Micro Controllers Summary

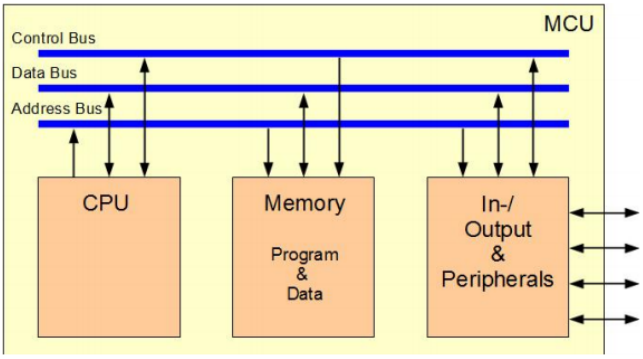
Lucien Zürcher

June 21, 2019

Contents

1 System Components

1.1 Von Neumann Architecture



Components:

- **CPU**, Central Processing Unit
- **Memory**, Program and Data
- **In-/Output**-Unit, Peripherals
- **Bus**-System: Communication

*One **shared bus and memory** for program and data.*

1.2 Harvard-Architecture

*basically same as Von Neumann, with the difference, that there are **two separate bus systems** for program and data*

1.3 Numerical Systems

Numerical value Z_B of a n-digit, integer number with base B ($B \geq 2$):

$$Z_B = \sum_{i=0}^{n-1} x_i \cdot B^i$$

Decimal	**Dual / Binary**	**Hexadecimal**
197	0b1100'0101	0xC5
$B = 10$	$B = 2$	$B = 16$
$= 1 \cdot 10^2 +$ $9 \cdot 10^1 +$ $7 \cdot 10^0$	$= 1 \cdot 2^7 + 1 \cdot 2^6 +$ $0 \cdot 2^5 + 0 \cdot 2^4 +$ $0 \cdot 2^3 + 1 \cdot 2^2 +$ $0 \cdot 2^1 + 1 \cdot 2^0$	$= C \cdot 16^1 + 5 \cdot 16^0$ $= 12 \cdot 16^1 + 5 \cdot 16^0$

The amount of presentable numbers is B^n The highest presentable number is $B^n - 1$. Calculated from $x_i = B - 1$ for $n - 1 \geq i \geq 0$

1.4 hex / binary

H	D	B	Dec	Bin	
0	0	0000	16	2^5	(max 31)
1	1	0001	32	2^6	(max 63)
2	2	0010	64	2^7	(max 127)
3	3	0100	128	2^8	(max 255)
4	4	0101	256	2^9	(max 511)
5	5	0110	512	2^{10}	(max 1'023)
6	6	0111	1'024	2^{11}	(max 2'047)
7	7	1000	2'048	2^{12}	(max 4'095)
9	9	1001	4'096	2^{13}	(max 8'191)
A	10	1010	8'192	2^{14}	(max 16'383)
B	11	1011	16'384	2^{15}	(max 31'767)
C	12	1110	32'768	2^{16}	(max 65'535)
D	13	1011			
E	14	1011			
F	15	1011			

1.5 Signed numbers

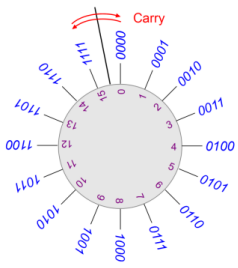
two's compliment is beeing used

$$Z_{signed} = -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$$

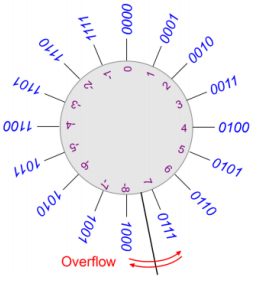
most significant bit is negative
Example: -1 as 16-bit Hex = $0xFFFF$
Conversion:

1. *Invert binary :* $-6 \rightarrow 0110 \rightarrow 1001$
2. *increment by 1 :* $1001 + 0001 \rightarrow 1010$

1.6 carry / overflow



***Carry** is set on crossover between lowest and highest number*

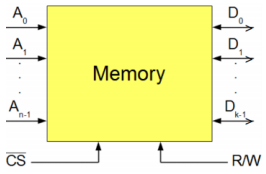


***Overflow** happens on crossover between highest absolut values*

1.7 Bit groups

***Nibble/Tetrade** has the size of 4 bits*
***Byte** has the size of 8 bits*
***Word** is MC9S08JM60 specific, it has 16 bits*

1.8 Quantity of address lines



$\mathbf{n}$ *Quantity of address lines*
$\mathbf{2^n}$ *Quant. of storage locations*
$\mathbf{2^n \cdot k}$ *Quant. of bits in memory*

1 K	=	2^{10}	= 1024 Bit	≙ 10 Adresslines
64 K	=	2^{16}	= 65536 Bit	≙ 16 Adresslines

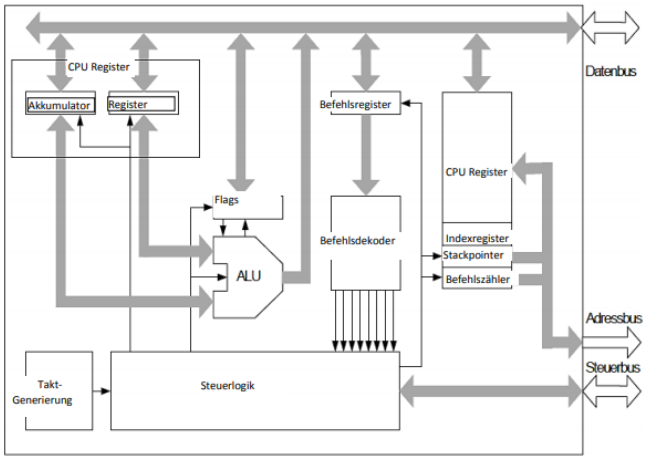
example, $32K \times 8$ memory storage space:

bits storage: $32 \cdot 2^10 \cdot 8 = 2^5 \cdot 2^10 \cdot 2^3 = 2^18 \rightarrow 18$ *Bits*
number address lines: $32 \cdot 2^10 = 2^15 = 32\,768$
highest address: $2^{18} - 1 = 0x7FFFF = 262'143$

1.9 Microprocessor vs Mircocontroller

Mircocontroller *contains CPU (Processor), Peripherals (I/O) and Memory (RAM / ROM). Basically a small computer.*
Mircoprocessor *has only CPU and som integrated Circuits.*

1.10 CPU components



ALU (Aritmetic Unit), AKKU (Accumulator), PC (Programming Counter), Busses, Instruction-Register, Address-Register, Operand-Register, Control Unit, ..

1.11 Instruction Cycle Steps

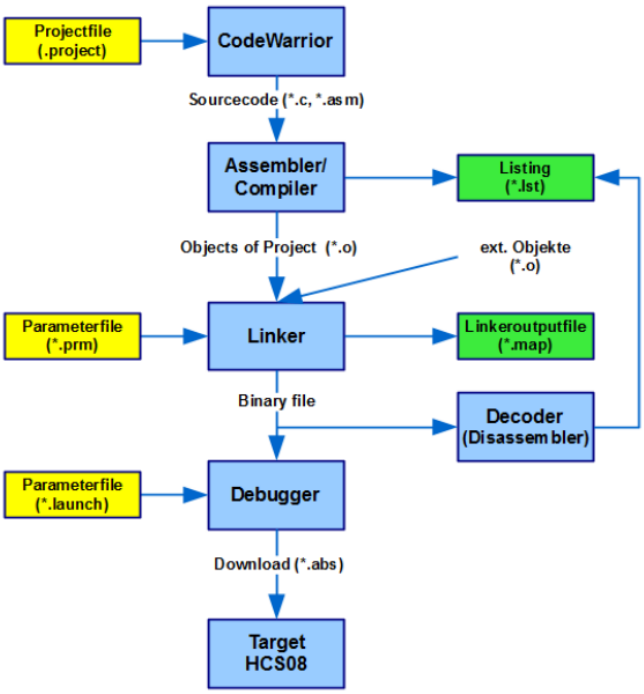
1. instruction fetch
2. instruction decode
3. (operand fetch)
4. instruction execute
5. next address and inc PC

1.12 Types of MCU Registers

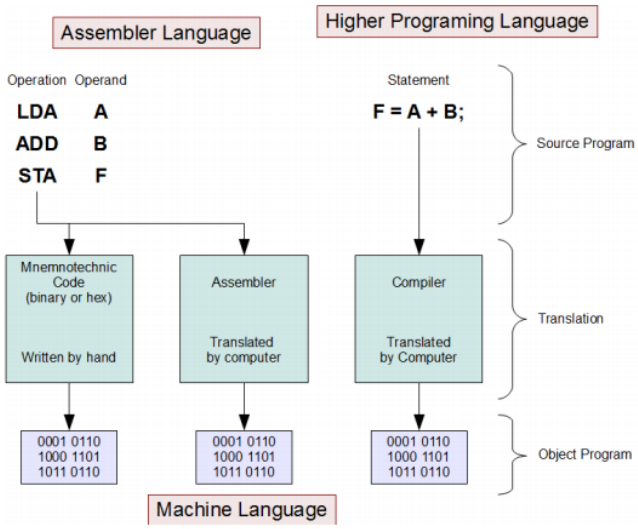
AKKU, PC, Instruction-Register (decoder), Operand-Register

2 Compiling

2.1 Codewarrior Designflow



2.2 Programming Language



High level programming languages are:

- portable
- efficient (normaly)
- Better readable
- easier to maintain

High level programming languages are usually prefered, if enough computational power and memory is available. Assembler is often used, if the application:

- is time critical and needs exact timing
- timing of the high level programming language to unpredictible is

2.3 Assembler Code-Format

	Label	Instruction	Operands	comment
Ex1	Limit:	EQU	$CD	; define limit
Ex2	Start:	LDA	#Limit	; load limit

***Instruction**: is a command for the processor*
***Directive**: are instructions that direct the assembler / compiler to do something*

	Type	Directed to	Results in program code
Ex1	**Instruction**	Target CPU	Yes
Ex2	**Directive**	Assembler	Only indirect
	Comment	Programmer	No

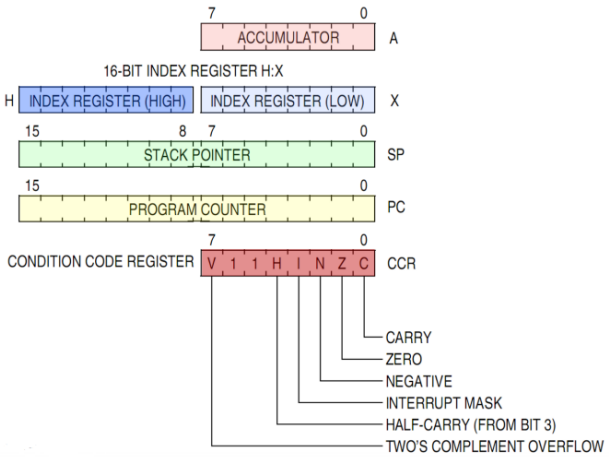
2.4 Parameter file

The Parameter file (.prm) is used for by the Linker. It takes the machine code and defines the location on the controller. It is important, so that jumps work correctly. It contains:*

- Memory-Map of the Prozessor (Location and size of Flash, RAM, ..)
- Extra definitions, where which parts of the code on the Controller should be located

3 Assembler & HCS08

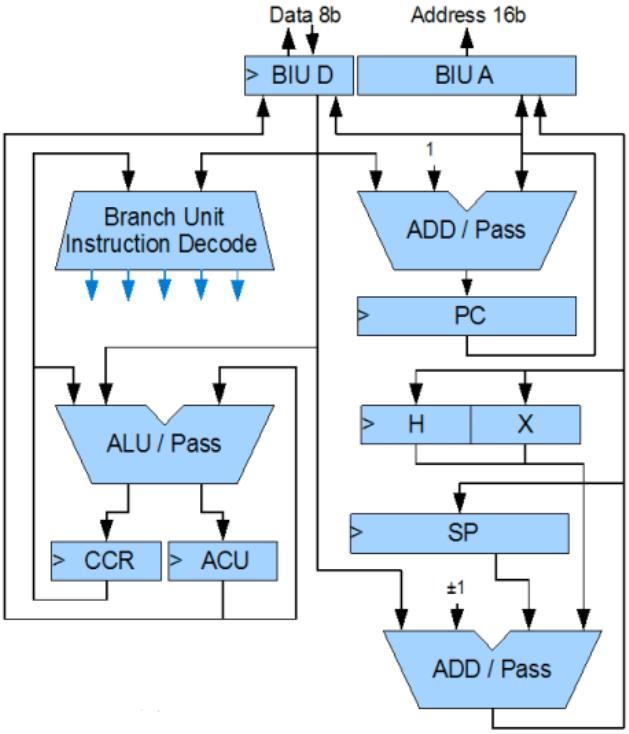
3.1 HCS08 CPU Registers



Registers the HCS08 contains:

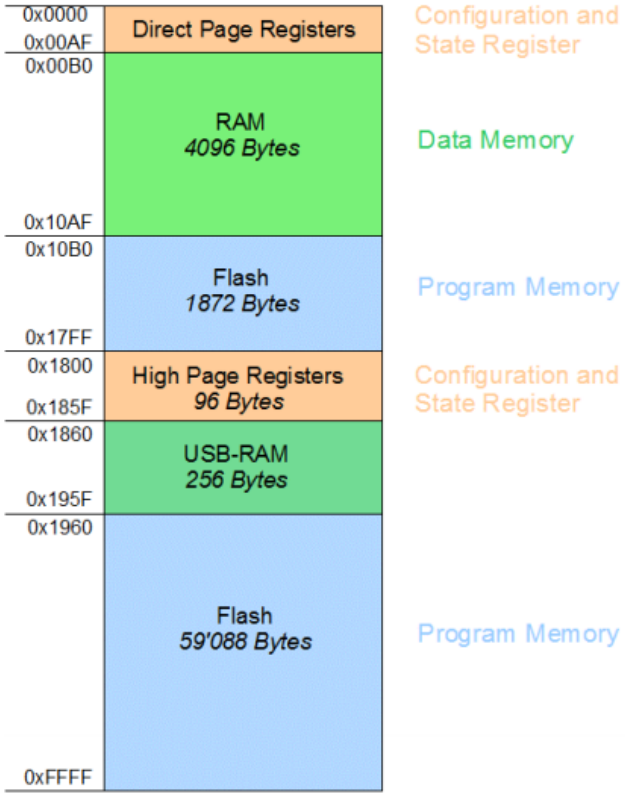
- HX Register
- PC
- Akku
- Stack Pointer
- CCR

3.2 HCS08 Processor



- 8 Bit, Von Neumann archidecture
- **BIU** Bus Interface Unit
- **PC** Program Counter
- **ACU** Accumulator
- **ALU** Arithmetic Logic Unit
- **CCR** Condition Code Register (Collection of status flags)
- **SP** Stack (LI-FO, Pointer for Context and Parameter)
- **H:X** Index Register

3.3 Memory Mapping



Access to the directpage (0x0000 - 0x0AF) needs less cycles, since the address is only 1 Bytes long.

3.4 Register configuration HCS08

```
// define the dataflow direction input = 0 |
    output = 1
PTADD = 0x04;

// set output value
PTAD = 0x04;
// read value
uint_8 val = PTAD;

// set pullup enable port
PTADD = 0x00;
PTAPE = 0x04;
```

Reg. Name	Description
PTxDD	*Data Direction of Port x*
PTxD	*Data value of Port x*
PTxPE	*Set Pullup Enable of Port x (PTxDD needs to be 0)*

Pullup Enable *is used to pullup the value of the output to 1. This is usually used on a bus system to prevent a short circuit.*

3.5 Differences of Operations

Comparing different operations, following should be taken in consideration:

- number of cycles
- memory usage, 8bit (directpage) / 16bit
- Set CCR bits / flags
- Used registers
- Address modes

4 Assembler Directives & Addressing Modes

4.1 Directives

Directive	Description
SECTION	*Defines the beginning of a relocatable section*
EQU	*Assigns an expression to a name. Not redefinable*
DC	*Defines one or more constants and their names. Will be stored at the set location*
DS	*Allocates memory(RAM) for variables*

The Assembler-Directive **SECTION** *defines program- and data section. Those section can be moved freely within the memory (relocative assembling),* ***after*** *the* ***assembly*** *process is finished.*
The final memory area location happens after the linking process. The locations of those sections can therefor be defined in the ***Linker-Parameterfile****.*

4.2 Basic Assembler Program

```
; include definitions
include 'MC9S08JM60.inc'

; -- globals
GLOBAL _Startup ; define start of programm
GLOBAL main
GLOBAL dummy    ; Dummy Interrupt Service
    Routine

; -- equations
StackSize: EQU   $60   ; stack size
pi:        EQU   31416 ; example of random equ

; -- stack
DATA_STACK: SECTION
TofStack:  DS    StackSize-1 ; definiton of "
    Top of Stack"
BofStack:  DS    1           ; definition of "
    Bottom of Stack"

; -- create space for data
DATA:   SECTION
var1:   DS    1   ; Example of a 1 Byte
    Variable
Array1: DS    $20 ; Example of an Array of $20
    Bytes

; -- setup constants
CONST:      SECTION
Maske1:       DC.B    %00000001
Parameter1:   DC.B    $3A    ; DC with a point
Parameter2:   DC.W    57100  ; word with int
    value
Reserve_Par: DS       16     ; reserve empty 16
    Bytes
VarArray:     DS.W    3      ; reserve 3 Words
STRING1:      DC.B    10,"Hello",$0D

; -- program start (initialisation)
PROGRAMM:   SECTION ; Code Segment
```

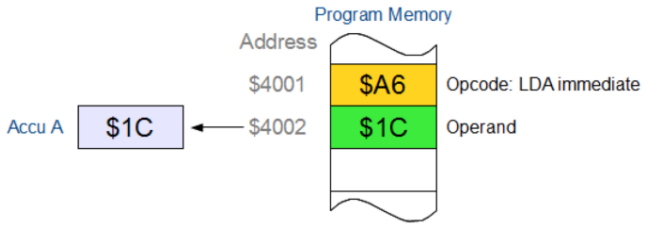
```
_Startup:           ; Resetvektor points to
    this
Stackinit:  LDHX   #(BofStack+1)
            TXS           ; decrement TXS, thats
                why +1 BofStack
            LDA    #$00
            STA    SOPT1   ; Disable Watchdog

; -- actual program
main:
    ; turn on backligths of the car
    BSET    PTDD_PTDD2, PTDD
    BSET    PTDDD_PTDDD2, PTDDD

    CLR     RamLoc

    BCLR    PTGDD_PTGDD0, PTGDD
    BCLR    PTGDD_PTGDD1, PTGDD
    BCLR    PTGDD_PTGDD2, PTGDD
EndlessLoop:
    ; load joystick values
    MOV     RamLoc, PTGD
    JMP     EndlessLoop

; (=ensure program end if endlessloop is
    missing)
EndLoop:    BRA     *

; catch any unexpected interrupts
dummy:          BGND
                BRA     dummy
```

4.3 Addressing Modes

- ***Immediate***: *1 Byte operand in instruction (LDA #$01)*
- ***Inherent***: *no operand required (e.g. NOP, INCA..)*
- ***Direct***: *onlu direct page, 1 address Byte*
- ***Extended***: *whole 64k area, 2 address Bytes*
- ***Indexed***: *with SP (Stack pointer) or HX (7 sub modes)*
- ***Relative***: *for branches, PC=PC+2+two's compl.*

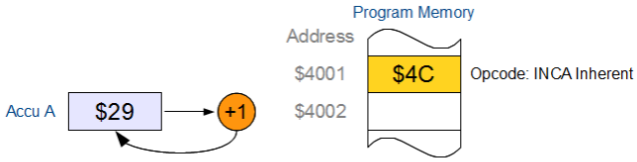
Different addressing modes of the same instruction type use differnet operation codes (e.g. LDA-MM: A6; LDA-DIR: B6).

4.3.1 Immediate (IMM)



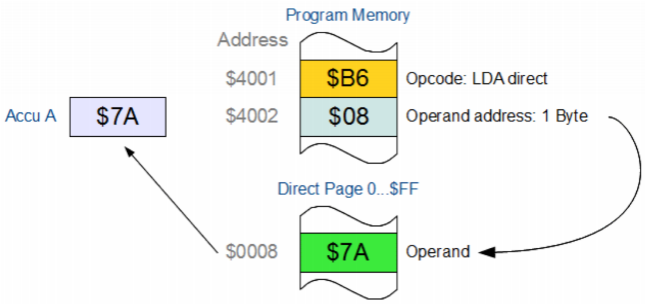
Immediat adressing *mode: the following Byte of the operation code is immediately used as the operand.*
Example: ***LDA #$1C***

4.3.2 Inherent (INH)



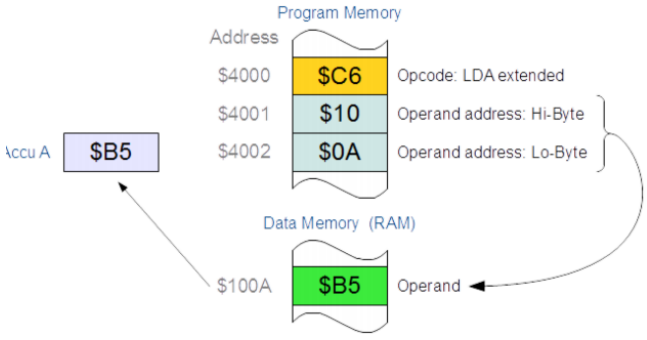
Inherent addressing *mode: no explicit operand address needed. All operands are in the CPU-registers*
Example: ***INCA***

4.3.3 Direct (DIR)



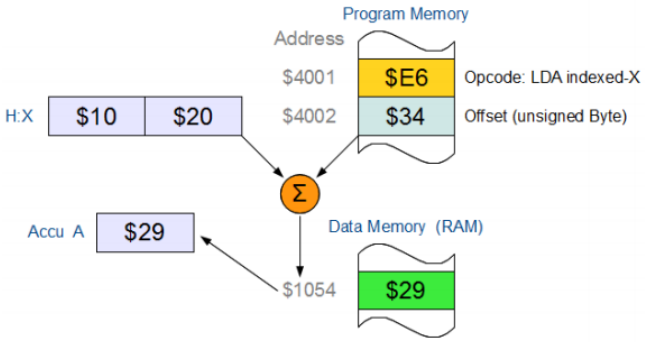
Direct addressing *mode: After the operation code, the* ***1-Byte*** *operand address follows in the program memory. Only operands in the address section between $00 and $FF are supported. (The Direct Page Registers 0x00-0xAF, Direct Page RAM 0xB0-0xFF)*
Example: ***LDA $08***

4.3.4 Extended (EXT)



Extended addressing *mode: After the operation code, the* ***2-Byte*** *operand address follows in the program memory.*
Supports the whole address section between 0x0000 - 0xFFFF. But is also slower.
Example: ***LDA $34,X***

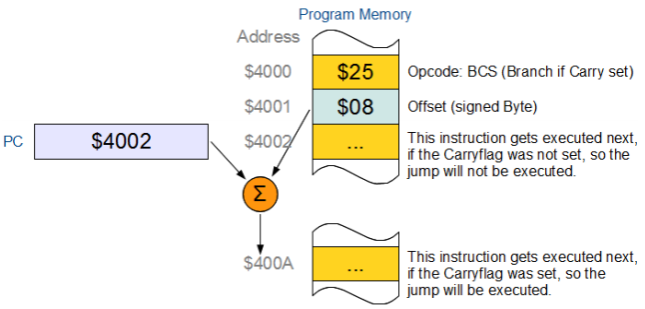
4.3.5 Indexed (IX1)



***Indexed addressing** mode: uses the **HX** or **SP** register. Through indexed addressing the final assigned operand address is dependent from the program behaviour (address arithmetics).*
Following are sub modes of the indexed addressing mode

IX	*Indexed addressing with H:X, without offset*	**LDA X**
IX1	*Indexed addressing with H:X and **8-bit offset***	**LDA $34, X**
IX2	*Indexed addressing with H:X and **16-bit offset***	**LDA $34A5, X**
IX+	*Indexed addressing with **H:X** and **H:X Increment**. Only for MOV and CBEQ (Compare Accu with value on the address that is stored in the H:X register. If values are equal, jump to Label and increment H:X) instructions*	**CBEQ X+, Label**
IX1+	*Same as IX+, with **Increment** and **8-bit offset** (Only available for instruction CBEQ)*	**CBEQ $34,X+, Label**
SP1	*Same as IX1, but with Stackpointer SP instead of H:X.*	**LDA $34, SP**
SP2	*Same as IX2, but with Stackpointer SP instead of H:X.*	**LDA $34A5, SP**

4.3.6 Relative (REL)



***PC relative addressing** mode: is only used with BRANCH-Instructions.*
*The following Byte after the operand is a **two's complement** offset to the already increased program counter.*
The address range with relaive addressing is -126 to +129. 129, since the PC is incremented before and after the jump (+2).

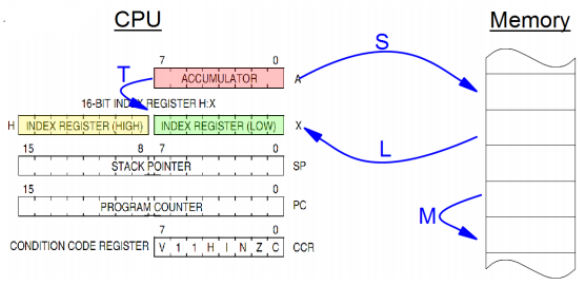
5 Assembler Addressing & Programming

5.1 Assembler Instructions

There are 3 main type of instructions:

- Data **Transport**
- **Operations** (Arithmetic, Logic, Bit-manipulation, Shift and Rotation)
- Program **Branches** with jump and branch operations

5.2 Transport Operations



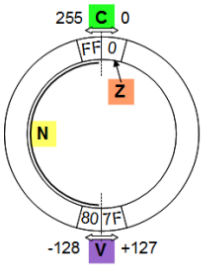
	Operation	Example
L	Load	*LDA, LDX, LDHX; PULA, PULX (Stackoperations)*
S	Store	*STA, STX, STHX; PSHA, PSHZ (Stackoperations)*
T	Transfer	*TAP, (CCR = Accu.), TPA, TAX, TSX*
M	Move	*MOV*

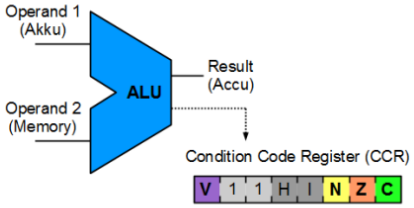
5.3 Arithmetic Operations

ADD	*Adds given operand to the ACC.*
SUB	*Works equivalent to the addition.*
ADC & SBC	*Include Carry bit and support additions and subtractions with numbers with more then 8 bits.*
MUL	*Multiplies the content of the accumulator A with the content of the index register X and stores the 16-bit result in X:A (MSB in X, LSB in A)* ***only unsigned**.*
DIV	*divides the 16-bit dividend in H:A (MSB in H, LSB in A) with the divisor in the index register X. The 8-bit result is written to A. If an overflow or division by 0 occurs, the Carry-bit is set.* ***only unsigned**.*

Results of arithmetic instructions are saved on the HCS08 eather in the X-Register or AKKU

5.4 Flags





CC	Name	Condition	Relevant for	
Z	Zero	Result = 0	unsigned	signed
N	Negative	Result < 0		signed
C	Carry	0 > Result > 255	unsigned	
V	Overflow	-128 > Result > 127		signed

Half-Carry *is used for binary-coded decimal calculations*

ADD instruction

C: A7&M7 | M7&$\overline{R7}$ | A7&$\overline{R7}$
V: $\overline{A7}$&$\overline{M7}$&R7 | A7&M7&$\overline{R7}$
N: R7
Z: $\overline{R7}$&$\overline{R6}$&$\overline{R5}$&$\overline{R4}$&$\overline{R3}$&$\overline{R2}$&$\overline{R1}$&$\overline{R0}$

SUB instruction

C: $\overline{A7}$&M7 | M7&R7 | $\overline{A7}$&R7
V: A7&$\overline{M7}$&$\overline{R7}$ | $\overline{A7}$&M7&R7
N: R7
Z: $\overline{R7}$&$\overline{R6}$&$\overline{R5}$&$\overline{R4}$&$\overline{R3}$&$\overline{R2}$&$\overline{R1}$&$\overline{R0}$

A (Operand 1)
M (Operand 2)
R (Result 1)

5.5 Logical Operations & Bit Masking

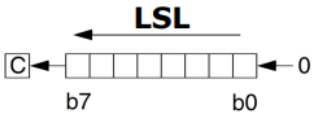
```
B7: EQU $80 ; Mask for Bit 7
B6: EQU $40 ; Mask for Bit 6
 :     :           :
B0: EQU $01 ; Mask for Bit 0

ORA #(B6 | B3) ; Set Bit 6 and 3 in ACCU
AND #(B5 | B4) ; Delete all Bits in ACCU except
     Bit 5 and 4
```

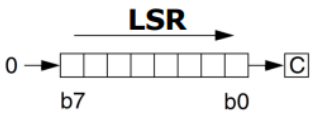
AND	*logical AND-operation*
ORA	*logical OR-operation*
EOR	*logical XOR-operation*
BCLR n,Addr	*Delete Bit n on a specific memory address*
BSET n,Addr	*Set Bit n on a specific memory address*
BIT Addr	*Bitwise AND operation of Accu with content of Addr, without changing content of Accu and Addr. Affects only **N- and Z-Flags**.*
CLC	*Delete **Carry**-Flag C*
SEC	*Set Carry-Flag C*
CLI	*Delete **Interrupt**-Mask Bit I (Interrupt **enable**)*
SEI	*Set Interrupt-Mask Bit I (Interrupt disable)*

5.6 Shift- and Rotation Operations

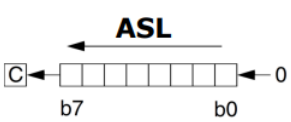
in direction MSB (left) **in direction LSB (rigth)**

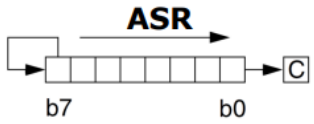
Logical Operations:



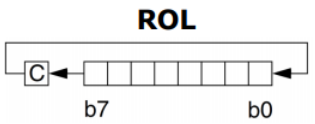


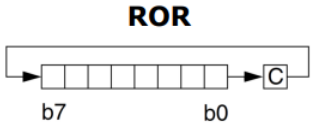
Arithmetic Operations:





Multiplication by 2, ASL=LSL *Division by 2*





5.7 Relative Branching

Unconditional Branch

Oper.	Meaning
BRA	Branch **always**
BRN	Branch never
BSR	Branch to **subroutine**

Testing a Single Flag

Oper.	Test	Meaning
BEQ	Z=1	Branch if **equal**
BNE	Z=0	Branch if not equal
BCS	C=1	Branch if **Carry** set
BCC	C=0	Branch if Carry clear
BMI	N=1	Branch if **Minus**
BPL	N=0	Branch if Plus

Arithmetic Comparison of Accu and Memory Location

Oper.	Test	Format
BGT	>	signed
BHI		unsigned
BGE	≥	signed
BHS, BCC		unsigned
BLE	≤	signed
BLS		unsigned
BLT	<	signed
BLO, BCS		unsigned
BEQ	=	signed
		unsigned

5.8 Branching Compare-Operation

*Compare instructions are **subtraction operations** that change status flags, but leave the data registers unchanged.*

CMP opr8	*Compare content of **ACCU** with 8-bit operand*
CPX opr8	*Compare content of **X-Register** with 8-bit operand*
CMP opr8	*Compare content of **HX-Register** with 16-bit operand*

Example, Test if a value is bigger or smaller than another value, branch afterwards

```
LDA Op1
CMP Op2   ; Calculates (Op1-Op2) and sets flags
BMI Label ; Branch if Op2 > Op1 (N=1) to Label
```

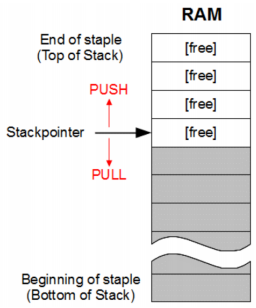
5.9 Direct relative Branching

*Those Branches are dependent on a single Bit of a memory located in the **Direct Page**.*

```
BRCLR n,Addr,Label ; Branches to Label, if Bit
    n of value on
                   ;address Addr is not set (
                       Addr only DIR)
BRSET n,Addr,Label ; Branches to Label, if Bit
    n of value on
                   ;address Addr is set (Addr
                       only DIR)
```

6 Subroutines & Stack

6.1 Stack



*The stack is a special memory section (in RAM) that works after the Last-In-First-Out (**LIFO**) principle.*
*It is addressed over the Stackpointerregister **SP** of the CPU.*
***PUSH** put and increment SP*
***PULL** get and decrement SP*

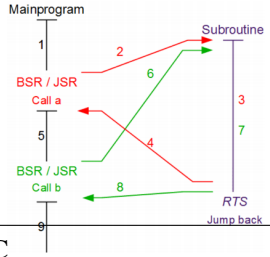
Stack grows from high addresses to lower

```
Stacksize: EQU $40
             :
DATA:      SECTION
TofStack:  DS Stacksize-1 ; reserve stack
BofStack:  DS 1
             :
PROGRAM:   SECTION
           LDHX #(BofStack+1) ; H:X := Bottom
               of Stack
           TXS                ; SP := HX -1
             :
           ; save CPU-Status on stack
           PSHA ; Akku auf Stack
           PSHX ; X-Register auf Stack
             :
           ; restore CPU-Status from stack
           ; order is imporant (LIFO!)
           PULX ; X-Register
           PULA ; Akku
```

Stacks are used for:
- *Subroutine calls (save return address)*
- *Store context*
- *Store parameters*
- *Store local variables*

malloc (heap) and global variables are not stored on the stack.

6.2 Subroutines



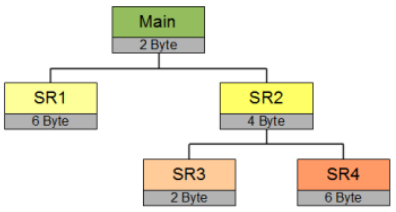
***BSR/JSR** push and inc. PC*
***RTS** pull and inc. PC*
***Parameters** passing on stack (used by C)*
***Local Variables** saved on stack (used by C)*

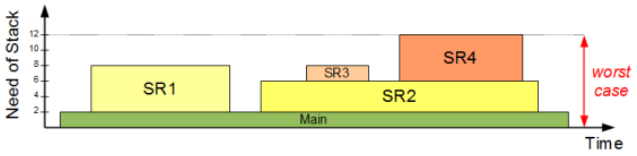
subroutines enable following:
- **less memory usage**; repeated command sequences are stored only once
- **less development effort**; tested command sequences can be reused
- **less error prone**; enable modular way of building software
- **higher team productivity**; multiple people can work parallel on different code sections
- **shorter compile time & libraries**; different parts of the code can be compiled seperatly

*The only **negative** about subroutines is calling of subroutines is **slower**. Time is needed for passing parameters and saving the context on the stack*

6.3 Stack size

To analyze the used stack size, it is helpful to create a tree with the subroutines, their calls and used stack space.

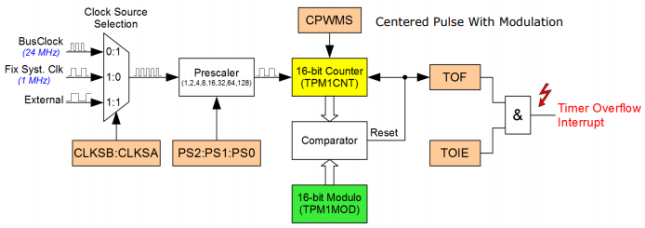




It is also possible to figure out the stack usage by filling the program-stack at the start with an bit pattern like 0xdeadbeef and stress test the program as much as possible. At the end, this will show which part and how much of the stack has been used during the program execution.

7 Timer and Interrupts

7.1 Modulo Counter



0x0060	TPM2SC	TOF	TOIE	CPWMS	CLKSB	CLKSA	PS2	PS1	PS0
0x0061	TPM2CNTH	Bit 15	14	13	12	11	10	9	Bit 8
0x0062	TPM2CNTL	Bit 7	6	5	4	3	2	1	Bit 0
0x0063	TPM2MODH	Bit 15	14	13	12	11	10	9	Bit 8
0x0064	TPM2MODL	Bit 7	6	5	4	3	2	1	Bit 0

7.2 Modulo Frequency

$$\mathbf{T_{TOF}} = (\mathbf{MOD} + 1) \cdot \mathbf{PS}/\mathbf{f_{Clk}}$$

- $\mathbf{T_{TOF}}$: Time between two Timer-Overflow events
- $\mathbf{MOD}$: Value of the Modulo set
- $\mathbf{PS}$: Presacler value
- $\mathbf{f_{Clk}}$: frequency of the controller

To calculate the modulo, the frequency (Clock Source) needs to be selected and the prescaler needs to be defined. To calculate the Modulo value, following can be used. The Modulo is 2 Bytes, so it needs to be between ***0 < MOD < 65536***

$$\mathbf{MOD} = (\frac{\mathbf{T_{TOF}} \cdot \mathbf{f_{Clk}}}{\mathbf{PS}}) - \mathbf{1}$$

7.3 Timer Control Registers

Address	Reg-Name	Bit-Name							
0x0060	TPM2SC	TOF	TOIE	CPWMS	CLKSB	CLKSA	PS2	PS1	PS0
0x0061	TPM2CNTH	Bit 15	14	13	12	11	10	9	Bit 8
0x0062	TPM2CNTL	Bit 7	6	5	4	3	2	1	Bit 0
0x0063	TPM2MODH	Bit 15	14	13	12	11	10	9	Bit 8
0x0064	TPM2MODL	Bit 7	6	5	4	3	2	1	Bit 0
0x0065	TPM2C0SC	CH0F	CH0IE	MS0B	MS0A	ELS0B	ELS0A	0	0
0x0066	TPM2C0VH	Bit 15	14	13	12	11	10	9	Bit 8
0x0067	TPM2C0VL	Bit 7	6	5	4	3	2	1	Bit 0
0x0068	TPM2C1SC	CH0F	CH0IE	MS0B	MS0A	ELS0B	ELS0A	0	0
0x0069	TPM2C1VH	Bit 15	14	13	12	11	10	9	Bit 8
0x006A	TPM2C1CL	Bit 7	6	5	4	3	2	1	Bit 0

SC = Status&Control, CNT = Counter, MOD = Modulo, V = Value; H = High-Byte, L = Low-Byte

CLKSB:CLKSA	TPM Clock Source to Prescaler Input
00	No clock selected (TPM counter disable)
01	Bus rate clock
10	Fixed system clock
11	External source

Table 16-4. Prescale Factor Selection

PS2:PS1:PS0	TPM Clock Source Divided-by
000	1
001	2
010	4
011	8
100	16
101	32
110	64
111	128

7.4 Polling and Interrupts

A MC-System has to react instantly to events (internal or external) (e.g. measure value monitoring, serial communication).
The ***instant of time*** *of these events is* ***not known in advance****.*

There are two ways to react to those kind of events:

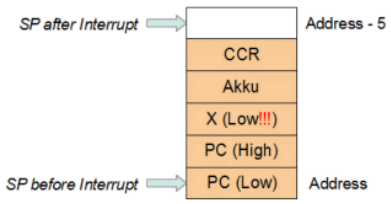
- **Interrupt** = Exception handling
  enables ***realtime capable (+)*** *systems (depends on interrupt* ***latency****). **Fast reaction time** through automatic reaction to events and interrupt of the program to execute an Interrupt-Service-Routine (ISR). Needs substancial effort for* ***state backup (-)****, because the instant of the program interruption is unknown.*
- **Polling** = cyclic requesting
  Shorter *program* ***interruption (+)****. Since the instant of time is known during programming, the state can be backed up more efficiently.*
  esier to understand / debug (+)
  Waste of caclulation time (-) *if events occure rarely*

Each MCU holds an Interrupt-Logic to realise real-time systems.

7.5 Interrupt execution

1. Interrupt called
2. Save current state onto stack
3. Call function
4. By Programming - clear interrupt flag
5. go back to code
6. load saved state from stack
7. keep running where stop before interrupt

7.6 Save Interrupt State



On entrance to an ISR the CPU-State is backed up automatically to the Stack.

Note: The ***H-Register*** *must be saved „manually“ on the HCS08 (only with Assembler)*

7.7 Difference ISR and Subroutines

ISR = Interrupt Service Routine / Interrupt Subroutine

	ISR	Unterprogramm
Call	spontaneous	BSR/JSR
State backup	automatic	Program (manual)
Return jump	RTI	RTS

7.8 Interrupt Sources Priority

In the MC9S08JM60 there are 29 Interrupt Sources, that are sorted by priority in the Interruptvector-Table

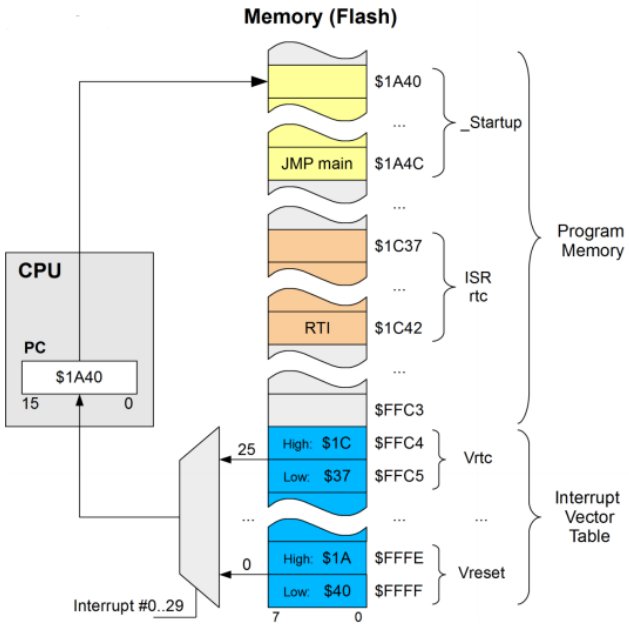
1x	Real-Time Clock		lowest priority
1x	IIC Module		
1x	Comparator Module		
1x	A/D-Converter		
1x	Keyboard-Interface		
6x	SCI 1/2 Module		
10x	Timersystem	maskable	
1x	USB Module		
2x	SPI 1/2 Module		
1x	Clock Generator		
1x	Low-Voltage Detection		
1x	External IRQ Pin		
1x	SW-Interrupt (SWI)	not maskable	
1x	Reset Interrupt	partwise maskable	highest priority

*By default the HCS08 does **not support nested Interrupts**, because the I-Flag gets set on an entrance into an ISR.*
If there are more Interrupt demands, the ISR with the highest priority (lowest vector number) is called first

7.9 Interrupt Counter

Setting the Interrupt Counter will set it always to 0.
Reading one of the Counter 8 Bit, the other one will be saved to a shadow register until read from.

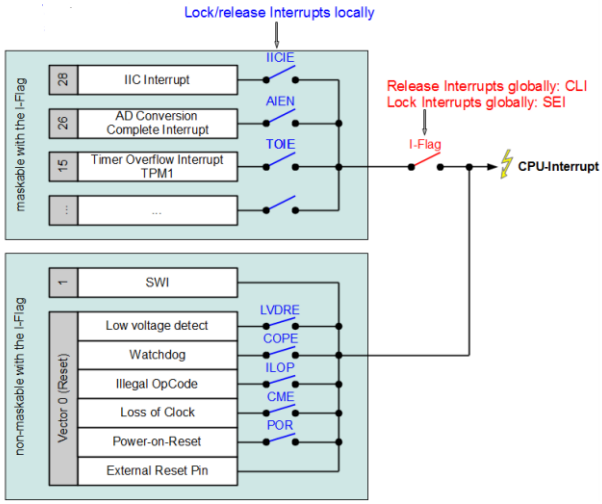
7.10 Interrupt Vectortable




```
// Extract out of .prm File
VECTOR ADDRESS 0xFFC4 ISR_RTI // RTC
VECTOR ADDRESS 0xFFC6 errISR_IIC // IIC
VECTOR ADDRESS 0xFFC8 errISR_ACMP // ACMP
VECTOR ADDRESS 0xFFCA errISR_ADC // ADC
    Conversion
...
VECTOR ADDRESS 0xFFDA motorBoosterISR // TPM2
    overflow
VECTOR ADDRESS 0xFFDC errISR_TPM2CH1 // TPM2
    channel 1
VECTOR ADDRESS 0xFFDE errISR_TPM2CH0 // TPM2
    channel 0
```

```
VECTOR ADDRESS 0xFFE0 errISR_TPM2O // TPM1
    overflow
VECTOR ADDRESS 0xFFE2 errISR_TPM1CH5 // TPM1
    channel 5
VECTOR ADDRESS 0xFFE4 errISR_TPM1CH4 // TPM1
    channel 4
VECTOR ADDRESS 0xFFE6 errISR_TPM1CH3 // TPM1
    channel 3
VECTOR ADDRESS 0xFFE8 errISR_TPM1CH2 // TPM1
    channel 2
VECTOR ADDRESS 0xFFEA errISR_TPM1CH1 // TPM1
    channel 1
VECTOR ADDRESS 0xFFEC ifrFrontISR // TPM1
    channel 0
```

7.11 Interrupt-Release Logic



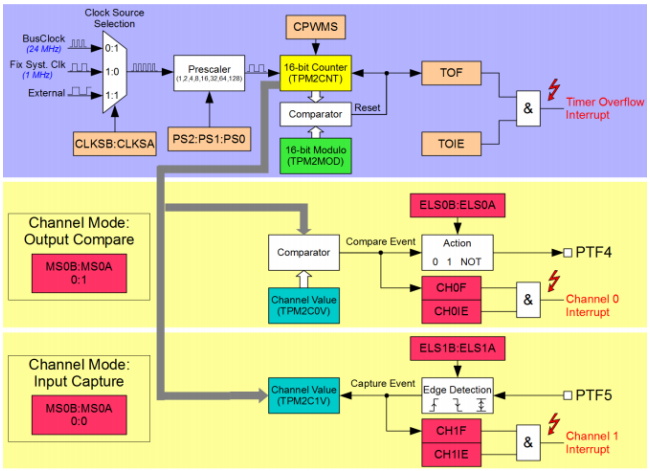
7.12 Programming of Interrupts

Following is important for programming interrupts:

- *Define **Interruptvectors**; at the place of the Interruptvector has to be the start address of the ISR (in CW definition in .prm file)*
- *Define and initialise **Stack***
- ***Delete** the Interrupt-**Flags before** you release them, so that the Interrupt does not get fired right away.*
- *Programming of ISR; CPU-State gets backed up automatically (H-Register only through C-Compiler)*
- ***Delete** the Interrupt-**Flag** in the ISR*
- *End the ISR with **RTI** (is done automatically on usage of C-Compiler)*
- Release Interrupts globally (**CLI**) in the main program (typically after initialisation part)

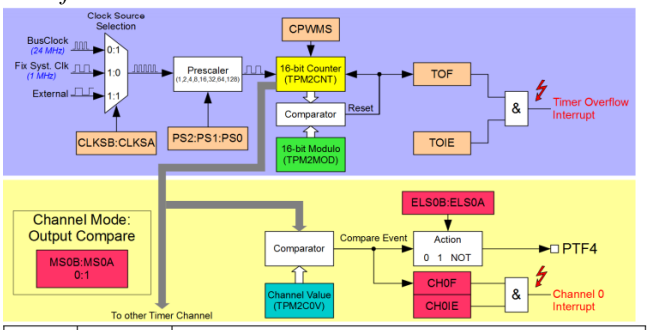
```
interrupt void myTofISR(void)
{
    // myTofISR function needs to be mapped
    // in the vectortable -> parameterfile (.
        prm).
    //reset the interrupt flag
    TPM1SC_TOF = 0;

    //run logic
}
```

```
void initTimer(void)
{
    //set module to 25780 / 0x64B4
    TPM1MODH = 0x64;
    TPM1MODL = 0xB4;
    //TPM1MOD = 25780;
    //Clock set to 1 MHz
    TPM1SC_CLKSA = 0;
    TPM1SC_CLKSB = 1;

    //define Prescaler to 128
    TPM1SC_PS0 = 1;
    TPM1SC_PS1 = 1;
    TPM1SC_PS2 = 1;

    // reset counter
    TPM1CNT = 0

    // enable timer Overflow Interrupt
    // this should be the last action
    TPM1SC_TOIE = 1;
    // Reset the Timer Overflow Interrupt
    TPM1SC_TOF = 0;
}
void main(void)
{
    initTimer();
    //enable interrupts
    EnableInterrupts;
}
```



8.1 Timer with Output-Compare

interrupt is occuring, when the content of the V-Register and of the counters have the same value.



Address	Reg-Name	Bit-Name							
0x0065	TPM2C0SC	CH0F	CH0IE	MS0B	MS0A	ELS0B	ELS0A	0	0
0x0066	TPM2C0VH	Bit 15	14	13	12	11	10	9	Bit 8
0x0067	TPM2C0VL	Bit 7	6	5	4	3	2	1	Bit 0

8 Output Compare & Input Capture

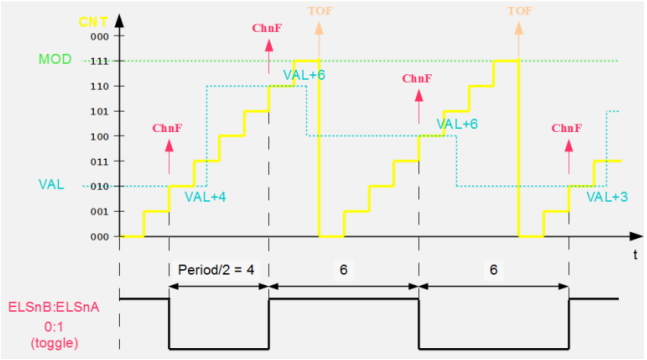
CPWMS	MSnB:MSnA	ELSnB:ELSnA	Mode	Configuration
X	XX	00	Pin not used for TPM - revert to general purpose I/O or other peripheral control	
0	00	01	Input capture	Capture on rising edge only
		10		Capture on falling edge only
		11		Capture on rising or falling edge
	01	01	Output compare	Toggle output on compare
		10		Clear output on compare
		11		Set output on compare
	1X	10	Edge-aligned PWM	High-true pulses (clear output on compare)
		X1		Low-true pulses (set output on compare)
1	XX	10	Center-aligned PWM	High-true pulses (clear output on compare-up)
		X1		Low-true pulses (set output on compare-up)

```
void initTimer(void){
    TPM1C1SC_CH1IE = 1; //Channel 1 Timer 1
        Interrupt enable
    TPM1C1SC_MS1A = 1; //A=1 ; B=0 - Output
        Compare
    TPM1C1SC_MS1B = 0;
    TPM1C1SC_ELS1A = 1; //A=1 ; B=0 - Toggle
        Output on Compare
    TPM1C1SC_ELS1B = 0;
    TPM1C1V = 0x95FF; //set 16bit channel value
    // (is compared with main timer, calc the
        timer on the base of the clock)
}
interrupt void ISR_outCompare(void){
    TPM1C1SC_CH1F = 0 ; //Timer 1 Channel 1
        overflowflag reset
    TPM1C1V += 0x95FF ; //Channel Value is set
        to new value
    // (add with the value, on how much time
        needs to pass)
}
```

8.2 Usage Output Compare Mode

Output Compare is usually used to set / clear or toggle output pins.
The output compare mode can additionally be used to setup different timers on base of the same timer wihout

changing the TPMxMOD value. To use the TPMxCx output pin for other purposes, following needs to be set ELSxA:ELSxB=00



The value to increment can be calculated as following:

$$ChannelValue = \frac{T_{CHnF} \cdot F}{Prescaler}$$

T_{CHnF}: *Target channel frequency*
F: *Clock frequency*

8.3 Input Capture

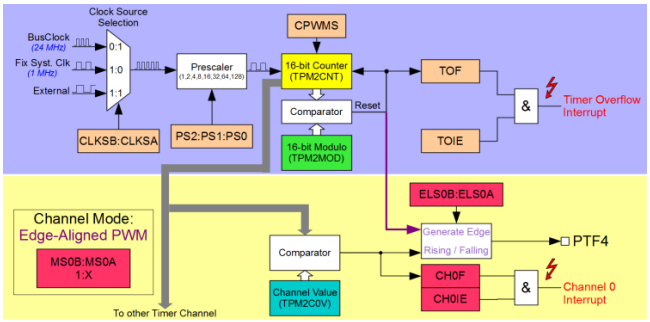
Input capture is used on the timer pins. it enables reacting on input (rising/falling or both).
If an input capture happens, the interrupt is executed and the current counter is saved in the channels value register for further usage.

8.4 Logic Analyzer

The old logic analyzer has modes for external clocks, but it is limited to memory

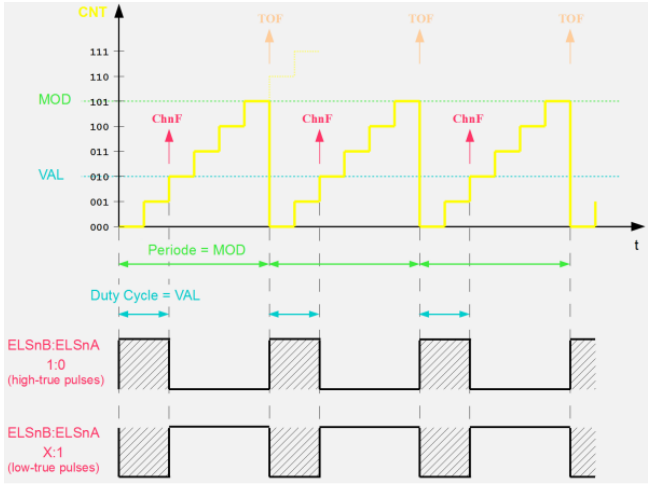
New Logic Analyzer will always use the internal clock and since it uses the computer, the memory is fine and oversampiling is fine.

9 Pulse-Width Modulation (PWM)



Address	Reg-Name	Bit-Name							
0x0065	TPM2C0SC	CH0F	CH0IE	MS0B	MS0A	ELS0B	ELS0A	0	0
0x0066	TPM2C0VH	Bit 15	14	13	12	11	10	9	Bit 8
0x0067	TPM2C0VL	Bit 7	6	5	4	3	2	1	Bit 0

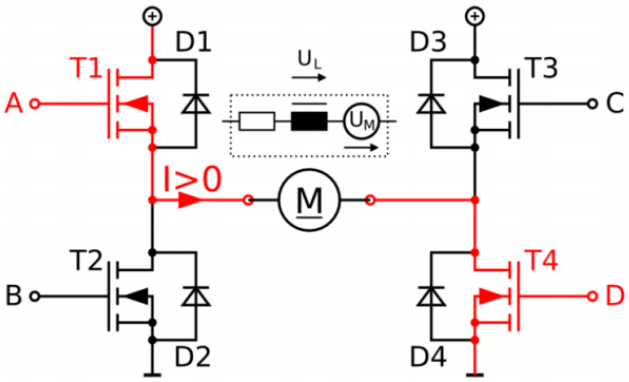
9.1 Edge Aligned PWM



VAL = 0: Duty Cycle = 0%
VAL > MOD: Duty Cycle = 100%

- ***High-true pulses**: Das beduetet, dass true = 1 ist. Also solange der Channel Value > Counter ist, ist der ausgehende Pin = 1.*
- ***Low-true pulses**: Das beduetet, dass true = 0 ist. Also solange der Channel Value > Counter ist, ist der ausgehende Pin = 0;*

9.2 H-Bridge (Fast / Slow Decay)



Energy in the Magnetic field
***Fast Decay**: "Brake" after T1 & T4 on, deletion with D2 & D3 (2 x 0.7V Voltage drop → Power loss → eventually puls with T2 & T3)*
***Slow Decay**: "Idle Running" after T1 & T4 on, delete with T2 & T4*

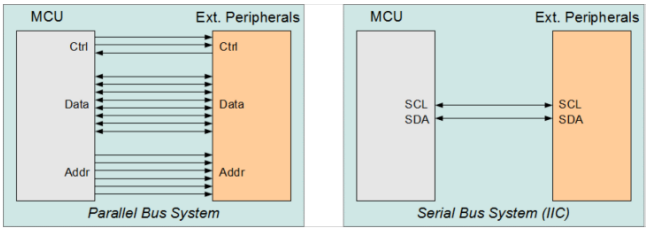
9.3 PWM Code

```
void initTimer(void) {
    //set clock to 24 MHz
    TPM1SC_CLKSA = 1;
    TPM1SC_CLKSB = 0;
    //Prescaler to 0
    TPM1SC_PS0 = 0;
    TPM1SC_PS1 = 0;
    TPM1SC_PS2 = 0;
    // set Modulo to 2^16 -1 stellen.
    // (-1, to enable PWM to reach 100%)
    TPM1MOD = 65534;
```

```
}

void initPWM(void) {
    //Channel 2 to Mode Edge-Aligned PWM
    TPM1C2SC_MS2A = 1;
    TPM1C2SC_MS2B = 1;
    //set Channel 2 to "Low-true pulses", since
         LED rect to 0 as on
    TPM1C2SC_ELS2A = 1;
    TPM1C2SC_ELS2B = 1;
    //Channel 2 Channel Value = Modulo * 0.3,
        since set LED to 30%.
    TPM1C2V = 21844;
}

void main(void)
{
    // enable as output
    PTFDD_PTFDD0 = 1;

    initTimer();
    initPWM();

    for(;;);
}
```

10 IIC-Bus

10.1 IIC-Bus Properties



- *2 bidirectional wires (Clock: **SCL**, Data: **SDA**)*
- *Clock rates: **Standard 100 kHz; Fast 400 kHz**; Fast Plus 1 MHz; High Speed 3,4 MHz*
- *Master/Slave-architecture, multiple masters are possible (**Bus Arbiter**)*
- *Number of participants is limited by number of addresses and wire capacity*
- *Bus participants of different speeds are possible (**Clock Stretching**). Clock Stretching enables the slave to slow down the master.*

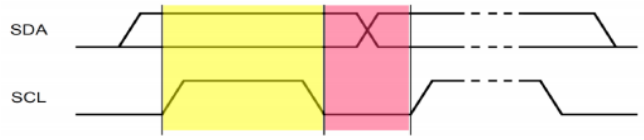
10.2 IIC-Bus stages

- *All bus participants use **Open-Drain Output stages** (no active H-Level possible).*
- ***External Pullup**-Resistors generate the H-Level (Default-State).*
- *All bus participants observe at all times the actual state of SCL (Clock) and SDA (Data).*

IIC without hardware support:

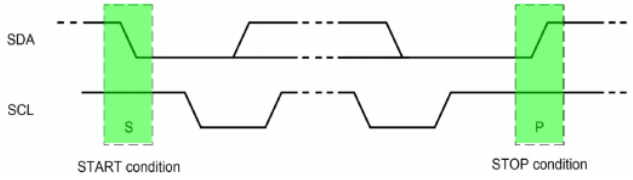
If the IIC is not supported by the hardware, the open drain can be simulated. This is possible by setting the pins Internal pullup to 1 (PTxPEn = 1). Set the Output to 0 (PTxDn = 0). To write 1 (PTxDDn = 0) and 0 (PTxDDn = 1) use the datadirection. This will prevent any short circuits.

10.3 Bit-Transfer



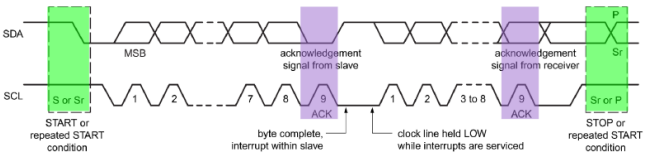
SDA (Data) can be changed if SCL=0, and is evaluated when SCL=1.

10.4 Start-/Stop-Condition



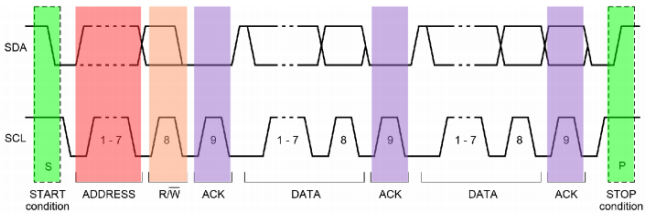
***Start-/Stop**-Conditions are always generated by the Master. As **protocol mismatch** they are detected by other Masters and Slaves and can easily be differenciated from normal data bits.*
***After** a **Start**-Condition, the bus is **busy**.*
***After** a **Stop**-Condition, the bus is **idle**.*

10.5 Byte Transfer (Blocks)



- *Data transmission from transmitter to receiver is **Byte-wise** (**MSB-first**).*
- *At the end of each byte, the **Receiver** generates an Acknowledge-Bit:* **SDA = 0 = Ack** **SDA = 1 = No-Ack** *Through the generation of No-Ack, a transfer can be cancelled (premature).*
- *A **Repeated-Start** (Sr) Condition can be generated by the active Master instead of a Stop-Condition, if the Master wants to continually use the bus*
- *A Slave can force a Master to slow transmission through **Clock Stretching**.*

10.6 Slave Addressing

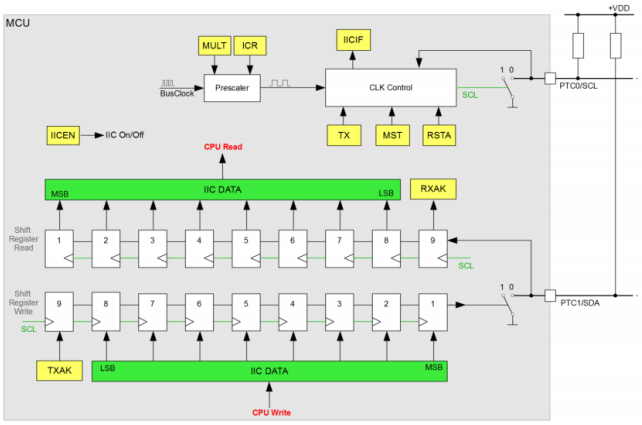


- *In the first byte after the Start-Condition the master sends a **7-Bit Address**.*
- *A slave with this address has to answer in the 9th bit with an Ack-Signal.*
- *In the **8th bit** the master sends the **R/W** direction-bit:*

R/W = 0 *: Write : Master-Transmitter to Slave-Receiver*
R/W = 1 *: Read : Master-Receiver from Slave-Transmitter*

- *Combined R/W Transfer-formats are possible through Repeated-Start Condition.*

10.7 Function Schema & Control Register



IICEN: Block enable
IICIE: Interrupt en.
MST: master & busy
Tx: transmit
TXAK: ack enable
RSTA: repeat start
$f_{IIC} = f_{BUS}/MULTxf(ICR)$
TCF: transfer done
IAAS: addressed
BUSY: bus busy
ARBL: arbitration lost
SRW: slave R/W
IICIF: int. Flag
RXAK: acknowledged

0x0058	IICA	AD7	AD6	AD5	AD4	AD3	AD2	AD1	0	slave address
0x0059	IICF	MULT		ICR (Table p. 167)						freq. divider
0x005A	IICC	IICEN	IICIE	MST	TX	TXAK	RSTA	0	0	control register
0x005B	IICS	TCF	IAAS	BUSY	ARBL	0	SRW	IICIF	RXAK	status register
0x005C	IICD	DATA								data reg. (r/w)

- *IICD: 7 Bit Daten, 1 Bit R/W*
- *IICS_IICIF: IIC Interrupt Flag wird gesetzt wenn: 1Byte transferiert wurde, Slave Adresse und angesprochene Adresse identisch sind oder Arbitration lost.*
- *IICS_RXAK: 0=Acknowlage recieved, 1=no Achnowlage recieved.*
- *IICC_MST: wechsel von 0 nach 1 generiert Stop.*
- *IICS_IICIF: =1 resetet den Interrupt.*
- *IICC_TSAK: =0 ein ACK wird nach empfangen eines Bytes gesendet, =1 kein ACK wird gesendet.*

10.8 Baud rate

The IIC baud rate can be calculated as following (standard: 100 kbit/s = 100'000)
$baudrate = \frac{f_{bus}[Hz]}{mul \cdot Divider}$

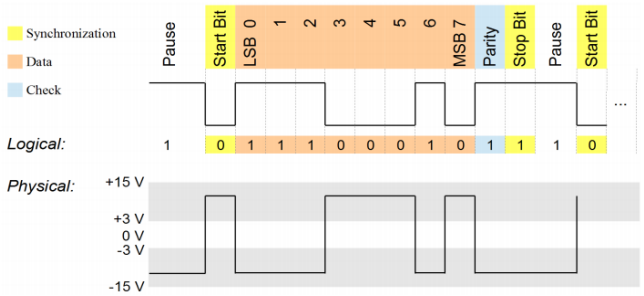
10.9 Code I2C Module

```
void main(void)
{
    uint8 i;
    TPM1SC = 0x10; // Timer init -> 1MHz
    ifrRxFrontInit(); // Infrared init
    motorInit(); // Motor init
    i2cInit(); // init i2c
    EnableInterrupts; // Interrupts enable
}

// Initialisiert den I2C-Bus
// -> enable I2C with 400 kHz SCL clock
    frequency
void i2cInit()
{
    // Frequency Divider Register: zur
        Einstellung der Baudrate
    IICF_ICR = 0x05; // 24 MHz/(2 * 30) = 400
        kHz
    IICF_MULT = 0x01; // IIC Baudrate =
        BusSpeed (Hz)/((MULT * SCLdivider)
    // SCLdivider -> Tabelle S.167
    // IIC Control Register
    IICC_IICEN = 1; // I2C enable
}

//Start
tError i2cStart(uint8 adr, bool write)
{
    while (IICS_BUSY); // Warte bis Bus frei
        ist. Notwendig falls 2x Sende-Befehle
        kurz nacheinander folgen
    IICS_IICIF = 1; // Interrupt Bit quittieren
         falls gesetzt
    IICC_TXAK = 0; // TXAK (ACK senden)
        deaktivieren falls
    aktiviert
    IICC |= 0x30; // MST=1, TX=1; =>
        StartCondition senden...
    if (write) IICD = (adr & 0xFE);// Adresse
        senden - Low aktives Write-Bit
    else IICD = adr | 0x01;

    while (!IICS_IICIF); // wait till sent
    IICS_IICIF = 1; // clear Interrupt-Flag
    if (IICS_RXAK) // check ACK received
    {
        IICC_MST = 0; // Stop-Condition
            generieren
        IICS_IICIF = 1; // clear Interrupt-Flag
        return EC_I2C_NO_ANSWER;// NACK =>
            Abbruch
    }
    return EC_SUCCESS;
}
//Repeated Start
tError i2cRepeatedStart(uint8 adr, bool write)
{
    IICC_RSTA = 1; // output repeated Start-
        Condition
    if (write) IICD = (adr & 0xFE); // send
        Adresse - Low activities Write-Bit
    else IICD = adr | 0x01;
    while (!IICS_IICIF); // wait till sent
    IICS_IICIF = 1; // clear Interrupt-Flag
    if (IICS_RXAK) // check ACK received
    {
        IICC_MST = 0; // generate Stop-
            Condition
        IICS_IICIF = 1; // clear Interrupt-Flag
        return EC_I2C_NO_ANSWER; // NACK =>
            cancel
    }
    return EC_SUCCESS;
}
//Stop
void i2cStop()
{
```

```
    IICC_MST = 0; // generate Stop-Condition
    IICS_IICIF = 1; // clear Interrupt-Flag
}
//send Data
tError i2cSendData(uint8 *buf, uint8 length)
{
    uint8 i;
    for (i=0; i<length; i++)
    {
        IICD = buf[i]; // send databyte
        while (!IICS_IICIF); // wait till
            transmission finished
        IICS_IICIF = 1; // clear Interrupt-Flag
        if (IICS_RXAK) // check ACK received
        {
            IICC_MST = 0; // Stop-Condition
                generieren
            IICS_IICIF = 1; // clear Interrupt-
                Flag
            return EC_I2C_NAK; // NACK =>
                Abbruch
        }
    }
    return EC_SUCCESS;
}
//recieve Data
void i2cReceiveData(uint8 *buf, uint8 length)
{
    uint8 i;
    IICC_TX = 0; // set Receive-Mode
    if(length > 1)
    {
        IICC_TXAK = 1; // enable ack for
            reaveive
        buf[0] = IICD; // dummy read
        while (!IICS_IICIF); // wait till
            transmission finished
        IICS_IICIF = 1
        for(i=0; i<length-2; i++)
        {
            buf[i] = IICD;
            while (!IICS_IICIF);
            IICS_IICIF = 1;
        }
        IICC_TXAK = 1;
        // start last data transfer
        buf[length - 2] = IICD;
        while (!IICS_IICIF);

        // create stop Condition
        IICC_MST = 0;
        buf[length-1] = IICD;
    }
    else
    {
        IICC_TXAK = 1; // send no Ack that a
            Stop-Condition can be sent
        buf[0] = IICD; // Dummy-Read and
            therefor last transmission  starten
            ...
        while (!IICS_IICIF); // wait till
            transmission finished
        IICS_IICIF = 1; // clear Interrupt-Flag
        IICC_MST = 0; // generate Stop-
            Condition
        buf[0] = IICD; // read last databyte
    }
}
```

11 RS-232

11.1 Protocol



9600 8-O-1 9600 Baud, 8 Data Bits, Odd Parity, 1 Stop Bit

Data Bits 0100 0111 = 0x47 = 'G'

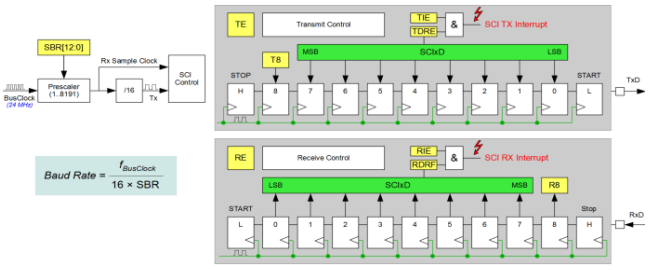
- *RS-232 works with an **asynchronous** point to point transmission with seperate **Rx** and **Tx** data wires (Receive/Transmit).*
- *Today more modern standards are used for the physical transmission of RS-232, e.g. USB or Bluetooth.*
- *In practice **most often the parity-bit** is **not** used, but instead a check is done on a higher protocol layer, e.g. **CRC**-check sum.*

11.2 Transmission-protocols comparision

Following protocols are used often with MCUs for wire-bound transmission. (SPI «Serial Peripheral Interface» from Motorola, similar to Micro-wire of NI)

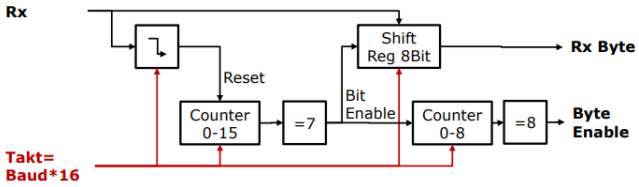
	RS-232	IIC	SPI	1-Wire
Duplex	full	half	full	half
Topology	point-to point	multi-master	master-slave	master-slave
Timing	asynchron	synchron	synchron	asynchron
Pin Count (w/out GND)	2	2	3 + #slaves (as ring: 4)	1
Typ. Data Rates [kbps]	10-100	100-3'400	1'000-15'000	16
Typical Application	off-board	on-board, address locations	on-board, streaming data	ultra low-cost/-power
Typical Devices	PC terminals	Sensors	ADC/DAC	Sensors
(other than MCU/DSP)		EEPROMS	Flash	

11.3 Function Schema & Control Register



0x0038	SCI1BDH	LBKDIE	RXEDGIE	0	SBR12	SBR11	SBR10	SBR9	SBR8	baud div H
0x0039	SCI1BDL	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0	baud div L
0x003A	SCI1C1	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT	control reg 1
0x003B	SCI1C2	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	control reg 2
0x003C	SCI1S1	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	status reg 1
0x003D	SCI1S2	LBKDIF	RXEDGIF	0	RXINV	RWUID	BRK13	LBKDE	RAF	status reg 3
0x003E	SCI1C3	R8	T8	TXDIR	TXINV	ORIE	NEIE	FEIE	PEIE	control reg 3
0x003F	SCIxD	Data								data r/w

11.4 Serial Bit-Synchronization

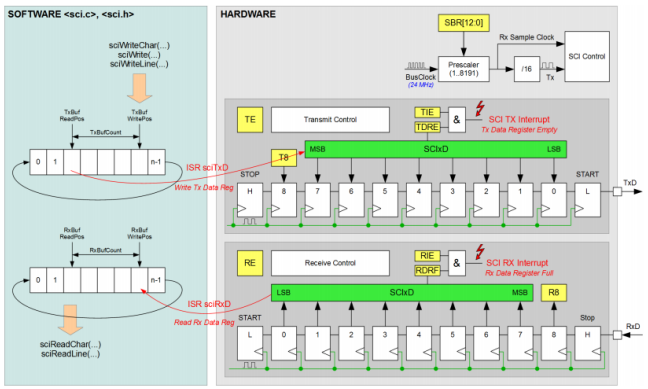


- *Bit-Enable is the recovered bit clock rougly in the middle of each bit.*
- *Serial transmitting fast fiber optical systems (eg. SDH) need a similar bit clock recovery (mostly done with Phase Locked Loops - PLL).*
- *Missing parts: Byte-Enable should stop Counter0-15 - Reset should start it; bit sample eg. at positions 5, 7, and 9 – majority decision, 8-bit parallel load byte register, Tx.*
- *noise bit is set if the 3 bits (5, 7 and 9) are not the same.*

11.5 Parity Bit

E: Even parity = even count num 1 => 0 uneven num 1 => 1
O: odd is other way around

11.6 Code RS232 with Ringbuffer



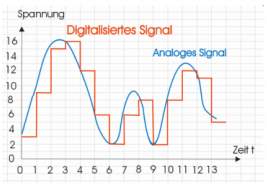

```
// sendqueue
static char tx1Buf[SCI1_TX_BUF_SIZE];
static uint8 tx1BufCount;
static uint8 tx1BufWritePos;
static uint8 tx1BufReadPos;

// receivequeue
static char rx1Buf[SCI1_RX_BUF_SIZE];
static uint8 rx1BufCount;
static uint8 rx1BufWritePos;
static uint8 rx1BufReadPos;

#define BUSCLOCK              24000000   // Hz
```

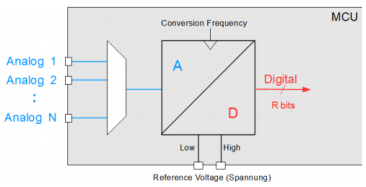
```
// init define baudrate; like 4800, 9600, 38400
void sci1Init(uint32 baudrate)
{
    // Berechnung Baudrate normalerweise:
        SCIxBD = Busclock / (16 x Baudrate)
    SCI1BD = (uint16) (((BUSCLOCK * (uint32)
        10) / ((uint32) 16 * baudrate) + 5) /
        10);

    SCI1C3 = 0x0F;              // activate
        error-Interrupts

    tx1BufCount = 0;            // TX-Buffer
        initialisieren
    tx1BufWritePos = 0;
    tx1BufReadPos = 0;

    SCI1C2_TE = 1;              // turn sender
        on

    rx1BufCount = 0;            // RX-Buffer
        initialisation
    rx1BufWritePos = 0;
    rx1BufReadPos = 0;

    SCI1C2_RE = 1;              // activate
        receiver;
    SCI1C2_RIE = 1;             // activate
        receiver interrupt
}

// error interrupt routine
interrupt void sci1Error(void)
{
    (void)SCI1S1;
    (void)SCI1D;
}

// receive data
interrupt void sc1RxD(void)
{
    char ch;
    (void)SCI1S1; // read state to reset
    ch = SCI1D;
    if(rx1BufCount < SCI1_RX_BUF_SIZE)
    {
        rx1Buf[rx1BufWritePos] = ch;
        rx1BufCount++;
        rx1BufWritePos++;
        if(rx1BufWritePos == SCI1_RX_BUF_SIZE)
            rx1BufWritePos = 0;
    }
}

// write next byte from ringbuffer
interrupt void sci1TxD()
{
    (void)SCI1S1;

    if(tx1BufCount != 0) {
        SCI1D = tx1Buf[tx1BufReadPos];
        tx1BufCount--;
        tx1BufReadPos++;
        if(tx1BufReadPos == SCI1_TX_BUF_SIZE)
            tx1BufReadPos = 0;
    } else {
        SCI1C2_TIE = 0;
    }
}

// write to the ringbuffer
char sci1ReadChar(void)
```

```
{
    char ch;
    while(rx1BufCount == 0);

    ch = rx1Buf[rx1BufReadPos];
    rx1BufCount--;
    rx1BufReadPos++;
    if (rx1BufReadPos == SCI1_RX_BUF_SIZE)
        rx1BufReadPos = 0;
    return ch;
}

// write a characater
void sci1WriteChar(char ch)
{
    while (tx1BufCount >= SCI1_TX_BUF_SIZE);

    tx1Buf[tx1BufWritePos] = ch;
    tx1BufCount++;
    tx1BufWritePos++;
    if (tx1BufWritePos == SCI1_TX_BUF_SIZE)
        tx1BufWritePos = 0;

    SCI1C2_TIE = 1;
}
```

12 Analog/Digital Converter

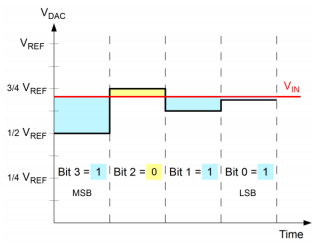
12.1 AD-Converter System





- *Many sensor measure values are provided as an analog signal (e.g. temperature, pressure, velocity, etc.). For further processing, the values are converted into **R bits (resolution)** wide digital signals.*
- *Many MCUs are equipped with an integrated **A/D-Converter**, that can convert multiple analog input voltages from the MCU-Pins with **timemultiplexing**.*

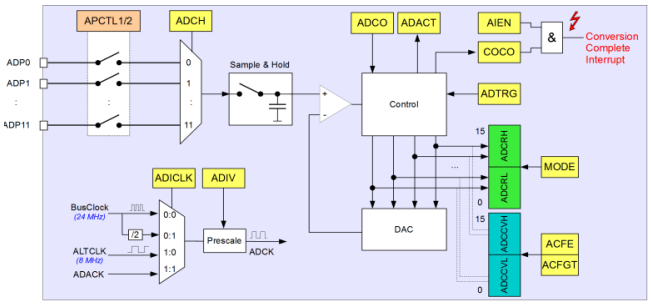
12.2 Successive Approximation



*The integrated **A/D-Converter** does comparisions step by step, beginning with the MSB. Simply its a binary search through comparing. The more Bits the generated digital word has, the closer the proximity will be.*

*During the conversion, the input voltage is kept steady (**Sample & Hold**).*

12.3 Function Scheme & Control Register



ADCO: 1x, continuous **ADACT**: busy
ADTRIG: ADCSC1 wr, ADHWT pin **MODE**: 8, 10, 12bit
ACFE: enable **ACFGT**: less, greater

0x0010	ADCSC1	COCO	AIEN	ADCO	ADCH				Status/Control 1
0x0011	ADCSC2	ADACT	ADTRG	ACFE	ACFGT	0	0	R	R	Status/Control 2
0x0012	ADCRH	0	0	0	0	ADR11	ADR10	ADR9	ADR8	Result High
0x0013	ADCRL	ADR7	ADR6	ADR5	ADR4	ADR3	ADR2	ADR1	ADR0	Result Low
0x0014	ADCCVH	0	0	0	0	ADCV11	ADCV10	ADCV9	ADCV8	Compare High
0x0015	ADCCVL	ADCV7	ADCV6	ADCV5	ADCV4	ADCV3	ADCV2	ADCV1	ADCV0	Compare Low
0x0016	ADCCFG	ADLPC	ADIV		ADLSMP	MODE		ADICLK		Configuration
0x0017	APCTL1	ADPC7	ADPC6	ADPC5	ADPC4	ADPC3	ADPC2	ADPC1	ADPC0	Pin Control 1
0x0018	APCTL2	-	-	-	-	ADPC11	ADPC10	ADPC9	ADPC8	Pin Control 2

12.4 Code Sample

```
#define ADC_RES_8BIT      0
#define ADC_RES_10BIT     2
#define ADC_RES_12BIT     1

// init, using - high speed mode, ADCK = 6 MHz
    (busclock / 4), enable four line sensor
void adcInit(void)
{
  // max conversion time: 40x ADCK cycles + 5
      bus clock cycles
  // 40x 167ns + 5x 42ns = 7.21us
  // long sample time for higher conversion
      accuracy
  ADCCFG_ADLPC = 0;         // high speed
  ADCCFG_ADLSMP = 1;        // long sample time
  ADCCFG_ADICLK = 0;        // bus clock = 24 MHz

  // create mask for only reading used channels
      , prevents external noise
  APCTL1 = 0xF0;            // LsL, LsML. LsMR,
      LsR
  //APCTL1 = (1<<adcLsL) | (1<<adcLsML) | (1<<
      adcLsMR) |  (1<<adcLsR);

  // valid clock: 0.4 - 4 MHz if ADLPC = 1
  ADCCFG_ADIV = 2;          // 0=/1,  1=/2,
      2=/4, 3=/8
}

// 12 bit resolution (most of the time to much
    noise prever 10/8 Bit)
uint16 adcGet12BitValue(AdcChannels ch)
{
    // set resolution (could also be
        ADC_RES_10BIT or ADC_RES_12BIT)
    ADCCFG_MODE = ADC_RES_12BIT;
    // start new conversion
    ADCSC1_ADCH = (uint8)ch;
    // wait until conversion has completed
    while(!ADCSC1_COCO);
    return ADCR;
}
```