

EZS Handwerkszeug

Übung zur Vorlesung EZS

Florian Franzmann, Martin Hoffmann, Isabella Stilkerich

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

5. November 2012

Exkursion zu Siemens

Siemens Motion Control veranstaltet am **20.11.2012 von 10:00 - 16:15 Uhr** eine **Exkursion ins Erlanger Gerätewerk**.

Das ist eine einmalige Gelegenheit, sich die Entwicklung und Fertigung von herausfordernden Echtzeitsystemen aus nächster Nähe anzusehen.

Verbindliche Anmeldungen bis spätestens 09.11.2012 per eMail an Fabian Scheler, die Anzahl der Teilnehmer ist auf 10 begrenzt!

1 Organisatorisches

- Übungsaufgaben

2 Handwerkszeug

- Zeitmessungen
- libEZS
- eCos Unterbrechungsbehandlung

3 Fehlersuche

- DDD - GDB

Implementierung der Übungsaufgaben

Programmieren in C/C++

- Grundlegende Programmierkenntnisse werden vorausgesetzt
- Übungen sind kein C/C++ Kurs → Selbstständiges einarbeiten

Übungsaufgaben

- Sprachkenntnisse nur in geringem Umfang nötig
- Alle notwendigen Funktionen werden besprochen
- Hilfestellung in der Rechnerübung

Hilfestellung und Selbststudium:

- Vorlesung SP1:
http://www4.cs.fau.de/Lehre/SS11/V_SP1/Folien/
- Bibliothek: Zahllose Bücher über C Programmierung

Interaktion mit dem System

Beobachtung

Simulator

- Virtueller Signalgenerator
~ Sinus

Hardware



Sampling eines Signals in der Anwendung

- libEVS bietet einheitliche Schnittstelle ~ qemu und Hardware
 - Liest den Wert des Software-Funktionsgenerators
 - Liefert den Inhalt des Hardware-ADC Registers zurück
- Wert lesen: `evs_adc_read()`

- 1 Organisatorisches
 - Übungsaufgaben

- 2 Handwerkszeug

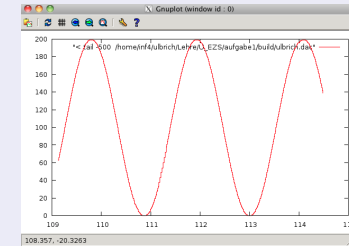
- Zeitmessungen
- libEVS
- eCos Unterbrechungsbehandlung

- 3 Fehlersuche
 - DDD - GDB

Interaktion mit dem System

Beeinflussung

Simulator



Hardware

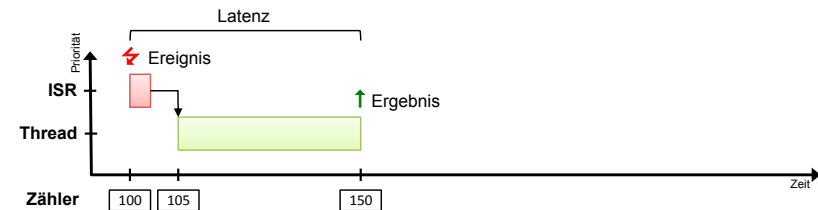


Analyse der Schrittfunktion der Anwendung

- libEVS bietet einheitliche Schnittstelle ~ qemu und Hardware
 - Schreibt den Wert auf einen DAC
- Wert schreiben: `evs_dac_set()`

Zeitmessung

Die Stoppuhr



Stoppuhr

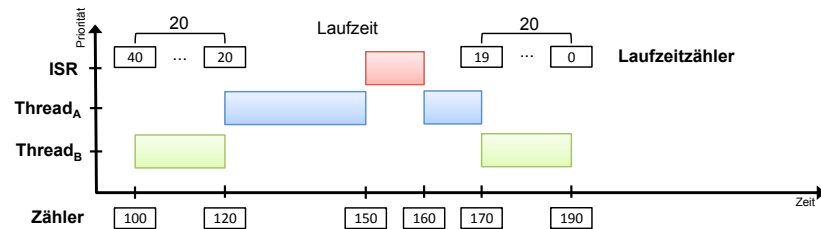
- Punkte auf der Zeitachse t_1 und t_2 ~ Ereignis und Ergebnis
- Latenz ist $\Delta t = t_2 - t_1$ (Beispiel: $150 - 100 = 50$ Zählerticks)

Umsetzung

- Zähler (durch libEVS vorgegeben), `start` und `stop` Funktion
- Startzeitpunkt t_1 ist Zustand der Messung ~ Globale Variable pro Messpaar

Zeitmessung

Rechenzeitverbrauch



Rechenzeitsimulation

- Verbrauchte **Laufzeit** eines Threads
- Vorgegebene Zeit aktiv warten $\leadsto$ Laufzeit verbrauchen

Umsetzung

- Funktion die aktiv t_{wcet} wartet $\leadsto$ Schleife auf Zählerwert
- Zähler läuft während Unterbrechungen weiter! $\leadsto$ lokaler Zähler
- Dekrement bei jeder Änderung! Beispiel: Sprung von 120 $\rightarrow$ 170

Zähler in Mikrocontrollern

Zähler (*Counter*) zählen hardwarebasiert Ereignisse z.B. von:

- Externem Drehgeber (Radumdrehung)
- Externem Quarz (Real-Time Clock)
- Internem Prozessortakt (hohe Auflösung)

Äquidistante Ereignisse ermöglichen einen **Zeitgeber** (*Timer*) für

- Periodische Aktivierung
- **Messen von Zeitabständen**
- (kontrolliertes Verbrennen von Prozessorzeit)

Zähler Betriebsmodi

Zähler bzw. Zeitgeber bieten zwei Betriebsmodi:

Abfragebetrieb (Polling) Aktives Auslesen des Zählers, bis zum Erreichen eines vorgegebenen Wertes.

Unterbrecherbetrieb (Interrupt) Der Zähler unterbricht das laufende System beim Erreichen eines vorkonfigurierten Zählerstandes.

Hinweis

Für die Laufzeit (WCET¹) Simulation nutzen wir den Abfragebetrieb.

¹Worst Case Execution Time = Größtmögliche Laufzeit (eines Threads, Funktion, ...)

libEZS Überblick

Plattformunabhängige Hilfsfunktionen

- Zeitmessung
- WCET Simulation
- ADC-Zugriff (Signalgenerator)
- DAC-Ausgabe
- ...

```
aufgabe2
|-- CMakeLists.txt
|-- app.c
|-- ecos
|-- drivers
|   |-- counter.h
|   |-- i386
|   |-- i386_adc.c
|   |-- i386_counter.c
|   |-- i386_dac.c
|-- libEZS
|   |-- adc.h
|   |-- dac.h
|   |-- stopwatch.c
|   |-- stopwatch.h
```

Die *libEZS* wird ständig (auch von euch) erweitert.

libEVS Zeitmessung *stopwatch.c/.h*

Für die Zeitmessung soll die libEVS zwei Funktionen bereitstellen:

```
void ezs_watch_start(cyg_uint32 * state);
cyg_uint32 ezs_watch_stop(cyg_uint32 * state);
```

- Parameter: Zeiger auf (globale) Variable
→ Unabhängige Messzeitpunkte
- `ezs_watch_stop(cyg_uint32 * state)` gibt die Zeitdifferenz in Ticks zurück

Hinweis

Für die Implementierung wird ein i386 Zähler mit einer bestimmten Auflösung bereitgestellt.

→ `ezs_counter_resolution_us()`, `ezs_counter_get()` in `drivers/counter.h`

1 Organisatorisches

- Übungsaufgaben

2 Handwerkszeug

- Zeitmessungen
- libEVS
- eCos Unterbrechungsbehandlung

3 Fehlersuche

- DDD - GDB

libEVS WCET-Simulator *stopwatch.c/.h*

Zur WCET-Simulation soll folgende Funktion implementiert werden:

```
void ezs_watch_wcet(cyg_uint32 wcet);
```

- Parameter: gewünschte WCET in 10 μ s Ticks
- Die Implementierung muss einen internen Zähler nutzen, der bei jeder Änderung des Systemzählers angepasst wird.
- Hierbei muss der Abfragebetrieb genutzt werden.
- Es wird keine globale Zustandsvariable benötigt.
→ Jeder Thread besitzt einen eigenen Stack!

Hinweis

Für die Implementierung wird ein i386 Zähler mit einer bestimmten Auflösung bereitgestellt.

→ `ezs_counter_resolution_us()`, `ezs_counter_get()` in `drivers/counter.h`

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR²

```
#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
- siehe Hardwarehandbuch

²ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR²

```
#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)

²ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR²

```
#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

²ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR²

```
#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
);
```

- Funktionspointer ISR Implementierung

Signatur:

cyg_uint32 (*) (cyg_vector_t, cyg_addrword_t)

²ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR²

```
#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
);
```

- Funktionspointer DSR Implementierung

Signatur:

cyg_uint32 (*) (cyg_vector_t, cyg_uint32 count, cyg_addrword_t)

²ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR²

```
#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

²ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html

ISR Implementierungsskelett

Beispiel für eine minimale Interrupt Service Routine

```
cyg_uint32 isr(cyg_vector_t vector, cyg_addrword_t data) {
    cyg_bool_t dsr_required = 0;
    ...
    cyg_acknowledge_isr(vector);
    return dsr_required ?
        (CYG_ISR_CALL_DSR | CYG_ISR_HANDLED) :
        CYG_ISR_HANDLED;
}
```

- 1 Beliebiger ISR Code
- 2 Bestätigung der Interruptbehandlung
- 3 Anforderung einer DSR
oder
- 4 Rückkehr ohne DSR

ISR Implementierungsskelett

Beispiel für eine minimale Interrupt Service Routine

```
cyg_uint32 isr(cyg_vector_t vector, cyg_addrword_t data) {
    cyg_bool_t dsr_required = 0;
    ...
    cyg_acknowledge_isr(vector);
    return dsr_required ?
        (CYG_ISR_CALL_DSR | CYG_ISR_HANDLED) :
        CYG_ISR_HANDLED;
}
```

- 1 Beliebiger ISR Code
- 2 Bestätigung der Interruptbehandlung
- 3 Anforderung einer DSR
oder
- 4 Rückkehr ohne DSR

ISR Implementierungsskelett

Beispiel für eine minimale Interrupt Service Routine

```
cyg_uint32 isr(cyg_vector_t vector, cyg_addrword_t data) {
    cyg_bool_t dsr_required = 0;
    ...
    cyg_acknowledge_isr(vector);
    return dsr_required ?
        (CYG_ISR_CALL_DSR | CYG_ISR_HANDLED) :
        CYG_ISR_HANDLED;
}
```

- 1 Beliebiger ISR Code
- 2 Bestätigung der Interruptbehandlung
- 3 Anforderung einer DSR
oder
- 4 Rückkehr ohne DSR

ISR Implementierungskelett

Beispiel für eine minimale Interrupt Service Routine

```
cyg_uint32 isr(cyg_vector_t vector, cyg_addrword_t data) {
    cyg_bool_t dsr_required = 0;
    ...
    cyg_acknowledge_isr(vector);
    return dsr_required ?
        (CYG_ISR_CALL_DSR | CYG_ISR_HANDLED) :
        CYG_ISR_HANDLED;
}
```

- 1 Beliebiger ISR Code
- 2 Bestätigung der Interruptbehandlung
- 3 Anforderung einer DSR
oder
- 4 Rückkehr ohne DSR

DSR Implementierungskelett

Beispiel für eine minimale Deferrable Service Routine

```
void dsr_function(
    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

DSR Implementierungskelett

Beispiel für eine minimale Deferrable Service Routine

```
void dsr_function(
    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

- 1 Organisatorisches
 - Übungsaufgaben

- 2 Handwerkszeug
 - Zeitmessungen
 - libEZS
 - eCos Unterbrechungsbehandlung

- 3 Fehlersuche
 - DDD - GDB

Die Anwendung im Debugger (DDD) ausführen

- Debugger starten und die Anwendung laden:

```
make simddd
```

- die Ausführung starten:

- **Cont**-Button klicken
- auf der GDB-Konsole folgendes eingeben:

```
(gdb) cont
```

Achtung

Der **Run**-Button bzw.

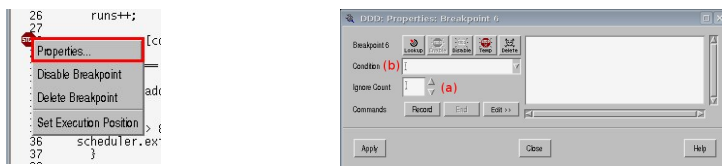
```
(gdb) run
```

wird **nicht** funktionieren!

Bedingungen für Breakpoints

Bevor das Programm an einem Breakpoint tatsächlich angehalten wird, kann man ...

- (a) ... den Breakpoint n-mal ignorieren
 - (b) ... auf die Erfüllung einer Bedingung warten.
- aus dem Quelltextfenster heraus

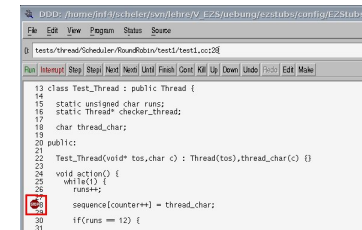


- auf der GDB-Konsole

```
(gdb) ignore <breakpoint_nr> <n> # (a)
(gdb) condition <breakpoint_nr> runs == 7 # (b)
```

Breakpoints setzen ...

- ... durch einen Doppelklick ins Quelltextfenster



- ... auf der GDB-Konsole für eine bestimmte Funktion

```
(gdb) break <function_name>
(gdb) break <file_name>:<line>
```

Achtung

Auf eingebettete Funktionen kann man keinen Breakpoint setzen!

Bedingungen für Breakpoints (Forts.)

Achtung

bedingte Breakpoints können **Ein-/Ausgabe-Operationen** zur Folge haben, die das **Laufzeitverhalten** des Programms beeinflussen können.

Weitere Kontrollmöglichkeiten

• Break

- Anhalten des Programms “von außen”.
- normalerweise per `Ctrl-C` in der GDB-Konsole

• Watchpoints

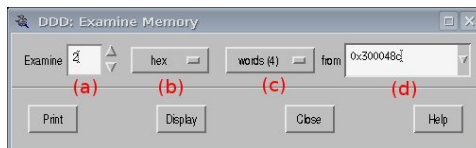
```
(gdb) watch xyz
```

- Programm anhalten wenn Variable `xyz` ihren Wert ändert

Speicherstellen beobachten

• Menü *Data* →

- *Memory*



• Parameter

- Wie viele Bytes, Half Words, Words, ...
- Hexadezimal, Oktal, ...
- Bytes, Half Words (2 Bytes), Words (4 bytes), ...
- Startadresse

- Hilfreich zum Kontrollieren von I/O-Registern

Werte von Variablen beobachten

• globale Variablen und Objekte

```
3 class Test_Thread : public Thread {
4
5 static unsigned char runs;
6 static Thread* checker; Print runs;
7
8 char thread_char;
9
10 public:
11     Test_Thread(void* tos, c
12     void action() {
13         while(1) {
14             runs++;
15             sequence[counter++] Break at r
16             if(runs == 12) {
17                 Clear at ru
18             }
19             scheduler.add(checker, thread);
20         }
21     }
22 }
```

```
(gdb) # Achtung: vollqual. Name!
(gdb) graph display Test_Thread::runs
```

• lokale Variablen und Argumente

• Menü *Data* →

- *Display Local Variables*
- *Display Arguments*

• Beobachtung problematisch:

- Speicherstellen/Register werden vom GCC wiederverwendet
- passende Debug-Information wird (noch) nicht erzeugt

Neustart ohne DDD zu beenden

• zunächst muss man die Verbindung zum GDB-Stub trennen

```
(gdb) detach
```

• erneut

eine Verbindung aufbauen (<PORT> steht in der Ausgabe von `make`)

```
(gdb) target remote localhost:<PORT>
```

• die Anwendung laden

```
load
```

• das hat einige Vorteile:

- Breakpoints und
- Watches bleiben erhalten

Debugger neu starten

Manchmal kann es vorkommen, dass der GDB-Stub bzw. der qemu sich komisch verhält. Man hat dann drei Möglichkeiten:

- 1 DDD beenden und die Anwendung erneut laden
 - Breakpoints und Watches gehen verloren
- 2 nur den GDB beenden
 - den arm-eabi-gdb beenden
- DDD wird das bemerken und folgender Dialog erscheint



- auf den "Restart GDB"-Button klicken

Fragen?

Debugger neu starten

- 8 den kompletten DDD neu starten
 - aus dem Menü *File* die Option *Restart* auswählen

