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

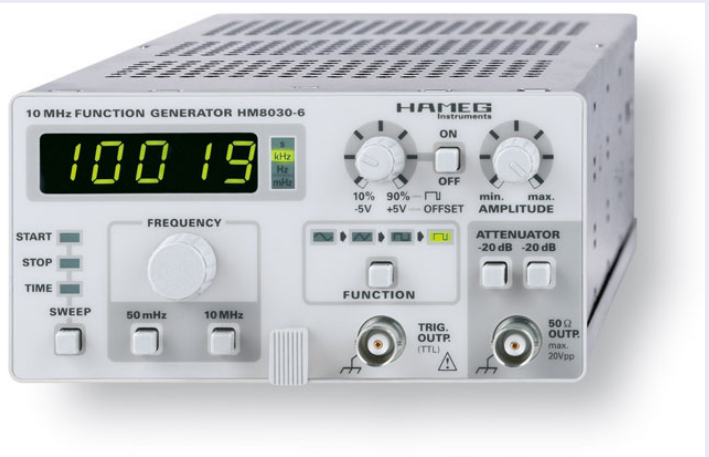
Interaktion mit dem System

Beobachtung

Simulator

- Virtueller Signalgenerator
  $\leadsto$ Sinus

Hardware



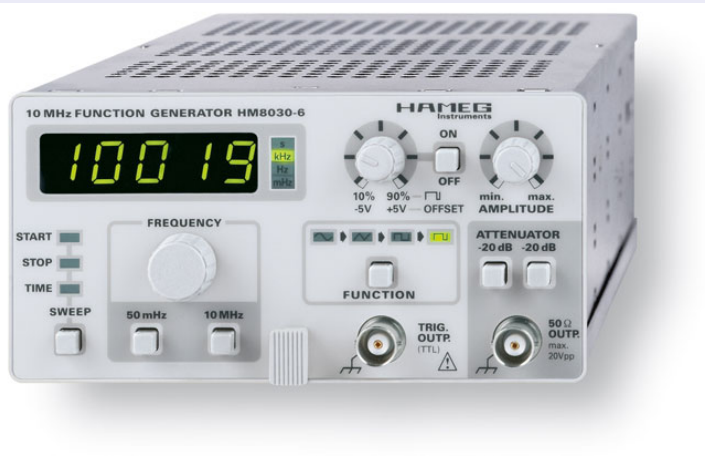
Interaktion mit dem System

Beobachtung

Simulator

- Virtueller Signalgenerator
  $\leadsto$ Sinus

Hardware



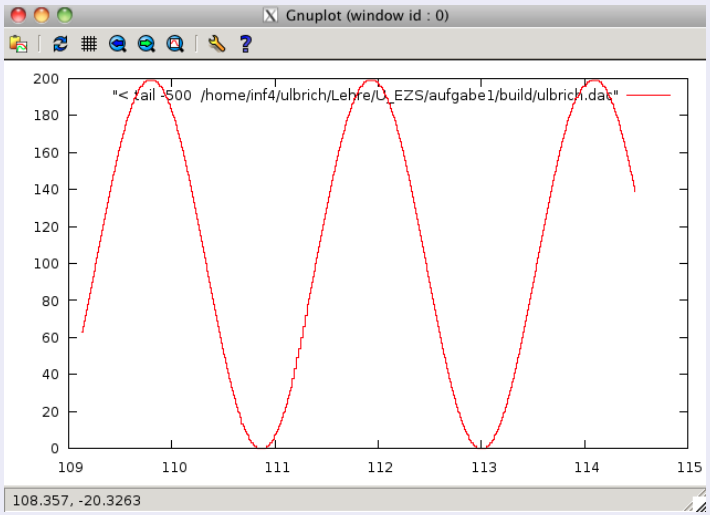
Sampling eines Signals in der Anwendung

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

Interaktion mit dem System

Beeinflussung

Simulator

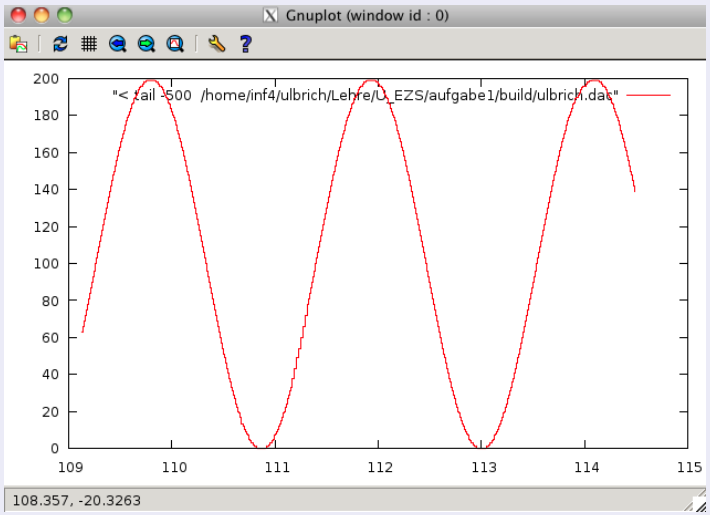


Hardware

Interaktion mit dem System

Beeinflussung

Simulator



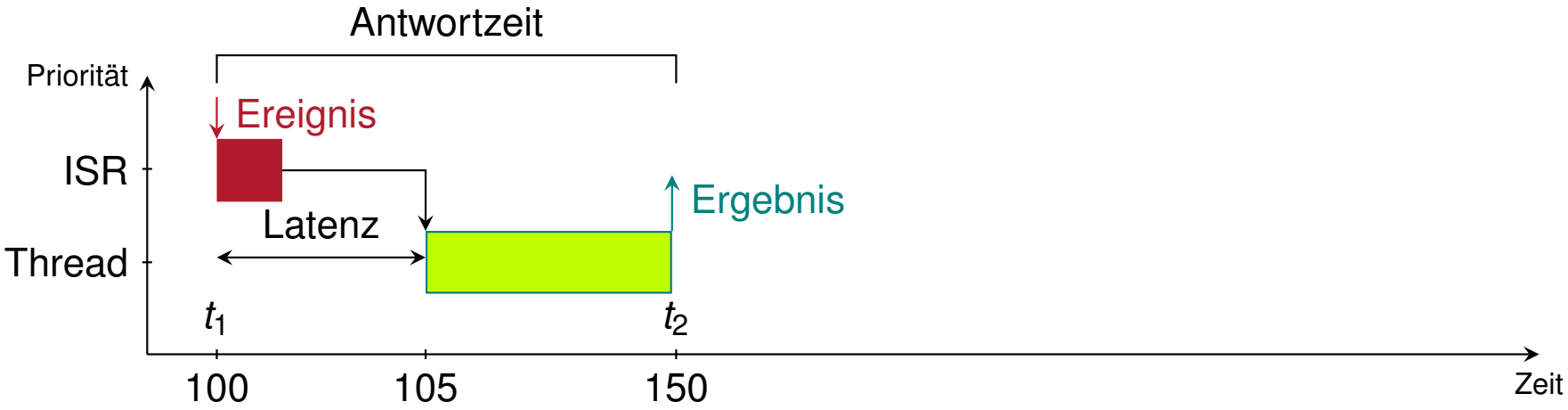
Hardware

Analyse der Schrittfunktion der Anwendung

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

Zeitmessung

Die Stoppuhr



Stoppuhr

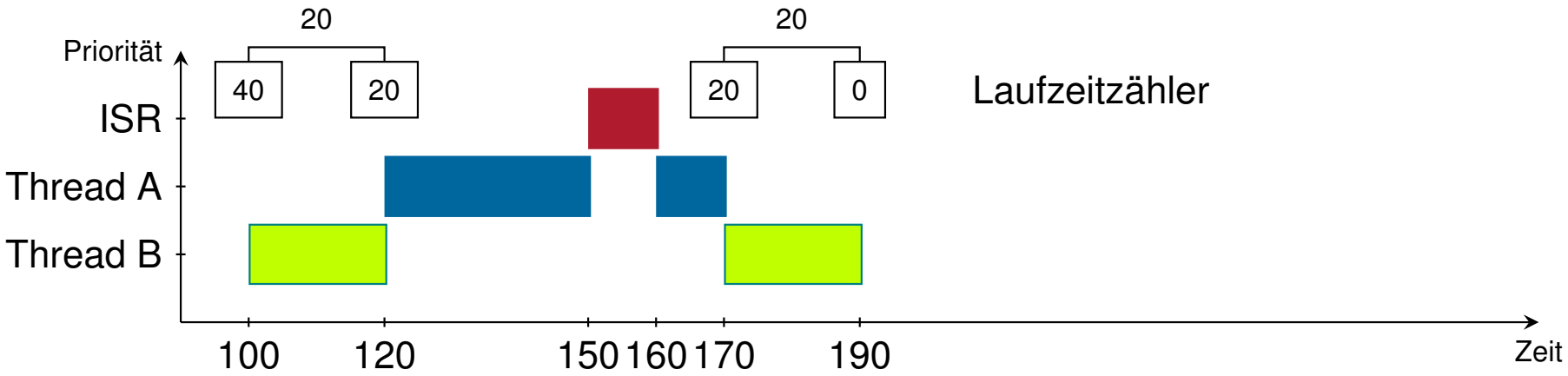
- Punkte auf der Zeitachse t_1 und $t_2 \rightsquigarrow$ Ereignis und Ergebnis
- Antwortzeit ist $\Delta t = t_2 - t_1$ (Beispiel: $150 - 100 = 50$ Zählerticks)

Umsetzung

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

Zeitmessung

Rechenzeitverbrauch



Rechenzeitsimulation

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

Umsetzung

- Funktion die aktiv t_{wcet} wartet $\rightsquigarrow$ Schleife auf Zählerwert
- HW-Zähler läuft bei Unterbrechungen weiter! $\rightsquigarrow$ 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 (WCET[1]) Simulation nutzen wir den *Abfragebetrieb*.

[1] Worst 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
|  |-- 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 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 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 μs 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.
  $\rightarrow$ Jeder Thread besitzt einen eigenen Stack!

Hinweis

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

$\rightarrow$ `ezs_counter_resolution_us()`, `ezs_counter_get()` in `drivers/include/ezs_counter.h`

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#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

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

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#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)

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

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#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

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

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#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)
```

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

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#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)
```

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

Wie behandle ich einen Interrupt?

Anmeldung von ISR und DSR[2]

```
#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

[2] 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
   - 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:

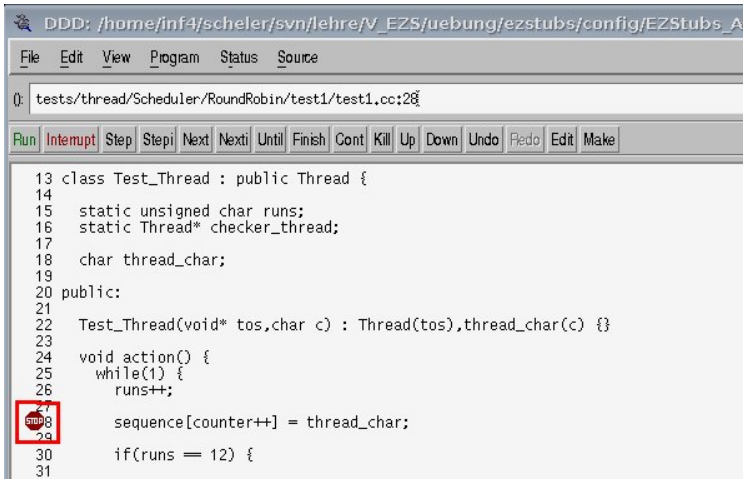
```
(gdb) cont
```

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!

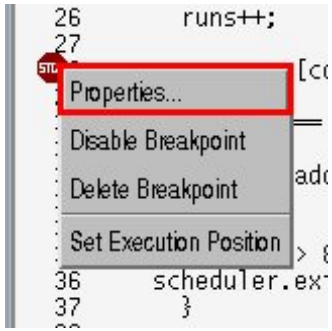
Bedingungen für Breakpoints

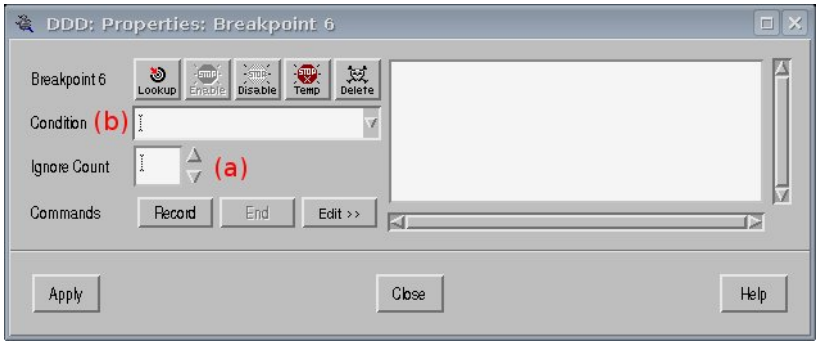
Bevor das Programm an einem Breakpoint tatsächlich angehalten wird, kann man ...

(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

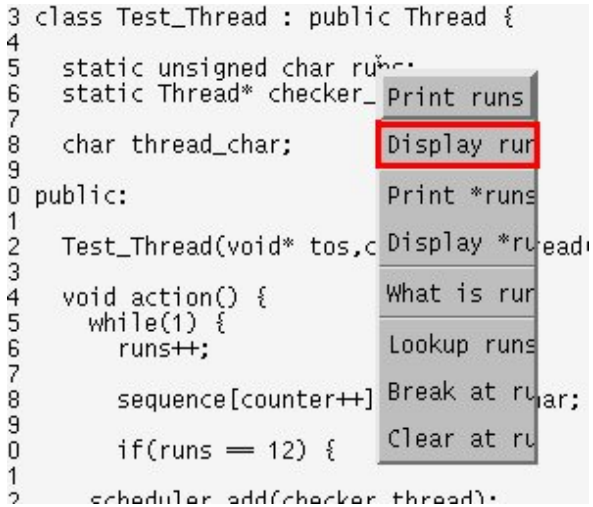
- Watchpoints

```
(gdb) watch xyz
```

  - Programm anhalten wenn Variable `xyz` ihren Wert ändert

Werte von Variablen beobachten

- globale Variablen und Objekte

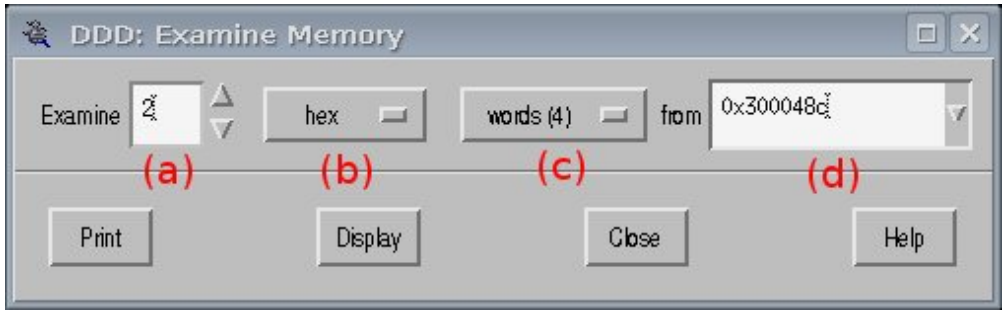



```
(gdb) # Achtung: vollqual. Name!
(gdb) graph display Test_Thread::runs
```

- 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
     ```

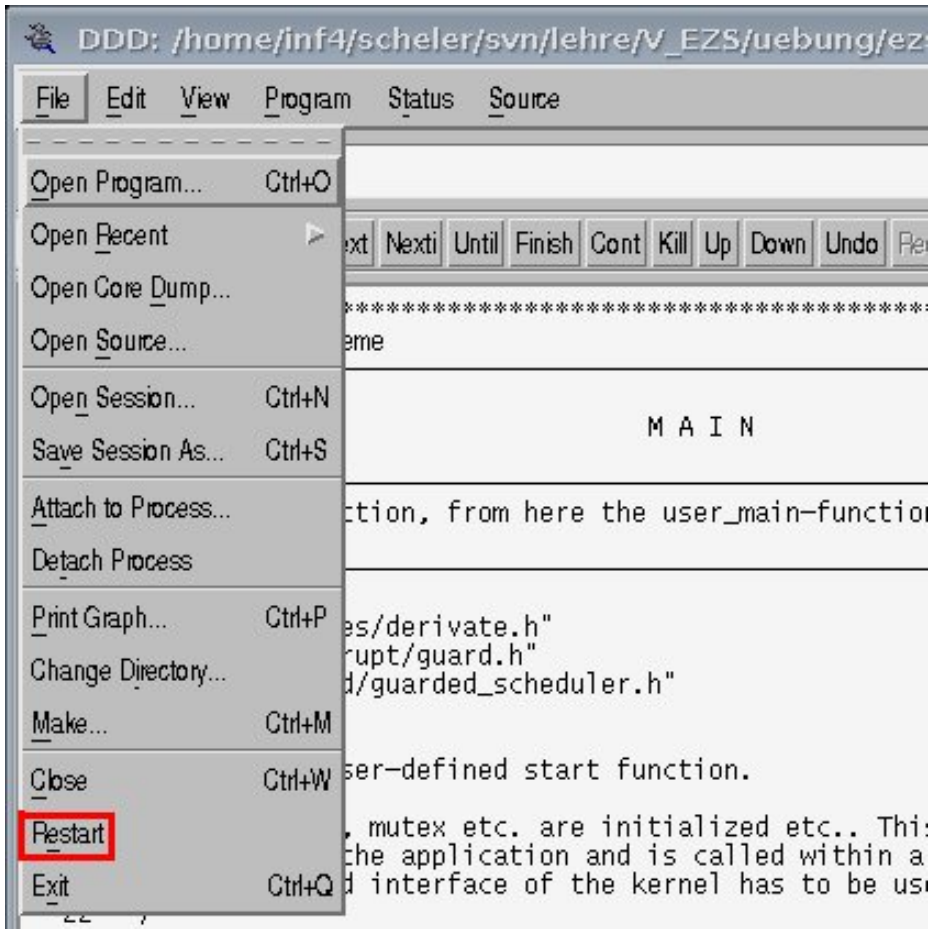
   - DDD wird das bemerken und folgender Dialog erscheint

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

Debugger neu starten

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



Fragen?