EZS Handwerkszeug

Übung zur Vorlesung EZS

Florian Franzmann Martin Hoffmann Tobias Klaus
Peter Wägemann

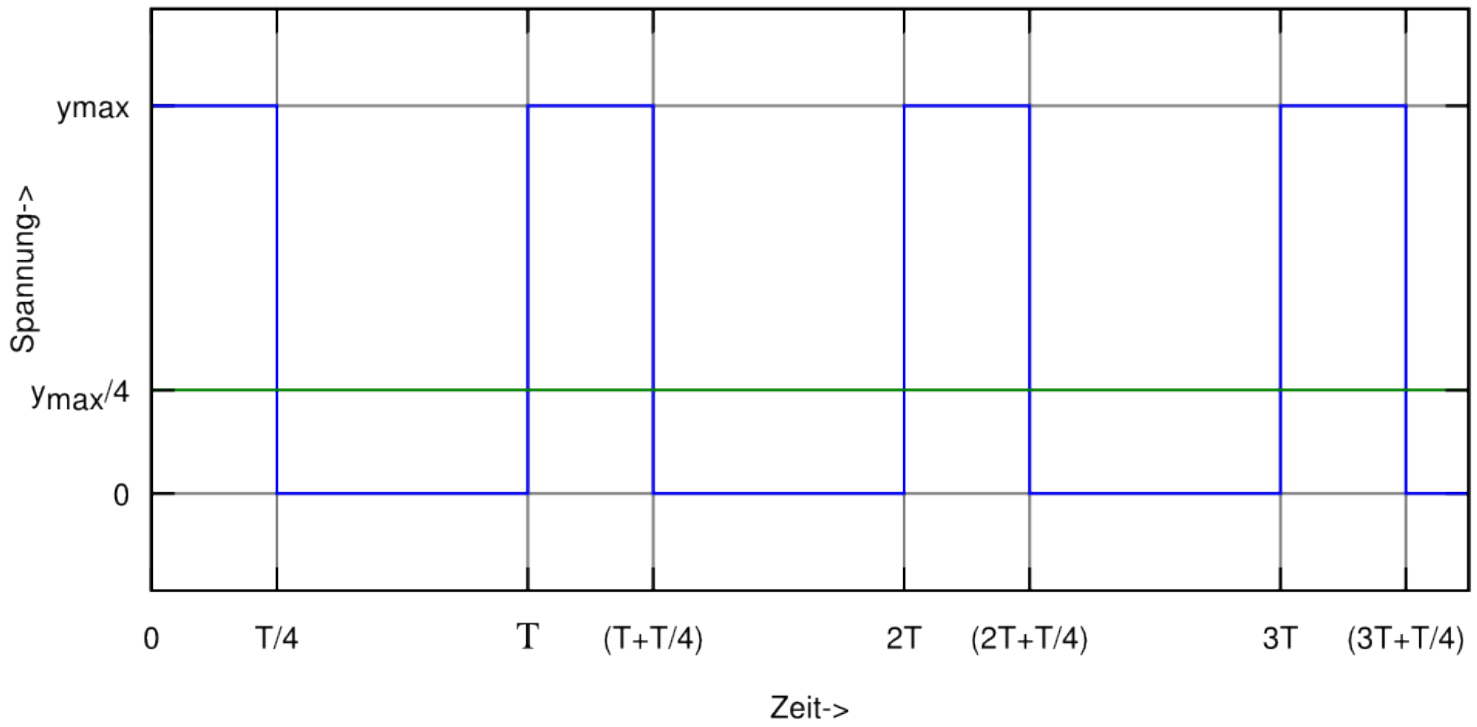
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
http://www4.cs.fau.de

20. Oktober 2014

1. Handwerkszeug
   - PWM
   - Oszilloskop Cursor

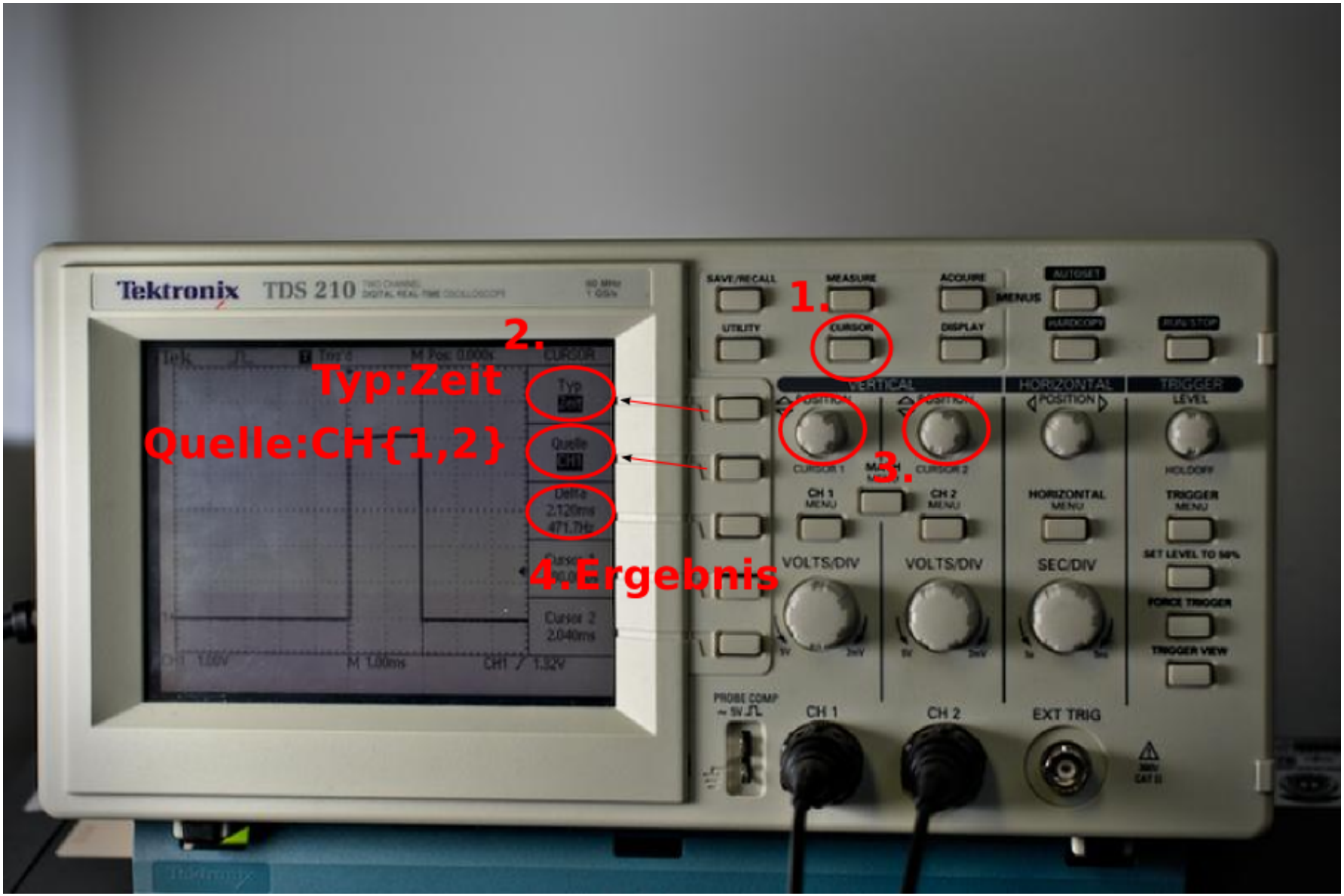
2. Ausblick
   - eCos Unterbrechungsbehandlung
   - DDD - GDB - CGDB

PulsWeiten Modulation



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

Cursor


Tektronix TDS 210
1.
2.
Typ:Zeit
Quelle:CH{1,2}
3.
4.Ergebnis
CURSOR
Typ
Quelle
Delta
Cursor 2
SAVE/RECALL
MEASURE
ACQUIRE
MENUS
UTILITY
CURSOR
DISPLAY
AUTOSET
VERTICAL
HORIZONTAL
TRIGGER
POSITION
LEVEL
HOLDOFF
CH 1 MENU
CH 2 MENU
HORIZONTAL MENU
TRIGGER MENU
VOLTS/DIV
VOLTS/DIV
SEC/DIV
SET LEVEL TO 50%
FORCE TRIGGER
TRIGGER VIEW
PROBE COMP
CH 1
CH 2
EXT TRIG

1. Handwerkszeug
   - PWM
   - Oszilloskop Cursor

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

Einordnung

- Vorschau
- Hilfestellung für die folgende Aufaben
- Allgemeingültige Hilfestellungen

- Schuld ist die Baustelle im Hochhaus ☹

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[1]

```
#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

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

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[1]

```
#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)

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

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[1]

```
#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

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

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[1]

```
#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)
```

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

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[1]

```
#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)
```

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

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[1]

```
#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

[1] ecos.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

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

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

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)

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
   - PWM
   - Oszilloskop Cursor

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

Debugger

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

Debugger

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

Was tut das Program *wirklich*?

Debugger

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

Was tut das Program *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:

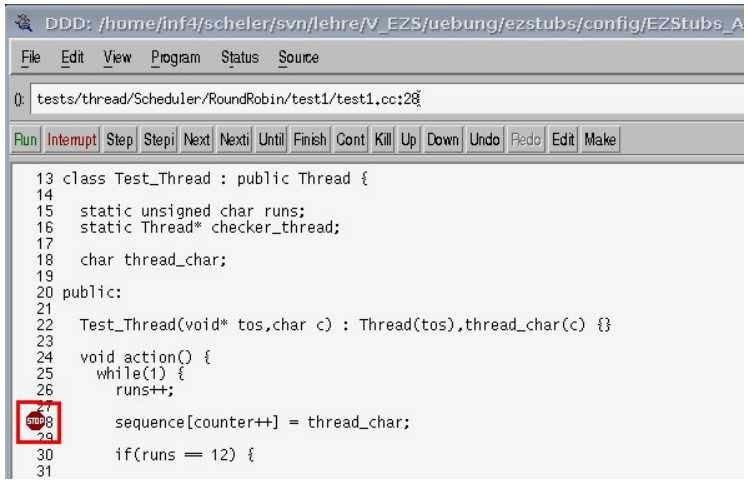
```
(gdb) cont
```

Achtung

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

Breakpoints setzen . . .

- . . . durch einen Doppelklick ins Quelltextfenster



- . . . 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!

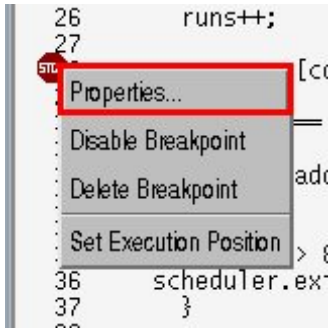
Bedingungen für Breakpoints

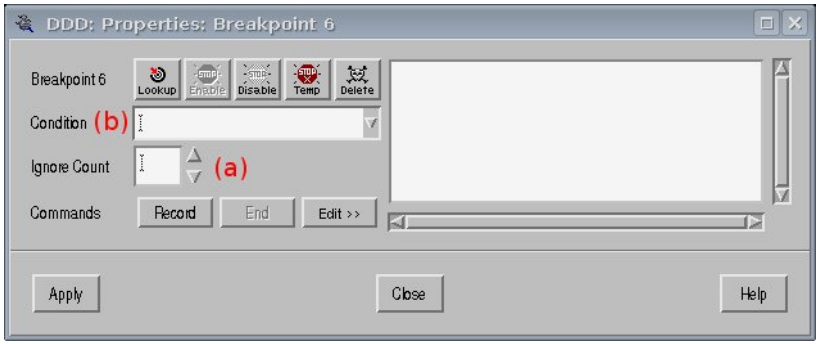
Bevor das Programm an einem Breakpoint tatsächlich angehalten wird, kann man ...

(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

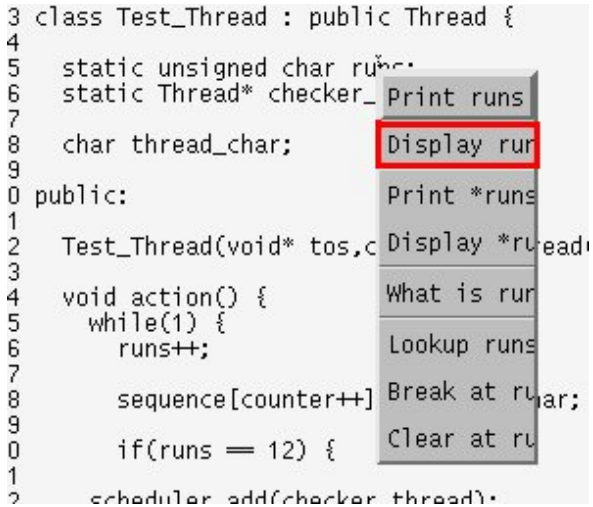
- Watchpoints

```
(gdb) watch xyz
```

  - Programm anhalten wenn Variable `xyz` ihren Wert ändert

Werte von Variablen beobachten

- globale Variablen und Objekte

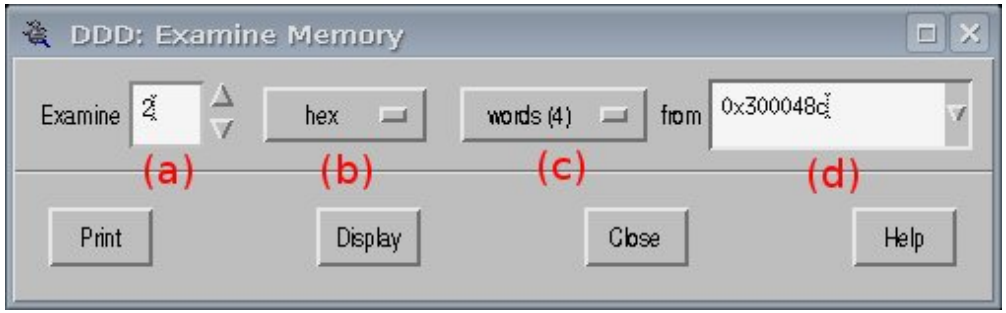



```
(gdb) # Achtung: vollqual. Name!
(gdb) graph display Test_Thread::runs
```

- 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
     ```

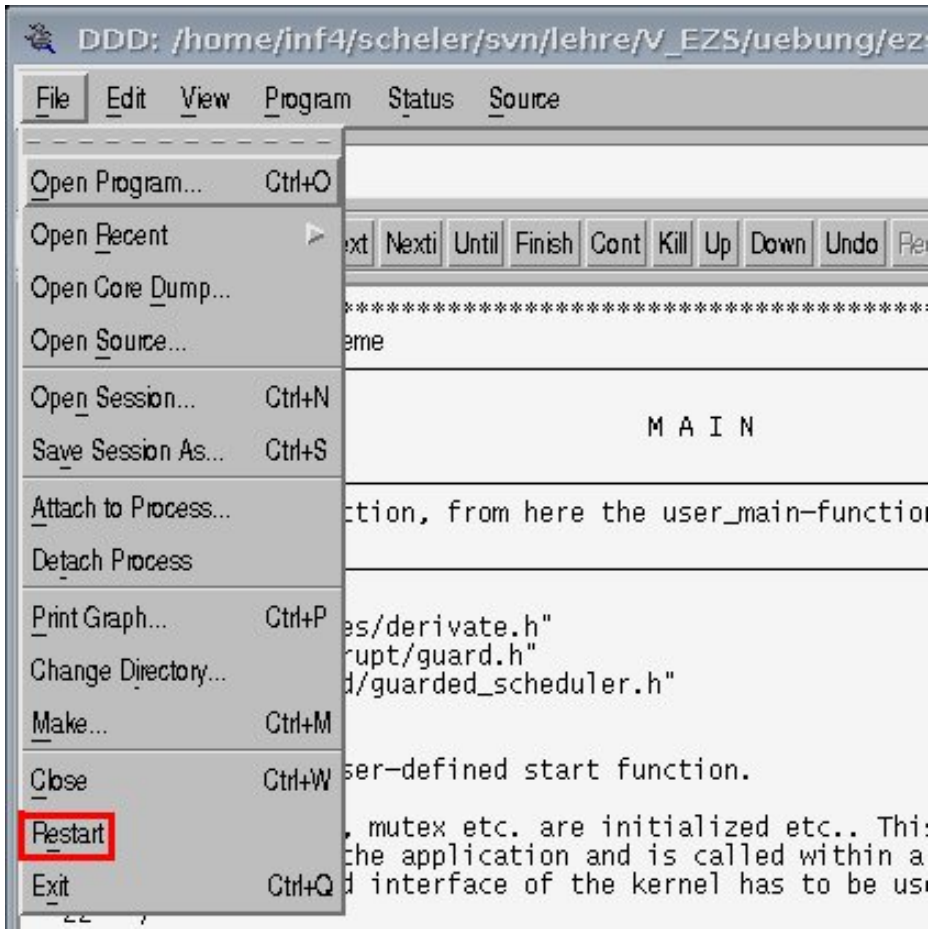
   - DDD wird das bemerken und folgender Dialog erscheint

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

Debugger neu starten

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



Fragen?