cyg_counter_tick(cyg_handle_t counter);
```

- Uhr anlegen: `cyg_clock_create()`:

→ Anwendung ist fürs Inkrementieren zuständig

[4] http://ecos.sourceware.org/docs-latest/ref/kernel-clocks.html

Alarme und periodische Aktivierung – 1

Anlegen eines Alarms

eCos-*Alarm*[5] führt Aktion bei Erreichen eines Zählerstandes aus

1 Anlegen:

```
void cyg_alarm_create(cyg_handle_t counter,
                      cyg_alarm_t* alarmfn,
                      cyg_addrword_t data,
                      cyg_handle_t* handle,
                      cyg_alarm* alarm);
```

- `counter` zugeordneter Zähler
- `alarmfn` Alarmbehandlung (Funktionspointer)
- `data` Parameter für Alarmbehandlung
- `handle` Alarm Handle (vgl. Threaderzeugung)
- `alarm` Speicher für Alarmobjekt (vgl. Threaderzeugung)

☞ alarmfn wird im DSR-Kontext ausgeführt
→ `cyg_thread_resume()`

[5] http://ecos.sourceware.org/docs-latest/ref/kernel-alarms.html

Alarme und periodische Aktivierung – 2

eCos-Alarm

eCos-*Alarm*[6] führt Aktion bei Erreichen eines Zählerstandes aus

2 Alarminitialisierung:

```
void cyg_alarm_initialize(cyg_handle_t alarm,
                          cyg_tick_count_t trigger,
                          cyg_tick_count_t interval);
```

- `alarm` Alarmhandle
- `trigger` *Absolute* Zählerticks der *ersten* Aktivierung
  - Nutze `cyg_current_time() + x` ⇝ *Phase*
  - Vorsicht: `cyg_current_time()` kann bei jedem Aufruf anderen Wert liefern
  - trigger muss in der Zukunft liegen. Warum?
- `interval` Zählerintervall für folgende *periodische* Aktivierungen

3 Alarm freischalten

```
void cyg_alarm_enable(cyg_handle_t alarm)
```

[6] http://ecos.sourceware.org/docs-latest/ref/kernel-alarms.html

Gliederung

eCos-Event-Flags[7]

Grundlagen

Signalisieren von Ereignissen

- Signale unterstützen *Produzent-Konsument Muster*
- Thread/DSR *signalisiert* Ereignis (z. B. Tastendruck)
  . . . konsumierender Thread *wartet*
- Umsetzung: 32-bit Integer $\rightsquigarrow$ 32 *Einzelsignale* pro Flag
  - Ein Flag erlaubt somit $2^{32} - 1$ Signalkombinationen
  - Threads können auf ein Signalmuster blockierend warten oder pollen

Achtung:

Flags zählen keine Ereignisse! (vgl. HW-Interrupts)

[7] http://ecos.sourceware.org/docs-latest/ref/kernel-flags.html

eCos-Event-Flags

API

- Produzenten/Konsumenten teilen sich eine Flag-Objekt
- Dieses wird von der *Anwendung* bereitgestellt (vgl. Alarmobjekt)
- Flag-Objekt muss initialisiert werden:

```
cyg_flag_init(cyg_flag_t* flag)
```

- Signal(e) im Flag setzen:

```
cyg_flag_setbits(cyg_flag_t* flag, cyg_flag_value_t value)
```

- Bzw. zurücksetzen:

```
cyg_flag_maskbits(cyg_flag_t* flag, cyg_flag_value_t value)
```

- Auf Signal warten/pollen:

```
cyg_flag_value_t cyg_flag_wait/poll(cyg_flag_t* flag,
                                    cyg_flag_value_t pattern,
                                    cyg_flag_mode_t mode);
```

eCos-Event-Flags

API – Konsumieren von Signalen

- `cyg_flag_value_t pattern` setzt gewünschte Signalkombination
- `cyg_flag_mode_t` legt Weckmuster fest
  - `CYG_FLAG_WAITMODE_AND`: Alle konfigurierten Signale müssen aktiv sein; Sie bleiben nach dem Aufwachen gesetzt.
  - `CYG_FLAG_WAITMODE_OR`: Mindestens eines der konfigurierten Signale muss aktiv sein; Alle Signale bleiben nach dem Aufwachen gesetzt.
  - `CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR`: Mindestens eines der konfigurierten Signale muss aktiv sein; Alle gesetzten Signale werden nach dem Aufwachen gelöscht.

eCos-Event-Flags

Beispiel

```
static cyg_flag_t flag0;

void my_dsr(cyg_vector_t v,
            cyg_ucount32 c,
            cyg_addrword_t d){
  cyg_flag_setbits(&flag0, 0x02);
}

void user_thread(cyg_addr_t data){
  while(true) {
    cyg_flag_wait(&flag0, 0x22,
                  CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR);
    printf("Event!\n");
  }
}

void cyg_user_start(void){
  ...
  cyg_flag_init(&flag0);
  ...
}
```


eCos-Mailboxes[8]

Grundlagen

- Zwischen Threads können *Nachrichten* versendet werden
- Konsument erzeugt einen Briefkasten (mailbox) fester Größe
- Produzenten legt Nachrichten dort ab
  - Inhalt: Zeiger auf beliebige Datenstruktur
  - Konsument kann auf *Nachrichtenempfang* blockieren
  - Produzent blockiert, falls Briefkasten *voll*
  - Aber auch *nicht-blockierende* Aufrufvarianten

[8] http://ecos.sourceware.org/docs-latest/ref/kernel-mail-boxes.html

Versenden von Nachrichten - eCos Mail Boxes[9]

API - Übersicht

- Mailbox anlegen:

```
cyg_mbox_create(cyg_handle_t* handle, cyg_mbox* mbox);
```

- Nachricht verschicken:

```
cyg_bool_t cyg_mbox_put(cyg_handle_t mbox, void* item);
```

- Nachricht empfangen:

```
void* cyg_mbox_get(cyg_handle_t mbox);
```

- Empfang *und* Versand können blockieren.
- `*try*`-Versionen: Würde ich blockieren?
- `*timed*`-Versionen: Blockieren, aber nur für bestimmte Zeit.

→ Selbststudium!

[9] http://ecos.sourceware.org/docs-latest/ref/kernel-mail-boxes.html

Versenden von Nachrichten – Beispiel

- Initialisierung:

```
static cyg_handle_t mailbox_handle;
static cyg_mbox     mailbox;
void cyg_user_start(void) {
  cyg_mbox_create(&mailbox_handle, &mailbox);
  ...
}
```

- Produzent (Sender):

```
void producer_entry(cyg_addrword_t data) {
  ...
  cyg_mbox_put(mailbox_handle, &my_message);
  ...
}
```

- Konsument (Empfänger):

```
void consumer_entry(cyg_addrword_t data) {
  ...
  void *message = cyg_mbox_get(mailbox_handle);
  ...
}
```

Gliederung

Gegenseitiger Ausschluss – eCos-NPCS[10]

Nicht-preemptiver kritischer Abschnitt durch Sperren des Schedulers

Kerneldatenstrukturen durch Sperren des Schedulers geschützt
⇝ **Big Kernel Lock** (BKL)

- Sperre: `void cyg_scheduler_lock(void);`
  - Sofortiges Anhalten des Schedulings
  - Verzögerung der DSR-Ausführungen
  - ISRs werden weiterhin zugestellt!
- Freigabe: `void cyg_scheduler_unlock(void);`
  - Sofortige Abarbeitung angelaufener DSRs
- Alle Systemaufrufe werden per NPCS synchronisiert
- Anwendungen sollten Mutexe, Semaphore, etc. nutzen
  - Ausnahme: Synchronisation zwischen DSR und Thread

Was sind die Vor- bzw. Nachteile des BKL Konzepts?

[10] http://ecos.sourceware.org/docs-latest/ref/kernel-schedcontrol.html

Gegenseitiger Ausschluss – eCos-Mutex[11]

Initialisierung

- Initialisierung

```
void cyg_mutex_init(cyg_mutex_t* mutex);
```

- Protokoll auswählen:

```
void cyg_mutex_set_protocol(cyg_mutex_t* mutex,
                            enum cyg_mutex_protocol protocol);
```

  - `CYG_MUTEX_NONE` keine Prioritätsvererbung
  - `CYG_MUTEX_INHERIT` erbe Priorität des aktuellen Inhabers
  - `CYG_MUTEX_CEILING` erbe Prioritätsobergrenze
- nur bei `CYG_MUTEX_CEILING`: Prioritätsobergrenze setzen

```
void cyg_mutex_set_ceiling(cyg_mutex_t* mutex,
                           cyg_priority_t priority);
```

- *Prioritätsobergrenze +1 höherprior als Thread*

[11] http://ecos.sourceware.org/docs-latest/ref/kernel-mutexes.html

Gegenseitiger Ausschluss – eCos-Mutex

Verwendung

- Mutex belegen

```
cyg_bool_t cyg_mutex_lock(cyg_mutex_t* mutex);
```

Rückgabewert
  - `true` falls Belegen erfolgreich
  - `false` sonst

- Mutex freigeben:

```
void cyg_mutex_unlock(cyg_mutex_t* mutex);
```

Gegenseitiger Ausschluss – eCos-Mutex

Beispiel

```
static cyg_mutex_t s_mutex;

void cyg_user_start(void) {
  // Mutex initialisieren
  cyg_mutex_init(&s_mutex);

  // Protokoll auswaehlen
  cyg_mutex_set_protocol(&s_mutex, CYG_MUTEX_CEILING);

  // Prioritaetsobergrenze festlegen
  cyg_mutex_set_ceiling(&s_mutex, 3);

  // Tasks, Alarme etc.
}

void task_entry(cyg_addrword_t data) {
  cyg_mutex_lock(&s_mutex); // auf Freigabe warten
  // kritischer Abschnitt
  cyg_mutex_unlock(&s_mutex); // Mutex freigeben
}
```

Gliederung

Dokumentation zum EZS-Board

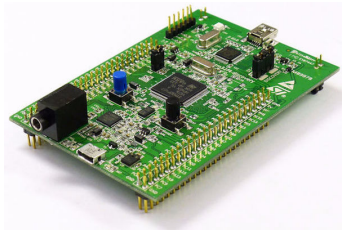
EZS-Wiki

`https://gitlab.cs.fau.de/ezs/ezs-board/wikis/home`

- Umstellung auf neues Entwicklungsboard
  - ⇝ Mögliche Probleme
- ☞ Dokumentation im Wiki
- Erweiterung durch *alle Teilnehmer*
  - gitlab account notwendig
- Besonders wertvolle Beiträge
  - ☞ Gutscheine für I4-Kaffeekarte
  - VM für weitere Plattformen (Windows, OSX...)

Entwicklungsplattform STM32F411

- ARM Cortex-M4 Prozessor
  - Flash-Speicher: 512 KB
  - RAM: 128 KB
- reichhaltige Peripherie
  - Serielle Kommunikation
  - Timer
  - GPIOs
  - ADCs
  - 3-Achsen Gyroskop
  - Beschleunigungssensor
  - Audio Sensor
  - *integrierte Debugging-Schnittstelle*
  - ...



eCos-Vorgabeandwendung

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 build[12]
5. Makefiles erzeugen: `cmake ..` (*cmake PUNKT PUNKT*)
6. Alles kompilieren: `make`
7. Flashen, ausführen, debuggen: `make flash`
   ⇝ Flasher & Debugger: `make gdb` (später ausführlicher)

[12] `www.cmake.org/Wiki/CMake_FAQ#Out-of-source_build_trees`

Flashen & Debuggen: Blackmagic Firmware

- Mini-USB-Kabel anschließen
- Nach Verbinden des USB-Kabels zwei serielle Ports
  - `/dev/ttyACM0`: Debugger
  - `/dev/ttyACM1`: serielle Kommunikation (Ausgabe z.B. mit `cutecom`)
- Ausleihbare Boards sind mit Blackmagic Firmware[13] bereits ausgestattet

[13] `https://github.com/blacksphere/blackmagic`

Flashen mittels Kommandozeile

- Bashprompt: , gdb-Prompt: >

```
cd <ezs-aufgabe1>
source ecosenv.sh
cd build
cmake ..
make ##Erstellen der Software
make flash ##Aufspielen der Software
oder
make debug ##Aufspielen und Debuggen mittels gdb
```


Gliederung

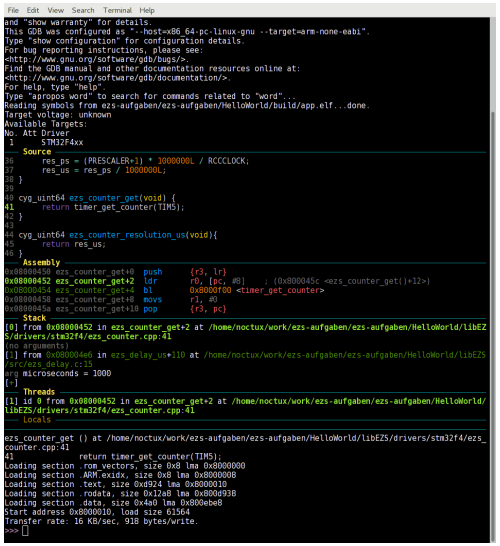
gdb-Dashboard

Aufrufen

- Interaktive gdb-Session:
  `% make gdb`
- gdb-Dashboard:
  `% make debug`
- Manueller Aufruf:
  `% arm-none-eabi-gdb \ -x ezs_dashboard.gdb app.elf`
- Parameter `-nh` verwenden falls `.gdbinit` vorhanden

Fenster

1. Source Code
2. Assembly
3. Stack
4. Threads
5. Lokale Varibalen

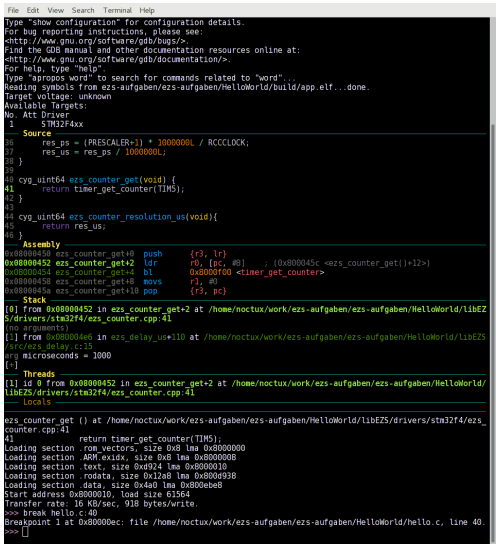


gdb Kommandos – I

Befehle haben Langformen (`break`) und Kurzformen (`b`)

Wichtige Befehle

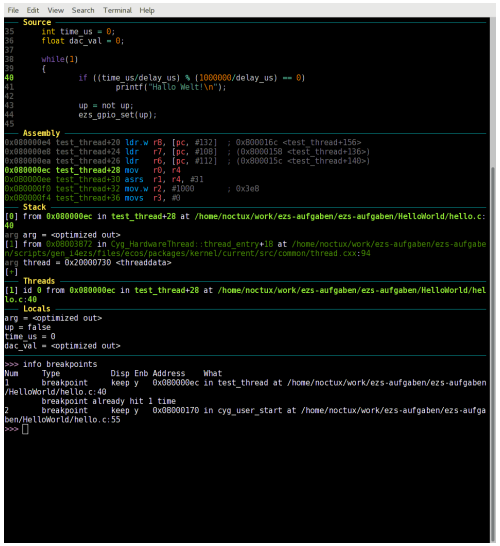
- Breakpoint setzten:
  `»» b(reak) cyg_user_start`
- Einzelschritt (Funktionen betreten):
  `»» s(tep)`
- Einzelschritt (Funktionen nicht betreten):
  `»» n(ext)`
- Programm fortsetzten:
  `»» c(ontinue)`
- Bis zum Ende der Funktion ausführen:
  `»» fin(ish)`
- Funktion anzeigen:
  `»» l(ist) <funktionsname>`
- gdb schließen:
  `»» q(uit) (oder Strg+D)`
- Neu Flashen:
  `»» l(oad)`



gdb Kommandos – II

Wichtige Befehle (Fortsetzung)

- Backtrace (Aufruf-Stack) anzeigen:
  `»» b(ack)t(race)`
- Dashboard neu zeichnen:
  `»» dashboard`
- Breakpoints anzeigen:
  `»» info breakpoints`
- Breakpoint löschen:
  `»» delete <nummer>`
- Variable anzeigen:
  `»» p(rint) <variablenname>`



Weiterführende Informationen

- gdb-Dashboard benötigt einen gdb mit python-Bindings
- gdb „abschießen“:
  `killall arm-none-eabi-gdb -s SIGKILL`
- EZS-Board in initialen Zustand setzen:
  USB-Kabel abstecken & wieder anstecken
- Serielle Terminals
  - cutecom (graphisch)
  - minicom (Terminal)
- `minicom -D /dev/ttyACM1` (Strg-A Z für Hilfe/weitere Befehle wie Exit)
- In Dashboard auch `make` & `make flash` verfügbar, anschließend Programm mit `c(ontinue)` weiter ausführen