

Simple Scope

eCos-Vertiefung

Florian Franzmann Tobias Klaus Peter Wägemann

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
<http://www4.cs.fau.de>

2. Dezember 2015



- Oszilloskope sind *Leihgabe* der E-Technik
- Stehen nur noch bis Ende des Jahres zur Verfügung
- Werden für Aufgabe 1, 2 und 3 benötigt

⇒ Abgabe Aufgabe 1, 2 und 3 ...

...nur noch bis Ende des Jahres möglich!



- 1 eCos-Vertiefung
 - Alarme
 - Time-Triggered eCos

- 2 Hinweise zu Aufgabe 4
 - Graphische Ausgabe
 - Software-Tracing



eCos-*Alarmer* basieren auf eCos-*Zählern* (Counter¹)

- Anwendung erzeugt Zähler für bestimmtes Ereignis

- Zeitgeberunterbrechung (→ DSR)
- externes Ereignis (Taster, etc.)

→ Zähler wird „von Hand“ inkrementiert

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

- Alternativ: eCos-interne Uhr als Zähler (→ Aufgabe 2)
- eCos verwaltet Zählerstand *intern*
- Aktiviert ggf. *Alarm* bei Erreichen eine Zählerstandes

¹ecos.sourceware.org/docs-latest/ref/kernel-counters.html



eCos-*Alarm*² 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)

²ecos.sourceforge.org/docs-latest/ref/kernel-alarms.html



eCos-*Alarm*³ 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 bis zur *ersten* Aktivierung
 - Nutze `cyg_current_time()` + x
 $\leadsto$ *Phase*
 - **Vorsicht:** `cyg_current_time()` liefert bei jedem Aufruf anderen Wert
- interval Zählerintervall für folgende *periodische* Aktivierungen

3. Alarm freischalten

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

³ecos.sourceware.org/docs-latest/ref/kernel-alarms.html



eCos-*Uhren* (Clocks⁴) sind spezialisierte *Zähler*

- Basierend auf *Zeitgeberunterbrechung*
- Festgelegte Zeitaufösung beim Erstellen

```
void cyg_clock_create(cyg_resolution_t resolution,  
    cyg_handle_t* handle, cyg_clock* clock);
```

- Uhrenzähler wird „von Hand“ inkrementiert

1. Handle auf Uhr-internen Zähler holen

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

2. Inkrementieren:

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

- *Alternativ:* Handle auf Scheduler Uhr (→ Aufgabe 2)

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

⁴ecos.sourceforge.org/docs/latest/ref/kernel-clocks.html

eCos ist eigentlich *ereignisgesteuert*

~> Studienarbeit: Time-Triggered eCos:

- Zeitgesteuerte Ausführung von Tasks in *Ablauftabellen*.
- Terminüberwachung mit *Ausnahmebehandlung*
- Angelehnt an *OSEKtime* (Automobilstandard)

Ausführliche Dokumentation

→ Ausarbeitung der Studienarbeit von Michael Lang:

https://opus4.kobv.de/opus4-fau/files/674/sa_michael_lang.pdf



Ablauf Tabellen und Tasks werden statisch (global) angelegt:

1. Definition der Ablauf Tabellen unter Angabe der maximalen Ereignisseinträge. (Makro!)

```
tt_DispatcherTable(string <Tabellenname>,  
                  tt_uint32 <Eintragsanzahl>)
```

2. Definition der Task(s) und Implementierung des Task-Programms

```
tt_Task ( string <Task-Name>) { .. Programm .. }
```

3. Definition des Idletasks und optionaler Hook-Routinen.

```
tt_IdleTask{.. Programm .. }
```



Initialisierung zur Laufzeit (in `cyg_user_start()`):

1. Initialisierung der Tasks unter Angabe ihrer Terminüberwachungsmethode.

```
tt_InitTask (tt_tasktype <Task-Name>,  
             tt_deadlinemethod <Terminmethode>);
```

2. Initialisierung der Ablauftabelle(n).

```
tt_InitDispatcherTable( tt_dispatchertabletype <Tabellenname>)
```

3. Definition der Ereignisse der einzelnen Tabellen.

```
tt_bool tt_DispatcherTableEntry(  
    tt_dispatchertabletype <Tabellenname>,  
    tt_ticktype <Zeitpunkt>,  
    tt_action <Ereignis>,  
    tt_tasktype <Task-ID> )
```

4. Starten des Betriebssystems.

```
void tt_startos( tt_dispatchertabletype <Anfangstabelle> )
```



- *Ereignisorientiertes* Ausführungsmodell ($\neq$ ereignisgesteuert!)
 - *keine* Endlosschleife in der Anwendung
 - ↪ Betriebssystem startet Faden, der Jobs abarbeitet und sich beendet
- Einlastung erfolgt *verdrängend*
 - Neue Aufgabe unterbricht Ausführung laufender Aufgabe
 - Anschliessend Fortsetzung der unterbrochenen Aufgabe
 - ↪ Terminüberprüfung möglich
- *Aber*: Faden blockiert sich nie selbst
 - sonst würde kein Fortschritt mehr stattfinden
 - ↪ *run-to-completion-Semantik*

Vergleich mit eCos: *Prozessorientiertes* Ausführungsmodell

- Anwendungsthread implementiert Endlosschleife ...
- ...die sich blockiert und auf Ereignis wartet



Für jeden Thread mittels `tt_deadlinemethod` konfigurierbar:

- `TT_STRINGENT` Strikte Terminüberprüfung *direkt* nach Ablauf des Termins
- `TT_NONSTRINGENT` Nicht-Strikte Terminüberprüfung zu einem späteren Zeitpunkt (Terminverletzung möglich)

Einplanung von Taskstart oder Terminüberwachung (`tt_action`):

- `TT_START_TASK` Task-Einplanung
- `TT_DEADLINE` Terminüberprüfung

```
tt_bool tt_DispatcherTableEntry(  
    tt_dispatchertabletype <Tabellename>,  
    tt_ticktype <Zeitpunkt>,  
    tt_action <Ereignis>,  
    tt_tasktype <Task-ID>)
```



- 1 eCos-Vertiefung
 - Alarme
 - Time-Triggered eCos

- 2 Hinweise zu Aufgabe 4
 - Graphische Ausgabe
 - Software-Tracing



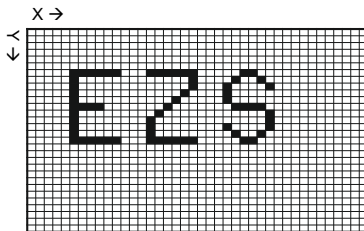
- Ab Aufgabe 4 verwenden wir einen i386-Emulator

↪ FailBochs

- Was ändert sich dadurch für euch?

- Zeitgeber *deutlich geringerer Auflösung*
- `make flash` funktioniert nicht mehr
↪ stattdessen `make sim`, `make run` und `make ddd`
- `gdb` bzw. `ddd` anstatt `trace32` ☹

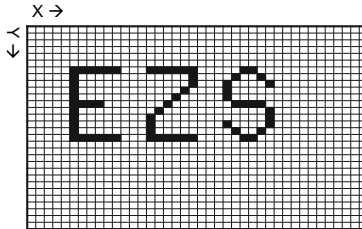




- **Framebuffer** $\leadsto$ Grafische Ausgabe
- eCos bietet einheitliche, *hardwareunabhängig Schnittstelle*⁵
- Auflösung in unserem Fall: **320x240x16 bit** (gemu & Hardware)
 - Zwecks Portabilität immer Variablen nutzen:
 - Breite: `CYG_FB_WIDTH(FRAMEBUF)`
 - Höhe: `CYG_FB_HEIGHT(FRAMEBUF)`

⁵ecos.sourceware.org/docs-latest/ref/io-framebuf.html





Nützliche Funktionen:

```
1 void ezs_fb_init(void);
2
3 void ezs_fb_clear(cyg_fb_colour color);
4
5 void ezs_fb_fill_block(cyg_ucount16 x, cyg_ucount16 y,
6     cyg_ucount16 width, cyg_ucount16 height, cyg_fb_colour color);
7
8 void ezs_fb_print_string_cur(char* c, cyg_ucount16 x,
9     cyg_ucount16 y, cyg_fb_colour color);
```



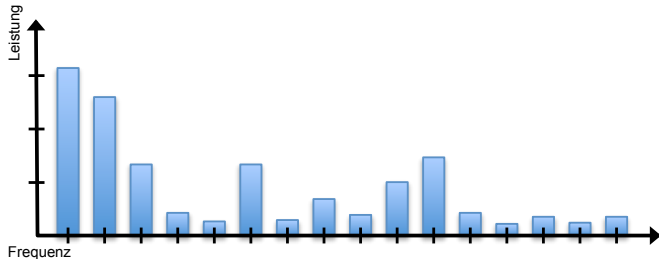
```
void ezs_power_density_spectrum( float in[], float out[], int N)
```

- in[] *Eingabe*, Abschnitt des Zeitbereichssignals
- out[] *Ausgabe*, Leistungsdichtespektrum (LDS)
- N *Anzahl der Abtastwerte*, wobei N Zweierpotenz
- Zeitbereichssignal der Länge N $\mapsto$ LDS⁶ der Länge $\frac{N}{2}$
- Höchste Frequenz im Spektrum aus Abtasttheorem

$$\Rightarrow f_{\max} = \frac{f_{\text{Abtast}}}{2} \quad (1)$$

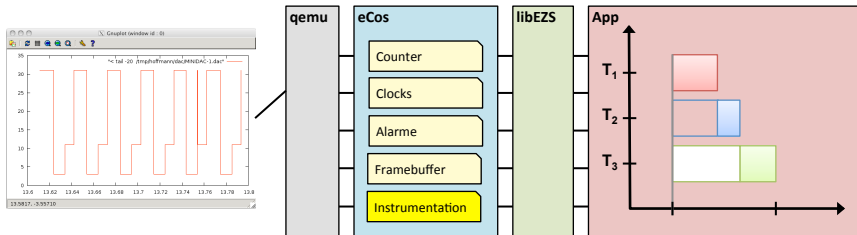
⁶http://en.wikipedia.org/wiki/Spectral_density





- Darstellung mit 16 Werten:
⇒ LDS der Länge 32 notwendig
 - 1Hz Abstand $\leadsto$ Spektrum bis 16 Hz erfassen
⇒ Abtastfrequenz 32 Hz
- ⇒ Balken repräsentieren **Leistung**





- **Visualisierung der Threads** $\leadsto$ Softwarebasiertes Tracing
- eCos Instrumentations⁷
- Ausgabe der **Priorität** $\leadsto$ *Ablaufgraph*

⁷ecos.sourceware.org/docs/latest/user-guide/kernel-instrumentation.html



⁷<https://commons.wikimedia.org/wiki/User:Pensiero~commonswiki>

