Echtzeitsysteme

Übungen zur Vorlesung

Messung von Ausführungszeiten & Antwortzeiten

Simon Schuster **Peter Wägemann**

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

26.10.2018

Übersicht

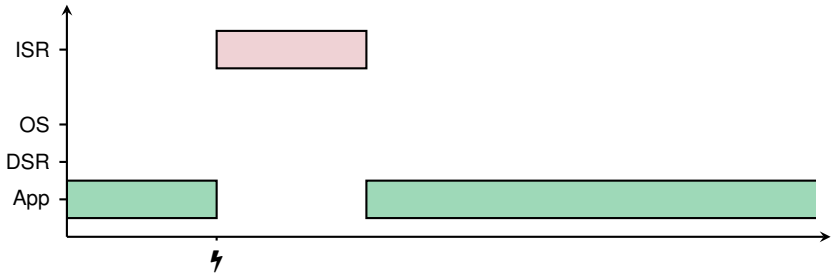
Bekannte Probleme & beliebte Fehler

Vorsicht eingebettete Hardware!

- Problem: Debugger reagiert nicht, Board lässt sich nicht flashen
  ☞ USB-Kabel abstecken & anstecken
- Falls immer noch keine Reaktion
  ☞ `make unbrick`, Anweisungen des Befehls folgen
- Keine leuchtenden LEDs
  ☞ anderes Board probieren
- Gehäuse für EZS-Board verwenden
- Vorsicht bei wackeligen Jumper-Kabeln
- `make gdb` weniger fragil als gdb-Dashboard (`make debug`)
- Vorsicht bei `printf`-Debugging
- Vorsicht `ezs_printf` macht gepufferte Ausgabe
- Verwendung von Fließkomma-Arithmetik und Typecasts ...

Übersicht

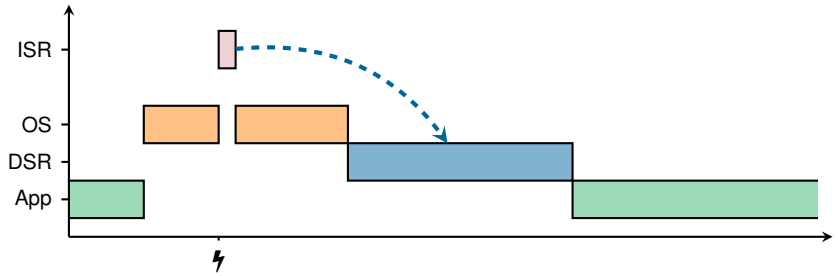
Unterbrechungen, Prologe, Epiloge



Interrupt-Service-Routinen-Ausführung

- Unverzüglich, *asynchron*, durch Hardware ausgelöst
  ⇝ auch innerhalb von Kernelfunktionen!
- Innerhalb ISR *keine Systemaufrufe* erlaubt!
  ⇒ Anmelden einer Deferrable Service Routine (DSR)

Unterbrechungen, Prologe, Epiloge



Deferrable-Service-Routinen-Ausführung

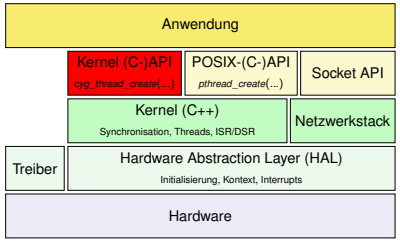
- *Synchron* zum Scheduler
- Falls Scheduler nicht verriegelt: *Unverzüglich* nach ISR
- sonst: Beim *Verlassen* des Kerns

Synonym: *Prolog-Epilog-Schema* bzw. *top/bottom half*

eCos-Systemarchitektur

Kernel – Interruptbehandlung

- Interrupt Service Routine (ISR)
  - Unverzügliche Ausführung
  - Asynchron
  - Kann DSR anfordern
- Deferred Service Routine (DSR)
  - Verzögerte Ausführung (beim Verlassen des Kernels)
  - Synchron



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
- ⇝ Hardwarehandbuch

[1] http://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] http://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] http://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
);
```

- Funktionszeiger auf *ISR-Implementierung*

Signatur:

```
cyg_uint32 (*)(cyg_vector_t, cyg_addrword_t)
```

[1] http://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
);
```

- Funktionszeiger auf *DSR-Implementierung*

Signatur:

```
cyg_uint32 (*)(cyg_vector_t, cyg_ucount32, cyg_addrword_t)
```

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

ISR-Implementierungsskelett

Beispiel einer minimalen ISR

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

1. Beliebiger ISR-Code
2. Bestätigung der Interruptbehandlung
   Wozu ist das gut?
3. Anforderung einer DSR
   oder
4. Rückkehr ohne DSR

ISR-Implementierungsskelett

Beispiel einer minimalen ISR

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

1. Beliebiger ISR-Code
2. Bestätigung der Interruptbehandlung
   Wozu ist das gut?
3. Anforderung einer DSR
   oder
4. Rückkehr ohne DSR

Schu, PW EZS (26.10.2018) 8/37
1 Interruptbehandlung – 1.2 eCos-Unterbrechungsbehandlung

ISR-Implementierungsskelett

Beispiel einer minimalen ISR

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

1. Beliebiger ISR-Code
2. Bestätigung der Interruptbehandlung
   Wozu ist das gut?
3. Anforderung einer DSR
   oder
4. Rückkehr ohne DSR

Schu, PW EZS (26.10.2018) 8/37
1 Interruptbehandlung – 1.2 eCos-Unterbrechungsbehandlung

ISR-Implementierungsskelett

Beispiel einer minimalen ISR

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

1. Beliebiger ISR-Code
2. Bestätigung der Interruptbehandlung
   Wozu ist das gut?
3. Anforderung einer DSR
   oder
4. Rückkehr ohne DSR

DSR-Implementierungskelett

Beispiel einer minimalen DSR

```
void dsr_function(
  cyg_vector_t vector,
  cyg_ucount32 count,
  cyg_addrword_t data)
{
...
}
```

1. Anzahl der ISRs, die diese DSR anforderten
   $\rightsquigarrow$ normalerweise 1
2. Ausführung *synchron* zum Scheduler
   Was bedeutet das?

DSR-Implementierungskelett

Beispiel einer minimalen DSR

```
void dsr_function(
  cyg_vector_t vector,
  cyg_ucount32 count,
  cyg_addrword_t data)
{
...
}
```

1. Anzahl der ISRs, die diese DSR anforderten
   $\rightsquigarrow$ normalerweise 1
2. Ausführung *synchron* zum Scheduler
   Was bedeutet das?

Übersicht

Beispiel: Ausführungszeit

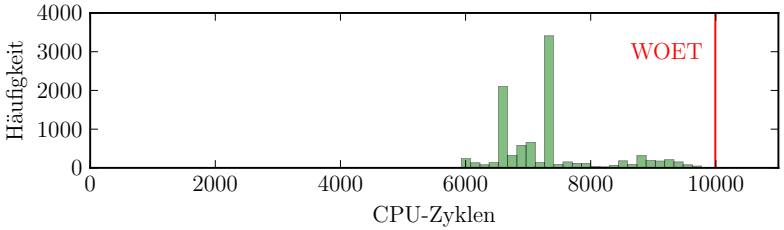
```
1  uint64_t count_positive(uint64_t* array, size_t size){
2    uint64_t count = 0;
3    for(int i = 0; i < size; i++){
4      if (array[i] > 0){
5        count++;
6      }
7    }
8    return count;
9  }
```

Benchmark-Programm

- Iteration über Array mit variabler Größe
- Zählen der positiven Zahlen in Array

☞ Wie verhält sich die **Ausführungszeit bei 10.000 Messungen**?

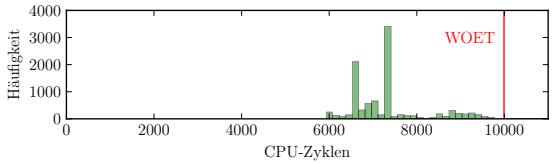
Histogramm: Ausführungszeiten



- Maximum der beobachteten Ausführungszeiten
  (engl. worst-observed execution time **WOET**)
- 10.000 Ausführungen der Funktion `count_positive()`
- Maximum: 9992 CPU-Zyklen
- Hohe Streuung der Ausführungszeiten

Warum?

Setup der Messung: Applikation



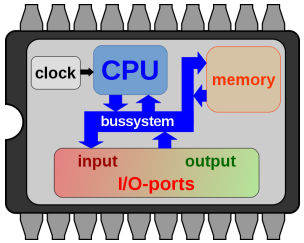

```
1  void main(void){
2    uint64_t array[ARRAY_SIZE];
3    // alle Eingabedaten mit 0 initialisiert!
4    memset(array, 0, sizeof(array));
5    start    = get_time();
6    positive = count_positive(array, ARRAY_SIZE);
7    stop     = get_time();
8    printf("%lu", stop-start);
9  }
```

- Ausführungszeit unabhängig von unterschiedlichen Eingabedaten
  - Feste Länge
  - Immer mit `0` initialisiert

☞ **kein Einfluss** der Eingabedaten

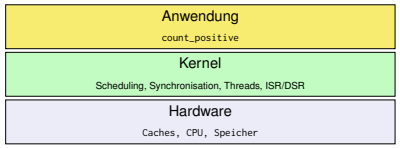
Woher kommen dann die Schwankungen?

Setup der Messung: Hardware



- CPU: Intel Core i7
- Takt: $\leq$ 3.3 GHz
- Cache: 4 MB *Smart* Cache
- Universalbetriebssystem
- Aufgaben-System: zusätzliche Last (`stress --cpu 8 --io 8 --hdd 8 --vm 8`)

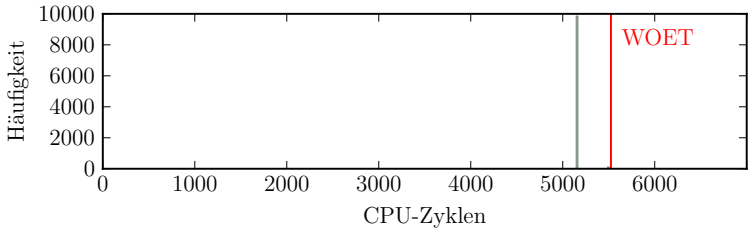
Einflüsse der Hardware & des Betriebssystems



- Pipelining: spekulative Ausführung (engl. Branch Prediction)
- Takt: dynamische Änderung möglich (engl. Dynamic Frequency Voltage Scaling, DVFS)
- Cache: heuristische Strategien
- Scheduling
  - Keine Priorisierung von Aufgabe: Completely Fair Scheduler
  - Timer-Interrupts möglich
    ☞ **Verdrängung** möglich

Wie verhält sich die Messung auf dem **EZS-Board**?

`count_positive` auf EZS-Board

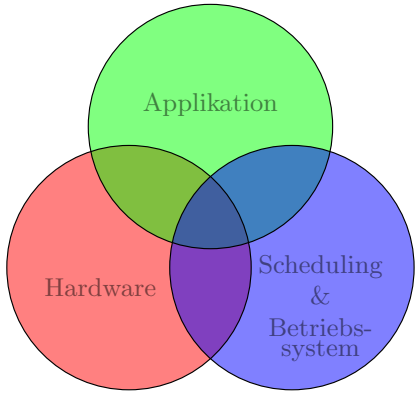


- Geringere Komplexität ⇝ *weniger* Streuung
- Trotzdem Unterschiede
  - Hardware
    - 3-stufige Pipeline
    - Spekulative Ausführung: ART Accelerator™
  - Betriebssystem: Schwankungen der Ausführungszeit trotz eines Threads?

Wissen über **Hard- & Software** essenziell für Echtzeitsysteme

Was beeinflusst die Ausführungszeit?

Zusammenfassung



1. **Applikation**: Eingabedaten, ...
2. **Hardware**: Caches, Pipelining, ...
3. **Scheduling**: Höherpriore Aufgaben, Interrupts, Overheads, ...

☞ (gegenseitige) Einflüsse **kaum vermeidbar**, aber reduzierbar

Übersicht

Zähler in Mikrocontrollern

Zähler (Counter) zählen Hardware-Ereignisse

- Externer Drehgeber (Radumdrehung)
- Interner Prozessortakt (hohe Auflösung)
- Externer Quarz (Real-Time Clock)

Äquidistante Ereignisse ermöglichen einen *Zeitgeber (Timer)* für

- Periodische Aktivierung
- *Messen von Zeitabständen*
- *Kontrolliertes Verbrennen von Prozessorzeit*

Zähler-Betriebsmodi

Zähler bzw. Zeitgeber bieten zwei Betriebsmodi:

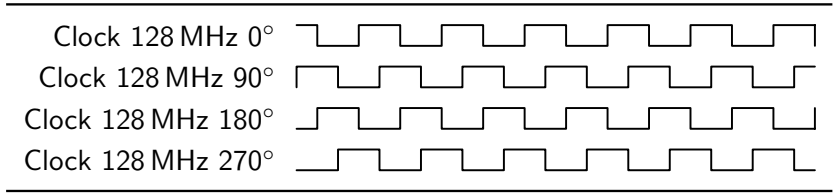
Abfragebetrieb (Polling) Aktives Auslesen des Zählers

⇝ bis Erreichen eines vorgegebenen Wertes

Unterbrecherbetrieb (Interrupt) Zähler unterbricht System

⇝ Erreichen eines konfigurierten Zählerstandes.

Probleme von Taktgebern



- Clock-Drift
  - Abweichtung der internen Uhr von Realzeit
  - Temperaturabhängiger Phasenunterschied
  - Äußerst kritisch in verteilten Echtzeitsystemen
  - Quarz: $\approx 10^{-6}$ sec/sec = 1 sec in 11,6 Tagen

Lösung

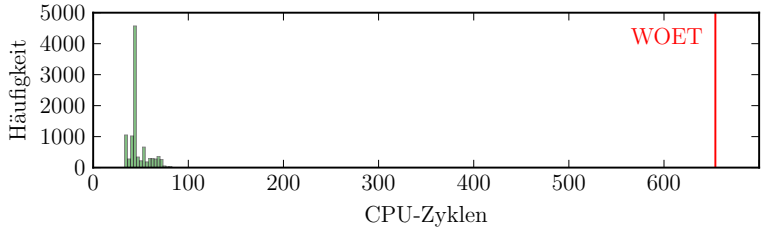
- Messung mit externer, hochauflösender Uhr

☞ *Oszilloskop* ⇝ Übungsaufgabe

Overhead der Messung

Wer misst misst Mist!

```
1 start = get_ticks(); // ticks ~ processor cycles/timer values
2 // nothing executed here
3 stop  = get_ticks();
4 printf("%lu", stop-start);
```



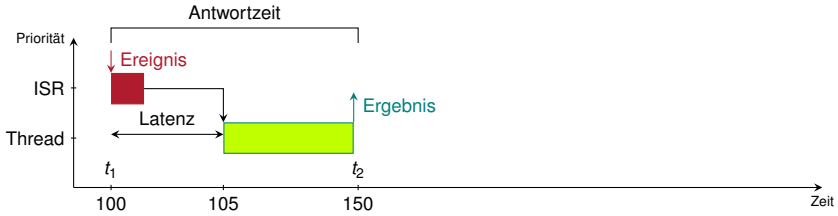
- Overhead-Bestimmung für Intel-CPU; EZS-Board ⇝ *erweiterte Übungen*
- Overhead-Bestimmung vernachlässigt
  - Instruktionen bis zum 1. Abfragen des Timer-Registers
  - Instruktionen nach dem 2. Abfragen des Timer-Registers
  - *Weitere Störfaktoren?*

Einschub: Zusammenfassung von Ergebnissen

- *How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results* [1]
- Wichtige Regeln
  1. **Für normalisierte Werte nicht das arithmetische Mittel verwenden**
  2. **Für normalisierte Werte das geometrische Mittel verwenden**
  3. **Für Rohdaten (mit Einheiten) das arithmetische Mittel verwenden**
- Arithmetisches Mittel: $x_{arith} = \frac{1}{n} \sum_{i=1}^{N} x_i$
- Geometrisches Mittel: $x_{geom} = \sqrt[n]{\prod_{i=1}^{N} x_i}$
- Für Messungen in Echtzeitsystemen
  4. Alle Standardabweichungen müssen **weniger als 1 %** betragen [2]

Übersicht

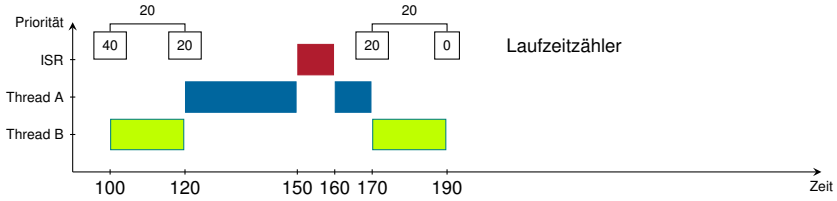
Ausführungszeit & Antwortzeit



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)

Messung der ausgeführten Zeit



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

Übersicht

libEZS-Überblick

Plattformunabhängige Hilfsfunktionen

- Timer-Zugriff (Zeitmessung)
- DAC-Zugriff
- GPIO-Zugriff
- ...

```
<aufgabe>
|-- CMakeLists.txt
|-- app.c
|-- ecos
`-- libEZS
  |-- include
  | |-- ezs_dac.h
  | |-- ezs_gpio.h
  | `-- ezs_stopwatch.h
  |-- src
  | `-- ezs_stopwatch.c
  `-- drivers
    `- stm32
       |-- ezs_dac.c
       |-- ezs_counter.c
       `-- ezs_gpio.c
```

Die *libEZS* wird im Laufe der Übungen erweitert

Zeitmessung in *ezs_stopwatch.c/.h*

Die Zeitmessung wird durch zwei Funktionen implementiert:

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

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

Hinweis

`ezs_counter_get ()` in `drivers/include/ezs_counter.h`

Hinweis

Auflösung der Zählers in Bruchteilen von Nanosekunden:
→ `ezs_counter_get_resolution ()`

WCET-Simulator in *ezs_stopwatch.c/.h*

Zu implementieren:

```
void ezs_lose_time(cyg_uint32 wcet, cyg_uint8 percentage);
```

- Parameter:
  1. Gewünschte WCET in *Ticks*
  2. *Maximum* des zufällig zu subtrahierenden *WCET-Anteils*
- Implementierung muss internen Zähler verwalten
  - ⇝ Bei welchen Änderung des Systemzählers anpassen?
  - ⇝ Welche Auflösung ist erreichbar
  - Jeder Thread besitzt einen eigenen Stack!
  - Thread-übergreifende Messungen möglich
- *Abfragebetrieb*

Hinweis

Auflösung des Zählers in Pikosekunden:
→ `ezs_counter_resolution_ps()`
Kann das problematisch sein?

Rechnen mit Timer-Auflösungen

```
1 // CPU_SPEED is 84MHz
2 // RCC_CLOCK is CPU_SPEED / 2 = 42MHz
3 // timer increment frequency is RCC_CLOCK / (PRESCALER+1)
4 // PRESCALE is configured as 1
5 // counter will overflow after ??? sec
```

- **Überläufe**: *Dauer der Messung* beachten
  - 32-Bit Timer in auf EZS-Board
- **Rundungsfehler**: *Auflösungen* beachten
  - Auflösung in „Mikro-Sekunden-Ticks“
  - $1000000/42 = 23809.5238...$ vs. $\texttt{floor}(1000000/42)$
  - $\approx 15\,\mu s$ Rundungsfehler bei 1 s Zeitmessung

Lösung

```
1 cyg_resolution_t ezs_counter_get_resolution(void);
```

- $\frac{\text{Dividend}}{\text{Divisor}}$ beinhaltet die Auflösung in Nanosekunden $\rightsquigarrow$ siehe `make doc`

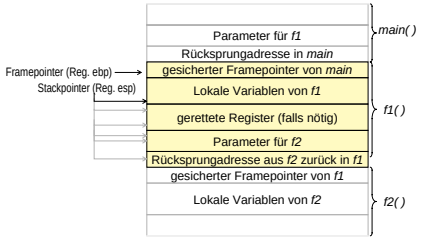
Übersicht

Wiederholungen notwendig?

- Stack-Aufbau bei Funktionsaufrufen
  - Stack-Tiefe
  - Backtrace
- Komponenten der Toolchain
  - Präprozessor
  - Compiler
  - Flasher
  - ...

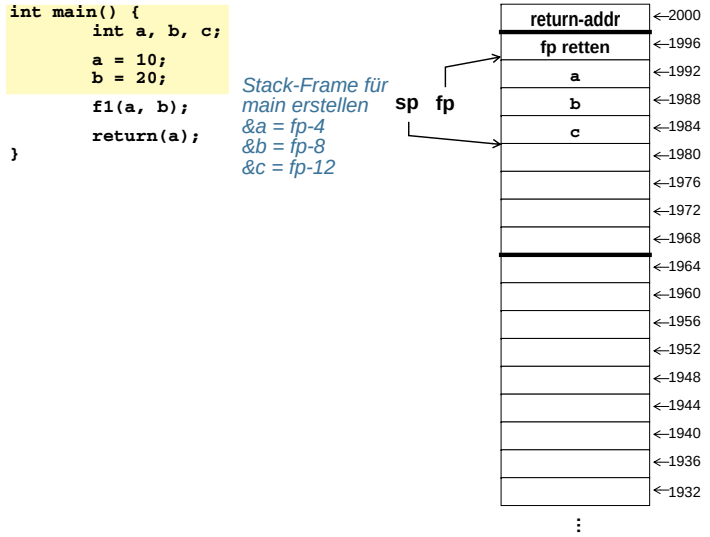
Wiederholung: Stack-Aufbau

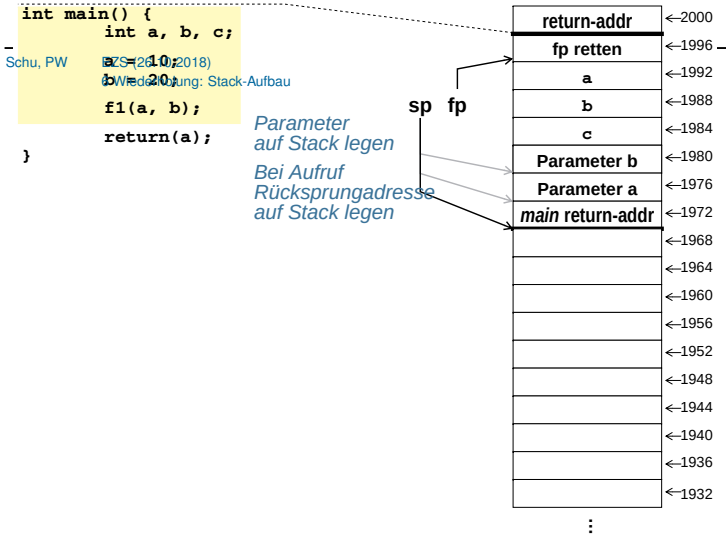
```
1  int main() {
2    int a, b, c;
3
4    a = 10;
5    b = 20;
6
7    f1(a, b);
8
9    return (a);
10 }
```

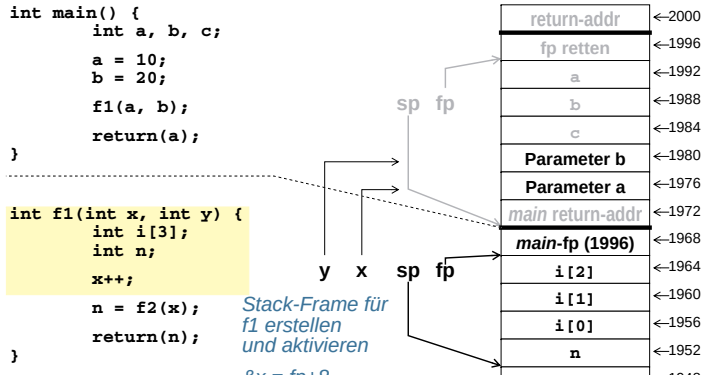


- Stack-Frame zur Verwaltung des Stacks
  - Lokale Variablen
  - Funktionsparameter
  - Rücksprungadressen

Ablauf Funktionsaufruf







Literatur

[1] Philip J Fleming and John J Wallace.
How not to lie with statistics: the correct way to summarize benchmark results.
Communications of the ACM, 29(3):218–221, 1986.

[2] Gernot Heiser.
Systems benchmarking crimes, 2016.
http://gernot-heiser.org/benchmarking-crimes.html.