

# EZS Handwerkszeug

## Übung zur Vorlesung EZS

Florian Franzmann   Martin Hoffmann   Tobias Klaus  
Peter Wägemann

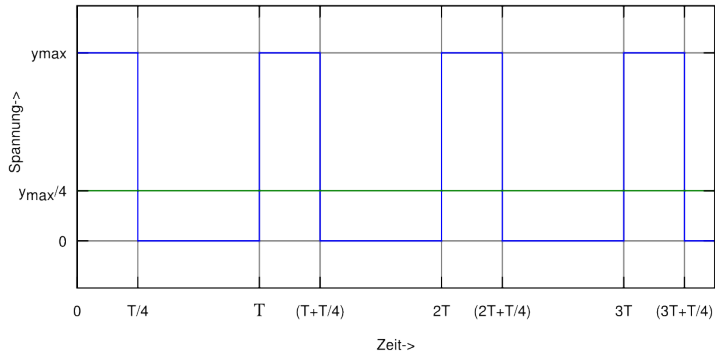
Friedrich-Alexander-Universität Erlangen-Nürnberg  
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)  
<http://www4.cs.fau.de>

20. Oktober 2014

- 1 Handwerkszeug
  - PWM
  - Oszilloskop Cursor

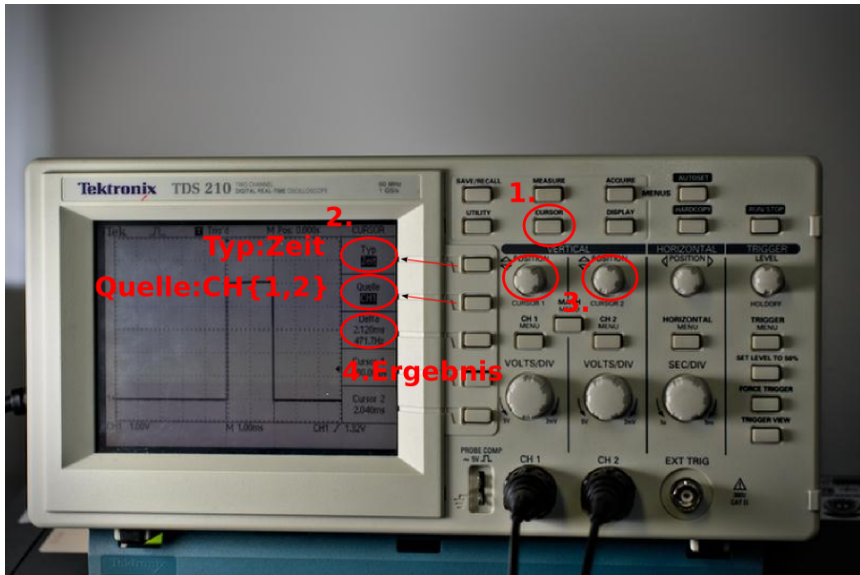
- 2 Ausblick
  - eCos Unterbrechungsbehandlung
  - DDD - GDB - CGDB

# Pulsweiten Modulation



- Tiefpass  $\rightsquigarrow$  Digital-Analog-Wandlung
- Weit verbreitet:
  - Motoren
  - LEDs
- `libezs`: `ezs_dac_write(uint8_t)`

# Cursor



- 1 Handwerkszeug
  - PWM
  - Oszilloskop Cursor

- 2 Ausblick
  - eCos Unterbrechungsbehandlung
  - DDD - GDB - CGDB

# Einordnung

- Vorschau
- Hilfestellung für die folgende Aufgaben
- Allgemeingültige Hilfestellungen
  
- Schuld ist die Baustelle im Hochhaus 😞

# Wie behandle ich einen Interrupt?

## Anmeldung von ISR und DSR<sup>1</sup>

```
#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

---

<sup>1</sup>[ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html](http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html)

# Wie behandle ich einen Interrupt?

## Anmeldung von ISR und DSR<sup>1</sup>

```
#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)

---

<sup>1</sup>[ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html](http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html)

# Wie behandle ich einen Interrupt?

## Anmeldung von ISR und DSR<sup>1</sup>

```
#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

---

<sup>1</sup>[ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html](http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html)

# Wie behandle ich einen Interrupt?

## Anmeldung von ISR und DSR<sup>1</sup>

```
#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)
```

---

<sup>1</sup>[ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html](http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html)

# Wie behandle ich einen Interrupt?

## Anmeldung von ISR und DSR<sup>1</sup>

```
#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)
```

---

<sup>1</sup>[ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html](http://ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html)

# Wie behandle ich einen Interrupt?

## Anmeldung von ISR und DSR<sup>1</sup>

```
#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

---

<sup>1</sup>[ecos.sourceware.org/docs-latest/ref/kernel-interrupts.html](http://ecos.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 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

# DSR Implementierungsskelett

## 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 Implementierungsskelett

## 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
  - PWM
  - Oszilloskop Cursor

- 2 Ausblick
  - eCos Unterbrechungsbehandlung
  - DDD - GDB - CGDB

# Debugger

- Grundlegendes Konzept:
  - Ausführung
    - Stoppen/Fortführen
    - Beim Erreichen von Codezeilen stoppen  $\leadsto$  Breakpoint
  - Speicher
    - Untersuchen/Ausgeben
    - Halt bei Veränderung  $\leadsto$  Watchpoint
    - Verändern

Was tut das Programm *wirklich*?

Im weiteren Verlauf nur ddd/gdb. Wichtig für spätere Aufgaben.

# 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

```

DDD: /home/inf1/scheler/svn/lehre/V_EZS/uebung/ezstubs/config/EZstubs_A
File Edit View Program Status Source
0 tests/thread/Scheduler/RoundRobin/test1/test1.cc:128
Run Interrupt Step Skip Next Next1 Until Finish Cont Kill Up Down Undo Redo Edit Make
13 class Test_Thread : public Thread {
14
15     static unsigned char runs;
16     static Thread* checker_thread;
17     char thread_char;
18
19 public:
20
21     Test_Thread(void* tos, char c) : Thread(tos), thread_char(c) {}
22
23     void action() {
24         while(1) {
25             runs++;
26
27             sequence[counter++] = thread_char;
28             if(runs == 12) {
29
30
31

```

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

```

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

```

## Achtung

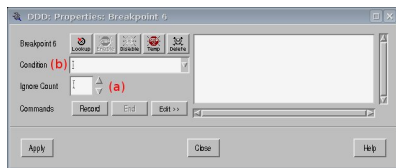
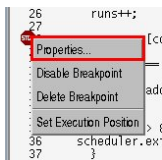
Auf eingebettete Funktionen kann man keinen Breakpoint setzen!

# 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;
7   Print runs
8   char thread_char;
9   Display runs
10 public:
11   Test_Thread(void* tos, c
12
13   void action() {
14     while(1) {
15       runs++;
16       Look up runs
17
18       sequence[counter++]
19       Break at runs;
20       Clear at ru
21   if(runs == 12) {
22
23   scheduler.add(checker, thread);

```

(gdb) # Achtung: vollqual. Name!  
 (gdb) graph display Test\_Thread::runs

- lokale Variablen und Argumente

- Menü *Data* →

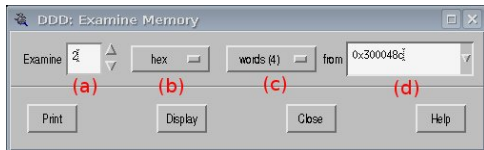
- Display Local Variables*
- Display Arguments*

- 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:

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

```
killall gdb
```

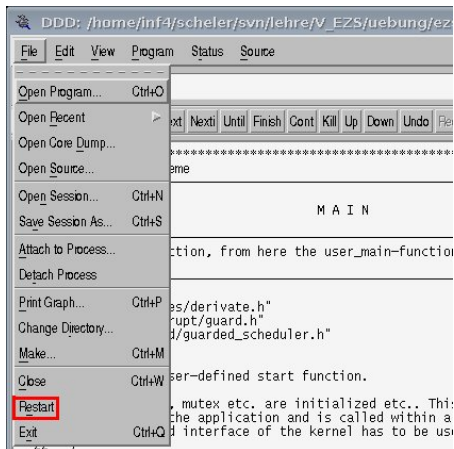
- DDD wird das bemerken und folgender Dialog erscheint



- auf den *“Restart GDB”*-Button klicken

# Debugger neu starten

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



# Fragen?