Echtzeitsysteme

Übungen zur Vorlesung

Entwicklungsumgebung

Simon Schuster **Peter Wägemann**

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

Sommersemester 2022

Übersicht

1. Einführung in eCos

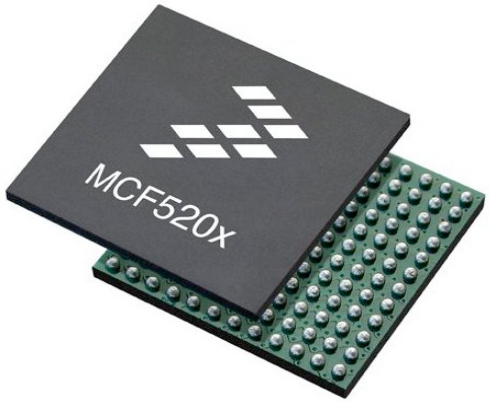
2. Entwicklungsumgebung
   - Werkzeugkette und CMake
   - Blackmagic
   - Pulsweitenmodulation
   - Tiefpassfilter
   - Oszilloskop-Bedienung

3. Debuggen mit GDB

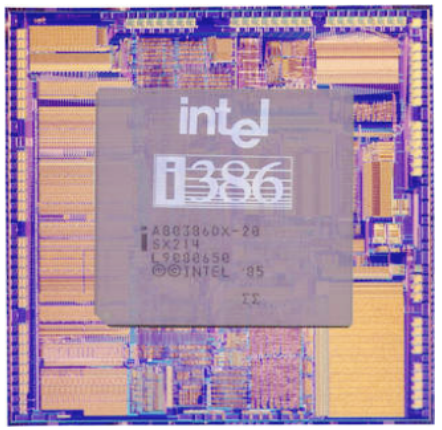
Gliederung

Prozessorvielfalt in der Echtzeitwelt

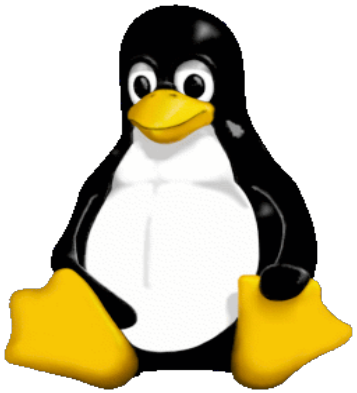
freescale™ semiconductor



ARM. Cortex™ Intelligent Processors by ARM®



Noch mehr Betriebssysteme



eCos

Embedded **C**onfigurable **O**perating **S**ystem

eCos is an embedded, highly configurable, open-source, royalty-free, real-time operating system.

- Ursprünglich von der Fa. Cygnus Solutions entwickelt (1997)
- Primäres Entwurfsziel:
  - „deeply embedded systems“
  - „high-volume application“
  - „consumer electronics, telecommunications, automotive, . . . “
- Zusammenarbeit mit Redhat (1999)
- Seit 2002 quelloffen (GPL)

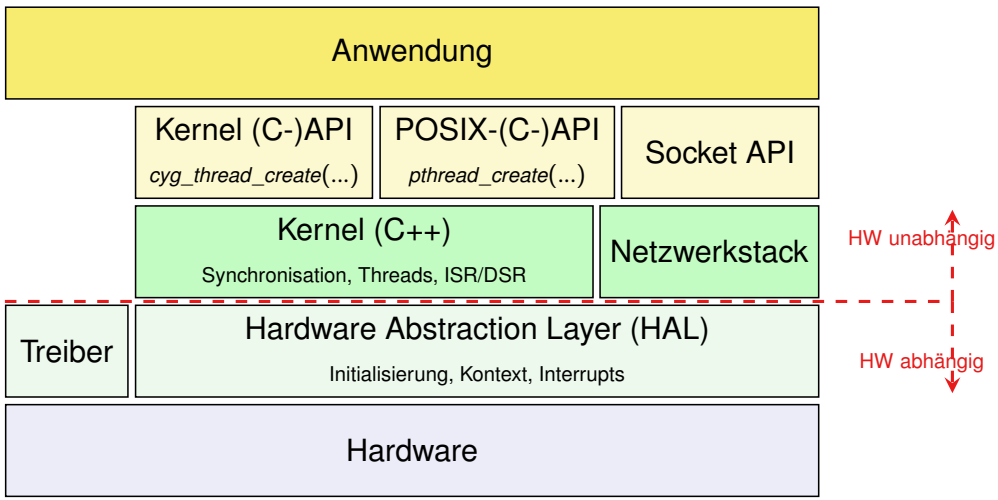
☞ `http://ecos.sourceware.org`

Unterstützte Plattformen

http://www.ecoscentric.com/ecos/examples.shtml

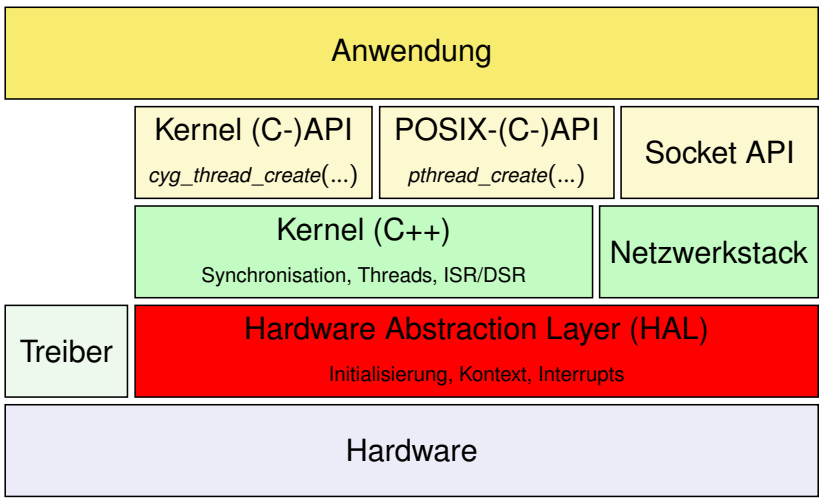
- Fujitsu SPARClite
- Matsushita MN10300
- Motorola PowerPC
- Advanced RISC Machines (ARM)
- Toshiba TX39
- Infineon TriCore
- Hitachi SH3
- NEC VR4300
- MB8683X
- ☞ *ARM Cortex*
- ☞ *Intel x86*
- ...

eCos-Systemarchitektur



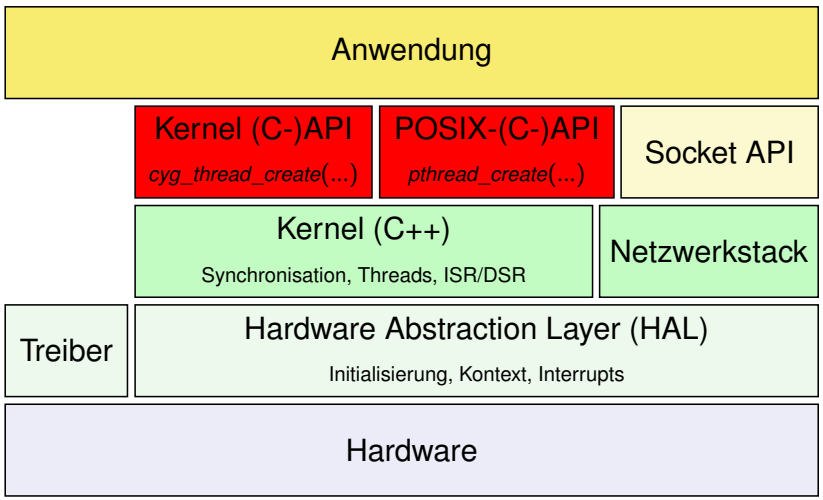
eCos-Systemarchitektur

- Abstrahiert CPU- und plattformspezifische Eigenschaften
  - Kontextwechsel
  - Interruptverwaltung
  - CPU-Erkennung, Startup
  - Zeitgeber, I/O-Registerzugriffe



eCos-Systemarchitektur

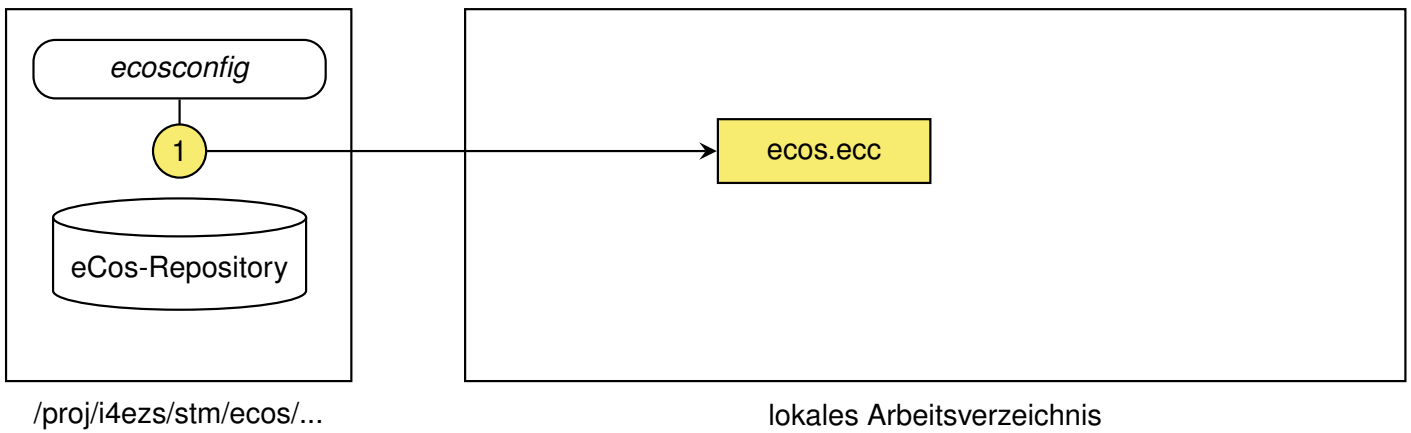
- Kernel API
  - vollständige C-Schnittstelle
  - siehe Dokumentation[1]
- (Optionale) POSIX-Kompatibilitätsschicht
  - Scheduling-Konfiguration, *pthread_∗*
  - Timer, Semaphore, Message Queues, Signale, ...



[1] http://ecos.sourceware.org/docs-2.0/ref/ecos-ref.html

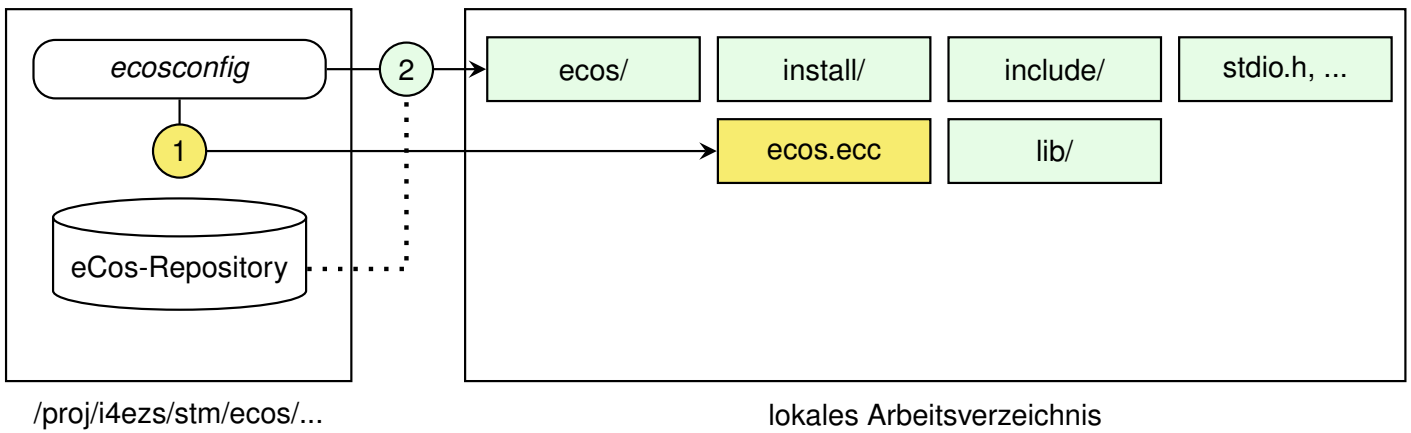
eCos-Entwicklungszyklus

1 Erstellen einer Konfiguration (configtool/ecosconfig)



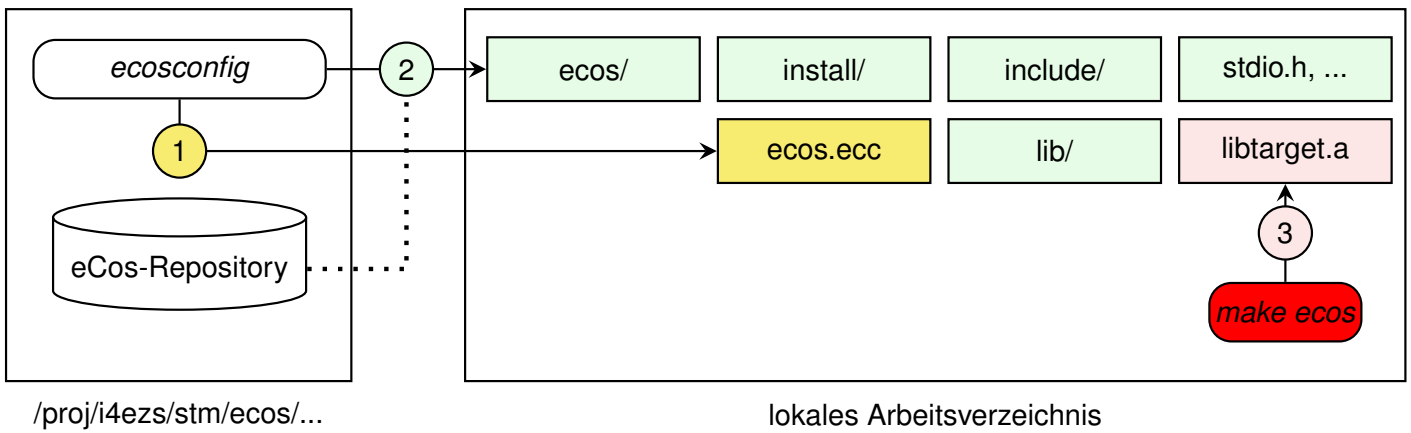
eCos-Entwicklungszyklus

1. Erstellen einer Konfiguration (configtool/ecosconfig)
2. Kopieren ausgewählter Komponenten (configtool/ecosconfig)



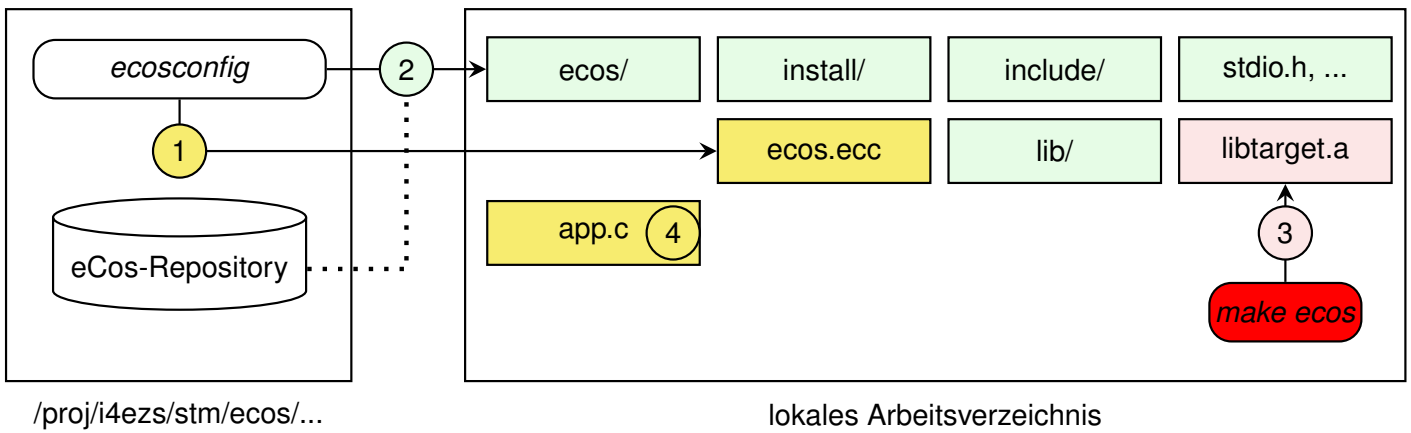
eCos-Entwicklungszyklus

1. Erstellen einer Konfiguration (configtool/ecosconfig)
2. Kopieren ausgewählter Komponenten (configtool/ecosconfig)
3. Erstellen einer Betriebssystembibliothek



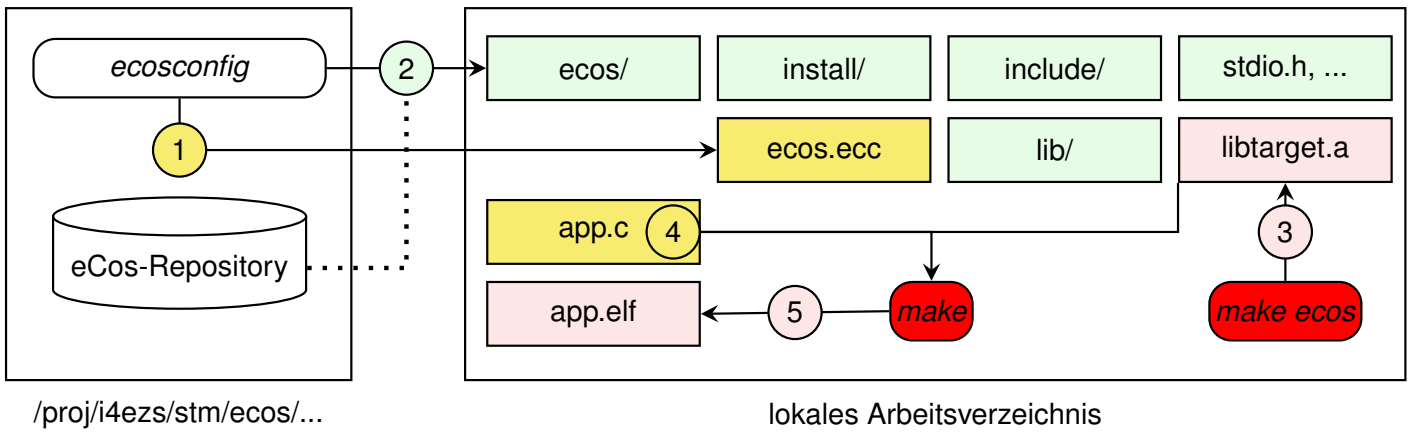
eCos-Entwicklungszyklus

1. Erstellen einer Konfiguration (configtool/ecosconfig)
2. Kopieren ausgewählter Komponenten (configtool/ecosconfig)
3. Erstellen einer Betriebssystembibliothek
4. Entwicklung der eigentlichen Anwendung



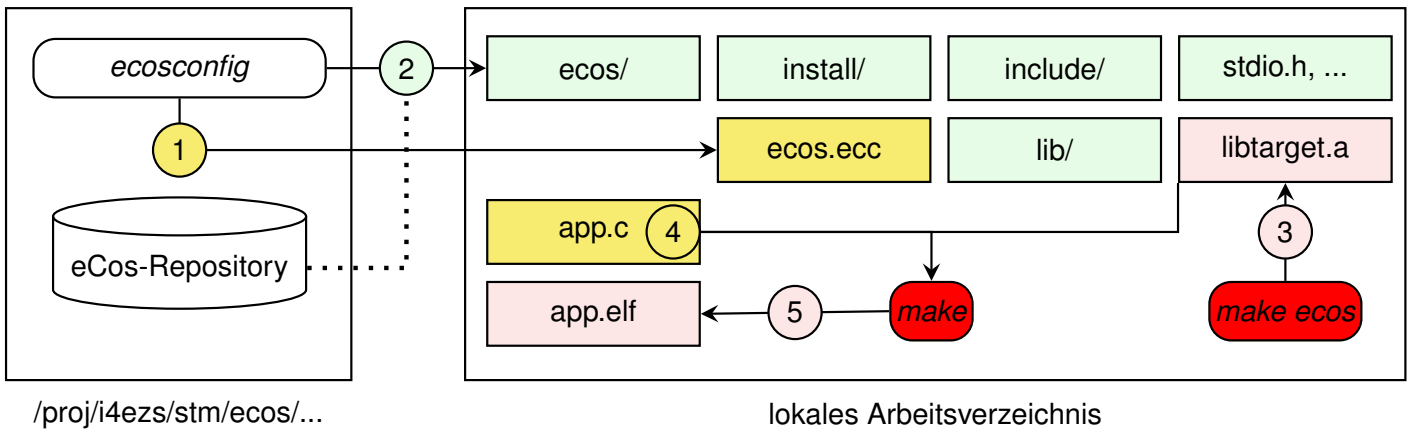
eCos-Entwicklungszyklus

1. Erstellen einer Konfiguration (configtool/ecosconfig)
2. Kopieren ausgewählter Komponenten (configtool/ecosconfig)
3. Erstellen einer Betriebssystembibliothek
4. Entwicklung der eigentlichen Anwendung
5. Kompilieren des Gesamtsystems



eCos-Entwicklungszyklus

1. Erstellen einer Konfiguration (configtool/ecosconfig)
2. Kopieren ausgewählter Komponenten (configtool/ecosconfig)
3. Erstellen einer Betriebssystembibliothek
4. Entwicklung der eigentlichen Anwendung
5. Kompilieren des Gesamtsystems



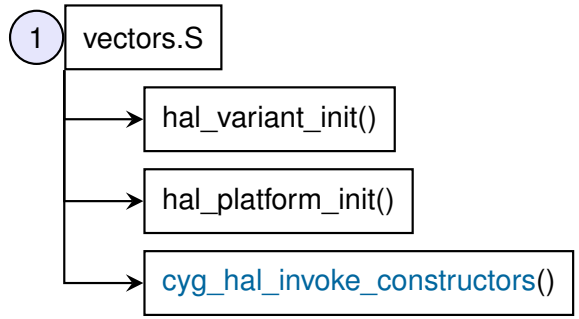
Wichtig!

Für jede Übung wird eine Konfiguration vorgegeben (Schritte 1–3)

eCos-Systemstart

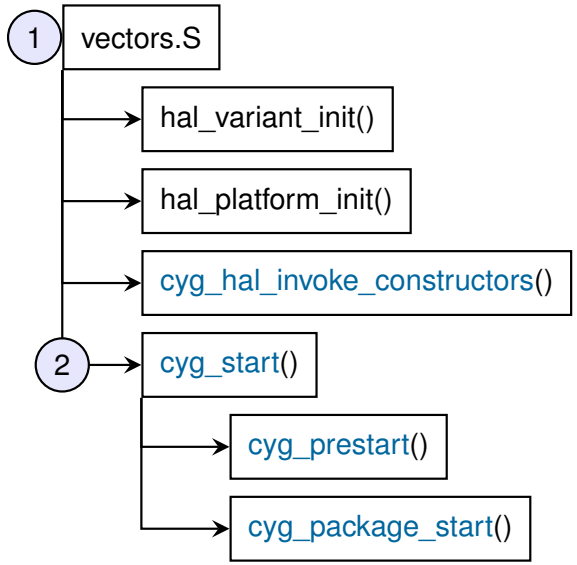
1 vectors.S

- Hardwareinitialisierung
- Globale Konstruktoren



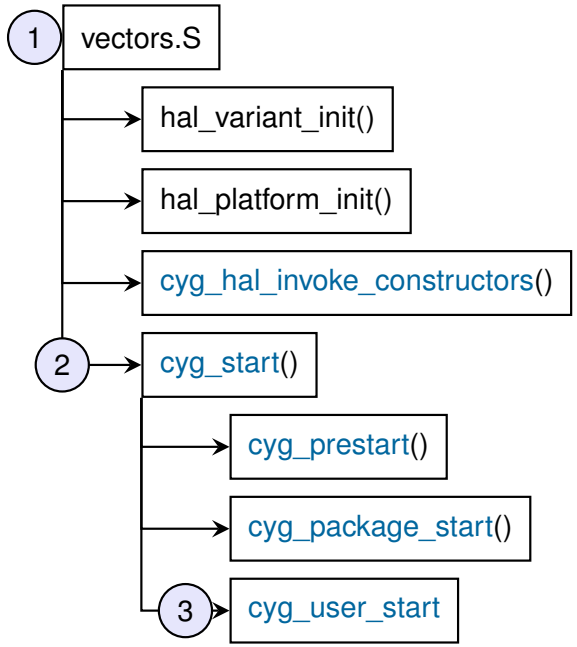
eCos-Systemstart

1. vectors.S
   - Hardwareinitialisierung
   - Globale Konstruktoren
2. `cyg_start()`:
   - Hardwareunabhängige Vorbereitungen



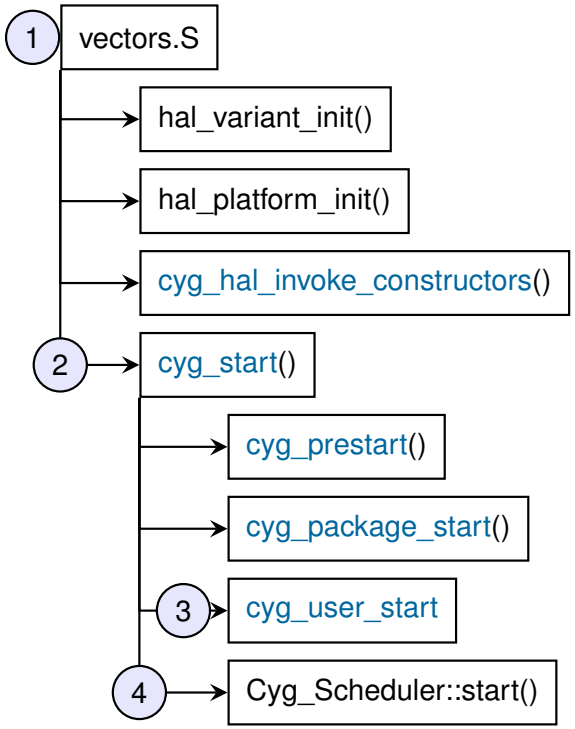
eCos-Systemstart

1. vectors.S
   - Hardwareinitialisierung
   - Globale Konstruktoren
2. `cyg_start()`:
   - Hardwareunabhängige Vorbereitungen
3. `cyg_user_start`:
   - Einsprungpunkt für Anwendungscode!
   - Erzeugen von Threads



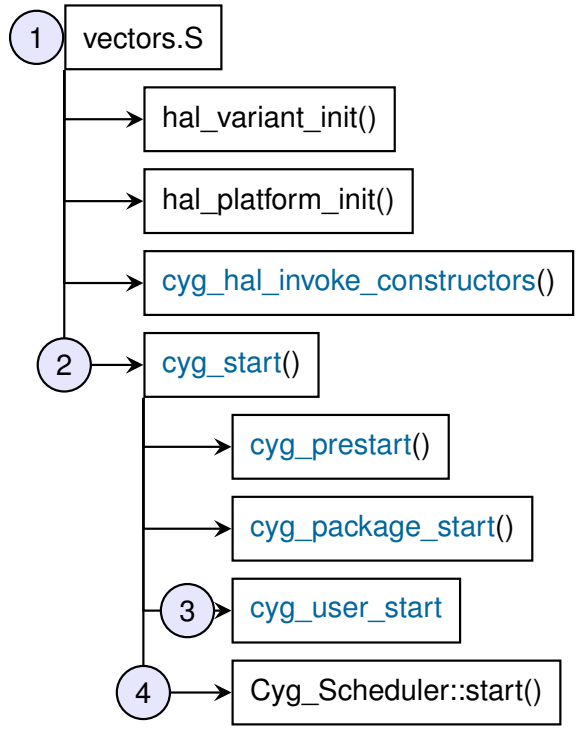
eCos-Systemstart

1. vectors.S
   - Hardwareinitialisierung
   - Globale Konstruktoren
2. `cyg_start()`:
   - Hardwareunabhängige Vorbereitungen
3. `cyg_user_start`:
   - Einsprungpunkt für Anwendungscode!
   - Erzeugen von Threads
4. Starten des Schedulers



eCos-Systemstart

1. vectors.S
   - Hardwareinitialisierung
   - Globale Konstruktoren
2. `cyg_start()`:
   - Hardwareunabhängige Vorbereitungen
3. `cyg_user_start`:
   - Einsprungpunkt für Anwendungscode!
   - Erzeugen von Threads
4. Starten des Schedulers



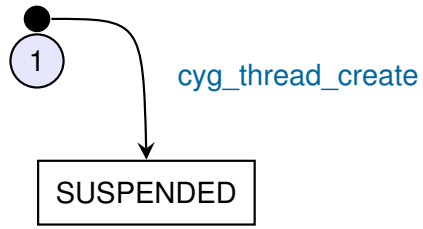
Wichtig!

In allen Übungsaufgaben muss man `cyg_user_start()` implementieren und dort alle Threads anlegen. *Die Funktion muss zurückkehren!*

eCos-Threads

Threadzustände und Übergänge

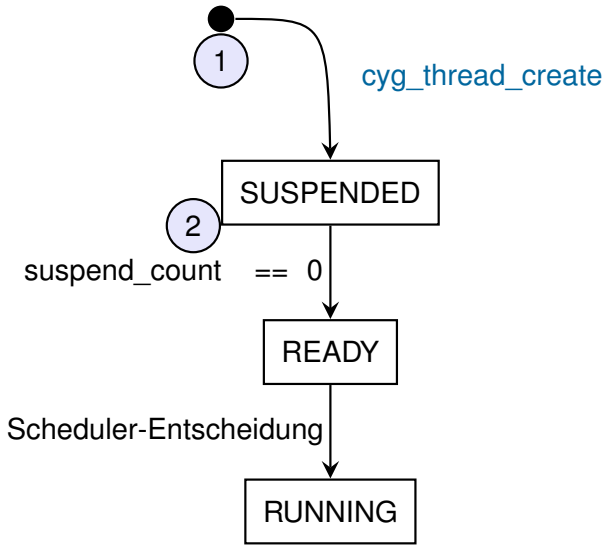
1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`



eCos-Threads

Threadzustände und Übergänge

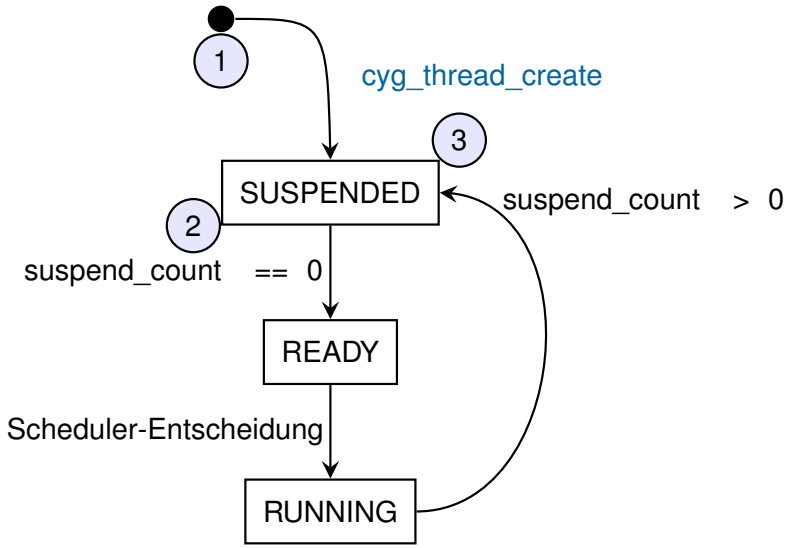
1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`
2. `cyg_thread_resume` aktiviert
   - `suspend_count--`



eCos-Threads

Threadzustände und Übergänge

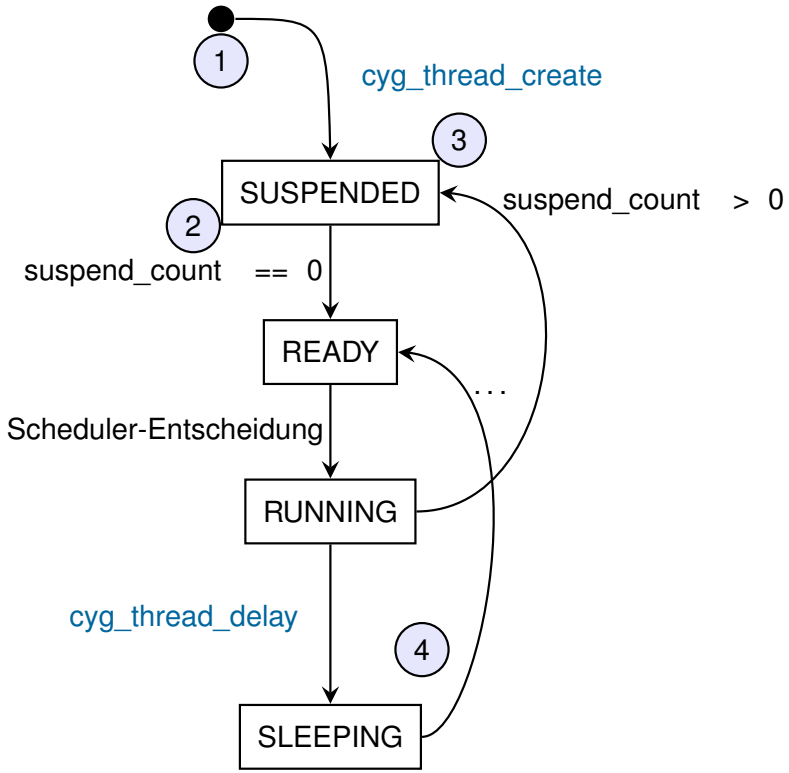
1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`
2. `cyg_thread_resume` aktiviert
   - `suspend_count--`
3. `cyg_thread_suspend` suspendiert
   - `suspend_count++`



eCos-Threads

Threadzustände und Übergänge

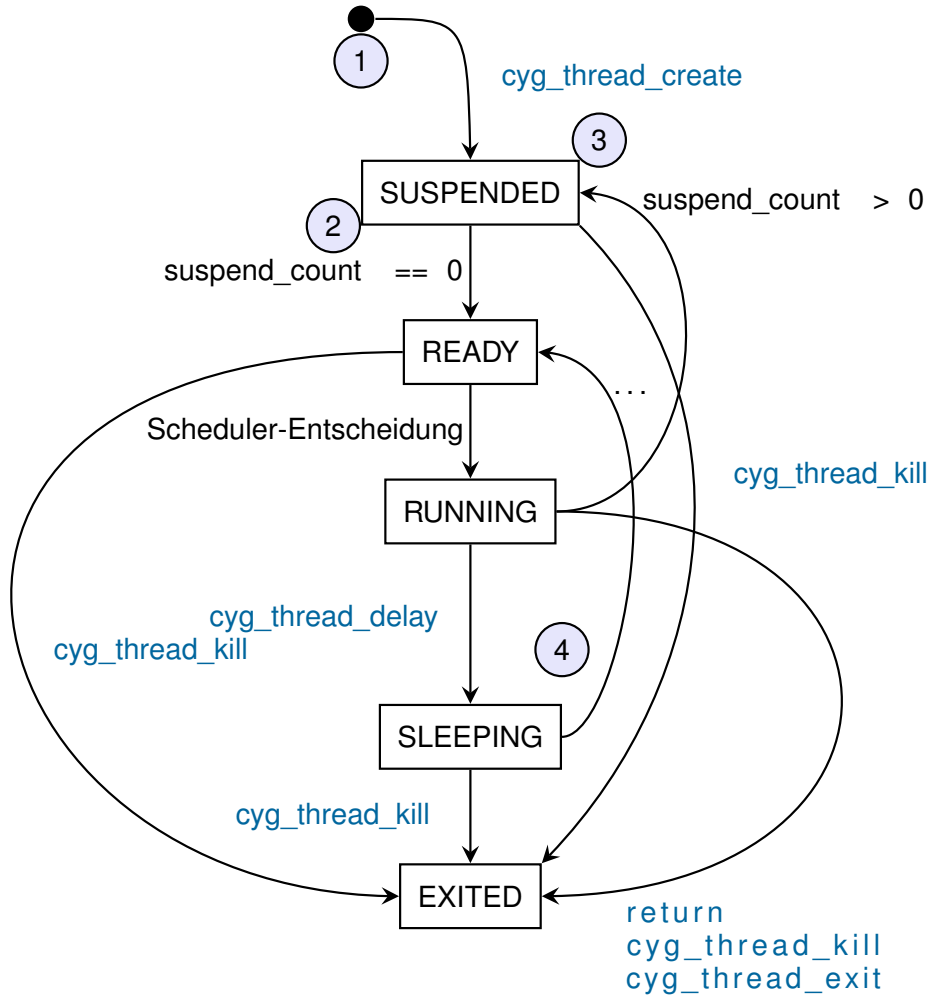
1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`
2. `cyg_thread_resume` aktiviert
   - `suspend_count--`
3. `cyg_thread_suspend` suspendiert
   - `suspend_count++`
4. delay, mutex, semaphore wait



eCos-Threads

Threadzustände und Übergänge

1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`
2. `cyg_thread_resume` aktiviert
   - `suspend_count--`
3. `cyg_thread_suspend` suspendiert
   - `suspend_count++`
4. delay, mutex, semaphore wait
5. Threadterminierung



Threaderzeugung in eCos

```
1  #include <cyg/kernel/kapi.h>
2  void cyg_thread_create
3  (
4      cyg_addrword_t sched_info,
5      cyg_thread_entry_t* entry,
6      cyg_addrword_t entry_data,
7      char* name,
8      void* stack_base,
9      cyg_ucount32 stack_size,
10     cyg_handle_t* handle,
11     cyg_thread* thread
12 );
```

- Einbinden der nötigen Headerdatei
- Anlegen eines neuen eCos-Threads

Threaderzeugung in eCos

```
1  #include <cyg/kernel/kapi.h>
2  void cyg_thread_create
3  (
4      cyg_addrword_t sched_info,
5      cyg_thread_entry_t* entry,
6      cyg_addrword_t entry_data,
7      char* name,
8      void* stack_base,
9      cyg_ucount32 stack_size,
10     cyg_handle_t* handle,
11     cyg_thread* thread
12 );
```

- faktisch: Threadpriorität
- 0
  - höchste Priorität
  - für Systemprozesse freilassen
- `CYG_THREAD_MIN_PRIORITY`
  - Idle-Thread

Threaderzeugung in eCos

```
1  #include <cyg/kernel/kapi.h>
2  void cyg_thread_create
3  (
4      cyg_addrword_t sched_info,
5      cyg_thread_entry_t* entry,
6      cyg_addrword_t entry_data,
7      char* name,
8      void* stack_base,
9      cyg_ucount32 stack_size,
10     cyg_handle_t* handle,
11     cyg_thread* thread
12 );
```

- Threadeinsprungpunkt
- Funktionszeiger
- Signatur:
  `void (*)(cyg_addrword_t)`

Threaderzeugung in eCos

```
1  #include <cyg/kernel/kapi.h>
2  void cyg_thread_create
3  (
4      cyg_addrword_t sched_info,
5      cyg_thread_entry_t* entry,
6      cyg_addrword_t entry_data,
7      char* name,
8      void* stack_base,
9      cyg_ucount32 stack_size,
10     cyg_handle_t* handle,
11     cyg_thread* thread
12 );
```

- Threadparameter
- Beliebiger Übergabeparameter
  - z. B. Zeiger auf threadlokale Daten

Threaderzeugung in eCos

```
1  #include <cyg/kernel/kapi.h>
2  void cyg_thread_create
3  (
4      cyg_addrword_t sched_info,
5      cyg_thread_entry_t* entry,
6      cyg_addrword_t entry_data,
7      char* name,
8      void* stack_base,
9      cyg_ucount32 stack_size,
10     cyg_handle_t* handle,
11     cyg_thread* thread
12 );
```

- Beliebiger Threadname
- Tipp: (gdb) info threads

Threaderzeugung in eCos

```
1  #include <cyg/kernel/kapi.h>
2  void cyg_thread_create
3  (
4      cyg_addrword_t sched_info,
5      cyg_thread_entry_t* entry,
6      cyg_addrword_t entry_data,
7      char* name,
8      void* stack_base,
9      cyg_ucount32 stack_size,
10     cyg_handle_t* handle,
11     cyg_thread* thread
12 );
```

- Basisadresse des Threadstacks (→ `&stack[0]`)
- `cyg_uint8`-Array
- Global definieren → Datensegment!
- *Warum ist die notwendig?*

Threaderzeugung in eCos

```
1  #include <cyg/kernel/kapi.h>
2  void cyg_thread_create
3  (
4      cyg_addrword_t sched_info,
5      cyg_thread_entry_t* entry,
6      cyg_addrword_t entry_data,
7      char* name,
8      void* stack_base,
9      cyg_ucount32 stack_size,
10     cyg_handle_t* handle,
11     cyg_thread* thread
12 );
```

- Stackgröße in Bytes

Threaderzeugung in eCos

```
1  #include <cyg/kernel/kapi.h>
2  void cyg_thread_create
3  (
4      cyg_addrword_t sched_info,
5      cyg_thread_entry_t* entry,
6      cyg_addrword_t entry_data,
7      char* name,
8      void* stack_base,
9      cyg_ucount32 stack_size,
10     cyg_handle_t* handle,
11     cyg_thread* thread
12 );
```

- Eindeutiger Identifikator
  - zur “Steuerung” z. B.:

`cyg_thread_resume(handle)`

Threaderzeugung in eCos

```
1  #include <cyg/kernel/kapi.h>
2  void cyg_thread_create
3  (
4      cyg_addrword_t sched_info,
5      cyg_thread_entry_t* entry,
6      cyg_addrword_t entry_data,
7      char* name,
8      void* stack_base,
9      cyg_ucount32 stack_size,
10     cyg_handle_t* handle,
11     cyg_thread* thread
12 );
```

- Speicher für interne Fadeninformationen
  - Fadenzustand u. a. `suspend_count`
- Vermeidung dynamischer Speicherallokation im Kernel

Threaderzeugung in eCos

```
1  #include <cyg/kernel/kapi.h>
2  void cyg_thread_create
3  (
4      cyg_addrword_t sched_info,
5      cyg_thread_entry_t* entry,
6      cyg_addrword_t entry_data,
7      char* name,
8      void* stack_base,
9      cyg_ucount32 stack_size,
10     cyg_handle_t* handle,
11     cyg_thread* thread
12 );
```

Ausführliche Dokumentation

http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html

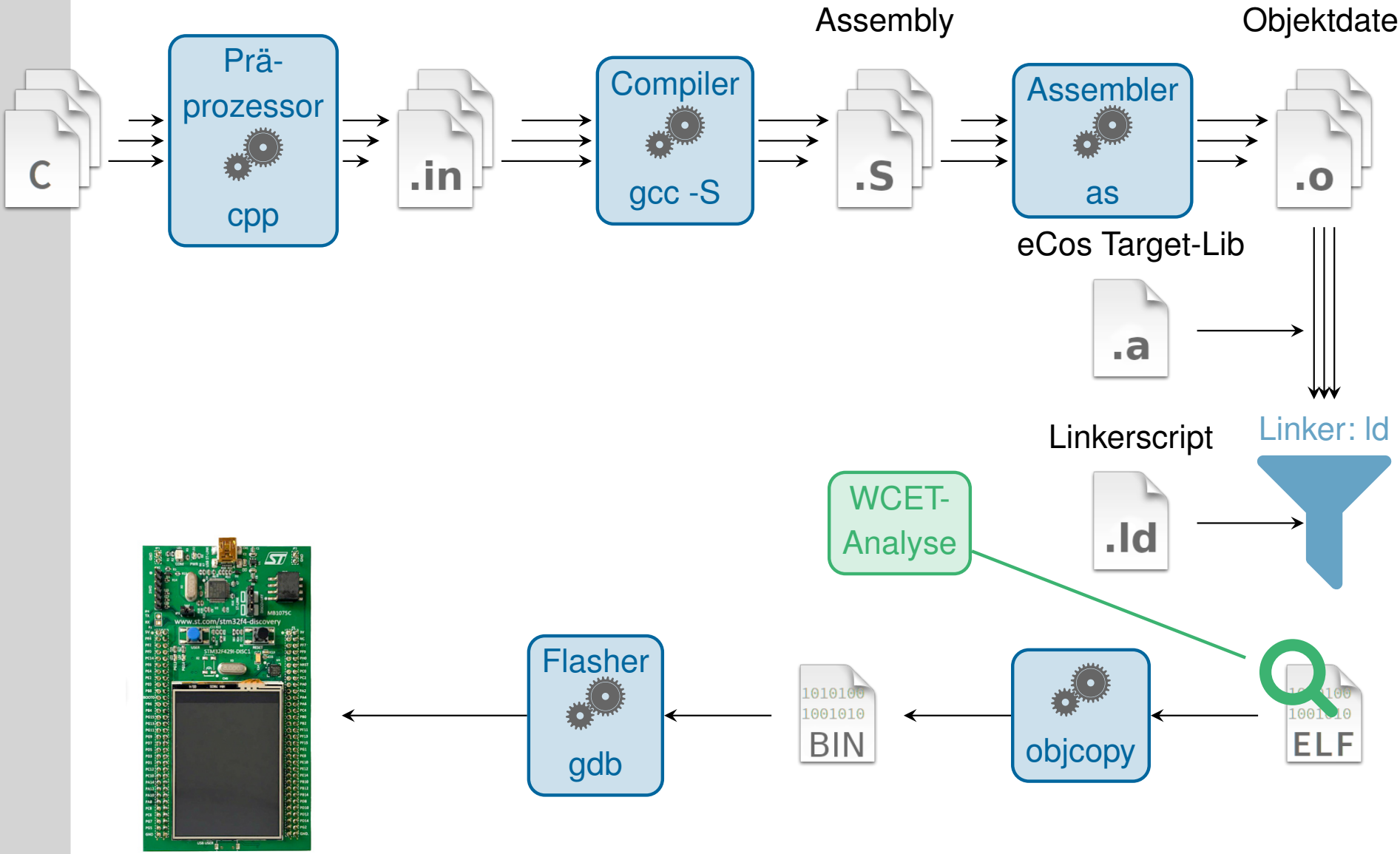
eCos-Beispielanwendung

```
1  #include <cyg/hal/hal_arch.h>
2  #include <cyg/kernel/kapi.h>
3  #include <stdio.h>
4  #define MY_PRIORITY     11
5  #define STACKSIZE       (CYGNUM_HAL_STACK_SIZE_MINIMUM+4096)
6  static cyg_uint8     my_stack[STACKSIZE];
7  static cyg_handle_t  my_handle;
8  static cyg_thread    my_thread;
9
10 static void my_entry(cyg_addrword_t data) {
11   int message = (int) data;
12   ezs_printf("Beginning execution: thread data is %d\n", message);
13   for (;;) {
14     ezs_printf("Hello World!\n"); // \n flushes output
15     ezs_delay_us(1000000); // Delay for 1000000 * 1us = 1 second
16   }
17 }
18 void cyg_user_start(void) {
19     ezs_printf("Entering cyg_user_start() function\n");
20     cyg_thread_create(MY_PRIORITY, &my_entry, 0, "thread 1",
21                       my_stack, STACKSIZE, &my_handle, &my_thread);
22     cyg_thread_resume(my_handle);  }
```

Gliederung

Gliederung

EZS-Toolchain



EZS-Entwicklungsumgebung

Wichtig!

Zu jeder Übungsaufgabe wird eine eCos-Konfiguration bereitgestellt.
Makefiles werden mit `cmake` generiert.

1. Vorgabe herunterladen, entpacken, Verzeichnis betreten
2. **Nötige Umgebungsvariablen setzen:** `source ecosenv.sh`
3. Eigene Quelldateien in `CMakeLists.txt` eintragen
4. `build` Verzeichnis betreten → out-of-source build[2]
5. Makefiles erzeugen: `cmake ..` (*cmake PUNKT PUNKT*)
6. Alles kompilieren: `make`
7. Flashen, ausführen, debuggen: `make flash`
   ⇝ Flasher & Debugger: `make gdb` (später ausführlicher)

[2] `https://gitlab.kitware.com/cmake/community/wikis/FAQ#out-of-source-build-trees`

Dokumentation zum EZS-Board

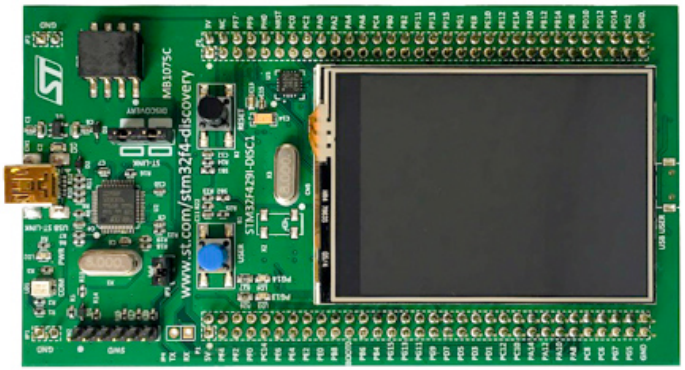
Wiki zum EZS-Board

`https://gitlab.cs.fau.de/ezs/ezs-board/wikis/home`

- Dokumentation im Wiki
  - ☞ Hardware (EZS-Board)
  - ☞ Entwicklungsumgebung
- Erweiterung durch *alle Teilnehmer an EZS* möglich
  - GitLab-Account notwendig

Entwicklungsplattform STM32F429

- ARM Cortex-M4 Prozessor
  - Flash-Speicher: 2 MB
  - RAM: 256 KB
- Umfangreiche Peripherie
  - Touch-LCD
  - Serielle Kommunikation
  - Timer
  - GPIOs
  - ADCs
  - 3-Achsen Gyroskop
  - Beschleunigungssensor
  - Audio Sensor
  - *integrierte Debugging-Schnittstelle*
  - ...



Gliederung

Flashen & Debuggen: Blackmagic Firmware

- Ausleihbare Boards sind mit Blackmagic Firmware[3] ausgestattet
- Mini-USB-Kabel anschließen
- Nach Verbinden des USB-Kabels zwei serielle Ports, z. B.:
  - `/dev/ttyACM0`: Debugger
  - `/dev/ttyACM1`: serielle Kommunikation (Ausgabe z.B. mit `cutecom`)

[3] `https://github.com/blacksphere/blackmagic`

Flashen & Debuggen: Blackmagic Firmware

- Ausleihbare Boards sind mit Blackmagic Firmware[3] ausgestattet
- Mini-USB-Kabel anschließen
- Nach Verbinden des USB-Kabels zwei serielle Ports, z. B.:
  - `/dev/ttyACM0`: Debugger
  - `/dev/ttyACM1`: serielle Kommunikation (Ausgabe z.B. mit `cutecom`)
- Exakte Ports zufällig $\rightsquigarrow$ `/dev/serial/by-id/*Black_Magic*`
- ☞ Für EZS: `/tmp/$USER-ezs-serial`

[3] `https://github.com/blacksphere/blackmagic`

Flashen mittels Kommandozeile

- Bashprompt: %, gdb-Prompt: >

```
% cd <ezs-aufgabe1>
% source ecosenv.sh
% cd build
% cmake ..
% make
% arm-none-eabi-gdb -nh app.elf          # Starten des Debuggers
> target extended-remote /dev/ttyACM0 # gdb ueber ttyACM0
> monitor swd           # Verwendung Serial Wire Debugging (SWD)
> attach 1              # Erstes Interface verwenden
> load                  # Laden des Systems in Flash (Flashen)
> continue              # Starten der Ausfuehrung
```

- gdb `-nh`: Verhindert Ausführung der Befehle aus `~/.gdbinit`
- gdb-Befehle in Datei `flash.gdb` zusammenfassen und starten mittels:
  `arm-none-eabi-gdb -batch -x flash.gdb -nh app.elf`

Flashen mittels Kommandozeile

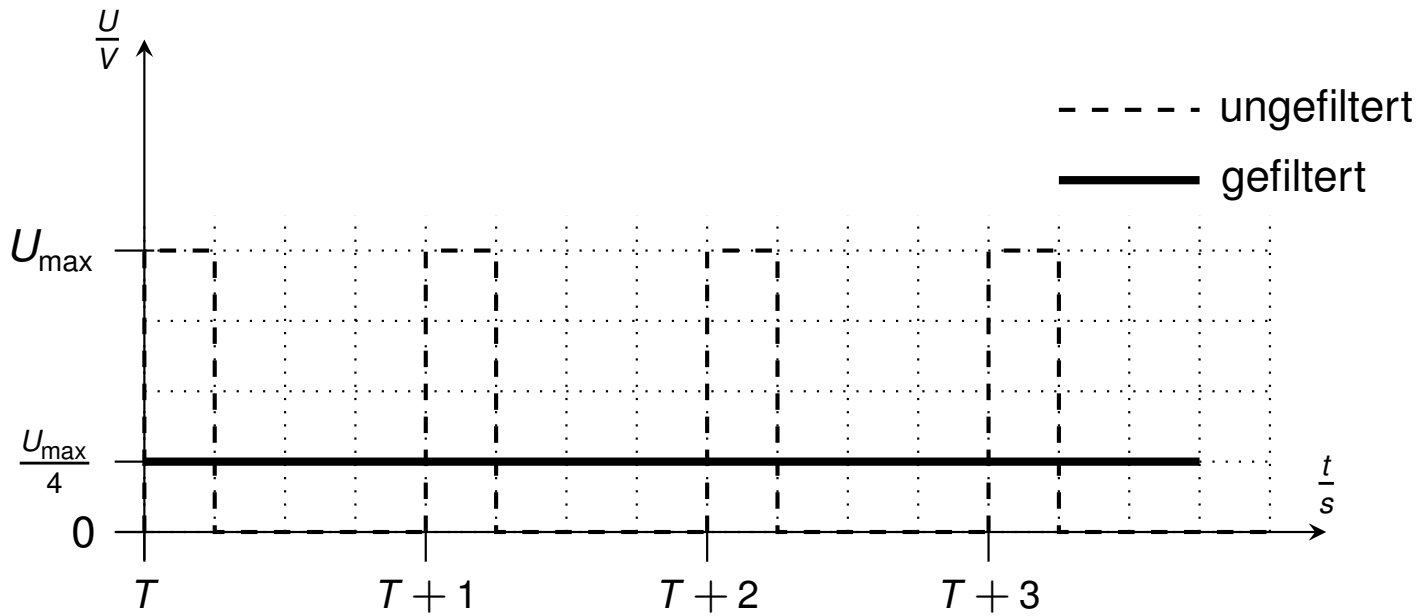
- Bashprompt: %, gdb-Prompt: >

```
% cd <ezs-aufgabe1>
% source ecosenv.sh
% cd build
% cmake ..
% make
% arm-none-eabi-gdb -nh app.elf          # Starten des Debuggers
> target extended-remote /dev/ttyACM0 # gdb ueber ttyACM0
> monitor swd           # Verwendung Serial Wire Debugging (SWD)
> attach 1              # Erstes Interface verwenden
> load                  # Laden des Systems in Flash (Flashen)
> continue              # Starten der Ausfuehrung
```

- gdb `-nh`: Verhindert Ausführung der Befehle aus `~/.gdbinit`
- gdb-Befehle in Datei `flash.gdb` zusammenfassen und starten mittels:
  `arm-none-eabi-gdb -batch -x flash.gdb -nh app.elf`
- bereits implementiert in Target `make flash`

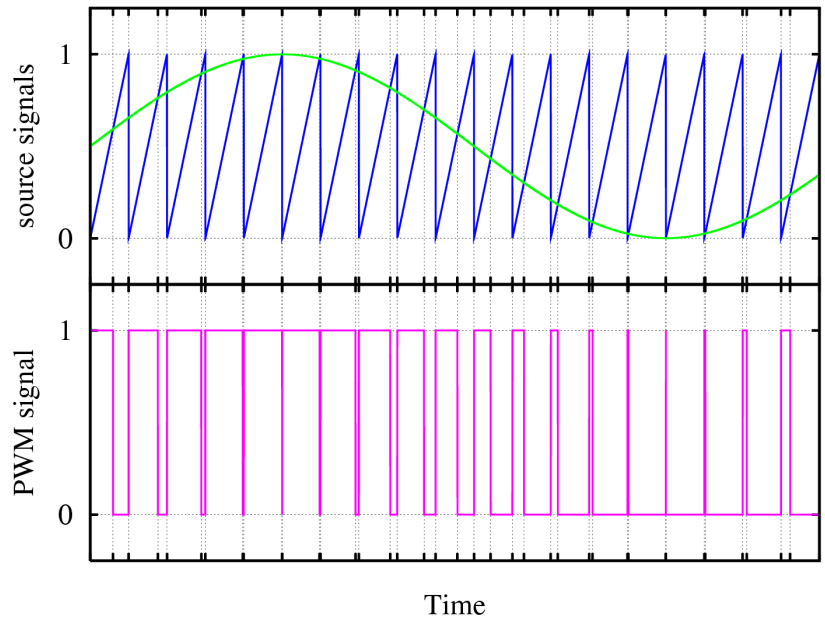
Gliederung

Digital-Analog Umwandlung & Pulsweitenmodulation I



- Einschaltdauer proportional zum Mittelwert des Ausgangssignals
- Variation der Pulsweite oder Einschaltdauer (engl. duty cycle)
- Pulsweitenmodulation (engl. pulse-width modulation, PWM)
- „Pseudo“ Digital-Analog-Wandler (engl. digital-analog converter, DAC)

Digital-Analog Umwandlung & Pulsweitenmodulation II



Verfahren zur **Signalerzeugung**

- Hardware: Vergleich periodischer Zähler und Wert der Einschaltdauer
- Weit verbreitet: Motorsteuerung, Class-D-Verstärker, Schaltnetzteile, Nachrichtenübertragung,. . .
- Mittels Tiefpass $\rightsquigarrow$ Digital-Analog-Wandlung
- libEZS: `void ezs_dac_write(uint8_t)` (zusätzlich *echter* DAC auf Board)
  $\rightsquigarrow$ Wertebereich 0-255

Gliederung

Tiefpassfilter

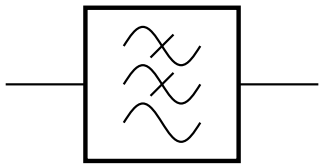


Abbildung: Schaltbild RC-Glied

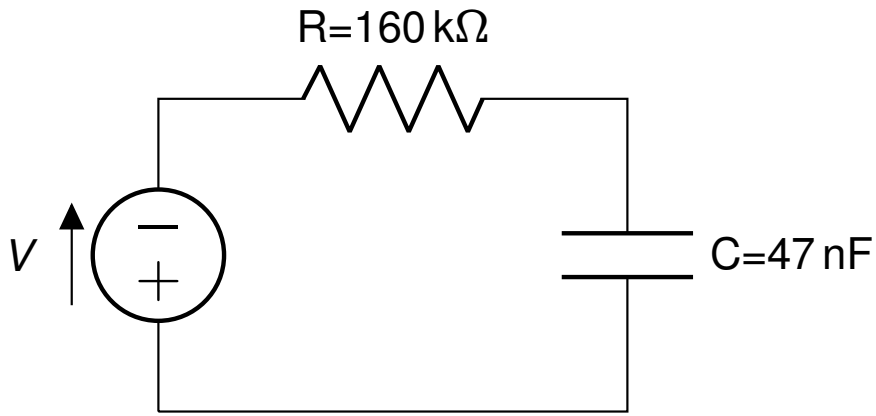


Abbildung: Schaltung RC-Filter auf Filterbaustein

- Filterung hochfrequenter Schwingungen
- Zeitkonstante $\tau = R \cdot C = 7,52\text{ms}$
- *Grenzfrequenz* (Dämpfung um 3dB $\approx 71\,\%$): $f_c = \frac{1}{2 \cdot \pi \cdot \tau} = 21\,Hz$

Tiefpassfilter anschließen

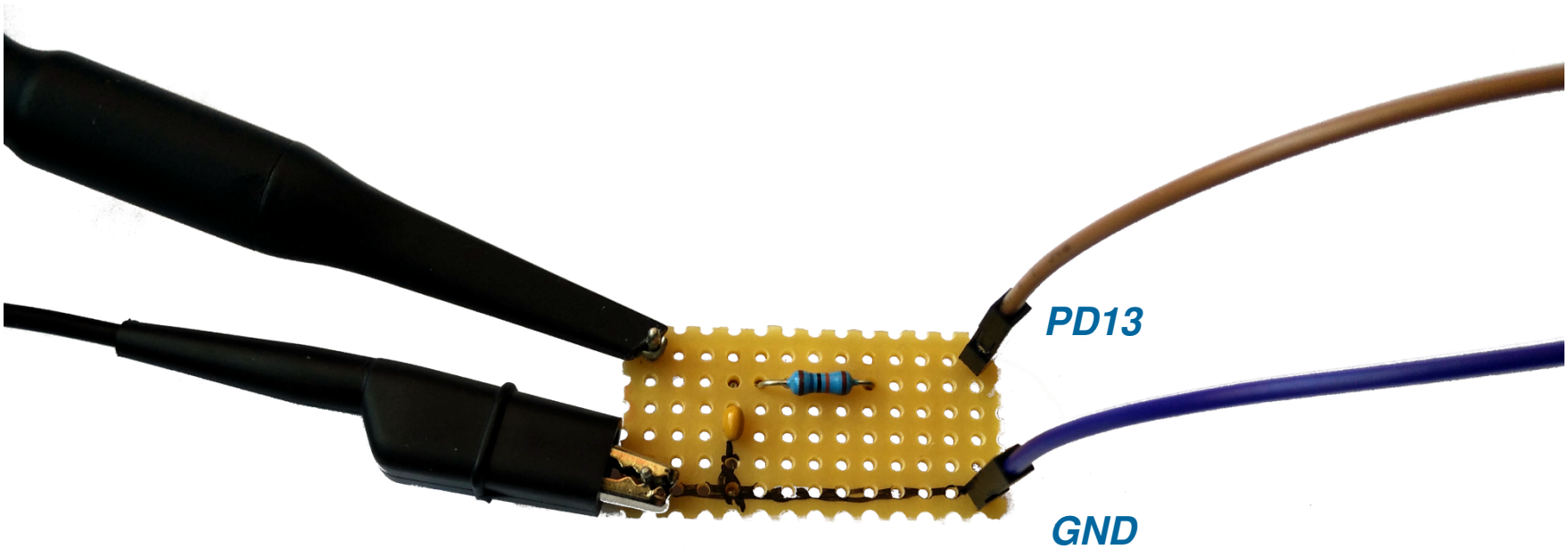
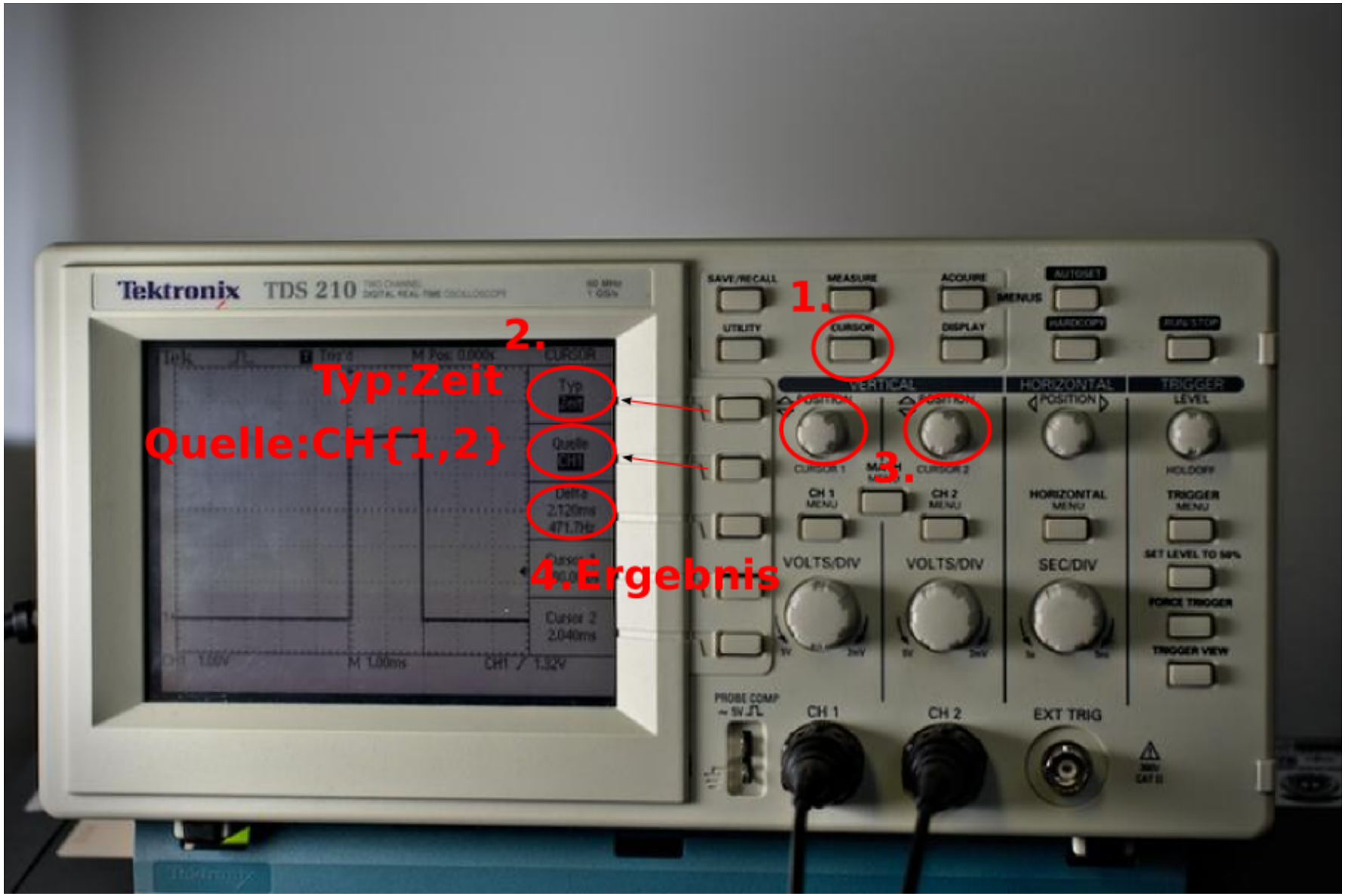


Abbildung: Filterbaustein anschließen, *schwarzer Strich markiert Masse*

Gliederung

Cursor


Tektronix TDS 210
1.
2.
Typ:Zeit
Quelle:CH{1,2}
3.
4.Ergebnis
SAVE/RECALL
MEASURE
ACQUIRE
UTILITY
CURSOR
DISPLAY
VERTICAL
HORIZONTAL
TRIGGER
VOLTS/DIV
VOLTS/DIV
SEC/DIV
CH 1
CH 2
EXT TRIG

Gliederung

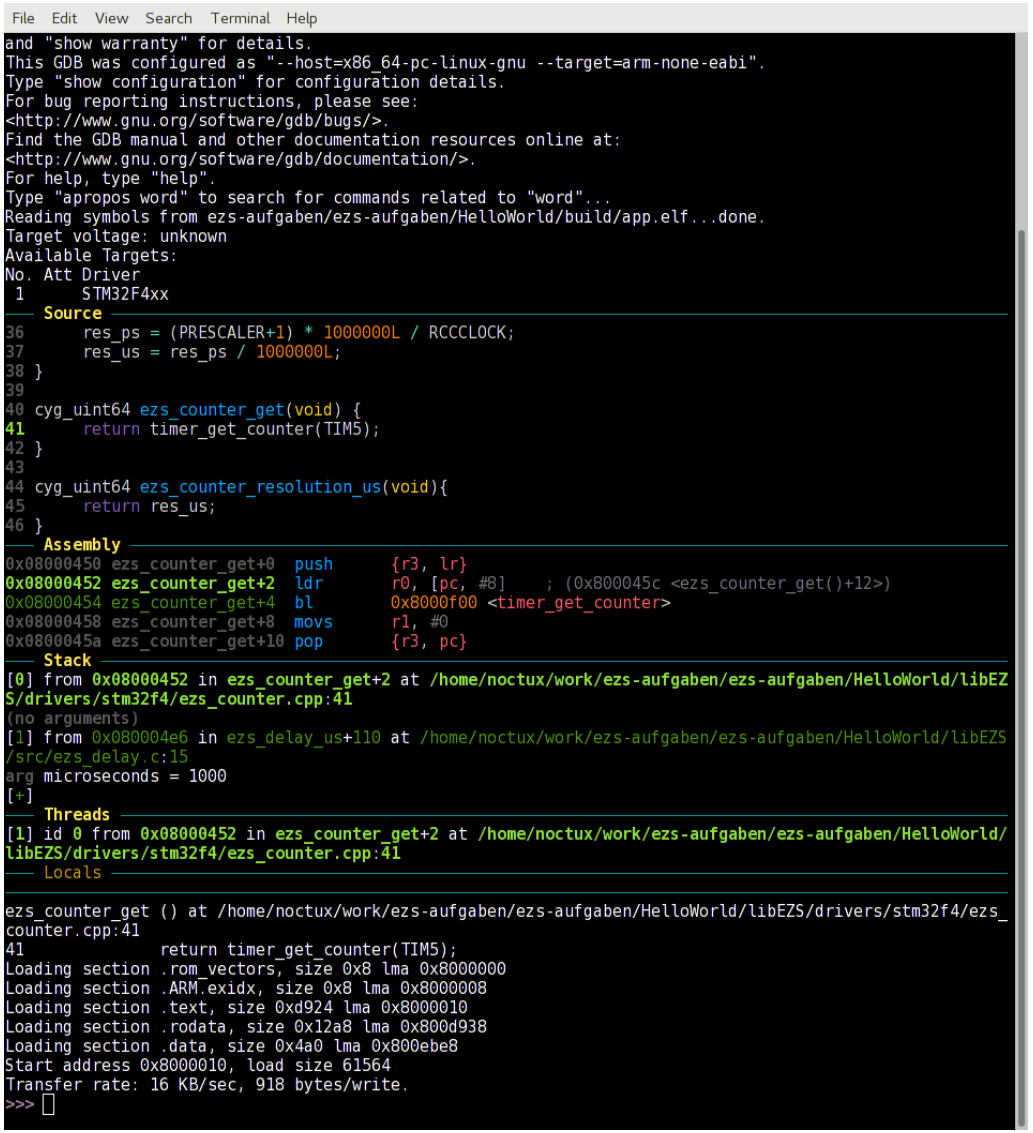
gdb-Dashboard

Aufrufen

- Interaktive gdb-Session:
  `% make gdb`
- gdb-Dashboard:
  `% make debug`
- Manueller Aufruf:
  `% arm-none-eabi-gdb \`
  `-x ezs_dashboard.gdb app.elf`
- Parameter `-nh` verwenden falls `.gdbinit` vorhanden

Fenster

1. Source Code
2. Assembly
3. Stack
4. Threads
5. Lokale Variablen

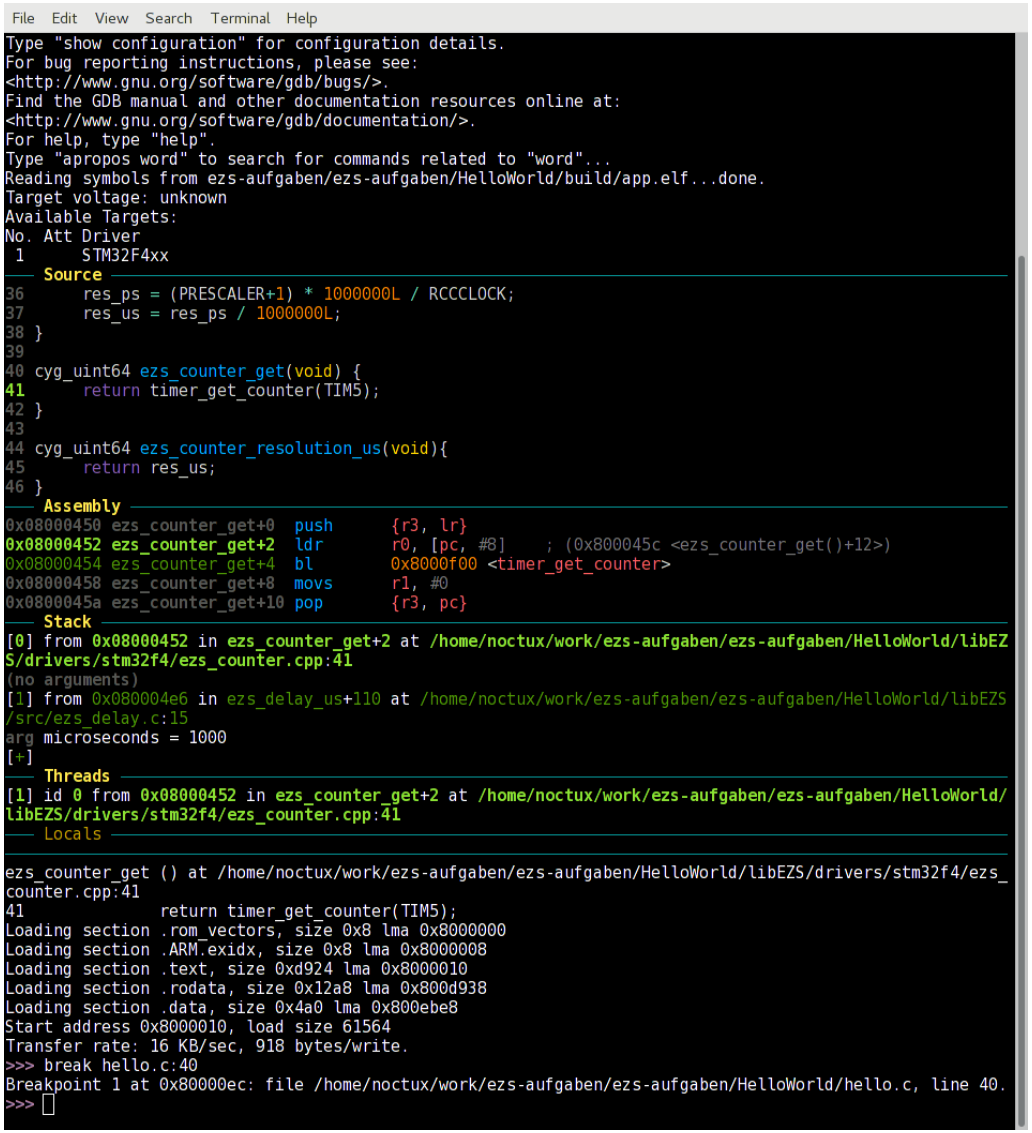


gdb Kommandos – I

Befehle haben Langformen (`break`) und Kurzformen (b)

Wichtige Befehle

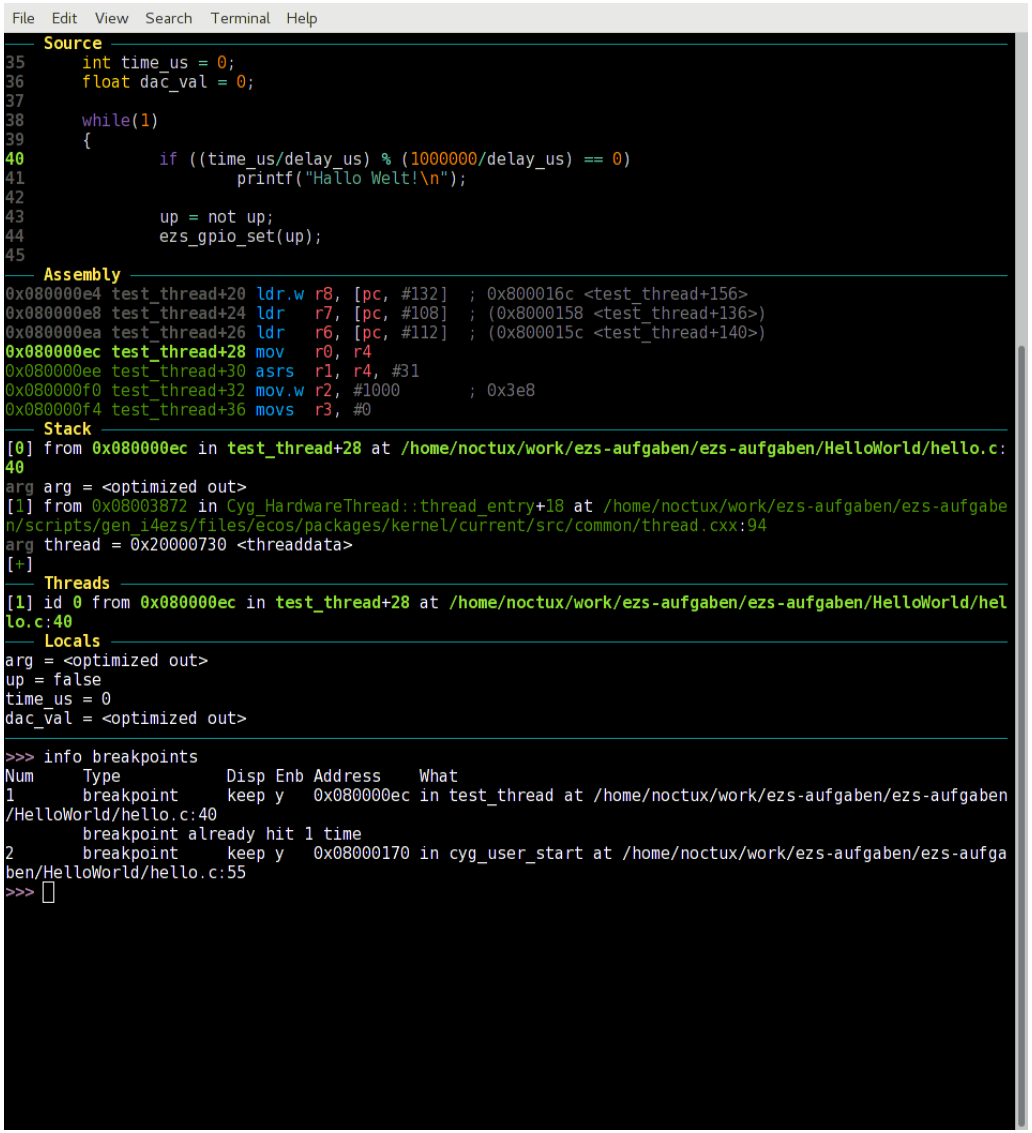
- Breakpoint setzten:
  `»> b(reak) cyg_user_start`
- Einzelschritt (Funktionen betreten):
  `»> s(tep)`
- Einzelschritt
  (Funktionen nicht betreten):
  `»> n(ext)`
- Programm fortsetzten:
  `»> c(ontinue)`
- Bis zum Ende der Funktion
  ausführen:
  `»> fin(ish)`
- Funktion anzeigen:
  `»> l(ist) <funktionsname>`
- gdb schließen:
  `»> q(uit) (oder Strg+D)`
- Neu Flashen:
  `»> l(oad)`



gdb Kommandos – II

Wichtige Befehle (Fortsetzung)

- Backtrace (Aufruf-Stack) anzeigen:
  `»> b(ack)t(race)`
- Dashboard neu zeichnen:
  `»> dashboard`
- Breakpoints anzeigen:
  `»> info breakpoints`
- Breakpoint löschen:
  `»> delete <nummer>`
- Variable anzeigen:
  `»> p(rint) <variablenname>`



Weiterführende Informationen

- gdb-Dashboard benötigt einen gdb mit python-Bindings
- gdb „abschießen“:[4]
  `% killall gdb-multiarch -s SIGKILL`
- EZS-Board in initialen Zustand setzen:
  USB-Kabel abstecken & wieder anstecken
- GDB-Spickzettel:
  `http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf`

GDB-Einführung aus BS

`https://www4.cs.fau.de/Lehre/WS19/V_BS/Uebungen/seminar-gdb.pdf`

[4] `gdb-multiarch` ist der Name des gdb-Binaries im CIP, kann bei euch zu Hause abweichen

Übersicht hilfreicher `make`-Targets

flash	Baut Anwendung und schreibt sie auf das Board.
gdb	Startet eine interaktiver GDB-Sitzung.
debug	Startet eine GDB-Sitzung mit dem EZS-Dashboard.
doc	Erstellt Dokumentation für die von uns bereitgestellten Funktionen.
sanity-test	Testet grundlegende Funktionalität der Aufgabe.
submit	Erzeugt eine Abgabe (*rechtzeitig* aufrufen).
diff	Zeigt Unterschiede zwischen dem aktuellen Stand und der letzten Abgabe.

Besprechung der Übungsaufgabe

„Hallo Welt!“

Fragen. . .

42