Anwendungsentwicklung in eCos

Vertiefung

Florian Franzmann, Martin Hoffmann, Isabella Stilkerich

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

12. November 2012

1. Errata
2. Rekapitulation der Vorlesung
3. Ablaufsteuerung
   - Alarme
4. Zeitsteuerung mit Time-triggered eCos
   - ttKernel
5. Fragen

Errata: Vorgabe 2, Übungsskript

Fehler in Vorgabe zu Aufgabe 2

- Qemu Startskript erwartet `image` Ordner

→ Leeren Ordner im Basisverzeichnis anlegen.

Fehler in Bild (Übungsskript 3, S. 8)

- ~~Latenz~~ → **Antwortzeit**

Rekapitulation der Vorlesung

Kapitel 3-1 und 3-2

Diskussion

Alarme und periodische Aktivierung

eCos Zähler Abstraktion

eCos **Alarme** basieren auf eCos **Zählern** (Counter[1])

- Anwendung erzeugt Zähler für bestimmtes Ereignis
  - Zeitgeberunterbrechung ($\rightarrow$ DSR)
  - externes Ereignis (Taster, etc.)
  - $\rightarrow$ Zähler wird "von Hand" inkrementiert

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

- Alternativ: eCos interne Uhr als Zähler ($\rightarrow$ Aufgabe 3)
- eCos verwaltet Zählerstand intern
- Aktiviert ggf. Alarm bei Erreichen eine Zählerstandes

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

Alarme und periodische Aktivierung – 1

eCos Alarm

eCos **Alarm**[2] führt Aktion bei Erreichen eines Zählerstandes aus

1. Anlegen eines Alarms:

```
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)

[2] ecos.sourceware.org/docs-latest/ref/kernel-alarms.html

Alarme und periodische Aktivierung – 2

eCos Alarm

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` Zählerticks bis zur **ersten** Aktivierung
   - `interval` Zählerintervall für folgende **periodische** Aktivierungen

3. Alarm freischalten

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

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

Alarme und periodische Aktivierung

eCos Uhr

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

- Basierend auf Zeitgeberunterbrechung
- Festgelegte Zeitauflösung 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 ($\rightarrow$ Aufgabe3)

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

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

tt-eCos

eCos ist eigentlich *ereignisgesteuert*

→ Studienarbeit: Time-triggered eCos:

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

Ausführliche Dokumentation

→ Ausarbeitung der Studienarbeit von Michael Lang:

`www.opus.ub.uni-erlangen.de/opus/volltexte/2008/1015/pdf/sa_michael_lang.pdf`

tt-eCos Taskkonstruktion

Ablauftabellen und Tasks werden statisch (global) angelegt:

1. Definition der Ablauftabellen unter Angabe der maximalen Ereigniseinträge. (Makro!)

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

2. Definition der Task(s) und Implementierung des Task-Programms. (Run-to-completion!)

```
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_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> )
```


Terminüberwachung

Für jeden Thread mttels `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 <
    Tabellenname>,
tt_ticktype <Zeitpunkt>,
tt_action <Ereignis>,
tt_tasktype  <Task–ID> )
```

Fragen

Fragen?