Hello RPi [1]
first IKT [2] tutorial with bare metal assembler

version 1, 2013-10

Matthias Vierthaler BSc

Graz University of Technology, ITI

matthias.vierthaler@student.tugraz.at

Oct 2013

[1] Raspberry Pi
[2] 448.016

schedule

hardware equipment - pcs

12 workplaces are available in the laboratory

- Windows Host PCs with Virtualbox 4.2.18
- virtual Linux Guest Xubuntu 12.04 32 bit

hardware equipment - rpis

12 RPi kits (labelled)

- Raspberry PI with Case
- FTDI UART-USB-Converter with 4-pin-jack and Switch
  - Switch resets RPi
  - also responsible for power supply
- 4 GB Kingston microSD-Card with SD-Adapter
  - images can be sent to RPi via serial-cable
  - content of card must not be changed

additional hardware is given out dependant on the chosen group project

software equipment - virtual linux

- XUbuntu 12.04 LTS
- user: ikt, pw: rpi (same as workstation)
- user ikt is logged on automatically
- is already installed on workplaces
- can be downloaded from workstations desktop (5GB)
  - please do not change exported appliances on desktop
  - virtualbox 4.3 is not compatible, use previous version 4.2.18 instead or extract virtualdisk-file

software equipment - virtual linux, packages

- ARMv6 toolchain and other build tools - crosscompile binaries for ARM
- raspbootin - send kernel-images via serial cable
- svn with rapidsvn-client
- lightweight IDE geany with preconfigured build targets
- dropdown-terminal (F12)
- ...

template

virtual linux contains template for basic program

- template-origin: baking pi [3]
- baking pi very good resource for assembler beginners
- contains linker-/make-/assembler-skeleton-file

[3] www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/

template: linker - file "kernel.ld"

- is responsible for correct appearance of binary
- can organize labels (jumpmarks) as sections

several sections (changes normally not necessary)

- if only assembler commands are linked less sections are needed
- c-files need more sections (.bss, text.startup)
- $\rightarrow$ always use latter

template: Makefile

ensures correct compiling

- make: produces binaryfiles and links them to "kernel.img" (that file should be executed on RPi)
- make clean: delete object-files and binaries

notes about the Makefile:

- if different toolchain is used $\rightarrow$ change variable PREFIX/ARMGNU
- codefiles belong in folder *source*
- buildfiles are created in folder *build*

template: sucessful build

created files after successful build

- kernel.img
  - target image-file
- kernel.list/kernel.map disassembly of binary
  - sections, assembler-commands, labels, ...

raspbootin/raspbootcom - introduction

traditional workflow of RPi

- create image-file kernel.img
- (prepare sd-card with correct bootfiles)
- exchange old kernel.img with new kernel.img
- insert sd-card

$\rightarrow$ poor sd-card!

$\rightarrow$ solution: send image via uart/usb

raspbootin/raspbootcom - introduction contd.

uses client-server architecture

- raspbootin: is placed on RPi-Side and waits for request
- raspbootcom: is executed on virtual linux and sends image to RPi

raspbootin/raspbootcom - workflow raspbootin

raspbootin workflow

- prepare sd card only once with raspbootin-kernel (already done)
- connect GPIO-pins UART-USB-Converter ("FTDI")

raspbootin/raspbootcom - workflow raspbootcom

raspbootcom workflow

- make kernel-image
- ensure that virtual linux can access FTDI (/dev/ttyUSB0) by adding it to virtualbox-devices list
- execute following command

```
sudo ./raspbootcom /dev/ttyUSB0 kernel.img
```

- restart RPi via switch on cable (transmission starts after restart)

geany - introduction

why geany?

- small and lightweight editor
- already contains some make-commands
- must not be used $\rightarrow$ sudo apt-get install gedit/vim/...

geany - make commands

can be accessed via *Build*

- make / make clean (modify working-directory)
- typical order of make commands:
  1. (make clean)
  2. make
  3. open terminal and start raspbootcom
- use projects in order to create new make commands for different work-directories

rapidsvn

- submissions always per svn!
- repositories already created for you
- svn could be controlled via console but also via already installed rapidsvn [4]
  - checkout once
  - add files
  - commit files if changed
  - update files if partner changed them

[4] `http://tinyurl.com/k3nluq9`

example 1: let the led blink - ASS

task

- use the led on the GPIO16
- let it blink repeatedly (around 10 seonds intervall)
- use assembler
- developer: YOU - start up your virtual machines ;D

example 1: start virtual machine

- start virtualbox
- (once) add FTDI to attached devices (usb device will be accessible by virtual machine)
  - Settings $\rightarrow$ USB $\rightarrow$ PLUS-Symbol $\rightarrow$ FTDI TTL232R-3V3 [0600]
- after git-checkout skeleton files for this exercise are in /home/ikt/WORK/__beginner_tutorial

examples preparation: clone git

first: please follow these instructions

- open /home/ikt/WORK
- delete the content of this folder (old version of tutorial)
- open terminal (via right mouseclick)
- download skeleton source files (don't forget fullstop)

```
~/WORK$ git clone http://tinyurl.com/ikt-2013-git .
```

- now open __beginner_tutorial_ASS/source/init.s

example 1: basic structure of ass-file

- entry point at the begin

```
.section .init
.globl _start
_start:
```

- infinite loop at the end

```
loop:
 b loop
```

- in our case: infinite loop will be the blinking of the led

```
loop:
 <LED-FUN>
 b loop
```

example 1: set LED as output

we would like to use the GPIO-pin which controls the LED (GPIO pin 16)
SoC-Peripherals.pdf:

example 1: set LED as output

we would like to use the GPIO-pin which controls the LED (GPIO pin 16)
SoC-Peripherals.pdf:

- page 89: there are 54 GPIO Pins ... each one has at least two alternative functions ... details are given in section 6.2 (page 102)
- we only need to set the GPIO as output $\rightarrow$ page 91

example 1: set LED as output

we would like to use the GPIO-pin which controls the LED (GPIO pin 16)
SoC-Peripherals.pdf:

- page 89: there are 54 GPIO Pins ... each one has at least two alternative functions ... details are given in section 6.2 (page 102)
- we only need to set the GPIO as output $\rightarrow$ page 91
- page 91: "The function select registers (FSEL) are used to define the operation of the general-purpose pins

example 1: set LED as output

we would like to use the GPIO-pin which controls the LED (GPIO pin 16)
SoC-Peripherals.pdf:

- page 89: there are 54 GPIO Pins ... each one has at least two alternative functions ... details are given in section 6.2 (page 102)
- we only need to set the GPIO as output $\rightarrow$ page 91
- page 91: "The function select registers (FSEL) are used to define the operation of the general-purpose pins
- page 91: there are 6 GPFSELs. Each of them is a 32 bit register and contains 10 FSEL – except for the last one (only 54 pins, 0-53)

example 1: set LED as output

we would like to use the GPIO-pin which controls the LED (GPIO pin 16)
SoC-Peripherals.pdf:

- page 89: there are 54 GPIO Pins ... each one has at least two alternative functions ... details are given in section 6.2 (page 102)
- we only need to set the GPIO as output → page 91
- page 91: "The function select registers (FSEL) are used to define the operation of the general-purpose pins
- page 91: there are 6 GPFSELs. Each of them is a 32 bit register and contains 10 FSEL – except for the last one (only 54 pins, 0-53)
- page 92: 3 bits form one FSEL: 001 means output

example 1: set LED as output

we would like to use the GPIO-pin which controls the LED (GPIO pin 16)
SoC-Peripherals.pdf:

- page 89: there are 54 GPIO Pins ... each one has at least two alternative functions ... details are given in section 6.2 (page 102)
- we only need to set the GPIO as output → page 91
- page 91: "The function select registers (FSEL) are used to define the operation of the general-purpose pins
- page 91: there are 6 GPFSELs. Each of them is a 32 bit register and contains 10 FSEL – except for the last one (only 54 pins, 0-53)
- page 92: 3 bits form one FSEL: 001 means output
- → new subtask: write 001 to the correct FSEL

example 1: set LED as output - FSEL

- page 92: FSEL16 is responsible for GPIO pin 16
- FSEL16 is on GPFSEL1
- inside GPFSEL1, FSEL16 is on bit 20-18
- if the first bit of FSEL16 must be set WHICH bit is that?
- HOW can bit 18 be set with a bit?

example 1: set LED as output - FSEL

- page 92: FSEL16 is responsible for GPIO pin 16
- FSEL16 is on GPFSEL1
- inside GPFSEL1, FSEL16 is on bit 20-18
- if the first bit of FSEL16 must be set WHICH bit is that?
- HOW can bit 18 be set with a bit?
- $\rightarrow$ logical left shift

example 1: set LED as output - FSEL

- page 92: FSEL16 is responsible for GPIO pin 16
- FSEL16 is on GPFSEL1
- inside GPFSEL1, FSEL16 is on bit 20-18
- if the first bit of FSEL16 must be set WHICH bit is that?
- HOW can bit 18 be set with a bit?
- $\rightarrow$ logical left shift
- WHERE is GPFSEL0 and GPFSEL1 located?

example 1: set LED as output - FSEL

- page 92: FSEL16 is responsible for GPIO pin 16
- FSEL16 is on GPFSEL1
- inside GPFSEL1, FSEL16 is on bit 20-18
- if the first bit of FSEL16 must be set WHICH bit is that?
- HOW can bit 18 be set with a bit?
- $\rightarrow$ logical left shift
- WHERE is GPFSEL0 and GPFSEL1 located?
- $\rightarrow$0x20200000, 0x20200004

example 1: set LED as output - FSEL

- page 92: FSEL16 is responsible for GPIO pin 16
- FSEL16 is on GPFSEL1
- inside GPFSEL1, FSEL16 is on bit 20-18
- if the first bit of FSEL16 must be set WHICH bit is that?
- HOW can bit 18 be set with a bit?
- $\rightarrow$ logical left shift
- WHERE is GPFSEL0 and GPFSEL1 located?
- $\rightarrow$0x20200000, 0x20200004
- HOW can this be used in order to write an API for accessing the bits?

example 1: set LED as output - FSEL

- page 92: FSEL16 is responsible for GPIO pin 16
- FSEL16 is on GPFSEL1
- inside GPFSEL1, FSEL16 is on bit 20-18
- if the first bit of FSEL16 must be set WHICH bit is that?
- HOW can bit 18 be set with a bit?
- $\rightarrow$ logical left shift
- WHERE is GPFSEL0 and GPFSEL1 located?
- $\rightarrow$0x20200000, 0x20200004
- HOW can this be used in order to write an API for accessing the bits?
- $\rightarrow$ automatic calculation of correct bits

example 1: set LED as output - CODE

```
ldr r0,=0x20200000 @store GPFSEL0 adress
mov r1,#1 @store immediate 1 in r1
lsl r1,#18 @logical left shift r1 18 times
str r1,[r0,#4] @store value inside GPFSEL1, (offset +4)
```

example 1: toggle LED

- led is switched on if its output value is set to zero
- GPCLR0 and GPCLR1 is responsible for clearing
- GPSET0 and GPSET1 is responsible for setting

example 1: toggle LED

- led is switched on if its output value is set to zero
- GPCLR0 and GPCLR1 is responsible for clearing
- GPSET0 and GPSET1 is responsible for setting
- WHAT adresses do this registers have?

example 1: toggle LED

- led is switched on if its output value is set to zero
- GPCLR0 and GPCLR1 is responsible for clearing
- GPSET0 and GPSET1 is responsible for setting
- WHAT adresses do this registers have?
- WHAT is the offset from first GPIO-adress?

example 1: toggle LED

- led is switched on if its output value is set to zero
- GPCLR0 and GPCLR1 is responsible for clearing
- GPSET0 and GPSET1 is responsible for setting
- WHAT adresses do this registers have?
- WHAT is the offset from first GPIO-adress?
- - GPCLR0: 0x28 = (dec)40
  - GPSET0: 0x1c = (dec)28

example 1: toggle LED - CODE

```
@ SWITCH ON LED
mov r1,#1
lsl r1,#16
str r1,[r0,#40]
@ SWITCH OFF LED
str r1,[r0,#28]
```

put this code inside your loop

example 1: toggle LED - CODE

```
@ SWITCH ON LED
mov r1,#1
lsl r1,#16
str r1,[r0,#40]
@ SWITCH OFF LED
str r1,[r0,#28]
```

put this code inside your loop

- WILL THIS CODE WORK?

example 1: toggle LED - CODE

```
@ SWITCH ON LED
mov r1,#1
lsl r1,#16
str r1,[r0,#40]
@ SWITCH OFF LED
str r1,[r0,#28]
```

put this code inside your loop

- WILL THIS CODE WORK?
- YES but one will not see much $\rightarrow$ we need delay

example 1: delay togglilng

```
@DELAY TOGGLE
mov r2,#0x3F0000 @about ten seconds
wait1$:
sub r2,#1 @decrease r2
cmp r2,#0 @compare r2 with zero
bne wait1$ @branch to wait1 if not equal
```

use two of those loops in order to delay the toggles

example 1: fin

questions/problems?

example 2: let the led blink - C

now use C instead of ARMv6 Assembler

example2: preparation, 1

- change to new folder __beginner_tutorial_C
  - Makefile was already able to compile c-files
  - linker-file now takes care of additional c-sections.
  - init.s only consists out of init-commands and main-branch-and-link
  - main.c will hold the important code now

example2: preparation, 2

- init.s branches and links to main-function

```
bl  main
```

- if main-function returns init.s must continue.

```
loop_inf:
b  loop_inf
```

hints on c-coding on RPi, 1

- use `int32_t` for integers
- shift is done by bit operation << or >>

```
int32_t val = 1 << 2
```

- use DEFINES – not hardcoded values

```
#define GPIO_FSEL_ADDR 0x20200000
#define GPIO_PIN_LED_ACT 16
```

- `volatile` variables are not optimized by gcc (needed for busy wait)

```
int32_t volatile wait = 0;
```

hints on c-coding on RPi, 2

- change value in memory (STR) via following command

```
(* (volatile uint32_t * ) (ADDR) ) = value
```

- even better by using following define :D

```
#define mmio32(x)    (*(volatile uint32_t *)(x))
```

- C has no real No Operation - Command (NOP) – the compiler will optimize (delete) loops which are only waiting with pseudo NOPs (";", "var=var",...)
- write single ASM-commands in c-code

```
__asm__("NOP");
```

- multiple ASM-commands

```
__asm__("mov r2,#0x3F0000\n\t"
  "wait_first$:"    );
```

hints on c-coding on RPi, 3

some good links:

- OReilly Programming Embedded Systems Second edition [5]
- `http://www4.in.tum.de/~blanchet/api-design.pdf`
- programming API in C [6]
- `http://www.slideshare.net/alexandruc/how-to-define-an-api`
- `http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/`

[5] http://tinyurl.com/m6alk8d

[6] `http://www.le.ac.uk/eg/mjp9/pes1ohp_a4.pdf`

thank you for your attention!

problems/*questions*?