

EZS Handwerkszeug

Übung zur Vorlesung EZS

Martin Hoffmann, Florian Franzmann, Tobias Klaus

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

4. November 2013

1 Handwerkszeug

- Zeitmessungen
- libEZS
- eCos Unterbrechungsbehandlung

2 Fehlersuche

- DDD - GDB - CGDB

1 Überblick

Interaktion mit dem System

Beobachtung

Hardware



Simulator

- Virtueller Signalgenerator
~ Sinus

Sampling eines Signals in der Anwendung

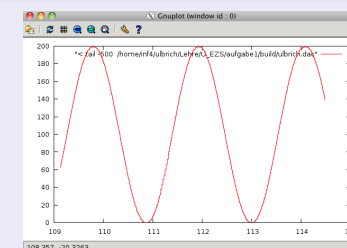
- libEZS bietet einheitliche Schnittstelle
 - Liest den Wert des Funktionsgenerators (ADC)
 - Liefert den Inhalt des Hardware-ADC Registers zurück
- Wert lesen: `ezs_adc_read()`

1 Überblick

Interaktion mit dem System

Beeinflussung

Simulator



Hardware



Analyse der Schrittfunktion der Anwendung

- libEZS bietet einheitliche Schnittstelle
 - Schreibt den Wert auf einen DAC
- Wert schreiben: `ezs_dac_write()`

1 Handwerkszeug

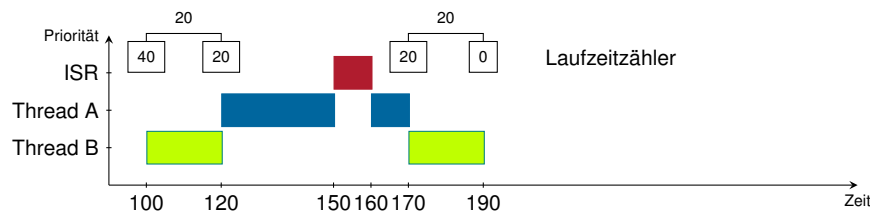
- Zeitmessungen
- libEzS
- eCos Unterbrechungsbehandlung

2 Fehlersuche

- DDD - GDB - CGDB

Zeitmessung

Rechenzeitverbrauch



Rechenzeitsimulation

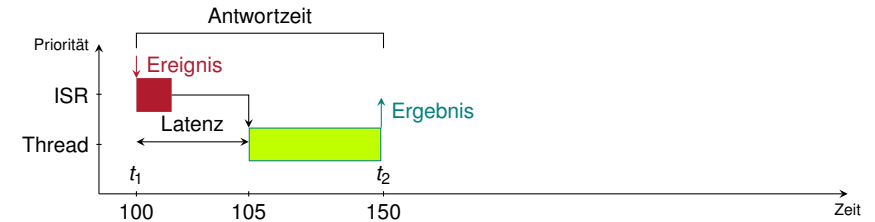
- Verbrauchte **Laufzeit** eines Threads
- Vorgegebene Zeit aktiv warten $\leadsto$ Laufzeit verbrauchen

Umsetzung

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

Zeitmessung

Die Stoppuhr



Stoppuhr

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

Umsetzung

- Zähler (durch libEzS vorgegeben), start und stop Funktion
- Startzeitpunkt t_1 ist Zustand der Messung $\leadsto$ Globale Variable pro Messpaar

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 Zeitmessung *stopwatch.c/.h*

Für die Zeitmessung soll die libEZS 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/include/ezs_counter.h`

libEZS Überblick

Plattformunabhängige Hilfsfunktionen

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

```
aufgabe2
|-- CMakeLists.txt
|-- app.c
|-- ecos
|-- drivers
| |-- include
| | |-- counter.h
| | |-- i386
| | |-- ezs_adc.c
| | |-- ezs_counter.c
| | |-- ezs_dac.c
|-- libEZS
| |-- include
| | |-- ezs_adc.h
| | |-- ezs_dac.h
| | |-- stopwatch.h
|-- src
| |-- stopwatch.c
```

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

libEZS 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/include/ezs_counter.h`

1 Handwerkszeug

- Zeitmessungen
- libEzS
- eCos Unterbrechungsbehandlung

2 Fehlersuche

- DDD - GDB - CGDB

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_ucount32 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 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

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

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

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 Handwerkszeug

- Zeitmessungen
- libEZS
- eCos Unterbrechungsbehandlung

2 Fehlersuche

- DDD - GDB - CGDB

Die Anwendung im Debugger (DDD) ausführen

- Debugger starten und die Anwendung laden:

```
make ddd
```

- die Ausführung starten:

- *Continue* ddd: Cont-Button)
- auf der GDB-Konsole folgendes eingeben:

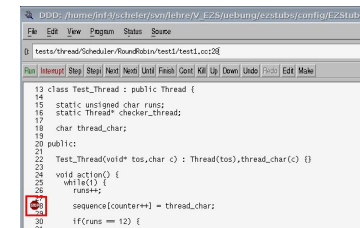
```
(gdb) cont
```

Achtung

Der *Run*-Button bzw. (gdb) *run* wird **nicht** funktionieren!

Breakpoints setzen ...

- ... durch einen Doppelklick ins Quelltextfenster



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

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

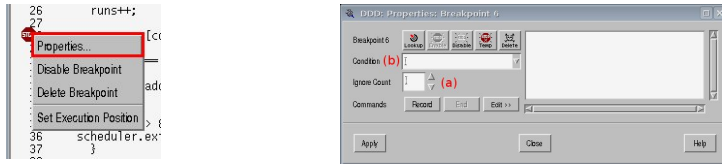
Achtung

Auf eingebettete Funktionen kann man keinen Breakpoint setzen!

Bedingungen für Breakpoints

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)
```

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

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 char thread_char; Display run;
8
9 public:
10 Test_Thread(void* tos, c Display *r) {
11 void action() { What is run;
12 while(1) {
13 runs++; Lookup runs;
14 sequence[counter++] Break at r;
15 if(runs == 12) { Clear at r;
16 scheduler.add(checker, thread);
17 }
```

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

- lokale Variablen und Argumente

- Menü *Data* →

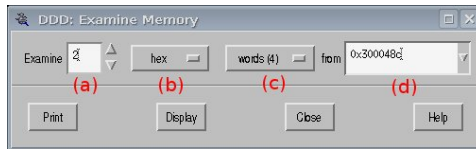
- *Display Local Variables*
- *Display Arguments*

- Beobachtung problematisch:

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

Speicherstellen beobachten

- Menü *Data* →
 - Memory*



- Parameter
 - (a) Wie viele Bytes, Half Words, Words, ...
 - (b) Hexadezimal, Oktal, ...
 - (c) Bytes, Half Words (2 Bytes), Words (4 bytes), ...
 - (d) Startadresse
- Hilfreich zum Kontrollieren von I/O-Registern

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:

- DDD beenden und die Anwendung erneut laden
 - Breakpoints und Watches gehen verloren
- nur den GDB beenden
 - den gdb beenden

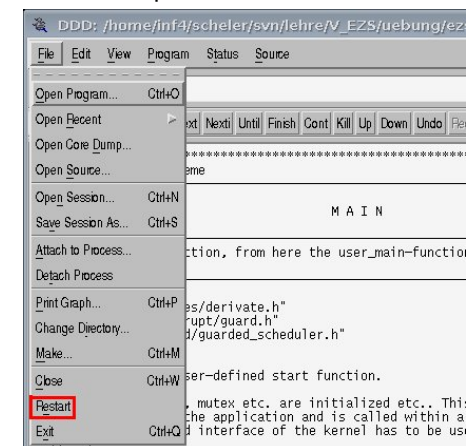

```
killall gdb
```
 - DDD wird das bemerken und folgender Dialog erscheint



- auf den "Restart GDB"-Button klicken

Debugger neu starten

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



Fragen?