

Echtzeitsysteme

Übungen zur Vorlesung

Simple Scope

Tobias Klaus Florian Schmaus Peter Wägemann

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

21. November 2016



14.11	15.11	16.11	17.11	18.11
Übung 4		Übung 4		Vorlesung 5
Ausgabe A3				Abgabe A2
21.11	22.11	23.11	24.11	25.11
Übung 5		Übung 5		Vorlesung 6
Ausgabe A4				
28.11	29.11	30.11	01.12	02.12
Vorlesungsfrei				
05.12	06.12	07.12	08.12	09.12
Übung 6		Übung 6		Vorlesung 7
Ausgabe A5				Abgabe A3
12.12	13.12	14.12	15.12	16.12
Übung 7		Übung 7		Vorlesung 8
				Abgabe A4
19.12	20.12	21.12	22.12	23.12
Übung 8	Abgabe A5	Übung 8		Vorlesungsfrei
26.12	27.12	28.12	29.12	30.12
Weihnachten/Neujahr				

■ Betrieb bis Weihnachten

- Vorlesungsfrei: KW48
- Letzte Übung: 21.12.
- Überlappende Ausgabe der Aufgaben

■ Bewertung der Übung

- Erinnerung:
Mitarbeit bringt Pluspunkte
- Prüfungsvoraussetzung: Erfolgreiche Bearbeitung aller Übungsaufgaben
- Anmeldung zur Abgabe ist **fester** Termin
- Verspätete / keine Abgabe → **Schwergewichtige** Minuspunkte



Nutzt die Zeit!



1 Wiederholung

- Ereignisorientierter Planer
- Berechnungskomplexität
- Ereignisgesteuerte & Zeitgesteuerte Echtzeitsysteme

2 eCos-Vertiefung

- Alarme
- Time-Triggered eCos

3 Hinweise zu Aufgabe 4

- Graphische Ausgabe
- Software-Tracing



1

Wiederholung

- Ereignisorientierter Planer
- Berechnungskomplexität
- Ereignisgesteuerte & Zeitgesteuerte Echtzeitsysteme

2

eCos-Vertiefung

- Alarme
- Time-Triggered eCos

3

Hinweise zu Aufgabe 4

- Graphische Ausgabe
- Software-Tracing





Einplanung von Arbeitsaufträgen erfolgt zu **Ereigniszeitpunkten**

- Ihr Auftreten ist nicht (exakt) vorhersehbar
- Ereignisauslöser sind kontrollierte Objekte/andere Arbeitsaufträge
- Die Ereignisverarbeitung unterliegt einer gewissen **Dringlichkeit**



Ereignisse haben Prioritäten die dem Ereignisauslöser und/oder der Ereignisverarbeitung zugeordnet sind

Feste Zuordnung → Ereignisverarbeitung/-auslöser

- Arbeitsaufträge erhalten **absolute Priorität**

Variable Zuordnung → Ereignisverarbeitung

- Arbeitsaufträge erhalten **relative Priorität**

Auch **prioritätsorientierte Einplanung** (engl. *priority-driven scheduling*)



- Einplanung ereignisbedingt ausgelöster Arbeitsaufträge resultiert in einer **dynamischen Datenstruktur** → sortierte Liste



Kritisch ist die **Berechnungskomplexität** und wann sie anfällt

- Gekoppelt mit der Einlastung: **online scheduling** (siehe III-2/?? ff)
 - Konstant oder variabel, dann jedoch mit oberer Schranke → WCET
- Zum **Auslöse-** oder **Auswahlzeitpunkt** von Arbeitsaufträgen



Priorität bildet den **Sortierschlüssel** (engl. *sort key*)

- Ergibt sich ggf. erst zum Ereigniszeitpunkt aus der Priorität der von ihm zu verarbeitenden **Ereignissen**
- Ist eindeutig abzubilden auf einen endlichen Wertebereich

 Auch **prioritätsorientierter Planer** (engl. *priority-driven scheduler*)



- **Ablaufliste** $\mapsto$ **Dynamische** Datenstruktur
 - Prioritäten entsprechen der Position innerhalb der Ablaufliste
 - Das (relative) Prioritätsgefüge passt sich zur Laufzeit an
 - $\rightarrow$ Eignung für die Implementierung **dynamischer Prioritäten**
 - Linearer Berechnungsaufwand zum Auslösezeitpunkt
 - Vorabwissen zur **WCET des Sortiervorgangs** ist gefordert
 - Nahezu konstanter Berechnungsaufwand zum Auswahlzeitpunkt
 - Aufträge vom Kopf her der (ggf. einfach verketteten) Liste entnehmen
- **Ablaufabelle** $\mapsto$ **Statische** Datenstruktur
 - Prioritäten werden fest auf Tabellenindizes abgebildet
 - Zur Laufzeit unveränderliches Gefüge absoluter Prioritäten
 - $\rightarrow$ Eignung für die Implementierung **fester Prioritäten**
 - Konstanter Berechnungsaufwand zum Auslösezeitpunkt
 - Aufträge durch indizierte Adressierung in die Tabelle aufnehmen
 - Ggf. ist ein Tabelleneintrag eine Auftragsliste (FIFO) gleicher Priorität
 - Linearer Berechnungsaufwand zum Auswahlzeitpunkt
 - Vorabwissen zur **WCET des Suchvorgangs** ist gefordert
 - Tabelleneinträge können leer sein und sind zu überspringen



Ablaufliste

```
Job *list = 0;

void release(Job *item) {
    Job* last = 0, tail = list;

    while(tail && outrank(tail,item)) {
        last = tail;
        tail = last->next;
    }

    if(!last) {
        item->next = list; list = item;
    } else {
        item->next = tail;
        last->next = item;
    }
}

Job* extract() {
    Job* item = list;
    if(item) list = item->next;
    return item;
}
```

release $O(n)$

extract $O(1)$

Ablauftabelle

```
Job* table[Jobs];

void release(Job *item) {
    assert((priority(item) >= 0)
        && (priority(item) <= Jobs - 1));
    item->state = Ready;
}

Job* extract() {
    for(uint slot = 0; slot < Jobs; slot++)
        if(table[slot]->state == Ready) {
            table[slot]->state = Selected;
            return table[slot];
        }

    return 0;
}
```

⚠️ Fest Anzahl an Aufträgen

release $O(1)$

extract $O(n)$

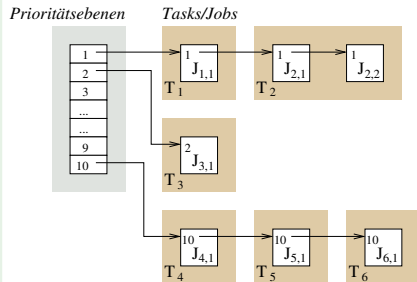


- Eine Ablaufliste je Priorität, organisiert als **FIFO**
- Ablauflisten werden in einer Ablaufabelle verwaltet

Multi-Level-Queue

```
Job* table[Jobs];  
  
void release(Job *item) {  
    assert((prio(item) >= 0)  
        && (prio(item) <= Jobs - 1));  
    item->state = Ready;  
    append(table[prio(item)], item);  
}  
  
Job* extract() {  
    for(uint slot = 0; slot < priors; slot++)  
        if(!empty(table[slot])) {  
            Job *item = head(table[slot]);  
            item->state = Selected;  
            return item;  
        }  
    return 0;  
}
```

- Mehrere Tasks pro Priorität
- Mehrere Aufträge pro Task
- Reihenfolge der Auslösung





Auftragsauslösung mit **konstantem Aufwand** $O(1)$ erfordert:

- 1 Ablaufplan ist dynamische Datenstruktur (Tabelle) aus mehrere Prioritätsebenen
 - Wartelisten $\mapsto$ LIFO
 - Warteschlangen $\leadsto$ FIFO
- 2 Aufträge die über denselben Tabelleneintrag erfasst werden besitzen dieselbe Priorität $\leadsto$ **Prioritätsschlange**
 - Sonst könnte LIFO/FIFO **Prioritätsverletzung** zur Folge haben
- 3 Anzahl der Tabelleneinträge entspricht mindestens der Anzahl statisch zugewiesener Prioritäten
 - Ggf. werden dann nahezu alle Tabelleneinträge nur einen Auftrag erfassen
 - Abhängig von der Echtzeitanwendung und dem Einplanungsverfahren

Auftragsauswahl ist unter diesen Bedingungen nicht in $O(1)$ möglich:

- Leere Tabelleneinträge sind ggf. zu überspringen





Vorrangsteuerung ist mit grundsätzlichen Konflikt konfrontiert:

- Entweder Auftragsauslösung oder Auftragsauswahl mit $O(1)$ zu versehen
 - Beides zugleich geht nicht



Für Auftragsauslösung in $O(1)$ spricht:

- Ereignisgesteuerte Auslösung benötigen konstante Zeit
 - Z.B. als Folge eines *Interrupts* oder der Zustellung eines *Zeitsignals*
 - Bedeutsam für voll-verdrängbare Systeme

→ Ereignisbedingte Auftragverzögerungen lassen sich exakt bestimmen

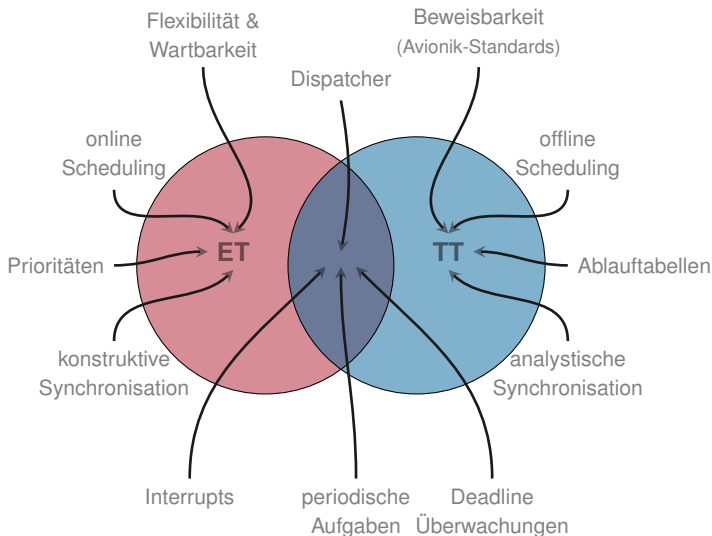


Für Auftragsauswahl in $O(1)$ spricht:

- Übergang zum nachfolgenden Auftrag benötigt konstante Zeit
 - Z.B. wenn der aktuelle Auftrag durchgelaufen ist oder blockiert

Linux (bis 2.6), Mach, QNX, . . . , VxWorks verhelfen Auftragsauslösung zu $O(1)$





1

Wiederholung

- Ereignisorientierter Planer
- Berechnungskomplexität
- Ereignisgesteuerte & Zeitgesteuerte Echtzeitsysteme

2

eCos-Vertiefung

- Alarme
- Time-Triggered eCos

3

Hinweise zu Aufgabe 4

- Graphische Ausgabe
- Software-Tracing



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

- 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);
```

¹<http://ecos.sourceware.org/docs-la/ref/kernel-counters.html>



eCos-*Uhren* (Clocks²) sind spezialisierte *Zähler*

- Basierend auf *Zeitgeberunterbrechung*
- Festgelegte Zeitaufösung 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

²<http://ecos.sourceware.org/docs-latest/ref/kernel-clocks.html>



eCos-*Alarm*³ 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()

³<http://ecos.sourceware.org/docs-latest/ref/kernel-alarms.html>

eCos-*Alarm*⁴ 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()` + $x \leadsto$ *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)
```

⁴<http://ecos.sourceware.org/docs-latest/ref/kernel-alarms.html>



eCos ist eigentlich *ereignisgesteuert*

~> Studienarbeit: Time-Triggered eCos:

- Zeitgesteuerte Ausführung von Tasks in **Ablauf tabellen**.
- 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 .. }
```



tt-eCos Initialisierung

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_dispatcher_tabletype <Tabellenname>)
```

- Mehrere Tabellen möglich

→ Wechsel zur Laufzeit

```
tt_statustype tt_switchtable(tt_dispatcher_tabletype <Tabellenname>)
```



- Definition der Ereignisse der einzelnen Tabellen.

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

- Starten des Betriebssystems.

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

Zeitpunkte?

Schon wieder Ticks...:

```
cyg_resolution_t ttEcos_get_resolution(void)
```



Für jeden Thread mittels `tt_deadlinemethod` konfigurierbar:

- **TT_STRINGENT**: Strikte Terminüberprüfung
 - *direkt* nach Ablauf des Termins
- **TT_NONSTRINGENT**: Nicht-Strikte Terminüberprüfung
 - einem späteren Zeitpunkt
 - Terminverletzung möglich
 - Mehr Flexibilität und Effizienz

Einplanung von Taskstart oder Terminüberwachung (`tt_action`):

- **TT_START_TASK**: Task-Einplanung
- **TT_DEADLINE**: Terminüberprüfung

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



- *Auftragsorientiertes* 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



1

Wiederholung

- Ereignisorientierter Planer
- Berechnungskomplexität
- Ereignisgesteuerte & Zeitgesteuerte Echtzeitsysteme

2

eCos-Vertiefung

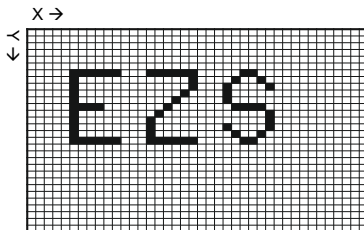
- Alarme
- Time-Triggered eCos

3

Hinweise zu Aufgabe 4

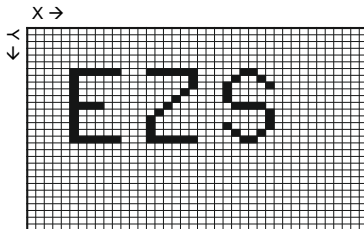
- Graphische Ausgabe
- Software-Tracing





- **Framebuffer** $\leadsto$ Grafische Ausgabe
- eCos bietet einheitliche, *hardwareunabhängig Schnittstelle*
- Darstellung auf Host-PC: `ezs_*` $\leadsto$ `make doc`
- Auflösung in unserem Fall: **800x600**
 - Zwecks Portabilität immer Variablen nutzen:
 - Breite: `CYG_FB_WIDTH(FRAMEBUF)`
 - Höhe: `CYG_FB_HEIGHT(FRAMEBUF)`





Nützliche Funktionen:

```
void ezs_fb_init(void);
```

```
void ezs_fb_clear(cyg_fb_colour color);
```

```
void ezs_fb_fill_block(cyg_ccount16 x, cyg_ccount16 y,  
    cyg_ccount16 width, cyg_ccount16 height, cyg_fb_colour color);
```

```
void ezs_fb_print_string(char* c, cyg_ccount16 x,  
    cyg_ccount16 y, cyg_fb_colour color);
```



- `printf` geht nicht mehr
~> `ezs_printf`
- Serielle Port wird gemultiplext:
 - Cutecom muss ab sofort neues serielles Gerät verwenden
~> `/tmp/<username>-ezs-serial`
- Wir unterstützen nur eine beschränkte Anzahl Farben.
~> `libEZS/include/ezs_io_fel.h`



- Erfassen analoger Spannungen
- „Umkehrung“ des DACs
- Kenngrößen
 - Abtastrate KSPS-GSPS
 - Auflösung 10-24bit
 - Latenz zwischen Messung und Verfügbarkeit der Werte



Bei uns:

- „simulierter“ ADC mit verrauschtem Muster → „unendliche“ Abtastfrequenz
- 8 bit Auflösung

Zugriff

```
void ezs_adc_init(void)
```

```
uint8_t ezs_adc_get(void)
```

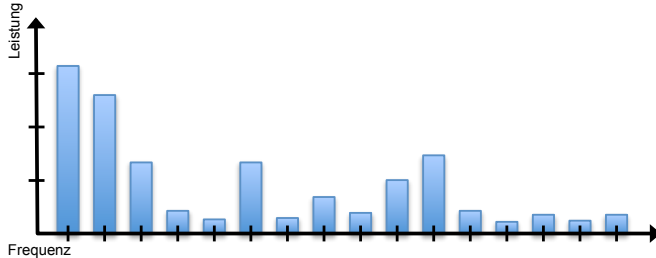


```
void ezs_power_density_spectrum(float in[], float out[], int N)
```

- in[] *Eingabe*, Abschnitt des Zeitbereichssignals
- out[] *Ausgabe*, Leistungsdichtespektrum (LDS)
- N *Anzahl der Abtastwerte*, wobei N Zweierpotenz
- Zeitbereichssignal der Länge N $\mapsto$ LDS⁵ der Länge $\frac{N}{2}$
- Höchste Frequenz im Spektrum aus Abtasttheorem

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

⁵http://en.wikipedia.org/wiki/Spectral_density

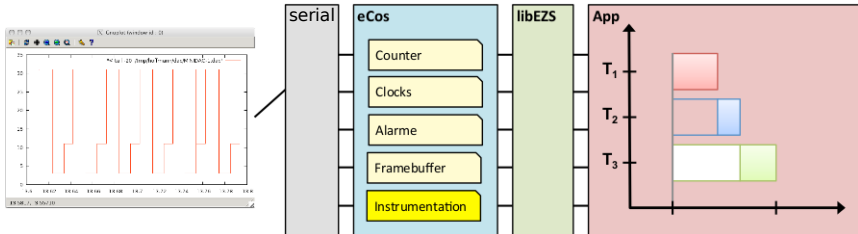


- Darstellung mit 16 Werten:
 - LDS der Länge 32 notwendig
- 1Hz Abstand $\leadsto$ Spektrum bis 16 Hz erfassen
 - Abtastfrequenz 32 Hz



Balken repräsentieren **Leistung**





■ **Visualisierung der Threads** $\leadsto$ Softwarebasiertes Tracing

■ eCos Instrumentations⁶

■ Ausgabe der **Priorität** $\leadsto$ *Ablaufgraph*

$\leadsto$ eindeutige Zuordnung Faden $\rightarrow$ Priorität notwendig

⁶<http://ecos.sourceware.org/docs-latest/user-guide/kernel-instrumentation.html>