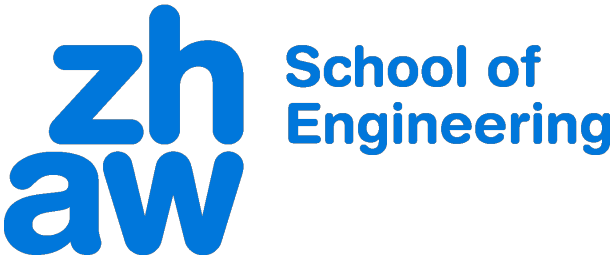
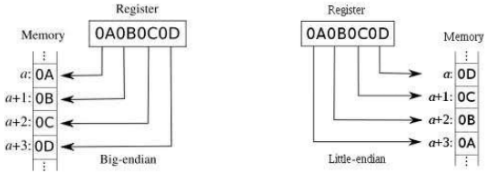
ZHAW Zurich University of Applied Sciences
Winterthur

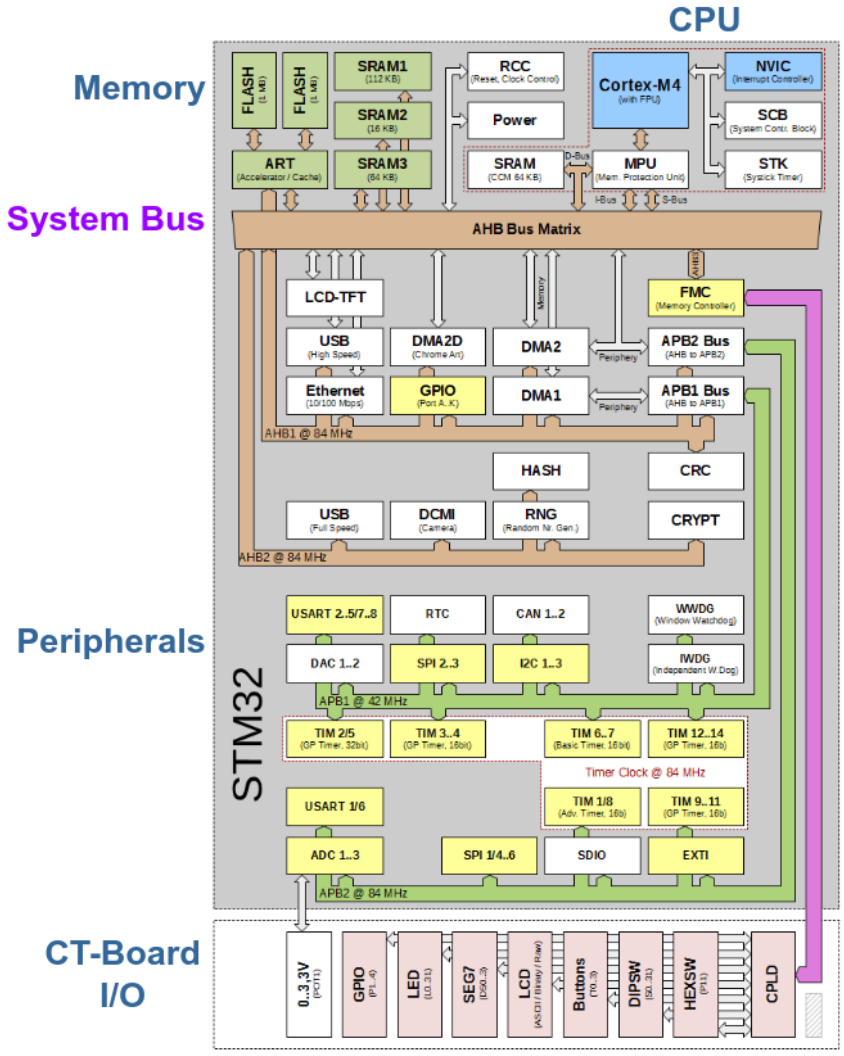


Zusammenfassung CT2
Studienwochen 1-14

Written by: Severin Sprenger
June 17, 2026
Zf. CT2 SW 1-14

1 General





Bus Timing Options

- Synchronous
  - Master and slaves use a common clock[1)]
  - Clock edges control bus transfer on both sides
  - Used by most on-chip busses
  - Off-chip: DDR and synchronous RAM

- Asynchronous
  - Slaves have no access to the clock of the master
  - Control signals carry timing information to allow synchronization
  - Widely used for low data-rate off-chip memories
    → parallel flash memories and asynchronous RAM

Access through Pointers

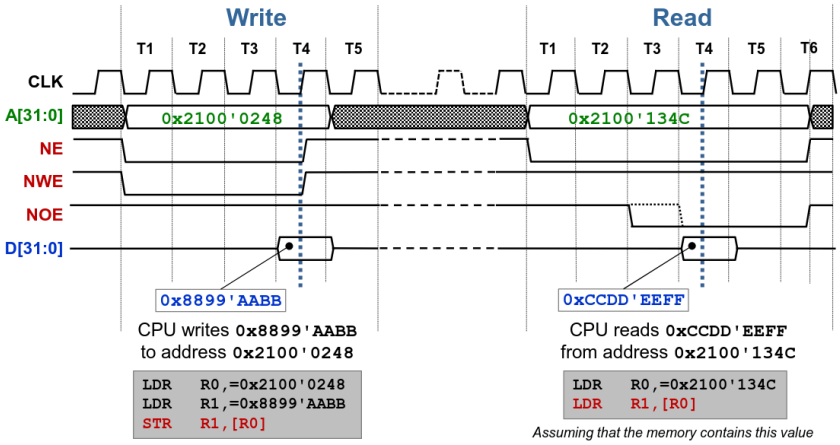
- E.g. writing to and reading from CT Board I/O

```
// a pointer called p_reg pointing to
// a volatile uint32_t
volatile uint32_t *p_reg;

// set LEDs
p_reg = (volatile uint32_t *) (0x60000100);
*p_reg = 0xAA55AA55;

// wait for dip_switches to be non-zero
p_reg = (volatile uint32_t *) (0x60000200);
while ( *p_reg == 0 ) {
}
```

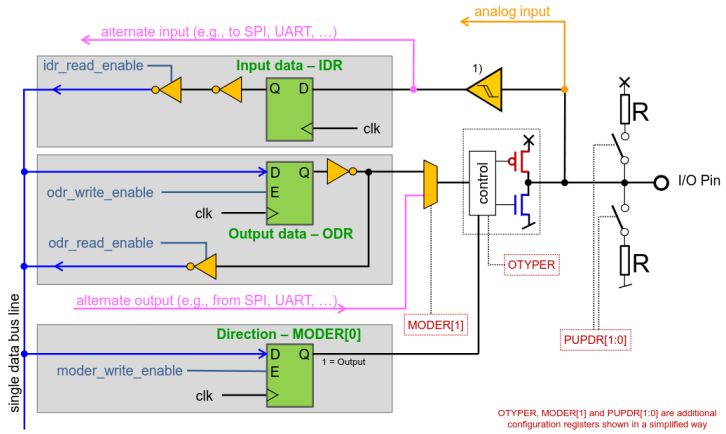
cast 'unsigned integer' to 'pointer to volatile uint32_t'

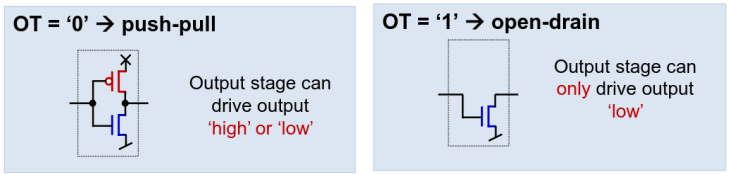
write 0xAA55'AA55 to LEDs

read dip-switches



2 GPIO

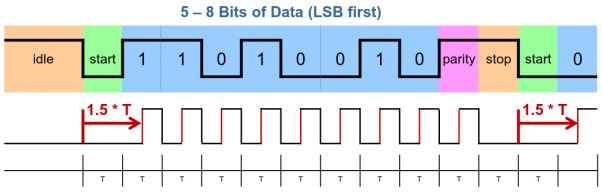


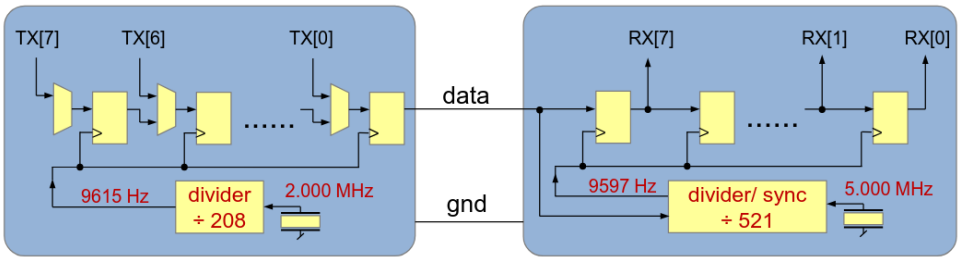


3 Peripheral Buses

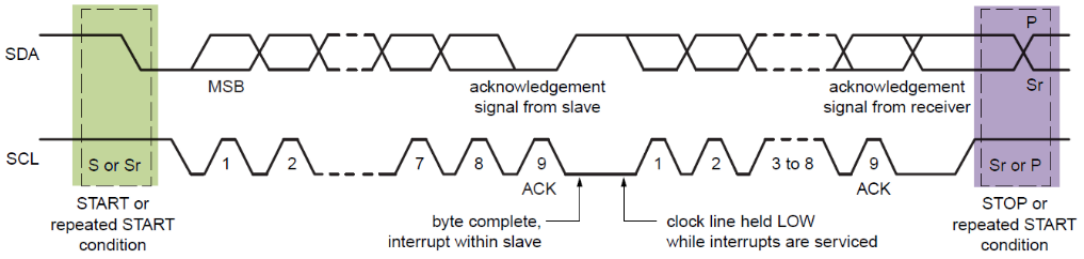
UART	SPI	I2C
serial ports (RS-232)	4-wire bus	2-wire bus
TX, RX opt. control signals	*MOSI, MISO, SCLK, SS*	*SCL, SDA*
point-to-point	point-to-multipoint	(multi-) point-to-multi-point
full-duplex	full-duplex	half-duplex
asynchronous	synchronous	synchronous
only higher layer addressing	slave selection through $\overline{SS}$ signal	7/10-bit slave address
parity bit possible	no error detection	no error detection
chip-to-chip, PC terminal program	chip-to-chip, on-board connections	chip-to-chip, board-to-board connections
The three interfaces provide the lowest layer of communication and require higher level protocols to provide and interpret the transferred data.		

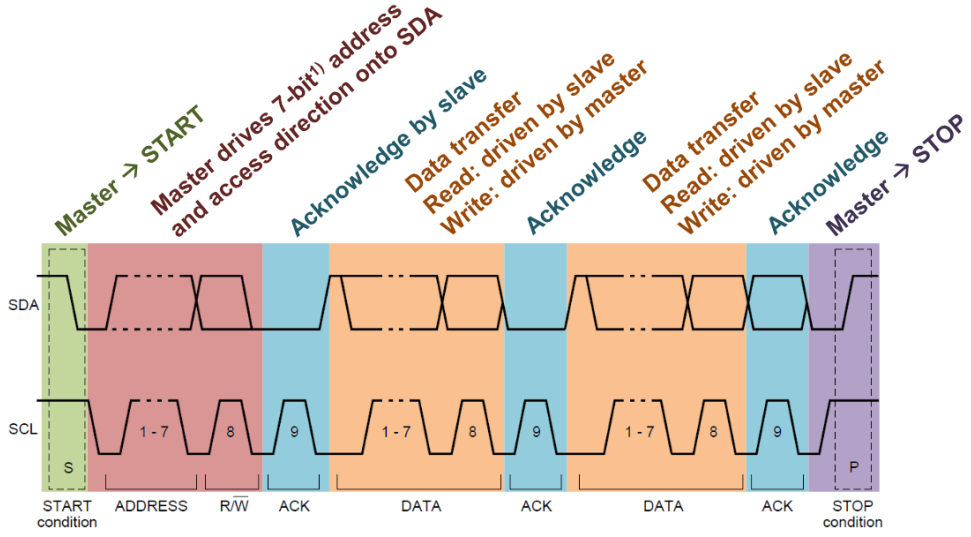
3.1 UART



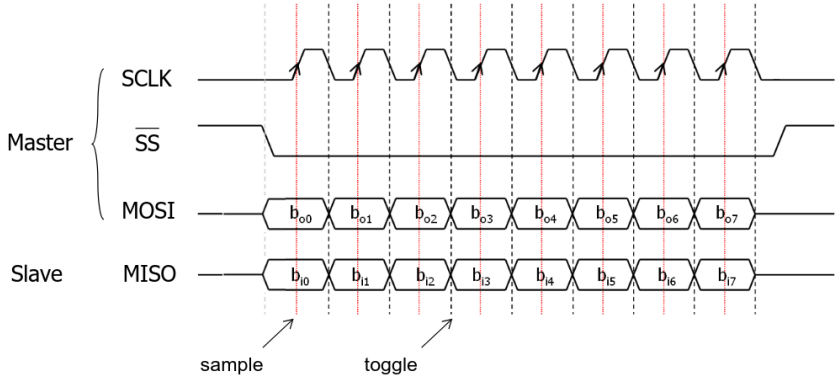


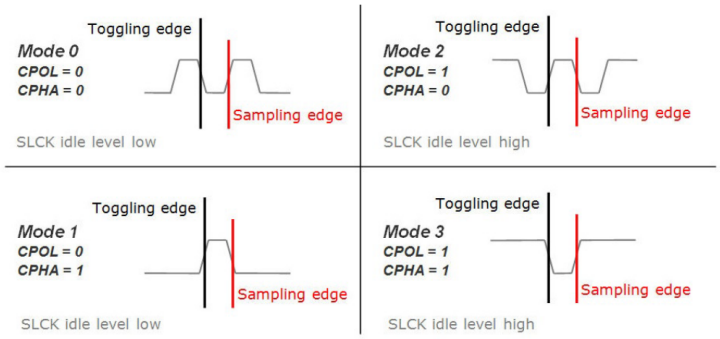
3.2 I2C



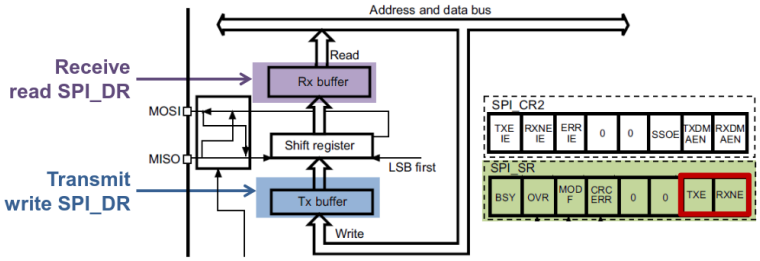


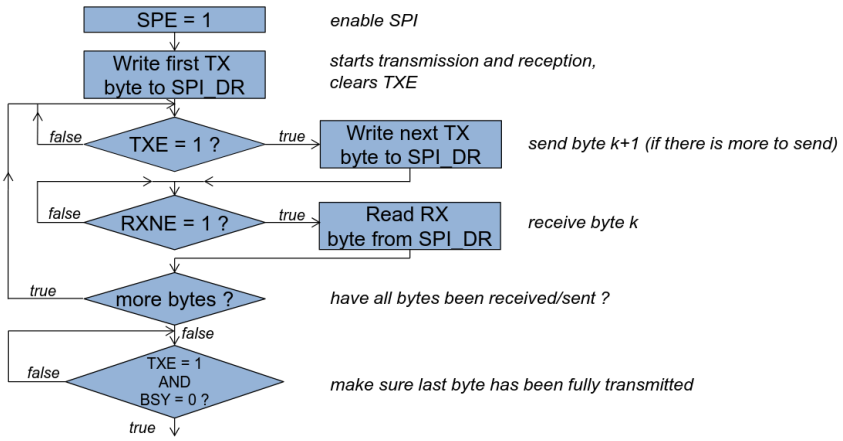
3.3 SPI



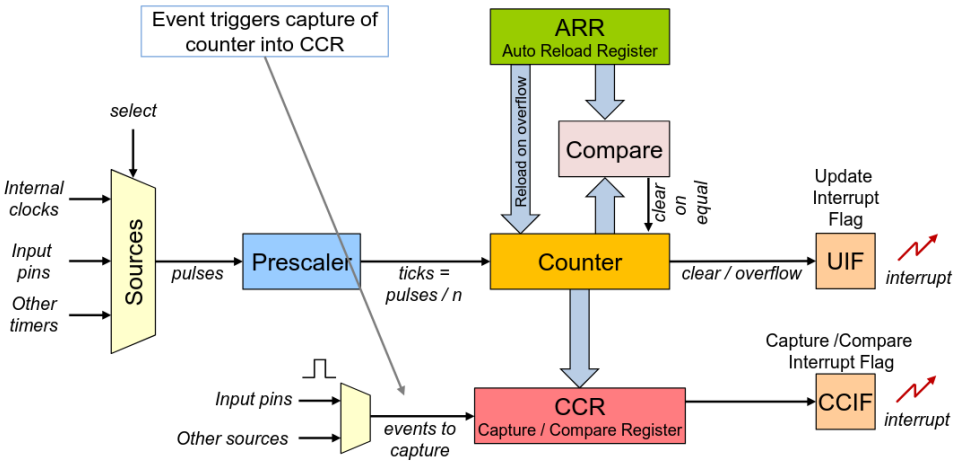


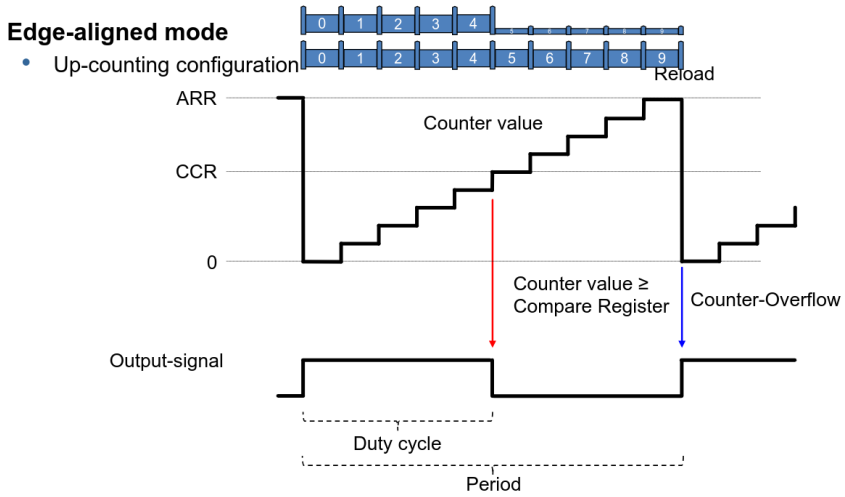
- **TXE** **TX Buffer Empty** **Software can write next TX Byte to register SPI_DR**
- **RXNE** **RX Buffer Not Empty** **A byte has been received. Software can read it from SPI_DR**





4 Timer

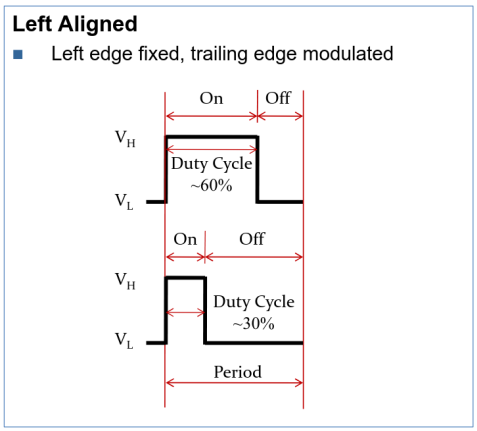


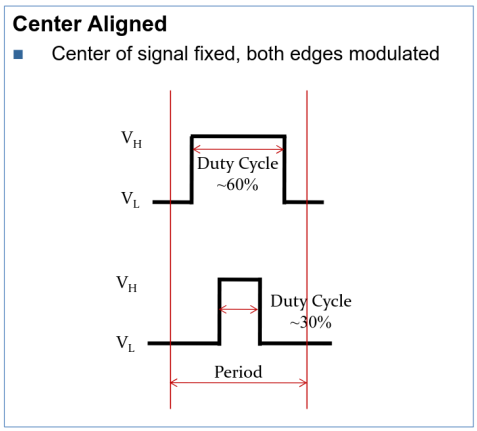


4.1 PWM

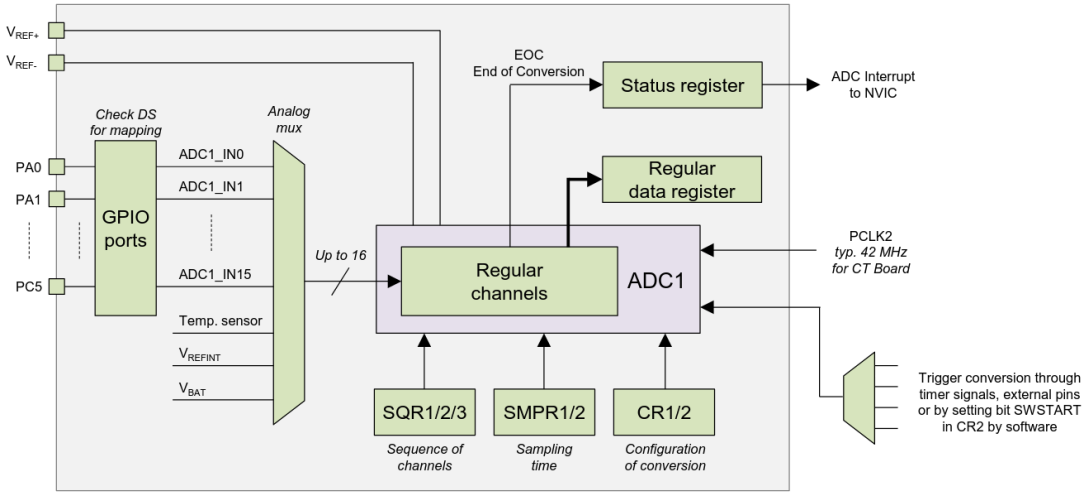
$$\text{Duty Cycle} = \frac{\text{On Time}}{\text{Period}} \cdot 100\ \%$$

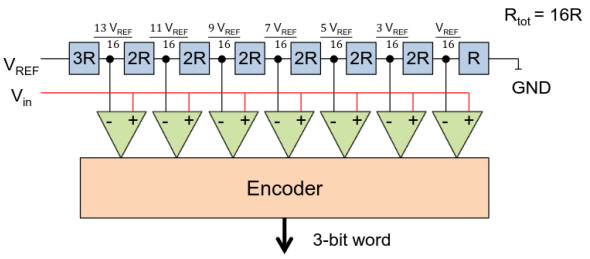
$$V_{avg} = D \cdot V_N + (1 - D) \cdot V_L$$

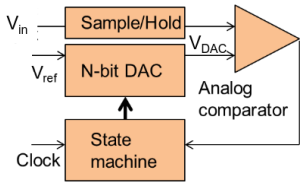




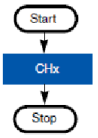
5 ADC





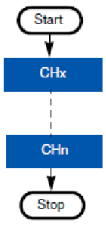


- **Single channel single conversion**
  - Possible use
    - Sensor tied to ADC input
    - On a push button you read and display it



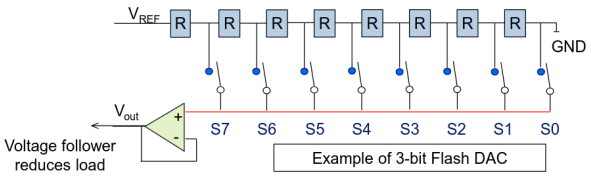
Channel programmed in SQR1/2/3

- **Multi-channel single conversion**
  - ADC sequencer used to set up order of conversions.
  - Possible use
    - Group of sensors need to be monitored
    - For each sensor there is an ADC input
    - Sensors are grouped and scanned one after the other every time a reading needs to be done

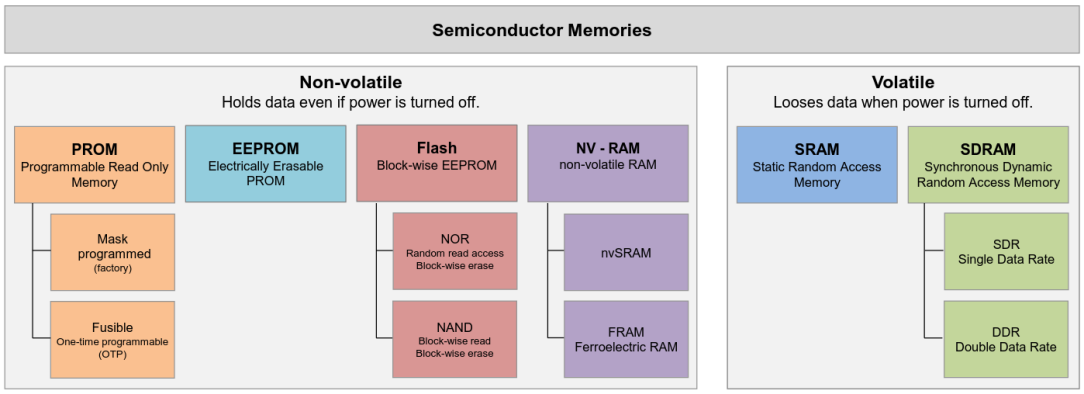


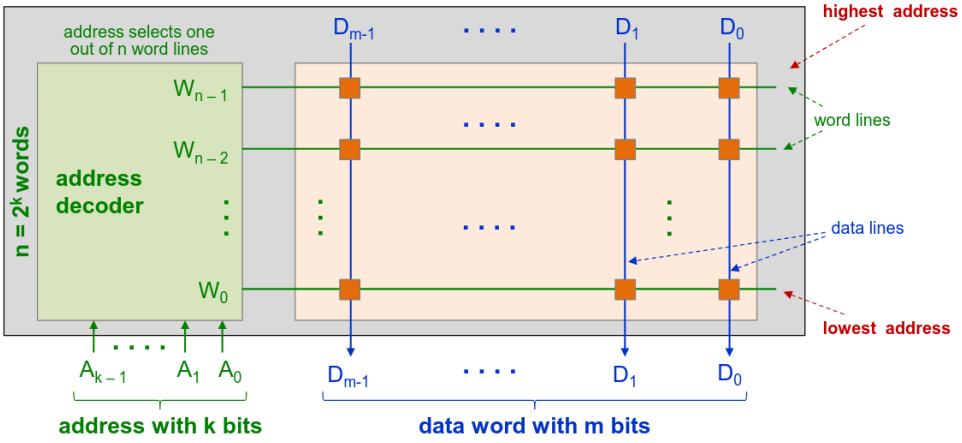
Sequence of channels programmed in SQR1/2/3

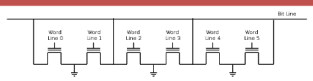
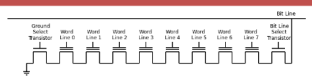
6 DAC



7 Memory

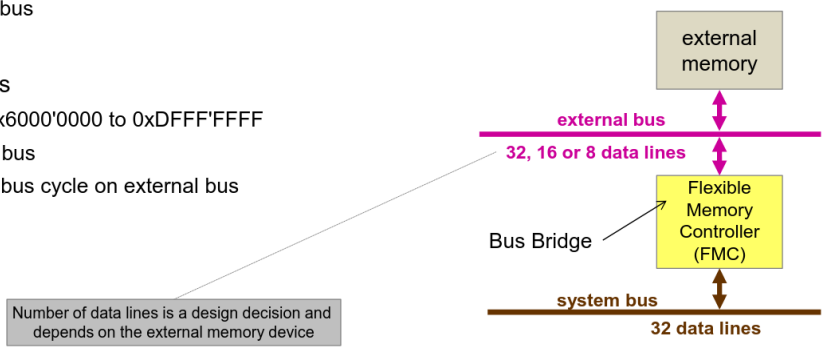


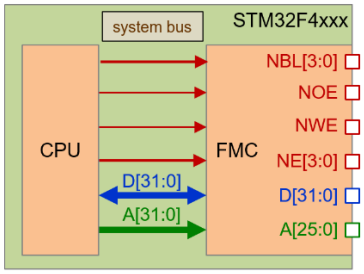


	NOR Flash	NAND Flash
Topology		
Applications	• **Execute code directly from memory** • **Persistent device configurations (replacement of EEPROM)**	• **File-based IO, disks** • **Large amounts of sequential data (images, SD cards, SSD)** • **Load programs into RAM before executing**
Density	• **Medium** Up to 2 GBit = 256 MByte	• **High** Up to 1 Tbit = 128 GByte
Interface	• Read same as asynchronous SRAM • Types with serial interface (SPI) available	• Special NAND flash interface • Error correction for defective blocks
Access	• Random access read ~0.12 µs • **Writing individual bytes possible** • Slow writes ~180 µs / 32 Byte	• **Slow random access** read: 1. Byte 25 µs, then 0.03 µs each • **Writing of individual bytes difficult** • **Fast block write** ~300 µs / 2'112 Bytes

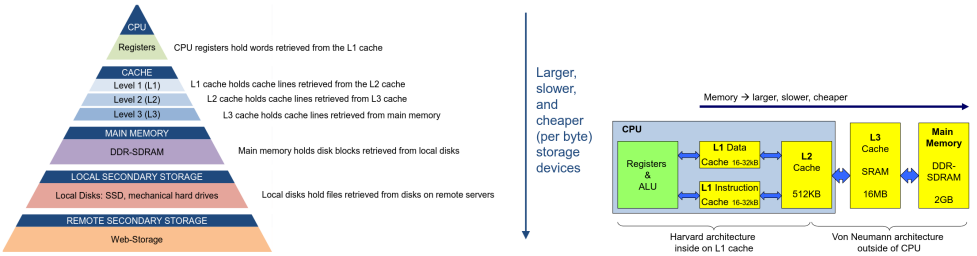
7.1 FMC

- Bridge between system bus and external bus
  - Slave on system bus
  - Master on external bus
- System bus accesses
  - In address range 0x6000'0000 to 0xDFFF'FFFF
  - Bridged to external bus
  - I.e. FMC initiates a bus cycle on external bus



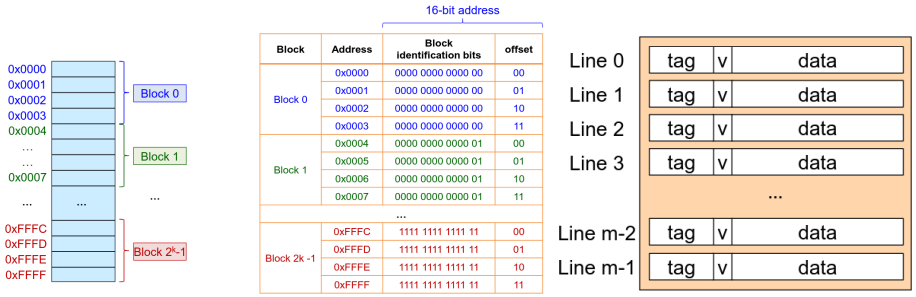


7.2 Cache

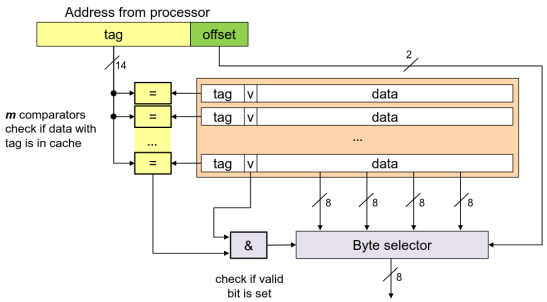


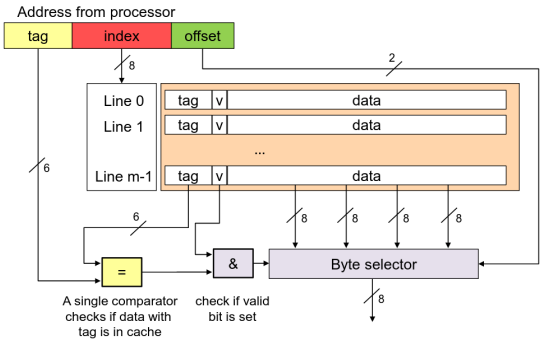
- **Spatial locality** → Current data location is likely being close to next accessed location
- **Temporal locality** → Current data location is likely being accessed again in near future

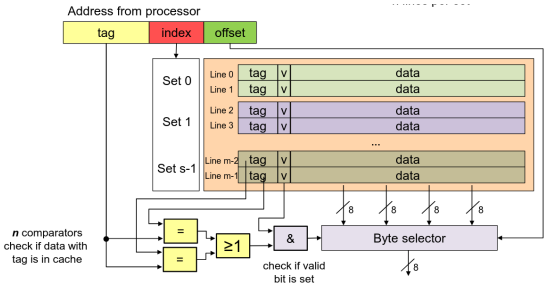
- Valid bit v → indicates that line contains valid data
- Tag → unique identifier for memory location (i.e. memory block)
- Data → data of exactly one memory block
- m = overall number of cache lines



Organization	Fully associative	Direct mapped	N-way set associative
Number of sets	1	m	m/n
Associativity	m(=n)	1	n
Advantages	• Fast, flexible • Highest hit rates • Advanced replacement strategies	• Simple logic • Replacement strategy defined by organization	Combination of both other concepts to combine advantages and to compensate disadvantages
Disadvantages	• Complex logic: one comparator per line • Requires large area on silicon • Replacement can be complex	• Lower hit rates	







Cold miss
- First access to a block

Capacity miss
- Working set larger than cache
  → the cache cannot hold all the data required by the system

Conflict miss
- Multiple data blocks map to same line or set
  → blocks compete for the same line or set in the cache

Hit rate / miss rate
- Fraction of memory references found / not found in cache
  - hit rate = nr_of_hits / nr_of_accesses
  - miss rate = nr_of_misses / nr_of_accesses = 1 – hit rate

Hit time
- Time to deliver a block in the cache to the processor

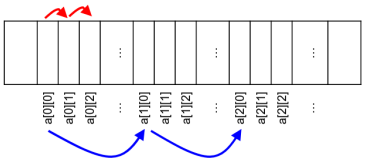
Miss penalty
- Additional time required to fetch data from memory because of a cache miss

High cache hit rate is important!
- 99% hit rate can be twice as fast as 97%

Example
- Cache hit time: 1 processor cycle
- Miss penalty: 100 processor cycles
- Average access time is:
  - 97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles average
  - 99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles average

```
for(j = 0; j < 10000; j++){
  for(i = 0; i < 40000; i++){
    c[i][j] = a[i][j] + b[i][j];
  }
}

// a[i][j] and a[i+1][j]
// are 10'000 elements apart
```

```
for(i = 0;i < 40000; i++){
  for(j = 0;j < 10000; j++){
    c[i][j] = a[i][j] + b[i][j];
  }
}

// a[i][j] and a[i][j+1]
// are next to each other
```

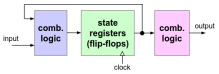


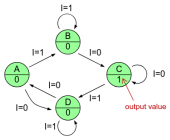
7.3 FSM

Terminology

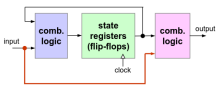
- State
  - Internal state of the system in which it is awaiting the next event
- Event
  - Asynchronous input that may cause a transition
- Transition
  - Reaction to an event: May change the state and/or trigger an action
- Action
  - Output associated with a transition
    - ► Either a directly carried out operation or a
    - ► Message to another FSM (seen as an event by the receiving FSM)
- Finite State Machine (FSM)
  - Machine with a finite number of states and transitions

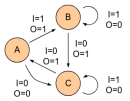
- **Moore**
  - Input signals influence state only

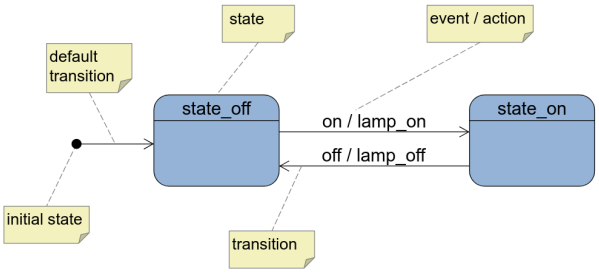




- **Mealy**
  - Input signals influence state AND output signals



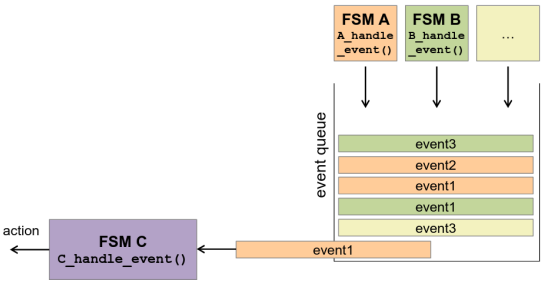





```
switch (state) {
    case STATE_A:
        switch (event) {
            case S0:
                action();
                state = NEW_STATE;
                break;
            default:
                state = WAIT;
        }
        break;

    case STATE_B:
        switch (event)
```

Event queue

- Collect events generated by different objects
- Buffered in event queue: Avoids losing events
- FSM processes one event after the other
- Events are deleted after processing



7.4 Interrupt Performance

Interrupt service time t_{ISR}

- Required time to process an interrupt
  - I.e. execution time of ISR
- Depends on
  - Number of instructions in ISR
  - Required number of clock cycles per instruction
    → depends on CPU architecture
  - CPU clock frequency
  - Time for switching to and returning from ISR

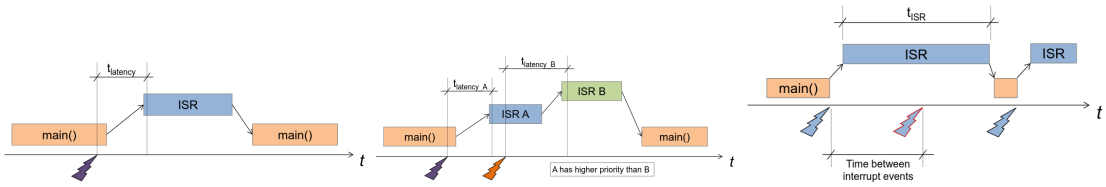
Interrupt frequency f_{INT}

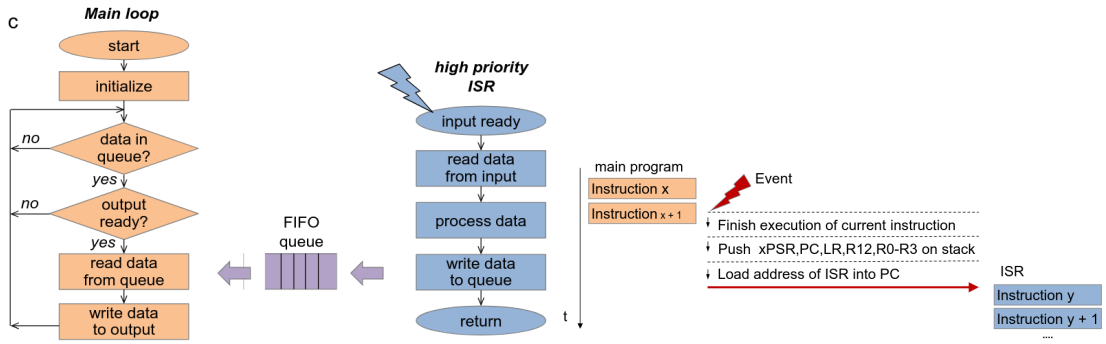
- How often does an interrupt occur
- Varies from source to source, e.g.

Impact on system performance

- Percentage of CPU time used to service interrupts

Impact = f_{INT} * t_{ISR} * 100 %



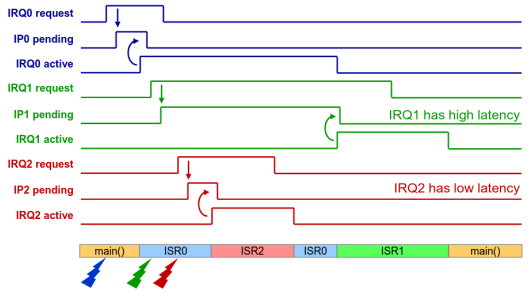


Example interrupt priorities

- ISR1 does not pre-empt ISR0
- ISR2 pre-empts ISR0

assuming

IRQ0	PL0 = 0x2	medium priority
IRQ1	PL1 = 0x3	lowest priority
IRQ2	PL2 = 0x1	highest priority



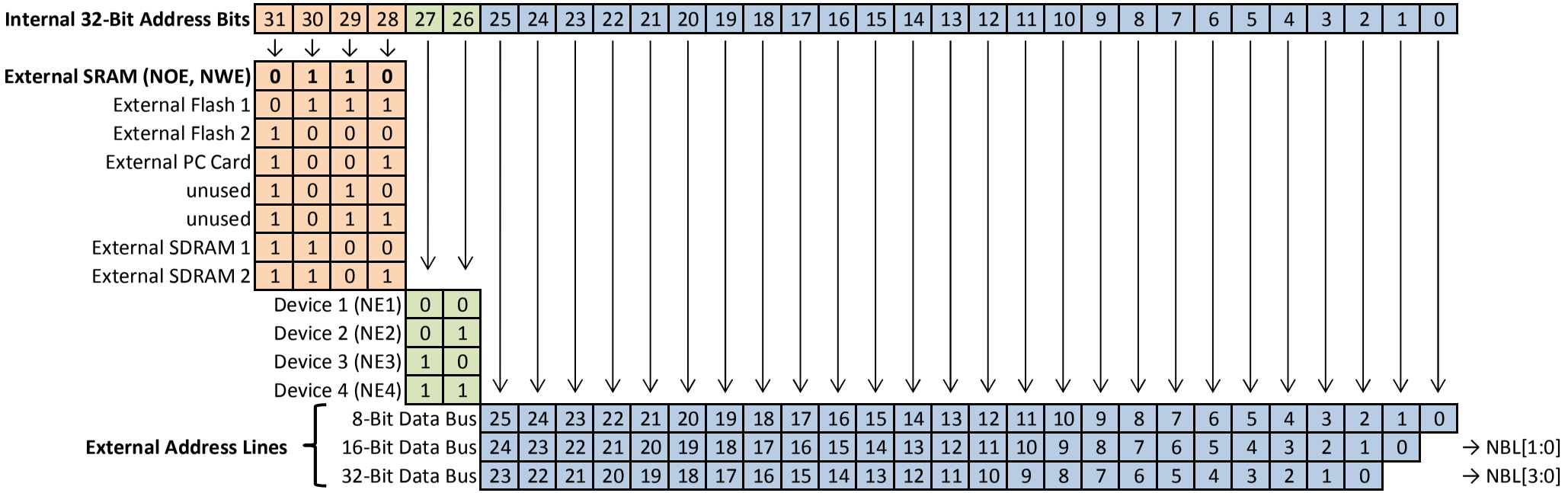
Required response time for a specific event

- Certain peripherals require fast response (< 1us)
  - Real-time systems, e.g. anti-lock braking system
- Cannot be guaranteed if maximum interrupt latency is too high

f_{INT} too high → Too many interrupts

- No CPU cycles left for data processing
  - System is busy calling ISRs
  - Neither ISR nor main loop can process any data
  - Performance of CPU is used only for latencies / context switching

STM32F429 Flexible Memory Controller (FMC) Decoding



A[31:28]

- 0x0 - 0x5 — Non-FMC access to chip internal peripherals like flash, SRAM, control registers
- 0x6 - 0xD — FMC access: Select one type of external memory
- 0xE - 0xF — Non-FMC access to chip internal ARM peripherals like NVIC

For external SRAM, A[27:26]

- 0x0 - 0x3 — Select one of 4 devices → controls NE[3:0]

For external SRAM, A[25:0]

- 8-Bit external data bus — Internal lines A[25:0] mapped to external A[25:0]; No NBL signals required
- 16-Bit external data bus — Internal lines A[25:1] mapped to external A[24:0]; Internal A[0] yields NBL[1:0]
- 32-Bit external data bus — Internal lines A[25:2] mapped to external A[23:0]; Internal A[1:0] yields NBL[3:0]

Datenblattauszug GPIO

Boundary address	Peripheral	Bus	Register map
0x4004 0000 - 0x4007 FFFF	USB OTG HS	AHB1	*Section 35.12.6: OTG_HS register map on page 1445*
0x4002 B000 - 0x4002 BBFF	DMA2D		*Section 11.5: DMA2D registers on page 349*
0x4002 9000 - 0x4002 93FF	ETHERNET MAC		*Section 33.8.5: Ethernet register maps on page 1214*
0x4002 8C00 - 0x4002 8FFF			
0x4002 8800 - 0x4002 8BFF			
0x4002 8400 - 0x4002 87FF			
0x4002 8000 - 0x4002 83FF			
0x4002 6400 - 0x4002 67FF	DMA2		*Section 10.5.11: DMA register map on page 332*
0x4002 6000 - 0x4002 63FF	DMA1		
0x4002 4000 - 0x4002 4FFF	BKPSRAM		
0x4002 3C00 - 0x4002 3FFF	Flash interface register		*Section 3.9: Flash interface registers*
0x4002 3800 - 0x4002 3BFF	RCC		*Section 7.3.25: RCC register map on page 263*
0x4002 3000 - 0x4002 33FF	CRC		*Section 4.4.4: CRC register map on page 114*
0x4002 2800 - 0x4002 2BFF	GPIOK		*Section 8.4.11: GPIO register map on page 284*
0x4002 2400 - 0x4002 27FF	GPIOJ		
0x4002 2000 - 0x4002 23FF	GPIOI		*Section 8.4.11: GPIO register map on page 284*
0x4002 1C00 - 0x4002 1FFF	GPIOH		
0x4002 1800 - 0x4002 1BFF	GPIOG		
0x4002 1400 - 0x4002 17FF	GPIOF		
0x4002 1000 - 0x4002 13FF	GPIOE		
0x4002 0C00 - 0x4002 0FFF	GPIOD		
0x4002 0800 - 0x4002 0BFF	GPIOC		
0x4002 0400 - 0x4002 07FF	GPIOB		
0x4002 0000 - 0x4002 03FF	GPIOA		
0x4001 6800 - 0x4001 6BFF	LCD-TFT	APB2	*Section 16.7.26: LTDC register map on page 504*
0x4001 5800 - 0x4001 5BFF	SAI1		*Section 29.17.9: SAI register map on page 944*
0x4001 5400 - 0x4001 57FF	SPI6	APB2	*Section 28.5.10: SPI register map on page 906*
0x4001 5000 - 0x4001 53FF	SPI5		

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

8.4.2 GPIO port output type register (GPIOx_OTYPER) (x = A..I/J/K)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OTy:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the output type of the I/O port.
0: Output push-pull (reset state)
1: Output open-drain

8.4.3 GPIO port output speed register (GPIOx_OSPEEDR) (x = A..I/J/K)

Address offset: 0x08

Reset values:

- 0x0000 00C0 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1 [1:0]		OSPEEDR0 1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **OSPEEDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O output speed.
00: Low speed
01: Medium speed
10: Fast speed
11: High speed

Note: Refer to the product datasheets for the values of OSPEEDRy bits versus V_{DD} range and external load.

8.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A..I/J/K)

Address offset: 0x0C

Reset values:

- 0x6400 0000 for port A
- 0x0000 0100 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **PUPDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

13.13.14 ADC regular data register (ADC_DR)

Address offset: 0x4C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **DATA[15:0]:** Regular data

These bits are read-only. They contain the conversion result from the regular channels. The data are left- or right-aligned as shown in *Figure 48* and *Figure 49*.

13.13.1 ADC status register (ADC_SR)

Address offset: 0x00

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										OVR	STRT	JSTRT	JEOC	EOC	AWD
										rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Bits 31:6 Reserved, must be kept at reset value.

Bit 5 **OVR:** Overrun

This bit is set by hardware when data are lost (either in single mode or in dual/triple mode). It is cleared by software. Overrun detection is enabled only when DMA = 1 or EOCS = 1.

0: No overrun occurred

1: Overrun has occurred

Bit 4 **STRT:** Regular channel start flag

This bit is set by hardware when regular channel conversion starts. It is cleared by software.

0: No regular channel conversion started

1: Regular channel conversion has started

Bit 3 **JSTRT:** Injected channel start flag

This bit is set by hardware when injected group conversion starts. It is cleared by software.

0: No injected group conversion started

1: Injected group conversion has started

Bit 2 **JEOC:** Injected channel end of conversion

This bit is set by hardware at the end of the conversion of all injected channels in the group. It is cleared by software.

0: Conversion is not complete

1: Conversion complete

Bit 1 **EOC:** Regular channel end of conversion

This bit is set by hardware at the end of the conversion of a regular group of channels. It is cleared by software or by reading the ADC_DR register.

0: Conversion not complete (EOCS=0), or sequence of conversions not complete (EOCS=1)

1: Conversion complete (EOCS=0), or sequence of conversions complete (EOCS=1)

Bit 0 **AWD:** Analog watchdog flag

This bit is set by hardware when the converted voltage crosses the values programmed in the ADC_LTR and ADC_HTR registers. It is cleared by software.

0: No analog watchdog event occurred

1: Analog watchdog event occurred

13.4 Data alignment

The ALIGN bit in the ADC_CR2 register selects the alignment of the data stored after conversion. Data can be right- or left-aligned as shown in *Figure 48* and *Figure 49*.

The converted data value from the injected group of channels is decreased by the user-defined offset written in the ADC_JOFRx registers so the result can be a negative value. The SEXT bit represents the extended sign value.

For channels in a regular group, no offset is subtracted so only twelve bits are significant.

Figure 48. Right alignment of 12-bit data

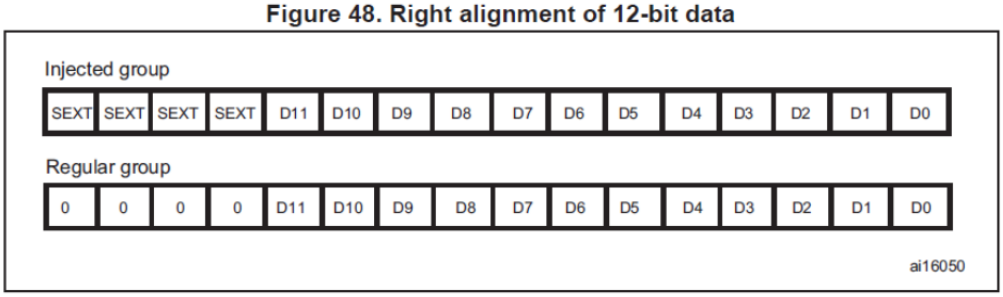
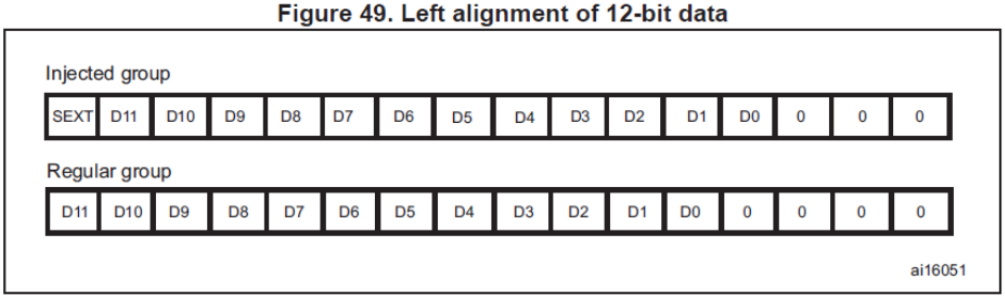


Figure 49. Left alignment of 12-bit data



Special case: when left-aligned, the data are aligned on a half-word basis except when the resolution is set to 6-bit. in that case, the data are aligned on a byte basis as shown in *Figure 50*.

ADC Base Address			Address of Register
0x4001 2000	ADC1	Specific registers	0x4001 2000 + 0x000 + register offset
	ADC2	Specific registers	0x4001 2000 + 0x100 + register offset
	ADC3	Specific registers	0x4001 2000 + 0x200 + register offset
	Common	Common registers	0x4001 2000 + 0x300 + register offset

C Reference Card (ANSI)

Program Structure/Functions

type fnc(*type*$_1$, ...);	function prototype
type name;	variable declaration
`int main(void) {`	main routine
declarations	local variable declarations
statements	
`}`	
type fnc(*arg*$_1$, ...) {	function definition
declarations	local variable declarations
statements	
`return` *value*;	
`}`	
`/* */`	comments
`int main(int argc, char *argv[])`	main with args
`exit(`*arg*`);`	terminate execution

C Preprocessor

include library file	`#include <`*filename*`>`
include user file	`#include "`*filename*`"`
replacement text	`#define` *name text*
replacement macro	`#define` *name*`(`*var*`)` *text*
Example. `#define max(A,B) ((A)>(B) ? (A) : (B))`	
undefine	`#undef` *name*
quoted string in replace	`#`
Example. `#define msg(A) printf("%s = %d", #A, (A))`	
concatenate args and rescan	`##`
conditional execution	`#if, #else, #elif, #endif`
is *name* defined, not defined?	`#ifdef, #ifndef`
name defined?	`defined(`*name*`)`
line continuation char	`\`

Data Types/Declarations

character (1 byte)	`char`
integer	`int`
real number (single, double precision)	`float, double`
short (16 bit integer)	`short`
long (32 bit integer)	`long`
double long (64 bit integer)	`long long`
positive or negative	`signed`
non-negative modulo 2^m	`unsigned`
pointer to `int`, `float`,...	`int*, float*,...`
enumeration constant	`enum` *tag* `{`*name*$_1$`=`*value*$_1$`,...};`
constant (read-only) value	*type* `const` *name*`;`
declare external variable	`extern`
internal to source file	`static`
local persistent between calls	`static`
no value	`void`
structure	`struct` *tag* `{...};`
create new name for data type	`typedef` *type name*`;`
size of an object (type is `size_t`)	`sizeof` *object*
size of a data type (type is `size_t`)	`sizeof(`*type*`)`

Initialization

initialize variable	*type name*`=`*value*`;`
initialize array	*type name*`[]={`*value*$_1$`,...};`
initialize char string	`char` *name*`[]="`*string*`";`

Constants

suffix: long, unsigned, float	`65536L, -1U, 3.0F`
exponential form	`4.2e1`
prefix: octal, hexadecimal	`0, 0x` or `0X`
Example. `031` is 25, `0x31` is 49 decimal	
character constant (char, octal, hex)	`'a', '\`*ooo*`', '\x`*hh*`'`
newline, cr, tab, backspace	`\n, \r, \t, \b`
special characters	`\\, \?, \', \"`
string constant (ends with `'\0'`)	`"abc...de"`

Pointers, Arrays & Structures

declare pointer to *type*	*type* `*`*name*`;`
declare function returning pointer to *type*	*type* `*f();`
declare pointer to function returning *type*	*type* `(*pf)();`
generic pointer type	`void *`
null pointer constant	`NULL`
object pointed to by *pointer*	`*`*pointer*
address of object *name*	`&`*name*
array	*name*`[`*dim*`]`
multi-dim array	*name*`[`*dim*$_1$`][`*dim*$_2$`]...`

Structures

`struct` *tag* `{`	structure template
declarations	declaration of members
`};`	

create structure	`struct` *tag name*
member of structure from template	*name*`.`*member*
member of pointed-to structure	*pointer* `->` *member*
Example. `(*p).x` and `p->x` are the same	
single object, multiple possible types	`union`
bit field with *b* bits	`unsigned` *member*`:` *b*`;`

Operators (grouped by precedence)

struct member operator	*name*`.`*member*
struct member through pointer	*pointer*`->`*member*
increment, decrement	`++, --`
plus, minus, logical not, bitwise not	`+, -, !, ~`
indirection via pointer, address of object	`*`*pointer*`, &`*name*
cast expression to type	`(`*type*`)` *expr*
size of an object	`sizeof`
multiply, divide, modulus (remainder)	`*, /, %`
add, subtract	`+, -`
left, right shift [bit ops]	`<<, >>`
relational comparisons	`>, >=, <, <=`
equality comparisons	`==, !=`
and [bit op]	`&`
exclusive or [bit op]	`^`
or (inclusive) [bit op]	`|`
logical and	`&&`
logical or	`||`
conditional expression	*expr*$_1$ `?` *expr*$_2$ `:` *expr*$_3$
assignment operators	`+=, -=, *=, ...`
expression evaluation separator	`,`

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Flow of Control

statement terminator	`;`
block delimiters	`{ }`
exit from `switch`, `while`, `do`, `for`	`break;`
next iteration of `while`, `do`, `for`	`continue;`
go to	`goto` *label*`;`
label	*label*`:` `statement`
return value from function	`return` *expr*

Flow Constructions

`if` statement	`if (`*expr*$_1$`)` *statement*$_1$ `else if (`*expr*$_2$`)` *statement*$_2$ `else` *statement*$_3$
`while` statement	`while (`*expr*`)` *statement*
`for` statement	`for (`*expr*$_1$`;` *expr*$_2$`;` *expr*$_3$`)` *statement*
`do` statement	`do` *statement* `while(`*expr*`);`

`switch` statement	`switch (`*expr*`) {`
	`case` *const*$_1$`:` *statement*$_1$ `break;`
	`case` *const*$_2$`:` *statement*$_2$ `break;`
	`default:` *statement*
	`}`

ANSI Standard Libraries

`<assert.h>`	`<ctype.h>`	`<errno.h>`	`<float.h>`	`<limits.h>`
`<locale.h>`	`<math.h>`	`<setjmp.h>`	`<signal.h>`	`<stdarg.h>`
`<stddef.h>`	`<stdio.h>`	`<stdlib.h>`	`<string.h>`	`<time.h>`

Character Class Tests `<ctype.h>`

alphanumeric?	`isalnum(c)`
alphabetic?	`isalpha(c)`
control character?	`iscntrl(c)`
decimal digit?	`isdigit(c)`
printing character (not incl space)?	`isgraph(c)`
lower case letter?	`islower(c)`
printing character (incl space)?	`isprint(c)`
printing char except space, letter, digit?	`ispunct(c)`
space, formfeed, newline, cr, tab, vtab?	`isspace(c)`
upper case letter?	`isupper(c)`
hexadecimal digit?	`isxdigit(c)`
convert to lower case	`tolower(c)`
convert to upper case	`toupper(c)`

String Operations `<string.h>`

`s` is a string; `cs`, `ct` are constant strings

length of `s`	`strlen(s)`
copy `ct` to `s`	`strcpy(s,ct)`
concatenate `ct` after `s`	`strcat(s,ct)`
compare `cs` to `ct`	`strcmp(cs,ct)`
only first `n` chars	`strncmp(cs,ct,n)`
pointer to first `c` in `cs`	`strchr(cs,c)`
pointer to last `c` in `cs`	`strrchr(cs,c)`
copy `n` chars from `ct` to `s`	`memcpy(s,ct,n)`
copy `n` chars from `ct` to `s` (may overlap)	`memmove(s,ct,n)`
compare `n` chars of `cs` with `ct`	`memcmp(cs,ct,n)`
pointer to first `c` in first `n` chars of `cs`	`memchr(cs,c,n)`
put `c` into first `n` chars of `s`	`memset(s,c,n)`

C Reference Card (ANSI)

Input/Output <stdio.h>

Standard I/O

standard input stream	stdin
standard output stream	stdout
standard error stream	stderr
end of file (type is int)	EOF
get a character	getchar()
print a character	putchar(*chr*)
print formatted data	printf("*format*",arg_1,...)
print to string s	sprintf(s,"*format*",arg_1,...)
read formatted data	scanf("*format*",&$name_1$,...)
read from string s	sscanf(s,"*format*",&$name_1$,...)
print string s	puts(s)

File I/O

declare file pointer	FILE *fp;
pointer to named file	fopen("*name*","*mode*")
modes: r (read), w (write), a (append), b (binary)	
get a character	getc(*fp*)
write a character	putc(*chr*,*fp*)
write to file	fprintf(*fp*,"*format*",arg_1,...)
read from file	fscanf(*fp*,"*format*",arg_1,...)
read and store n elts to *ptr	fread(*ptr,eltsize,n,*fp*)
write n elts from *ptr to file	fwrite(*ptr,eltsize,n,*fp*)
close file	fclose(*fp*)
non-zero if error	ferror(*fp*)
non-zero if already reached EOF	feof(*fp*)
read line to string s (< max chars)	fgets(s,max,*fp*)
write string s	fputs(s,*fp*)

Codes for Formatted I/O: "%-+ 0*w.pmc*"

-	left justify
+	print with sign
space	print space if no sign
0	pad with leading zeros
w	min field width
p	precision
m	conversion character: h short, l long, L long double
c	conversion character:

d,i	integer	u	unsigned
c	single char	s	char string
f	double (printf)	e,E	exponential
f	float (scanf)	lf	double (scanf)
o	octal	x,X	hexadecimal
p	pointer	n	number of chars written
g,G	same as f or e,E depending on exponent		

Variable Argument Lists <stdarg.h>

declaration of pointer to arguments	va_list *ap*;
initialization of argument pointer	va_start(*ap*,*lastarg*);
lastarg is last named parameter of the function	
access next unnamed arg, update pointer	va_arg(*ap*,*type*)
call before exiting function	va_end(*ap*);

Standard Utility Functions <stdlib.h>

absolute value of int n	abs(n)
absolute value of long n	labs(n)
quotient and remainder of ints n,d	div(n,d)
returns structure with div_t.quot and div_t.rem	
quotient and remainder of longs n,d	ldiv(n,d)
returns structure with ldiv_t.quot and ldiv_t.rem	
pseudo-random integer [0,RAND_MAX]	rand()
set random seed to n	srand(n)
terminate program execution	exit(status)
pass string s to system for execution	system(s)

Conversions

convert string s to double	atof(s)
convert string s to integer	atoi(s)
convert string s to long	atol(s)
convert prefix of s to double	strtod(s,&endp)
convert prefix of s (base b) to long	strtol(s,&endp,b)
same, but unsigned long	strtoul(s,&endp,b)

Storage Allocation

allocate storage	malloc(size), calloc(nobj,size)
change size of storage	newptr = realloc(ptr,size);
deallocate storage	free(ptr);

Array Functions

search array for key	bsearch(key,array,n,size,cmpf)
sort array ascending order	qsort(array,n,size,cmpf)

Time and Date Functions <time.h>

processor time used by program	clock()
Example. clock()/CLOCKS_PER_SEC is time in seconds	
current calendar time	time()
$time_2$-$time_1$ in seconds (double)	difftime($time_2$,$time_1$)
arithmetic types representing times	clock_t,time_t
structure type for calendar time comps	struct tm

tm_sec	seconds after minute
tm_min	minutes after hour
tm_hour	hours since midnight
tm_mday	day of month
tm_mon	months since January
tm_year	years since 1900
tm_wday	days since Sunday
tm_yday	days since January 1
tm_isdst	Daylight Savings Time flag

convert local time to calendar time	mktime(tp)
convert time in tp to string	asctime(tp)
convert calendar time in tp to local time	ctime(tp)
convert calendar time to GMT	gmtime(tp)
convert calendar time to local time	localtime(tp)
format date and time info	strftime(s,smax,"*format*",tp)
tp is a pointer to a structure of type tm	

Mathematical Functions <math.h>

Arguments and returned values are double

trig functions	sin(x), cos(x), tan(x)
inverse trig functions	asin(x), acos(x), atan(x)
arctan(y/x)	atan2(y,x)
hyperbolic trig functions	sinh(x), cosh(x), tanh(x)
exponentials & logs	exp(x), log(x), log10(x)
exponentials & logs (2 power)	ldexp(x,n), frexp(x,&e)
division & remainder	modf(x,ip), fmod(x,y)
powers	pow(x,y), sqrt(x)
rounding	ceil(x), floor(x), fabs(x)

Integer Type Limits <limits.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system, followed by minimum required values (if significantly different).

CHAR_BIT	bits in char	(8)
CHAR_MAX	max value of char	(SCHAR_MAX or UCHAR_MAX)
CHAR_MIN	min value of char	(SCHAR_MIN or 0)
SCHAR_MAX	max signed char	(+127)
SCHAR_MIN	min signed char	(−128)
SHRT_MAX	max value of short	(+32,767)
SHRT_MIN	min value of short	(−32,768)
INT_MAX	max value of int	(+2,147,483,647) (+32,767)
INT_MIN	min value of int	(−2,147,483,648) (−32,767)
LONG_MAX	max value of long	(+2,147,483,647)
LONG_MIN	min value of long	(−2,147,483,648)
UCHAR_MAX	max unsigned char	(255)
USHRT_MAX	max unsigned short	(65,535)
UINT_MAX	max unsigned int	(4,294,967,295) (65,535)
ULONG_MAX	max unsigned long	(4,294,967,295)

Float Type Limits <float.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system.

FLT_RADIX	radix of exponent rep	(2)
FLT_ROUNDS	floating point rounding mode	
FLT_DIG	decimal digits of precision	(6)
FLT_EPSILON	smallest x so $1.0\text{f} + x \neq 1.0\text{f}$	($1.1\text{E} - 7$)
FLT_MANT_DIG	number of digits in mantissa	
FLT_MAX	maximum float number	(3.4E38)
FLT_MAX_EXP	maximum exponent	
FLT_MIN	minimum float number	($1.2\text{E} - 38$)
FLT_MIN_EXP	minimum exponent	
DBL_DIG	decimal digits of precision	(15)
DBL_EPSILON	smallest x so $1.0 + x \neq 1.0$	($2.2\text{E} - 16$)
DBL_MANT_DIG	number of digits in mantissa	
DBL_MAX	max double number	(1.8E308)
DBL_MAX_EXP	maximum exponent	
DBL_MIN	min double number	($2.2\text{E} - 308$)
DBL_MIN_EXP	minimum exponent	

Send comments and corrections to J.H. Silverman, Math. Dept., Brown Univ., Providence, RI 02912 USA. ⟨jhs@math.brown.edu⟩