



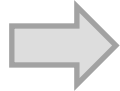
Microcontroller Programming (3)

Gerald Kupris, 22.10.2013

Lectures Microcontroller Programming WS2013/14

08.10.2013 Microcontroller, Programming and Debugging Interfaces

15.10.2013 Reading and Writing of Registers



22.10.2013 I/O-Pins, Reading and Writing of Single Bits

29.10.2013 Clock Generation, CPU und Computing Power

05.11.2013 Interrupts

12.11.2013 No lecture !

19.11.2013 Memory

26.11.2013 Timer and PWM, Watchdog Timer

03.12.2013 Analog to Digital Converter

10.12.2013 Serial Interfaces: SPI, IIC and UART

17.12.2013 Additional Explanation of the Freescale Cup Cars

14.01.2014 Project Work on the Freescale Cup Cars

21.01.2014 Project Work on the Freescale Cup Cars

Hands-On Workshops Microcontroller Programming

08.10.2013 Workshop 1: Preparation of the Work Place

15.10.2013 Workshop 2: Loading and Debugging of Programs

22.10.2013 Workshop 3: Using the GPIO Pins

29.10.2013 Workshop 4: Clock Generation and Calculations

05.11.2013 Workshop 5: Interrupts

12.11.2013 No Workshop !

19.11.2013 Workshop 6: Using the Flash Memory

26.11.2013 Workshop 7: Timer and Pulse Width Modulation (PWM)

03.12.2013 Workshop 8: Analog to Digital Conversion

10.12.2013 Workshop 9: Serial Communication

17.12.2013 Project work on the Freescale Cup Cars

14.01.2014 Project work on the Freescale Cup Cars

21.01.2014 Project work on the Freescale Cup Cars

Participation on all workshops is required for admittance to the final project!

Start Time Tuesday: 15:45 p.m.

New Start Time Thursday: 14:45 p.m.

Recap: Peripheral Registers

A **peripheral register** is part of the peripheral block outside of the CPU.

For example: serial interface, timer, A/D converter ...

The peripheral registers are used to:

- configure the peripheral itself,
- read the status of the peripheral,
- exchange data with the peripheral.

Peripheral registers are often **memory mapped**, so the programmer can access these registers with C language instructions.

Recap: Address Space of ARM Cortex-M3/M4 in general

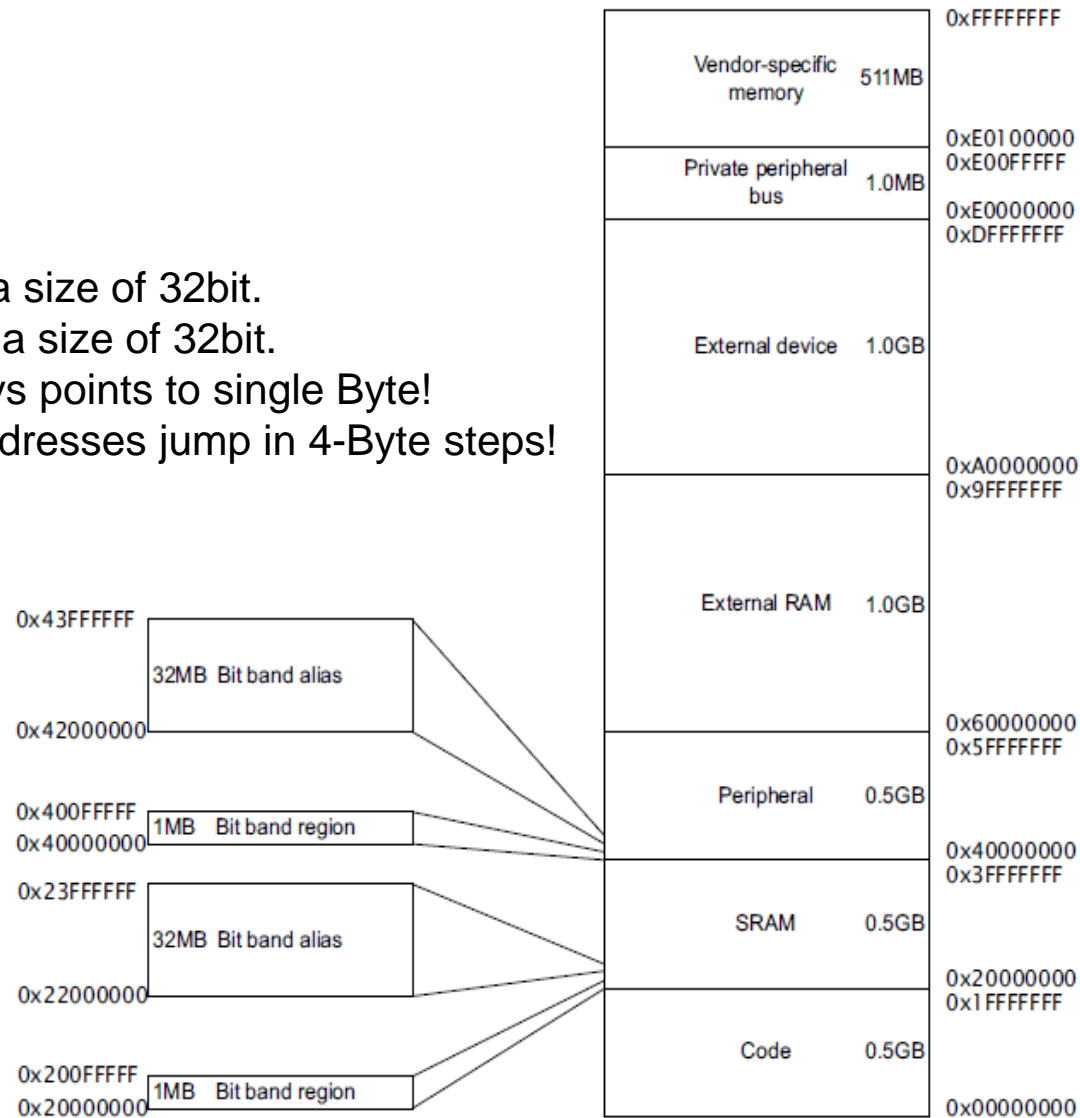
Please note:

Addresses have a size of 32bit.

Data words have a size of 32bit.

An address always points to single Byte!

Therefore, the addresses jump in 4-Byte steps!



Recap: Bit Fields

A bit field is set up with a structure declaration that labels each field and determines its width.

A bit field is used in computer programming to store multiple, logical, neighboring bits, where each of the sets of bits, and single bits can be addressed. A bit field is most commonly used to represent integral types of known, fixed bit-width. A well-known usage of bit-fields is to represent a set of bits, and/or series of bits, known as flags. For example, the first bit in a bit field can be used to determine the state of a particular attribute associated with the bit field.

Access to the Elements of a Struct Object

In order to access the elements of a struct object in C language there are two operators: the point operator (`.`) and the arrow operator (`->`).

Both operators work with two operands, the right operand is the name of the selected element. The left operand is an expression, which delivers the struct object.

The elements of the struct object are located in the memory in the sequence of their declaration. The address of the first element is identical to the address of the struct object itself. The elements which are declared later have a higher address than the elements which are declared earlier.

If there is a pointer to a structure, it is possible to use instead of the two operators `*` and `.` the arrow operator `->` for the access of the elements of the struct.

An expression in the form of `p->m` is equivalent to `(*p) .m`.

Bitfield Example

A general solution is to use a union to combine the bitfields with wider access

```
union {  
    uint8_t byte;  
    struct { uint8_t b0:1; ... uint8_t b7:1; } bits;  
} reg;
```

The whole register can be initialized quickly

```
reg.byte = BITMASKA | BITMASKB;
```

And individual bits easily set/cleared/toggled

```
if (reg.bits.b7) { ... }
```


Mapping of the Peripheral Registers within a Bitfield

```
#define UCHAR unsigned char
struct Timer // Mapping of the Peripheral
{
    struct
    {
        bool enable      : 1;
        bool intEnable   : 1;
        bool intPeriod   : 1;
        bool              : 1;
        UCHAR preScale   : 4;
    } reg0;
    UCHAR reg1;
    struct
    {
        bool pending     : 1;
        UCHAR             : 6;
        bool overf       : 1;
    } reg2;
} *pTimer = (struct Timer*)0x0100;
```

```
// Access to registers
pTimer->reg0.enable = true;
error = pTimer->reg2.overf;
```

Recap: Implementation in MK60DZ10.h

```
/** PORT - Peripheral register structure */
typedef struct PORT_MemMap {
    uint32_t PCR[32];           **< Pin Control Register n, array of
    uint32_t GPCLR;             **< Global Pin Control Low Register,
    uint32_t GPCHR;             **< Global Pin Control High Register,
    uint8_t RESERVED_0[24];
    uint32_t ISFR;              /**< Interrupt Status Flag Register,
    uint8_t RESERVED_1[28];
    uint32_t DFER;              /**< Digital Filter Enable Register,
    uint32_t DFCCR;             /**< Digital Filter Clock Register, of
    uint32_t DFWR;              /**< Digital Filter Width Register, of
} volatile *PORT_MemMapPtr;
```

Reminder: Important Documents

K60 Sub-Family Reference Manual (K60P144M100SF2RM)
1800 pages!

Kinetis Peripheral Module Quick Reference (KQRUG)
with software examples KINETIS512_SC.zip

K60 Sub-Family Reference Manual

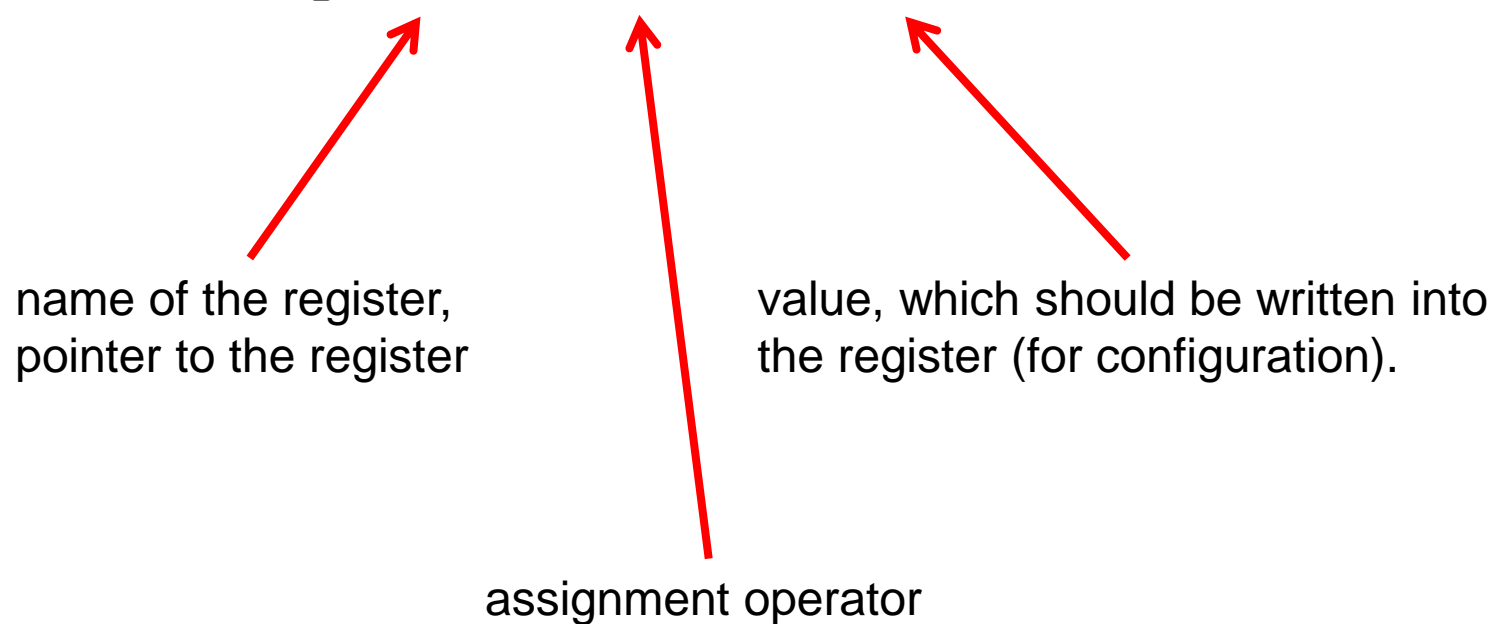
Supports: MK60DN256ZVLQ10, MK60DX256ZVLQ10,
MK60DN512ZVLQ10, MK60DN256ZVMD10, MK60DX256ZVMD10,
MK60DN512ZVMD10



Document Number: K60P144M100SF2RM
Rev. 6, Nov 2011

Writing a Register

```
*pointer = 0xFFFF0000 ;
```



name of the register,
pointer to the register

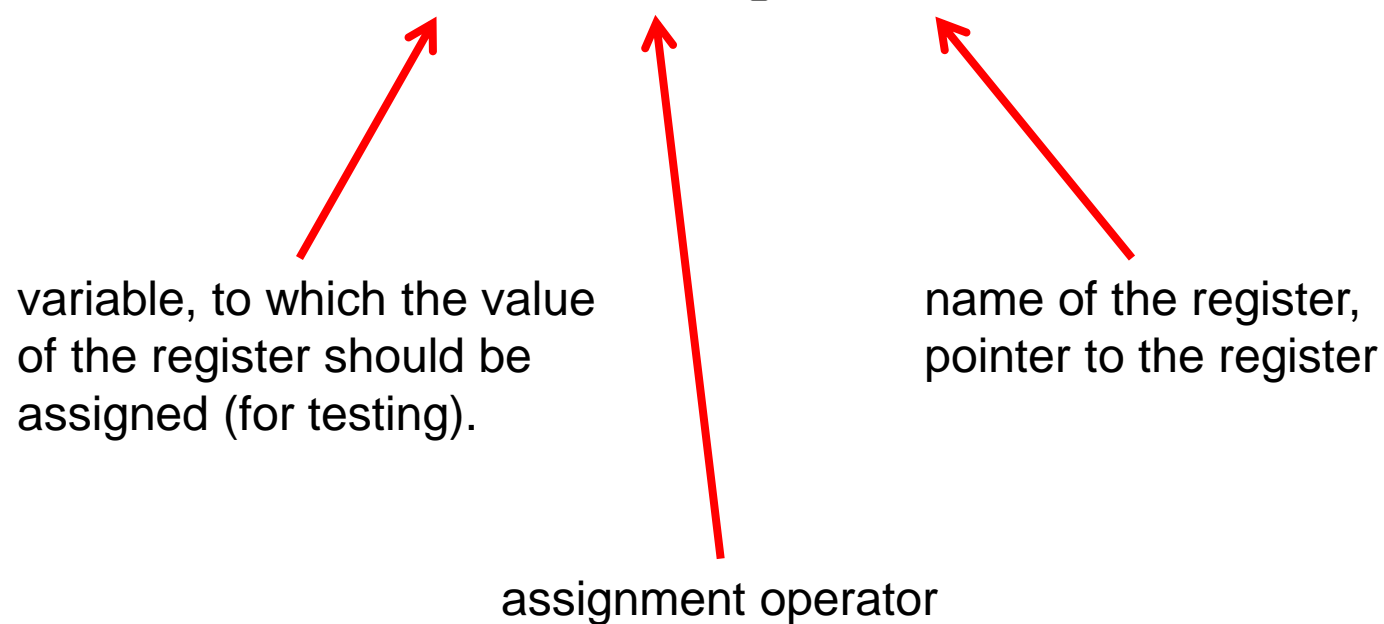
value, which should be written into
the register (for configuration).

assignment operator

But: This operation would write the whole register, that means all bits of the register would be overwritten! This is not desirable.

Reading a Register

```
variable = *pointer ;
```



variable, to which the value
of the register should be
assigned (for testing).

assignment operator

name of the register,
pointer to the register

But: This operation would read the whole register, that means all bits of the register would be detected! This is not desirable.

Bitwise Data (Bit Masks)

You can't write binary literals in C

Hexadecimal works best

`"0xE8" == "0xe8" => 1110 1000b`

`"0x1234" => 0001 0010 0011 0100b`

Hex literal uses:

- Initialize a variable
- Test a bit (or bits) in a variable
- Set/reset/toggle a bit (or bits) in a variable

<code>0x0</code>	<code>= 0000</code>
<code>0x1</code>	<code>= 0001</code>
<code>0x2</code>	<code>= 0010</code>
<code>0x3</code>	<code>= 0011</code>
<code>0x4</code>	<code>= 0100</code>
<code>0x5</code>	<code>= 0101</code>
<code>0x6</code>	<code>= 0110</code>
<code>0x7</code>	<code>= 0111</code>
<code>0x8</code>	<code>= 1000</code>
<code>0x9</code>	<code>= 1001</code>
<code>0xA</code>	<code>= 1010</code>
<code>0xB</code>	<code>= 1011</code>
<code>0xC</code>	<code>= 1100</code>
<code>0xD</code>	<code>= 1101</code>
<code>0xE</code>	<code>= 1110</code>
<code>0xF</code>	<code>= 1111</code>

Bit Masks

Individual bits (or groups) in a register may be significant on their own:

```
0xC000 -enable timer (1), disable (0)
0x2000 -interrupt when done (1), do not interrupt (0)
0x0020 -max count reached (1), still counting (0)
0x0001 -periodic timer (1), one-shot timer (0)
```

Use bitmasks to test/set/clear/toggle bits

```
#define TIMER_ENABLE 0xC000
#define TIMER_INTERRUPT 0x2000
#define TIMER_COMPLETE 0x0020
#define TIMER_PERIODIC 0x0001
```

Bitwise Operations

AND: “&” (not the same as “&&”)

OR: “|” (not the same as “||”)

NOT: “~” (not the same as “!”)

XOR: “^”

LEFT SHIFT: “<< *n*”

RIGHT SHIFT*: “>> *n*”

*Note: Compiler decides what happens to uppermost bit for signed data!

Bit Mask Examples

Testing bits

```
if (pTimer->control & TIMER_COMPLETE) { }
```

Setting bits

```
pTimer->control |= TIMER_INTERRUPT;
```

Clearing bits

```
pTimer->control &= ~TIMER_INTERRUPT;
```

Toggling bits

```
pTimer->control ^= TIMER_PERIODIC;
```

Bitwise Operations with Bit Masks: Testing Bits

AND: “&” (not the same as “&&”)

Register = reg = 0xAF64 = 1010111101100100

if (reg & 0x20) { ... }

1010111101100100

0000000000100000

0000000000100000

~~if (reg & ~0x20) { ... }~~

~~1010111101100100~~

~~1111111111011111~~

~~1010111101000100~~

Bitwise Operations with Bit Masks: Setting Bits

OR: “|” (not the same as “||”)

Register = reg = 0xAF64 = 1010111101100100

reg |= 0x01;

Read-Modify-Write

~~reg |= ~0x01;~~

1010111101100100

0000000000000001

1010111101100101

~~1010111101100100~~

~~1111111111111110~~

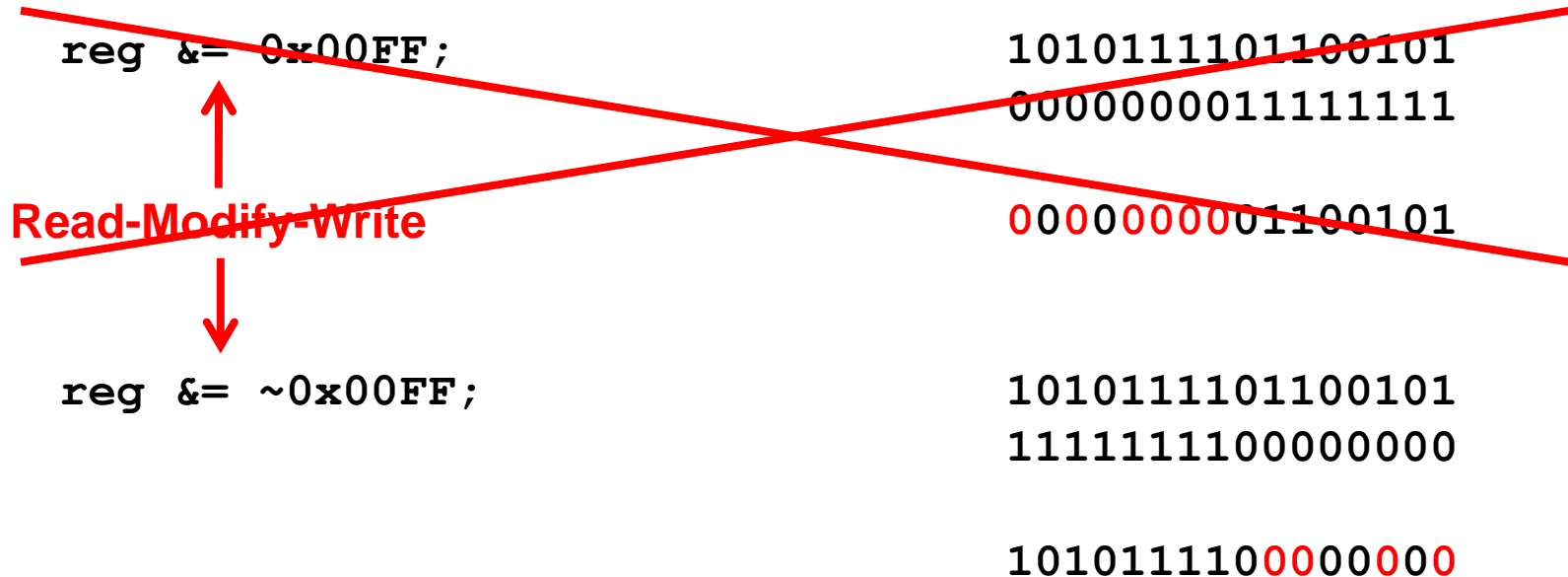
~~1111111111111110~~

`GPIOA_PDOR |= GPIO_PDOR_PDO(GPIO_PIN(11)); // orange LED off`

Bitwise Operations with Bit Masks: Deleting Bits

AND: “&” (not the same as “&&”)

Register = reg = 0xAF64 = 1010111101100100



```
GPIOA_PDOR &= ~GPIO_PDOR_PDO(GPIO_PIN(11)); // orange LED on
```

Bitwise Operations with Bit Masks: Toggling Bits

XOR: “^”

Register = reg = 0xAF64 = 1010111101100100

reg ^= 0x8000;



Read-Modify-Write



~~reg ^= ~0x8000;~~

1010111101100101

1000000000000000

0010111101100101

~~1010111101100101~~

~~0111111111111111~~

~~1101000010011010~~

Access to Registers

R/W: Read/Write

It is possible to read the register and to write to the register.

RO: Read Only

It is only possible to read the register.

WO: Write Only

It is only possible to write to the register.

You'll have to keep a local shadow copy of its state and can't use |=, &=, ^=, etc.

also possible: **Write Once**

In most cases used for configuration registers.

Example of a Bit Mask

```
LPC_GPIO1->FIOSET |= (1<<18); //P1.18 ON and LED ON
```

name of the register
(pointer to the address)

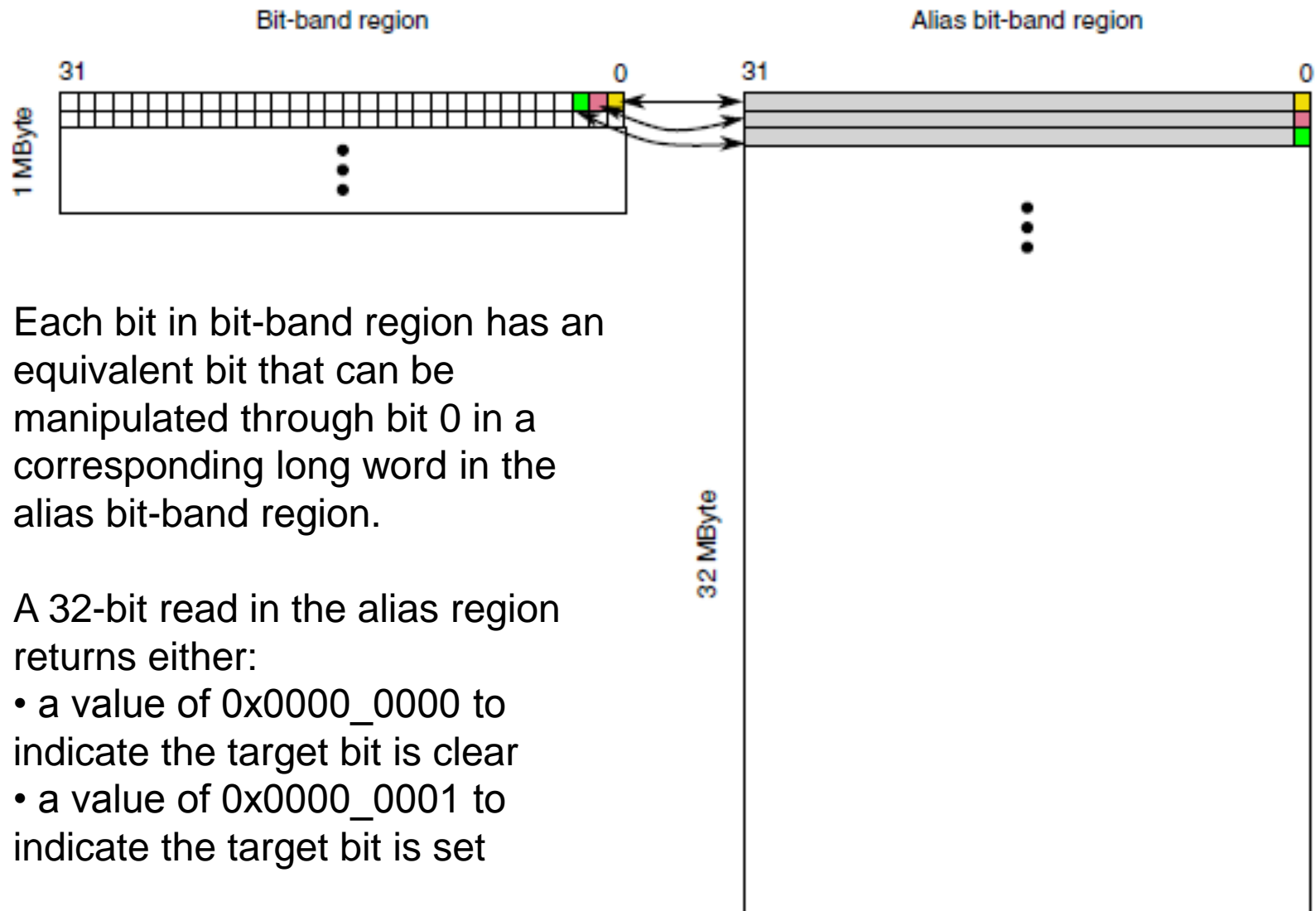
operation

bit mask

```
LPC_GPIO1->FIOSET |= 0x00040000;
```

Bit mask (binary): 0000 0000 0000 0100 0000 0000 0000 0000

Special Feature of ARM Cortex M3/4: Bit-band



Each bit in bit-band region has an equivalent bit that can be manipulated through bit 0 in a corresponding long word in the alias bit-band region.

A 32-bit read in the alias region returns either:

- a value of 0x0000_0000 to indicate the target bit is clear
- a value of 0x0000_0001 to indicate the target bit is set

I/O Pin, also called: GPIO

General-purpose input/output (GPIO) is a generic pin on an integrated circuit whose behavior (including whether it is an input or output pin) can be controlled (programmed) by the user at run time.

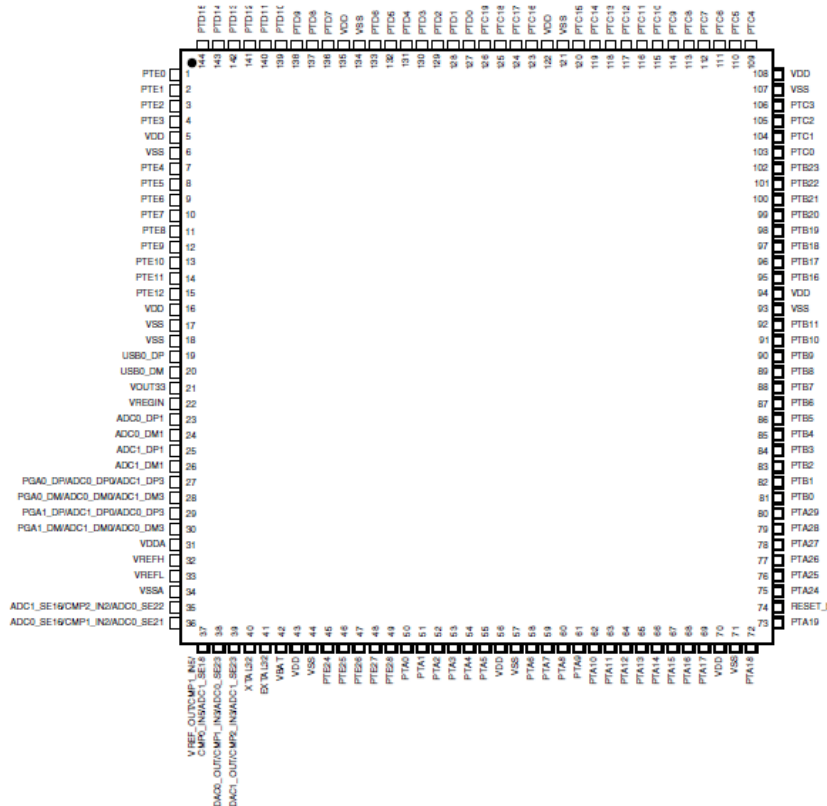
GPIO capabilities may include:

- GPIO pins can be configured to be input or output
- GPIO pins can be enabled/disabled
- Input values are readable (typically high=1, low=0)
- Output values are writable/readable
- Input values can often be used as IRQs (typically for wakeup events)

Several input/output pins are connected together to form an I/O port. At the Kinetis K60 these I/O ports are 32 bits (4 Byte) wide and include the ports **A**, **B**, **C**, **D** and **E**.

Not all of these port signals are connected to the actual pins of the microcontroller!

Kinetis K60 Pinouts (144 Connections)



	1	2	3	4	5	6	7	8	9	10	11	12	
A	PTD7	PTD6	PTD5	PTD4	PTD3	PTD2	PTD1	PTD0	PTC16	PTC15	PTC14	PTC13	PTC12
B	PTD12	PTD11	PTD10	PTD9	PTD8	PTD7	PTD6	PTD5	PTD4	PTD3	PTD2	PTD1	PTD0
C	PTD15	PTD14	PTD13	PTD12	PTD11	PTD10	PTD9	PTD8	PTD7	PTD6	PTD5	PTD4	PTD3
D	PTD2	PTD1	PTD0	PTC17	PTC16	PTC15	PTC14	PTC13	PTC12	PTC11	PTC10	PTC9	PTC8
E	PTD18	PTD17	PTD16	PTD15	PTD14	PTD13	PTD12	PTD11	PTD10	PTD9	PTD8	PTD7	PTD6
F	PTD19	PTD18	PTD17	PTD16	PTD15	PTD14	PTD13	PTD12	PTD11	PTD10	PTD9	PTD8	PTD7
G	PTD20	PTD19	PTD18	PTD17	PTD16	PTD15	PTD14	PTD13	PTD12	PTD11	PTD10	PTD9	PTD8
H	PTD21	PTD20	PTD19	PTD18	PTD17	PTD16	PTD15	PTD14	PTD13	PTD12	PTD11	PTD10	PTD9
J	PTD22	PTD21	PTD20	PTD19	PTD18	PTD17	PTD16	PTD15	PTD14	PTD13	PTD12	PTD11	PTD10
K	PTD23	PTD22	PTD21	PTD20	PTD19	PTD18	PTD17	PTD16	PTD15	PTD14	PTD13	PTD12	PTD11
L	PTD24	PTD23	PTD22	PTD21	PTD20	PTD19	PTD18	PTD17	PTD16	PTD15	PTD14	PTD13	PTD12
M	PTD25	PTD24	PTD23	PTD22	PTD21	PTD20	PTD19	PTD18	PTD17	PTD16	PTD15	PTD14	PTD13

144 LQFP

144 MAPBGA

Using the Pins

10.3.1 K60 Signal Multiplexing and Pin Assignments (Excerpt) p. 233

144 LQFP	144 MAP BGA	Pin Name	Default	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
66	L10	PTA14	DISABLED		PTA14	SPI0_PCS0	UART0_TX	RMII0_CRS_DV/ MII0_RXDV		I2S0_TX_B CLK	
67	L11	PTA15	DISABLED		PTA15	SPI0_SCK	UART0_RX	RMII0_TXE N/ MII0_TXEN		I2S0_RXD	
68	K10	PTA16	DISABLED		PTA16	SPI0_SOUT	UART0_CTS_b	RMII0_TXD0/ MII0_TXD0		I2S0_RX_FS	
69	K11	PTA17	ADC1_SE17	ADC1_SE17	PTA17	SPI0_SIN	UART0_RTS_b	RMII0_TXD1/ MII0_TXD1		I2S0_MCLK	I2S0_CLKIN
70	E8	VDD	VDD	VDD							
71	G8	VSS	VSS	VSS							

Important Recommendation: Create a Resource Plan!

This is upmost important for every project, also for the Freescale Cup Car.

Resource plan:

Which resources are required?

- depends on the project / wish list of your manager

Which resources are available?

- depends on the microcontroller you use

Which pins are allocated or can be used?

- depends on the hardware / schematics

A resource plan is the allocation of the resources (pins) of your microcontroller to the desired functionality and existing hardware.

Using the I/O Pins

The configuration of the I/O pins takes place in most cases at the very beginning of the program, directly after the reset of the microcontroller (so called initialization):

- 1) Step 1: Enable clock of the I/O pins
- 2) Step 2: Define pin function as a GPIO
- 3) Step 3: Define the direction of the GPIO (input or output)
- 4) Step 4: Set additional options (Pull-Up, Pull-Down, Open Drain ...)
- 5) Step 5: Set interrupt options

Then, the I/O pin can be used during the program itself:

- 1) Set the output value (0 or 1) or
- 2) Read the input value (0 or 1) or
- 3) Use the Interrupts

Port Registers at the Kinetis K60

Port Data Output Register (GPIOx_PDOR)

- R/W

0 Logic level 0 is driven on pin

1 Logic level 1 is driven on pin

Port Data Input Register (GPIOx_PDIR)

- Read Only!

0 Pin logic level is logic zero.

1 Pin logic level is logic one.

Port Set Output Register (GPIOx_PSOR)

- Write Only!

0 Corresponding bit in PDORn does not change.

1 Corresponding bit in PDORn is set to logic one.

Port Clear Output Register (GPIOx_PCOR)

- Write Only!

0 Corresponding bit in PDORn does not change.

1 Corresponding bit in PDORn is set to logic zero.

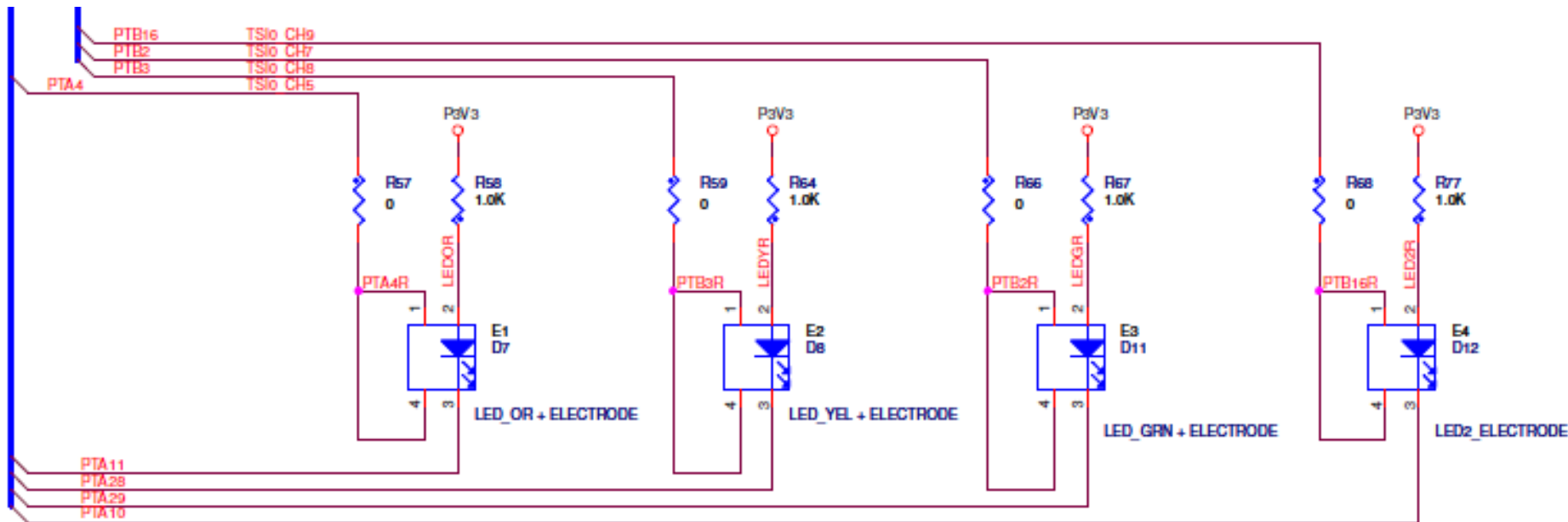
Port Toggle Output Register (GPIOx_PTOR)

- Write Only!

0 Corresponding bit in PDORn does not change.

1 Corresponding bit in PDORn is set to the inverse of its existing logic state.

I/O-Pins as Outputs



orange LED:	Port Pin PTA11
yellow LED:	Port Pin PTA28
green LED:	Port Pin PTA29
blue LED:	Port Pin PTA10

Step 1: Enable Clock of the I/O Pins

To configure the I/O pin muxing options, the port clocks must first be enabled. This allows the pin functions to later be changed to the desired function for the application.

```
// Turn on all port clocks
```

```
SIM_SCGC5 = SIM_SCGC5_PORTA_MASK
           | SIM_SCGC5_PORTB_MASK
           | SIM_SCGC5_PORTC_MASK
           | SIM_SCGC5_PORTD_MASK
           | SIM_SCGC5_PORTE_MASK;
```

```
0x200u = 1000000000
0x400u = 1000000000
0x800u = 100000000000
0x1000u = 10000000000000
0x2000u = 100000000000000
```

12.2.12 System Clock Gating Control Register 5 (SIM_SCGC5)

Address: SIM_SCGC5 is 4004_7000h base + 1038h offset = 4004_8038h

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0													1	0	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
R	0		PORTE		PORTD		PORTC		PORTB		PORTA		1		0	TSI		0		REGFILE		LPTIMER	
W			PORTE		PORTD		PORTC		PORTB		PORTA				TSI				REGFILE		LPTIMER		
Reset	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Step 2: Define Pin Function as a GPIO

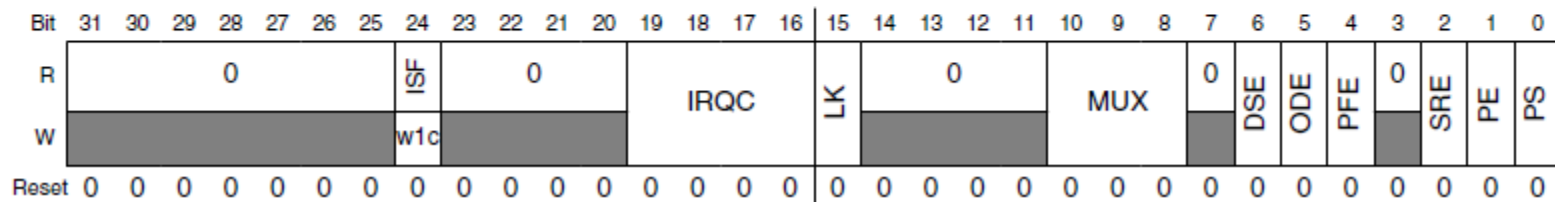
```
// Set PTA10, PTA11, PTA28, and PTA29 for GPIO functionality
PORTA_PCR10=(0|PORT_PCR_MUX(1));
PORTA_PCR11=(0|PORT_PCR_MUX(1));
PORTA_PCR28=(0|PORT_PCR_MUX(1));
PORTA_PCR29=(0|PORT_PCR_MUX(1));
```



11.4.1 Pin Control Register n (PORTx_PCRn)

For PCR1 to PCR5 of the port A, bit 0, 1, 6, 8, 9, 10 reset to 1; for the PCR0 of the port A, bit 1, 6, 8, 9, 10 reset to 1; in other conditions, all bits reset to 0.

Addresses: 4004_9000h base + 0h offset + (4d × n), where n = 0d to 31d




```
#define PORT_PCR_MUX_MASK 0x700u
#define PORT_PCR_MUX_SHIFT 8
#define PORT_PCR_MUX(x)
((uint32_t)((uint32_t)(x)<<PORT_PCR_MUX_SHIFT)&PORT_PCR_MUX_MASK)
```

0x700u = 1110000000

Step 3: Define the Direction of the GPIO

```
// Change PTA10, PTA11, PTA28, PTA29 to outputs
GPIOA_PDDR = GPIO_PDDR_PDD(GPIO_PIN(10)
                             | GPIO_PIN(11)
                             | GPIO_PIN(28)
                             | GPIO_PIN(29) );
```



54.2.6 Port Data Direction Register (GPIOx_PDDR)

The PDDR configures the individual port pins for input or output.

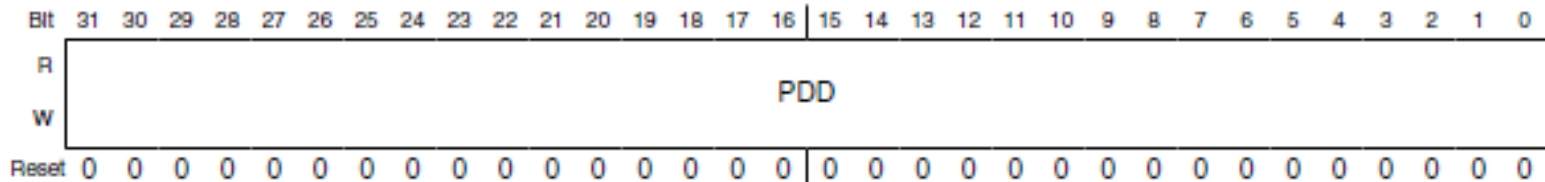
Addresses: GPIOA_PDDR is 400F_F000h base + 14h offset = 400F_F014h

GPIOB_PDDR is 400F_F040h base + 14h offset = 400F_F054h

GPIOC_PDDR is 400F_F080h base + 14h offset = 400F_F094h

GIOD_PDDR is 400F_F0C0h base + 14h offset = 400F_F0D4h

GPIOE_PDDR is 400F_F100h base + 14h offset = 400F_F114h



Set the Value of the GPIO

```
// alle LEDs am Anfang ausschalten
GPIOA_PDOR |= GPIO_PDOR_PDO(GPIO_PIN(11)); // aus
GPIOA_PDOR |= GPIO_PDOR_PDO(GPIO_PIN(28)); // aus
GPIOA_PDOR |= GPIO_PDOR_PDO(GPIO_PIN(29)); // aus
GPIOA_PDOR |= GPIO_PDOR_PDO(GPIO_PIN(10)); // aus
...
GPIOA_PDOR &= ~GPIO_PDOR_PDO(GPIO_PIN(11)); // an
...
GPIOA_PDOR |= GPIO_PDOR_PDO(GPIO_PIN(11)); // aus
```



54.2.1 Port Data Output Register (GPIOx_PDOR)

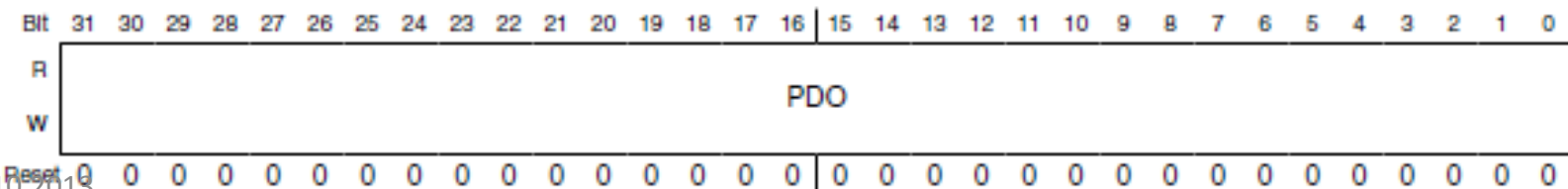
Addresses: GPIOA_PDOR is 400F_F000h base + 0h offset = 400F_F000h

GPIOB_PDOR is 400F_F040h base + 0h offset = 400F_F040h

GPIOC_PDOR is 400F_F080h base + 0h offset = 400F_F080h

GPIOD_PDOR is 400F_F0C0h base + 0h offset = 400F_F0C0h

GPIOE_PDOR is 400F_F100h base + 0h offset = 400F_F100h



Additional Options: Pin Control Register

11.4.1 Pin Control Register n (PORTx_PCRn)

For PCR1 to PCR5 of the port A, bit 0, 1, 6, 8, 9, 10 reset to 1; for the PCR0 of the port A, bit 1, 6, 8, 9, 10 reset to 1; in other conditions, all bits reset to 0.

Addresses: 4004_9000h base + 0h offset + (4d × n), where n = 0d to 31d

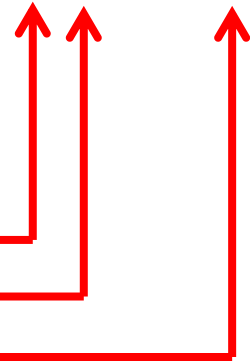
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0							ISF	0			IRQC				LK	0				MUX				0	DSE	ODE	PFE	0	SRE	PE	PS
W								w1c																		DSE	ODE	PFE		SRE	PE	PS
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

If the pin is an output:

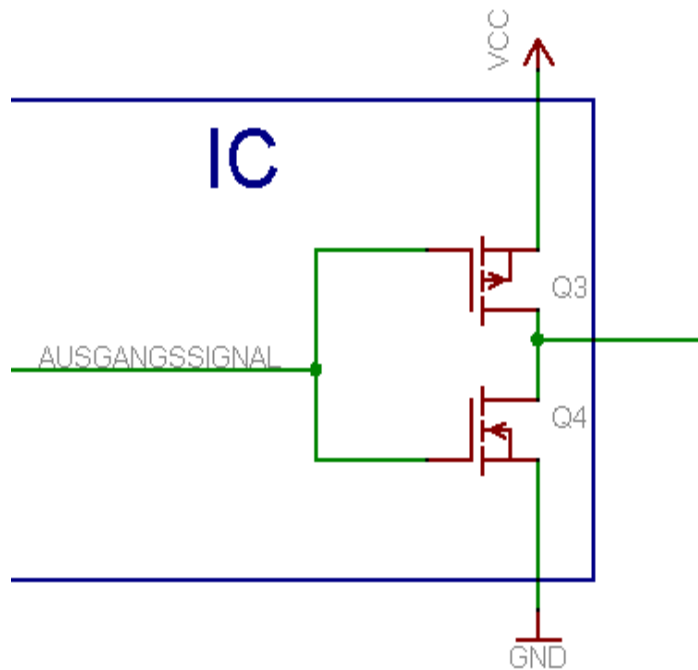
Drive Strength Enable

Open Drain Enable

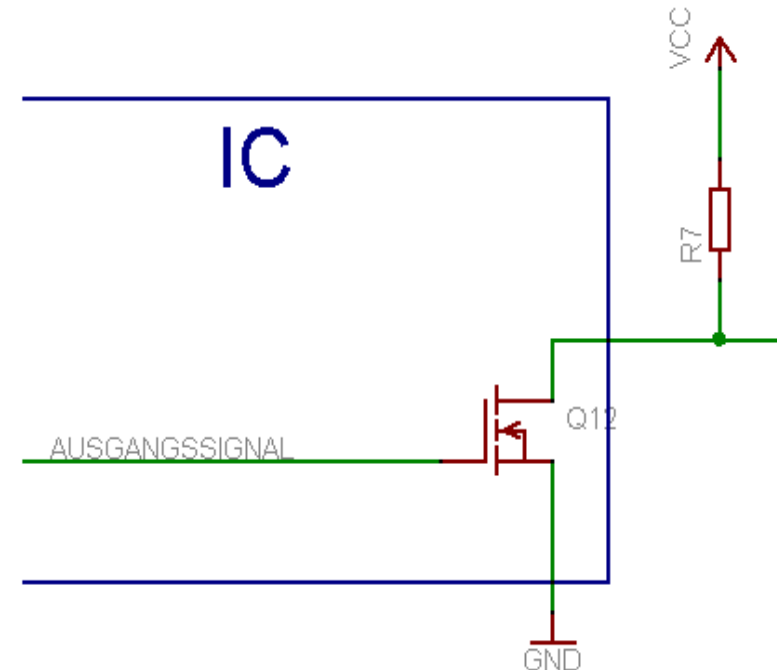
Slew Rate Enable



Output Options of the I/O Pins

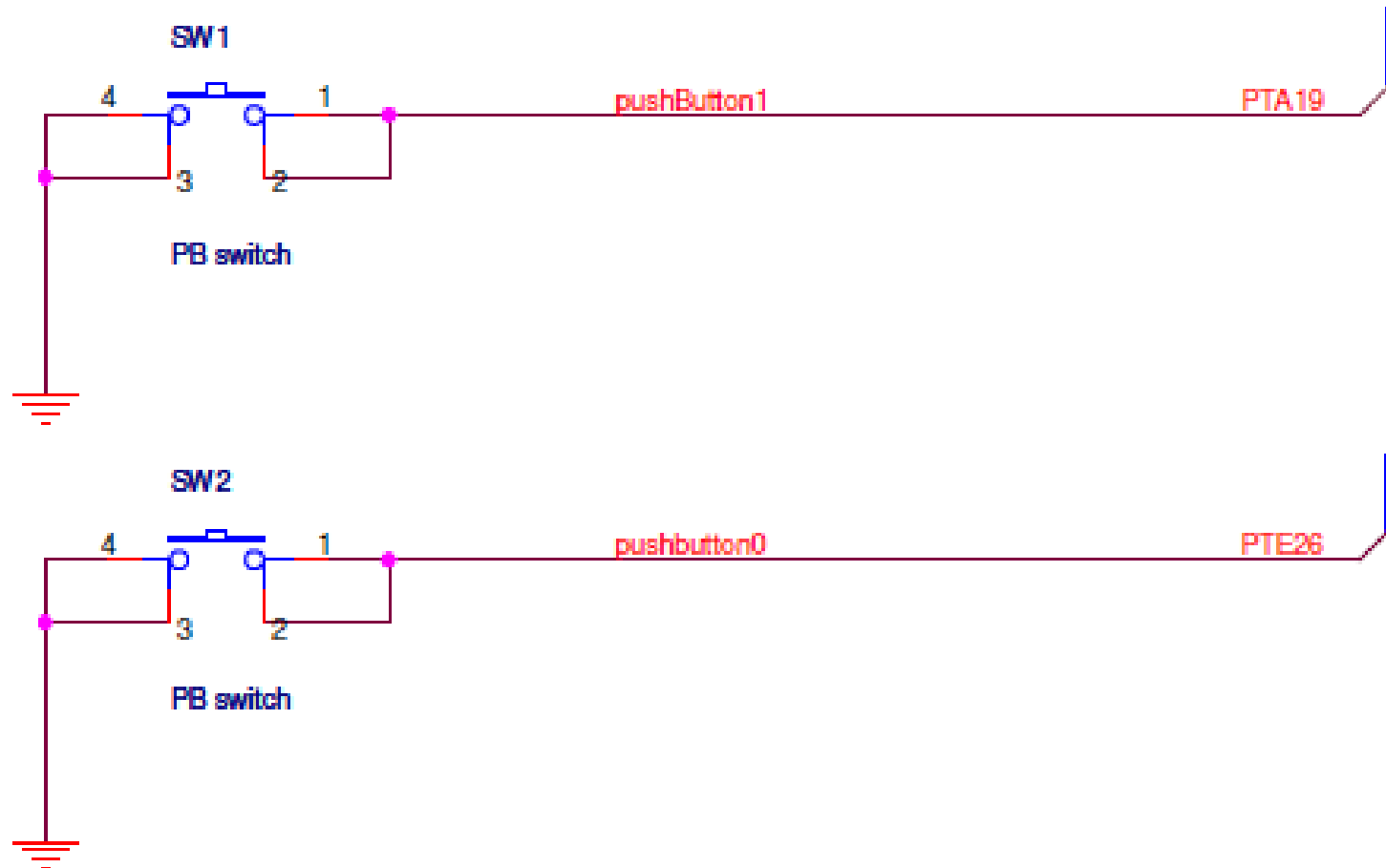


Push / Pull



Open Drain / Open Collector

I/O Pins as Inputs



SW1: Port Pin PTA19
SW2: Port Pin PTE26

Configuration of the I/O-Pins as Input

```
// Set PTA 19 and PTE26 for GPIO functionality
PORTA_PCR19 = (0|PORT_PCR_MUX(1));
PORTE_PCR26 = (0|PORT_PCR_MUX(1));
```

```
// Set PTA19 and PTE26 to inputs
GPIOA_PDDR &= ~GPIO_PDDR_PDD(GPIO_PIN(19));
GPIOE_PDDR &= ~GPIO_PDDR_PDD(GPIO_PIN(26));
```



54.2.6 Port Data Direction Register (GPIOx_PDDR)

The PDDR configures the individual port pins for input or output.

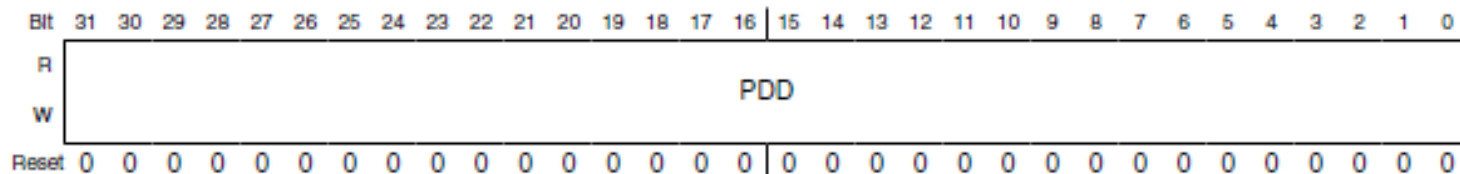
Addresses: GPIOA_PDDR is 400F_F000h base + 14h offset = 400F_F014h

GPIOB_PDDR is 400F_F040h base + 14h offset = 400F_F054h

GPIOC_PDDR is 400F_F080h base + 14h offset = 400F_F094h

GIOD_PDDR is 400F_F0C0h base + 14h offset = 400F_F0D4h

GPIOE_PDDR is 400F_F100h base + 14h offset = 400F_F114h



Using the I/O Pin as Input

```
if (!(GPIOA_PDIR & (GPIO_PIN(19))))    // bit 19 not set
{                                       // button SW1 on
    . . . . .
}
```



54.2.5 Port Data Input Register (GPIOx_PDIR)

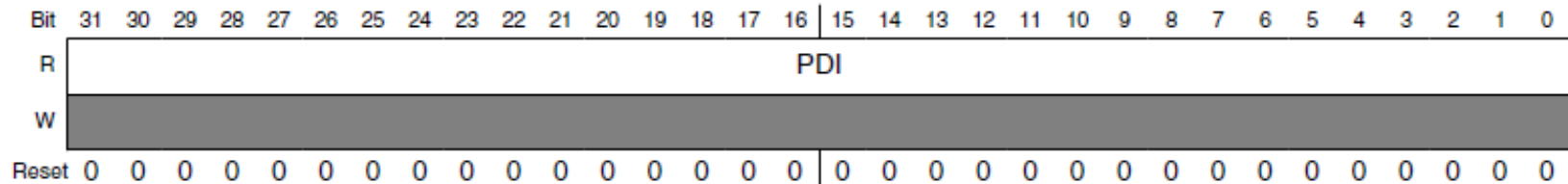
Addresses: GPIOA_PDIR is 400F_F000h base + 10h offset = 400F_F010h

GPIOB_PDIR is 400F_F040h base + 10h offset = 400F_F050h

GPIOC_PDIR is 400F_F080h base + 10h offset = 400F_F090h

GPIOD_PDIR is 400F_F0C0h base + 10h offset = 400F_F0D0h

GPIOE_PDIR is 400F_F100h base + 10h offset = 400F_F110h



Additional Options: Pin Control Register

11.4.1 Pin Control Register n (PORTx_PCRn)

For PCR1 to PCR5 of the port A, bit 0, 1, 6, 8, 9, 10 reset to 1; for the PCR0 of the port A, bit 1, 6, 8, 9, 10 reset to 1; in other conditions, all bits reset to 0.

Addresses: 4004_9000h base + 0h offset + (4d × n), where n = 0d to 31d

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
R	0							ISF	0			IRQC				LK	0				MUX				0	DSE	ODE	PFE	0	SRE	PE	PS	
W								w1c									LK										DSE	ODE	PFE		SRE	PE	PS
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

If the pin is an input:

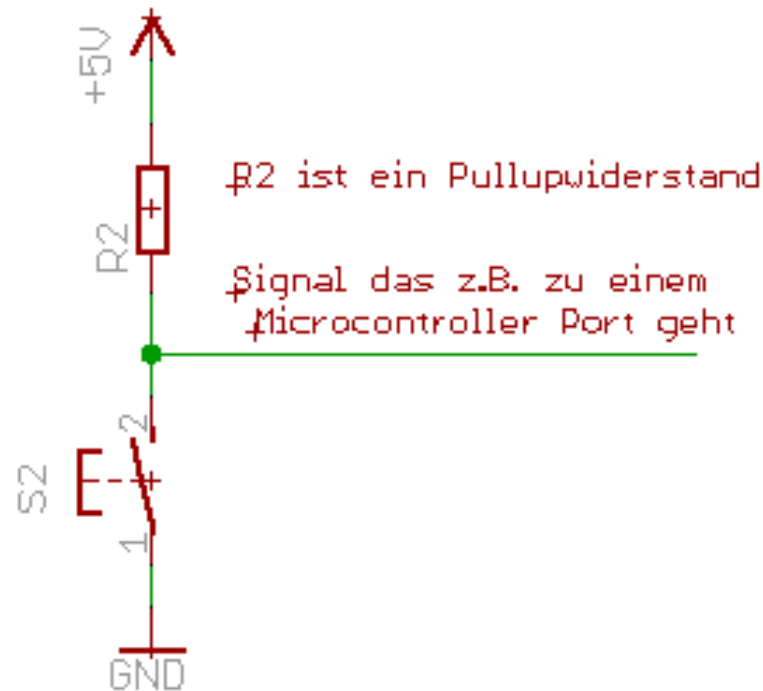
Interrupt / DMA Request

Passive Filter Enable

Pull Enable

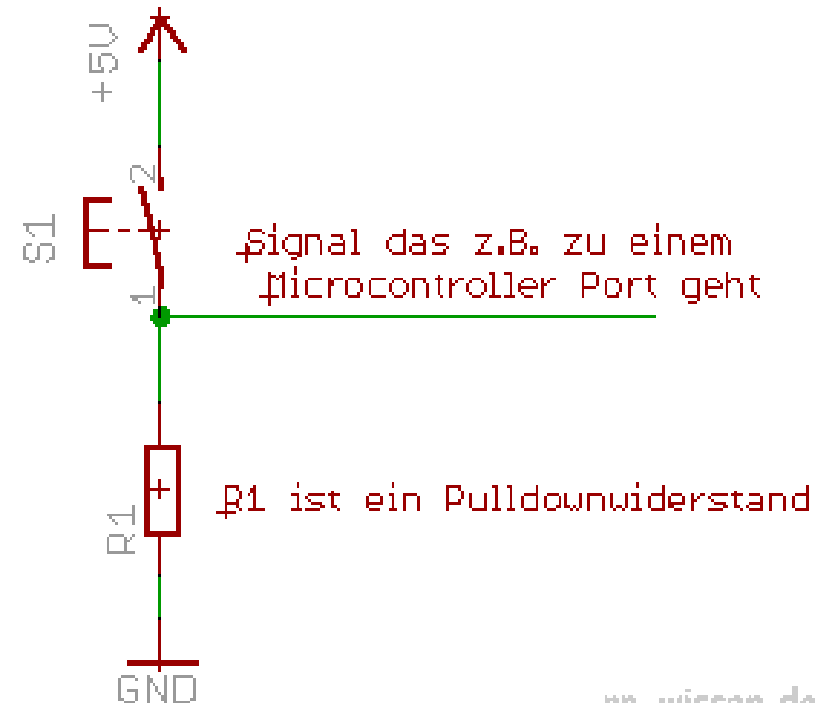
Pull Select

Input Options of the I/O Pins



rn-wissen.de

Pull-Up



rn-wissen.de

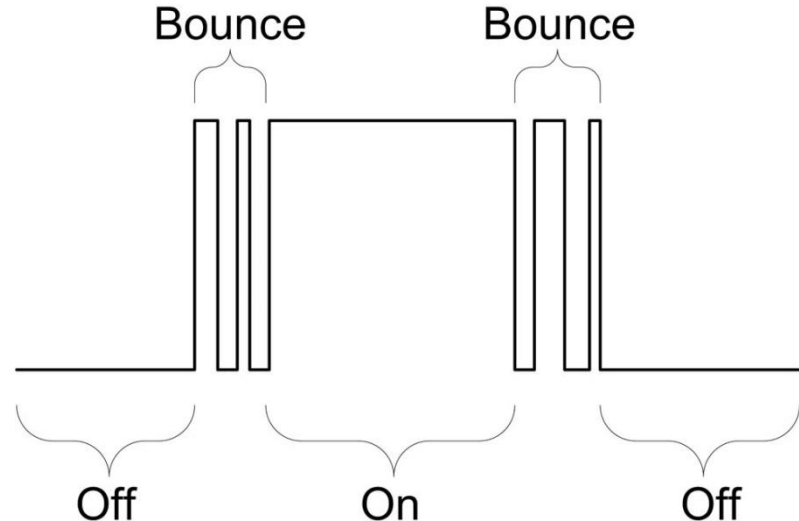
Pull-Down

Debouncing of contacts

Bouncing is the tendency of any two metal contacts in an electronic device to generate multiple signals as the contacts close or open; debouncing is any kind of hardware device or software that ensures that only a single signal will be acted upon for a single opening or closing of a contact.

When you press a key on your keyboard, you expect a single contact to be recorded by your computer. In fact, however, there is an initial contact, a slight bounce or lightening up of the contact, then another contact as the bounce ends, yet another bounce back, and so forth.

A similar effect takes place when a switch made using a metal contact is opened. The usual solution is a debouncing device or software that ensures that only one digital signal can be registered within the space of a given time (usually milliseconds).



Design Recommendations

2.3.4.2 General purpose I/O

General purpose inputs, such as low speed inputs, timer inputs, and signals from off-board should have low pass filters (series resistor and capacitor to ground) to prevent data corruption due to crosstalk or transients. The filter capacitor should be placed close to the MCU pin, while the resistor can be placed closer to the source.

Inputs that come from connectors should have low pass filtering at the connector to prevent noise from propagating onto the PCB. This requires a robust ground structure around the connector. Series resistors for signals that come from off-board should be placed as close to the connector as possible. A filter cap closer to the MCU input pin may be required if the signal trace length is very long and can pick up noise from other circuits.

Output pins should not have any significant capacitance placed close to the MCU. These signals can have capacitors at the load or connector to minimize radiated emissions if necessary.

Literature and Links

<http://www.arm.com/products/processors/cortex-m/cortex-m3.php>

<http://www.arm.com/products/processors/technologies/instruction-set-architectures.php>

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=K60

K60 Sub-Family Reference Manual

Kinetis Peripheral Module Quick Reference

Cortex-M4 Technical Reference Manual

<http://www.arm.com/products/processors/cortex-m/index.php>



Technische Hochschule Deggendorf – Edlmairstr. 6 und 8 – 94469 Deggendorf