Echtzeitsysteme

Übungen zur Vorlesung

Entwicklungsumgebung

Florian Schmaus Peter Wägemann

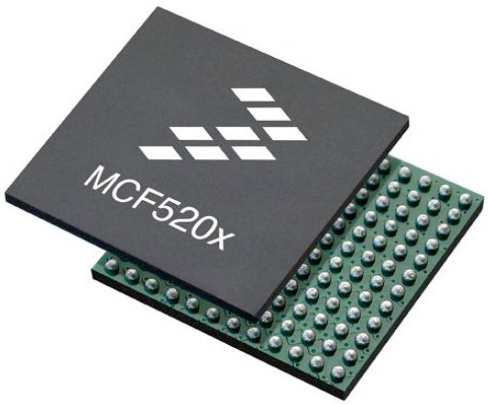
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)
https://www4.cs.fau.de

19.10.2017

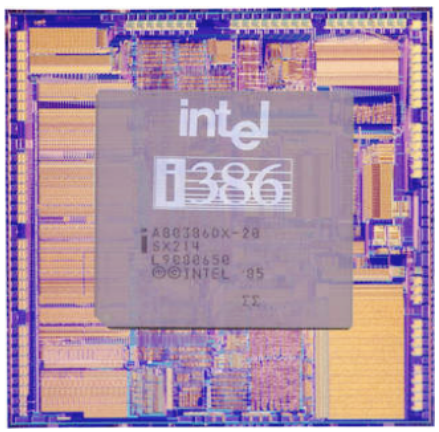
Übersicht

Prozessorvielfalt in der Echtzeitwelt

freescale™ semiconductor



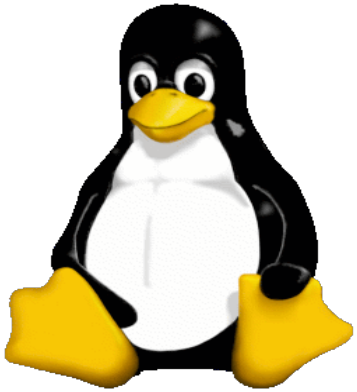
ARM. Cortex
Intelligent Processors by ARM®



Noch mehr Betriebssysteme

freeRTOS

µClinux



android

eCos

QNX

TinyOS

µC/OS-II™
The Real-Time Kernel

HIGH TEC

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)

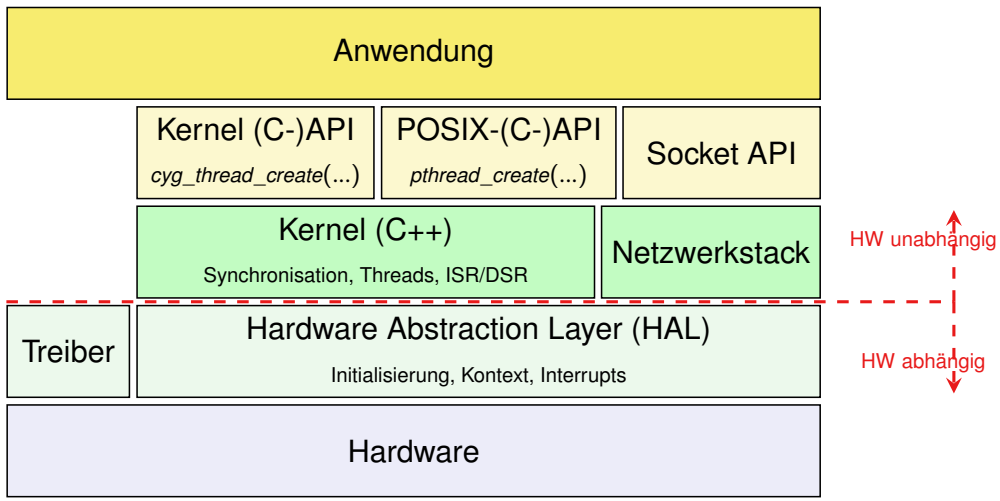
☞ `http://ecos.sourceware.org`

Unterstützte Plattformen

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

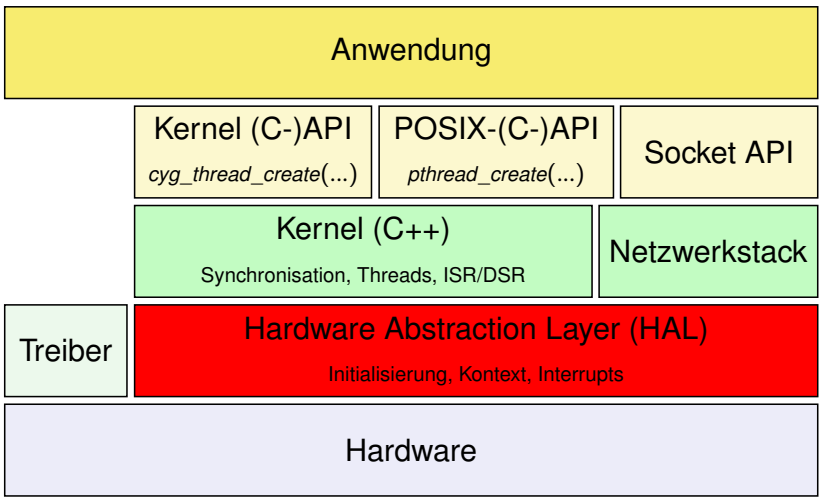
- Fujitsu SPARClite
- Matsushita MN10300
- Motorola PowerPC
- Advanced RISC Machines (ARM)
- Toshiba TX39
- Infineon TriCore
- Hitachi SH3
- NEC VR4300
- MB8683X
- ☞ *ARM Cortex*
- ☞ *Intel x86*
- ...

eCos-Systemarchitektur



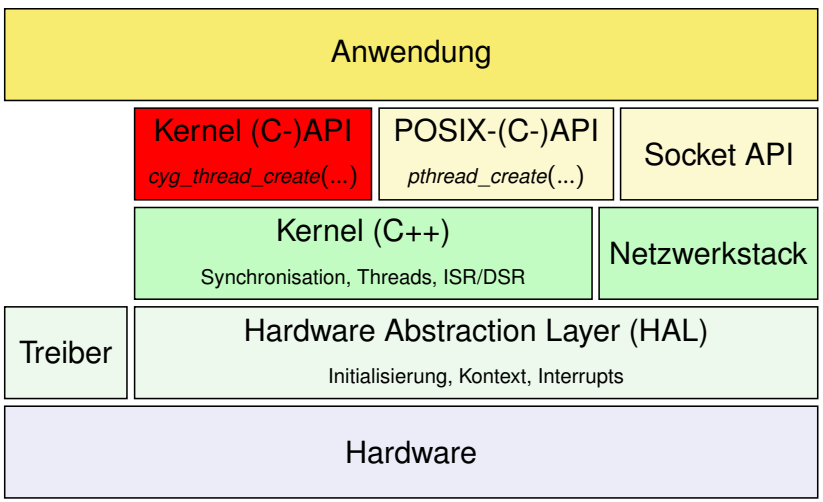
eCos-Systemarchitektur

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



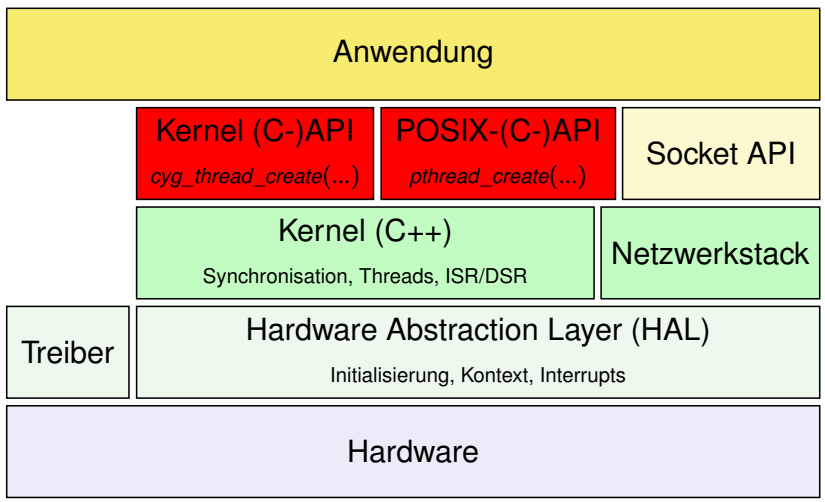
eCos-Systemarchitektur

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



eCos-Systemarchitektur

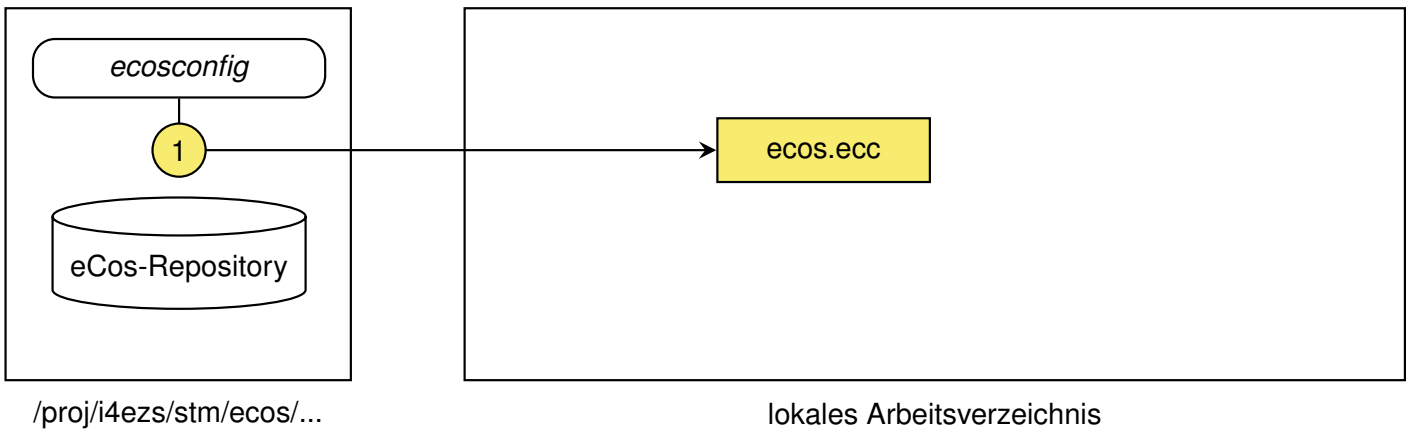
- Kernel API
  - vollständige C-Schnittstelle
  - siehe Dokumentation[1]
- (Optionale) POSIX-Kompatibilitätsschicht
  - Scheduling-Konfiguration, *pthread_*∗
  - Timer, Semaphore, Message Queues, Signale, ...



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

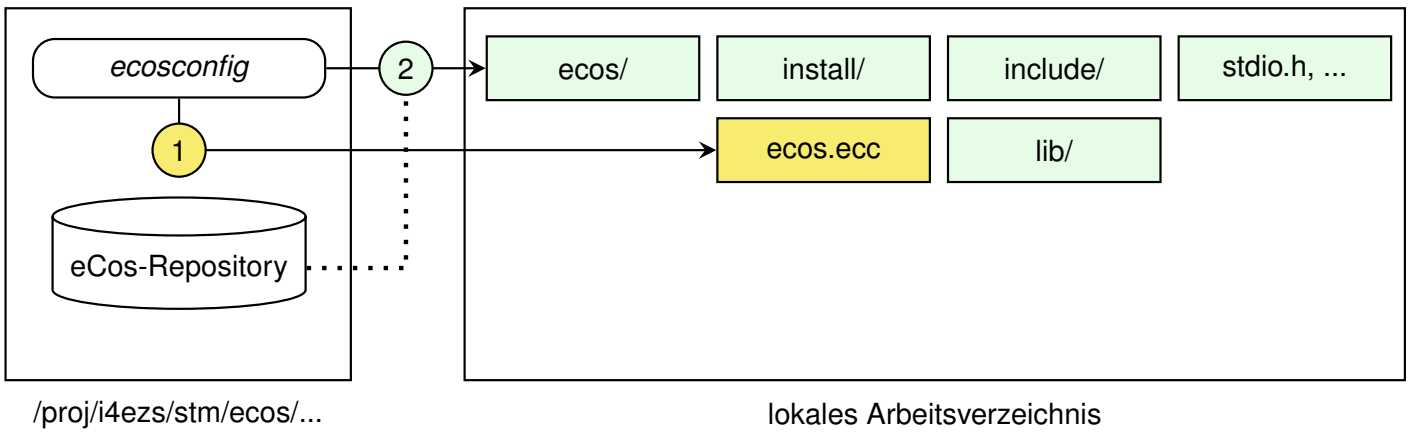
eCos-Entwicklungszyklus

1 Erstellen einer Konfiguration (configtool/ecosconfig)



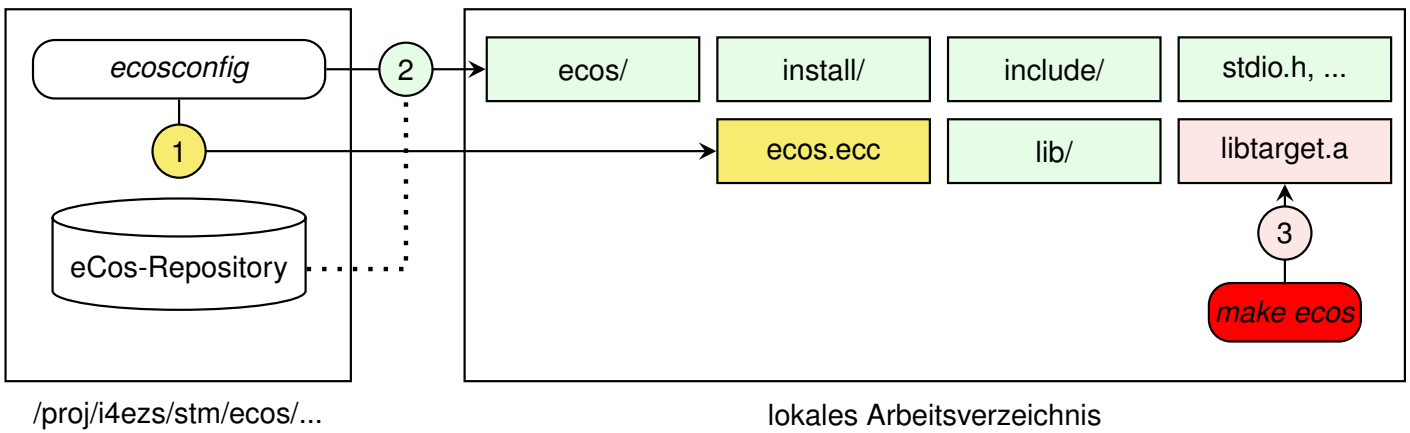
eCos-Entwicklungszyklus

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



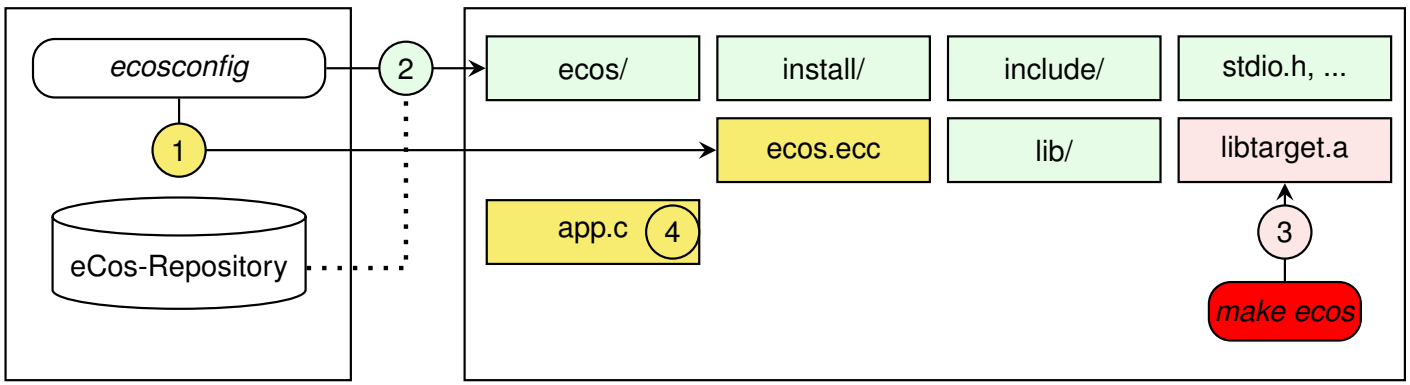
eCos-Entwicklungszyklus

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



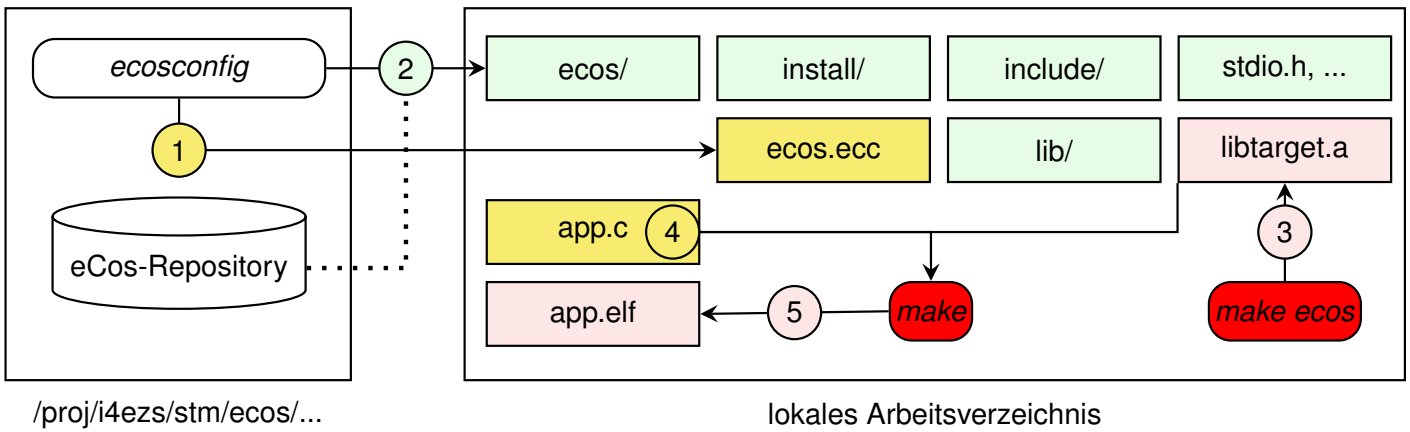
eCos-Entwicklungszyklus

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



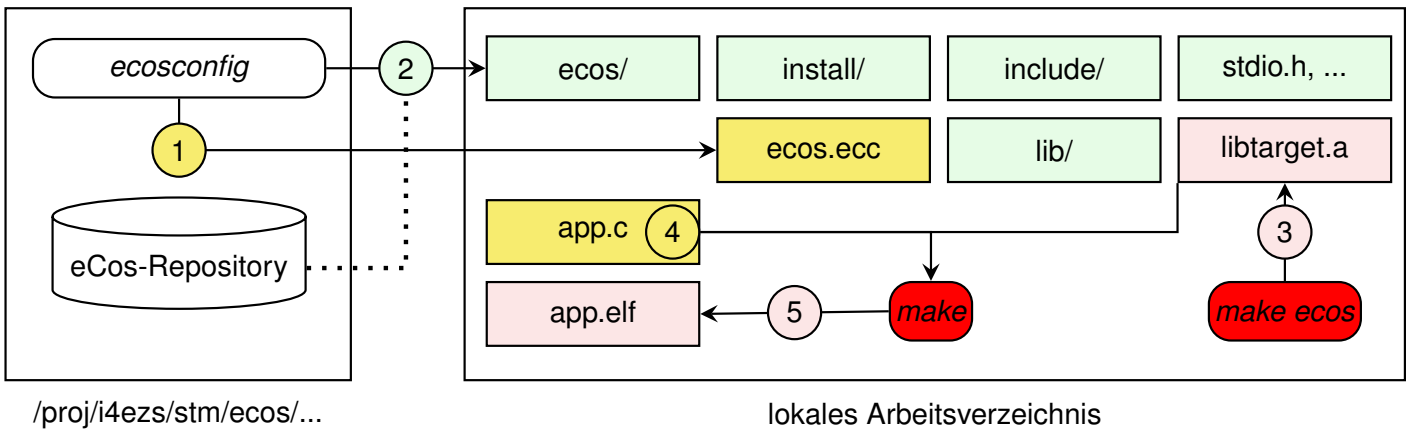
eCos-Entwicklungszyklus

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

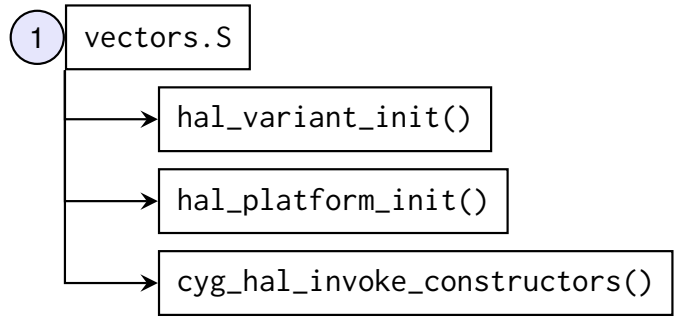


Wichtig!

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

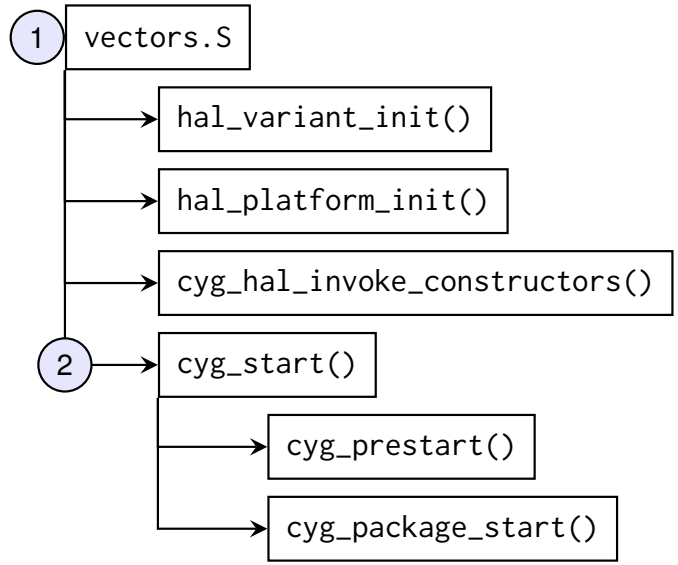
eCos-Systemstart

1. vectors.S
   - Hardwareinitialisierung
   - Globale Konstruktoren



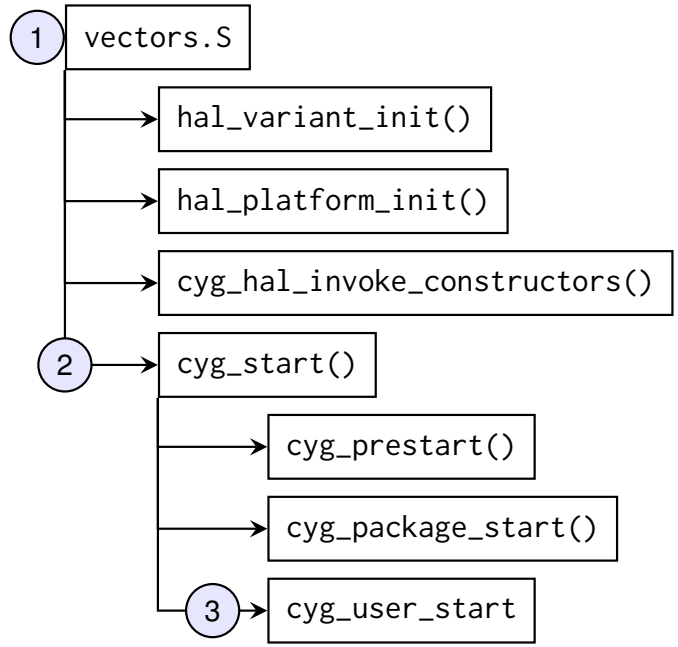
eCos-Systemstart

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



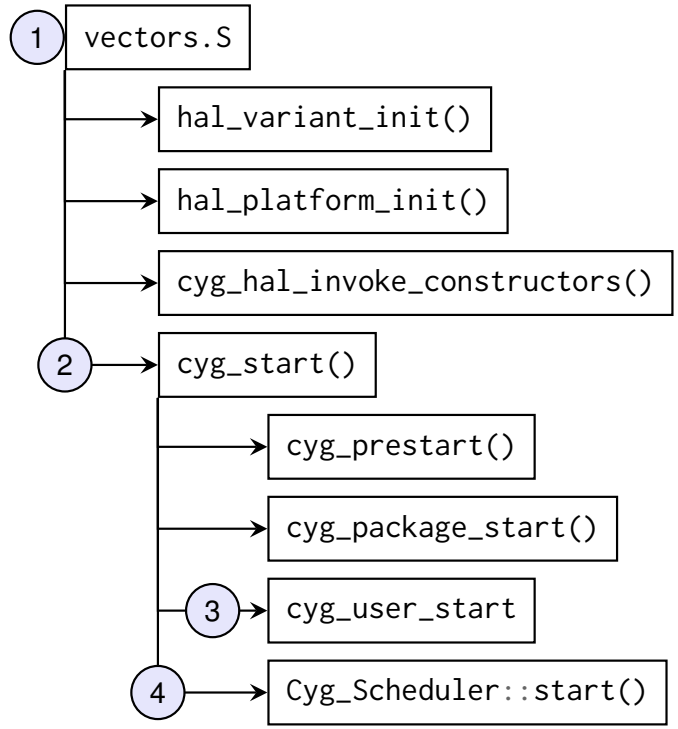
eCos-Systemstart

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



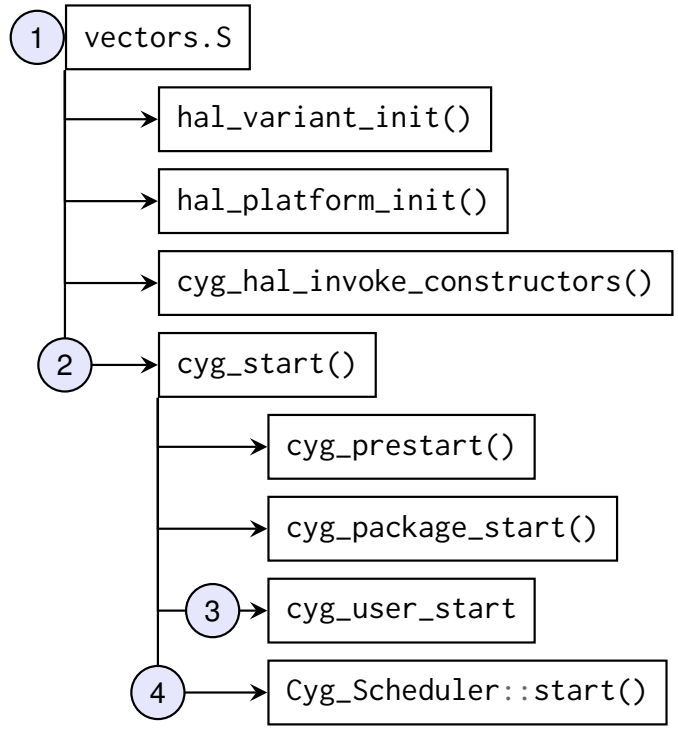
eCos-Systemstart

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

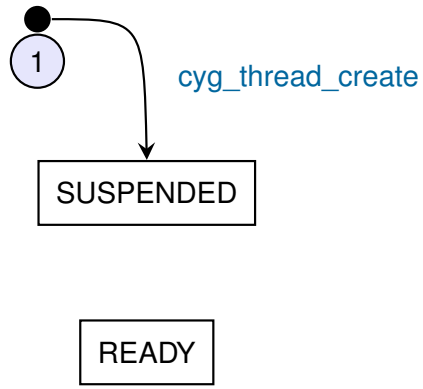


Wichtig!

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

eCos-Threads

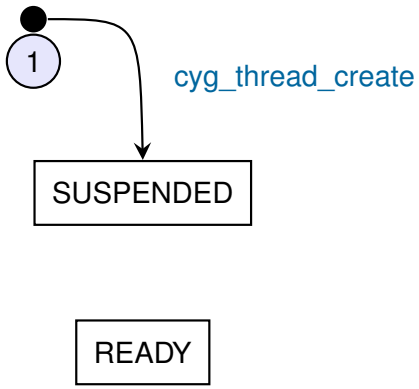
1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`



eCos-Threads

Threadzustände und Übergänge

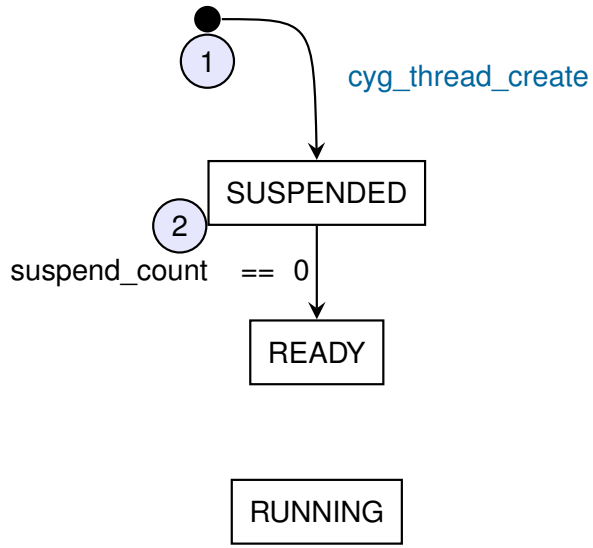
1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`
2. `cyg_thread_resume` aktiviert
   - `suspend_count --`



eCos-Threads

Threadzustände und Übergänge

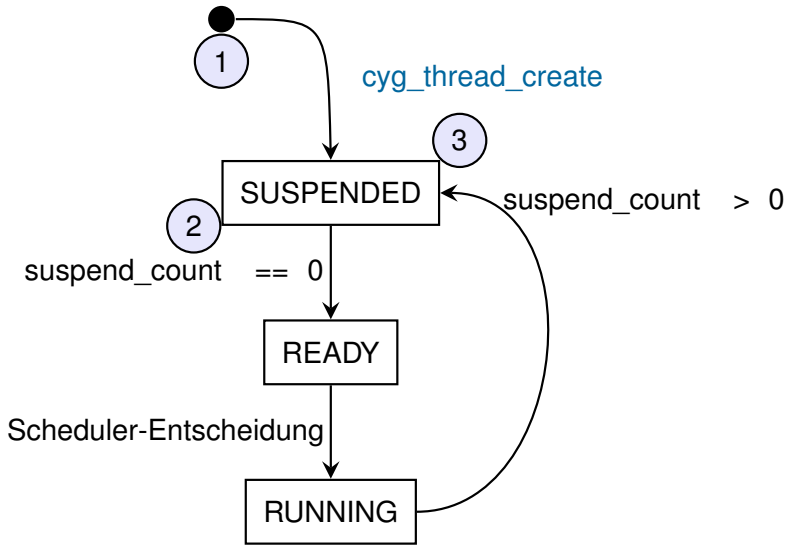
1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`
2. `cyg_thread_resume` aktiviert
   - `suspend_count --`
3. `cyg_thread_suspend` suspendiert
   - `suspend_count++`



eCos-Threads

Threadzustände und Übergänge

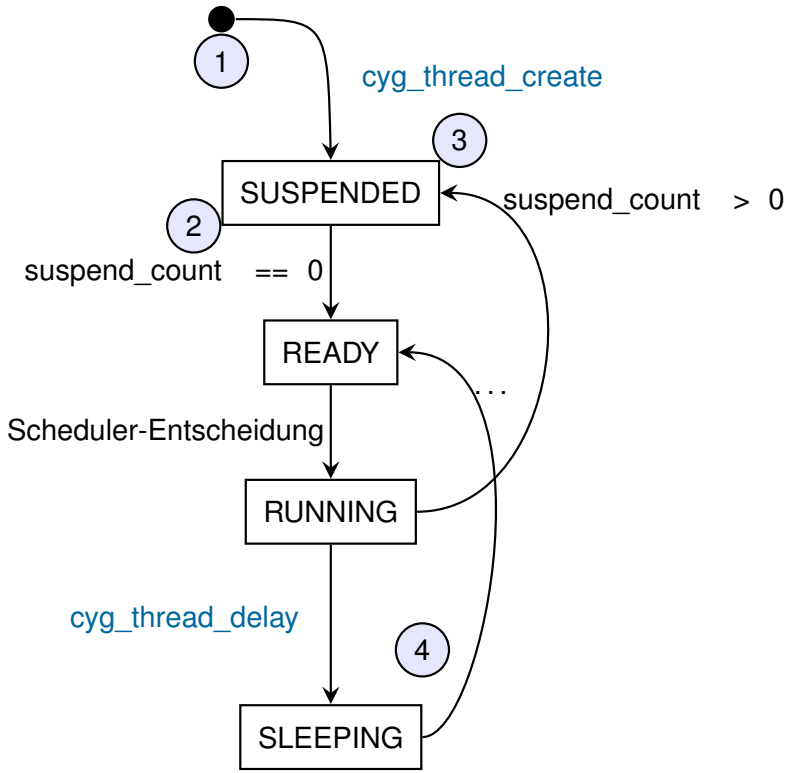
1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`
2. `cyg_thread_resume` aktiviert
   - `suspend_count --`
3. `cyg_thread_suspend` suspendiert
   - `suspend_count++`
4. bereit



eCos-Threads

Threadzustände und Übergänge

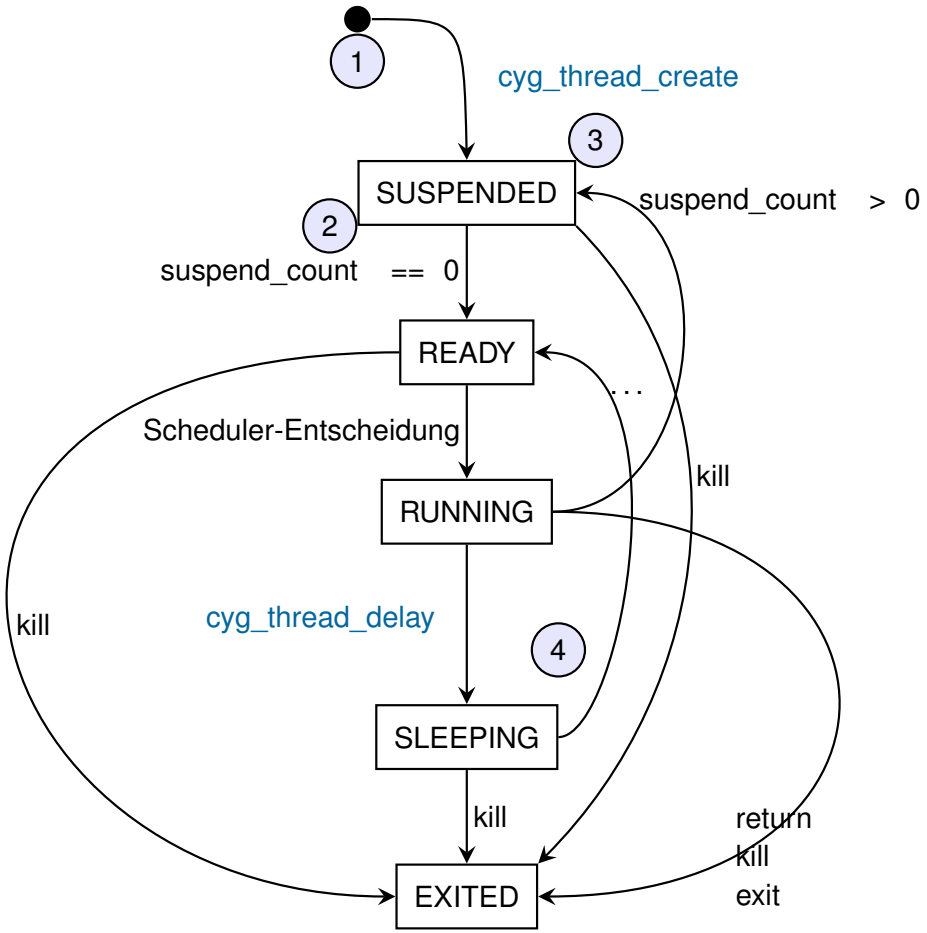
1. Thread wird im Zustand *suspended* erzeugt.
   - `suspend_count = 1`
2. `cyg_thread_resume` aktiviert
   - `suspend_count --`
3. `cyg_thread_suspend` suspendiert
   - `suspend_count++`
4. bereit
5. delay, mutex, semaphore wait



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. bereit
5. delay, mutex, semaphore wait
6. Threadterminierung



eCos-Beispielanwendung


```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

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

eCos-Beispielanwendung


```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

- Scheduling-Informationen
- z. B. MLQ-Scheduler
  - Threadpriorität
  - Datentyp `cyg_uint8`

eCos-Beispielanwendung


```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

- Thread Einsprungpunkt
- Funktionszeiger
- Signatur:
  `void (*)(cyg_addrword_t)`

eCos-Beispielanwendung


```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

- Thread Parameter
- Beliebige Übergabeparameter
  - z. B. Zeiger auf threadlokale Daten

eCos-Beispielanwendung


```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

- Beliebiger Threadname
- Tipp: (gdb) info threads

eCos-Beispielanwendung


```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

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

eCos-Beispielanwendung


```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

- Stackgröße in Bytes

eCos-Beispielanwendung


```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

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

`cyg_thread_resume(handle)`

eCos-Beispielanwendung


```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

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

eCos-Beispielanwendung

Threads erzeugen

```
#include <cyg/kernel/kapi.h>
void cyg_thread_create
(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t* entry,
    cyg_addrword_t entry_data,
    char* name,
    void* stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t* handle,
    cyg_thread* thread
);
```

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  static void my_entry(cyg_addrword_t data) {
10   int message = (int) data;
11   ezs_printf("Beginning execution: thread data is %d\n", message);
12   for (;;) {
13     ezs_printf("Hello World!\n"); // \n flushes output
14     ezs_delay_us(1000000); // Delay for 1000000 * 1us = 1 second
15   }
16 }
17 void cyg_user_start(void) {
18     ezs_printf("Entering cyg_user_start() function\n");
19     cyg_thread_create(MY_PRIORITY, &my_entry, 0, "thread 1",
20                       my_stack, STACKSIZE, &my_handle, &my_thread);
21     cyg_thread_resume(my_handle);  }
```

eCos-Beispielanwendung

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] `www.cmake.org/Wiki/CMake_FAQ#Out-of-source_build_trees`

Übersicht

Dokumentation zum EZS-Board

Wiki zum EZS-Board

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

- Umstellung auf neues Entwicklungsboard

  ⇝ Mögliche Probleme

- Nicht unterstützte Plattform

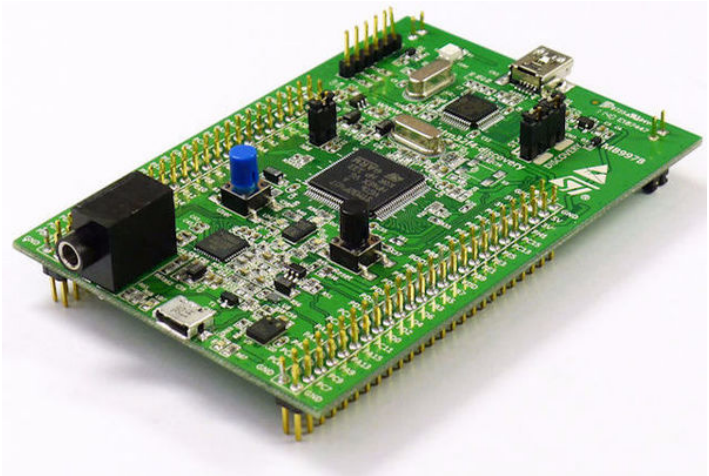
☞ Dokumentation im Wiki

- Erweiterung durch *alle Teilnehmer an EZS*
  - gitlab account notwendig
- Besondere Aufgaben
  - Anleitung „EZS-Board unter Windows“
  - Anleitung „EZS-Board unter macOS“

☞ Gutscheine für I4-Kaffeekarte

Entwicklungsplattform STM32F411

- ARM Cortex-M4 Prozessor
  - Flash-Speicher: 512 KB
  - RAM: 128 KB
- Umfangreiche Peripherie
  - Serielle Kommunikation
  - Timer
  - GPIOs
  - ADCs
  - 3-Achsen Gyroskop
  - Beschleunigungssensor
  - Audio Sensor
  - *integrierte Debugging-Schnittstelle*
  - ...



Flashen & Debuggen: Blackmagic Firmware

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

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

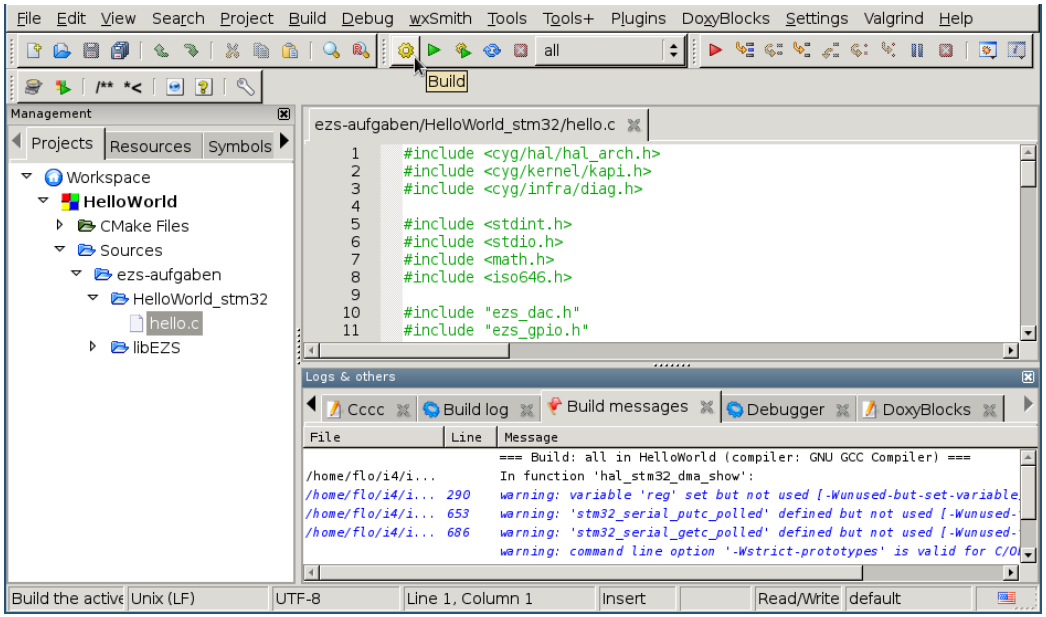
Flashen mittels Kommandozeile

- 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`

Editieren mit kate und codeblocks



- Anleitungen zum Editieren mit kate und codeblocks im Wiki
- Generieren der Projektdateien
  - `cmake -G"Kate - Unix Makefiles"..`
  - `cmake -G"CodeBlocks - Unix Makefiles"..`
- Weitere Makefile-Targets
  - `make flash`, `make gdb`, `make edit`, ...
  - Mehr dazu ⇝ Rechnerübung

Übersicht

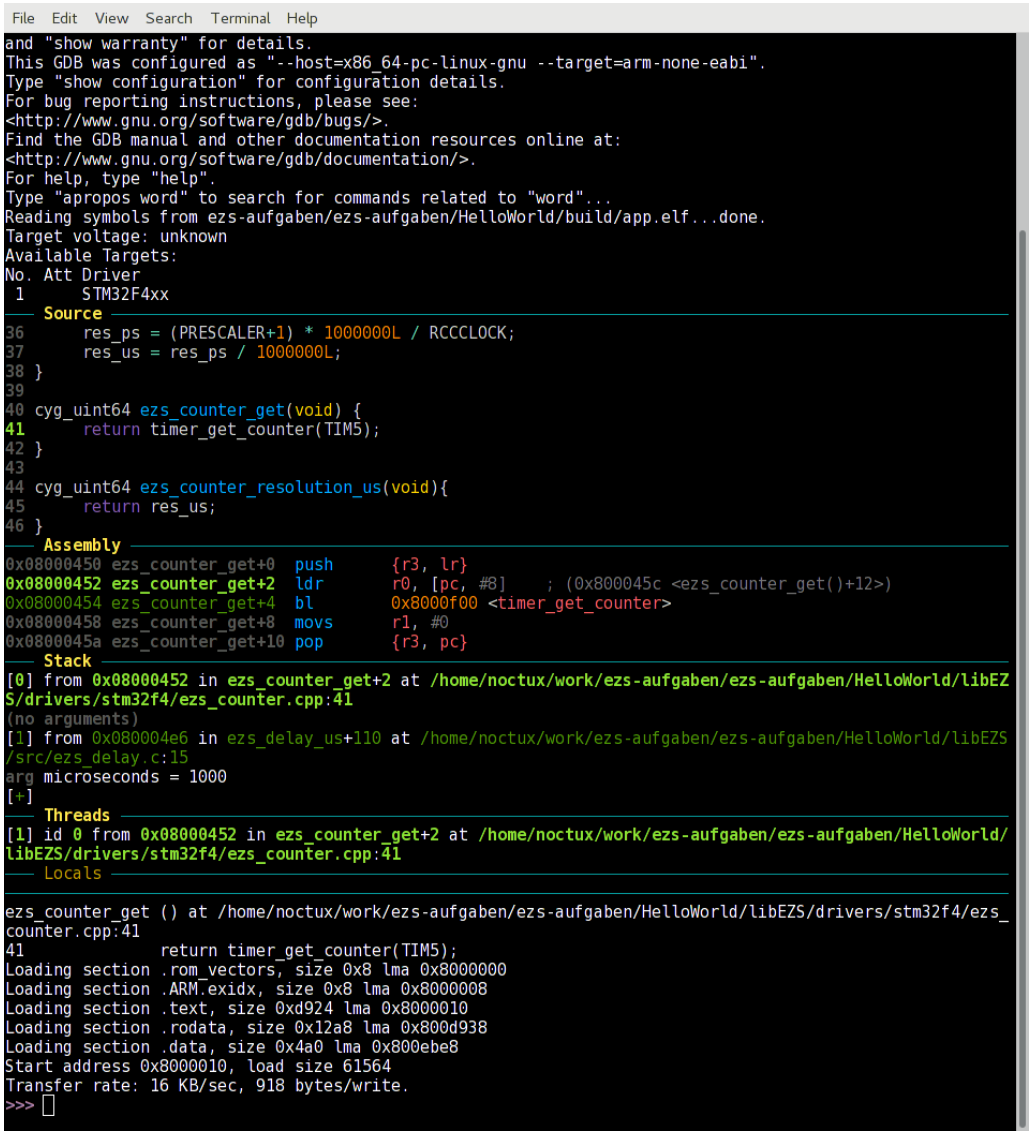
gdb-Dashboard

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 Varibalen

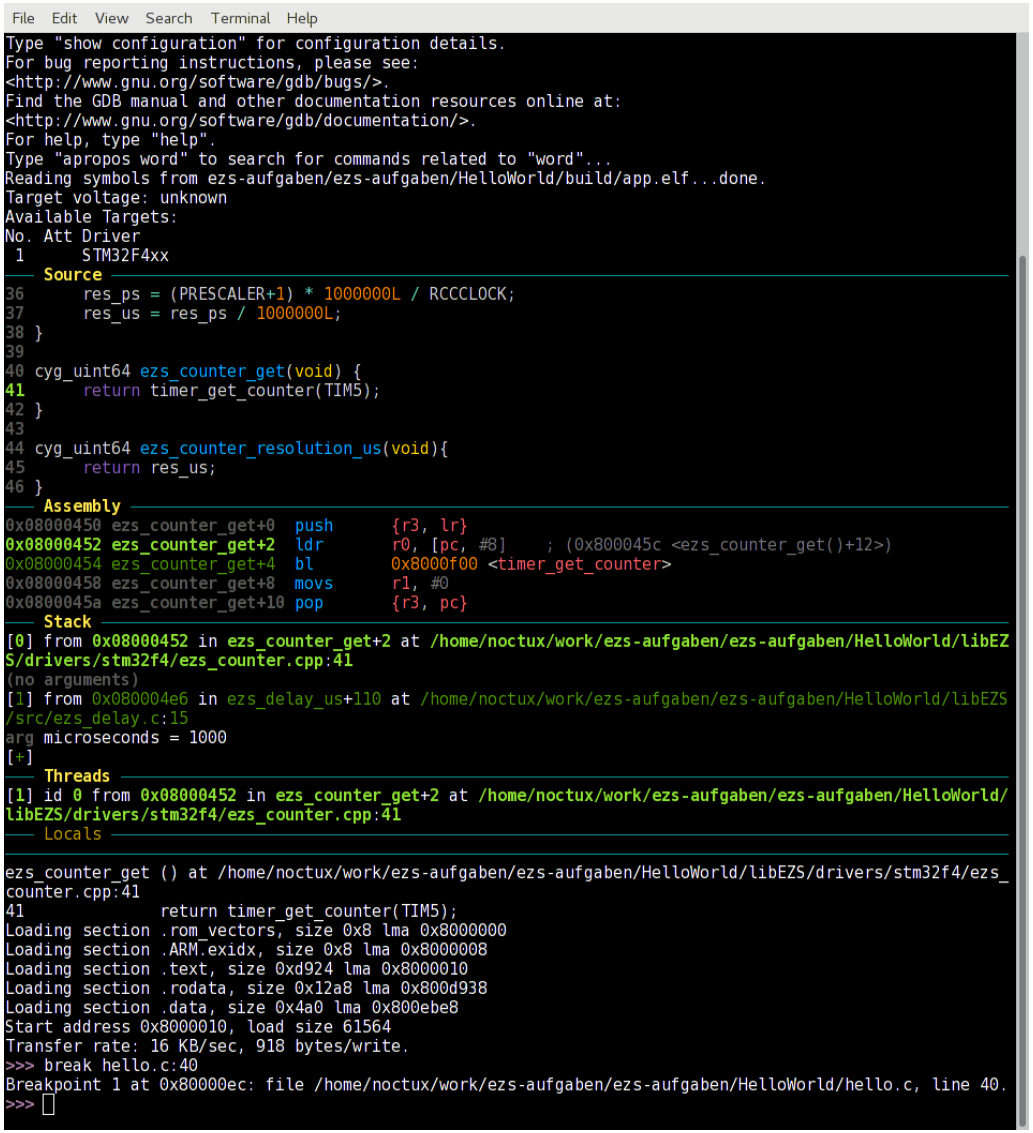


gdb Kommandos – I

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

Wichtige Befehle

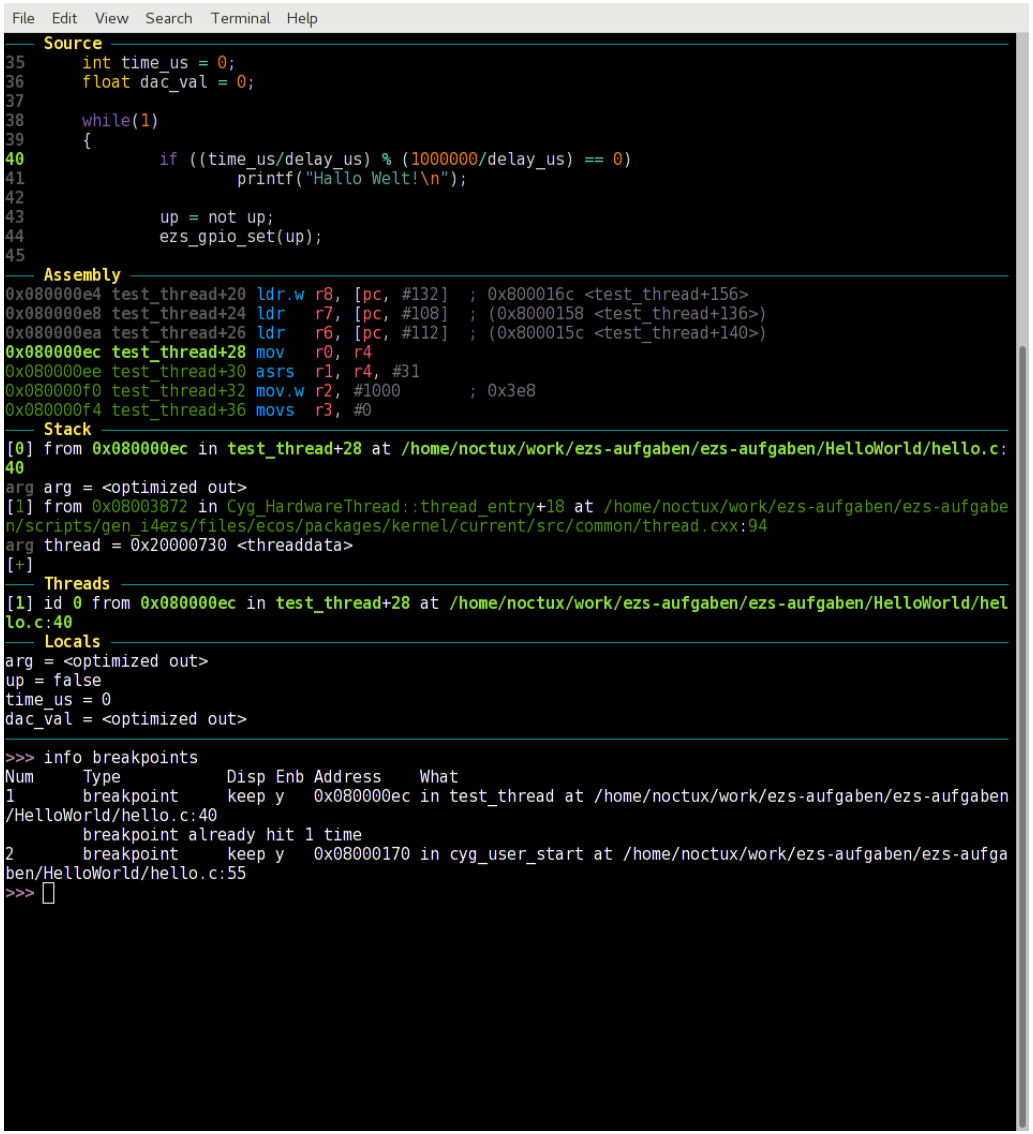
- Breakpoint setzten:
  `>>> b(reak) cyg_user_start`
- Einzelschritt (Funktionen betreten):
  `>>> s(tep)`
- Einzelschritt
  (Funktionen nicht betreten):
  `>>> n(ext)`
- Programm fortsetzten:
  `>>> c(ontinue)`
- Bis zum Ende der Funktion
  ausführen:
  `>>> fin(ish)`
- Funktion anzeigen:
  `>>> l(ist) <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“:

```
% killall arm-none-eabi-gdb -s SIGKILL
```

- EZS-Board in initialen Zustand setzen:
  USB-Kabel abstecken & wieder anstecken
- GDB Spickzettel:
  `http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf`

Besprechung der Übungsaufgabe

„Hallo Welt!“

Fragen?

Fragen?