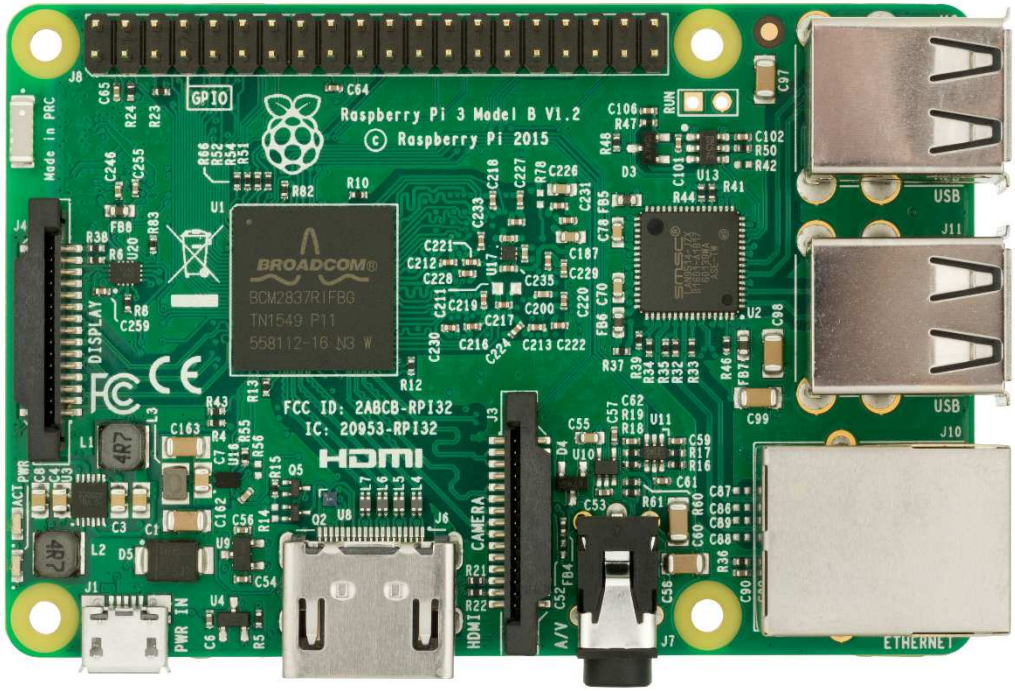
Einstieg in ein ARM-Entwicklungsboard

In diesem Versuch wird ein Raspberry Pi 3 ARM-Entwicklungsboard mit dem ARM Cortex-A53 Prozessor BCM2837 von Broadcom mit diverser Peripherie vorgestellt.

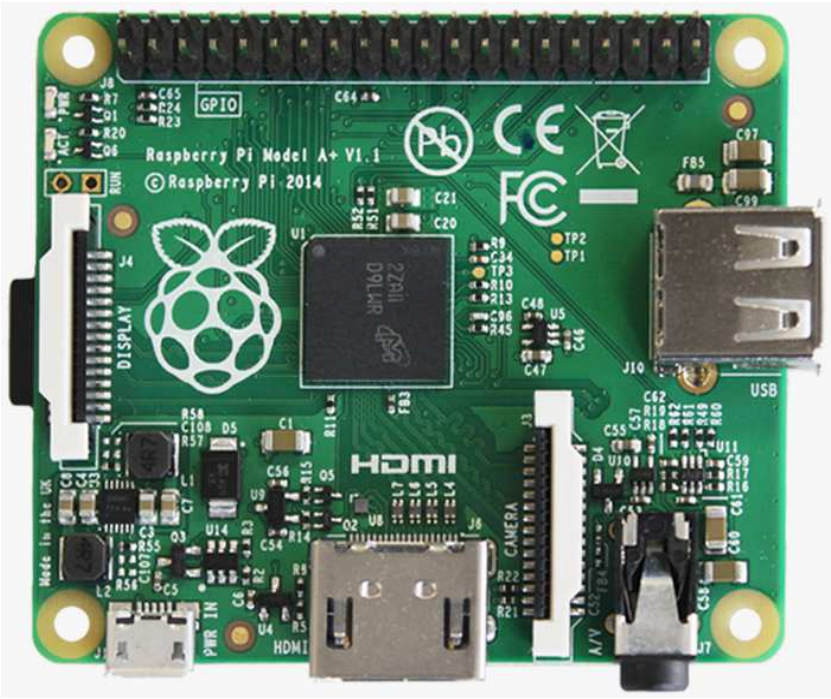
- 1 GB RAM
- kein Flash: SD-Karte wird genutzt
- GPIO Pins: ähnlich einem Mikrocontroller
- UART / SPI / I2C
- Timer / PWM
- USB
- Interrupt / DMA Controller

Hinweis: im Vergleich zu einem Mikrocontroller ist die Stromaufnahme mit max. 2.5 A mit Linux-Betriebssystem sehr hoch. Für stromsparende Embedded Anwendungen wird daher hauptsächlich das Raspberry Pi 1 mit einem Echtzeit-Betriebssystem mit max. 0.5 A genutzt.

GPIO



GPIO



Raspberry Pi 3
BCM2837 ARM Cortex-A53 32/64-Bit

Raspberry Pi 1
BCM2835 ARM11 32-Bit

Echtzeit-Betriebssystem

Ein Echtzeit-Betriebssystem (real-time operating system / RTOS) ist ein Betriebssystem, das in der Lage ist, Echtzeit-Anforderungen zu erfüllen d.h. das Garantieren von Verzögerungs- und Bearbeitungszeiten.

Die Verwendung eines Raspberry Pi als direkten Ersatz für ein Mikrocontroller-Board erfordert Single-Task und RESET ohne vorherigen Shutdown

Auswahl von RISCOS pico mit Speicherbedarf im Flash von nur 2 MB

https://www.riscosopen.org/content/downloads/raspberry-pi

Die Single-Task Programmierung in RISCOS pico erfolgt standardmäßig in der BASIC Shell

Eine Programmierung in C wird hingegen üblicherweise auf einem RISCOS Desktop System durchgeführt (mit ARM DDE oder GCC) und dann die sich ergebende HEX-Datei oder Binär-Datei mit einem Bootloader auf das RISCOS pico System geladen.

Start

Nach dem Einschalten ist das System in der BASIC Shell – zu erkennen an der Eingabeaufforderung >

```
RISC OS
Cortex-A53 Processor
ARM BBC BASIC
>
```

Skript-Sprachen:
BASIC / Python

Zur erstmaligen Konfiguration des Systems starten Sie das Konfigurationsprogramm wie folgt:

```
> LOAD "Config"
> RUN
```

Information • das Konfigurationsprogramm enthält folgendes:

```
MODE 800,600,32       // Monitor-Auflösung
COLOUR 75             // Text YELLOW
CLS
Keyboard Germany      // Deutsche Tastatur
Dir                   // Root-Verzeichnis
Dir Projects          // Projektverzeichnis
Cat                   // Dateien zeigen
```

Hinweis: wenn Sie zu einem späteren Zeitpunkt die Dateien erneut zeigen möchten, geben Sie folgenden OS Shell Befehl ein:

```
> *Cat
```

oder wechseln Sie dafür direkt in die OS Shell * und dann zurück nach BASIC

```
> QUIT
* Cat
* BASIC
```

1. Einführung in BASIC

a) Laden Sie das Programm „1_BASIC“ mit

```
> LOAD "1_BASIC"
```

(oder auch abgekürzt mit)

```
> LOAD "1*"
```

und zeigen Sie das Programm an mit

```
> LIST
```

Ergebnis:

```
1  REPEAT
2    PRINT "Hello"
3  UNTIL FALSE
```

b) Führen Sie das Programm aus mit

```
> RUN
```

c) Unterbrechen Sie das Programm mit der „Escape“ Taste

d) Ändern Sie eine Zeile: hierzu wird entweder die Zeile mit der Zeilennummer komplett neu eingegeben

```
> 2  PRINT "Hello World"
> LIST
> RUN
```

oder der Editor genutzt:

BASIC Editor

```
> EDIT

Shift-F4 : SAVE and EXIT
F3       : SAVE under changed program name
F2       : LOAD other existing program
```

e) Speichern Sie das geänderte Programm als Binärdatei

```
> SAVE "1_BASIC_NEW"
```

(oder als Textdatei)

```
> TEXTSAVE "1_BASIC_NEW/txt"
```

Die Textdatei können Sie dann auch auf einen USB-Stick kopieren:

Dateien von SD-Karte auf USB-Stick kopieren:

```
* Copy Filename/txt SCSI::0.$.*
```

2. Einführung in ARM Cortex A 32-Bit Assembler

a) Arithmetik: Laden Sie das Programm „2a_ASM“ und führen Sie es aus

```
DIM CODE% (&100)
P% = CODE%
[
  MOV   R0, #2
  ADD   R0, R0, #1

  MOV   PC, LR  ; return
]
PRINT "R0 = " USR(CODE%)
```

Programmgröße im RAM 256 Byte

Hinweis:
Binärzahl %1010 oder 1010B
Hexadezimalzahl &FFFF oder 0FFFFH

Rückgabewert in R0

b) Ändern Sie das Programm so dass eine Subtraktion um 1 ausgeführt wird.

c) Logik: Laden Sie das Programm „2b_ASM“und führen Sie es aus

```
…
  MOV   R0, #%111
  BIC   R0, R0, #%010
…
```

d) Ändern Sie das Programm so dass stattdessen Bit 0 gelöscht wird.

e) Testen Sie im Programm weitere Logik-Befehle wie AND / ORR / EOR

f) Laden Sie das Programm „2c_ASM“und führen Sie es aus

```
DIM CODE% (&100)
P% = CODE%
[
.string
  EQUS  "Hello"
  EQUB  0         ; string terminator NULL
  ALIGN           ; fill up last word (= 4 bytes)

.main
  ADR   R0, string    ; string address
  SWI   "OS_Write0"   ; print NULL terminated string
  SWI   "OS_NewLine"

  SWI   "OS_ReadEscapeState"  ; if ESC pressed break
  BCC   main

  MOV   PC, LR  ; return
]
CALL main
```

ARM A53 ist 32/64-Bit Prozessor: Programmzeile muss Vielfaches von 4 Bytes sein.

Nur hierdurch kann das Programm mit der „Escape“ Taste unterbrochen werden

Beispiel für einen Kernel-Aufruf:

```
OS_Write0(string-pointer) : string-pointer wird in Register R0 übergeben
```

Hinweis: wie bei einem Mikrocontroller-Board läuft auf dem Pi 3 im Single-Task ein Programm in einer Endlosschleife. Sollte sich das Programm nicht mehr unterbrechen lassen ist das Ziehen des Netzsteckers erforderlich, da das Pi 3 keinen RESET Taster hat.

Um das zu verhindern gibt es zwei Möglichkeiten:

- **Abfrage der „Escape“ Taste in der Endlosschleife**
- **Abfrage eines externen Tasters an einem Port Pin**

Bei Embedded Anwendungen sind i.d.R. weder Monitor noch Tastatur angeschlossen, somit bleibt dann nur ein externer Taster.

g) Ändern Sie die Ausgabe des Programms auf „Hello World“

3. Port Pins (GPIO)

Pin#	NAME		NAME	Pin#
01	3.3v DC Power		DC Power 5v	02
03	GPIO02 (SDA1 , I2C)		DC Power 5v	04
05	GPIO03 (SCL1 , I2C)		Ground	06
07	GPIO04 (GPIO_GCLK)		(TXD0) GPIO14	08
09	Ground		(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)		(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)		Ground	14
15	GPIO22 (GPIO_GEN3)		(GPIO_GEN4) GPIO23	16
17	3.3v DC Power		(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)		Ground	20
21	GPIO09 (SPI_MISO)		(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)		(SPI_CE0_N) GPIO08	24
25	Ground		(SPI_CE1_N) GPIO07	26
27	ID_SD (I2C ID EEPROM)		(I2C ID EEPROM) ID_SC	28
29	GPIO05		Ground	30
31	GPIO06		GPIO12	32
33	GPIO13		Ground	34
35	GPIO19		GPIO16	36
37	GPIO26		GPIO20	38
39	Ground		GPIO21	40

Am Pi 3 sind 40 Port Pins an einem Steckverbinder herausgeführt (dies sind allerdings nicht alle Port Pins)

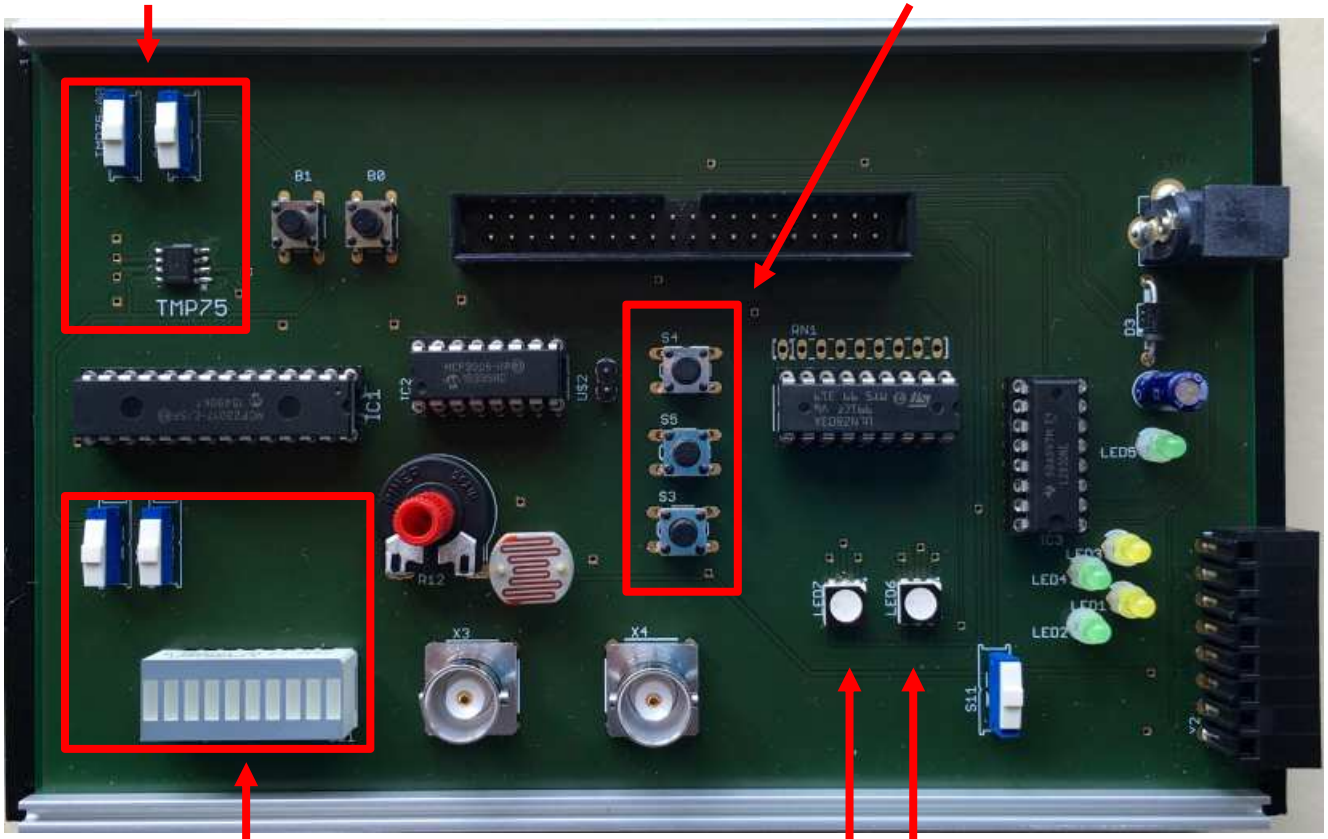
Vorsicht:

Leider wurde keine Schutz-elektronik verbaut, somit kann der Anschluss von >3.3V an einem Pin oder die Verbindung von 2 Pins (ohne 1k Widerstand) zur Zerstörung des Pi 3 führen.

a) Schließen Sie das Expansion Board an das Pi 3 an:

I2C TEMP + Address switches

Buttons S4 S5 S3



I2C LEDs 1-10 + Address switches

LED7-RGB

LED6-RGB

Port Pins:

	Pin
LED6-R	21
LED6-G	20
LED6-B	17
LED7-R	24
LED7-G	23
LED7-B	22
BUT-S4	4
BUT-S5	6
BUT-S3	13

Hochschule Ravensburg-Weingarten

b) Laden Sie das Programm „3a_LED1_on“

```
REMARK : set to output
SYS "GPIO_WriteMode", 21, %001

REMARK : switch on
SYS "GPIO_WriteData", 21, 1
```

REMARK = Kommentar

Es kann auch die Abkürzung REM verwendet werden. Allerdings nicht das aus Visual BASIC bekannte '

Die LED6-R wird eingeschaltet.

c) Ändern Sie das Programm so ab dass die LED6-R ausgeschaltet wird

d) Laden Sie das Programm „3c_Blinky“

```
PRINT "Blinky - To end press Escape"
LED1 = 21
toggle% = 0
REMARK : set to output
SYS "GPIO_WriteMode", LED1, %001
REPEAT
  SYS "GPIO_WriteData", LED1, toggle%
  toggle% = 1 - toggle%

  REMARK : delay
  timeact = TIME + 25
  REPEAT
  UNTIL TIME > timeact
UNTIL FALSE
```

e) Ändern Sie das Programm so ab dass LED6-R und LED7-R abwechselnd blinken. Hinweis: Musterlösung „3d_Blinky“

f) Laden Sie das Programm „3e_BUT“

```
PRINT "Press Button - To end press Escape"
BUT1 = 4
val% = 0
REMARK : set to input
SYS "GPIO_WriteMode", BUT1, %000
REPEAT
  SYS "GPIO_ReadData", BUT1 TO val%

  IF val% = 0 THEN
    PRINT "S4"
  ENDIF

  REMARK : delay for de-bounce
  timeact = TIME + 25
  REPEAT
  UNTIL TIME > timeact
UNTIL FALSE
```

Das Drücken des Tasters BUT-S4 wird angezeigt.

g) Ändern Sie das Programm so ab, dass das Drücken des Tasters BUT-S4 die LED6-R toggelt. Hinweis: Musterlösung „3f_BUT_TOGGLE“

4. Port Pins (GPIO) in ARM Cortex A 32-Bit Assembler

a) Laden Sie das Programm „4a_LED_ON_ASM“

```
DIM CODE% (&100)
P% = CODE%
[
.main
  ; LED to output
  MOV   R0, #21
  MOV   R1, #%001
  SWI   "GPIO_WriteMode"

  ; LED on = high
  MOV   R0, #21
  MOV   R1, #1
  SWI   "GPIO_WriteData"

  MOV   PC, LR  ; return
]
CALL main
```

Hinweis:	BASIC	Assembler / C
	SYS	SWI

Die LED6-R wird eingeschaltet.

Vergleichen Sie mit dem BASIC Program „3a_LED1_on“

b) Ändern Sie das Programm so ab dass die LED6-R ausgeschaltet wird

c) Laden Sie das Programm „4b_Blinky_ASM“

```
  ; function delay (parameter passed in R0)
.delay
  STMDB SP!, {R1}
  MOV   R1, #1
  MOV   R1, R1, LSL R0  ; do x cycles
.delay_loop
  SUBS  R1, R1, #1
  BNE   delay_loop
  LDMIA SP!, {R1}
  MOV   PC, LR          ; return

.main
  STMDB SP!, {LR}  ; push return address to stack

  ; LED to output
  MOV   R0, #21
  MOV   R1, #%001
  SWI   "GPIO_WriteMode"
.main_loop
  ; LED on = high
  MOV   R0, #21
  MOV   R1, #1
  SWI   "GPIO_WriteData"

  MOV   R0, #27
  BL    delay
```

Unterprogramm

Stack (descending)

Warum muss LR auf den Stack?

```
; LED off = low
MOV   R0, #21
MOV   R1, #0
SWI   "GPIO_WriteData"

MOV   R0, #27
BL    delay

SWI   "OS_ReadEscapeState"  ; if ESC pressed then break
BCC   main_loop

LDMIA SP!, {PC}  ; pop return address from stack (PC = LR return)
```

Wie funktioniert das RETURN?

d) Lassen Sie die LED6-R langsamer und schneller blinken

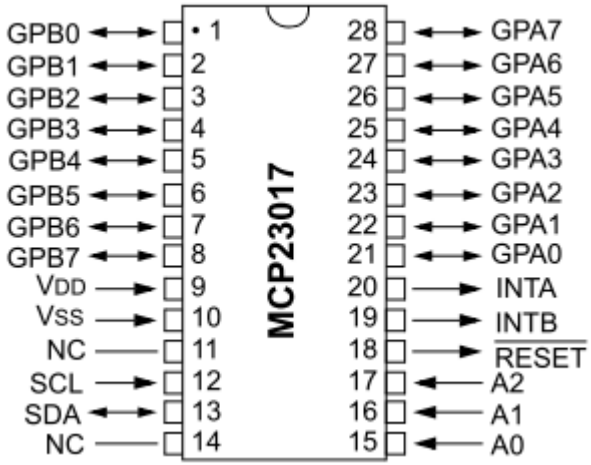
e) Schreiben Sie ein Assembler-Programm, das die LED6-R einschaltet und durch Drücken des Tasters BUT-S4 ausschaltet.
Hinweis: Musterlösung „4c_BUT_ASM“

Hinweis: zur Funktion von I2C siehe Versuch 3

5. Port-Expander (I2C)

Ein Port Expander bietet zusätzliche GPIO Pins mit einfachem Zugriff:

I2C Port Expander MCP23017



Address IOCON.BANK = 1	Address IOCON.BANK = 0	Access to:
00h	00h	IODIRA
10h	01h	IODIRB
09h	12h	GPIOA
19h	13h	GPIOB

Der LED-Bar mit den LEDs 1-10 ist an die Port-Pins PA.0-PA.7 und PB.0-PB.1 angeschlossen.

a) Laden Sie das Programm „5a_I2C_LED_ON“

```
REMARK : I2C buffer (byte 3, byte 2, byte 1, byte 0)
DIM buffer% 3

REMARK : port register 0=0 (port=output) : buffer = &----0000
?buffer% = 0
?(buffer%+1) = 0
REMARK : port address &20 + write : 2 bytes
SYS "IIC_Control", (&20<<1), buffer%, 2

REMARK : port register &12=1 (portbit0=1) : buffer = &----0112
?buffer% = &12
?(buffer%+1) = 1
REMARK : port address &20 + write : 2 bytes
SYS "IIC_Control", (&20<<1), buffer%, 2
```

Buffergröße 4 byte: 3…0

Falls Adresse geändert:
***i2cdetect nutzen !**

I2C Kernel-Aufruf:

```
IIC_Control(I2C-slave-address, buffer-pointer, buffer-size)
```

Hinweise:	BASIC	Assembler / C
	buffer%	unsigned char *buffer
	?(buffer%+1)	*(buffer+1)

Die LED 1 wird eingeschaltet.

b) Erklären Sie die Funktion des Programms

c) Ändern Sie das Programm so ab dass die LED 1 ausgeschaltet wird

d) Ändern Sie das Programm so ab dass:

- alle LEDs 1-8 leuchten
- alle ungeraden LEDs leuchten
- alle geraden LEDs leuchten

e) Wie müsste das Programm ergänzt werden, damit auch die LEDs 9 und 10 leuchten? Hinweis: Port B

f) Laden Sie das Programm „5c_I2C_LED_ON_ASM“

```
.buffer
  EQUD  &00000000  ; I2C buffer (byte 3, byte 2, byte 1, byte 0)

.main
  ; port address &20 + write 2 bytes
  ; port register 0=0 (port=output)
  MOV   R0, #(&20<<1)  ; value %0100000.0
  ADR   R1, buffer     ; value &00000000
  MOV   R2, #2
  SWI   "IIC_Control"

  ; port register &12
  ADR   R1, buffer
  MOV   R0, #&12
  STRB  R0, [R1]       ; access byte 0
  ; port register &12=1 (portbit0=1)
  MOV   R0, #1
  STRB  R0, [R1, #1]   ; access byte 1
  ; port address &20 + write 2 bytes
  MOV   R0, #(&20<<1)  ; value %0100000.0
  ADR   R1, buffer     ; value &00000112
  MOV   R2, #2
  SWI   "IIC_Control"

  MOV   PC, LR  ; return
```

g) Ändern Sie das Programm so ab dass die LED 1 ausgeschaltet wird

h) Ändern Sie das Programm so ab dass:

- alle LEDs 1-8 leuchten
- alle ungeraden LEDs leuchten
- alle geraden LEDs leuchten

6. Port-Expander (I2C) • Vertiefung

a) Laden Sie das Programm „6a_I2C_LED_RUN“

```
DIM buffer% 3

REMARK : port register 0=0 (port=output) : buffer = &----0000
?buffer% = 0
?(buffer%+1) = 0
REMARK : port address &20 + write : 2 bytes
SYS "IIC_Control", (&20<<1), buffer%, 2

REPEAT
  REMARK : loop all 8 LEDs
  index = 7
  REPEAT
    REMARK : port register &12=x (portbit0=1) : buffer = &----xx12
    ?buffer% = &12
    ?(buffer%+1) = (1<<index)
    REMARK : port address &20 + write : 2 bytes
    SYS "IIC_Control", (&20<<1), buffer%, 2

    REMARK : delay
    timeact = TIME + 40
    REPEAT
    UNTIL TIME > timeact

    index = index - 1
  UNTIL index < 0
UNTIL FALSE
```

b) Erklären Sie die Funktion des Programms

c) Laden Sie das Programm „6b_I2C_LED_RUN_ASM“

```
.buffer
  EQUD  &00000000  ; I2C buffer (byte 3, byte 2, byte 1, byte 0)

  …

.main
  STMDB SP!, {LR}  ; push return address to stack

  ; port address &20 + write 2 bytes
  ; port register 0=0 (port=output)
  MOV   R0, #(&20<<1)        ; value %0100000.0
  ADR   R1, buffer
  MOV   R2, #2
  SWI   "IIC_Control"

  ; port register &12
  MOV   R0, #&12
  ADR   R1, buffer
  STRB  R0, [R1]             ; access byte 0

  MOV   R3, #1               ; portbit0
  MOV   R4, #7               ; loop all 8 LEDs
```

```
.main_loop
  ; port address &20 + write 2 bytes
  ; port register &12 (portbitx=1)
  MOV   R0, #(&20<<1)        ; value %0100000.0
  ADR   R1, buffer
  MOV   R2, R3, LSL R4       ; set bit no. R4
  STRB  R2, [R1, #1]         ; access byte 1
  MOV   R2, #2
  SWI   "IIC_Control"

  MOV   R0, #26
  BL    delay    ; overwrites return address

  SUBS  R4, R4, #1           ; until last LED
  MOVMI R4, #7               ; if < 0 loop all 8 LEDs again

  SWI   "OS_ReadEscapeState"  ; if ESC pressed then break
  BCC   main_loop

  LDMIA SP!, {PC}  ; pop return address from stack (PC = LR return)
```

d) Erklären Sie die Funktion des Programms

7. Temperatursensor (I2C)

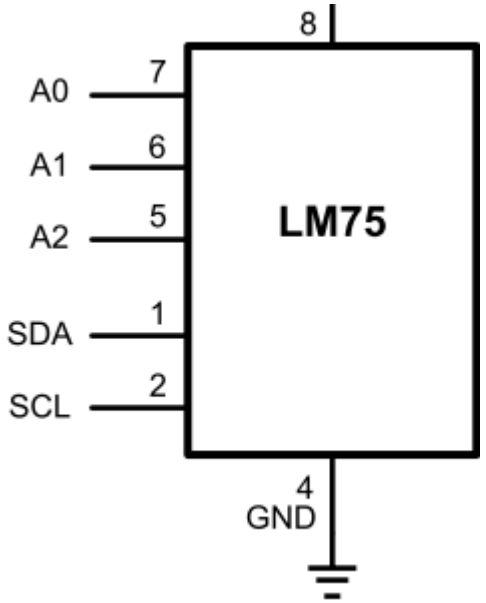
Der verwendete LM75 hat eine Auflösung von 0.5°C bzw. 9 Bit:

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MSB	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	LSB	X	X	X	X	X	X	X

a) Laden Sie das Programm „7a_I2C_TEMP“

```
DIM REMARK : I2C buffer (byte 3, byte 2, byte 1, byte 0)
DIM buffer% 3

REPEAT
  REMARK : temp address &48 + read : 2 bytes
  SYS "IIC_Control", (&48<<1)+1, buffer%, 2
  REMARK : print temperature
  PRINT "TEMP = " ?buffer% + (?(buffer%+1))/256

  REMARK : delay
  timeact = TIME + 40
  REPEAT
  UNTIL TIME > timeact
UNTIL FALSE
```

***i2cdetect nutzen !**



b) Erklären Sie die Funktion des Programms

c) Laden Sie das Programm „7b_I2C_TEMP_ASM“ und erklären Sie die Funktion des Programms

```
.buffer
  EQUD  &00000000  ; I2C buffer (byte 3, byte 2, byte 1, byte 0)
.string
  EQUD  &00000000  ; string of 3 chars and NULL terminator
  …
.main
  STMDB SP!, {LR}  ; push return address to stack
.main_loop
  ; temp address &48 + read 2 bytes
  MOV   R0, #((&48<<1)+1)      ; value %1001000.1
  ADR   R1, buffer
  MOV   R2, #2
  SWI   "IIC_Control"

  ; print temperature
  ADR   R3, buffer

  LDRB  R0, [R3]               ; read byte 0
  ADR   R1, string
  MOV   R2, #3                 ; convert 3 digits (0-100)
  SWI   "OS_ConvertCardinal1"
  SWI   "OS_Write0"
  LDRB  R0, [R3, #1]           ; read byte 1
  TSTS  R0, #%10000000         ; check bit 7
  MOV   R0, #ASC(".")
  SWI   "OS_WriteC"
```

Kernel-Aufruf (konvertiert unsigned char nach string):

```
OS_ConvertCardinal1(value, buffer-pointer, buffer-size):
string-pointer wird in R0 zurückgegeben
```

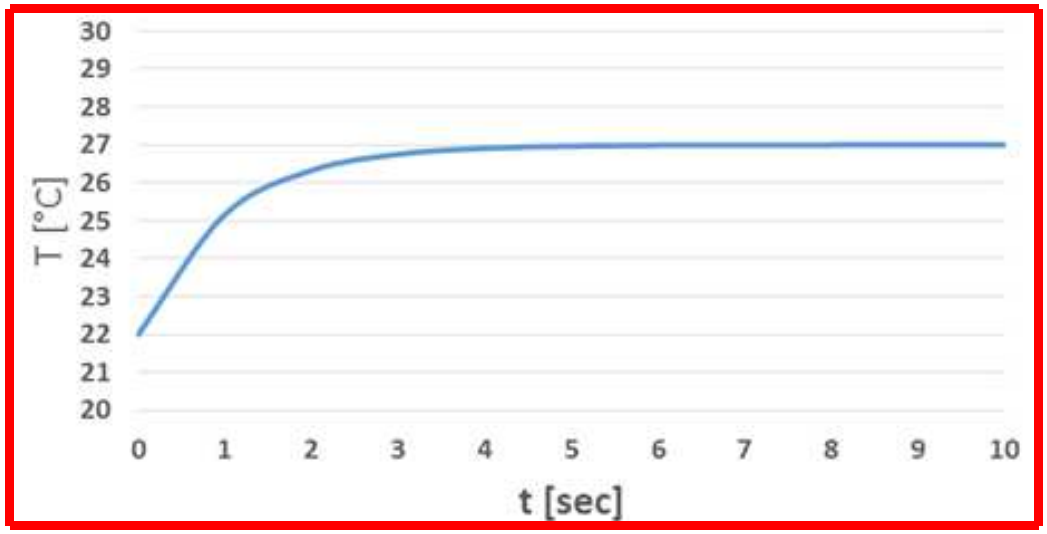
```
MOVEQ R0, #ASC("0")  ; .0
MOVNE R0, #ASC("5")  ; .5
SWI   "OS_WriteC"
SWI   "OS_NewLine"

MOV   R0, #26
BL    delay

SWI   "OS_ReadEscapeState"
BCC   main_loop

LDMIA SP!, {PC}
```

d) Laden Sie das Programm „7c_I2C_TEMP_BAR“

e) Passen Sie das Programm an die aktuelle Raumtemperatur an

f) Laden Sie das Programm „7d_I2C_TEMP_BAR_ASM“

g) Laden Sie das Programm „7e_I2C_TEMP_GRAPH“



8. Port Pins (GPIO) Fast Access in ARM A 32-Bit Assembler

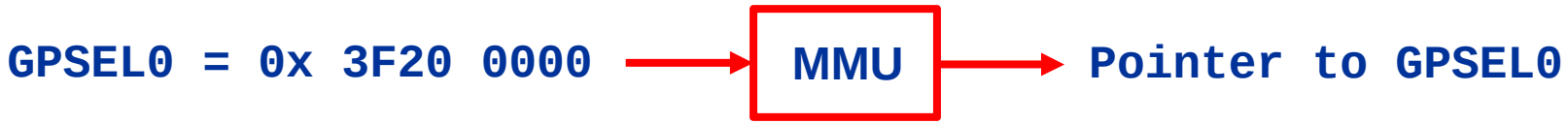
Bisher erfolgte der Zugriff auf die GPIO mit Kernel-Aufruf. Dadurch kann es zu geringen Latenzen kommen, die für harte Echtzeit u.U. nicht akzeptabel sind.

Alternativ kann daher wie bei einem Mikrocontroller direkt auf die GPIO-Register zugegriffen werden – unter Umgehung des Betriebssystems:
Bare Metal Programming

Register	Bus address	Physical address Pi 1	Physical address Pi 3
GPSEL0	0x 7E20 0000	0x 2020 0000	0x 3F20 0000
GPSEL1	0x 7E20 0004	0x 2020 0004	0x 3F20 0004
GPSEL2	0x 7E20 0008	0x 2020 0008	0x 3F20 0008
GPSET0	0x 7E20 001C	0x 2020 001C	0x 3F20 001C
GPCLR0	0x 7E20 0028	0x 2020 0028	0x 3F20 0028
GPLEV0	0x 7E20 0034	0x 2020 0034	0x 3F20 0034

Der direkte Zugriff (über Bus-Adresse) ist aber nur von der Firmware (Bootloader) aus möglich: daher Zugriff über MMU (Memory Management Unit)

MMU Kernel-Aufruf auf dem Pi 3:




```
OS_Memory(action, physical-address, block-size, returned logical-pointer) :

Pointer auf logischen Speicherblock wird in R3 zurückgegeben
```

Port Pins:

	Pin	Pin Mode	Mode Bits	Set Output	Clear Output	Input
LED6-R	21	GPSEL2	5: 3	GPSET0	GPCLR0	-
LED6-G	20	GPSEL2	2: 0	GPSET0	GPCLR0	-
LED6-B	17	GPSEL1	23:21	GPSET0	GPCLR0	-
LED7-R	24	GPSEL2	14:12	GPSET0	GPCLR0	-
LED7-G	23	GPSEL2	11: 9	GPSET0	GPCLR0	-
LED7-B	22	GPSEL2	8: 6	GPSET0	GPCLR0	-
BUT-S4	4	GPSEL0	14:12	-	-	GPLEV0
BUT-S5	6	GPSEL0	20:18	-	-	GPLEV0
BUT-S3	13	GPSEL1	11: 9	-	-	GPLEV0

a) Laden Sie das Programm „8a_LED_ON_ASM“

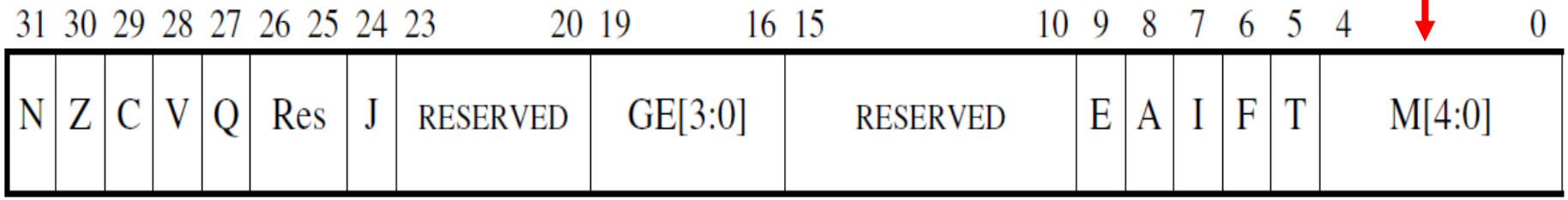
```
.gpiobase
  EQUD  &3F200000  ; GPIO base address on Pi 3

.main
  ; LED6 R=21 on GPIO pin 21 is GPSEL2 bits 5-3, GPSET0, GPCLR0

  ; protected memory access
  MOV   R0, #&0D      ; physical to logical address map (permanent)
  LDR   R1, gpiobase  ; physical address
  MOV   R2, #&100     ; logical address space size 256 bytes
  SWI   "OS_Memory"   ; MMU
  MOV   R12, R3       ; returned logical address pointer

  ; LED to output
  SWI   "OS_EnterOS"          ; supervisor mode
  LDR   R0, [R12, #&08]       ; load GPSEL2 (offset &08 to gpio base address)
  BIC   R0, R0, #(%111<<3)    ; clear bits 5-3
  ORR   R0, R0, #(%001<<3)    ; set bits 5-3 to %001
  STR   R0, [R12, #&08]       ; store GPSEL2

  ; LED on = high
  MOV   R0, #1
  MOV   R0, R0, LSL #21  ; bit 21
  STR   R0, [R12, #&1C]  ; store GPSET0 (offset &1C to gpio base address)
  SWI   "OS_LeaveOS"     ; user mode

  MOV   PC, LR  ; return
```

Hardware-Zugriff nur im ARM Supervisor Mode

Hinweis: Der Single-Task läuft im User Mode. Hardware-Zugriff ist aber nur im Supervisor Mode möglich.

Mode-Wechsel durch Processor Status Register (PSR)

```
PSR |= %10011
```



ist nicht möglich (warum ?) daher Dummy Software-Interrupt (SWI):

```
OS_EnterOS()
```

b) Ändern Sie das Programm so ab dass anstatt LED6-R nun LED7-R genutzt wird. Hinweis: Musterlösung „8b_LED_ON_ASM“

c) Laden Sie das Programm „8c_BUT_ASM“

```
.main
  ; S4=4 on GPIO pin 4
  …
.main_loop
  SWI   "OS_ReadEscapeState"  ; if ESC pressed
  MOVCS PC, LR                ; return

  ; check BUT
  SWI   "OS_EnterOS"      ; supervisor mode
  MOV   R0, #1
  MOV   R0, R0, LSL #4    ; bit 4
  LDR   R1, [R12, #&34]   ; load GPLEV0 (offset &34 to gpio base address)
  SWI   "OS_LeaveOS"      ; user mode

  TSTS  R1, R0            ; check BUT
  BNE   main_loop

  ; LED off = low
  SWI   "OS_EnterOS"      ; supervisor mode
  MOV   R0, #1
  MOV   R0, R0, LSL #21   ; bit 21
  STR   R0, [R12, #&28]   ; store GPCLR0 (offset &28 to gpio base address)
  SWI   "OS_LeaveOS"      ; user mode
  …
```

d) Ändern Sie das Programm so ab dass nun der Taster BUT-S5 genutzt wird.
Hinweis: Musterlösung „8d_BUT_ASM“

9. Timer

Wie auch bei einem Mikrocontroller ist die exakte Zeitmessung durch ein Delay nur unzureichend möglich. Dies soll zunächst gezeigt werden:

a) Laden Sie das Programm „9a_Blinky_ASM“ und versuchen Sie die LED6-R mit 0.5 sec blinken zu lassen

Für eine exakte Zeitmessung können Software-Timer des Betriebssystems oder Hardware-Timer genutzt werden:

b) Laden Sie das Programm „9b_Timer_ASM“ und lassen Sie die LED6-R langsamer und schneller blinken

```
.toggle
  EQUD  0  ; timer toggle

  ; timer irq handler, GPIO base address (logical) passed in R12
  ; CAUTION supervisor mode
.timer
  STMDB SP!, {R0-R1}  ; push to (supervisor) stack

  ADR   R1, toggle
  LDR   R0, [R1]
  RSBS  R0, R0, #1    ; toggle = 1 - toggle (and set flags)
  STR   R0, [R1]
```

Called on every ticker timer interrupt

SP_svc

```
  MOV   R0, #1
  MOV   R0, R0, LSL #21  ; bit 21
  ; toggle = 1 LED on = high
  MOVNE R1, #&1C         ; GPSET0 (offset &1C to gpio base address)
  ; toggle = 0 LED off = low
  MOVEQ R1, #&28         ; GPCLR0 (offset &28 to gpio base address)
  STR   R0, [R12, R1]    ; store GPSET0 or GPCLR0

  LDMIA SP!, {R0-R1}  ; pop from (supervisor) stack
  MOV   PC, LR        ; return

.main
  STMDB SP!, {LR}  ; push return address to stack
  BL    init       ; GPIO base address (logical) returned in R12
  …
  MOV   R0, #50          ; timer 0.5 sec
  ADR   R1, timer
  MOV   R2, R12          ; GPIO base address (logical)
  SWI   "OS_CallEvery"   ; timer start
.main_loop
  ; wait for irq
  SWI   "OS_ReadEscapeState"  ; if ESC pressed then break
  BCC   main_loop

  ADR   R0, timer
  MOV   R1, R12                  ; GPIO base address (logical)
  SWI   "OS_RemoveTickerEvent"   ; timer stop
  …
```

Warum haben die ARM Modes verschiedene Stacks?

SP_usr

Pass address of ticker timer interrupt handler

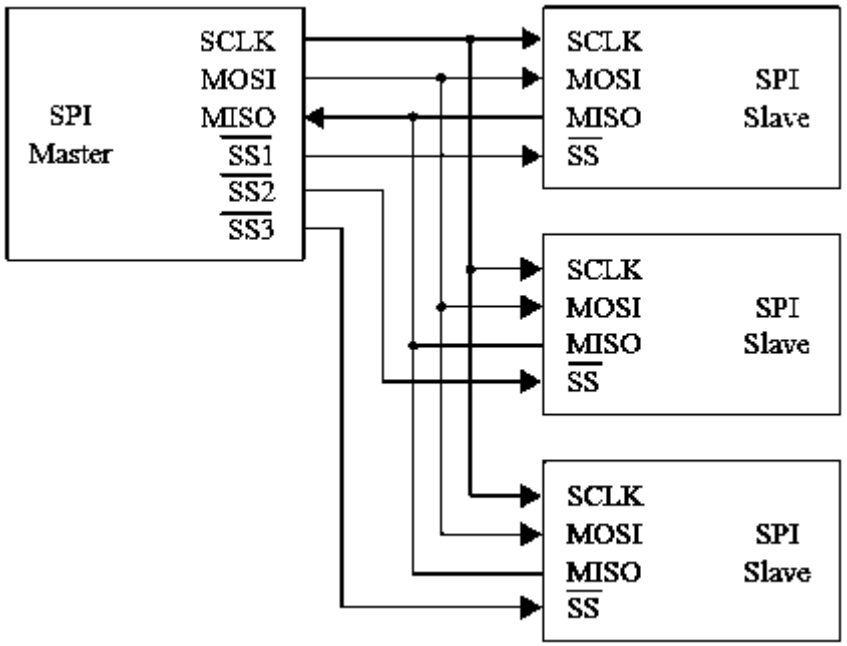
Do nothing – wait for interrupts

Was passiert ohne diesen Kernel-Aufruf?

Anhang

Serial Peripheral Interface (SPI)

Der Analog-Digital-Wandler (ADC) für Poti, Lichtsensor ist über SPI angebunden. SPI ist wie I2C eine synchrone, serielle Schnittstelle. SPI verwendet zum Ansprechen der Peripherie-Geräte allerdings keine Protokoll-Addressen, sondern für jedes Gerät eine Enable-Leitung. Dadurch ist die Latenz von SPI geringer als I2C



Testen Sie dazu das ADC-Beispiel für den verwendeten MCP3008 „A_ADC“

```
SYS "GPIO_WriteMode", CS  , %001
SYS "GPIO_WriteMode", MISO, %000
SYS "GPIO_WriteMode", MOSI, %001
SYS "GPIO_WriteMode", SCLK, %001

channel = 0
REPEAT
  SYS "GPIO_WriteData", CS  , HIGH
  SYS "GPIO_WriteData", CS  , LOW
```

channel = 0 : Poti
channel = 1 : Lichtsensor

```
  SYS "GPIO_WriteData", SCLK, LOW

  cmd = channel
  cmd = cmd OR %00011000
  FOR index = 1 TO 5
    IF (cmd AND %00010000) THEN
      SYS "GPIO_WriteData", MOSI, HIGH
    ELSE
      SYS "GPIO_WriteData", MOSI, LOW
    ENDIF
    SYS "GPIO_WriteData", SCLK, HIGH
    SYS "GPIO_WriteData", SCLK, LOW
    cmd = cmd << 1
  NEXT index

  value = 0
  bit = 0
  FOR index = 1 TO 11
    SYS "GPIO_WriteData", SCLK, HIGH
    SYS "GPIO_WriteData", SCLK, LOW
    value = value << 1
    SYS "GPIO_ReadData" , MISO TO bit
    IF (bit = HIGH) THEN
      value = value OR 1
    ENDIF
  NEXT index
  PRINT "Trimmer = " value
  …
UNTIL FALSE
```

ADC command 5-Bits

CLK falling edge

Return value 10-Bits and NULL bit