SpartanMC

Users Manual

Table of Contents

List of Figures

List of Tables

1. Instruction Set Architecture

The SpartanMC uses two register addresses per instruction. The first operand (RD register) is automatically used as the destination of the operation. This slightly reduces the effectiveness of the compiler, but it is a reasonable decision with respect to the very limited instruction bit width of 18 bit.

The code efficiency is improved with an additional condition code register which is used to store the result of compare instructions (used for branches).

1.1. Instruction Types

The instruction set is composd of fixed 18 bit instructions grouped in the four types:

- R-Type (register)
- I-Type (immidiate)
- M-Type (memory)
- J-Type (jump)

1.1.1. R-Type

R-Type instructions are used for operations which takes two register values and computes a result, which is stored back into operand one.

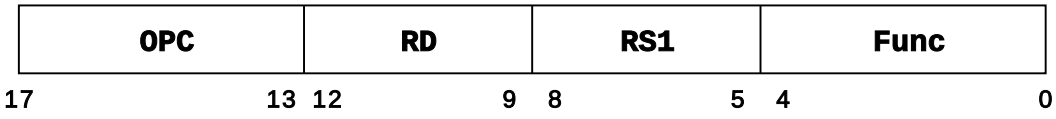


Figure 1-1: R-Type instruction

1.1.2. I-Type

This group includes all operations which take one register value and a constant to carry out an operation.

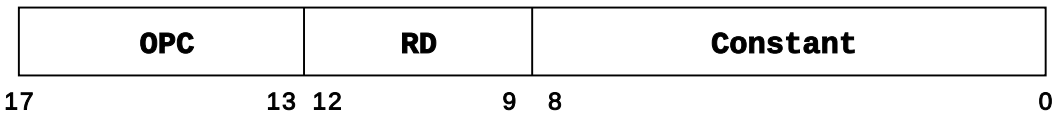


Figure 1-2: I-Type instruction

1.1.3. M-Type

This group is used for memory access operations. All load and store operations are available as half word (9 bit) or full word (18 bit) operation.

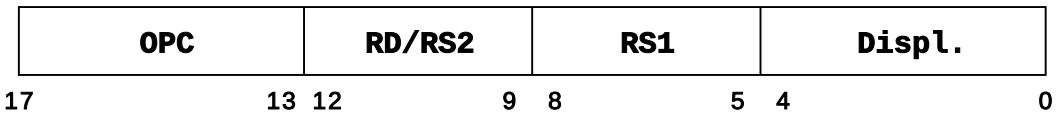


Figure 1-3: M-Type instruction

1.1.4. J-Type

This group includes the jump instruction and two branch instructions. The branch instructions interpret the condition code flag (see registers) to decide either to branch or not.

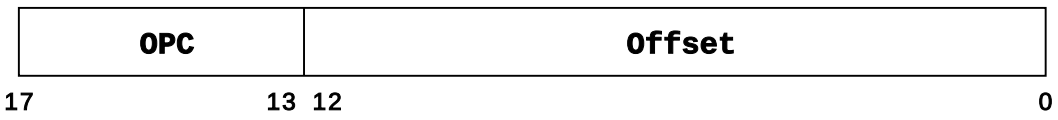


Figure 1-4: J-Type Instruction

1.2. Instruction Coding Matrices

The following table shows the instruction coding used on the SpartanMC.

Table 1-1: Main Matrix using IR 17-13

IR 17-13	..000	..001	..010	..011	..100	..101	..110	..111
00..	Special 1	Special 2	J	JALS	BEQZ	BNEZ	BEQZC	BNEZC
01..	ADDI	MOVI	LHI	SIGEX	ANDI	ORI	XORI	MULI
10..	L9	S9	L18	S18	SLLI	*	SRLI	SRAI
11..	SEQI	SNEI	SLTI	SGTI	SLEI	SGEI	IFADDUI	IFSUBUI

Table 1-2: Submatrix Special 1 using IR 4-0

IR 4-0	..000	..001	..010	..011	..100	..101	..110	..111
00..	orcc	andcc	*	*	SLL	MOV	SRL	SRA
01..	SEQU	SNEU	SLTU	SGTU	SLEU	SGEU	*	*
10..	*	*	*	*	*	*	CBITS	SBITS
11..	*	*	*	*	*	*	*	NOT

Table 1-3: Submatrix Special 2 using IR 4-0

IR 4-0	..000	..001	..010	..011	..100	..101	..110	..111
00..	RFE	TRAP	JR	JALR	JRS	JALRS	*	*
01..	*	*	*	*	*	*	*	*
10..	ADD	ADDU	SUB	SUBU	AND	OR	XOR	MUL
11..	SEQ	SNE	SLT	SGT	SLE	SGE	MOVI2S	MOVS2I

Note: * Code not used

Instructions written in lower case are currently not supported.

1.3. Register Window

The SpartanMC uses 16 addressable 18 bit registers which are stored in a 1k x 18 bit FPGA BlockRAM. The memory block is fully utilized through a sliding window technique. Registers 0 to 3 are used as global registers, registers 8 to 11 are locale registers. The registers 4 to 7 are used as function input window for parameter transfer from the calling function. It equals registers 12 to 15 of the calling function which allows up

to four parameters for a function call without external memory. Each shift consumes eight positions in the block memory which results in a total of 127 call levels. Register 11 is reserved for the return address of subroutines or interupt service routines (ISR).

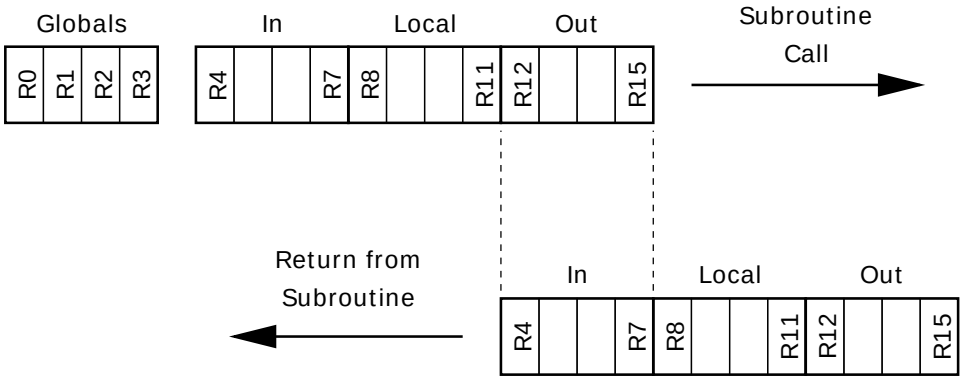


Figure 1-5: SpartanMC register window

1.4. Special Function Registers

For special purposes the SpartanMC contains four special function registers (SFR). These registers could be modified via SBITS/CBITS instructions.

Note: The contents of all SFRs remain constant until the next access to the corresponding register value.

1.4.1. Status Register (SFR_STATUS)

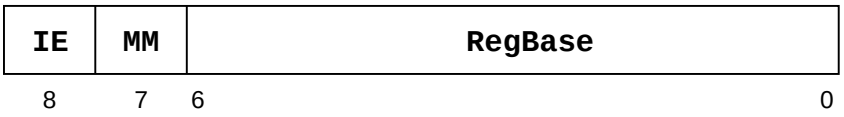


Figure 1-6: Status Register

SFR-Name: SFR_STATUS

SFR-Nr.: 0

SFR_STATUS [6:0]: Register Base (RegBase) - It contains the number of the current register window. The first window starts at 0. Each subroutine call increments the register by one up to the maximum value of 126.

SFR_STATUS [7]: Memory Management (MM) - This bit is set to 1 if the most significant address bit (address bit nr. 17) is used for memory access (see Address Management).

SFR_STATUS [8]: Interrupt Enable (IE) - If this bit is set to 0, the hardware interrupts are disabled. Setting IE to 1 enables the hardware interrupts.

1.4.2. LED Register (SFR_LEDS)

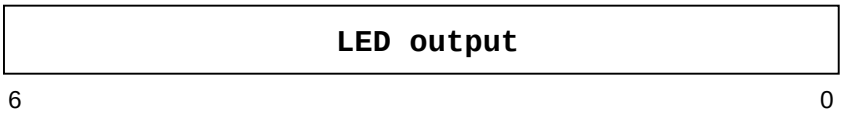


Figure 1-7: LED Register

SFR-Name: SFR_LEDS

SFR-Nr.: 1

SFR_LEDS [6:0]: This register is usable for custom status outputs.

1.4.3. MUL Register (SFR_MUL)

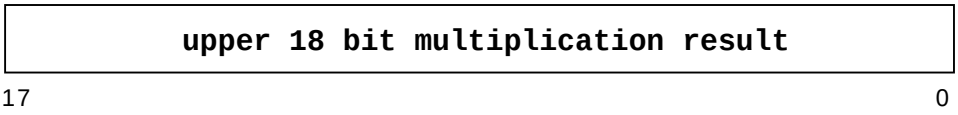


Figure 1-8: MUL Register

SFR-Name: SFR_MUL

SFR-Nr.: 2

SFR_MUL [17:0]: This register contains the upper 18 bit part [35:18] of a 36 bit result after a multiplication of two 18 bit values.

1.4.4. Condition Code Register (SFR_CC)

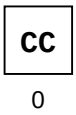


Figure 1-9: CC Register

SFR-Name: SFR_CC

SFR-Nr.: 3

SFR_CC [0]: Condition Code (CC) - The CC bit is used to store jump conditions. Furthermore it is used to signal an overflow after a signed arithmetic operation.

1.4.5. Interrupt Vector Register (SFR_IV)

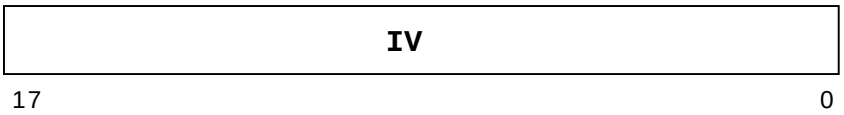


Figure 1-10: IV Register

SFR-Name: SFR_IV

SFR-Nr.: 4

SFR_IV [17:0]: Interrupt Vector (IV) - This Register contains the start address for the interrupt handling code (context switch and interrupt table lookup). After system reset this address is set to the value defined in the system configuration generated from jConfig. The start address of the interrupt handler can be changed by writing this register. This technique allows the usage of different interrupt service code for identical interrupts. It is recommended to disable the interrupts (set SFR_STATUS [8] to 0) befor writing SFR_IV.

1.4.6. Trap Vector Register (SFR_TR)

Figure 1-11: TR Register

SFR-Name: SFR_TR

SFR-Nr.: 5

This register contains the start address for trap service routines.

SFR_TR [17:8]: Trap (TR) - The upper 10 bits contain the base address of the trap table.

SFR_TR [7:0]: Trap (TR) - The lower 8 bits contain the number of the trap (read only - return 0x00 on read request). These bits are set via `trap` instruction.

1.5. Instruction Set Details

This section is a reference to the entire SpartanMC instruction set.

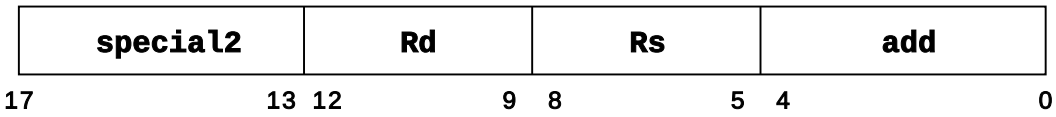
Each of the following pages covers a single SpartanMC instruction. They are organized alphabetically by instruction mnemonic.

add add

add

Mnemonic

add Rd , Rs



Pseudocode

Rd ← Rd + Rs

CC ← OV

Description

The content of GPR Rd and the content of GPR Rs are arithmetically added and form an 18 bit two's complement result, which is written to GPR Rd. If the result of the addition is greater than 2^{17}-1 (i.e.: = 0x1FFFF) or lower than -2^{17}(i.e.: 0x20000), an overflow occurs and CC is set to 1.

Comments

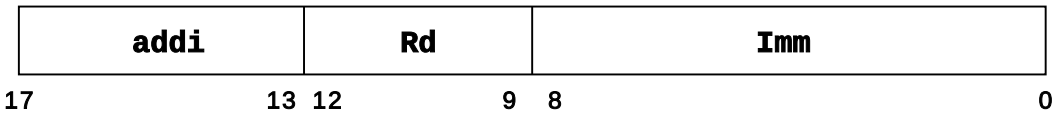
R-Typ

addi addi

add immediate

Mnemonic

```
addi        Rd ,  Imm
```



Pseudocode

$Rd \leftarrow Rd + IR_{8}{}^{9} \#\# IR_{8:0}$

Description

The content of GPR Rd and the immediate Imm are arithmetically added and form an 18 bit two's complement result, which is written to GPR Rd.

Comments

I-Typ

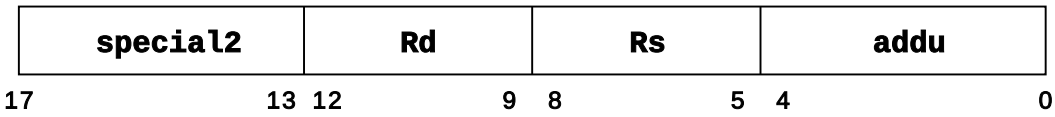
$IR_{8}{}^{9}\#\# IR_{8:0}$performs a sign extension for the 9 bit immediate.

addu addu

add unsigned

Mnemonic

addu Rd , Rs



Pseudocode

Rd ← Rd + Rs

Description

The content of GPR Rd and the content of GPR Rs are arithmetically added and form an 18-bit two's complement result which is written to GPR Rd.

Comments

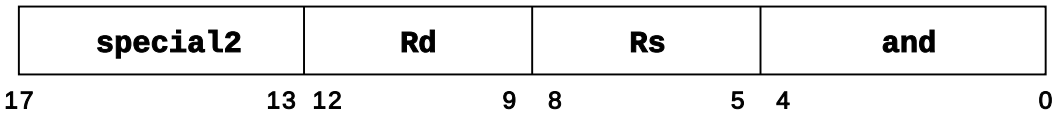
R-Typ

and and

and

Mnemonic

and Rd , Rs



Pseudocode

Rd ← Rd and Rs

Description

The content of GPR Rd is combined with the content of GPR Rs in a bitwise logical AND operation. The result is written to GPR Rd.

Comments

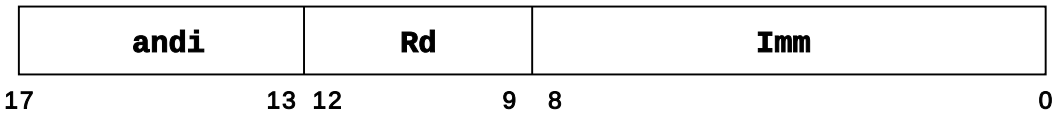
R-Typ

andi andi

and immediate

Mnemonic

```
andi        Rd ,  Imm
```



Pseudocode

$Rd \leftarrow Rd \text{ and } 0^{9} \ \# \# \ IR_{8:0}$

Description

The zero extended 9 bit immediate is combined with the content of GPR Rd in a bitwise logical AND operation. The result is written to GPR Rd.

Comments

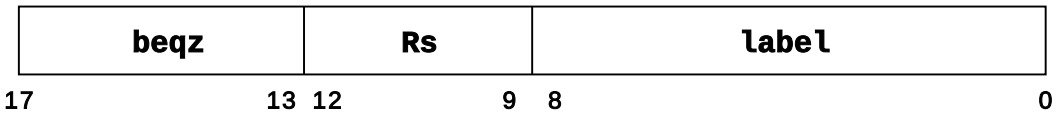
I-Typ

beqz beqz

branch equal zero

Mnemonic

```
beqz        Rs ,  label
```



Pseudocode

```
IF Rs=0; PC ← label
```

Description

Sets the program counter to the content of "label" if GPR Rs equals zero.

Comments

I-Typ

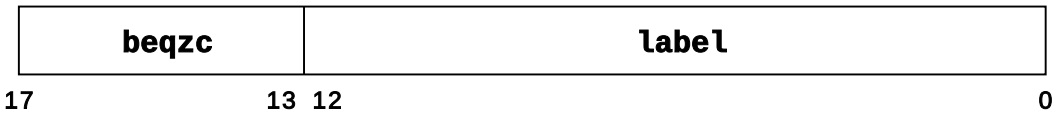
The value of GPR Rs must have been set at least two instructions before it is used by the `beqz` instruction.

beqzc beqzc

branch equal zero condition bit

Mnemonic

```
beqzc       label
```



Pseudocode

```
IF CC=0; PC ← label
```

Description

Sets the program counter to the content of "label" if CC has a value of zero.

Comments

J-Typ

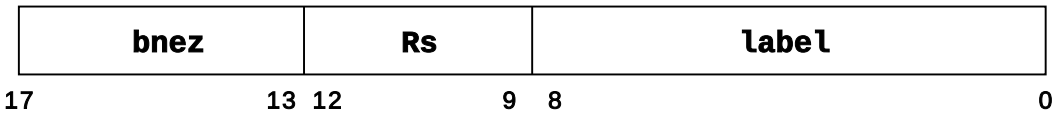
The value of CC must have been set at least two instructions before it is used by the `beqzc` instruction.

bnez bnez

branch not equal zero

Mnemonic

```
bnez        Rs ,  label
```



Pseudocode

```
IF Rs!=0; PC ← label
```

Description

Sets the programm counter to the content of "label" if GPR Rs is unequal to zero.

Comments

I-Typ

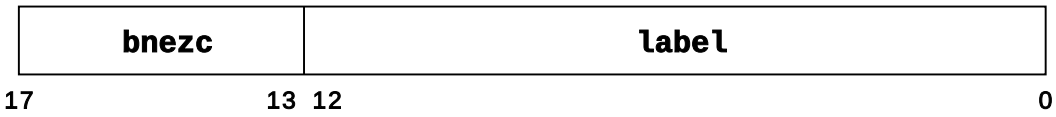
The value of GPR Rs must have been set at least two instructions before it is used by the `bnez` instruction.

bnezc bnezc

branch not equal zero condition bit

Mnemonic

```
bnezc       label
```



Pseudocode

```
IF CC!=0; PC ← label
```

Description

Sets the program counter to the content of "label" if CC is unequal to zero.

Comments

J-Typ

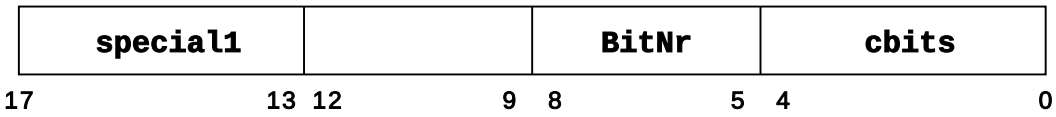
The value of CC must have been set at least two instructions before it is used by the `bnezc` instruction.

cbits cbits

clears bit at SFR

Mnemonic

```
cbits       BitNr
```



Pseudocode

```
SFR_Status_Bit ← 0
BitNr.: 0 = clears SFR_CC (CC)
BitNr.: 1 = clears SFR_STATUS7(MM)
BitNr.: 2 = clears SFR_STATUS8(IE)
```

Description

Clears a SFR bit according to the given BitNr. A BitNr of zero sets the CC bit to zero, a BitNr of one sets the MM bit to zero and a BitNr of two sets the IE bit to zero.

Comments

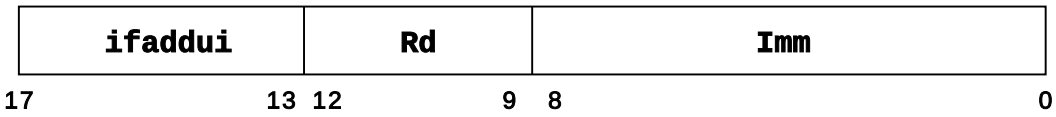
R-Typ

ifaddui ifaddui

conditional addition with an unsigned immediate

Mnemonic

```
ifaddui     Rd ,  Imm
```



Pseudocode

$$\texttt{IF CC = 1; Rd} \leftarrow \texttt{Rd + } 0^{9} \texttt{ \#\# IR}_{8:0}$$

Description

If the value of CC is one, the addition of the zero extended 9 bit immediate with the content of GPR Rd is carried out. The unsigned 18 bit result is written to GPR Rd. Otherwise GPR Rd remains unmodified.

Comments

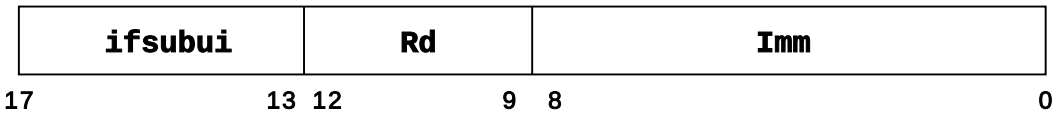
I-Typ

ifsubui ifsubui

conditional subtraction with an unsinged immediate

Mnemonic

```
ifsubui     Rd ,  Imm
```



Pseudocode

$IF\ CC=1;\ Rd \leftarrow Rd\ -\ 0^{9}\ \#\#\ IR_{8:0}$

Description

If the value of CC is one, the subtraction of the zero extended 9 bit immediate from the content of GPR Rd is carried out. The unsigned 18 bit result is written to GPR Rd.

Comments

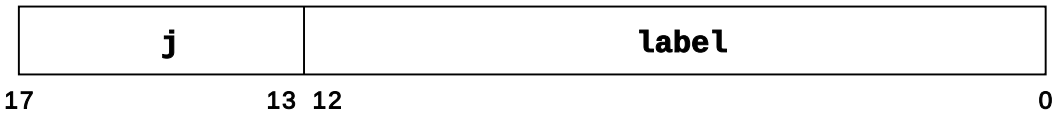
I-Typ

j j

jump

Mnemonic

```
j           label
```



Pseudocode

PC ← label

Description

Sets the PC unconditionally to the target address given with the value of "label".

Comments

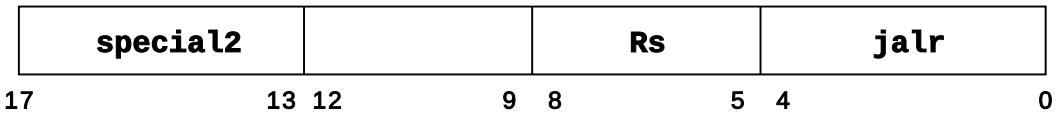
J-Typ

jalr jalr

jump and link register

Mnemonic

```
jalr        Rs
```



Pseudocode

```
R11 ← PC + 1
PC ← Rs
```

Description

Sets the program counter (PC) to the value of GPR Rs. The address of the instruction after the delay slot is written to GPR R11.

Comments

R-Typ

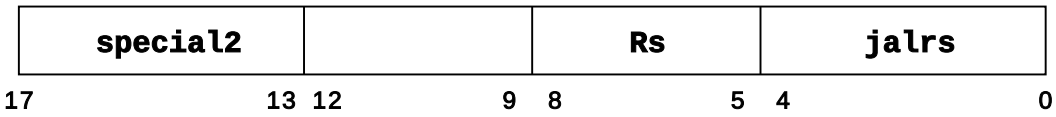
The value of the destination address in GPR Rs must have been set at least two instructions before it is used by the `jalr` instruction. Thus, each subroutine must have at least one instruction and the return code `jrs R11` at its end.

jalrs jalrs

jump and link and shift register window

Mnemonic

```
jalrs       Rs
```



Pseudocode

```
RegBase ← RegBase + 1
R11 ← PC + 1
PC ← Rs
```

Description

This instruction performs a shift of the register window for eight register positions. This is used for subroutine calls. The current PC is incremented and stored in R11 of the new register window. R11 is used to store the return address of the calling function. The value for RegBase which holds the current subroutine call level ($SFR\_STATUS_{6:0}$) is also incremented. Finally, the PC is set to the given address in GPR Rs.

Comments

R-Typ

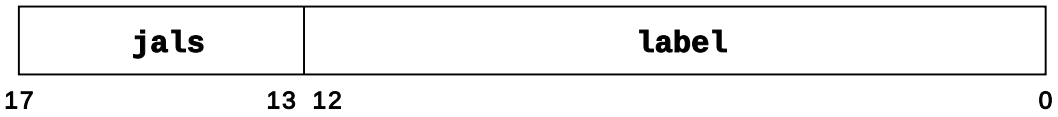
The value of destination address in GPR Rs must have been set at least two instructions before it is used by the `jalrs` instruction. Thus, each subroutine must have at least one instruction and the return code `jrs R11` at its end.

jals jals

jump and link and shift register window

Mnemonic

```
jals        label
```



Pseudocode

```
RegBase ← RegBase + 1
R11 ← PC + 1
PC ← label
```

Description

This instruction performs a shift of the register window for eight register positions. This is used for subroutine calls. The current PC is incremented and stored in R11 of the new register window. R11 is used to store the return address of the calling function. The value for RegBase which holds the current function call level ($SFR\_STATUS_{6:0}$) is also incremented. Finally, the PC is set to the 13 bit value of "label".

Comments

J-Typ

The subroutine must have at least one instruction and the return code `jrs R11` at its end.

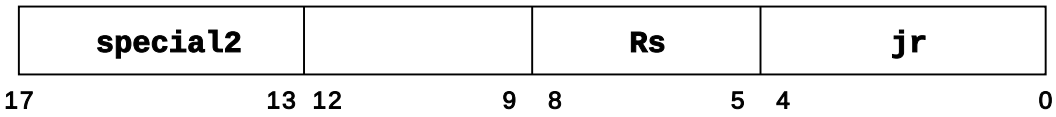
jr

jr

jump register

Mnemonic

```
jr          Rs
```



Pseudocode

PC ← Rs

Description

Set the PC unconditinally to the content of GPR Rs.

Comments

R-Typ

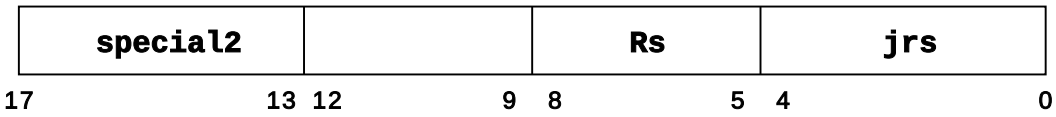
The value of the destination address in GPR Rs must have been set at least two instructions before it is used by the `jr` instruction.

jrs jrs

jump register shift register window (return subroutine)

Mnemonic

```
jrs          Rs
```



Pseudocode

```
PC ← Rs
RegBase ← RegBase - 1
```

Description

This instruction performs the return from a subroutine by a back-shift of the register window for eight register positions. The program counter (PC) is set to the content of GPR RS.

Comments

R-Typ

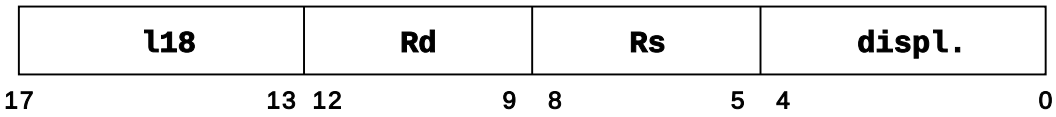
The value of the destination address in GPR Rs must have been set at least two instructions before it is used by the `jrs` instruction.

l18 l18

load 18 bit from memory

Mnemonic

```
l18         Rd ,  disp(Rs)
```



Pseudocode

Rd ← M[disp+Rs] ## M[disp+Rs+1]

Description

This instruction loads a sequence of two 9 bit words to an 18 bit register. The 5 bit displacement (disp) is zero-extended and added to the content of GPR Rs to form an unsigned 18 bit address. The 9 bit content of this address and the successor address is written to GPR Rd.

Comments

M-Typ

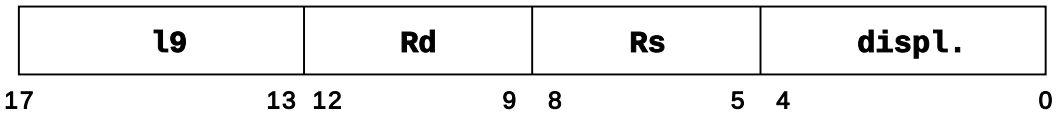
The given address must be even.

l9 l9

load 9 bit from memory

Mnemonic

```
l9          Rd ,  disp(Rs)
```



Pseudocode

$Rd \leftarrow 0^9$ ## M[disp+Rs]

Description

The 5 bit displacement (disp) is zero-extended and added to the content of GPR Rs to form an unsigned 18 bit address. The 9 Bit content of this address is written to GPR Rd.

Comments

M-Typ

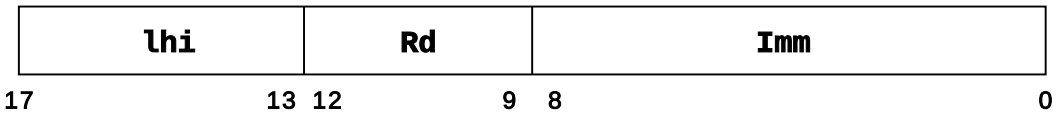
The given address can be even or odd.

lhi lhi

load high immediate

Mnemonic

```
lhi         Rd ,  Imm
```



Pseudocode

$Rd \leftarrow IR_{8:0} \ \#\# \ 0^{9}$

Description

This instruction writes the upper 9 bit part of GPR Rd. Therefore, the 9 bit immediate is concatenated with a 9 bit zero value and written to GPR Rd.

Comments

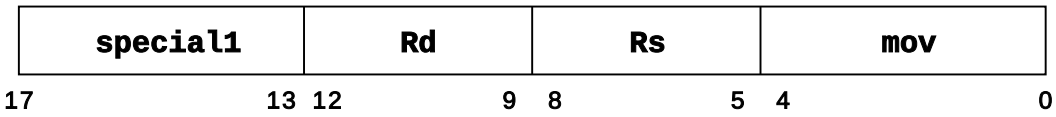
I-Typ

mov mov

move

Mnemonic

mov Rd , Rs



Pseudocode

Rd ← Rs

Description

The content of GPR Rs is written to GPR Rd.

Comments

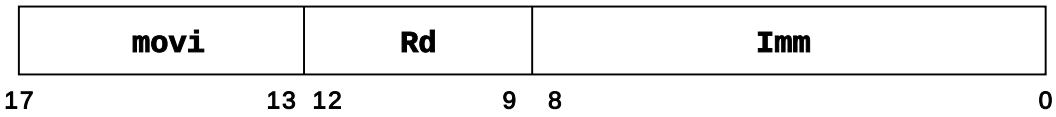
R-Typ

movi movi

move immediate

Mnemonic

```
movi        Rd ,  Imm
```



Pseudocode

$Rd \leftarrow 0^{9} \ \# \# \ IR_{8:0}$

Description

The content of a zero-extended 9 bit immediate is written to GPR Rd.

Comments

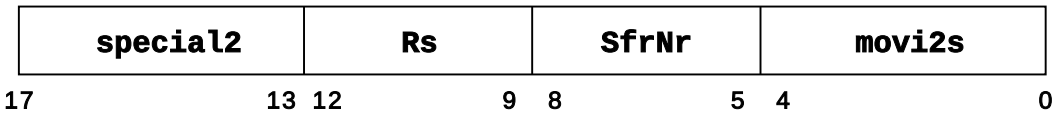
I-Typ

movi2s movi2s

move integer to special

Mnemonic

```
movi2s      SfrNr ,  Rs
```



Pseudocode

SFR ← Rs

Description

The content of GPR Rs is written to the SFR with the given SfrNr.

Comments

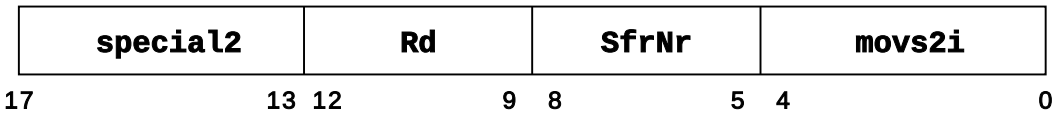
R-Typ

movs2i movs2i

move from special register to integer

Mnemonic

```
movs2i      Rd ,  SfrNr
```



Pseudocode

Rd ← SFR

Description

The content of the SFR with SfrNr is written to GPR Rd.

Comments

R-Typ

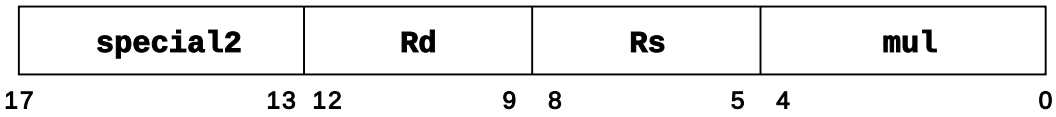
In this instruction code, $IR_{8:5}$holds the number of the SFR which is used as destination register for this instruction. The source register is given in $IR_{12:9}$.

mul mul

multiply

Mnemonic

```
mul         Rd ,  Rs
```



Pseudocode

```
SFR_MUL ## Rd ← Rd  *  Rs
```

Description

The content of GPR Rd and the content of GPR Rs are arithmetically multiplied, treating both operands as 18 bit two's complements values, and form a 36 bit two's complements result. The upper 18 bit part is written to SFR_MUL, the lower 18 bit part is written to GPR Rd.

Comments

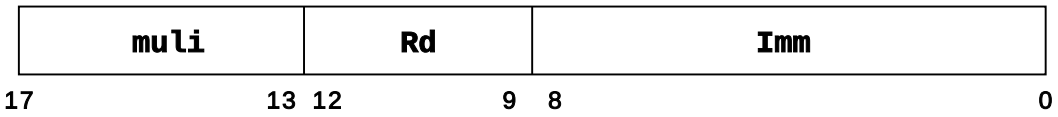
R-Typ

muli muli

multiply immediate

Mnemonic

```
muli        Rd ,  Imm
```



Pseudocode

$SFR\_MUL \,\#\#\, Rd \leftarrow Rd \times IR_{8}^{9} \,\#\#\, IR_{8:0}$

Description

The sign-extended 9 bit immediate and the content of GPR Rd are arithmetically multiplied, treating both operants as 18 bit two's complement values, and form a 36-bit two's complement result. The upper 18 bit part is written to SFR_MUL, the lower 18 bit part is written to GPR Rd.

Comments

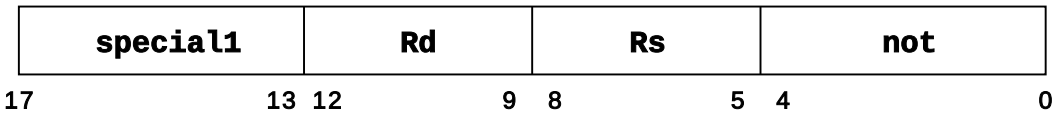
I-Typ

not not

not

Mnemonic

```
not         Rd ,  Rs
```



Pseudocode

```
Rd ← !Rs
```

Description

The content of GPR Rs is negated bitwise and the results is written to GPR Rd.

Comments

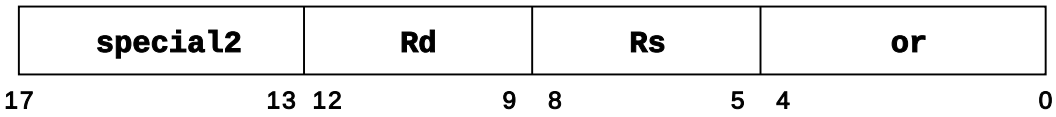
R-Typ

or or

or

Mnemonic

```
or          Rd ,  Rs
```



Pseudocode

```
Rd ← Rd or Rs
```

Description

The content of GPR Rs is combined with the content of GPR Rd in a bitwise logical OR operation, and the result is written to GPR Rd.

Comments

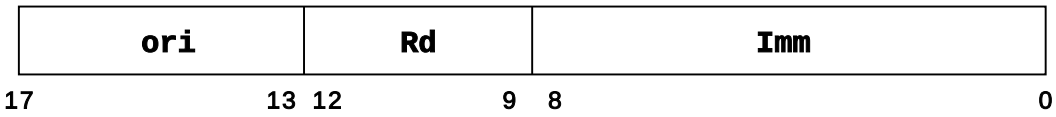
R-Typ

ori ori

or immediate

Mnemonic

```
ori         Rd ,  Imm
```



Pseudocode

$Rd \leftarrow Rd \text{ or } 0^{9} \ \# \# \ IR_{8:0}$

Description

The zero-extended 9 bit immediate is combined with the content of GPR Rd in a bitwise OR operation, and the result is written to GPR Rd.

Comments

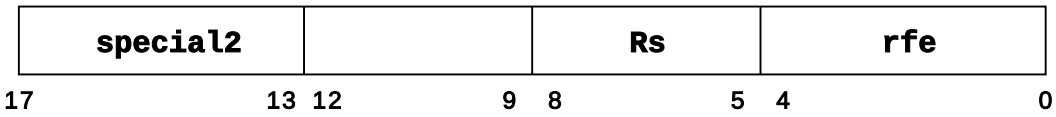
I-Typ

rfe rfe

return from exception

Mnemonic

```
rfe          Rs
```



Pseudocode

```
PC ← Rs
RegBase ← RegBase - 1
```

Description

This instruction performs the return from interrupt handling by a back-shift of the register window for eight register positions. The program counter (PC) is set to the content of GPR Rd and the interrupt is acknowledged.

Comments

R-Typ

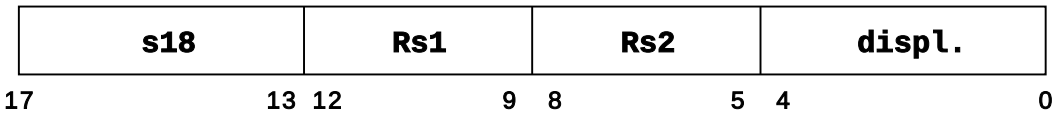
The value of the destination address in GPR Rs must have been set at least two instructions before it is used by the `rfe` instruction.

s18 s18

store 18 bit to memory

Mnemonic

```
s18         disp(Rs2) ,  Rs1
```



Pseudocode

```
M[disp+Rs2] ## M[disp+Rs2+1] ← Rs1
```

Description

The zero-extended 5 bit displacement (disp) is added to the content of GPR Rs2 to form an unsigned 18 bit address. The content of GPR Rs1 is stored at this address.

Comments

M-Typ

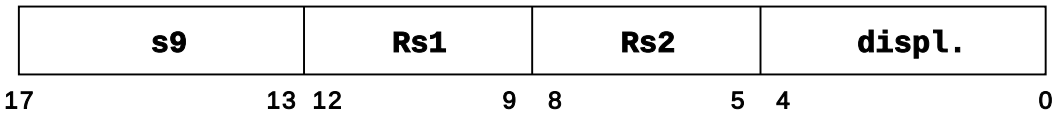
The 18 bit address must be even.

s9 s9

store 9 Bit to memory

Mnemonic

```
s9          disp(Rs2) ,  Rs1
```



Pseudocode

```
M[disp+Rs2] ← Rs1
```

Description

The 5 bit displacement (disp) is zero-extended and added to the content of GPR Rs2 to form an unsigned 18 bit address. The lower 9 bit part of GPR Rs1 is stored at this address.

Comments

M-Typ

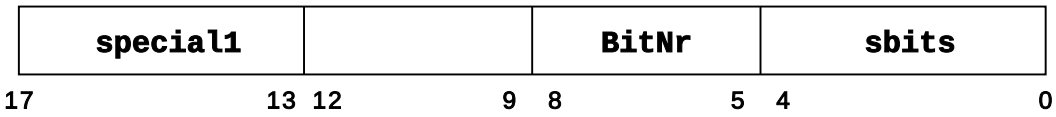
The given address can be even or odd.

sbits sbits

set bit at SFR-Register

Mnemonic

```
sbits       BitNr
```



Pseudocode

```
SFR_Status_Bit ← 1
BitNr.: 0 = sets SFR_CC
BitNr.: 1 = sets SFR_STATUS7(MM)
BitNr.: 2 = sets SFR_STATUS8(IE)
```

Description

This instruction sets a SFR bit according to the given BitNr. A BitNr of zero sets the CC bit to one, a BitNr of one sets the MM bit to one and a BitNr of two sets the IE bit to one.

Comments

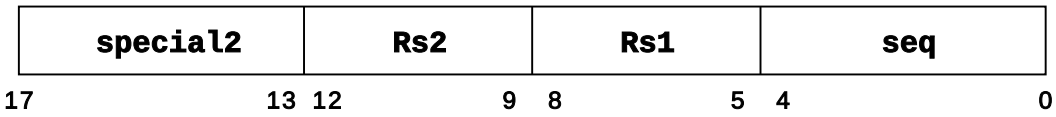
R-Typ

seq seq

set equal

Mnemonic

```
seq         Rs2 ,  Rs1
```



Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If both values are equal, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

R-Typ

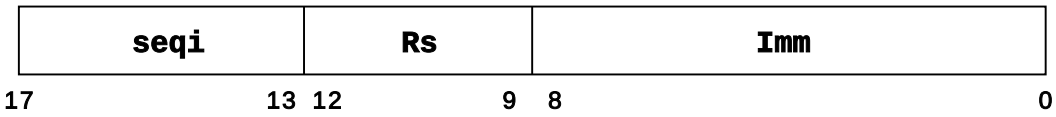
seqi

seqi

set equal immediate

Mnemonic

```
seqi        Rs ,  Imm
```



Pseudocode

$CC \leftarrow Rs - IR_{8}{}^{9} \#\# IR_{8:0}$

Description

This instruction compares the content of GPR Rs and the 9 bit immediate $IR_{8}{}^{9}\#\# IR_{8:0}$. If both values are equal, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The content of GPR Rs is lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

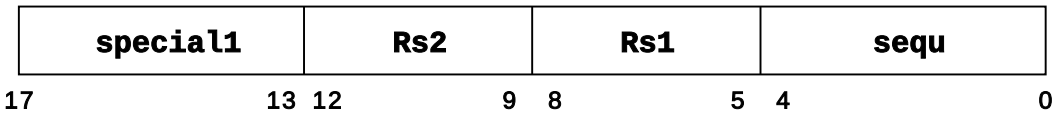
I-Typ

sequ sequ

set equal unsigned

Mnemonic

```
sequ        Rs2 ,  Rs1
```



Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If both values are equal, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{17}-1 and greater than or equal to zero.

Comments

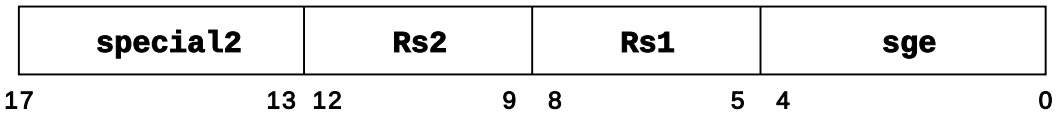
R-Typ

sge sge

set greater than or equal

Mnemonic

```
sge         Rs2 ,  Rs1
```



Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is equal to or greater than the value of Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

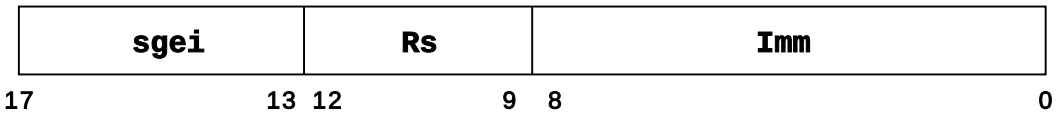
R-Typ

sgei sgei

set greater than or equal immediate

Mnemonic

```
sgei        Rs ,  Imm
```



Pseudocode

$CC \leftarrow Rs - IR_{8}^{9} \,\#\#\, IR_{8:0}$

Description

This instruction compares the content of GPR Rs and the content of a 9 bit immediate. If the value of Rs is equal to or greater than the immediate, the result will be one, otherwise the reslt will be zero. The 18-bit result is written to SFR_CC. The content of GPR Rs is lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

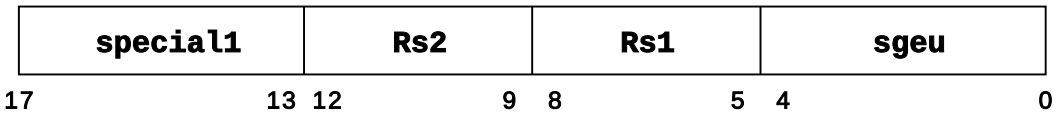
I-Typ

sgeu sgeu

set greater than or equal unsigned

Mnemonic

```
sgeu        Rs2 ,  Rs1
```



Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is equal to or greater than the value of Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{18}-1 and greater than or equal to zero.

Comments

R-Typ

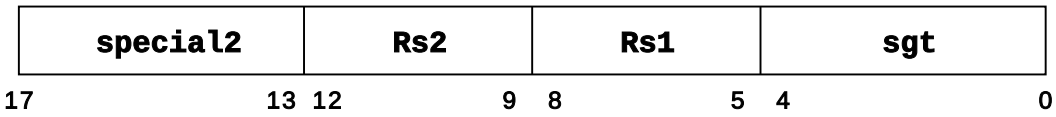
In this instruction -1 = 0x3FFFF is bigger than Ox1FFFF.

sgt sgt

set greater than

Mnemonic

```
sgt         Rs2 ,  Rs1
```



Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of GPR Rs2 is greater than the value of GPR Rs1, the result will be one, otherwise, the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than 2^{17}-1 and greater than -2^{17}.

Comments

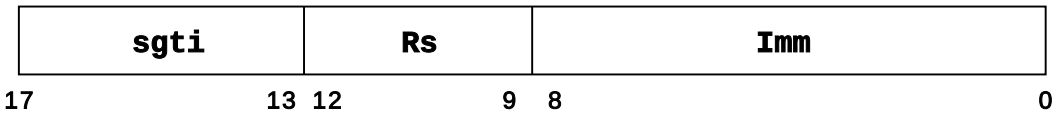
R-Typ

sgti sgti

set greater than immediate

Mnemonic

```
sgti        Rs ,  Imm
```



Pseudocode

$CC \leftarrow Rs - IR_{8}{}^{9} \,\#\#\, IR_{8:0}$

Description

This instruction compares the content of GPR Rs and a 9 bit immediate. If the value of GPR Rs is greater than the immediate, the result will be one, otherwise the result will be zero. This result is written to SFR_CC. The content of GPR Rs is lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

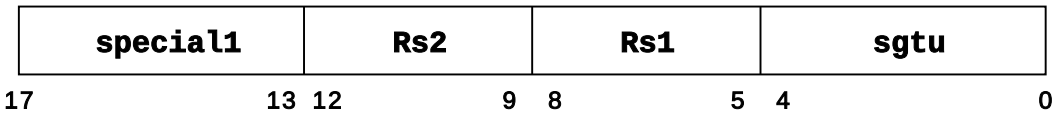
I-Typ

sgtu sgtu

set greather than unsigned

Mnemonic

```
sgtu        Rs2 ,  Rs1
```



Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is greater than the value of Rs1, the result will be one, otherwise, the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{18}-1 and greater than or equal to zero.

Comments

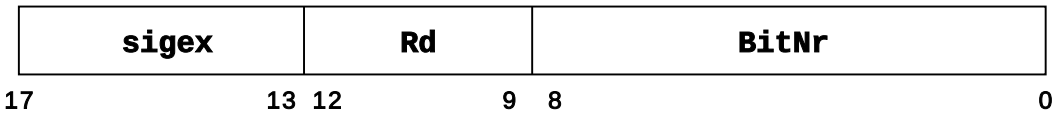
R-Typ

sigex sigex

signum extention

Mnemonic

```
sigex       Rd ,  BitNr
```



Pseudocode

$Rd \leftarrow Rd_{BitNr-1}^{17-BitNr-1} \# \# Rd_{(BitNr-1):0}$

Description

This instruction expands the content of Rd to an 18 bit value using the value of Rd at the given bit number (BitNr).

Comments

I-Typ

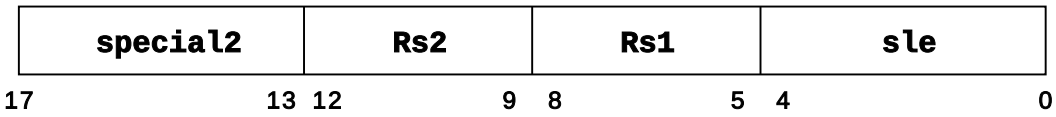
The allowed values for BitNr are 8, 9 or 16. Other values will be treated as 8.

sle sle

set less than or equal

Mnemonic

```
sle         Rs2 ,  Rs1
```



Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is equal to or less than the value of Rs2, the result will be one, otherwise, the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

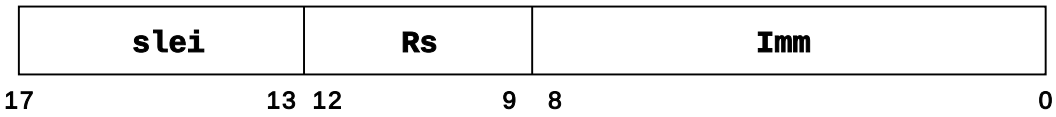
R-Typ

slei slei

set less than or equal immediate

Mnemonic

```
slei        Rs ,  Imm
```



Pseudocode

$CC \leftarrow Rs - IR_{8}^{9} \,\#\#\, IR_{8:0}$

Description

This instruction compares the content of GPR Rs and a 9 bit immediate. If the value of Rs is equal to or less than the immediate, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The content of GPR Rs is lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

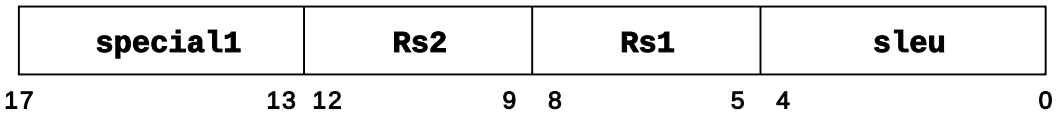
I-Typ

sleu sleu

set less than or equal unsigned

Mnemonic

```
sleu        Rs2 ,  Rs1
```



Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is equal to or less than the value of Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{18}-1 and greater than or equal to zero.

Comments

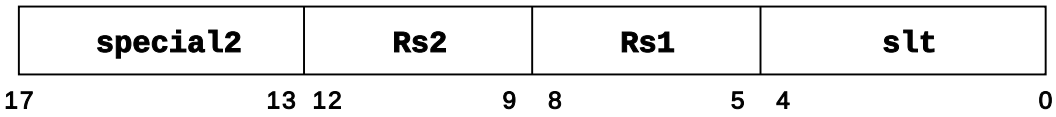
R-Typ

slt slt

set less than

Mnemonic

```
slt         Rs2 ,  Rs1
```



Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of GPR Rs2 is less than the value of GPR Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than 2^{17}-1 and greater than -2^{17}.

Comments

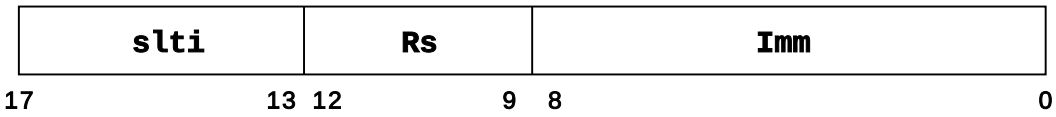
R-Typ

slti slti

set less than immediate

Mnemonic

```
slti        Rs ,  Imm
```



Pseudocode

$CC \leftarrow Rs - IR_{8}^{9} \,\#\#\, IR_{8:0}$

Description

This instruction compares the content of GPR Rs and a 9 bit immediate. If the value of Rs is less than the immediate, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The content of GPR Rs is lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

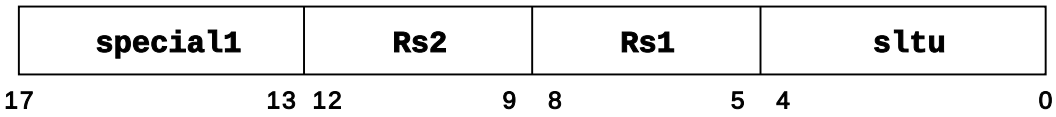
I-Typ

sltu sltu

set less than unsigned

Mnemonic

```
sltu        Rs2 ,  Rs1
```



Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is less than the value of Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{18}-1 and greater than or equal to zero.

Comments

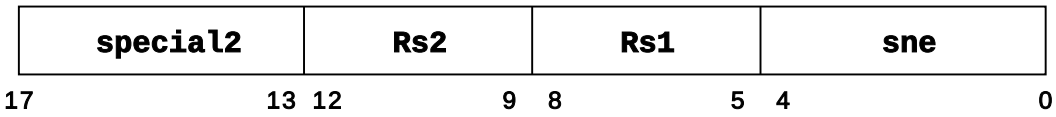
R-Typ

sne sne

set not equal

Mnemonic

sne Rs2 , Rs1



Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of GPR Rs2 is lower or greater than the value of GPR Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than 2^{17}-1 and greater than -2^{17}.

Comments

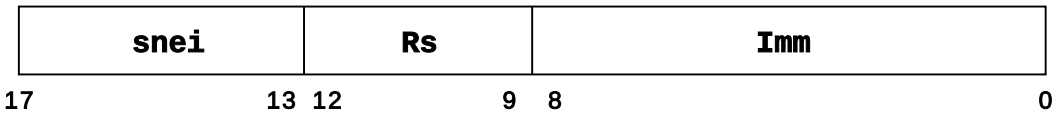
R-Typ

snei snei

set not equal immediate

Mnemonic

```
snei        Rs ,  Imm
```



Pseudocode

$CC \leftarrow Rs - IR_{8}{}^{9} \# \# IR_{8:0}$

Description

This instruction compares the content of GPR Rs and the content of 9 bit immediate. If the value of Rs is greater or lower than the immediate, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The content of GPR Rs is lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

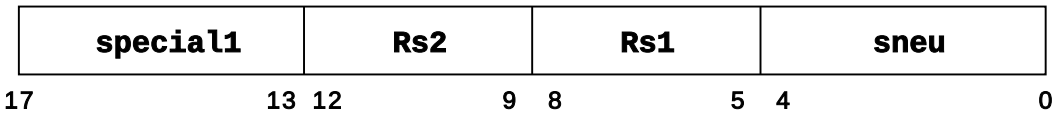
I-Typ

sneu sneu

set not equal unsigned

Mnemonic

sneu Rs2 , Rs1



Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is lower or greater than the value of Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{18}-1 and greater than or equal to zero.

Comments

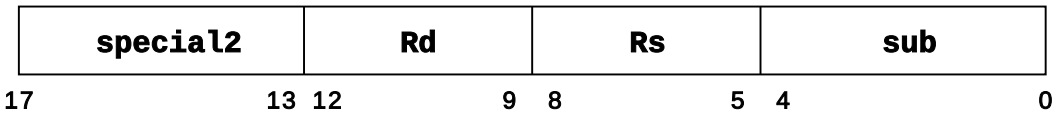
R-Typ

sub sub

subtract

Mnemonic

sub Rd , Rs



Pseudocode

Rd ← Rd - Rs

cc ← ov

Description

The content of GPR Rs is arithmetically subtracted from the content of GPR Rd and forms an 18 bit two's complement result, which is written to GPR Rd. If the result of the subtraction is greater than 2^{17}-1 (i.e.: = 0x1FFFF) or lower than -2^{17}(i.e.: 0x20000), an overflow occurs and CC is set to 1.

Comments

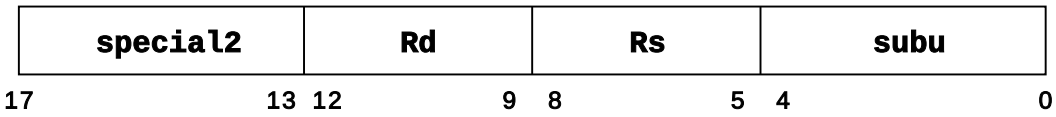
R-Typ

subu subu

subtract unsigned

Mnemonic

subu Rd , Rs



Pseudocode

Rd ← Rd - Rs

Description

The content of GPR Rs is arithmetically subtracted from the content of GPR Rd and forms an 18 bit unsigned result which is written to GPR Rd. This instruction can not produce an overflow exception, which is the only difference between this instruction and the sub instruction.

Comments

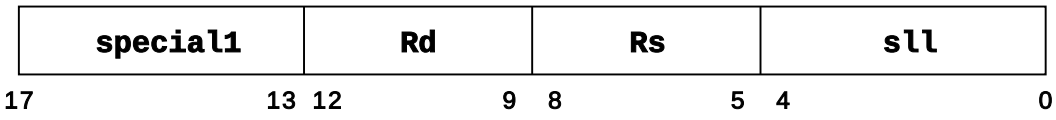
R-Typ

sll sll

shift left logical

Mnemonic

```
sll         Rd ,  Rs
```



Pseudocode

```
Rd ← CC ## Rd  «  Rs
```

Description

This instruction performs a left shift operation of GPR Rd. The shift width is set to the value of GPR Rs. The free bit positions are filled with zeros. The value of the highest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.

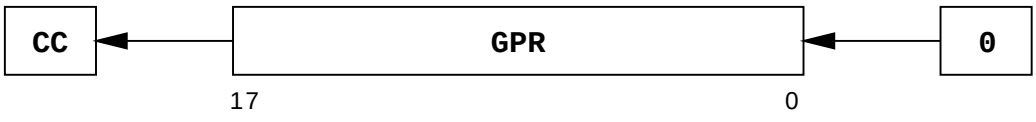


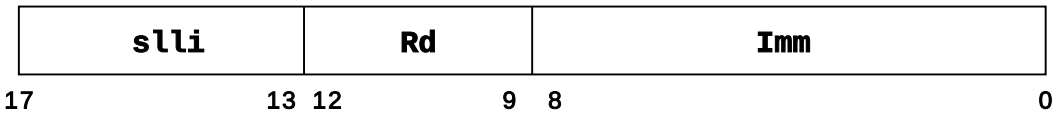
Figure 1-12: shift left logical

slli slli

shift left logical immediate

Mnemonic

```
slli        Rd ,  Imm
```



Pseudocode

$Rd \leftarrow (CC \,\#\#\, Rd) \ll (0^{9} \,\#\#\, IR_{8:0})$

Description

This instruction performs a left shift operation of GPR Rd. The shift width is set by a zero-extended 9 bit immediate. The free bit positions are filled with zero. The value of the highest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.

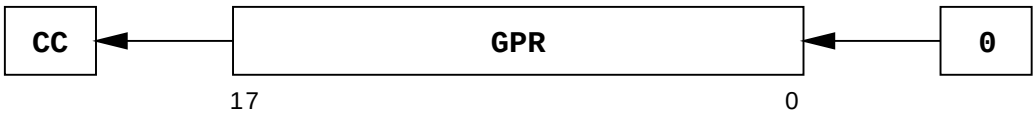


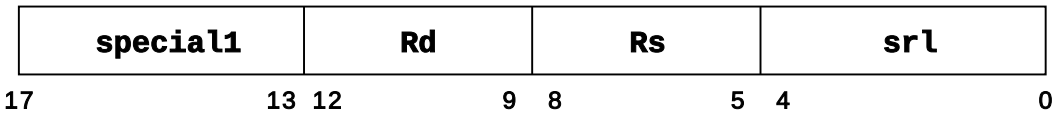
Figure 1-13: shift left logical immediate

srl srl

shift right logical

Mnemonic

```
srl         Rd ,  Rs
```



Pseudocode

```
Rd ← (Rd ## CC) » Rs
```

Description

This instruction performs a right shift operation of GPR Rd. The shift width is set to the value of GPR Rs. The free bit positions are filled with zeros. The value of the lowest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.

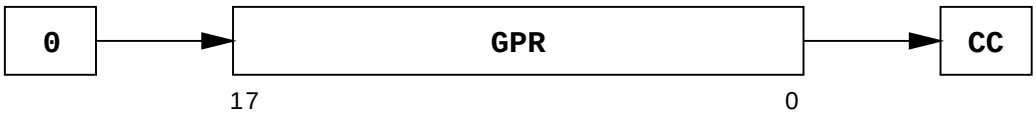


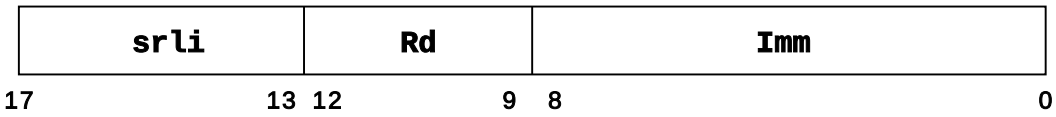
Figure 1-14: shift right logical

srli srli

shift right logical immediate

Mnemonic

```
srli        Rd ,  Imm
```



Pseudocode

$Rd \leftarrow Rd \,\#\#\, CC \gg 0^{9} \,\#\#\, IR_{8:0}$

Description

This instruction performs a right shift operation of GPR Rd. The shift width is set by a zero-extended 9 bit immediate. The free bit positions are filled with zero. The value of the lowest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.

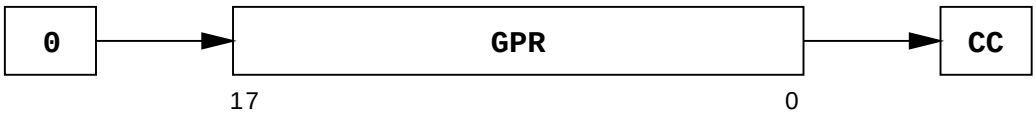


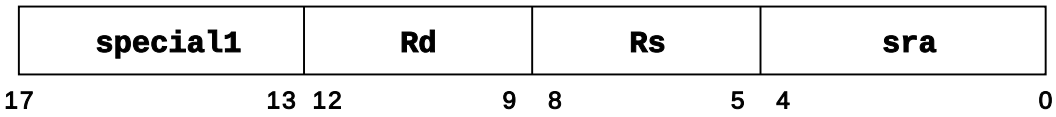
Figure 1-15: shift right logical immediate

sra sra

shift right arithmetic

Mnemonic

```
sra         Rd ,  Rs
```



Pseudocode

$Rd \leftarrow (Rd \,\#\#\, CC) \gg_a Rs$

Description

This instruction performs a right shift operation of GPR Rd. The shift width is set to the value of GPR Rs. The free bit positions are filled with the highest bit of GPR Rd. The value of the lowest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.

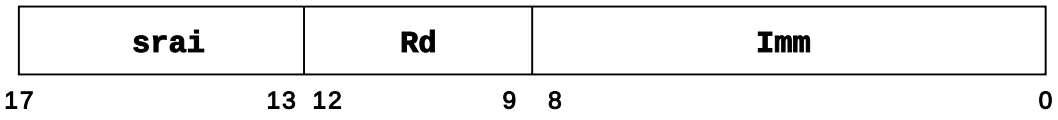
Figure 1-16: shift right arithmetic

srai srai

shift right arithmetic immediate

Mnemonic

```
srai        Rd ,  Imm
```



Pseudocode

$Rd \leftarrow Rd \,\#\#\, CC \gg {}_{a}0^{9} \,\#\#\, IR_{8:0}$

Description

This instruction performs a right shift operation of GPR Rd. The shift width is set by a zero-extended 9 bit immediate. The free bit positions are filled with the highest bit of GPR Rd. The value of the lowest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.

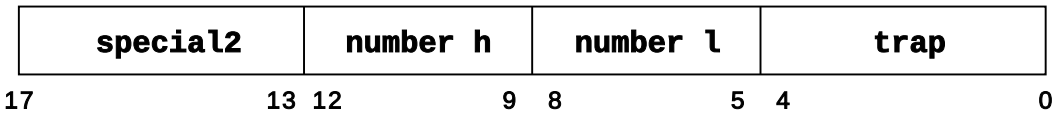
Figure 1-17: shift right arithmetic immediate

trap trap

trap

Mnemonic

trap Number



Pseudocode

RegBase ← RegBase + 1

R11 ← PC + 1

PC ← $SFR\_TR_{17:8}$ ## Number

Description

This instruction performs a shift of the register window for eight register positions. The current PC is incremented and stored in R11 of the new register window. R11 is used to store the return address of the calling function. The value for RegBase holding the current subroutine call level ($SFR\_STATUS_{6:0}$) is also incremented. Finally, the PC is set to the 10 bit value of $SFR\_TR_{17:8}$ and to the 8 bit value of number (number h ## number l).

Comments

R-Typ

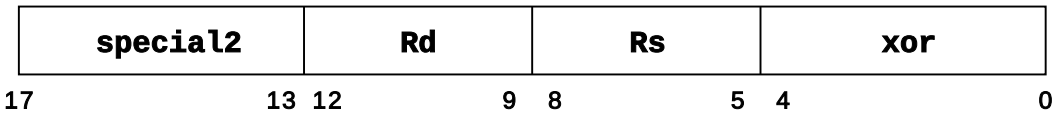
number = number h ## number l

xor xor

exclusive or

Mnemonic

```
xor         Rd ,  Rs
```



Pseudocode

```
Rd ← Rd xor Rs
```

Description

The content of GPR Rd is combined with the content of GPR Rs in a bitwise logical XOR operation, and the result is written to GPR Rd.

Comments

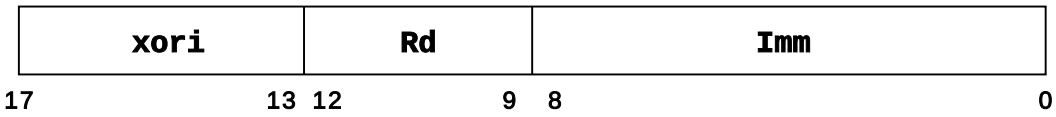
R-Typ

xori xori

exclusvie or immediate

Mnemonic

```
xori        Rd ,  Imm
```



Pseudocode

$Rd \leftarrow Rd \ \texttt{xor}\ 0^{9} \ \#\# \ IR_{8:0}$

Description

The zero-extended 9 bit immediate is combined with the content of GPR Rd in a bitwise logical XOR operation, and the result is written to GPR Rd.

Comments

I-Typ

2. Memory Organization

The SpartanMC main memory is a compound of single memory blocks of 2k rows with 18 bit width. The number of blocks and therefore the size of the main memory is configurable. The memory blocks are implemented by using the FPGA internal BlockRAMs. Each block consists of two FPGA BlockRAMs of 2k rows and 9 bit width. Since the FPGA BlockRAMs are dual ported, one port is used to read instructions and the other port is used to read and write data.

The SpartanMC stores data in big endian byte order.

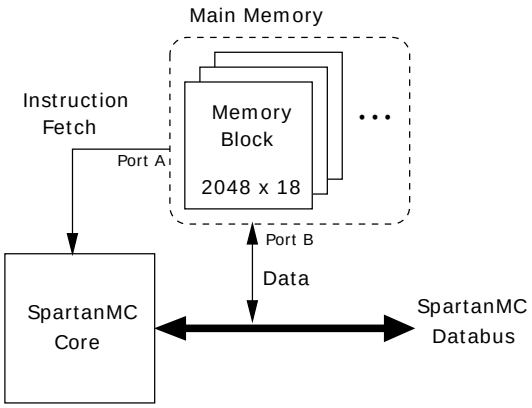


Figure 2-18: Dual ported main memory

2.1. Address Management

Each port of the main memory is connected to a 18 bit address bus. Since the main memory consists of 2k x 18 bit blocks, there are 11 bit required to address the rows within a block. The remaining 7 bit of the address bus are used to select the memory block. Therefore a possible maximum of addressable memory of 256k of 18 bit words distributed to 128 memory blocks could be instantiated. For the instruction port of the memory, the program counter (PC) is used as address bus.

For better memory utilization of the data section the data port provides a 9 bit wise memory access. Therefore the least significant bit (Align) of the data address bus is used to select the upper or lower half word which is used in load and store instructions (l9,s9). The remaining 17 bit are used to address the lower 128k of the memory. To address the upper 128k, the content of SFR_STATUS_7(MM) is used as most significant bit of the data address bus.

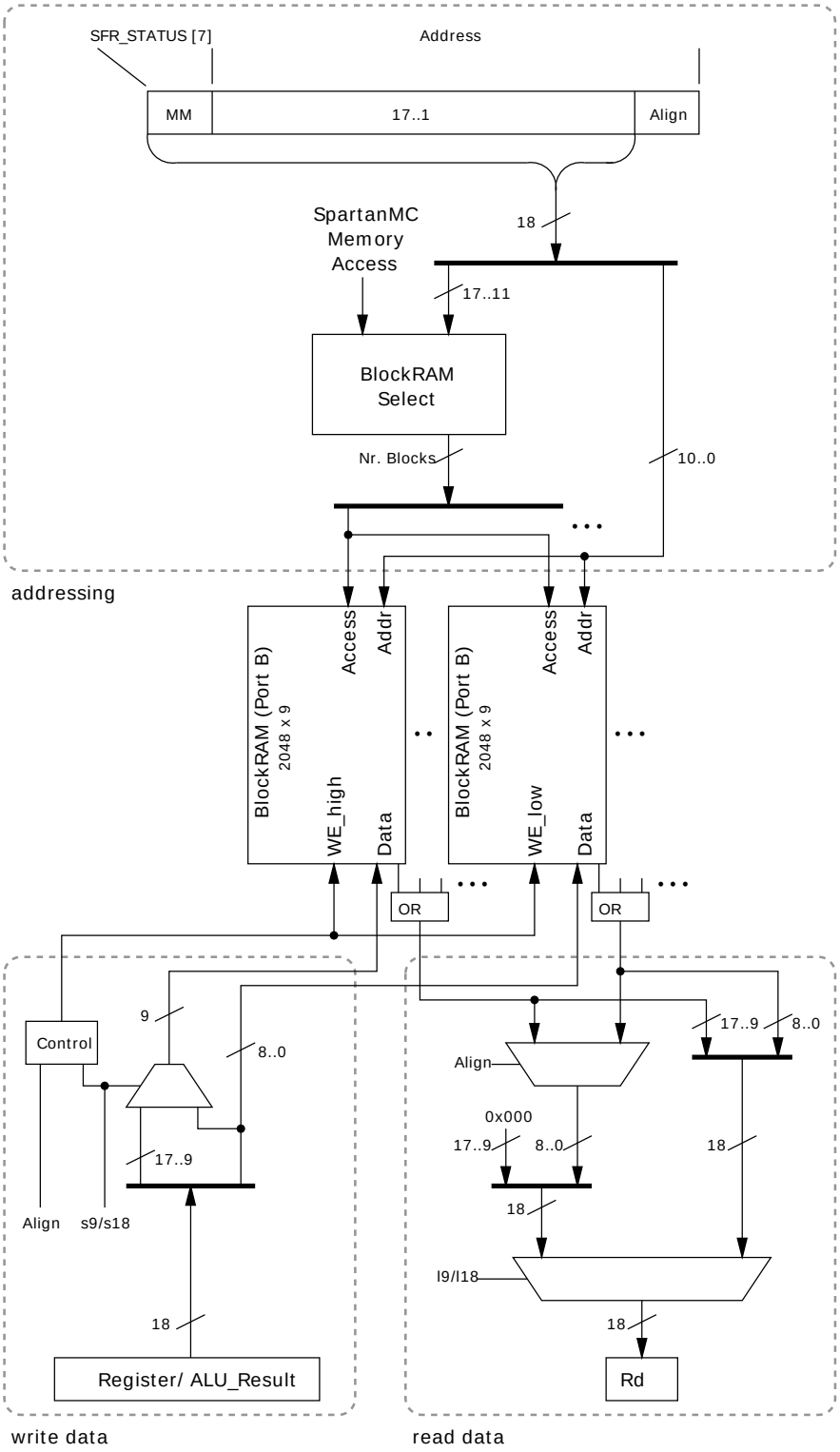


Figure 2-19: Data address management

Note: Due to the 9 bit wise data access, the correct address assignment of data addresses in assembler code has to be assured. The address value of the data address has to be twice the size of the regular instruction address.

2.2. Peripheral Access

2.2.1. Memory Mapped

Peripherals are connected to the regular data and address bus of the SpartanMC. Thus, peripheral devices are mapped to the SpartanMC address space at a dedicated address (IO_BASE_ADR). For exchanging small amounts of data between processor and peripheral, peripherals can provide a set of 18 bit registers. These registers are implemented as distributed memory on the FPGA.

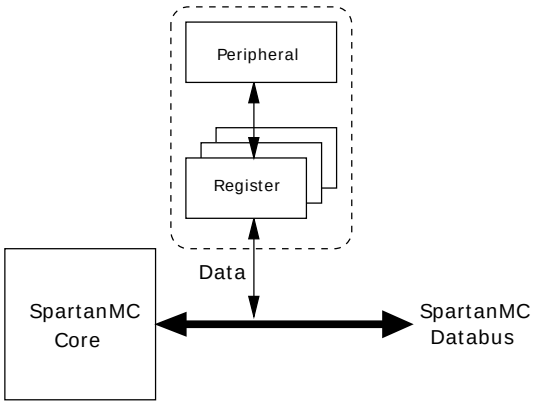


Figure 2-20: Memory mapped registers

The upper 8 bit part of the 18 bit address is used to select the peripheral address space. The selection is carried out by comparing the upper 8 bit part of the current address with the upper 8 bit of the configured base address (IO_BASE_ADR). The lower 10 bit are used to select the peripheral register within this address space. Therefore the 10 bit are divided into two parts: the first 9..n bit to access the correct peripheral module according to the BASE_ADR of the module and the second n-1..0 bit to access the 18 bit register within this peripheral module. The value of n depends on the number of registers provided by the peripheral (e.g. a value of n=3 implies a maximum of 8 registers for that module).

Note: The base address of the peripheral modules should be sorted by the number of registers. Starting with the peripheral using the most registers. This scheme avoids the overlapping of address spaces between different peripherals.

The data access to the registers is similar to the access to the main memory. For reading data (l9/l18) the align bit (LSB of the address) can be used to select the upper or lower half word of the register. For writing data the align bit is meaningless therefore only the s18 operation can be performed on peripheral memory.

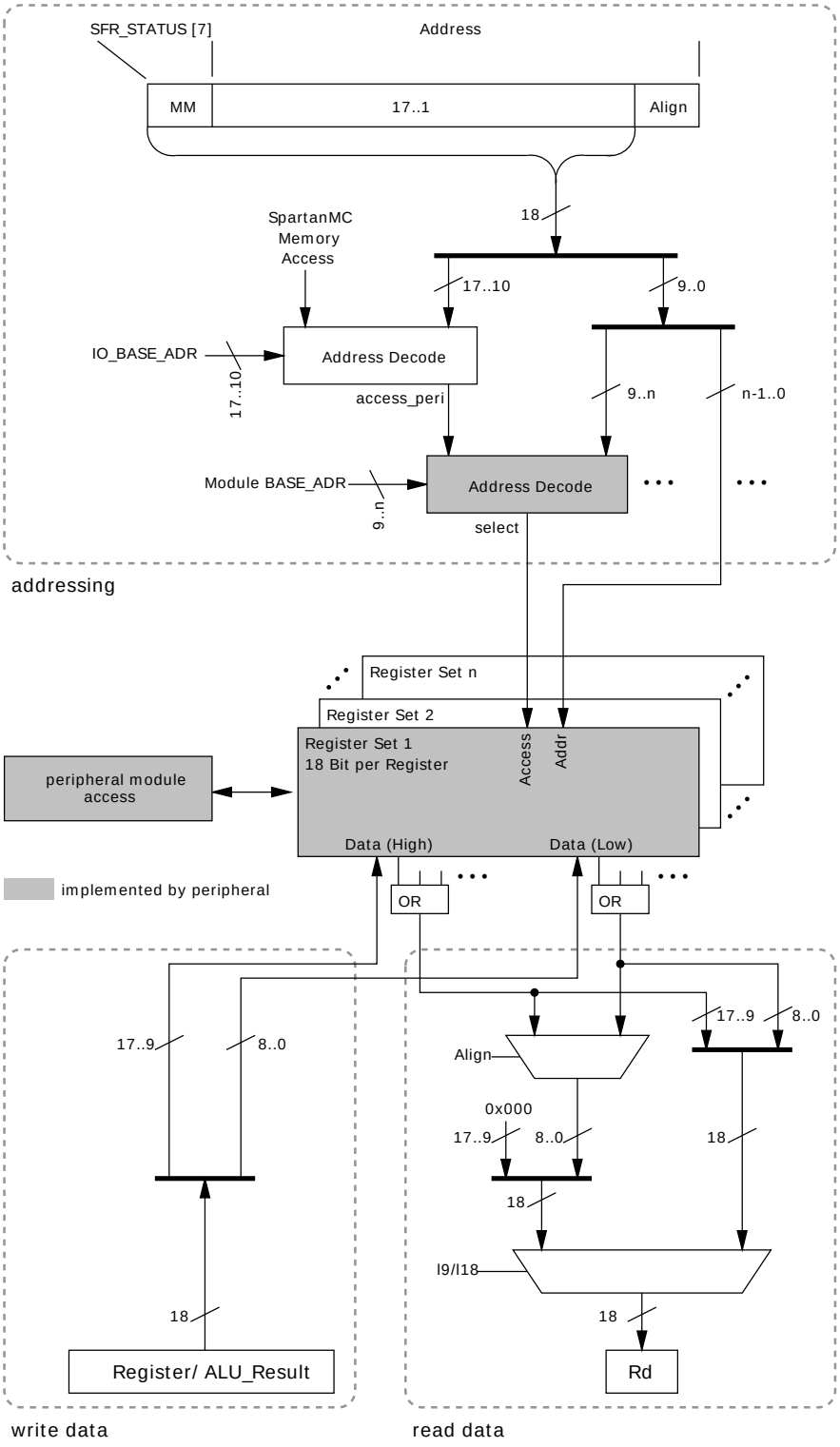


Figure 2-21: Peripheral register address management

2.2.2. Direct Memory Access (DMA)

Peripherals that work on large volumes of data can use BlockRAMs as data interface to the processor. In this case the first port is connected to the SpartanMC address and data bus and the second port is connected to the peripheral which works autonomously on the data in the memory block. This can be regarded as DMA style operation.

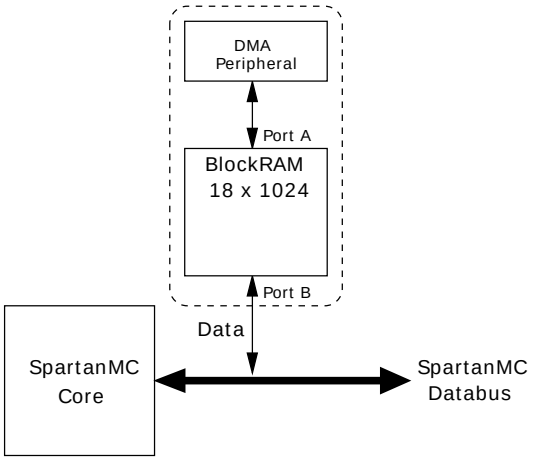


Figure 2-22: DMA with dual ported BlockRAM

Note: Due to the SpartanMC memory management which uses the second port of the BlockRAM as instruction fetch, the processor can not execute code form the DMA memory since the second port is used as peripheral interface. This missing master mode DMA would lead to copying overhead if data needs to be buffered between processing it with different peripherals.

The upper 8 bit part of the 18 bit address is used to select the DMA device. The selection is carried out by comparing the upper 8 bit part of the current address with the upper 8 bit of the configured base address (DMA_BASE_ADR). The lower 10 bit are used to select the row within the DMA BlockRAM.

The data access to the DMA memory is similar to the access to the main memory. For reading data (l9/l18) the align bit (LSB of the address) can be used to select the upper or lower half word of the register. For writing data the align bit is meaningless therefore only the s18 operation can be performt on peripheral memory.

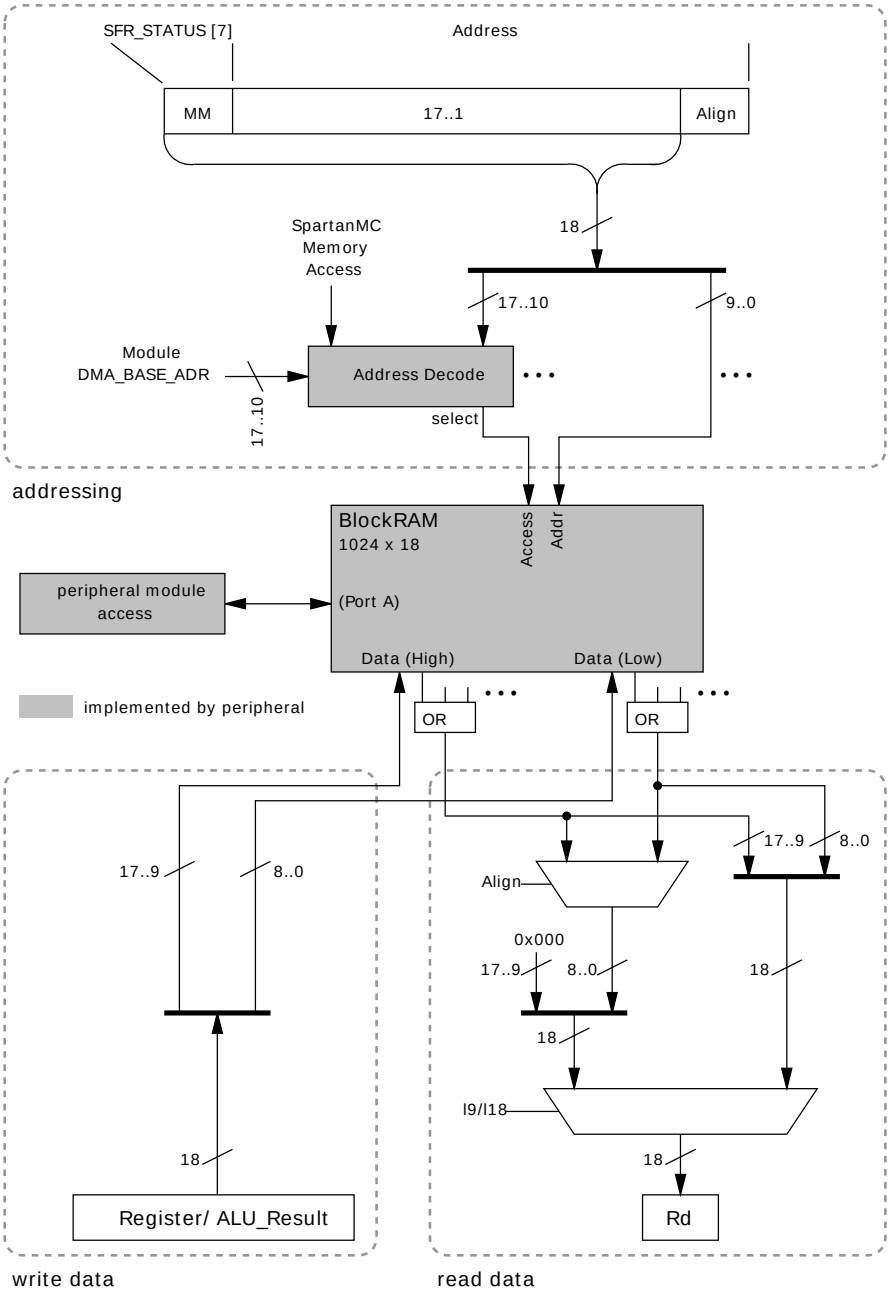


Figure 2-23: DMA address management

2.2.3. Data Read Interface

The main memory blocks and the peripheral memory are connected to the data memory interface of the processor core. In order to avoid tri-state buffers, all incoming data is combined through a wide or-gate. Thus, all memory blocks and peripherals that are currently not addressed must provide a value of zero on their outputs.

2.3. Example Memory Map

The following image describes a memory map for an SpartanMC example system and an application using traps and interrupts. The specfic addresses of the different application parts (ISR, Traps, IRQ Handler etc.) are automatically defined through compiler tools. The start addresses of DMA memory (in this example 0x19000) can be defined in the hardware configuraion generated through jConfig.

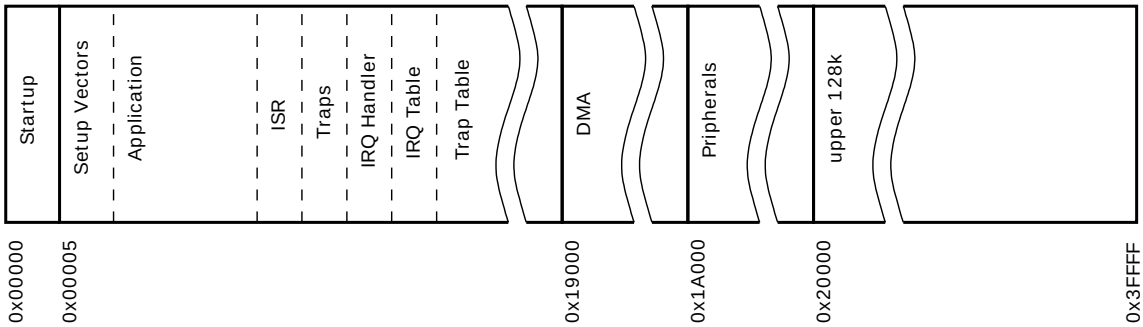


Figure 2-24: Example memory map

Startup: The startup code is generated by compiler tools at address 0x0000. It contains a branch to the application specific *Setup Vectors* -subroutine. The required branch address is defined within the system headers generated via jConfig.

Setup Vectors: Setup the address for the interrupt handler and the trap base address for this application.

Application: Contains the application code.

ISR: The interrupt service routines for the defined interrupts

Traps: The trap code for the defined traps.

IRQ Handler: Performs the IRQ prolog and epilog and links the IRQ table of the application.

IRQ Table: The interrupt branch table. Each 18 bit address contains the jump instructions to the interrupt code. The table length depends on the number of configured interrupts.

Trap Table: The branch table for traps. Each 18 bit address contains the jump instructions to a specific trap code. Since the the upper 10 bit are used as trap base address a maximum of 255 traps can be defined using the lower 8 bit. The implemented table length depends on the number of traps defined in the application

DMA: The memory section for DMA capable peripherals. This memory section is 18 bit aligned and contains data only.

Peripherals: The memory section for memory mapped peripherals. The start address of this section has to be set beyond the actual configured main memory section.

3. Simple Interrupt Controller (IRQ-Ctrl)

Depending on the requirements of the target application two types of interrupt controllers could be instantiated, IRQ-Ctrl and IRQ-Ctrlp. The simple interrupt controller (IRQ-Ctrl) provides a small resource footprint, but handles only one interrupt at once. Incoming interrupts (even of higher priority) will be ignored during the interrupt handling until the Interrupt Service Routine (ISR) is completed. Thus, the running ISR execution is **not** interruptable.

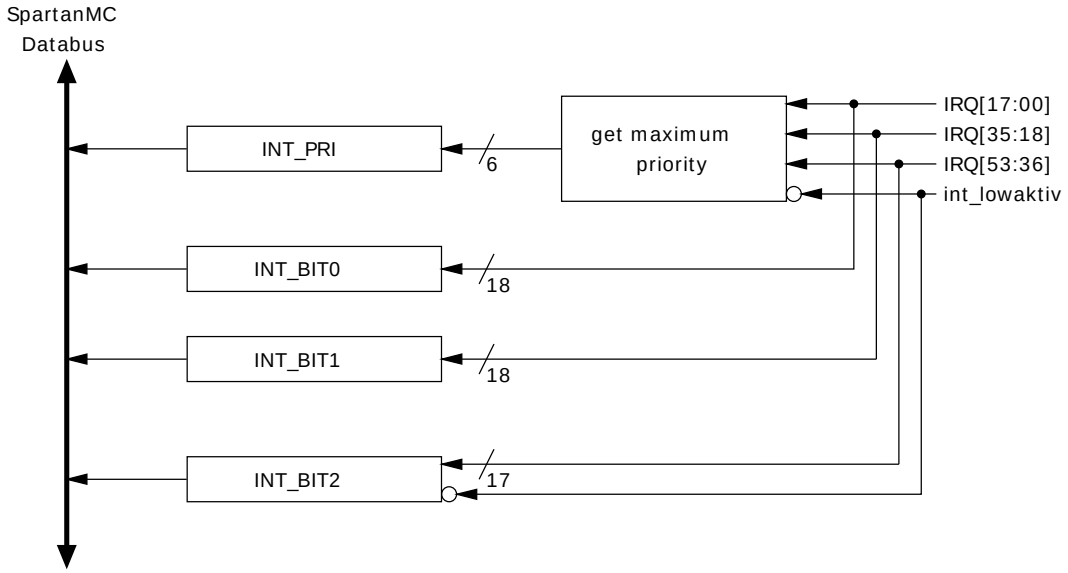


Figure 3-25: IRQ-Ctrl block diagram for IR_SOURCES=54

3.1. Function

The SpartanMC interrupt controller handles multiple interrupt sources as input sorted by their prority. Each interrupt capable peripheral must use a dedicate controller input signal for its interrupt request. If an interrupt occures the controller sets the interrupt input signal of the pipeline. The interrupt number of the pending interrupt with the highest priority could be read from INT_PRI register. In order to check all interrupt sources, the controller provides a configurable number of 18 bit registers (INT_BIT0..2) each with 18 interrupt flags.

Pending interrups are **not** stored within the interrupt controller. Thus, the logic to set/ hold, reset and mask interrups must be provided by the peripheral which is connected to the interrupt controller.

Note: The highest interrupt priority is reserved for the "int_lowactive" signal.

3.2. Module parameters

Table 3-4: IRQ-Ctrl modul parameters

Parameter	Default Value	Descripton
BASE_ADR	0x40	Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
IR_SOURCES	8	Number of IRQ Sources fot the SpartanMC core.

3.3. Peripheral Registers

3.3.1. IRQ-Ctrl Register Description

The IRQ-Ctrl peripheral provides three 18 bit registers for 18 interrupt sources. Register four and five could be used if additional interrupts are required. The registers are mapped to the SpartanMC address space located at 0x1A000 + BASE_ADR + Offset. Register three and four could be used if additional interrupts are required.

Table 3-5: IRQ-Ctrl registers

Offset	Name	Access	Description
0	INT_PRI	read	Register for the current max. Priority IRQ number.
1	INT_BIT0	read	Contains the current IRQ-sinals 0 to 17.
2	INT_BIT1	read	Contains the current IRQ-sinals 18 to 35.
3	INT_BIT2	read	Contains the current IRQ-sinals 36 to 54.

3.3.2. IRQ-Ctrl C-Header for Register Description

```
#ifndef INTR_CTRL_H_
#define INTR_CTRL_H_

   // Number of interrupts (i_bits)
   // is set by jconfig.

   typedef struct {
       volatile unsigned int int_pri;   // read
       volatile unsigned int int_bit0;   // read IRQ[17:00]
       volatile unsigned int int_bit1;   // read IRQ[35:18]
       volatile unsigned int int_bit2;   // read IRQ[53:36]
   } intr_ctrl_t;

#endif /*INTR_CTRL_H_*/
```

4. Complex Interrupt Controller (IRQ-Ctrlp)

Depending on the requirements of the target application two types of interrupt controllers could be instantiated (IRQ-Ctrl and IRQ-Ctrlp). The complex interrupt controller (IRQ-Ctrlp) allows the interruption of a running Interrupt Service Routine (ISR) by an interrupt of higher priority. Therefore, the interrupt enable bit (function call of "interrupt_enable()") must be set within the ISR. The complex interrupt controller can handle a maximum of 8 nested interrupts. The 9'th nested interrupt invocation is executed after completion of the 8'th ISR. It should be noted, that IRQ-Ctrlp requires much more FPGA resources than the simple IRQ-Ctrl.

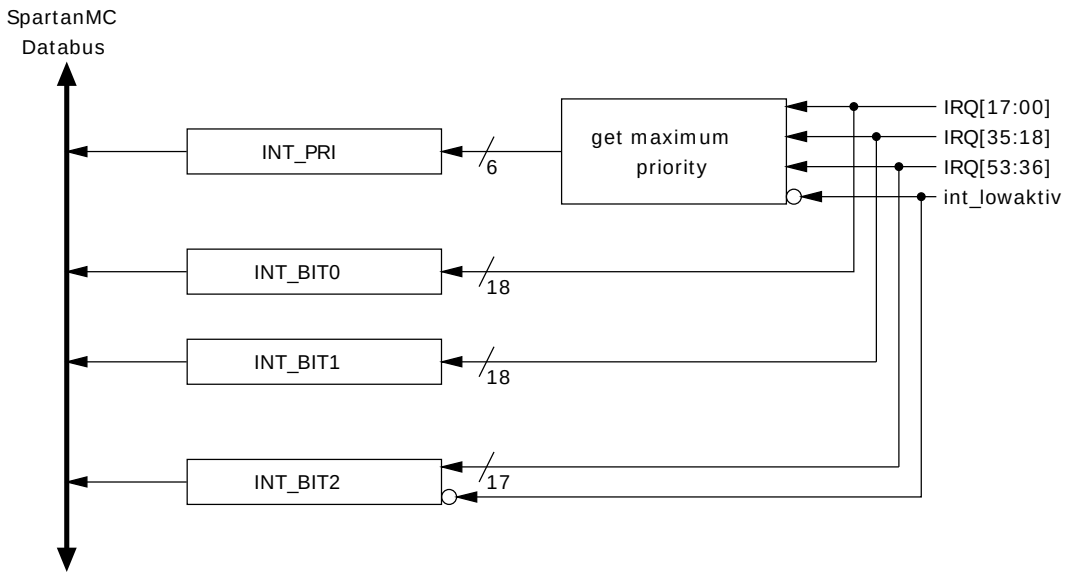


Figure 4-26: IRQ-Ctrl block diagram for IR_SOURCES=54

4.1. Function

The SpartanMC interrupt controller handles multiple interrupt sources as input sorted by their prority. Each interrupt capable peripheral must use a dedicate controller input signal for its interrupt request. If an interrupt occures the controller sets the interrupt input signal of the pipeline. The interrupt number of the pending interrupt with the highest priority could be read from INT_PRI register. In order to check all interrupt sources, the controller provides a configurable number of 18 bit registers (INT_BIT0..2) each with 18 interrupt flags.

Pending interrups are **not** stored within the interrupt controller. Thus, the logic to set/ hold, reset and mask interrups must be provided by the peripheral which is connected to the interrupt controller.

Note: The highest interrupt priority is reserved for the "int_lowactive" signal.

4.2. Module parameters

Table 4-6: IRQ-Ctrl modul parameters

Parameter	Default Value	Descripton
BASE_ADR	0x40	Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
IR_SOURCES	8	Number of IRQ Sources at SpartanMC Core.

4.3. Peripheral Registers

4.3.1. IRQ-Ctrl Register Description

The IRQ-Ctrl peripheral provides three or more 18 bit registers which are mapped to the SpartanMC address space located at 0x1A000 + BASE_ADR + Offset.

Table 4-7: IRQ-Ctrl registers

Offset	Name	Access	Description
0	INT_PRI	read	Register for the current max. Priority IRQ-Number.
1	INT_BIT0	read	Contains the current IRQ-Sinals 0 to 17.
2	INT_BIT1	read	Contains the current IRQ-Sinals 18 to 35.
3	INT_BIT2	read	Contains the current IRQ-Sinals 36 to 54.

4.3.2. IRQ-Ctrl Register Priority IRQ-Number

Table 4-8: Priority IRQ-Number register layout

Bit	Name	Access	Reset	Description
0-7	INT_PRI	read	0	Contains the current max. priority.
8-11	INT_CNT	read	0	Contains the current Number of nested interrupts.
12-17	x	read	0	Not used.

4.3.3. IRQ-Ctrl C-Header for Register Description

```
#ifndef INTR_CTRL_H_
   #define INTR_CTRL_H_

   // Number of interrupts (i_bits)
   // is set by jconfig.

   typedef struct {
       volatile unsigned int int_pri;   // read
       volatile unsigned int int_bit0;   // read IRQ[17:00]
       volatile unsigned int int_bit1;   // read IRQ[35:18]
       volatile unsigned int int_bit2;   // read IRQ[53:36]
   } intr_ctrl_t;

#endif /*INTR_CTRL_H_*/
```

5. Universal Asynchronous Receiver Transmitter (UART)

The UART is a SpartanMC peripheral device for serial communication with external systems. The UART enables a bit stream of 5-8 bits to be shifted in and out of the peripheral at a programmed bit rate. The peripheral is connected with the external system environment by the two signals Rx and Tx. If the UART is parametrized as modem, the additional signals DTR, DCD and CTS are usable. The incoming and outgoing data is written to configurable FIFO memory modules which are connected to the input and output shift registers of the UART.

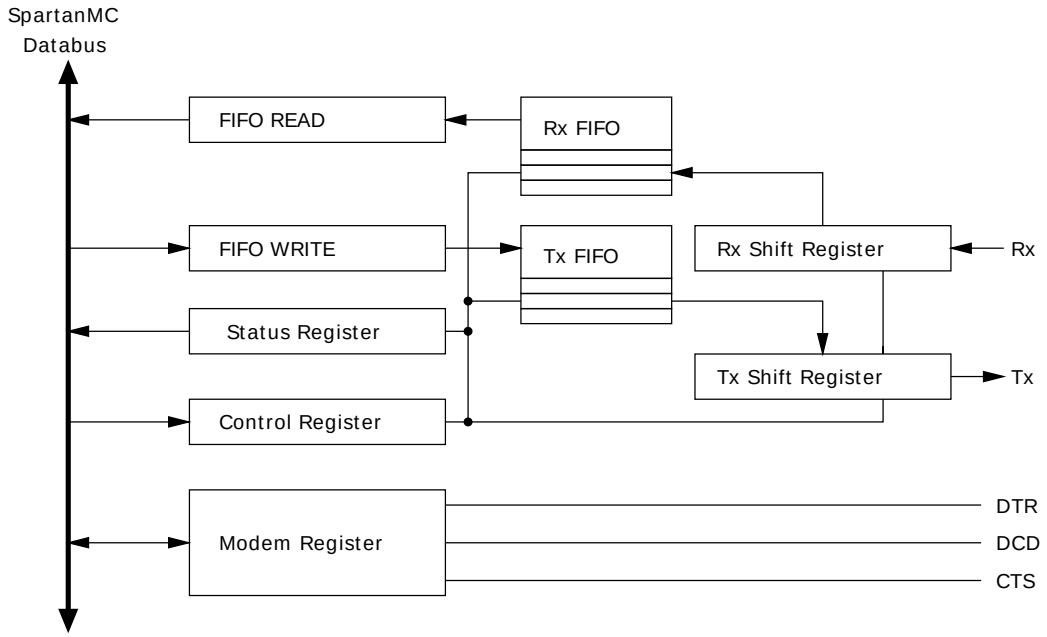


Figure 5-27: UART block diagram

5.1. Framing

Each frame starts with a logic low start bit followed by a configurable number of data bits (5-8), an optional parity bit and one or more logic hight stop bits. The parity bit can be defined as odd or even. If the parity is set to "even", the hamming weight of all data bits including the partiy bit have to be even for a valid frame. Contrariwise, the hamming weight of all data bits and the parity bit have to be odd if the parity bit is set to "odd". The transmission of the data field starts with the least significant bit (LSB).

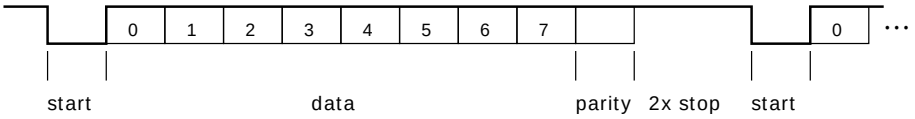


Figure 5-28: UART frame example

5.2. Module parameters

Table 5-9: UART module parameters

Parameter	Default Value	Descripton
BASE_ADR	0x10	Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
FIFO_RX_DEPTH	8	Number of 8 bit cells of the receiver FIFO memory for incoming data.
FIFO_TX_DEPTH	8	Number of 8 bit cells of the sender FIFO memory for outgoing data.
MODEM	0	Allows the usage of the UART peripheral control signals. If MODEM is set to one the control signals RTS, DCD and DTR are active. If MODEM is set to zero RTS, DTR and DTR are set to logic high.
CLKIN_FREQ	25000000	The clock frequency in Hz which the module is currently driven by. This frequency is used to calculate the prescalers for the different baud rates available in UART_CTRL.

5.3. Interrupts

For interrupt driven serial communication the UART provides two interrupt signals which will be generated at the following conditions:

- TX interrupt is set for each sent character. The interrupt remains one until the next read access to UART_CTRL (Register Nr. 3).
- RX interrupt is set for each received character. The interrupt remains one until the next read acces to UART_FIFO_READ (Register Nr. 1).

Note: In case the software application uses these interrupts, the SpartanMC SoC requires an interrupt controller which provides an appropriate interface for the interrupt signals.

5.4. Peripheral Registers

5.4.1. UART Register Description

The UART peripheral provides four 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 5-10: UART registers

Offset	Name	Access	Description
0	UART_STATUS	read	Contains the current Rx/Tx FIFO status, current framing/parity errors, the modem signals if available and the the busy signals for the Rx/Tx shift registers.
1	UART_FIFO_READ	read	Register for incoming data. The LSB of the data word is written to $\texttt{UART\_FIFO\_READ}_0$.
2	UART_FIFO_WRITE	write	Register for outgoing data. The LSB of the data word is written $\texttt{UART\_FIFO\_WRITE}_0$.
3	UART_CTRL	read/ write	Contains the current UART setting e.g. baud rate, data field length and parity and interrupt settings.
4	UART_MODEM	read/ write	Contains the input/output setting for the modem signals.

5.4.2. UART_Status Register

Table 5-11: UART status register layout

Bit	Name	Access	Default	Description
0	RX_EMPTY	read	1	Set to one if the Rx FIFO is empty otherwise the value is zero.
1	RX_FULL	read	0	Set to one if the Rx FIFO is full otherwise the value is zero.
2	TX_EMPTY	read	1	Set to one if the Tx FIFO is empty otherwise the value is zero.
3	TX_FULL	read	0	Set to one if the Tx FIFO is full otherwise the value is zero.
4	TX_IRQ_PRE	read	1	Set to one while data were witten to the Tx FIFO. The value is reset to zero through the next TX access to the Tx FIFO memory.
5	TX_IRQ_FLAG	read	0	Set to one while data were sent. The value is reset to zero through the next write access to TX_IRQ_FLAG.
6	RX_P_ERR	read	0	Set to one until the next frame if a parity error occurs.
7	RX_F_ERR	read	0	Set to one until the next frame if a frameing error (break character or high impetance input) occurs.
8	RX_D_ERR	read	0	Set to one if the Rx FIFO is full while receiving a frame. The value remains one until the status of the Rx FIFO is not full.
9	M_DCD	read	0	The current value of the modem signal DCD if the parameter MODEM is set to one.
10	M_CTS	read	0	The current value of the modem signal CTS if the parameter MODEM is set to one.
11	M_DSR	read	0	The current value of the modem signal DSR if the parameter MODEM is set to one.
12	RX_CLK	read	0	Current clock of the Rx shift register. (For debugging purposes)
13	RX_STOP	read	0	It is set to one while the Rx shift register is not busy.
14	TX_CLK	read	0	Current clock of the Tx shift register. (For debugging purposes)
15	RST_UART	read	0	Is set to one during a UART reset. Typically, the reset is started with the SpartanMC processor core reset signal. The reset remains one for one UART bit period. This is equivalent to 8.6 us at 115200 Baud.
16	TX_STOP	read	0	It is set to one while the Tx shift register is not busy.
17	x	read	0	Not used.

Table 5-11: UART status register layout

5.4.3. UART_FIFO_READ Register

Table 5-12: UART status register layout

Bit	Name	Access	Default	Description
0-7	RX	read	x	Register for received data.
8-17	x	read	0	Not used.

5.4.4. UART_FIFO_WRITE Register

Table 5-13: UART status register layout

Bit	Name	Access	Default	Description
0-7	TX	write	x	Register for data to send.
8-17	x	write	x	Not used.

5.4.5. UART_CTRL Register

Note: Befor writing UART_CTRL it has to be assured that RST_UART are set to zero and RX_EMPTY, TX_EMPTY, RX_STOP and TX_STOP are set to one.

Table 5-14: UART control register layout

Bit	Name	Access	Default	Description
0	RX_EN	write	0	Turns the Rx shift register off if set to zero.
1	TX_EN	write	0	Turns the Tx shift register off if set to zero.
2	PARI_EN	write	0	If set to one the usage of parity bits in frames will be enabled.
3	PARI_EVEN	write	0	If set to one the parity is even otherwise the parity is odd.
4	TWO_STOP	write	0	Enables the usage of a second stop bit.
5-7	DATA_LEN	write	000	Sets the length of the data field : 111 = 8 bit data 110 = 7 bit data 101 = 6 bit data 100 = 5 bit data
8	x	write	x	Not used.
9-12	BPS	write	0000	Sets the baud rate: 0000 = 115200 Baud 0001 = 57600 Baud 0010 = 38400 Baud 0011 = 31250 Baud (MIDI data rate) 0100 = 19200 Baud 0101 = 9600 Baud 0110 = 4800 Baud 0111 = 2400 Baud 1000 = 1200 Baud 1001 = 600 Baud 1010 = 300 Baud 1011 = 150 Baud 1100 = 75 Baud 1101 = 50 Baud All other values are mapped to 7812.5 Baud.
13	RX_IE	write	0	If set to one the Rx interrupt will be enabled.

Bit	Name	Access	Default	Description
14	TX_IE	write	0	If set to one the Tx interrupt will be enabled.
15	TX_BREAK	write	0	If set to one the UART will sent a break signal. (Tx logic low) The duration of the break signal must be longer than a complete frame (start, data, stop and parity). To identify breaks the framing error detection can be used.
16-17	x	write	x	Not used.

Table 5-14: UART control register layout

5.4.6. UART_MODEM Register

The UART peripheral provides three control lines for hardware handshaking and flow control: Data Carrier Detect (DCD), Data Set Ready (DSR)/Data Terminal Ready (DTR) and Request To Send (RTS)/Clear TO Send (CTS). The signal name and the i/o-direction depends on the RS-232 device class - Data Terminal Equipment (DTE) or Data Communication Equipment (DCE). For more information cf. the EIA-232 or RS-232 standard.

Table 5-15: UART modem register layout

Bit	Name	Access	Default	Description
0	DTR_DSR	read/ write	0	If DTR is used as output DTR tells DCE that DTE is ready to be connected. If DTR is used as input DSR tells DTE that DCE is ready to receive commands or data.
1	RTS_CTS	write	0	If RTS is used as input RTS tells DCE to prepare to accept data from DTE. If RTS is used as output CTS acknowledges RTS and allows DTE to transmit.
2	DCD	write	0	Tells the DTE that DCE is connected to the carrier line.
3	DCD_OUT	write	0	If set to one DCD works as output which indicates the peripheral is used as DCE. If set to zero DCD works as input and the peripheral is used as DTE.
4	RTS_OUT	write	0	If set to one RTS works as output which indicates the peripheral is used as DTE. If set to zero RTS works as input and the peripheral is used as DCE.
5	DTR_OUT	write	0	If set to one DTR works as output which indicates the peripheral is used as DTE. If set to zero DTR works as input and the peripheral is used as DCE.
6-17	x	write	x	Not used.

Table 5-15: UART modem register layout

5.4.7. UART C-Header for Register Description

```
#ifndef __UART_H
#define __UART_H

#include ‹stdint.h›

// Status Signale
#define UART_RX_EMPTY (1 « 0)
#define UART_RX_FULL (1 « 1)
#define UART_TX_EMPTY (1 « 2)
#define UART_TX_FULL (1 « 3)
#define UART_TX_IRQ_PRE (1 « 4)
#define UART_TX_IRQ_FLAG (1 « 5)
#define UART_RX_P_ERR (1 « 6)
#define UART_RX_F_ERR (1 « 7)
#define UART_RX_D_ERR (1 « 8)
#define UART_M_DCD (1 « 9)
#define UART_M_CTS (1 « 10)
#define UART_M_DSR (1 « 11)
#define UART_RX_CLK (1 « 12)
#define UART_RX_STOP (1 « 13)
#define UART_TX_CLK (1 « 14)
#define UART_RST_UART (1 « 15) // UART noch im RESET, wenn = 1
#define UART_TX_STOP (1 « 16)

// Steuersignale
#define UART_RX_EN (1 « 0)
#define UART_TX_EN (1 « 1)
#define UART_PARI_EN (1 « 2)
#define UART_PARI_EVEN (1 « 3) // 0 = ungerade / 1 = gerade
#define UART_TWO_STOP (1 « 4) // 0 = ein / 1 = zwei Stopbits

#define UART_DATA_LEN_5 (4 « 5) // 0x00080
#define UART_DATA_LEN_6 (5 « 5) // 0x000A0
#define UART_DATA_LEN_7 (6 « 5) // 0x000C0
#define UART_DATA_LEN_8 (7 « 5) // 0x000E0

#define UART_BPS_115200 (0 « 9) // 0x00000
#define UART_BPS_57600 (1 « 9) // 0x00200
#define UART_BPS_38400 (2 « 9) // 0x00400
#define UART_BPS_31250 (3 « 9) // 0x00600 MIDI Datenrate
#define UART_BPS_19200 (4 « 9) // 0x00800
#define UART_BPS_9600 (5 « 9) // 0x00A00
#define UART_BPS_4800 (6 « 9) // 0x00C00
#define UART_BPS_2400 (7 « 9) // 0x00E00
#define UART_BPS_1200 (8 « 9) // 0x01000
```

```
#define UART_BPS_600 (9 « 9) // 0x01200
#define UART_BPS_300 (10 « 9) // 0x01400
#define UART_BPS_150 (11 « 9) // 0x01600
#define UART_BPS_75 (12 « 9) // 0x01800
#define UART_BPS_50 (13 « 9) // 0x01A00
#define UART_BPS_7812 (14 « 9) // 0x01C00 Boot 68hc11
       // Boot 68hc11 mit 7812,5 Baud

#define UART_RX_IE (1 « 13)
#define UART_TX_IE (1 « 14)
#define UART_TX_BREAK (1 « 15)

// Modem Outputs und Richtung (optional)
#define UART_DTR_DSR (1 « 0)
#define UART_RTS_CTS (1 « 1)
#define UART_DCD (1 « 2)
#define UART_DCD_OUT (1 « 3)
#define UART_RTS_OUT (1 « 4)
#define UART_DTR_OUT (1 « 5)

// Rueckgabewerte fuer non blocking read
#define UART_OK 0
#define UART_NO_DATA 1

typedef struct {
   volatile uint18_t status;   // read
   volatile uint18_t rx_data;   // read (Reset Rx Interrupt)
   volatile uint18_t tx_data;   // write
   volatile uint18_t ctrl_stat;   // write (or read)
           // write (or read = status and Reset Tx Interrupt)
   volatile uint18_t modem;   // write (optional)
} uart_regs_t;

#endif
```

6. Simple Universal Asynchronous Receiver Transmitter (UART Light)

The UART Light is a SpartanMC peripheral device for serial communication with external systems. Compared to the standard UART the UART Light uses a lightwight interface which enables a smaller resource footprint. To provide the minimum interface the peripheral is primarily configured via pre synthesis parameters.

The UART Light bitstream is fixed to 8 databits per frame, the datarate and FIFO buffer depth is configurable via module paramater. The signals Rx and Tx are used as connection to the external system environment. The incoming and outgoing data is written to FIFO memory modules which are connected to the input and output shift registers of the UART.

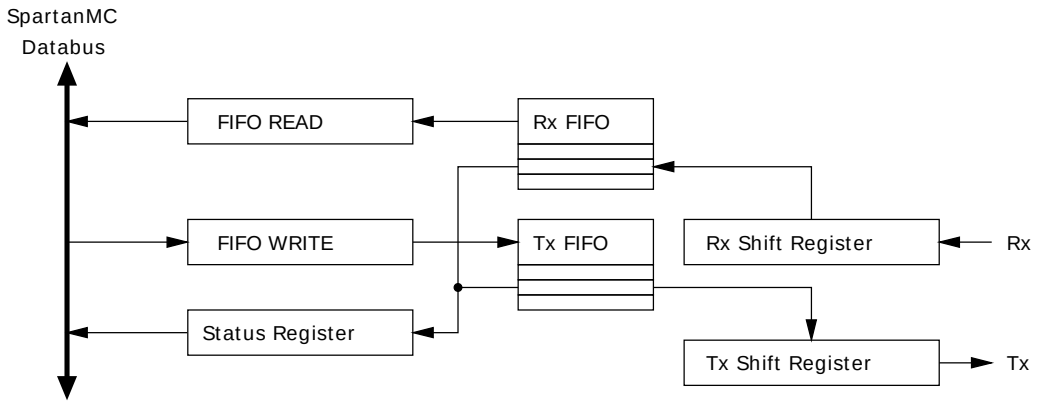


Figure 6-29: UART Light block diagram

6.1. Framing

Each frame starts with a logic low start bit followed by a fixed number of 8 databits. Parity bits and additional stop bits are not supported. The transmission of the data field starts with the least significant bit (LSB).

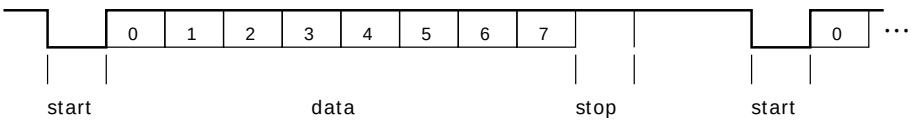


Figure 6-30: UART Light frame

6.2. Module parameters

Table 6-16: UART module parameters

Parameter	Default Value	Descripton
BASE_ADR	0x10	Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
FIFO_RX_DEPTH	8	Number of 8 bit cells for incoming data of the receiver FIFO memory. (maximum 2048)
FIFO_TX_DEPTH	8	Number of 8 bit cells for outgoing data of the sender FIFO memory. (maximum 2048)
BAUDRATE	115200	Defines the required baud rate: 921600 Baud 576000 Baud 460800 Baud 230400 Baud 115200 Baud 57600 Baud 28000 Baud 14400 Baud 7200 Baud 4800 Baud 2400 Baud 1200 Baud 600 Baud 300 Baud
INTERRUPT_SUPPORTED	FALSE	If this parameter is set to FALSE, the content of the FIFO memories must be determined via polling. If this parameter is set to TRUE, each received or sent frame can generates an interrupt.
CLOCK_FREQUENCY	25000000	The clock frequency in Hz which the module is currently driven by. This frequency is used to calculate the prescalers for the configured baudrate.

6.3. Interrupts

In case the UART Light is synthezied for interrupt mode it provides two interrupt signals. Both interrupt signals must be enabled in the UART STATUS register. The interrupts will be generated at the following conditions:

- The Tx interrupt (intr_tx) is set to one if the Tx FIFO status not full and write in last time to Tx-Data. The interrupt indicates that one data in the Tx FIFO memory were sent.
- The Rx interrupt (intr_rx) is set if the first frame is written to the Rx FIFO memory.

The Tx interrupt reset is perfomed by reading the UART_STATUS register. The Rx interrupt reset is perfomed by reading data from the Rx FIFO memory.

Note: In case the software application uses this interrupt, the SpartanMC SoC requires an interrupt controller which provides an appropriate inferface for the interrupt signal.

6.4. Peripheral Registers

6.4.1. UART Register Description

The UART peripheral provides three 18 bit registers which are mapped to the Spartan-MC address space located at 0x1A000 + BASE_ADR + Offset.

Table 6-17: UART registers

Offset	Name	Access	Description
0	UART_STATUS	read/ (write)	Contains the current Rx/Tx FIFO status. For resetting the tx interrupt this register should be write.
1	UART_FIFO_READ	read	Register for incoming data. The LSB of the data word is written to $UART\_FIFO\_READ_0$. For resetting the rx interrupt this register should be read.
2	UART_FIFO_WRITE	write	Register for outgoing data. The LSB of the data word is written $UART\_FIFO\_WRITE_0$.

6.4.2. UART_STATUS Register

Table 6-18: UART status register layout

Bit	Name	Access	Default	Description
0	RX_EMPTY	read	1	Set to one if the Rx FIFO is empty otherwise the value is zero.
1	RX_FULL	read	0	Set to one if the Rx FIFO is full otherwise the value is zero.

Bit	Name	Access	Default	Description
2	TX_EMPTY	read	1	Set to one if the Tx FIFO is empty otherwise the value is zero.
3	TX_FULL	read	0	Set to one if the Tx FIFO is full otherwise the value is zero.
4	TX_IRQ_PRE	read	1	Set to one while data were witten to the Tx FIFO. The value is reset to zero through the next TX access to the Tx FIFO memory.
5	TX_IRQ_FLAG	read	0	Set to one while data were sent. The value is reset to zero through the next write access to TX_IRQ_FLAG.
6-8	x	read	0	Not used.
9	RX_IRQ_ENABLE	read/ write	0/0	To enable rx interrupt set this bit should be set to otherwise set to zero. This bit is only in not polling mode available.
10	TX_IRQ_ENABLE	read/ write	0/0	To enable tx interrupt this bit should be set to one otherwise set to zero. This bit is only in not polling mode available.
11-17	x	read	0	Not used.

Table 6-18: UART status register layout

6.4.3. UART_FIFO_READ Register

Table 6-19: UART status register layout

Bit	Name	Access	Default	Description
0-7	RX	read	0	Register for received data.
8-17	x	read	0	Not used.

6.4.4. UART_FIFO_WRITE Register

Table 6-20: UART status register layout

Bit	Name	Access	Default	Description
0-7	TX	write	0	Register for data to send.
8-17	x	write	x	Not used.

6.4.5. UART C-Header for Register Description

```
#ifndef UART_LIGHT_H_
#define UART_LIGHT_H_

#include ‹stdint.h›

// Status Signale
#define UART_LIGHT_RX_EMPTY (1 « 0)
#define UART_LIGHT_RX_FULL (1 « 1)
#define UART_LIGHT_TX_EMPTY (1 « 2)
#define UART_LIGHT_TX_FULL (1 « 3)
#define UART_LIGHT_TX_IRQ_PRE (1 « 4)
#define UART_LIGHT_TX_IRQ_FLAG (1 « 5)

// Steuersignale
#define UART_LIGHT_RX_IE (1 « 13)
#define UART_LIGHT_TX_IE (1 « 14)

// Rueckgabewerte fuer non blocking read
#define UART_LIGHT_OK 0
#define UART_LIGHT_NO_DATA 1

typedef struct {
   volatile uint18_t status; // r/w w = Reset Tx Interrupt
   volatile uint18_t rx_data; // r r = Reset Rx Interrupt
   volatile uint18_t tx_data; // w
} uart_light_regs_t;

void uart_light_send (uart_light_regs_t *uart, unsigned char
value);
unsigned char uart_light_receive (uart_light_regs_t *uart);
int uart_light_receive_nb (uart_light_regs_t *uart, unsigned
char *value);

#endif /*UART_LIGHT_H_*/
```

7. Serial Peripheral Interface Bus (SPI)

The SPI is a SpartanMC peripheral device for serial communication using the SPI-bus. The SPI enables a bit stream of 8 bits to be shifted in and out of the component at programmable speed. The peripheral can be connected with up to 15 SPI-slaves which shares 3 Wires: SCK (serial clock), MOSI (master out slave in) and MISO (master in slave out). Each slave requires one slave-select-signal to activate the slave and connect it to the other wires.

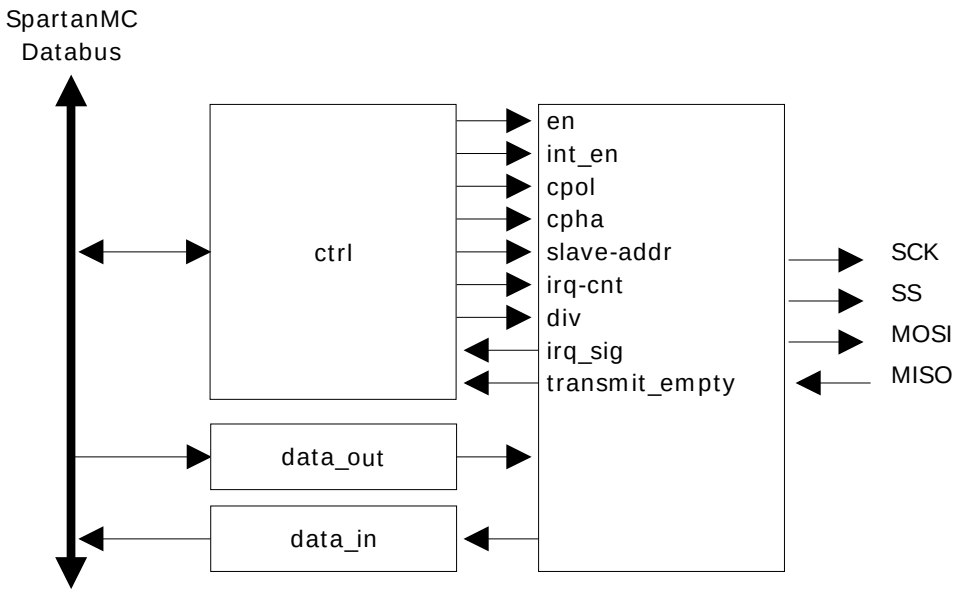


Figure 7-31: SPI block diagram

7.1. Communication

To start a transfer to a slave, the master has to clear the select signal for this slave. After this the SPI-Master can generate the clock signal and shift data to the slave. During each SPI clock cycle, one bit is sent to the slave and one bit is received from the slave. The polarity of the clock is controlled by the CPOL bit in the control register. The phase of the data is controlled by a bit called CPHA.

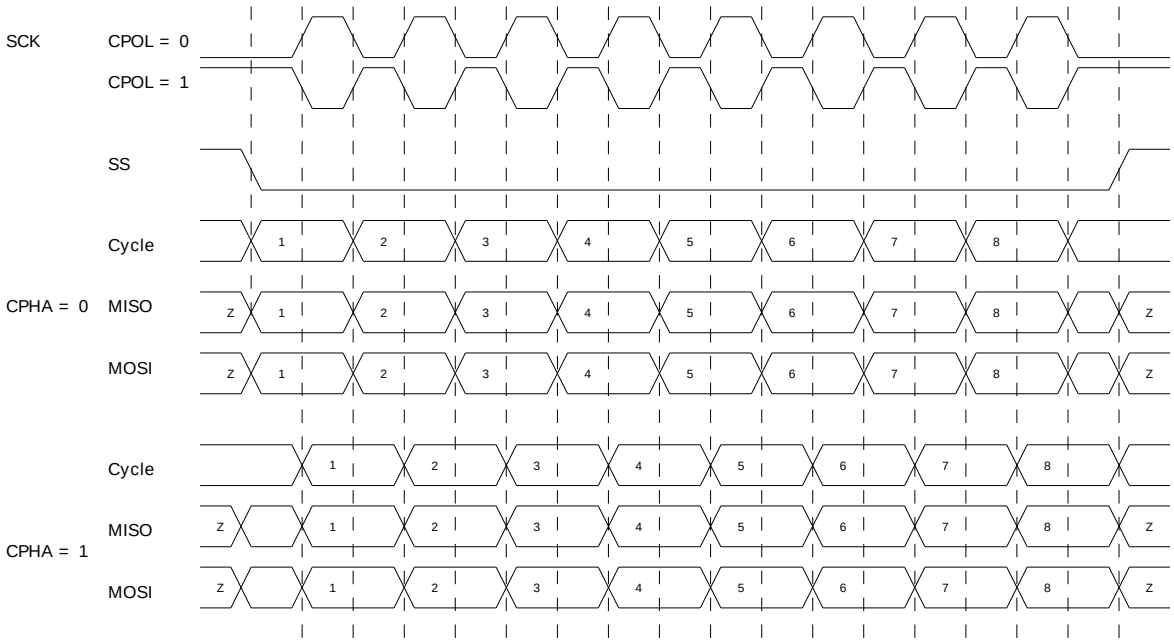


Figure 7-32: SPI frame

7.2. Module parameters

Table 7-21: SPI module parameters

Parameter	Default Value	Descripton
SPI_SS	0x1	Number of generated slave-select-signals

7.3. Peripheral Registers

7.3.1. SPI Register Description

The SPI peripheral provides three 18 bit registers which are mapped to the SpartanMC address space.

Table 7-22: SPI registers

Offset	Name	Access	Description
0	spi_control	read/ write	Contains the current SPI setting e.g. clock divider, CPOL, CPHA interrupt settings and status.
1	spi_data_out	read/ write	Register for outgoing data.
2	spi_data_in	read	Register for incoming data.

7.3.2. SPI Control Register

Table 7-23: SPI control register layout

Bit	Name	Access	Def	Description
0	EN	r/w	0	Enable the Controller.
1	IRQ_EN	r/w	0	Enable sending IRQs.
2	CPOL	r/w	0	The base value of the clock (at CPOL=0 the base value is zero).
3	CPHA	r/w	0	A value of CPHA=0 means sample on the leading (first) clock edge, while CPHA=1 means sample on the trailing (second) clock edge, regardless of whether that clock edge is rising or falling.
4-7	Slave	r/w	0	Decimal number of the slave to communicate with. 0 = 0000 = deactivates all slave-select-signals 1 = 0001 = activates the first slave-select-signal 2 = 0010 = activates the second slave-select-signal .... 15 = 1111 = activates the 15th slave-select-signal
8	transmit empty	r	0	If set to one the transmit register is empty.
9	IRQ_Sig	r	0	Interrupt-Signal goes to one when finishing a transmission. It will be cleared on a read access to the data_in register.
10-12	Bitcnt	r/w	000	Field named bitcount. 0 = 000 = 8 Bit Word

Bit	Name	Access	Def	Description
				1 = 001 = 7 Bit Word 2 = 010 = 6 Bit Word .... 7 = 111 = 1 Bit Word
13-15	clk_div	r/w	111	Sets the clock divider. Output clk = (spi_clk) / (4 * devider_value) 0 = 000 = 2^0 1 = 001 = 2^1 2 = 010 = 2^2 .... 7 = 111 = 2^7
16	ss_on	r	0	0 = all slave-select-signals deactivate
17	ss_set	r	1	0 = slave-select-signals not ready

Table 7-23: SPI control register layout

7.3.3. SPI C-Header spi.h for Register Description

```
#ifndef __SPI_H
#define __SPI_H

#include <peripherals/spi_master.h>
#include <peripherals/spi_slave.h>

// the following lines is kept for compatibility with older
// projects and may be removed in the future

// master and slave regs are identical, so we just pick one to
// emulate the old spi_t type
typedef spi_slave_regs_t spi_t;

// end compatibility section

#include <bitmagic.h>

unsigned char spi_readwrite(spi_t *spi, unsigned char data);
void spi_write(spi_t *spi, unsigned char data);
void spi_activate(spi_t *spi,unsigned int device);
void spi_deactivate(spi_t *spi);
void spi_enable(spi_t *spi);
void spi_disable(spi_t *spi);
void spi_enable_irq(spi_t *spi);
void spi_disable_irq(spi_t *spi);
```

```
void spi_set_spol(spi_t *spi, unsigned int cpol);
void spi_set_cpah(spi_t *spi, unsigned int cpah);
void spi_set_bitcnt(spi_t *spi, unsigned int bitcnt);
void spi_set_div(spi_t *spi, unsigned int div);

#endif
```

7.3.4. SPI C-Header spi_master.h for Register Description

```
#ifndef __SPI_MASTER_H
#define __SPI_MASTER_H

#define SPI_MASTER_CTRL_EN           0x00001
#define SPI_MASTER_CTRL_INT_EN       0x00002
#define SPI_MASTER_CTRL_CPOL       0x00004
#define SPI_MASTER_CTRL_CPHA       0x00008
#define SPI_MASTER_CTRL_SLAVE       0x000F0
#define SPI_MASTER_CTRL_TRANS_EMPTY 0x00100
#define SPI_MASTER_CTRL_INT           0x00200
#define SPI_MASTER_CTRL_BITCNT       0x01C00
#define SPI_MASTER_CTRL_DIV           0x0E000
#define SPI_MASTER_CTRL_SS_ON       0x10000
#define SPI_MASTER_CTRL_SS_SET       0x20000

typedef volatile struct {
   volatile unsigned int spi_control;   // (r/w)
   volatile unsigned int spi_data_out; // (r/w) (reset-int)
   volatile unsigned int spi_data_in;   // (r) (reset-int)
} spi_master_regs_t;

#endif
```

7.3.5. SPI C-Header spi_slave.h for Register Description

```
#ifndef __SPI_SLAVE_H
#define __SPI_SLAVE_H

#define SPI_SLAVE_CTRL_EN       0x00001
#define SPI_SLAVE_CTRL_INT_EN   0x00002
#define SPI_SLAVE_CTRL_CPOL   0x00004
#define SPI_SLAVE_CTRL_CPHA   0x00008
#define SPI_SLAVE_CTRL_DONE   0x00100
#define SPI_SLAVE_CTRL_INT       0x00200
#define SPI_SLAVE_CTRL_BITCNT   0x01C00
```

```
typedef volatile struct {
   volatile unsigned int spi_control;   // (r/w)
   volatile unsigned int spi_data_out; // (r/w) (reset-int)
   volatile unsigned int spi_data_in;   // (r) (reset-int)
} spi_slave_regs_t;

#endif
```

7.3.6. SPI Sample Application

This sample application reads the Circuit-ID from an M25P32 Flash EPROM via SPI. The application was implemented on an Xilinx ML507 evaluation board.

```
#include <system/peripherals.h>
#include <uart.h>
#include <stdio.h>
#include <spi.h>
#include "m25p32.h"


void main() {
   unsigned int i;
   stdio_uart_open(UART_LIGHT_0);
   printf("\r\nHello SPI_Sample:");

   printf(\r\nEnable the SPI-Core:");
   SET(SPI_MASTER_0->spi_control, SPI_MASTER_CTRL_EN);

   printf("\r\nPower-Up the connected SPI-Flash:");
   spi_activate(SPI_MASTER_0,1);
   spi_readwrite(SPI_MASTER_0,0xAB);
   spi_deactivate(SPI_MASTER_0);

   printf("\r\nRead ID of the SPI-Flash:\r\n");
   unsigned int id[4];
   m25p32_read_id(SPI_MASTER_0, &id[0]);

   for(i=0;i<3;i++) {
       printf("ID %u : %x\r\n",i,id[i]);
   }
   UNSET(SPI_MASTER_0->spi_control, SPI_MASTER_CTRL_EN);
   while(1);
}

void m25p32_read_id(spi_t* spi, unsigned int* data) {
   unsigned int i;
   spi_activate(spi,1);
```

```
   spi_readwrite(spi,M25P32_RDID);
   for (i = 0; i< 3; i++){
       data[i] = spi_readwrite(spi,0);
   }
   spi_deactivate(spi);
}
```

The output is send via serial connection to a host PC. Therefore an UART peripheral is required in the SoC. ID 0 = 020 specify the menufacturer type (ST), ID 1 = 020 specify device type and ID 2 = 016 indicates the memory capacity.

```
SpMC loader v20120927

Hello SPI_Sample:
Enable the SPI-Core:
Power-Up the connected SPI-Flash:
Read ID of the SPI-Flash:
ID 0 : 20
ID 1 : 20
ID 2 : 16
```

8. I2C Master

The SpartanMC I2C master controller provides a logical connection to a I2C bus. To ensure a proper physical connection it requires pull up resistors on both signal lines (s. physical connection section).

I2C (Inter-Integrated Circuit) also referred to as "Two-Wire Interface" is a single-ended multi-master bus which is used to attach low-speed peripherals to other electronic devices. Originally, the bus system was invented by Philips Semiconductur in 1982. Today, the implementation of the I2C protocol itself is free, but the global assignment of I2C Slave addresses by NXP still requires a fee. A detailed description of the I2C protocol and its license condition could be found on "http://www.nxp.com" or "http://www.nxp.com/documents/user_manual/UM10204.pdf".

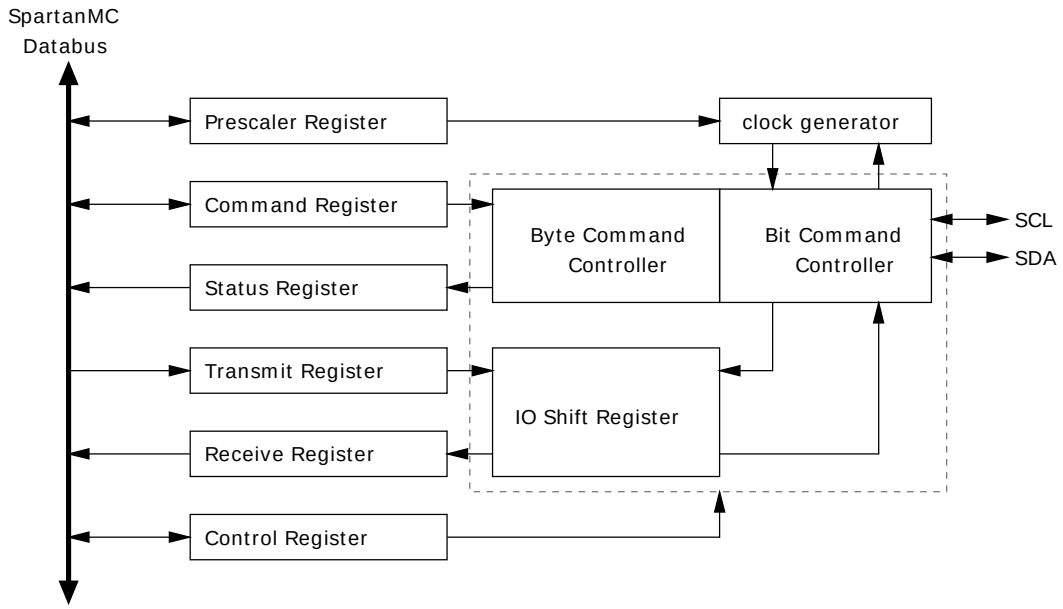


Figure 8-33: I2C block diagram

8.1. Physical Connection

The I2C bus requires two signal lines: Seriel Clock Line (SCL) and Serial Data Line (SDA). SDA is used for data transmission while the clock signal SCL is required to synchronize master and slave. The clock signal is always generated by the I2C master. Both lines are connected to VCC via pull up resistors. Thus, a logical zero is generated be pulling the line to ground while a logical one is generated by letting the line float.

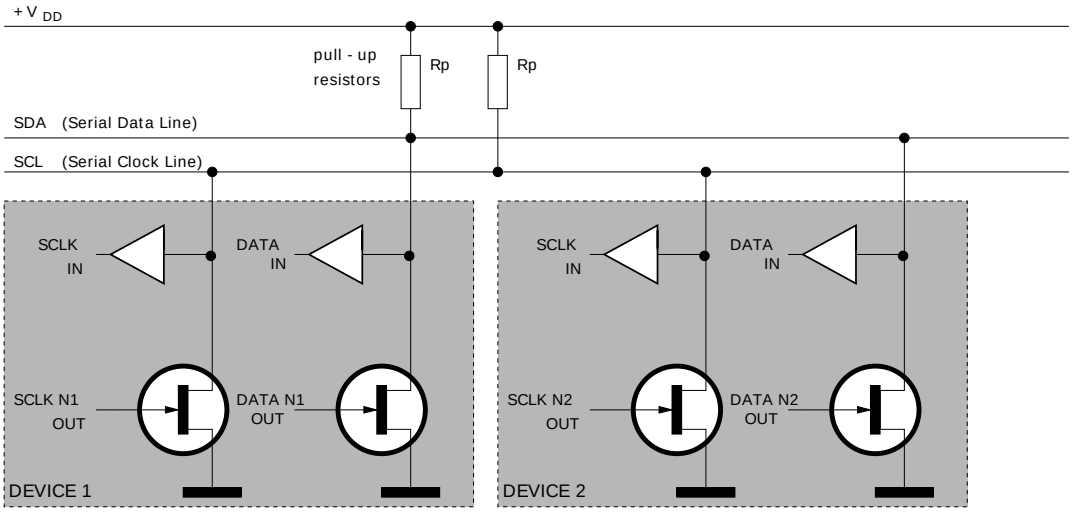


Figure 8-34: I2C Pyhsical Connection

8.2. Bus Arbitration

The physical connection mechanism enables a dominant bus level (logical zero) and rezessive bus level (logical one). This could be used for bus arbitration. If different nodes try to drive SDA simultaniously, the node which sends the first zero dominates the bus. The same technique can be used on SCL to give the slave nodes a flow control mechanism. Thus, the data transfer to the master will be delayed when pulling the clock line to zero.

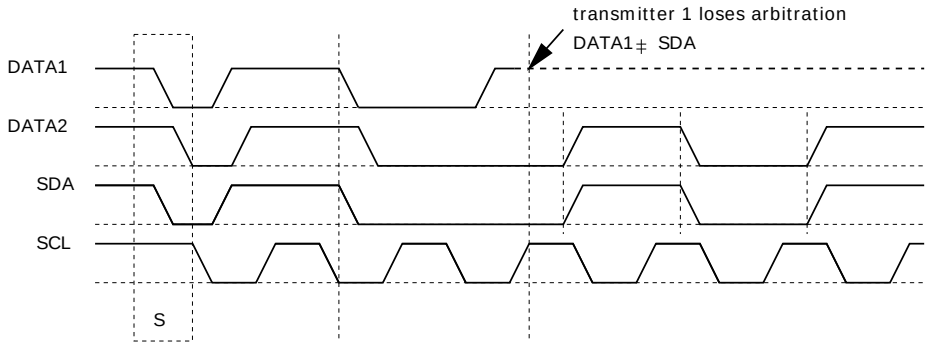


Figure 8-35: I2C Arbitration

The I2C controller will lost the arbitration when the controller drives SDA high, but the received SDA is low or the controller detects a stop sequence, but non requested one.

8.3. Framing

To start a transmission, SDA is pulled low while SCL remains high. Each following bit, after this start sequenz, is transmitted with a raising edge of SCL. After transmission SDA must be released to float high again which is used as stop bit and idle marker. Except for the start and stop marker, the SDA line only changes while SCL is low.

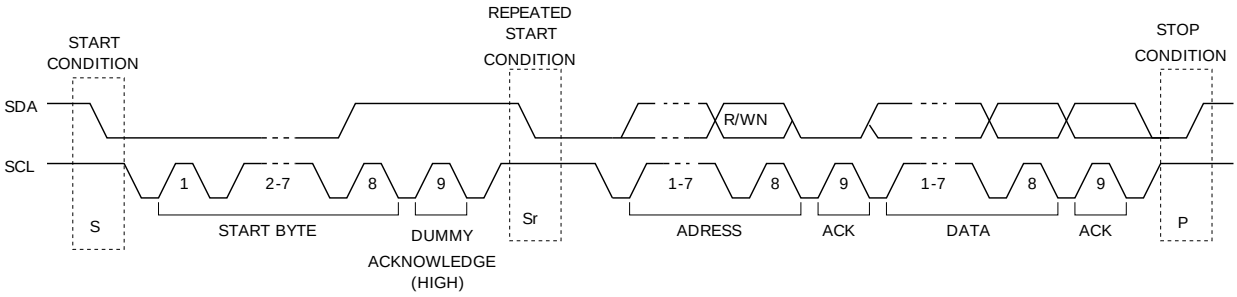


Figure 8-36: SCL, SDA Timing for the First Byte

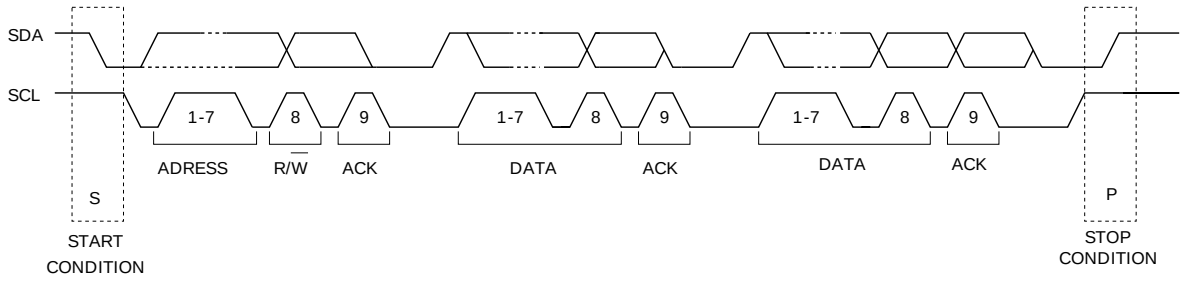


Figure 8-37: SCL, SDA Timing for Data Transmission

The data transfer is always initilzed by the master. After the start signal the master sends the first byte of the slave address (1 byte for a 7 bit address - 2 byte for a 10 bit address) and a direction bit (1 for read from slave, 0 for write to slave). During data transmission each data byte must be acknowledged throuth the master or slave by pulling SDA to zero.

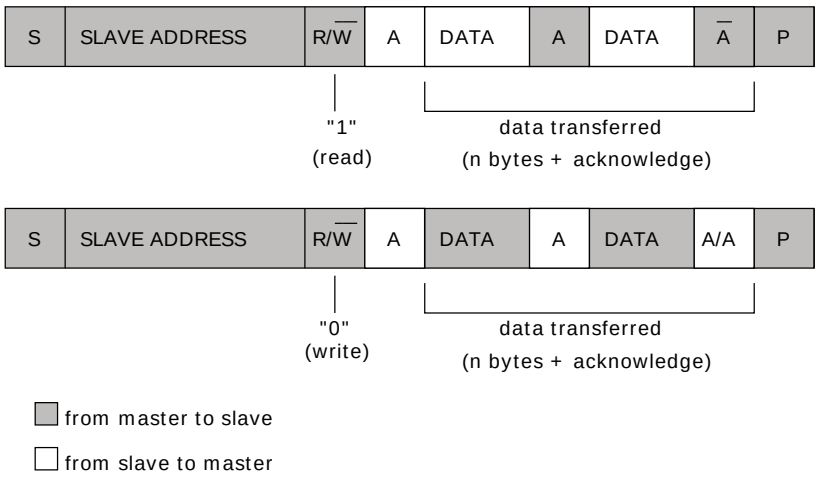


Figure 8-38: I2C Receive Date from Save and Send Data to Slave

8.4. Modul parameters

The I2C controller enables a variable registers address offset. Changing the offset to the peripheral base address could be done via following synthesis parameters.

Table 8-24: I2C modul parameters

Parameter	Default Value	Descripton
CONTROL	0	Address offset for the control register.
TX	1	Address offset for the tx register.
RX	2	Address offset for the rx register.
COMMAND	3	Address offset for the command register.
STATUS	4	Address offset for the status register.

8.5. Peripheral Registers

8.5.1. I2C Register Description

The I2C peripheral provides six 18 bit registers which are mapped to the SpartanMC address space. The address offset for the registers refers to to given synthesis parameter (s. module parameter section)

Table 8-25: I2C registers

Offset	Name	Access	Description
0	CONTROL	read/write	Contains a 16 bit clock divider for SCL gerenration and is used to set and reset the control bits of the I2C controller.
1	TX	write	Contains the current byte to send.
2	RX	read	Contains the current recieved byte.
3	COMMAND	write	Used to set I2C commandos.
4	STATUS	read	Contains the controller status flags.

8.5.2. CONTROL Register

Table 8-26: I2C control register layout

Bit	Name	Access	Default	Description
0-15	PRESCALER	read/ write	65535	This register is used to prescale the clock frequency of the SCL line. To change the prescale register the CORE_EN bit must be set to zero. The prescale factor can be dermined through following equation: prescale = (peripheral_clock / (5 * desired_SCL)) -1.
16	CORE_EN	read/ write	0	Enable I2C core. If set to 1 the I2C core is enabled. (The presacler value remains constant.)
17	IEN	read/ write	0	Interrupt enable. If set to 1 the interrupt is enabled.

8.5.3. TX Register

Table 8-27: I2C transmit data register layout

Bit	Name	Access	Default	Description
0-7	TX	write	0	Register for data to send. The LSB (Bit 0) of this register represets the read/write bit (1 for read from slave, 0 for write to slave).
8-17	x	write	0	Not used.

8.5.4. RX Register

Table 8-28: I2C receive data register layout

Bit	Name	Access	Default	Description
0-7	RX	read	0	Register for received data.
8-17	x	read	0	Not used. (Read as zero)

8.5.5. COMMAND Register

Table 8-29: I2C command register layout

Bit	Name	Access	Default	Description
0	IACK	read/ write	0	Interrupt achknowledge. If set to one the pending interrupt will be cleared.
1-2	x	read/ write	0	Not used.
3	ACK	read/ write	0	If set to 0 the acknowledgement of a received frame will be carried out (ACK bit in I2C frame = 0). When set to 1 the I2C frame will be not acknowledged (ACK bit in frame = 1).
4	WR	read/ write	0	If set to 1 the TX register will be written to slave.
5	RD	read/ write	0	If set to 1 the RX register will be filled with data from slave.
6	STO	read/ write	0	Sends stop sequence.
7	STA	read/ write	0	Sends (re-)start sequence.
8-17	x	read/ write	0	Not used.

Note: In case the values for WR and RD are set to 1 simultaneously the controller will set the read bit within the I2C frame.

8.5.6. STATUS Register

Table 8-30: I2C status register layout

Bit	Name	Access	Default	Description
0	IF	read	0	This bit is set to 1 when an interrupt is pending and IEN in I2C control register is set. Interrupts are set on the following conditions: • A byte transfer has been completed. • The arbitration is lost.
1	TIP	read	0	Is set to 1 when an I2C data transfer is in progress.
2-4	x	read	0	Not used.
5	AL	read	0	Is set to 1 after an arbitration lost. See arbitration section for a detailed description of arbitration lost conditions.
6	I2C_BUSY	read	0	Is set to 1 after a frame start sequence is detected and set to 0 when the stop sequence occures.
7	RX_ACK	read	0	Is set to 1 if the acknowledgement from slave failed.
8-17	x	read	0	Not used.

9. JTAG-Controller

The JTAG-Controller for the SpartanMC is a JTAG-Master. It can communicate with JTAG-Slaves, which are connected through the 4 JTAG-Pins described in the following Table. If you need a TRST-Port you can use a PortOut Component of the SpartanMC. This component implements an extra feature for MSP430 micro controllers to control the internal clock signal.

Table 9-31: JTAG Basics

Pin	Description
TCK	Test Clock - this pin is the clock signal used for ensuring the timing of the boundary scan system. The TDI shifts values into the appropriate register on the rising edge of TCK. The selected register contents shift out onto TDO on the falling edge of TCK.
TDI	Test Data Input - Test instructions shift into the device through this pin.
TDO	Test Data Output - This pin provides data from the boundary scan registers, i.e. test data shifts out on this pin.
TMS	Test Mode Select - This input which also clocks through on the rising edge of TCK determines the state of the TAP controller.
TRST	This is an optional active low test reset pin. It permits asynchronous TAP controller initialization without affecting other device or system logic.

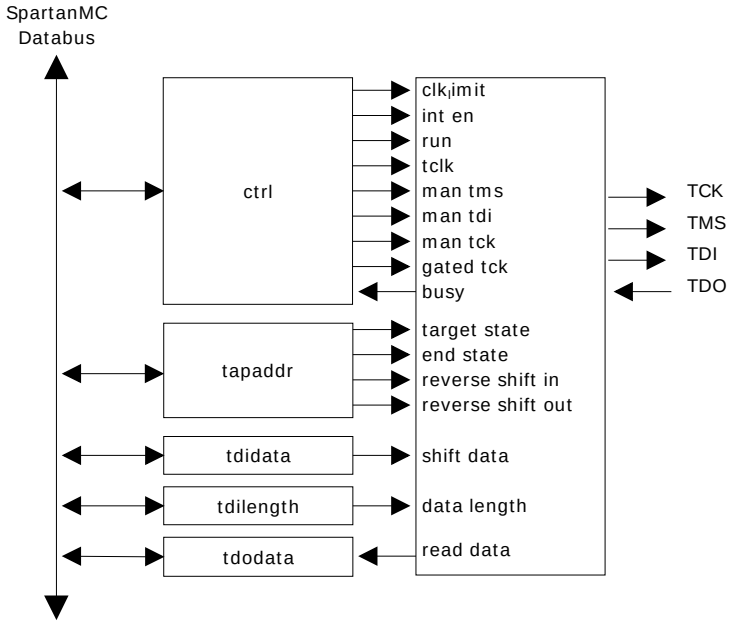


Figure 9-39: JTAG block diagram

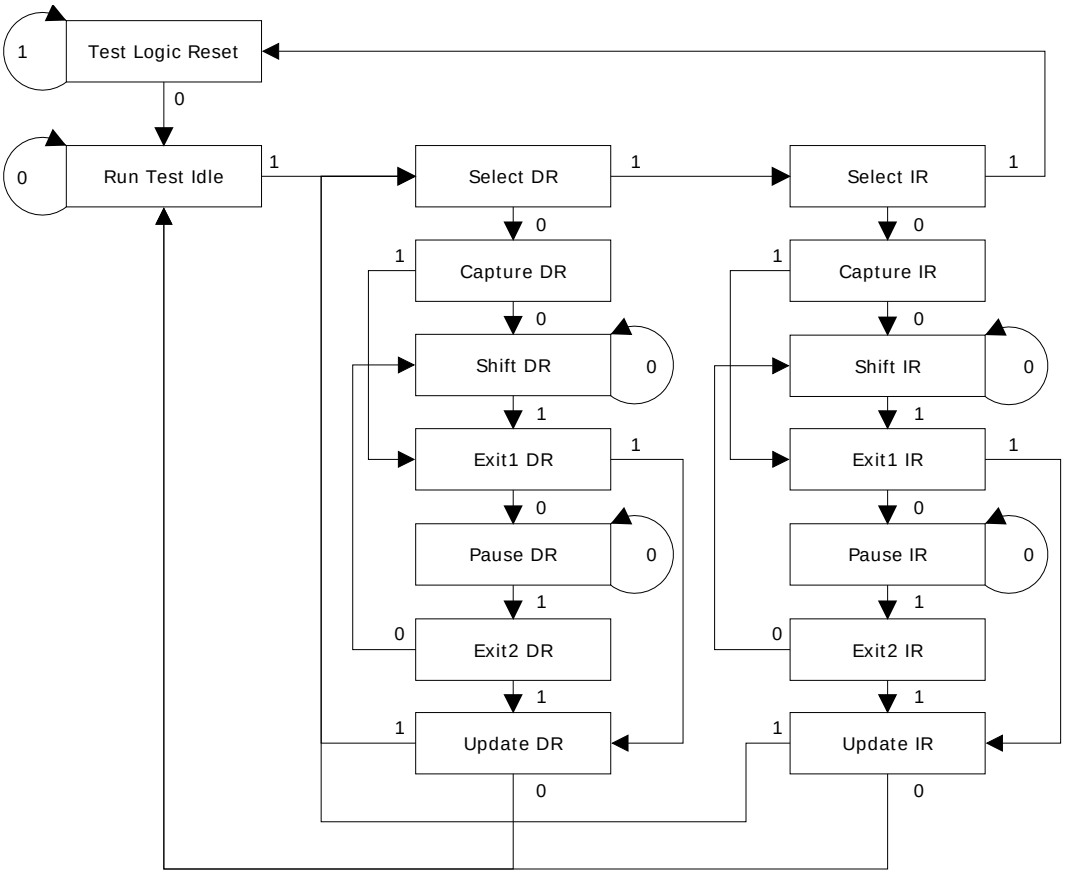


Figure 9-40: JTAG TAP Controller State Machine

9.1. Communication

To shift data to a connected device, you have to bring the TAP Controller in the device into the Run-Test-Idle-State. This can be done by resetting the Controller (clocking TMS=1 6 times) manually and clocking one TMS=0 in to go to the RTI-State. Now the Run-Bit (Bit 11 in the ctrl-register) can be set one and the controller is in automatic mode. Shifting data is very simple. Set target state in the TAP-Control-Register (maybe SHIFT_DR = 4) and set the end state (maybe RUN-TEST-IDLE). Then you put the data into the tdidata register and set the length of this data. Now the controller drives the TAP-Controller in your connected device into the target state (here: 1-0-0), shifts the data and generates a TMS-sequence to drive the TAP-Controller into the end state (here 1-1-0). When the process is done a interrupt is generated when interrupts are enabled. You can also poll the control-register and check the busy-bit (bit 18) to know when the process is done.

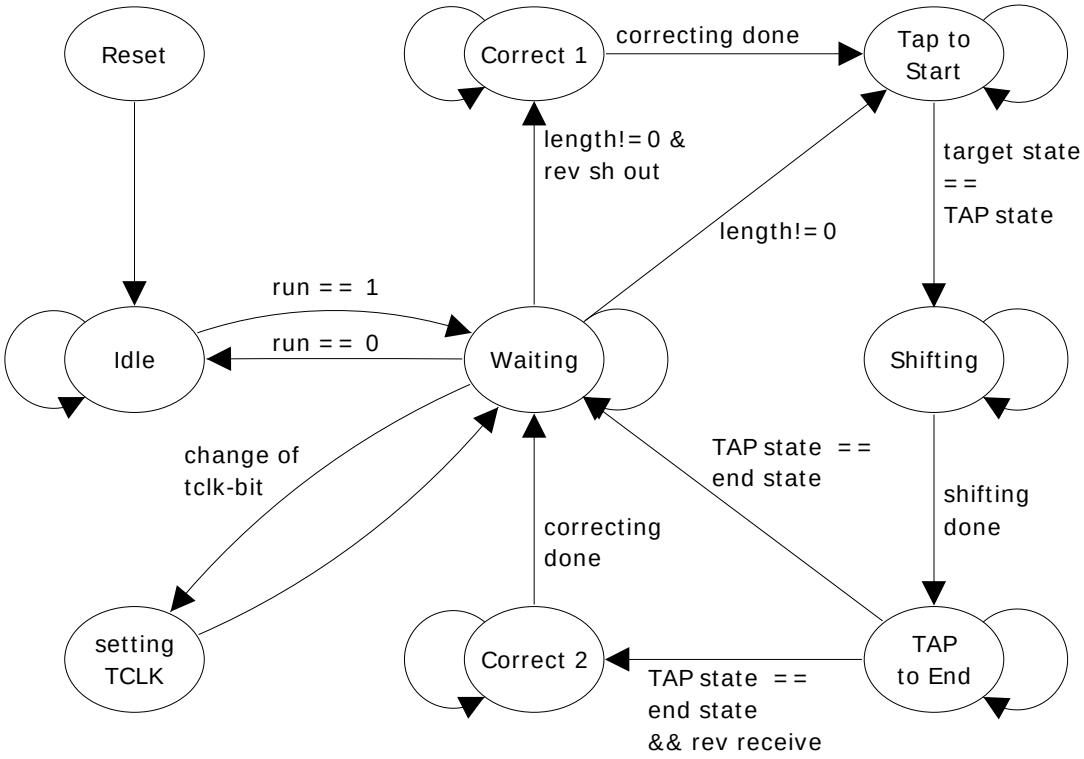


Figure 9-41: JTAG State machine

9.2. Module parameters

This module does not have Synthesis-Parameter

9.3. Peripheral Registers

9.3.1. JTAG Register Description

The JTAG peripheral provides 5 18 bit registers which are mapped to the SpartanMC address space.

Table 9-32: JTAG registers

Offset	Name	Access	Description
0	ctrl	read/ write	Contains the current JTAG setting e.g. clock divider, irq settings, jtag signal levels and generates a busy signal.
1	tapaddresses	read/ write	Register for setting the target- and end state of the shifting process. Also configuring reverse send and reverse receive.
2	tdidata	read/ write	Register for data which should be shifted out.
3	tdilength	read/ write	Register for the length of the data. Writing this registers starts a shift process.
4	tdidata	read	Register for received data.

9.3.2. JTAG Control Register (ctrl)

Table 9-33: JTAG control register layout

Bit	Name	Access	Default	Description
0-9	EN	r/w	1	Clock devive register.
10	IRQ_EN	r/w	0	Enable sending irqs.
11	Run	r/w	0	Enables the automatic mode of the Controller. Before you enable the automatic mode you have to put the TAP controller in your connected device into the RUN-TEST-IDLE mode using the bits for TMS and TCK in this register.
12	TCLK	r/w	0	This bit is a special control bit for Ti-MSP430 microcontroller. It controls the internal clk signal of this microcontroller. Setting TDI to Logic One and holding the TCK for 3 cycles high, set the internal clock signal to one. Setting TDI to zero and holding TCK for three cycles high, set the internal clock signal to zero. The Value of this bit will be set to the clock signal in the MSP430.
13	man TMS	r/w	0	Logical level of the TMS Pin when not in auto mode.

Bit	Name	Access	Default	Description
14	man TDI	r/w	0	Logical level of the TDI Pin when not in auto mode.
15	man TCK	r/w	0	Logical level of the TCK Pin when not in auto mode.
16	gated clk	r/w	0	If this Bit is set to one, the controller does not generate the tck-signal when the controller is idle.
17	busy	r	0	This Bit is set to one when the controller performing a shift operation or changes the tclk-signal (MSP430-feature)

Table 9-33: JTAG control register layout

9.3.3. JTAG TAP Control Register (tapaddr)

Table 9-34: JTAG TAP control register layout

Bit	Name	Access	Default	Description
0-3	TAP target	r/w	0	In this state of the TAP-Controller the provided data will be shifted in.
4-7	TAP end	r/w	0	After shifting the data, the controller will drive the TAP-Controller in the connected device into this state.
8	reverse send	r/w	0	If this bit is set to one, the controller will shift out the MSB of the given data first else the LSB will be shifted out first.
9	reverse receive	r/w	0	If this bit is set to one, the controller will receive the data as MSB else as LSB in the receive shift register.
10-17	not used	-	-	not used

Table 9-34: JTAG TAP control register layout

10. Configurable Parallel Output for 1 to 18 Bit (port_out)

The port output module provides up to 18 output signals. Each output pin can be activated through the corresponding bit in the control register PORT_OUT_OE. If an output is not activated it is set to high-impetance.

10.1. Module Parameters

Table 10-35: PORT_OUT module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
PORT_WIDTH	18	Number of output bits.

10.2. Peripheral Registers

10.2.1. Output Port Register Description

The output port peripheral provides two 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 10-36: PORT_OUT registers

Offset	Name	Access	Description
0	PIN_OUT_DAT	read/ write	Register for outgoing data.
1	PIN_OUT_OE	read/ write	If set to one the corresponding output pin in PIN_OUT_DAT is enabled. After system reset all PIN_BI_OE bits are initialized with zero.

10.2.2. PORT_OUT C-Header for Register Description

```
#ifndef PORT_OUT_H_
#define PORT_OUT_H_

#define OU00 (1 « 0)
#define OU01 (1 « 1)
#define OU02 (1 « 2)
#define OU03 (1 « 3)
#define OU04 (1 « 4)
#define OU05 (1 « 5)
#define OU06 (1 « 6)
#define OU07 (1 « 7)
#define OU08 (1 « 8)
#define OU09 (1 « 9)
#define OU10 (1 « 10)
#define OU11 (1 « 11)
#define OU12 (1 « 12)
#define OU13 (1 « 13)
#define OU14 (1 « 14)
#define OU15 (1 « 15)
#define OU16 (1 « 16)
#define OU17 (1 « 17)

typedef struct port_out {
   volatile unsigned int data;   // read/write
   volatile unsigned int oe;   // read/write
} port_out_t;

#endif /* PORT_OUT_H_ */
```

11. Configurable Parallel Input for 1 to 18 Bit (port_in)

The input port module provides up to 18 input signals. These inputs can be used for switches, buttons etc.. Furthermore, they can be configured to generate interrupts (triggerd by a raising or falling edge) as SoC input. Each input pin provides seperate configuration bits.

11.1. Module Parameters

Table 11-37: PORT_IN module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
PORT_WIDTH	18	Number of input bits.

11.2. Interrupts

An interrupt signals will be generated for each enabled PORT_IN bit. To delete the interrupt flag a read access on PIN_IN_DAT, PIN_IN_IE or PIN_IN_EDGSEL is required.

11.3. Peripheral Registers

11.3.1. Input Port Register Description

The input port peripheral provides four 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 11-38: PORT_IN registers

Offset	Name	Access	Description
0	PIN_IN_DAT	read	Register for incomming data.

Offset	Name	Access	Description
1	PIN_IN_IE	read/ write	Enables the interrupts on PIN_IN_DAT register. After system reset all PIN_IN_IE bits are initialized with zero.
2	PIN_IN_EDGSEL	read/ write	Specify the input edge which triggers the interrupt (0 = falling edge, 1 = raising edge) After system reset all PIN_IN_EDGSEL bits are initialized with zero.
3	PIN_IN_IR_STATUS	read	Register for interrupt flags. If set to one it indicates an interrupt on the corresponding input pin. The interrupt flag will be deleted with a read access on all other module registers except this one. After system reset all PIN_IN_IR_STATUS bits are initialized with zero.

11.3.2. PORT_IN C-Header for Register Description

```
#ifndef PORT_IN_H_
#define PORT_IN_H_

#define IN00 (1 « 0)
#define IN01 (1 « 1)
#define IN02 (1 « 2)
#define IN03 (1 « 3)
#define IN04 (1 « 4)
#define IN05 (1 « 5)
#define IN06 (1 « 6)
#define IN07 (1 « 7)
#define IN08 (1 « 8)
#define IN09 (1 « 9)
#define IN10 (1 « 10)
#define IN11 (1 « 11)
#define IN12 (1 « 12)
#define IN13 (1 « 13)
#define IN14 (1 « 14)
#define IN15 (1 « 15)
#define IN16 (1 « 16)
#define IN17 (1 « 17)

typedef struct port_in {
   volatile unsigned int data; // (r) (r = reset-int)
   volatile unsigned int ie;   // (r/w) (r = reset-int)
   volatile unsigned int edgsel; // (r/w) (r = reset-int)
   volatile unsigned int ir_stat; // (r)
} port_in_t;

#endif /*PORT_IN_H_*/
```

12. Parallel Input/Output for 1 to 18 Bit (port_bi)

The bidirectional port module provides up to 18 inputs or outputs. Each signal pin can be configured through the corresponding bit in the control registers (PIN_BI_DIR, PIN_BI_OE). If configured as input they can be used to generate interrupts (triggerd by a raising or falling edge) on the SoC inputs. If configured as output the pins can be drove in open drain mode or as tri-state output. (The usage in open drain mode requires at least one pull up resistor.)

12.1. Module Parameters

Table 12-39: Bidirectional port module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
PORT_WIDTH	18	Number of Input/Output Bits.
OD_OUTPUT	0	Specify the output mode (0 = tri-state output, 1 = open drain output).

12.2. Interrupts

An interrupt signals will be generated for each enabled PORT_BI bit. To delete the interrupt flag a read access on PIN_BI_DAT, PIN_BI_IR_EDGSEL, PIN_BI_IE, PIN_BI_DIR or PIN_BI_OE is required.

12.3. Peripheral Registers

12.3.1. PORT_BI Register Description

The bidirectional port peripheral provides six 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 12-40: PORT_BI registers

Offset	Name	Access	Description
0	PIN_BI_DAT	read/ write	Register for incomming or outgoing data.
1	PIN_BI_IE	read/ write	Enables the interrupts on PIN_BI_DAT register. After system reset all PIN_BI_IE bits are initialized with zero.
2	PIN_BI_OE	read/ write	If set to one the corresponding output pin in PIN_BI_DAT is enabled. After system reset all PIN_BI_OE bits are initialized with zero.
3	PIN_BI_DIR	read/ write	Specify the direction (input/output) of the port signal. After system reset all PIN_BI_DIR bits are initialized with zero.
4	PIN_BI_EDGSEL	read/ write	Specify the input edge which triggers the interrupt (0 = falling edge, 1 = raising edge) After system reset all PIN_BI_EDGSEL bits are initialized with zero.
5	PIN_BI_IR_STATUS	read	Register for interrupt flags. If set to one it indicates an interrupt on the corresponding input pin. The interrupt flag will be deleted with a read access on all other module registers except this one. After system reset all PIN_BI_IR_STATUS bits are initialized with zero.

12.3.2. PORT_BI C-Header for Register Description

```
#ifndef PORT_BI_H_
#define PORT_BI_H_

#define IO00 (1 « 0)
#define IO01 (1 « 1)
#define IO02 (1 « 2)
#define IO03 (1 « 3)
#define IO04 (1 « 4)
#define IO05 (1 « 5)
#define IO06 (1 « 6)
#define IO07 (1 « 7)
#define IO08 (1 « 8)
#define IO09 (1 « 9)
#define IO10 (1 « 10)
#define IO11 (1 « 11)
#define IO12 (1 « 12)
#define IO13 (1 « 13)
#define IO14 (1 « 14)
#define IO15 (1 « 15)
#define IO16 (1 « 16)
#define IO17 (1 « 17)

typedef struct port_bi {
   volatile unsigned int data; // (r/w) (r = reset-int)
   volatile unsigned int ie;   // (r/w) (r = reset-int)
   volatile unsigned int oe;   // (r/w)
   volatile unsigned int dir;   // (r/w) (r = reset-int)
   volatile unsigned int edgsel; // (r/w) (r = reset-int)
   volatile unsigned int ir_stat; // (r)
} port_bi_t;

#endif /*PORT_BI_H_*/
```

13. Basic Timer (Timer)

The Basic Timer module can be used to devide the system clock frequency to a user defined periodic signal required by the application. For this purpose the Basic Timer provides a configurable prescaler. If enabled, the prescaler allows the usage of all powers of two between 2 and 256 as prescaler value. The input of prescaler block could be connected to the system clock, a dedicated DCM output or to the output (data register) of a previous timer module. The output of the prescaler is used as input of an 18 bit counter which counts up to a programable value and restarts afterwards.

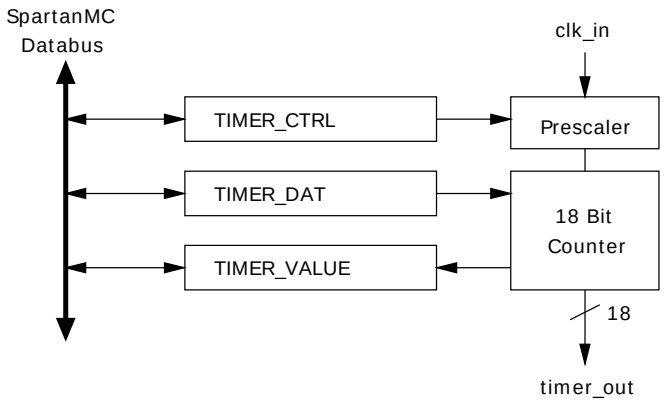


Figure 13-42: Timer block diagram

The basic timer could be used for the following peripherals: Real Time Interrupt (RTI), Watchdog, Capture-, Compare-Module and Pulse-Accumulator-Module. The capture- and compare-module require on their input a complete 18 bit counter register output. While all other modules (RTI, Pulse-Accumulator, Watchdog and additional basic timer) require only one single output bit for their clock input.

13.1. Module parameters

Table 13-41: TIMER module parameters

Parameter	Default Value	Descripton
BASE_ADR	0x10	Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**

13.2. Peripheral Registers

13.2.1. Timer Register Description

The timer peripheral provides three 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 13-42: TIMER registers

Offset	Name	Access	Description
0	TIMER_CTRL	read/ write	Configuration of the timer.
1	TIMER_DAT	read/ write	Maximum value of the 18 bit counter.
2	TIMER_VALUE	read/ write	Current counter value.

13.2.2. TIMER_CTRL Register

Table 13-43: TIMER_CTRL register layout

Bit	Name	Access	Default	Description
0	TI_EN	read/ write	0	If set to one the timer is enabled.
1	TI_PRE_EN	read/ write	0	If set to one the prescaler is enabled.
2-4	TI_PRE_VAL	read/ write	000	Sets the prescaler value : 000 = 2^1 001 = 2^2 010 = 2^3 011 = 2^4 100 = 2^5 101 = 2^6 110 = 2^7 111 = 2^8
5-17	x	read	0	Not used.

Table 13-43: TIMER_CTRL register layout

13.2.3. TIMER_DAT Register

Table 13-44: TIMER_DAT register layout

Bit	Name	Access	Default	Description
0-17	Max Counter	read/ write	x	Register for the maximum counter value.

13.2.4. TIMER_VALUE Register

Table 13-45: TIMER_VALUE register layout

Bit	Name	Access	Default	Description
0-17	Main Counter	read/ write	0	Register for the current counter value. The content of this 18 bit register is used as timer_output and could be connected to other peripherals e.g. capture- or compare-logic. A single bit of this register could also be used to cascade multiple timers.

13.2.5. TIMER C-Header for Register Description

```
#ifndef TIMER_H_
#define TIMER_H_

#define TI_EN (1 « 0)
#define TI_PRE_EN (1 « 1)
#define TI_PRE_VAL (1 « 2) // *0 fuer 2^1 bis *7 fuer 2^8
#define TI_PRE_2 (TI_PRE_VAL * 0)
#define TI_PRE_4 (TI_PRE_VAL * 1)
#define TI_PRE_8 (TI_PRE_VAL * 2)
#define TI_PRE_16 (TI_PRE_VAL * 3)
#define TI_PRE_32 (TI_PRE_VAL * 4)
#define TI_PRE_64 (TI_PRE_VAL * 5)
#define TI_PRE_128 (TI_PRE_VAL * 6)
#define TI_PRE_256 (TI_PRE_VAL * 7)

typedef struct timer {
   volatile unsigned int control; // (r/w)
   volatile unsigned int limit;   // (r/w)
   volatile unsigned int value;   // (r/w)
} timer_t;

#endif /* TIMER_H_ */
```

14. Timer Capture Module (timer-cap)

The timer capture module is used to capture the value of a timer register after an external trigger signal.

Note: **The timer capture module always requires a basic timer module as input. Hence, it can not work autonomously.**

(Otherwise, a basic timer could be used as input for multiple capture moduls.)

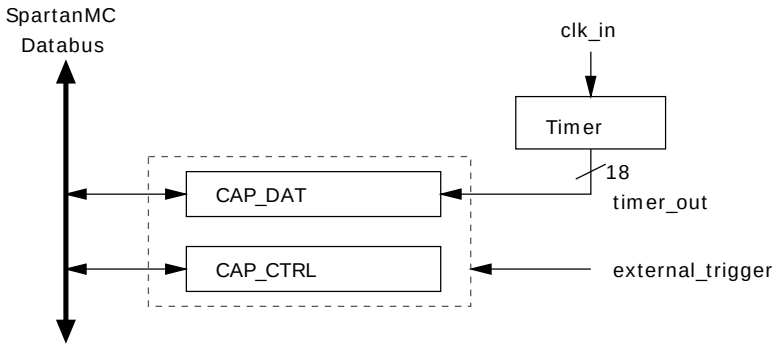


Figure 14-43: Capture module block diagram

14.1. Usage and Interrupts

If this module is triggerd by an external Signal, the current timer value is stored in the local capture register.

Optionally, an interrupt could be generated for each capture event. The interrupt flag is cleared with an access on the data register or control register.

The capture module provides two operation modes for interrupt generation: On the one hand, it could generate its interrupt on edges of a input signal. On the other hand, it could generate its interrupt during a specific level of the input signal. After completion of the capture procedure the enable bit in the control register is cleared. To start a new capture procedure this bit has to be set again.

14.2. Module parameters

Table 14-46: TIMER Capture module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**

14.3. Peripheral Registers

14.3.1. Timer Capture Register Description

The timer capture module provides two 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 14-47: Timer capture registers

Offset	Name	Access	Description
0	CAP_CTRL	read/ write	Configuration of the operation mode. (An acces on this register clears the interrupt flag)
1	CAP_DAT	read	Register for captured data. (An acces on this register clears the interrupt flag)

14.3.2. CAP_DAT Register

Table 14-48: CAP_DAT register layout

Bit	Name	Access	Default	Description
0-17	Capture Value	read	x	The captured data.

14.3.3. CAP_CTRL Register

Table 14-49: CAP_CTRL register layout

Bit	Name	Access	Default	Description
0	CAP_EN	read/ write	0	If set to one the capture logic is enabled. This bit is cleared after capture event completion.
1	CAP_EN_INT	read/ write	0	If set to one the interrupt is enabled.
2-4	CAP_MODE	read/ write	000	Sets the operation mode: 000 = capture disable 001 = not used 010 = capture on falling edge 011 = capture on raising edge 100 = capture on low input signal level 101 = capture on high input signal level 110 = capture on all edges 111 = capture on all edges
5-17	x	read	0	Not used.

Table 14-49: CAP_CTRL register layout

14.3.4. TIMER_CAP C-Header for Register Description

```
#ifndef TIMER_CAP_H_
#define TIMER_CAP_H_

#define CAP_EN       (1 « 0)
#define CAP_EN_INT (1 « 1)
#define CAP_EDGE   (1 « 2)

#define CAP_NON           (CAP_EDGE * 0)
#define CAP_FALLING_EDGE (CAP_EDGE * 2)
#define CAP_RISING_EDGE   (CAP_EDGE * 3)
#define CAP_LOW_LEVEL       (CAP_EDGE * 4)
#define CAP_HIGH_LEVEL   (CAP_EDGE * 5)
#define CAP_ANYTHING_EDGE (CAP_EDGE * 7)

typedef struct cap {
   volatile unsigned int CAP_CTRL; // (r/w)
   volatile unsigned int CAP_DAT;   // (r)
} cap_t;

#endif /* TIMER_CAP_H_ */
```

15. Timer Compare Module (timer-cmp)

The timer compare module is used to generate variable frequencies or programmable duty cycles by comparing an internal value to a given timer value.

Note: **The timer compare module always requires a basic timer module as input. Hence, it can not work autonomously.**

(Otherwise, a basic timer could be used as input for multiple capture moduls.)

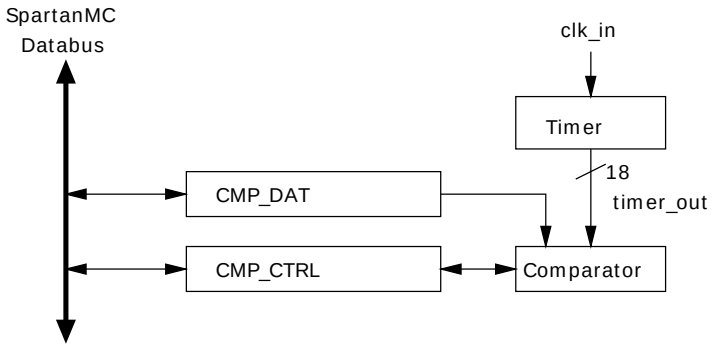


Figure 15-44: Timer compare module block diagram

15.1. Usage and Interrupts

If the programmend value of the compare register equals the current timer value the timer compare module triggers an event. These events could be the generation of an interrupt or the switching of the output pin (set, reset, or negate). In case of an interrupt generation, the interrupt is cleared on each access to the modules registers. In case the module output pin is used, the compare module contains a control register which specifies the behavior of this pin.

15.2. Module parameters

Table 15-50: TIMER Compare module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**

15.3. Peripheral Registers

15.3.1. Timer Compare Register Description

The timer compare module provides tow 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 15-51: Timer Compare registers

Offset	Name	Access	Description
0	CMP_CTRL	read/ write	Specify the operation mode. (An access on this register clears the interrupt flag)
1	CMP_DAT	read/ write	Compare value for the 18 bit counter of the basic timer modules.

15.3.2. Compare Control Register

Table 15-52: CMP_CTRL register layout

Bit	Name	Access	Default	Description
0	CMP_EN	read/ write	0	If set to one the compare logic is enabled.
1	CMP_EN_INT	read/ write	0	If set to one the interrupt is enabled.
2-4	CMP_MODE	read/ write	000	Operation mode (if bit 4 = 0): 000 = Output remains constant 001 = Set output (After trigger event the oputput is always set to 1). 010 = Clear output (After trigger event the oputput is always set to 0). 011 = Toggle output after trigger event
4	OUT_TYP	read/ write	0	If the fourth bit of the operation mod register is set to 1 the output pin switches two times per period. Firstly, on each zero crossing and secondly on the configured maximum value (COMP_DAT). This mechanism enables the usage of the compare module for pulse width modulation (PWM).
2-4	CMP_MODE	read/ write	000	Opertaion mode (if bit 4 = 1): 100 = Output remains constant 101 = Output is set to 1 if timer value equals COMP_DAT -- output is set to 0 if timer value equals 0. 110 = Output is set to 0 if timer value equals COMP_DAT -- output is set to 1 if timer value equals 0.

Bit	Name	Access	Default	Description
				111 = Output is set to 1 if timer value equals COMP_DAT -- output is set to 0 if timer value equals 0.
5	CMP_EN_OUT	read/ write	0	If set to one the comparator output is enabled.
6	CMP_VAL_OUT	read	0	Comparator output bit.
7-17	x	read	0	Not used.

Table 15-52: CMP_CTRL register layout

15.3.3. Compare Value Register

Table 15-53: CMP_DAT register layout

Bit	Name	Access	Default	Description
0-17	CMP_DAT	read/ write	x	18 bit compare value

15.3.4. TIMER_CMP C-Header for Register Description

```
#ifndef TIMER_CMP_H_
#define TIMER_CMP_H_

#define CMP_EN       (1 « 0)
#define CMP_EN_INT (1 « 1)
#define CMP_MODE   (1 « 2)

#define CMP_NON_FRQ   (CMP_MODE * 0)
#define CMP_SET_OUT   (CMP_MODE * 1)
#define CMP_CLEAR_OUT (CMP_MODE * 2)
#define CMP_TOGGLE_OUT (CMP_MODE * 3)
#define CMP_NON_IMP   (CMP_MODE * 4)
#define CMP_C0_N1       (CMP_MODE * 6)
#define CMP_C1_N0       (CMP_MODE * 7)

#define CMP_EN_OUT   (1 « 5)
#define CMP_VAL_OUT   (1 « 6)

typedef struct cmp {
   volatile unsigned int CMP_CTRL; // (r/w)
   volatile unsigned int CMP_DAT;   // (r/w)
} cmp_t;

#endif /* TIMER_CMP_H_ */
```

16. Timer Real Time Interrupt Module (timer-rti)

The Timer RTI module can be used to divide the system clock frequency to a user defined periodic signal required by the application. For this purpose the Timer RTI provides a configurable prescaler. If enabled, the prescaler allows the usage of all powers of two between 2 and 32768 as prescaler value. The input of the prescaler block can be connected to the system clock, a dedicated DCM output or to the output of a previous timer module. The output of the prescaler could be connected to another timer module or could be used to generates an interrupt which for the application.

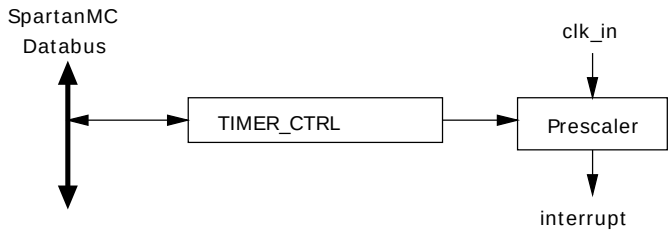


Figure 16-45: Timer RTI block diagram

Note: The timer RTI module could be used as stand alone peripheral or in connection with another timer module (used as input).

16.1. Interrupts

The peripheral generates a cyclic interrupt signals on the maximum value of the timer period.

16.2. Module Parameters

Table 16-54: Timer RTI module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**

16.3. Peripheral Registers

16.3.1. Timer RTI Register Description

The timer RTI peripheral provides one 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 16-55: TIMER RTI registers

Offset	Name	Access	Description
0	RTI_CTRL	read/ write	Specify the operation mode. (An access on this register clears the counter value)

16.3.2. RTI_CTRL Register

Table 16-56: RTI_CTRL register layout

Bit	Name	Access	Default	Description
0	RTI_EN	read/ write	0	If set to one the timer RTI logic is enabled.
1	RTI_EN_INT	read/ write	0	If set to one the timer RTI interrupt is enabled.
2-5	RTI_PRE_VAL	read/ write	0000	Specify the prescaler value: 0000 = 2^0 0001 = 2^1 0010 = 2^2 0011 = 2^3 0100 = 2^4 0101 = 2^5 0110 = 2^6 0111 = 2^7 1000 = 2^8 1001 = 2^9 1010 = 2^10 1011 = 2^11 1100 = 2^12 1101 = 2^13 1110 = 2^14

Bit	Name	Access	Default	Description
				1111 = 2^15
6-17	x	read	0	Not used.

Table 16-56: RTI_CTRL register layout

16.3.3. RTI C-Header for Register Description

```
#ifndef RTI_H_
#define RTI_H_

#define RTI_EN (1 « 0)
#define RTI_EN_INT (1 « 1)
#define RTI_PRE_VAL (1 « 2) // *0 fuer 2^0 bis *15 fuer 2^15

#define RTI_PRE_1 (RTI_PRE_VAL * 0)
#define RTI_PRE_2 (RTI_PRE_VAL * 1)
#define RTI_PRE_4 (RTI_PRE_VAL * 2)
#define RTI_PRE_8 (RTI_PRE_VAL * 3)
#define RTI_PRE_16 (RTI_PRE_VAL * 4)
#define RTI_PRE_32 (RTI_PRE_VAL * 5)
#define RTI_PRE_64 (RTI_PRE_VAL * 6)
#define RTI_PRE_128 (RTI_PRE_VAL * 7)
#define RTI_PRE_256 (RTI_PRE_VAL * 8)
#define RTI_PRE_512 (RTI_PRE_VAL * 9)
#define RTI_PRE_1024 (RTI_PRE_VAL * 10)
#define RTI_PRE_2048 (RTI_PRE_VAL * 11)
#define RTI_PRE_4096 (RTI_PRE_VAL * 12)
#define RTI_PRE_8192 (RTI_PRE_VAL * 13)
#define RTI_PRE_16384 (RTI_PRE_VAL * 14)
#define RTI_PRE_32765 (RTI_PRE_VAL * 15)

typedef struct rti {
   volatile unsigned int control; // (r/w)
} rti_t;

#endif /* RTI_H_ */
```

17. Timer Pulse Accumulator Module (timer-pulseacc)

The timer pulse accumulator module counts impulses from an external input. The module supports two operation modes: Either it counts impulses on an input called PIN or it counts impulses on RTI input until the next impulse on PIN.

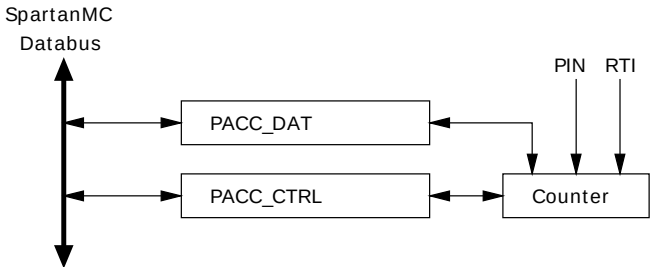


Figure 17-46: Timer Pulse Accumulator block diagram

In the first operation mode the module counts continously all impulses from the input PIN. In the second mode the counter stops if an impulse on RTI occurs. If the counter has stopped (due to an RTI impulse) a read access to the counter register will clear the counter value. Whereas a read access to the control register always clears the counter value in both operation modes.

Note: The timer pulse accumulator can be used as stand alone peripheral or in connection with an basic timer module (used as impulse source).

17.1. Module Parameters

Table 17-57: Timer Pulse Accumulator module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**

17.2. Peripheral Registers

17.2.1. Timer Pulse Accumulator Register Description

The timer pulse accumulator peripheral provides two 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 17-58: Timer Pulse Accumulator Registers

Offset	Name	Access	Description
0	PACC_CTRL	read/ write	Specify the operation mode. (An access on this register clears the counter value)
1	PACC_DAT	read	Counter value register.

17.2.2. PACC_CTRL Register

Table 17-59: PACC_CTRL register layout

Bit	Name	Access	Default	Description
0	PACC_EN	read/ write	0	If set to one the pulse accumulator logic is enabled.
1	PACC_MODE	read/ write	0	Operation mode: 0 = Count all impulses (raising edges) on input PIN. 1 = Count all impulses (raising edges) on imput RTI until an input on PIN occurs.
2-17	x	read	0	Not used.

Table 17-59: PACC_CTRL register layout

17.2.3. PACC_DAT Register

Table 17-60: PACC Counter register layout

Bit	Name	Access	Default	Description
0-17	Counter	read/ write	x	18 bit counter value.

17.2.4. PACC C-Header for Register Description

```
#ifndef PACC_H_
#define PACC_H_

#define PACC_EN (1 « 0)
#define PACC_MODE (1 « 1)

#define PACC_INPMODE (PACC_MODE * 0)
#define PACC_RTIMODE (PACC_MODE * 1)

typedef struct pacc {
   volatile unsigned int control; // (r/w) reset conter
   volatile unsigned int conter;   // (r)
} pacc_t;

#endif /* PACC_H_ */
```

18. Timer Watchdog Module (timer-wdt)

The timer watchdog module can be used for system monitoring purposes. Typically, the application has to clear the watchdog counter at regular intervals otherwise it generates an system reset or interrupt. The timer watchdog module requires two clock inputs: On the one hand CLK_1 which is used as timer input for the watchdog counter, on the other hand CLK_X which garantees the functionality of this module during system reset. (CLK_X has to be completly independend of the remaining SoC design.)

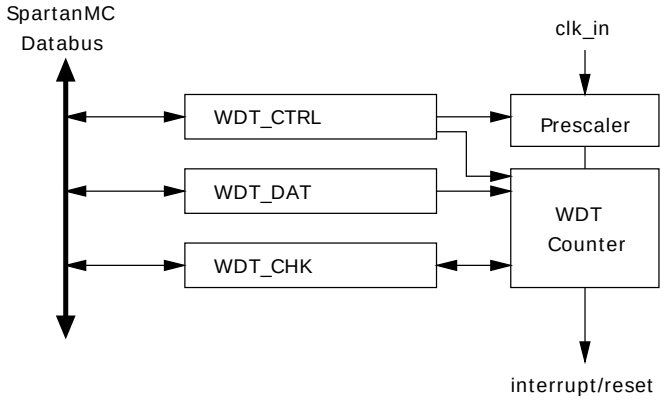


Figure 18-47: Watchdog timer block diagram

Note: The timer watchdog moudule can be used as stand alone peripheral or in connection with an basic timer module (used as counter clock input).

18.1. Usage

During the operation of the watchdog timer peripheral the value in WDT_DAT register is incremented continuously. If the value in WDT_DAT reaches a configured maximum value the peripheral performs a globel reset or an interrupt (with maximum priority). The reset of the watchdog counter is perfomend by writing a specific data word to WDT_CHK register. To determine the alert status of the watchdog module after a system reset bit 5 (ALARM bit) of WDT_CTRL has to be read .

18.2. Module Parameters

Table 18-61: Timer watchdog module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
WDT_RESET_PIN	0x12345	Code word to clear the watchdog timer.

18.3. Interrupts

If the watchdog counter reaches its maximum value an interrupt can be generated. The interrupt can be cleared by writing the WTD_CTRL register.

18.4. Peripheral Registers

18.4.1. Timer Watchdog Register Description

The timer watchdog peripheral provides three 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 18-62: Timer watchdog registers

Offset	Name	Access	Description
0	WDT_CTRL	read/ write	Specify the operation mode of the watchdog timer. (Each write access clears the ALARM bit)
1	WDT_DAT	read/ write	Maximum value of watchdog timer.
2	WDT_CHK	read	If read it contains the current value of the watchdog timer.
2	WDT_CHK	write	**Clears the watchdog timer if written with the configured code word.**

18.4.2. WDT_CTRL Register

Table 18-63: WDT_CTRL register layout

Bit	Name	Access	Default	Description
0	WDT_EN	read/ write	0	If set to one the watchdog timer is enabled.
1	WDT_EN_PRE	read/ write	0	If set to one the prescaler is enabled.
2-4	WDT_PRE_VAL	read/ write	000	Specify the prescaler value : 000 = 2^1 001 = 2^2 010 = 2^3 011 = 2^4 100 = 2^5 101 = 2^6 110 = 2^7 111 = 2^8
5	WDT_ALARM	read/ write	0	Determines a watchdog alert. Set to null on each write access to this register.
6-17	x	read	0	Not used.

Table 18-63: WDT_CTRL register layout

18.4.3. WDT_DAT Register

Table 18-64: WDT maximum value register layout

Bit	Name	Access	Default	Description
0-17	Max Counter	read/ write	x	Specify the maximum counter value.

18.4.4. WDT_CHK Register

Table 18-65: WDT counter register layout

Bit	Name	Access	Default	Description
0-17	Main Counter	read/ (write)	0	If read it contains the current watchdog counter value. If written with WDT_RESET_PIN it clears the watchdog counter value.

18.4.5. WDT C-Header for Register Description

```
#ifndef WDT_H_
#define WDT_H_

#define WDT_EN (1 « 0)
#define WDT_EN_INT (1 « 1)
#define WDT_PRE_VAL (1 « 2) // *0 fuer 2^1 bis *7 fuer 2^8

#define WDT_PRE_2 (WDT_PRE_VAL * 0)
#define WDT_PRE_4 (WDT_PRE_VAL * 1)
#define WDT_PRE_8 (WDT_PRE_VAL * 2)
#define WDT_PRE_16 (WDT_PRE_VAL * 3)
#define WDT_PRE_32 (WDT_PRE_VAL * 4)
#define WDT_PRE_64 (WDT_PRE_VAL * 5)
#define WDT_PRE_128 (WDT_PRE_VAL * 6)
#define WDT_PRE_256 (WDT_PRE_VAL * 7)

#define WDT_ALARM (1 « 5)

typedef struct wdt {
   volatile unsigned int control; // (r/w)
   volatile unsigned int limit;   // (r/w)
   volatile unsigned int val_rst; // (r = val / w PIN = rst)
} wdt_t;

#endif /* WDT_H_ */
```

19. Universal Serial Bus v1.1 Device Controller (USB 1.1)

19.1. Overview

Das Interface wird mit einem auf 18 Bit Breite konfigurierten Blockram realisiert. Ein Port des Blockrams ist mit dem Systembus des SpartanMC verbunden und das zweite Port mit dem USB-Interface. Das Modul hat keine I/O-Register. Die Komunikation erfolgt nur durch Daten lese und schreib Zugriffe in diesen Blockram. Jeder der installierten Endpunkte kann einen Interrupt auslösen, wenn der Host Daten von einem IN-Endpunkt (Tx) gelesen hat oder wenn der Host Daten auf einen OUT-Endpunkt (Rx) abgelegt hat. Das Interface kann maximal 6 Endpunkte realisieren. An den drei erzeugten Signale ist nur noch folgende externe Beschaltung notwendig:

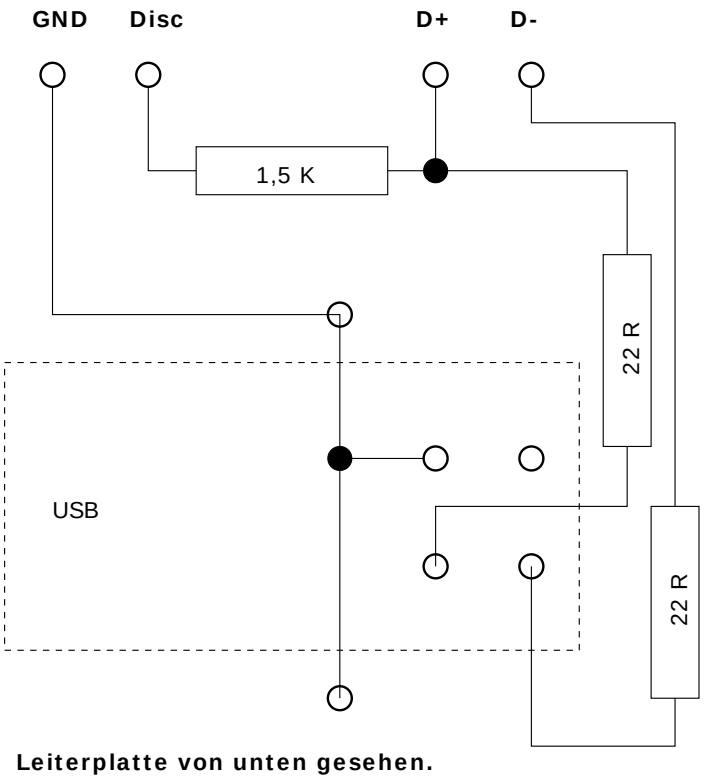


Figure 19-48: Der 1,5K Widerstand zieht D+ bei Disc=1 auf 3,3V wodurch das Interface im FULL-Speed Mode angemeldet wird.

19.2. Speicherorganisation

Die Basiadresse des USB Modul liegt oberhalb des Arbeitsspeichers der Konfiguration. Die Adressen (Offset) 0x000 bis 0x07f des DMA-Speichers sind für die Konfiguration des USB-Interfaces reserviert. Im verbleibenden Bereich (0x080 bis 0x3ff) befinden sich 28 Datenspeicher mit je 32 Worten (64 Bytes). Sie werden je nach Konfiguration den Endpunkten zugeordnet. Bei deaktiviertem Double Buffering sind die Puffer aufeinander folgend den Endpunkten 0 bis 15 zugeordnet. Der Bereich ab 0x280 bleibt unbenutzt. Ist Double Buffering aktiviert werden in aufsteigender Reihenfolge jedem der Endpunkte 0 bis 13 zwei Puffer zugeordnet. Endpunkt 14 und 15 kann nicht verwendet werden! Siehe Adresstabelle. **Die aktuelle Implementierung der Hardware kann nur maximal 6 Endpunkte verwalten!**

19.3. Konfigurations- und Statusregister

Offset	Register	Bemerkung
0x00	ep0c1	Globales Kommandoregister 1
0x01	ep0c2	Globales Kommandoregister 2
0x02	ep1c	Kommandos für Endpunkt 1
0x03	ep1s	Status von Endpunkt 1
0x04	ep2c	Kommandos für Endpunkt 2
0x05	ep2s	Status von Endpunkt 2
...	...	...
0x1e	ep15c	Kommandos für Endpunkt 15
0x1f	ep15s	Status von Endpunkt 15
0x20	glob	Globales Kommandoregister

Table 19-66: Die aktuelle Implementierung unterstützt nur 6 Endpunkte!

19.4. Descriptoren (read only)

Die Descriptoren könnem vom Host nur gelesen aber vom SpartanMC gelesen und geschrieben werden. **Sie müssen vor der Anmeldung am Host initialisiert sein!**

Offset	Bemerkung
0x21	Device Descriptor
0x2a	Configuration Descriptor
0x68	Language Descriptor
0x6a	String Descriptor describing manufacturer

Offset	Bemerkung
0x73	String Descriptor describing product
0x7c	String Descriptor describing serial number

Table 19-67: Descriptoren

19.5. Puffer

Offset	Puffer	Bemerkung	EP ohne double buffering	EP mit double buffering
0x080	data00	Puffer 0 für 64 Byte	0	0 (0)
0x0a0	data01	Puffer 1 für 64 Byte	1	0 (1)
0x0c0	data02	Puffer 2 für 64 Byte	2	1 (0)
0x0e0	data03	Puffer 3 für 64 Byte	3	1 (1)
...	...	...	...	...
0x240	data14	Puffer 14 für 64 Byte	14	7 (0)
0x260	data15	Puffer 15 für 64 Byte	15	7 (1)
...	...	...	...	...
0x3c0	data26	Puffer 26 für 64 Byte	26	13 (0)
0x3e0	data27	Puffer 27 für 64 Byte	27	13 (1)

Table 19-68: Adressen der Puffer

Die aktuelle Implementierung unterstützt nur 6 Endpunkte! Die Anordnung der Bytes in den Puffern kann mit dem Parameter **NOGAP** verändert werden. Mit NOGAP=0 werden die Bytes in der 9 Bit Anordnung des SpartanMC abgelegt. Diese Anordnung ist für die Übertragung von Zeichenketten sinnvoll. Sollen 16 Bit Werte vom SpartanMC in dem DMA-Puffer abgelegt werden, dann müssen die Byte im 8 Bit Abstand in das 18 Bit Wort eingetragen Werden. Diese Anordnung der Bytes wird mit NOGAP=1 eingestellt.

Lesen eines 16 Bit Wortes aus einem Puffer mit NOGAP=0 oder NOGAP=1

```
// lesen 16 Bit
   unsigned int wert16;
#if SB_USB11_0_NOGAP == 0
   unsigned int i;
   unsigned int j;
   i = USB11_0_DMA->data02[0];
   // 2 SpMC Byte zu 16 Bit zusammen fassen
   j = i & 0x3fe00;
   i = i & 0x000ff;
   j = j >> 1;
```

```
   wert16 = j | i;
#else
   wert16 = USB11_0_DMA->data02[0];
#endif
```

Schreiben eines 16 Bit Wortes in einen Puffer mit NOGAP=0 oder NOGAP=1

```
// schreiben 16 Bit
   unsigned int wert16;
#if SB_USB11_0_NOGAP == 0
   unsigned int k;
   unsigned int l;
   k = wert16;
   // 16 Bit in der SpMC Bytanordnung in k bilden
   l = k & 0x3ff00;
   k = k & 0x000ff;
   l = l < < 1;
   k = l | k;
   USB11_0_DMA->data01[0] = k;
#else
   USB11_0_DMA->data01[0] = wert16;
#endif
```

19.6. Bitbelegung der Register

19.6.1. epXc Register

Bit	Bezeichnu	Bedeutung	
6-0	Size	Anzahl zu sendender Bytes	Wert mit 0x3f maskieren (0 entspricht 64 Byte)
10-7	Reserviert		
11	bufsel	Auswahl des aktiven Puffers bei double buffering.	0=Unterer Puffer, 1=Oberer Puffer (im Speicherbereich)
12	in	Tx Endpunkt zum Senden von Daten zum Host	1=Endpunkttyp IN, sonst 0
13	out	Rx Endpunkt zum Empfangen von Daten von Host	1=Endpunkttyp OUT, sonst 0
14	control	Steuerinformationen des Interface	1=Endpunkttyp CONTROL, sonst 0
15	mode	Datentransfer	1=synchron, 0=asynchron
16	intr	enable Interrupt	1=Interrupt enable, 0=Interrupt disable
17	en	enable (HOST darf lesen bzw. schreiben)	EP IN: 1=Puffer enhällt Daten, EP OUT: 1=Puffer leer

Table 19-69: epXc Register

19.6.2. epXs Register (read only)

Bit	Bezeichnung	Bedeutung
6-0	Anzahl empfangender Bytes	Der Wert wird maximal 63 (0 entspricht 64 Byte)
10-7	not used	

Table 19-70: epXs Register (read only)

19.6.3. Globales Steuerregister

Bit	Bezeichnung	Bedeutung
0	iep00	Impuls setzt Interrupt EP 0 zurück
...	...	...
15	iep15	Impuls setzt Interrupt EP 15 zurück
16	ep0ie	enable EP0 Interrupt
17	en	enable USB-Interface

Table 19-71: Globales Steuerregister

19.6.4. USB11 C-Quelle zur Initialisierung des USB DMA Speichers

Die Vorinitialisierung der USB DMA Puffer und Descriptoren muss vor dem Setzen von Bit 17 im Globalen Steuerregister erfolgen! Das Beispiel hier ist dem Zufallszahlen Programm aus dem Quickguide entnommen und initialisiert auch gleich noch die Register ep01c und ep02c. Wird die Initialisierung dieser Register gleich mit in der usb_init.c ausgeführt, dann sind im eigentlichen Programm nach dem Einschalten keine weiteren Initialisierungen mehr notwendig. **Ohne Iniialisierung der Descriptoren ist das Interface nicht funktionsfähig und darf nicht durch setzen von Bit 17 im Globalen Steuerregister aktivert werden!**

```
#include ‹system/peripherals.h›
#include ‹usb.h›

#define NUM_ENDPOINTS SB_USB11_0_ENDPOINTS

struct usb dma_usb_init
__attribute__((section("dma_usb11_0"))) = {
   .ep01c = ( USB_CTRL_MODE | USB_CTRL_IN | 8 ), // EP1 nicht
bereit kein Intr. setzen
   .ep02c = ( USB_CTRL_MODE | USB_CTRL_OUT ), // EP2 nicht
bereit kein Intr. setzen
//   .ep03c = ( USB_CTRL_EN | USB_CTRL_INTR | USB_CTRL_MODE |
USB_CTRL_IN | 8 ),
//   .ep04c = ( USB_CTRL_EN | USB_CTRL_INTR | USB_CTRL_MODE |
USB_CTRL_OUT ),

.device = {
   .hdr.size = 0x12, //num_configs - hdr.size +1
   .hdr.type = USB_DEVICE_DESCRIPTOR,
   .usb_spec = 0x01,
   .class = USB_CLASS_VENDOR_SPECIFIC,
   .subclass = 0x00,
   .protocol = 0xFF,
   .packet_size = 0x40,
   .vendor_id_high = 0x66,
   .vendor_id_low = 0x66,
   .product_id_high = 0x56,
   .product_id_low = 0x78,
   .release_low = 0x10,
   .release_high = 0x00,
   .vendor_descr_idx = 0x01,
   .product_descr_idx = 0x02,
   .serial_descr_idx = 0x03,
   .num_configs = 0x01
},

.config = {
```

```
   .hdr.size = 0x09, //power_cons - hdr.size +1
   .hdr.type = USB_CONFIG_DESCRIPTOR,
   .config_size_low = sizeof(struct usb_config_descr)
       + sizeof(struct usb_iface_descr)
       + sizeof(struct usb_endpoint_descr) * NUM_ENDPOINTS,
   .config_size_high = 0x00,
   .num_ifaces = 0x01,
   .index = 0x01,
   .descr_idx = 0x00,
   .power_mode = USB_SELF_POWERED,
   .power_cons = 0x00
},

.iface = {
   .hdr.size = 0x09, //descr_idx - hdr.size +1
   .hdr.type = USB_INTERFACE_DESCRIPTOR,
   .index = 0x00,
   .alt_setting = 0x00,
   .num_endpoints = NUM_ENDPOINTS,
   .class = USB_CLASS_VENDOR_SPECIFIC,
   .subclass = 0x01,
   .protocol = 0xFF,
   .descr_idx = 0x02
},

.ep01 = {
   .hdr.size = 0x07, //poll_interval - hdr.size +1
   .hdr.type = USB_ENDPOINT_DESCRIPTOR,
   .adr = {
       .direction = USB_DIRECTION_IN,
       .number = 1
   },

   .attr = USB_ATTRIBUTE_BULK,
   .packet_size_low = 0x40,
   .packet_size_high = 0x00,
   .poll_interval = 0x01
},

.ep02 = {
   .hdr.size = 0x07, //poll_interval - hdr.size +1
   .hdr.type = USB_ENDPOINT_DESCRIPTOR,
   .adr = {
       .direction = USB_DIRECTION_OUT,
       .number = 2
   },
```

```
   .attr = USB_ATTRIBUTE_BULK,
   .packet_size_low = 0x40,
   .packet_size_high = 0x00,
   .poll_interval = 0x01
},

.ep03 = {
   .hdr.size = 0x07, //poll_interval - hdr.size +1
   .hdr.type = USB_ENDPOINT_DESCRIPTOR,
   .adr = {
       .direction = USB_DIRECTION_IN,
       .number = 3
   },

   .attr = USB_ATTRIBUTE_BULK,
   .packet_size_low = 0x40,
   .packet_size_high = 0x00,
   .poll_interval = 0x01
},

.ep04 = {
   .hdr.size = 0x07, //poll_interval - hdr.size +1
   .hdr.type = USB_ENDPOINT_DESCRIPTOR,
   .adr = {
       .direction = USB_DIRECTION_OUT,
       .number = 4
   },

   .attr = USB_ATTRIBUTE_BULK,
   .packet_size_low = 0x40,
   .packet_size_high = 0x00,
   .poll_interval = 0x01
},

.lang = {
   .hdr.size = 0x04, //lang_id_high - hdr.size +1
   .hdr.type = USB_STRING_DESCRIPTOR,
   .lang_id_low = 0x09,
   .lang_id_high = 0x04
},

.vendor_str_hdr.size = 0x8, //product_str_hdr.size -
vendor_str_hdr.size
.vendor_str_hdr.type = USB_STRING_DESCRIPTOR,
.vendor_str = { 'T',0,'U',0,'D',0 },
.product_str_hdr.size = 0xA, //serial_str_hdr.size -
product_str_hdr.size
```

```
.product_str_hdr.type = USB_STRING_DESCRIPTOR,
.product_str = { 'S',0,'p',0,'M',0,'C',0 },
.serial_str_hdr.size = 0x8, //data00 - serial_str_hdr.size
.serial_str_hdr.type = USB_STRING_DESCRIPTOR,
.serial_str = { '0',0,'.',0,'0',0 },

// Vorinitialisierung der Puffer mit 8 Byte (es sind maximal 6
EP moeglich)
.data01 = {0x06030,0x06030,0x06031,0x00000}, // EP01 bei
einfacher Pufferung oder EP00.2
.data02 = {0x06030,0x06030,0x06032,0x00000}, // EP02 bei
einfacher Pufferung oder EP01.1
.data03 = {0x06030,0x06030,0x06033,0x00000}, // EP03 bei
einfacher Pufferung oder EP01.2
.data04 = {0x06030,0x06030,0x06034,0x00000}, // EP04 bei
einfacher Pufferung oder EP02.1
.data05 = {0x06030,0x06030,0x06035,0x00000}, // EP05 bei
einfacher Pufferung oder EP02.2
.data06 = {0x06030,0x06030,0x06036,0x00000}, // EP06 bei
einfacher Pufferung oder EP03.1
.data07 = {0x06030,0x06030,0x06037,0x00000}, // EP07 bei
einfacher Pufferung oder EP03.2
.data08 = {0x06030,0x06030,0x06038,0x00000}, // EP08 bei
einfacher Pufferung oder EP04.1
.data09 = {0x06030,0x06030,0x06039,0x00000}, // EP09 bei
einfacher Pufferung oder EP04.2
.data10 = {0x06030,0x06030,0x06230,0x00000}, // EP10 bei
einfacher Pufferung oder EP05.1
.data11 = {0x06030,0x06030,0x06231,0x00000}, // EP11 bei
einfacher Pufferung oder EP05.2
.data12 = {0x06030,0x06030,0x06232,0x00000}, // EP12 bei
einfacher Pufferung oder EP06.1
.data13 = {0x06030,0x06030,0x06233,0x00000} // EP13 bei
einfacher Pufferung oder EP06.2

};
```

19.6.5. USB C-Header for Register Description ("./spartanmc/include/peripherls/usb11.h")

```
#ifndef __USB11_H
#define __USB11_H

// Konstanten und Masken fuer:

////////////////////////////////////////////////////////////
// //
// Konstanten und Datenfelder fuer das USB-Modul des
SpartanMC //
// //
////////////////////////////////////////////////////////////

#define USB11_ATTRIBUTE_BULK 0x02
#define USB11_DIRECTION_IN 0x01
#define USB11_DIRECTION_OUT 0x00
#define USB11_DEVICE_DESCRIPTOR 0x01
#define USB11_CONFIG_DESCRIPTOR 0x02
#define USB11_STRING_DESCRIPTOR 0x03
#define USB11_INTERFACE_DESCRIPTOR 0x04
#define USB11_ENDPOINT_DESCRIPTOR 0x05
#define USB11_SELF_POWERED 0x40

#define USB11_CLASS_VENDOR_SPECIFIC 0xFF

/**
* Konstanten fuer epXXc-Register
*/
#define USB11_CTRL_EN 0x20000 //Bereit zum Datenaustausch mit
dem Host
#define USB11_CTRL_INTR 0x10000 //Interruptfreigabe fuer den
EP
#define USB11_CTRL_MODE 0x08000 //synchron
#define USB11_CTRL_CTRL 0x04000
#define USB11_CTRL_OUT 0x02000
#define USB11_CTRL_IN 0x01000
#define USB11_CTRL_BUFSEL 0x00800 //Buffer HIGH bei
Doppelpufferung waehlen
#define USB11_CTRL_SIZE 0x0007F //Werte von 0 bis 63, wobei 0
als 64 gewertet wird.

/**
* Konstanten fuer das globale Konfigurationsregister
*/
#define USB11_EN 0x20000
```

```
#define USB11_DI 0x00000
#define USB11_IEN_EP0 0x10000 //Interruptfreigabe fuer EP0
#define USB11_IEP15 0x08000
#define USB11_IEP14 0x04000
#define USB11_IEP13 0x02000
#define USB11_IEP12 0x01000
#define USB11_IEP11 0x00800
#define USB11_IEP10 0x00400
#define USB11_IEP09 0x00200
#define USB11_IEP08 0x00100
#define USB11_IEP07 0x00080
#define USB11_IEP06 0x00040
#define USB11_IEP05 0x00020
#define USB11_IEP04 0x00010
#define USB11_IEP03 0x00008
#define USB11_IEP02 0x00004
#define USB11_IEP01 0x00002
#define USB11_IEP00 0x00001

// descriptor header
struct __attribute__((__packed__)) usb_descr_header {
   unsigned char size; // size of descriptor structure
   unsigned char type; // descriptor type
           // 0x01 = device descriptor
           // 0x02 = control descriptor
           // 0x03 = string descriptor
           // 0x04 = interface descriptor
           // 0x05 = endpoint descriptor
};

// device descriptor
struct __attribute__((__packed__)) usb_device_descr {
   struct usb_descr_header hdr; // descriptor header
           // .size = 0x12
           // .type = 0x01
   unsigned char _r0; // reserverd
   unsigned char usb_spec; // usb specification number = 0x01
   unsigned char class; // class code
           // 0xFF = class code is vendor specified
   unsigned char subclass; // subclass code = 0x00
   unsigned char protocol; // protocol code = 0xFF
   unsigned char packet_size; // max packet size for EP0 = 0x40
   unsigned char vendor_id_low;
   unsigned char vendor_id_high;
   unsigned char product_id_low;
   unsigned char product_id_high;
   unsigned char release_low; // device release number
```

```
   unsigned char release_high;
   unsigned char vendor_descr_idx; // index of vendor string
descriptor
   unsigned char product_descr_idx;// index of product string
descriptor
   unsigned char serial_descr_idx; // index of serial string
descriptor
   unsigned char num_configs; // number of possible
configurations
};

// configuration descriptor
struct __attribute__((__packed__)) usb_config_descr {
   struct usb_descr_header hdr; // descriptor header
           // .size = 0x09
           // .type = 0x02
   unsigned char config_size_low; // number of bytes used for
endpoint
   unsigned char config_size_high; // descriptors, including
this structure
   unsigned char num_ifaces; // number of interfaces
   unsigned char index; // value to use as argument to select
this
           // configuration
   unsigned char descr_idx; // index of config string
descriptor
   unsigned char power_mode; // power mode for device
           // 0x40 = self powered
   unsigned char power_cons; // max power consumption in 2 mA
units
};

// interface descriptor
struct __attribute__((__packed__)) usb_iface_descr {
   struct usb_descr_header hdr; // descriptor header
           // .size = 0x09
           // .type = 0x04
   unsigned char index; // index of interface
   unsigned char alt_setting; // alternativ setting
           // 0x00 = no
           // 0x01 = yes (?)
   unsigned char num_endpoints; // number of endpoints
   unsigned char class; // class code
   unsigned char subclass; // subclass code
   unsigned char protocol; // protocol code
   unsigned char descr_idx; // index of string descriptor
};
```

```
// endpoint address
struct __attribute__((__packed__)) usb_endpoint_address {
   unsigned _r0: 1; // reserved
   unsigned direction: 1; // direction
           // 0 = out
           // 1 = in
   unsigned _r1: 3; // reserved
   unsigned number: 4; // endpoint number
};

// endpoint descriptor
struct __attribute__((__packed__)) usb_endpoint_descr {
   struct usb_descr_header hdr; // descriptor header
           // .size = 0x07
           // .type = 0x05
   struct usb_endpoint_address adr;// endpoint address
   unsigned char attr; // endpoint attributes
           // 0x02 = bulk
   unsigned char packet_size_low; // maximum packet size this
endpoint
   unsigned char packet_size_high; // is able to handle
   unsigned char poll_interval; // polling interval
           // (ignored for bulk and
           // control endpoints)
};

// language descriptor
struct __attribute__((__packed__)) usb_language_descr {
   struct usb_descr_header hdr; // descriptor header
           // .size = 0x04
           // .type = 0x03
   unsigned char lang_id_low; // language id
   unsigned char lang_id_high; // 0x0409 = US English
};

// USB-Register und -puffer
typedef struct usb {

// 0x00 - 0x3F
volatile unsigned int ep0c0; // Globales Kommandoregister 1
volatile unsigned int ep0c1; // Globales Kommandoregister 2
volatile unsigned int ep01c; // Kommados fuer Endpunkt 1
volatile unsigned int ep01s; // Status vom Endpunkt 1
volatile unsigned int ep02c; // Kommados fuer Endpunkt 2
volatile unsigned int ep02s; // Status vom Endpunkt 2
volatile unsigned int ep03c; // Kommados fuer Endpunkt 3
```

```
volatile unsigned int ep03s; // Status vom Endpunkt 3
volatile unsigned int ep04c; // Kommados fuer Endpunkt 4
volatile unsigned int ep04s; // Status vom Endpunkt 4
volatile unsigned int ep05c; // Kommados fuer Endpunkt 5
volatile unsigned int ep05s; // Status vom Endpunkt 5
volatile unsigned int ep06c; // Kommados fuer Endpunkt 6
volatile unsigned int ep06s; // Status vom Endpunkt 6
volatile unsigned int ep07c; // Kommados fuer Endpunkt 7
volatile unsigned int ep07s; // Status vom Endpunkt 7
volatile unsigned int ep08c; // Kommados fuer Endpunkt 8
volatile unsigned int ep08s; // Status vom Endpunkt 8
volatile unsigned int ep09c; // Kommados fuer Endpunkt 9
volatile unsigned int ep09s; // Status vom Endpunkt 9
volatile unsigned int ep10c; // Kommados fuer Endpunkt 10
volatile unsigned int ep10s; // Status vom Endpunkt 10
volatile unsigned int ep11c; // Kommados fuer Endpunkt 11
volatile unsigned int ep11s; // Status vom Endpunkt 11
volatile unsigned int ep12c; // Kommados fuer Endpunkt 12
volatile unsigned int ep12s; // Status vom Endpunkt 12
volatile unsigned int ep13c; // Kommados fuer Endpunkt 13
volatile unsigned int ep13s; // Status vom Endpunkt 13
volatile unsigned int ep14c; // Kommados fuer Endpunkt 14
volatile unsigned int ep14s; // Status vom Endpunkt 14
volatile unsigned int ep15c; // Kommados fuer Endpunkt 15
volatile unsigned int ep15s; // Status vom Endpunkt 15

// 0x40
volatile unsigned int glob; // Globales Register

// 0x42
struct usb_device_descr device; // device descriptor
// 0x54
struct usb_config_descr config; // configuration descriptor
// 0x5D
struct usb_iface_descr iface; // interface descriptor

// 0x66 - 0xCE
struct usb_endpoint_descr ep01; // endpoint descriptor
struct usb_endpoint_descr ep02; // endpoint descriptor
struct usb_endpoint_descr ep03; // endpoint descriptor
struct usb_endpoint_descr ep04; // endpoint descriptor
struct usb_endpoint_descr ep05; // endpoint descriptor
struct usb_endpoint_descr ep06; // endpoint descriptor
struct usb_endpoint_descr ep07; // endpoint descriptor
struct usb_endpoint_descr ep08; // endpoint descriptor
struct usb_endpoint_descr ep09; // endpoint descriptor
struct usb_endpoint_descr ep10; // endpoint descriptor
```

```
struct usb_endpoint_descr ep11; // endpoint descriptor
struct usb_endpoint_descr ep12; // endpoint descriptor
struct usb_endpoint_descr ep13; // endpoint descriptor
struct usb_endpoint_descr ep14; // endpoint descriptor
struct usb_endpoint_descr ep15; // endpoint descriptor

unsigned char _padding;

// 0xD0
struct usb_language_descr lang; // language descriptor

// 0xD4
struct usb_descr_header vendor_str_hdr; // vendor string
descriptor header
           // .size = 0x12
           // .type = 0x03
unsigned char vendor_str[16];
// 0xE6
struct usb_descr_header product_str_hdr; // product string
descriptor header
           // .size = 0x12
           // .type = 0x03
unsigned char product_str[16];
// 0xF8
struct usb_descr_header serial_str_hdr; // serial string
descriptor header
           // .size = 0x8
           // .type = 0x03
unsigned char serial_str[6];


// 0x100 - 0x420
volatile unsigned int data00[32]; // Puffer 0 EP00 1
volatile unsigned int data01[32]; // Puffer 1 EP00 2 fuer
Doppelpufferung

volatile unsigned int data02[32]; // Puffer 2 EP01
volatile unsigned int data03[32]; // Puffer 3
volatile unsigned int data04[32]; // Puffer 4 EP02
volatile unsigned int data05[32]; // Puffer 5
volatile unsigned int data06[32]; // Puffer 6 EP03
volatile unsigned int data07[32]; // Puffer 7
volatile unsigned int data08[32]; // Puffer 8 EP04
volatile unsigned int data09[32]; // Puffer 9
volatile unsigned int data10[32]; // Puffer 10 EP05
volatile unsigned int data11[32]; // Puffer 11
volatile unsigned int data12[32]; // Puffer 12 EP06
```

```
volatile unsigned int data13[32]; // Puffer 13
volatile unsigned int data14[32]; // Puffer 14 EP07
volatile unsigned int data15[32]; // Puffer 15
volatile unsigned int data16[32]; // Puffer 16 EP08
volatile unsigned int data17[32]; // Puffer 17
volatile unsigned int data18[32]; // Puffer 18 EP09
volatile unsigned int data19[32]; // Puffer 19
volatile unsigned int data20[32]; // Puffer 20 EP10
volatile unsigned int data21[32]; // Puffer 21
volatile unsigned int data22[32]; // Puffer 22 EP11
volatile unsigned int data23[32]; // Puffer 23
// volatile unsigned int data24[32]; // Puffer 24 EP12
// volatile unsigned int data25[32]; // Puffer 25
// volatile unsigned int data26[32]; // Puffer 26 EP13
// volatile unsigned int data27[32]; // Puffer 27
} usb11_dma_t;


#endif
```

19.6.6. USB C-Header for USB C Funktion and Register Description ("./spartanmc/include/usb.h")

```
#ifndef __USB_H
#define __USB_H

#include ‹peripherals/usb11.h›

// section kept for compatibility with older projects, may be
removed in the
// future

// usb structure type moved to peripherals/usb11.h
#include ‹peripherals/usb11.h›

#define USB_ATTRIBUTE_BULK USB11_ATTRIBUTE_BULK
#define USB_DIRECTION_IN USB11_DIRECTION_IN
#define USB_DIRECTION_OUT USB11_DIRECTION_OUT
#define USB_DEVICE_DESCRIPTOR USB11_DEVICE_DESCRIPTOR
#define USB_CONFIG_DESCRIPTOR USB11_CONFIG_DESCRIPTOR
#define USB_STRING_DESCRIPTOR USB11_STRING_DESCRIPTOR
#define USB_INTERFACE_DESCRIPTOR USB11_INTERFACE_DESCRIPTOR
#define USB_ENDPOINT_DESCRIPTOR USB11_ENDPOINT_DESCRIPTOR
#define USB_SELF_POWERED USB11_SELF_POWERED

#define USB_CLASS_VENDOR_SPECIFIC USB11_CLASS_VENDOR_SPECIFIC

/**
* Konstanten fuer epXXc-Register
*/
#define USB_CTRL_EN USB11_CTRL_EN
#define USB_CTRL_INTR USB11_CTRL_INTR
#define USB_CTRL_MODE USB11_CTRL_MODE
#define USB_CTRL_CTRL USB11_CTRL_CTRL
#define USB_CTRL_OUT USB11_CTRL_OUT
#define USB_CTRL_IN USB11_CTRL_IN
#define USB_CTRL_BUFSEL USB11_CTRL_BUFSEL
#define USB_CTRL_SIZE USB11_CTRL_SIZE

/**
* Konstanten fuer das globale Konfigurationsregister
*/
#define USB_EN USB11_EN
#define USB_DI USB11_DI
#define USB_IEN_EP0 USB11_IEN_EP0
#define USB_IEP15 USB11_IEP15
#define USB_IEP14 USB11_IEP14
```

```
#define USB_IEP13 USB11_IEP13
#define USB_IEP12 USB11_IEP12
#define USB_IEP11 USB11_IEP11
#define USB_IEP10 USB11_IEP10
#define USB_IEP09 USB11_IEP09
#define USB_IEP08 USB11_IEP08
#define USB_IEP07 USB11_IEP07
#define USB_IEP06 USB11_IEP06
#define USB_IEP05 USB11_IEP05
#define USB_IEP04 USB11_IEP04
#define USB_IEP03 USB11_IEP03
#define USB_IEP02 USB11_IEP02
#define USB_IEP01 USB11_IEP01
#define USB_IEP00 USB11_IEP00

// end compatibility section

#define USB_ENDPOINT(usb_base, ep_num) { \
   .ctrl = (volatile unsigned int*)(usb_base + ep_num * 4), \
   .data = (volatile unsigned int*)(usb_base + 0x100 + ep_num *
64), \
   .usb = (struct usb*)(usb_base), \
   .index = (ep_num) \
}

#define USB_ENDPOINT_DB(usb_base, ep_num) { \
   .ctrl = (volatile unsigned int*)(usb_base + ep_num * 4), \
   .data = (volatile unsigned int*)(usb_base + 0x100 + ep_num *
128), \
   .usb = (struct usb*)(usb_base), \
   .index = (ep_num) \
}

#define USB_BUFFER_CURRENT 0
#define USB_BUFFER_OTHER 1

struct usb_ep {
   volatile unsigned int *ctrl;
   volatile unsigned int *data;
   struct usb* usb;
   int index;
};

void usb_init(usb11_dma_t *usb, unsigned int delay);
int usb_ep_poll_txready(struct usb_ep *ep);
void usb_ep_wait_txready(struct usb_ep *ep);
int usb_ep_poll_rxdata(struct usb_ep *ep);
```

```
void usb_ep_wait_rxdata(struct usb_ep *ep);
void usb_ep_packet_send(struct usb_ep *ep, unsigned int size);
void usb_ep_packet_receive(struct usb_ep *ep);
void usb_ep_bufsel(struct usb_ep *ep, int buf_no);
void usb_ep_intr_en(struct usb_ep *ep);
void usb_ep_intr_dis(struct usb_ep *ep);
void usb_ep_intr_clear(struct usb_ep *ep);
volatile unsigned int *usb_ep_get_buffer(struct usb_ep *ep,
unsigned int buf);
void usb_ep_switch_buffer(struct usb_ep *ep);


#endif
```

Eine externe Dokumentation findet sich bei OpenCores:

(http://opencores.org/websvn,listing?repname=usb&path=%2Fusb%2Ftrunk%2Frtl%2Fverilog%2F#path_usb_trunk_rtl_verilog_).

Beschaltung eines USB-Interface:

(http://www.beyondlogic.org/usbnutshell/usb2.shtml).

20. SpartanMC CAN Interface -- Controller area network

20.1. Overview

Das CAN-Interface des SpartanMC besitzt 15 Ein-/Ausgaberegister zur Funktionsauswahl und Funktionssteuerung sowie einen DMA Speicherbereich für die zu sendenden und für die empfangenen CAN-Nachrichten. Im DMA-Speicher können 1 ... 18 Empfangspuffer und 1 ... 18 Sendepuffer installiert werden. Zusäzlich können auch 1 ... 18 Nachrichten Filter Installiert werden die bei ihrer Aktivierung dafür sorgen das nur noch Nachrichten einen Empfangsinterrupt auslösen die mit den eingestellten Filterwerten übereinstimmen. Das CAN-Interface kann Nachrichten mit dem Standard 11 Bit ID und auch mit dem Erweiterten 29 Bit ID empfangen und versenden. Beide ID Formen können gemicht auftreten. Die Datenrate ist von 5 KB/s bis 1000KB/s in den üblichen Stufen varierbar. Die DMA Puffer können beim Systemstart vorinitialisiert werden. Zum Anschluß an einen CAN-Bus muss nur noch der Treiber Schaltkreis MCP2551-I/P mit dem FPGA-Board verbunden werden.

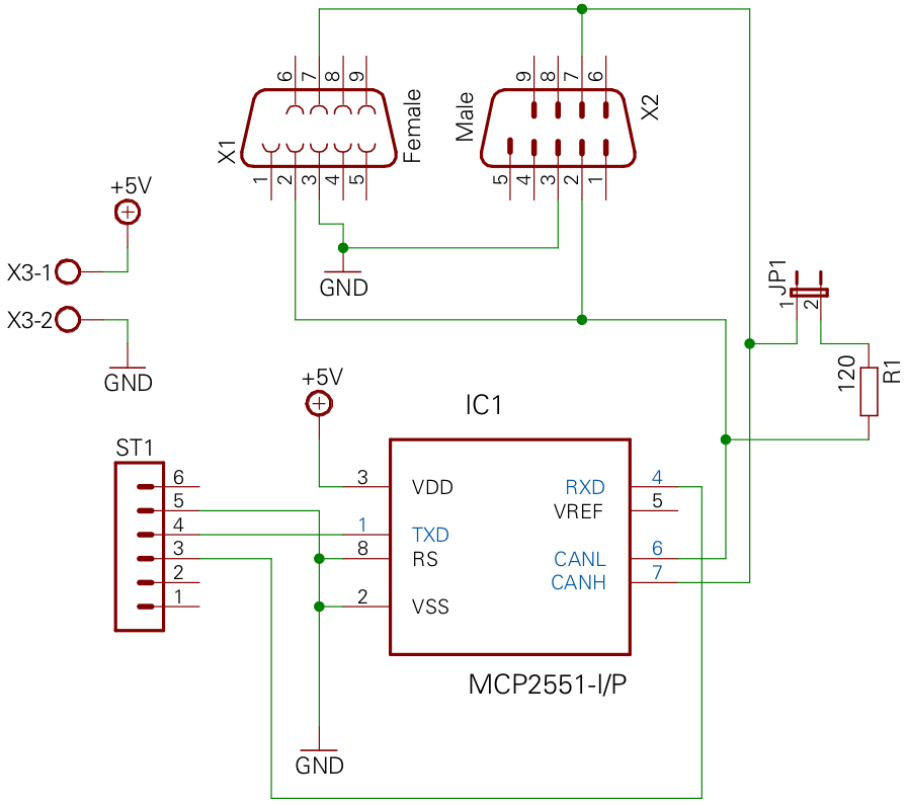


Figure 20-49: Anbindung des CAN-Interface

20.2. Funktion

Die 18 Rx und Tx Puffer werden entsprechend der Priorität gesendet, oder für den Empfang genutzt. Die Puffer Rx 0 und Tx 0 haben die höchste Priorität. Werden alle Tx Puffer im Regiter **tx_request_buffer** als belegt gekennzeichnet, dann wird mit dem Senden von Tx 0 (Bit 17 im Register) begonnen und erst nach dem Senden von Tx 17 (Bit 0 im Register) aufgehört. Nach dem senden eines der Puffer wird das zugehörige Bit im **tx_request_buffer** sofort gelöscht. Achtung, solange noch nicht alle Bits im Register gelöscht sind sollte kein Bit erneut gesetzt werden, da dann dieser Puffer eine höhere Priorität hat als die noch nicht gesendeten Puffer und damit vor den noch nicht gesendeten Puffern gesendet würde. Die Daten aller bereits gesendeten Puffer können aber schon mit neuen Daten belegt werden. Sie müssen dann nur noch freigegeben werden sobald alle Puffer des letzten Starts gesendet sind. Beim Empfang wird immer nach dem höchsten nicht gesetzten Bit im Register **rx_used_buffer** gesucht und im zugeordneten Puffer werden dann die Daten der empfangen Nachricht abgelegt.

Das CAN-Interface kann auch im **Listen Only Mode** eingesetzt werden um alle Nachrichten auf einem CAN-Bus zu protokollieren. Dabei werden auch fehlerhafte und nicht beantwortete Nachrichten angezeigt. Bei Aktivierung der Filter werden nur noch die Nachrichten an ein bestimmtes Gerät oder eine Gerätegruppe angezeigt. Mit diesem Mode kann gut nach Fehlern auf dem CAN-Bus gesucht werden.

20.3. Speicherorganisation

In den ersten 4 Worten der Rx Puffer stehen die maximal 8 Datenbyte der Nachricht. Danach stehen die unteren 18 Bit eines Extended ID, wenn die letzte Nachricht in diesem Format gesendet wurde. Daran folgt ein Wort mit einer Reihe von Statusbits, der Nachrichten Länge und den 11 Bit des Standard ID oder der oberen 11 Bit eines Extendet ID. Im Wort 7 werden die Ausgangssignale aller 18 Filter angezeigt. Hat ein Filter einen Treffer, so wird das Bit auf 1 gesetzt. Nicht initialisierte Filter (alle Bits in den Filterregistern sind 0) liefern immer eine 1. Die Bits sind aber nicht wirksam, wenn keines der Bits im Register **acf_enable** gesetzt ist. Sobald Bits in diesem Register gesetzt sind, werden auch nur noch diese Bits im 7. Wort angezeigt. Man kann daran erkennen welcher Filter beim Empfang der Nachricht aktiv war. Sobald ein oder mehrere Filter aktiv sind, können auch nur noch diese Nachrichten empfangen werden. Gefilter werden kann nach einem ID oder nach ID-Gruppen im Extendet oder Standard ID, auf Remoteframe und auf die oberen 4 Bit des 1. Bytes einer Nachricht. Dazu sind im Filter 2*36 Bit Register installiert. In den ersten 36 Bit muss der Wert eingetragen werden, nachdem gesucht werden soll in dem 2. Register werden die Bits maskiert, die vom Vergleich ausgeschlossern werden sollen. Damit kann man auch Teile des ID maskieren, wodurch die Auswahl von ID Gruppen möglich wird. Im 8. Wort des Rx Puffers wird der CRC der empfangenen Nachricht abgelegt. Die Tx Puffer sind ähnlich aufgebaut. In den ersten 4 Worten stehen auch die 8 Datenbyte der zu sendenden Nachricht. Auch die nächten beiden Worte sind wie beim Rx Puffer mit den unteren 18 Bit des Extendet ID und das Wort 6 mit dem RTR-Bit, dem IDE-Bit, der Länge und den

11 Bit des Standard ID oder den oberen 11 Bit des Extendet ID zu laden. Im 7. Wort ist die Anzahl der Sendewiederholungen abzulegen. Null bedeutet die Nachricht soll nur einmal gesendet werden. Das 8. Wort im Puffer wird nicht genutzt.

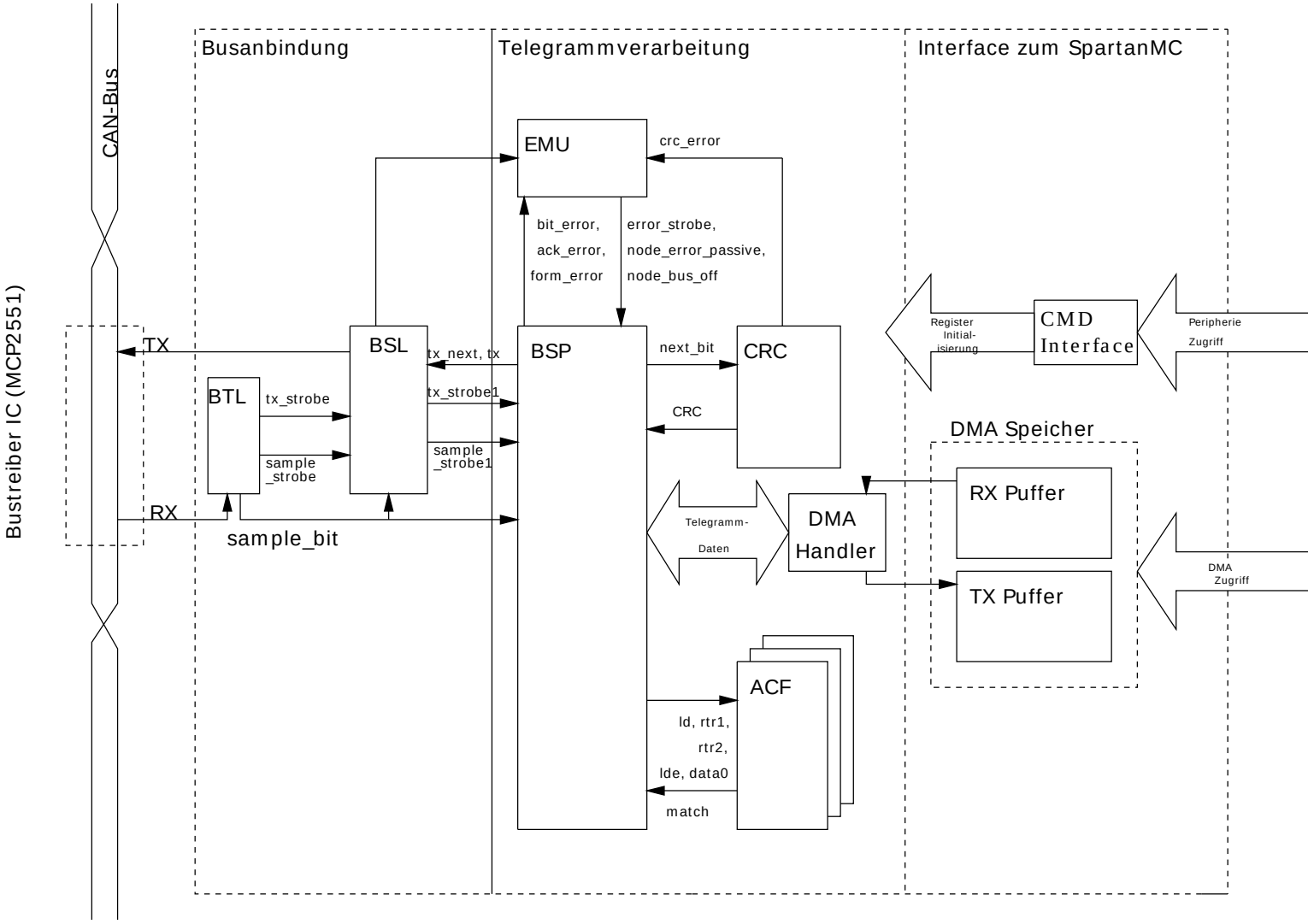


Figure 20-50: CAN-Interface block diagram

20.4. Konfigurations- und Statusregister

Offset	Register	Bit	Verilog Name	Bemerkung (Rechte)	Initialwert
0	Arbeitsregister		work	Schreiben und Lesen immer möglich	
		17-10		(nur Lesen) nicht genutzt	
		9-8	can_mode	setzt Arbeitsmodus des CAN-Controllers 00 = Normal Mode	h3

Offset	Register	Bit	Verilog Name	Bemerkung (Rechte)	Initialwert
				01 = Test Mode 10 = Listen Mode 11 = Konfigurations Mode	
		7	abort_tx	bricht Telegrammsendung ab	b0
		6	en_exception_ir	aktiviert/deaktiviert Exception Interrupt	b0
		5	en_tx_successful_ir	aktiviert/deaktiviert TX Interrupt	b0
		4	en_rx_successful_ir	aktiviert/deaktiviert RX Interrupt	b0
		3	overload_request	fordert ein Overload Frame an	b0
		2	transmitting	(nur Lesen) gesetzt wenn Gesendet wird	b0
		1	receiving	(nur Lesen) gesetzt wenn Empfangen wird	b0
		0	bus_free	(nur Lesen) gesetzt wenn der Bus frei ist	b1
1	Konfigurations Register		choose_config		
		17-2		(nur Lesen) nicht genutzt	
		1	sample_mode	schaltet 3-fach Abtastung an	b0
		0	extended_mode	Ermöglicht das Versenden von Extended Frames	b1
2	Bus-Timing-Reg.		bustiming	Schreiben nur im **Config_Mode** möglich	
		17-10	baudrate_presc	**BRP**	h0
		9-8	sync_jump_width	**SJW**	h2
		7-3	time_segment1	Länge des Prop_Seg + Phase_Seg1 in tq = **tseg1**	h15
		2-0	time_segment2	Länge des Phase_Seg2 in tq-2 = **tseg2**	h4
3	Fehlerzähler-Reg.		error_cnt	Schreiben und Lesen immer möglich	
		17-9	rx_error_cnt	setzen oder lesen Empfangsfehlerzähler	h0
		8-0	tx_error_cnt	setzen oder lesen Sendefehlerzähler	h0
4	Fehlerwarnungs-Reg.		warn_state	Schreiben nur im **Config_Mode + Extended_Mode**	
		17-8		(nur Lesen) nicht genutzt	h0
		7-0	error_warning_limit	setzt Errorwarnungswert (96 = 0x60)	h60
5	ACF Adress-Reg.		acf_select	Schreiben nur im **Config_Mode**	
		17-5		nicht genutzt	h0
		4-0	acf_adr	ACF-Adresse setzen übernimmt die Filterregeln in Register 6-9	h0
5	ACF Freigabe		acf_enable	Schreiben und lesen, wenn **kein Config_Mode**	

Offset	Register	Bit	Verilog Name	Bemerkung (Rechte)	Initialwert
		17-0	acf_en	Jeder ACF-Filter wird mit einer 1 freigegeben. Sind alle Bits 0, werden alle Telegramme angenommen.	h0
6	ACF Code-Reg.1		acf_ac1	Schreiben nur im **Config_Mode**	
		17	ac_rtr1	setzt ACF Code für RTR Bit (SSR bei Ext. Frames)	b0
		16	ac_rtr2	setzt ACF Code für RTR Bit für Ext. Frames	b0
		15-12	ac_data	setzt ACF Code für ersten 4 Bit des Datenteils	h0
		11	ac_we	ist 1 solange das Schreiben noch nicht beendet (read only)	b0
		10-0	ac_b_id	setzt ACF Code für die ersten 11 Bit Identifier	h0
7	ACF Masken-Reg. 1		acf_am1	Schreiben nur im **Config_Mode** (gesetzte Bits werden nicht Verglichen)	
		17	am_rtr1	setzt ACF Maske für RTR Bit (SSR bei Ext. Frames)	b0
		16	am_rtr2	setzt ACF Maske für RTR Bit für Ext. Frames	b0
		15-12	am_data	setzt ACF Maske für ersten 4 Bit des Datenteils	h0
		11	frei		b0
		10-0	am_b_id	setzt ACF Maske für die ersten 11 Bit Identifier	h000
8	ACF Code-Reg. 2		acf_ac2	Schreiben nur im **Config_Mode**	
		17-0	ac_e_id	setzt den ACF Code für den zweiten 18 Bit Identifier	h00000
9	ACF Masken-Reg. 2		acf_am2	Schreiben nur im **Config_Mode** (gesetzte Bits werden nicht Verglichen)	
		17-0	am_e_id	setzt die ACF Maske für den zweiten 18 Bit Identifier	h00000
10	Fehler-Code-Reg.		error_code	(nur Lesen) **Schreiben setzt Exception Interrupt zurück**	
		17	error_warning	gesetzt, wenn error_warning_limit überschritten ist	b0
		16	error_dirction	0 - CAN write / 1 - CAN read	b0
		15	frei		b0
		14-12	error_capture_code [8:6]	Bit-, Form-, Stuff-, CRC-, ACK-Error 000 = Bit-Error 001 = Form-Error 010 = Stuff-Error	h0

Offset	Register	Bit	Verilog Name	Bemerkung (Rechte)	Initialwert
				011 = ACK-Error 100 = CRC-Error 101 = 110 = 111 = alle anderen	
		11-9	frei		h0
		8-4	error_capture_code [4:0]	Zustand des Rx Automaten 0 0000 = 0 0001 = 0 0010 = id1 0 0011 = idle 0 0100 = rtr1 0 0101 = ide 0 0110 = id1 & (bit_cnt > 7) 0 0111 = id2 & (bit_cnt < 5) 0 1000 = crc 0 1001 = r0 0 1010 = data 0 1011 = dlc 0 1100 = rtr2 0 1101 = r1 0 1110 = id2 & (bit_cnt > 13) 0 1111 = id2 & (bit_cnt > 4) & (bit_cnt < 13) 1 0000 = error_lim 1 0001 = error & node_error_active 1 0010 = inter 1 0011 = 1 0100 = overload_lim 1 0101 = 1 0110 = error & node_error_passive 1 0111 = 1 1000 = crc_lim 1 1001 = ack 1 1010 = eof 1 1011 = ack_lim 1 1100 = overload	h00

Offset	Register	Bit	Verilog Name	Bemerkung (Rechte)	Initialwert
				1 1101 = 1 1110 = data >0 1 1111 =	
		3	node_error_passive	gesetzt, wenn der Controller nur noch passiv mithört (127 < Fehlerzähler <= 255)	b0
		2	node_bus_off	gesetzt, wenn sich der Controller vom Bus getrennt hat (Fehlerzähler > 255)	b0
		1	rx_buffer_full_q	gesetzt wenn RX Puffer voll sind	b0
		0	send_failed_q	gesetzt wenn Senden endgültig fehlschlug	b0
11	TX Success-Reg.		tx_succ	(nur Lesen) **Schreiben setzt TX Interrupt zurück**	
		17-10	frei		h00
		9-0	tx_start_adr	Adresse die als letztes im TX Puffer gesendet wurde	h000
12	RX Success-Reg.		rx_succ	(nur Lesen) **Schreiben setzt RX Interrupt zurück**	
		17-10	frei		h00
		9-0	rx_start_adr	Adresse die als letztes im RX Puffer geschrieben wurde	h000
13	TX Puffer-Reg.		tx_buffer	Schreiben und Lesen immer möglich	
		17-0	tx_request_buffer	Belegungsstatus des TX Puffer (1= voll, 0= leer)	h00000
14	RX Puffer-Reg.		rx_buffer	Schreiben und Lesen immer möglich	
		17-0	rx_used_buffer	Belegungsstatus des RX Puffer (1= voll, 0= leer)	h00000

Table 20-72: Die aktuelle Implementierung unterstützt maximal 18 Rx Puffer, 18 Tx Puffer und 18 Filter

20.5. Berechnung der CAN-Bitfrequenz aus den Werten im Bus-Timing Register

f = can_clk / ((BRP+2)*(1+tseg1+tseg2+2)) = 28000 kHz / ((6+2)*(1+21+4+2)) = 28000 kHz / (8*28) = 125 kHz

- Je größer die Summe von (1+tseg1+tseg2+2), desto besser kann synchronisiert werden.
- Je größer SJW, desto mehr Bits kann der Synchronisationszeitpunkt automatisch verschoben werden.

- Der Teilerwert, der sich aus 1+ tseg1 + tseg2 + 2 ergibt, ist die Anzahl der Zeitquanten (tq) der Nominal CAN Bit Time = 100%
- tseg2+2 Minimum sollte > (100 - 87,5)% = 12,5% von der Nominal CAN Bit Time betragen.
- tseg2+2 Maximum sollte < (100 - 75,0)% = 25,0% von der Nominal CAN Bit Time betragen.

Baudrate	BRP	SJW	tseg 1	tseg 2	Bus-Timing Reg.	tseg2+2 in %	tseg2+2 in % bei +SJW	tseg2+2 in % bei -SJW
	8 Bit	2 Bit	5 Bit	3 Bit	Summe = 18 Bit			
1000 kBit/s	0	1	10	1	0x00151	21,4	(3/(1+11+3))*100=20,0	(2/(1+10+2))*100=15,4
500 kBit/s	0	2	21	4	0x002AC	21,4	(6/(1+23+6))*100=20,0	(4/(1+21+4))*100=15,4
250 kBit/s	2	2	21	4	0x00AAC	21,4	(6/(1+23+6))*100=20,0	(4/(1+21+4))*100=15,4
125 kBit/s	6	2	21	4	0x01AAC	21,4	(6/(1+23+6))*100=20,0	(4/(1+21+4))*100=15,4
100 kBit/s	6	3	26	6	0x01BD6	22,9	(8/(1+29+8))*100=21,1	(5/(1+26+5))*100=15,6
50 kBit/s	14	3	26	6	0x03BD6	22,9	(8/(1+29+8))*100=21,1	(5/(1+26+5))*100=15,6
20 kBit/s	38	3	26	6	0x09BD6	22,9	(8/(1+29+8))*100=21,1	(5/(1+26+5))*100=15,6
10 kBit/s	78	3	26	6	0x13BD6	22,9	(8/(1+29+8))*100=21,1	(5/(1+26+5))*100=15,6
5 kBit/s	158	3	26	6	0x27BD6	22,9	(8/(1+29+8))*100=21,1	(5/(1+26+5))*100=15,6

Table 20-73: CAN-Datenraten

Name	Bedeutung	Wertebeeich
BRP	Baud Rate Prescaler	0 ... 253
SJW	Synchroniation Jump Width	1 ... 3
tseg1	Time Segment 1 (= Prop_Segment + Phase_Segment_1)	4 ... 31
tseg2	Time Segment 2 -2 (= Phase_Segment_2 -2	0 ... 7

Table 20-74: Bedeutung der Bezeichner

20.6. Offset im DMA Puffer für den TX Puffer mit der höchsten Priorität abgeleitet aus dem Inhalt vom TX Puffer-Register bei 18 Puffern.

Register Belegung vom tx_buffer		BlockRam Word Adressen	dezimal	Adresse in C
1x xxxx xxxx xxxx xxxx	-->	00 0000 0000	= 00*8	= data_tx00

Register Belegung vom tx_buffer		BlockRam Word Adressen	dezimal	Adresse in C
01 xxxx xxxx xxxx xxxx	-->	00 0000 1000	= 01*8	= data_tx01
00 1xxx xxxx xxxx xxxx	-->	00 0001 0000	= 02*8	= data_tx02
00 01xx xxxx xxxx xxxx	-->	00 0001 1000	= 03*8	= data_tx03
00 001x xxxx xxxx xxxx	-->	00 0010 0000	= 04*8	= data_tx04
00 0001 xxxx xxxx xxxx	-->	00 0010 1000	= 05*8	= data_tx05
00 0000 1xxx xxxx xxxx	-->	00 0011 0000	= 06*8	= data_tx06
00 0000 01xx xxxx xxxx	-->	00 0011 1000	= 07*8	= data_tx07
00 0000 001x xxxx xxxx	-->	00 0100 0000	= 08*8	= data_tx08
00 0000 0001 xxxx xxxx	-->	00 0100 1000	= 09*8	= data_tx09
00 0000 0000 1xxx xxxx	-->	00 0101 0000	= 10*8	= data_tx10
00 0000 0000 01xx xxxx	-->	00 0101 1000	= 11*8	= data_tx11
00 0000 0000 001x xxxx	-->	00 0110 0000	= 12*8	= data_tx12
00 0000 0000 0001 xxxx	-->	00 0110 1000	= 13*8	= data_tx13
00 0000 0000 0000 1xxx	-->	00 0111 0000	= 14*8	= data_tx14
00 0000 0000 0000 01xx	-->	00 0111 1000	= 15*8	= data_tx15
00 0000 0000 0000 001x	-->	00 1000 0000	= 16*8	= data_tx16
00 0000 0000 0000 0001	-->	00 1000 1000	= 17*8	= data_tx17
00 0000 0000 0000 0000	-->	alle Puffer leer		

Table 20-75: Adressen der Tx Puffer

In C werden die Puffer mit z.B.: **CAN_0_DMA->data_tx00** angesprochen.

20.7. Offset im DMA Puffer für den ersten freien RX Puffer abgeleitet aus dem Inhalt vom RX Puffer-Register bei 18 Puffern.

Register Belegung vom rx_buffer		BlockRam Word Adressen	dezimal	Adresse in C
0x xxxx xxxx xxxx xxxx	-->	00 1001 0000	= (00+18)*8	= data_rx00
10 xxxx xxxx xxxx xxxx	-->	00 1001 1000	= (01+18)*8	= data_rx01
11 0xxx xxxx xxxx xxxx	-->	00 1010 0000	= (02+18)*8	= data_rx02
11 10xx xxxx xxxx xxxx	-->	00 1010 1000	= (03+18)*8	= data_rx03
11 110x xxxx xxxx xxxx	-->	00 1011 0000	= (04+18)*8	= data_rx04
11 1110 xxxx xxxx xxxx	-->	00 1011 1000	= (05+18)*8	= data_rx05
11 1111 0xxx xxxx xxxx	-->	00 1100 0000	= (06+18)*8	= data_rx06

Register Belegung vom rx_buffer		BlockRam Word Adressen	dezimal	Adresse in C
11 1111 10xx xxxx xxxx	-->	00 1100 1000	= (07+18)*8	= data_rx07
11 1111 110x xxxx xxxx	-->	00 1101 0000	= (08+18)*8	= data_rx08
11 1111 1110 xxxx xxxx	-->	00 1101 1000	= (09+18)*8	= data_rx09
11 1111 1111 0xxx xxxx	-->	00 1110 0000	= (10+18)*8	= data_rx10
11 1111 1111 10xx xxxx	-->	00 1110 1000	= (11+18)*8	= data_rx11
11 1111 1111 110x xxxx	-->	00 1111 0000	= (12+18)*8	= data_rx12
11 1111 1111 1110 xxxx	-->	00 1111 1000	= (13+18)*8	= data_rx13
11 1111 1111 1111 0xxx	-->	01 0000 0000	= (14+18)*8	= data_rx14
11 1111 1111 1111 10xx	-->	01 0000 1000	= (15+18)*8	= data_rx15
11 1111 1111 1111 110x	-->	01 0001 0000	= (16+18)*8	= data_rx16
11 1111 1111 1111 1110	-->	01 0001 1000	= (17+18)*8	= data_rx17
11 1111 1111 1111 1111	-->	alle Puffer voll		

Table 20-76: Adressen der Rx Puffer

In C werden die Puffer mit z.B.: **CAN_0_DMA->data_rx00** angesprochen.

20.8. Anordnung der Telegramme im DMA-Speicher

CAN Telegramme im Speicher

Word Adr. im Blockran	Inhalt des 18 Bit Wort	freie	Inhalt initialisiert durch can_init.c	Bemerkung
hexadez	Telegrammnummer / Inhalt	bits	hexadezi	aus can_init.c
0x000	Tx01 / frei, Byte 0, Byte 1	2	0x0f0f0	0,0,data0,data1 (00 11110000 11110000)
0x001	Tx01 / frei, Byte 2, Byte 3	2	0x0aa55	0,0,data2,data3 (00 10101010 01010101)
0x002	Tx01 / frei, Byte 4, Byte 5	2	0x0f0f0	0,0,data4,data5 (00 11110000 11110000)
0x003	Tx01 / frei, Byte 6, Byte 7	2	0x0aa55	0,0,data6,data7 (00 10101010 01010101)
0x004	Tx01 / Extended ID	0	0x00000	extID (00 00000000 00000000)
0x005	Tx01 / RTR, IDE, DLC, frei, Standard ID	1	0x28555	stdID (10 1000 0 10101010101)Remoteframe 8 Byte
0x006	Tx01 / frei, Retry Zähler -1	12	0x00003	retry_tx -1 (00 0000 0000 00 000011)
0x007	Tx01 / frei	18	0x03000	leere Speicherstelle
0x008	Tx02 / frei, Byte 0, Byte 1	2	0x0cc33	0,0,data0,data1 (00 11001100 00110011)
0x009	Tx02 / frei, Byte 2, Byte 3	2	0x0f0f0	0,0,data2,data3 (00 11110000 11110000)
0x00A	Tx02 / frei, Byte 4, Byte 5	2	0x0cc33	0,0,data4,data5 (00 11001100 00110011)

Word Adr. im Blockran	Inhalt des 18 Bit Wort	freie	Inhalt initialisiert durch can_init.c	Bemerkung
hexadez	Telegrammnummer / Inhalt	bits	hexadezir	aus can_init.c
0x00B	Tx02 / frei, Byte 6, Byte 7	2	0x0f0f0	0,0,data6,data7 (00 11110000 11110000)
0x00C	Tx02 / Extended ID	0	0x00000	extID (00 00000000 00000000)
0x00D	Tx02 / RTR, IDE, DLC, frei, Standard ID	1	0x0841f	stdID (00 1000 0 10000011111) 8 Byte
0x00E	Tx02 / frei, Retry Zähler -1	12	0x00000	retry_tx -1 (00 0000 0000 00 000000)
0x00F	Tx02 / frei	18	0x00000	leere Speicherstelle
0x010	Tx03 / frei, Byte 0, Byte 1	2	0x0cc33	0,0,data0,data1 (00 11001100 00110011)
0x011	Tx03 / frei, Byte 2, Byte 3	2	0x0f0f0	0,0,data2,data3 (00 11110000 11110000)
0x012	Tx03 / frei, Byte 4, Byte 5	2	0x0cc33	0,0,data4,data5 (00 11001100 00110011)
0x013	Tx03 / frei, Byte 6, Byte 7	2	0x0f0f0	0,0,data6,data7 (00 11110000 11110000)
0x014	Tx03 / Extended ID	0	0x00000	extID (00 00000000 00000000)
0x015	Tx03 / RTR, IDE, DLC, frei, Standard ID	1	0x0841f	stdID (00 1000 0 10000011111) 8 Byte
0x016	Tx03 / frei, Retry Zähler -1	12	0x00000	retry_tx -1 (00 0000 0000 00 000000)
0x017	Tx03 / frei	18	0x00000	leere Speicherstelle
0x018	TX 4 / frei, Byte 0, Byte 1	2	0x0f0f0	0,0,data0,data1 (00 11110000 11110000)
0x019	Tx04 / frei, Byte 2, Byte 3	2	0x10000	0,0,data2,data3 (01 00000000 00000000)
0x01A	Tx04 / frei, Byte 4, Byte 5	2	0x00000	0,0,data4,data5 (00 00000000 00000000)
0x01B	Tx04 / frei, Byte 6, Byte 7	2	0x00000	0,0,data6,data7 (00 00000000 00000000)
0x01C	Tx04 / Extended ID	0	0x3C3C3	extID (11 11000011 11000011)
0x01D	Tx04 / RTR, IDE, DLC, frei, Standard ID	1	0x12533	extID (01 0010 0 10100110011) Extended ID 2 Byte
0x01E	Tx04 / frei, Retry Zähler -1	12	0x00000	retry_tx -1 (00 0000 0000 00 000000)
0x01F	Tx04 / frei	18	0x00000	leere Speicherstelle
0x020	Tx05 / frei	2	0x00000	leere Speicherstelle
◦◦◦				◦◦◦
0x08F	Tx18 / frei			

Table 20-77: Tx Puffer

Durch die Belegung von rx_buffer mit 0x36000 und tx_buffer mit 0x28000 ergibt sich folgender Aufbau der Rx Puffer nach dem 1. Aufruf des Selbsttests.

Word Adr. im Blockran	Inhalt des 18 Bit Wort	freie	Rx Inhalte nach den Tests	Bemerkung zum Inhalte nach Ausführung des Tests
hexadez	Telegrammnummer / Inhalt	bits	hexadezir	
0x090	Rx01 / frei, Byte 0, Byte 1	2	0x00000	00,data0,data1 (00 00000000 00000000) frei
0x09F	Rx02 / CRC	2	0x00000	frei.CRC (00 0000 0000 0000 0000) frei
0x0A0	Rx03 / frei, Byte 0, Byte 1	2	0x00000	00,data0,data1 (00 00000000 00000000) frei
0x0A1	Rx03 / frei, Byte 2, Byte 3	2	0x00000	00,data2,data3 (00 00000000 00000000) frei
0x0A2	Rx03 / frei, Byte 4, Byte 5	2	0x00000	00,data4,data5 (00 00000000 00000000) frei
0x0A3	Rx03 / frei, Byte 6, Byte 7	2	0x00000	00,data6,data7 (00 00000000 00000000) frei
0x0A4	Rx03 / Extended ID	0	0x00000	extID (00 00000000 00000000) frei
0x0A5	Rx03 / RTR, IDE, DLC, frei, Standard ID	1	**0x28555**	Tx1 stdID (10 1000 0 10101010101) Remoteframe 8 Byte
0x0A6	Rx03 / ACF match Bits	0	**0x3FFF2**	Tx1 ACF match (11 1111 1111 1111 0010)
0x0A7	Rx03 / CRC	2	**0x0608E**	00,CRC (00 0110 0000 1000 1110)
0x0A8	Rx04 / frei, Byte 0, Byte 1	2	0x00000	00,data0,data1 (00 00000000 00000000) frei
0x0AF	Rx04 / frei, CRC	2	0x00000	00,CRC (00 0000 0000 0000 0000) frei
0x0B0	Rx05 / frei, Byte 0, Byte 1	2	0x00000	00,data0,data1 (00 00000000 00000000) frei
0x0B7	Rx05 / frei, CRC	2	0x00000	00,CRC (00 0000 0000 0000 0000) frei
0x0B8	Rx06 / frei, Byte 0, Byte 1	2	**0x0CC33**	Tx3 00,data0,data1 (00 11001100 00110011)
0x0B9	Rx06 / frei, Byte 2, Byte 3	2	**0x0f0f0**	Tx3 00,data2,data3 (00 11110000 11110000)
0x0BA	Rx06 / frei, Byte 4, Byte 5	2	**0x0cc33**	Tx3 00,data4,data5 (00 11001100 00110011)
0x0BB	Rx06 / frei, Byte 6, Byte 7	2	**0x0f0f0**	Tx3 00,data6,data7 (00 11110000 11110000)
0x0BC	Rx06 / Extended ID	0	**0x00000**	Tx3 extID (00 00000000 00000000)
0x0BD	Rx06 / RTR, IDE, DLC, frei, Standard ID	1	**0x0841f**	Tx3 stdID (00 1000 0 10000011111) 8 Byte
0x0BE	Rx06 / ACF match Bits	0	**0x3FFF1**	Tx3 ACF match (11 11111111 11110001)
0x0BF	Rx06 / frei,CRC	2	**0x06A52**	00,CRC (00 0110 1010 0101 0010)

Word Adr. im Blockran	Inhalt des 18 Bit Wort	freie	Rx Inhalte nach den Tests	Bemerkung zum Inhalte nach Ausführung des Tests
hexadez	Telegrammnummer / Inhalt	bits	hexadezir	
°°°				°°°
0x11F	Rx18 / CRC	2	0x00000	CRC (00 0110 0000 1000 1110) frei

Table 20-78: Rx Puffer

20.8.1. CAN C-Quelle zur Initialisierung der CAN Telegramm Puffer

```
#include ‹system/peripherals.h›
#include ‹can.h›

// Initialisierung der Tx DMA Telegramm Puffer

struct can_dma dma_can_init
__attribute__((section("dma_can_0"))) = {

// Telegramm 1 Remoteframe 8 Byte
.data_tx00 = {
0x0f0f0, // 0,0,data0,data1 (00 1111 0000 1111 0000)
0x0aa55, // 0,0,data2,data3 (00 1010 1010 0101 0101)
0x0f0f0, // 0,0,data4,data5 (00 1111 0000 1111 0000)
0x0aa55, // 0,0,data6,data7 (00 1010 1010 0101 0101)
0x00000, // extID (00 0000 0000 0000 0000)
0x28555, // stdID (10 1000 0 101_0101_0101)Remoteframe 8 Byte
0x00000, // retry_tx -1 (00 0000 0000 00 00_0011)
0x03000}, // leere Speicherstelle

// Telegramm 2 8 Byte
.data_tx01 = {
0x0cc33, // 0,0,data0,data1 (00 1100 1100 0011 0011)
0x0f0f0, // 0,0,data2,data3 (00 1111 0000 1111 0000)
0x0cc33, // 0,0,data4,data5 (00 1100 1100 0011 0011)
0x0f0f0, // 0,0,data6,data7 (00 1111 0000 1111 0000)
0x00000, // extID (00 0000 0000 0000 0000)
0x0841f, // stdID (00 1000 0 100_0001_1111) 8 Byte
0x00004, // retry_tx -1 (00 0000 0000 00 00_0100)
0x00000}, // leere Speicherstelle

// Telegramm 3 8 Byte
.data_tx02 = {
0x0cc33, // 0,0,data0,data1 (00 1100 1100 0011 0011)
0x0f0f0, // 0,0,data2,data3 (00 1111 0000 1111 0000)
0x0cc33, // 0,0,data4,data5 (00 1100 1100 0011 0011)
0x0f0f0, // 0,0,data6,data7 (00 1111 0000 1111 0000)
0x00000, // extID (00 0000 0000 0000 0000)
0x0841f, // stdID (00 1000 0 100_0001_1111) 8 Byte
0x00000, // retry_tx -1 (00 0000 0000 00 00_0000)
0x00000}, // leere Speicherstelle

// Telegramm 4 --> 2 Byte Extended ID
.data_tx03 = {
0x0f0f0, // 0,0,data0,data1 (00 1111 0000 1111 0000)
0x10000, // 0,0,data2,data3 (01 0000 0000 0000 0000)
```

```
0x00000, // 0,0,data4,data5 (00 0000 0000 0000 0000)
0x00000, // 0,0,data6,data7 (00 0000 0000 0000 0000)
0x3c3c3, // extID (11 1100 0011 1100 0011)
0x12533, // extID (01 0010 0 101_0011_0011) Extended ID 2 Byte
0x00000, // retry_tx -1 (00 0000 0000 00 00_0000)
0x00000}, // leere Speicherstelle

// Telegramm 5 --> 8 Byte Extended ID
.data_tx04 = {
0x0f0f0, // 0,0,data0,data1 (00 1111 0000 1111 0000)
0x0f0f0, // 0,0,data2,data3 (00 1111 0000 1111 0000)
0x0cc33, // 0,0,data4,data5 (00 1100 1100 0011 0011)
0x0aa55, // 0,0,data6,data7 (00 1010 1010 0101 0101)
0x3c3c3, // extID (11 1100 0011 1100 0011)
0x18533, // extID (01 1000 0 101_0011_0011) Extended ID 8 Byte
0x00001, // retry_tx -1 (00 0000 0000 00 00_0001)
0x00000}, // leere Speicherstelle

// Telegramm 6 fuer Eingabe im Programm oder --> Extended ID
Remoteframe 8 Byte
.data_tx05 = {
0x0f0f0, // 0,0,data0,data1 (00 1111 0000 1111 0000)
0x0f0f0, // 0,0,data2,data3 (00 1111 0000 1111 0000)
0x0cc33, // 0,0,data4,data5 (00 1100 1100 0011 0011)
0x0aa55, // 0,0,data6,data7 (00 1010 1010 0101 0101)
0x3c3c3, // extID (11 1100 0011 1100 0011)
0x38533, // extID (11 1000 0 101_0011_0011) Extended ID
Remoteframe 8 Byte
0x00001, // retry_tx -1 (00 0000 0000 00 00_0001)
0x00000}, // leere Speicherstelle

// Telegramm 7 --> 1 Byte
.data_tx06 = {
0x00b00, // 0,0,data0,data1 (00 0000 1011 0000 0000)
0x00000, // 0,0,data2,data3 (00 0000 0000 0000 0000)
0x00000, // 0,0,data4,data5 (00 0000 0000 0000 0000)
0x00000, // 0,0,data6,data7 (00 0000 0000 0000 0000)
0x00000, // extID (00 0000 0000 0000 0000)
0x01720, // stdID (00 0001 0 111_0010_0000) 1 Byte
0x00000, // retry_tx -1 (00 0000 0000 00 00_0000)
0x00000}, // leere Speicherstelle

// Telegramm 8 --> 0 Byte
.data_tx07 = {
0x01122, // 0,0,data0,data1 (00 0001 0001 0010 0010)
0x03344, // 0,0,data2,data3 (00 0011 0011 0100 0100)
0x05566, // 0,0,data4,data5 (00 0101 0101 0110 0110)
```

```
0x07788, // 0,0,data6,data7 (00 0111 0111 1000 1000)
0x00000, // extID (00 0000 0000 0000 0000)
0x00720, // stdID (00 0000 0 111_0010_0000) 0 Byte
0x00000, // retry_tx -1 (00 0000 0000 00 00_0000)
0x00000}, // leere Speicherstelle

// Telegramm 9 Remoteframe 1 Byte
.data_tx08 = {
0x0f0f0, // 0,0,data0,data1 (00 1111 0000 1111 0000)
0x0aa55, // 0,0,data2,data3 (00 1010 1010 0101 0101)
0x0f0f0, // 0,0,data4,data5 (00 1111 0000 1111 0000)
0x0aa55, // 0,0,data6,data7 (00 1010 1010 0101 0101)
0x00000, // extID (00 0000 0000 0000 0000)
0x21555, // stdID (10 0001 0 101_0101_0101)Remoteframe 1 Byte
0x00000, // retry_tx -1 (00 0000 0000 00 00_0011)
0x03000}, // leere Speicherstelle

// Telegramm 10 7 Byte
.data_tx09 = {
0x0cc33, // 0,0,data0,data1 (00 1100 1100 0011 0011)
0x0f0f0, // 0,0,data2,data3 (00 1111 0000 1111 0000)
0x0cc33, // 0,0,data4,data5 (00 1100 1100 0011 0011)
0x0f0f0, // 0,0,data6,data7 (00 1111 0000 1111 0000)
0x00000, // extID (00 0000 0000 0000 0000)
0x0741f, // stdID (00 0111 0 100_0001_1111) 7 Byte
0x00004, // retry_tx -1 (00 0000 0000 00 00_0100)
0x00000}, // leere Speicherstelle

// Telegramm 11 6 Byte
.data_tx10 = {
0x0cc33, // 0,0,data0,data1 (00 1100 1100 0011 0011)
0x0f0f0, // 0,0,data2,data3 (00 1111 0000 1111 0000)
0x0cc33, // 0,0,data4,data5 (00 1100 1100 0011 0011)
0x0f0f0, // 0,0,data6,data7 (00 1111 0000 1111 0000)
0x00000, // extID (00 0000 0000 0000 0000)
0x0641f, // stdID (00 0110 0 100_0001_1111) 6 Byte
0x00004, // retry_tx -1 (00 0000 0000 00 00_0100)
0x00000}, // leere Speicherstelle

// Telegramm 12 --> 0 Byte Extended ID
.data_tx11 = {
0x0f0f0, // 0,0,data0,data1 (00 1111 0000 1111 0000)
0x10000, // 0,0,data2,data3 (01 0000 0000 0000 0000)
0x00000, // 0,0,data4,data5 (00 0000 0000 0000 0000)
0x00000, // 0,0,data6,data7 (00 0000 0000 0000 0000)
0x3c3c3, // extID (11 1100 0011 1100 0011)
0x10533, // extID (01 0000 0 101_0011_0011) Extended ID 0 Byte
```

```
0x00000, // retry_tx -1 (00 0000 0000 00 00_0000)
0x00000}, // leere Speicherstelle

// Telegramm 13 --> 7 Byte Extended ID
.data_tx12 = {
0x0f0f0, // 0,0,data0,data1 (00 1111 0000 1111 0000)
0x0f0f0, // 0,0,data2,data3 (00 1111 0000 1111 0000)
0x0cc33, // 0,0,data4,data5 (00 1100 1100 0011 0011)
0x0aa55, // 0,0,data6,data7 (00 1010 1010 0101 0101)
0x3c3c3, // extID (11 1100 0011 1100 0011)
0x17533, // extID (01 0111 0 101_0011_0011) Extended ID 7 Byte
0x00001, // retry_tx -1 (00 0000 0000 00 00_0001)
0x00000}, // leere Speicherstelle

// Telegramm 14 --> Extended ID Remoteframe 7 Byte
.data_tx13 = {
0x0f0f0, // 0,0,data0,data1 (00 1111 0000 1111 0000)
0x0f0f0, // 0,0,data2,data3 (00 1111 0000 1111 0000)
0x0cc33, // 0,0,data4,data5 (00 1100 1100 0011 0011)
0x0aa55, // 0,0,data6,data7 (00 1010 1010 0101 0101)
0x3c3c3, // extID (11 1100 0011 1100 0011)
0x37533, // extID (11 0111 0 101_0011_0011) Extended ID
Remoteframe 7 Byte
0x00001, // retry_tx -1 (00 0000 0000 00 00_0001)
0x00000}, // leere Speicherstelle

// Telegramm 15 --> 2 Byte
.data_tx14 = {
0x00b0c, // 0,0,data0,data1 (00 0000 1011 0000 1100)
0x00000, // 0,0,data2,data3 (00 0000 0000 0000 0000)
0x00000, // 0,0,data4,data5 (00 0000 0000 0000 0000)
0x00000, // 0,0,data6,data7 (00 0000 0000 0000 0000)
0x00000, // extID (00 0000 0000 0000 0000)
0x02720, // stdID (00 0010 0 111_0010_0000) 2 Byte
0x00000, // retry_tx -1 (00 0000 0000 00 00_0000)
0x00000}, // leere Speicherstelle

// Telegramm 16 --> 6 Byte Extended ID
.data_tx15 = {
0x0f0f0, // 0,0,data0,data1 (00 1111 0000 1111 0000)
0x0f0f0, // 0,0,data2,data3 (00 1111 0000 1111 0000)
0x0cc33, // 0,0,data4,data5 (00 1100 1100 0011 0011)
0x0aa55, // 0,0,data6,data7 (00 1010 1010 0101 0101)
0x3c3c3, // extID (11 1100 0011 1100 0011)
0x16533, // extID (01 0110 0 101_0011_0011) Extended ID 6 Byte
0x00001, // retry_tx -1 (00 0000 0000 00 00_0001)
0x00000}, // leere Speicherstelle
```

```
// Telegramm 17 --> 3 Byte
.data_tx16 = {
0x00b0c, // 0,0,data0,data1 (00 0000 1011 0000 1100)
0x00d00, // 0,0,data2,data3 (00 0000 1101 0000 0000)
0x00000, // 0,0,data4,data5 (00 0000 0000 0000 0000)
0x00000, // 0,0,data6,data7 (00 0000 0000 0000 0000)
0x00000, // extID (00 0000 0000 0000 0000)
0x03720, // stdID (00 0011 0 111_0010_0000) 3 Byte
0x00000, // retry_tx -1 (00 0000 0000 00 00_0000)
0x00000}, // leere Speicherstelle

// Telegramm 18 --> 4 Byte
.data_tx17 = {
0x00b0c, // 0,0,data0,data1 (00 0000 1011 0000 1100)
0x00d0e, // 0,0,data2,data3 (00 0000 1101 0000 1110)
0x00000, // 0,0,data4,data5 (00 0000 0000 0000 0000)
0x00000, // 0,0,data6,data7 (00 0000 0000 0000 0000)
0x00000, // extID (00 0000 0000 0000 0000)
0x04720, // stdID (00 0100 0 111_0010_0000) 4 Byte
0x00000, // retry_tx -1 (00 0000 0000 00 00_0000)
0x00000}, // leere Speicherstelle

// Initialisierung der Rx DMA Telegramm Puffer

.data_rx00 = {
0x00000, // 0,0,data0,data1 (00 0000 0000 0000 0000)
0x00000, // 0,0,data2,data3 (00 0000 0000 0000 0000)
0x00000, // 0,0,data4,data5 (00 0000 0000 0000 0000)
0x00000, // 0,0,data6,data7 (00 0000 0000 0000 0000)
0x00000, // extID (00 0000 0000 0000 0000)
0x00000, // stdID (00 0000 0 000_0000_0000)
RTR,IDE,DLC,frei,stdID
0x00000, // ACF match Bits
0x00000}, // CRC des empfangen Telegramms

.data_rx01 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx02 = {
```

```
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx03 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx04 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx05 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx06 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
```

```
0x00000},

.data_rx07 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx08 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx09 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx10 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx11 = {
0x00000,
0x00000,
0x00000,
0x00000,
```

```
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx12 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx13 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx14 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx15 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx16 = {
0x00000,
```

```
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000},

.data_rx17 = {
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000,
0x00000}

};
```

Eine can_init.c zur Vorinitialisierung der CAN DMA Puffer ist im CAN Test Project zu finden" ./hw-tests/can-test/atlys/app/PCAN-longtime-tx/src".

20.8.2. CAN C-Header for Register Description ("./spartanmc/include/peripherls/can.h")

```
#ifndef __CAN_HW_H
#define __CAN_HW_H

// Konstanten und Masken fuer:

// 1. work Register
#define CAN_MODE_NORMAL 0x00000
#define CAN_MODE_CONFIG 0x00300
#define CAN_MODE_LISTEN_ONLY 0x00200
#define CAN_MODE_SELF_TEST 0x00100
#define CAN_ABORT_TX 0x00080
#define CAN_EN_EXCEPTION_IR 0x00040
#define CAN_TX_SUCCESSFUL_IR 0x00020
#define CAN_RX_SUCCESSFUL_IR 0x00010
#define CAN_OVERLOAD_REQUEST 0x00008
#define CAN_TRANSMITTING 0x00004
#define CAN_RECEIVING 0x00002
#define CAN_BUS_FREE 0x00001

// 2. choose_config Register
#define CAN_SAMPLE_MODE 0X00002
#define CAN_EXTENDED_MODE 0x00001

// 3. bustiming Register
#define CAN_BAUDRATE_PRESC_000 0x00000
#define CAN_BAUDRATE_PRESC_001 0x00400
#define CAN_BAUDRATE_PRESC_002 0x00800
#define CAN_BAUDRATE_PRESC_003 0x00C00
#define CAN_BAUDRATE_PRESC_004 0x01000
#define CAN_BAUDRATE_PRESC_005 0x01400
#define CAN_BAUDRATE_PRESC_006 0x01800
#define CAN_BAUDRATE_PRESC_007 0x01C00
#define CAN_BAUDRATE_PRESC_008 0x02000
#define CAN_BAUDRATE_PRESC_009 0x02400
#define CAN_BAUDRATE_PRESC_010 0x02800
#define CAN_BAUDRATE_PRESC_011 0x02C00
#define CAN_BAUDRATE_PRESC_012 0x03000
#define CAN_BAUDRATE_PRESC_013 0x03400
#define CAN_BAUDRATE_PRESC_014 0x03800
#define CAN_BAUDRATE_PRESC_015 0x03C00
#define CAN_BAUDRATE_PRESC_016 0x04000
#define CAN_BAUDRATE_PRESC_017 0x04400
#define CAN_BAUDRATE_PRESC_018 0x04800
#define CAN_BAUDRATE_PRESC_019 0x04C00
```

```
#define CAN_BAUDRATE_PRESC_020 0x05000
#define CAN_BAUDRATE_PRESC_021 0x05400
#define CAN_BAUDRATE_PRESC_022 0x05800
#define CAN_BAUDRATE_PRESC_023 0x05C00
#define CAN_BAUDRATE_PRESC_024 0x06000
#define CAN_BAUDRATE_PRESC_025 0x06400
#define CAN_BAUDRATE_PRESC_026 0x06800
#define CAN_BAUDRATE_PRESC_027 0x06C00
#define CAN_BAUDRATE_PRESC_028 0x07000
#define CAN_BAUDRATE_PRESC_029 0x07400
#define CAN_BAUDRATE_PRESC_030 0x07800
#define CAN_BAUDRATE_PRESC_031 0x07C00
#define CAN_BAUDRATE_PRESC_032 0x08000
#define CAN_BAUDRATE_PRESC_033 0x08400
#define CAN_BAUDRATE_PRESC_034 0x08800
#define CAN_BAUDRATE_PRESC_035 0x08C00
#define CAN_BAUDRATE_PRESC_036 0x09000
#define CAN_BAUDRATE_PRESC_037 0x09400
#define CAN_BAUDRATE_PRESC_038 0x09800
#define CAN_BAUDRATE_PRESC_039 0x09C00
#define CAN_BAUDRATE_PRESC_040 0x0A000
#define CAN_BAUDRATE_PRESC_041 0x0A400
#define CAN_BAUDRATE_PRESC_042 0x0A800
#define CAN_BAUDRATE_PRESC_043 0x0AC00
#define CAN_BAUDRATE_PRESC_044 0x0B000
#define CAN_BAUDRATE_PRESC_045 0x0B400
#define CAN_BAUDRATE_PRESC_046 0x0B800
#define CAN_BAUDRATE_PRESC_047 0x0BC00
#define CAN_BAUDRATE_PRESC_048 0x0C000
#define CAN_BAUDRATE_PRESC_049 0x0C400
#define CAN_BAUDRATE_PRESC_050 0x0C800
#define CAN_BAUDRATE_PRESC_051 0x0CC00
#define CAN_BAUDRATE_PRESC_052 0x0D000
#define CAN_BAUDRATE_PRESC_053 0x0D400
#define CAN_BAUDRATE_PRESC_054 0x0D800
#define CAN_BAUDRATE_PRESC_055 0x0DC00
#define CAN_BAUDRATE_PRESC_056 0x0E000
#define CAN_BAUDRATE_PRESC_057 0x0E400
#define CAN_BAUDRATE_PRESC_058 0x0E800
#define CAN_BAUDRATE_PRESC_059 0x0EC00
#define CAN_BAUDRATE_PRESC_060 0x0F000
#define CAN_BAUDRATE_PRESC_061 0x0F400
#define CAN_BAUDRATE_PRESC_062 0x0F800
#define CAN_BAUDRATE_PRESC_063 0x0FC00
#define CAN_BAUDRATE_PRESC_064 0x10000
#define CAN_BAUDRATE_PRESC_065 0x10400
#define CAN_BAUDRATE_PRESC_066 0x10800
```

```
#define CAN_BAUDRATE_PRESC_067 0x10C00
#define CAN_BAUDRATE_PRESC_068 0x11000
#define CAN_BAUDRATE_PRESC_069 0x11400
#define CAN_BAUDRATE_PRESC_070 0x11800
#define CAN_BAUDRATE_PRESC_071 0x11C00
#define CAN_BAUDRATE_PRESC_072 0x12000
#define CAN_BAUDRATE_PRESC_073 0x12400
#define CAN_BAUDRATE_PRESC_074 0x12800
#define CAN_BAUDRATE_PRESC_075 0x12C00
#define CAN_BAUDRATE_PRESC_076 0x13000
#define CAN_BAUDRATE_PRESC_077 0x13400
#define CAN_BAUDRATE_PRESC_078 0x13800
#define CAN_BAUDRATE_PRESC_079 0x13C00
#define CAN_BAUDRATE_PRESC_080 0x14000
#define CAN_BAUDRATE_PRESC_081 0x14400
#define CAN_BAUDRATE_PRESC_082 0x14800
#define CAN_BAUDRATE_PRESC_083 0x14C00
#define CAN_BAUDRATE_PRESC_084 0x15000
#define CAN_BAUDRATE_PRESC_085 0x15400
#define CAN_BAUDRATE_PRESC_086 0x15800
#define CAN_BAUDRATE_PRESC_087 0x15C00
#define CAN_BAUDRATE_PRESC_088 0x16000
#define CAN_BAUDRATE_PRESC_089 0x16400
#define CAN_BAUDRATE_PRESC_090 0x16800
#define CAN_BAUDRATE_PRESC_091 0x16C00
#define CAN_BAUDRATE_PRESC_092 0x17000
#define CAN_BAUDRATE_PRESC_093 0x17400
#define CAN_BAUDRATE_PRESC_094 0x17800
#define CAN_BAUDRATE_PRESC_095 0x17C00
#define CAN_BAUDRATE_PRESC_096 0x18000
#define CAN_BAUDRATE_PRESC_097 0x18400
#define CAN_BAUDRATE_PRESC_098 0x18800
#define CAN_BAUDRATE_PRESC_099 0x18C00
#define CAN_BAUDRATE_PRESC_100 0x19000
#define CAN_BAUDRATE_PRESC_101 0x19400
#define CAN_BAUDRATE_PRESC_102 0x19800
#define CAN_BAUDRATE_PRESC_103 0x19C00
#define CAN_BAUDRATE_PRESC_104 0x1A000
#define CAN_BAUDRATE_PRESC_105 0x1A400
#define CAN_BAUDRATE_PRESC_106 0x1A800
#define CAN_BAUDRATE_PRESC_107 0x1AC00
#define CAN_BAUDRATE_PRESC_108 0x1B000
#define CAN_BAUDRATE_PRESC_109 0x1B400
#define CAN_BAUDRATE_PRESC_110 0x1B800
#define CAN_BAUDRATE_PRESC_111 0x1BC00
#define CAN_BAUDRATE_PRESC_112 0x1C000
#define CAN_BAUDRATE_PRESC_113 0x1C400
```

```
#define CAN_BAUDRATE_PRESC_114 0x1C800
#define CAN_BAUDRATE_PRESC_115 0x1CC00
#define CAN_BAUDRATE_PRESC_116 0x1D000
#define CAN_BAUDRATE_PRESC_117 0x1D400
#define CAN_BAUDRATE_PRESC_118 0x1D800
#define CAN_BAUDRATE_PRESC_119 0x1DC00
#define CAN_BAUDRATE_PRESC_120 0x1E000
#define CAN_BAUDRATE_PRESC_121 0x1E400
#define CAN_BAUDRATE_PRESC_122 0x1E800
#define CAN_BAUDRATE_PRESC_123 0x1EC00
#define CAN_BAUDRATE_PRESC_124 0x1F000
#define CAN_BAUDRATE_PRESC_125 0x1F400
#define CAN_BAUDRATE_PRESC_126 0x1F800
#define CAN_BAUDRATE_PRESC_127 0x1FC00
#define CAN_BAUDRATE_PRESC_128 0x20000
#define CAN_BAUDRATE_PRESC_129 0x20400
#define CAN_BAUDRATE_PRESC_130 0x20800
#define CAN_BAUDRATE_PRESC_131 0x20C00
#define CAN_BAUDRATE_PRESC_132 0x21000
#define CAN_BAUDRATE_PRESC_133 0x21400
#define CAN_BAUDRATE_PRESC_134 0x21800
#define CAN_BAUDRATE_PRESC_135 0x21C00
#define CAN_BAUDRATE_PRESC_136 0x22000
#define CAN_BAUDRATE_PRESC_137 0x22400
#define CAN_BAUDRATE_PRESC_138 0x22800
#define CAN_BAUDRATE_PRESC_139 0x22C00
#define CAN_BAUDRATE_PRESC_140 0x23000
#define CAN_BAUDRATE_PRESC_141 0x23400
#define CAN_BAUDRATE_PRESC_142 0x23800
#define CAN_BAUDRATE_PRESC_143 0x23C00
#define CAN_BAUDRATE_PRESC_144 0x24000
#define CAN_BAUDRATE_PRESC_145 0x24400
#define CAN_BAUDRATE_PRESC_146 0x24800
#define CAN_BAUDRATE_PRESC_147 0x24C00
#define CAN_BAUDRATE_PRESC_148 0x25000
#define CAN_BAUDRATE_PRESC_149 0x25400
#define CAN_BAUDRATE_PRESC_150 0x25800
#define CAN_BAUDRATE_PRESC_151 0x25C00
#define CAN_BAUDRATE_PRESC_152 0x26000
#define CAN_BAUDRATE_PRESC_153 0x26400
#define CAN_BAUDRATE_PRESC_154 0x26800
#define CAN_BAUDRATE_PRESC_155 0x26C00
#define CAN_BAUDRATE_PRESC_156 0x27000
#define CAN_BAUDRATE_PRESC_157 0x27400
#define CAN_BAUDRATE_PRESC_158 0x27800
#define CAN_BAUDRATE_PRESC_159 0x27C00
#define CAN_BAUDRATE_PRESC_160 0x28000
```

```
#define CAN_BAUDRATE_PRESC_161 0x28400
#define CAN_BAUDRATE_PRESC_162 0x28800
#define CAN_BAUDRATE_PRESC_163 0x28C00
#define CAN_BAUDRATE_PRESC_164 0x29000
#define CAN_BAUDRATE_PRESC_165 0x29400
#define CAN_BAUDRATE_PRESC_166 0x29800
#define CAN_BAUDRATE_PRESC_167 0x29C00
#define CAN_BAUDRATE_PRESC_168 0x2A000
#define CAN_BAUDRATE_PRESC_169 0x2A400
#define CAN_BAUDRATE_PRESC_170 0x2A800
#define CAN_BAUDRATE_PRESC_171 0x2AC00
#define CAN_BAUDRATE_PRESC_172 0x2B000
#define CAN_BAUDRATE_PRESC_173 0x2B400
#define CAN_BAUDRATE_PRESC_174 0x2B800
#define CAN_BAUDRATE_PRESC_175 0x2BC00
#define CAN_BAUDRATE_PRESC_176 0x2C000
#define CAN_BAUDRATE_PRESC_177 0x2C400
#define CAN_BAUDRATE_PRESC_178 0x2C800
#define CAN_BAUDRATE_PRESC_179 0x2CC00
#define CAN_BAUDRATE_PRESC_180 0x2D000
#define CAN_BAUDRATE_PRESC_181 0x2D400
#define CAN_BAUDRATE_PRESC_182 0x2D800
#define CAN_BAUDRATE_PRESC_183 0x2DC00
#define CAN_BAUDRATE_PRESC_184 0x2E000
#define CAN_BAUDRATE_PRESC_185 0x2E400
#define CAN_BAUDRATE_PRESC_186 0x2E800
#define CAN_BAUDRATE_PRESC_187 0x2EC00
#define CAN_BAUDRATE_PRESC_188 0x2F000
#define CAN_BAUDRATE_PRESC_189 0x2F400
#define CAN_BAUDRATE_PRESC_190 0x2F800
#define CAN_BAUDRATE_PRESC_191 0x2FC00
#define CAN_BAUDRATE_PRESC_192 0x30000
#define CAN_BAUDRATE_PRESC_193 0x30400
#define CAN_BAUDRATE_PRESC_194 0x30800
#define CAN_BAUDRATE_PRESC_195 0x30C00
#define CAN_BAUDRATE_PRESC_196 0x31000
#define CAN_BAUDRATE_PRESC_197 0x31400
#define CAN_BAUDRATE_PRESC_198 0x31800
#define CAN_BAUDRATE_PRESC_199 0x31C00
#define CAN_BAUDRATE_PRESC_200 0x32000
#define CAN_BAUDRATE_PRESC_201 0x32400
#define CAN_BAUDRATE_PRESC_202 0x32800
#define CAN_BAUDRATE_PRESC_203 0x32C00
#define CAN_BAUDRATE_PRESC_204 0x33000
#define CAN_BAUDRATE_PRESC_205 0x33400
#define CAN_BAUDRATE_PRESC_206 0x33800
#define CAN_BAUDRATE_PRESC_207 0x33C00
```

```
#define CAN_BAUDRATE_PRESC_208 0x34000
#define CAN_BAUDRATE_PRESC_209 0x34400
#define CAN_BAUDRATE_PRESC_210 0x34800
#define CAN_BAUDRATE_PRESC_211 0x34C00
#define CAN_BAUDRATE_PRESC_212 0x35000
#define CAN_BAUDRATE_PRESC_213 0x35400
#define CAN_BAUDRATE_PRESC_214 0x35800
#define CAN_BAUDRATE_PRESC_215 0x35C00
#define CAN_BAUDRATE_PRESC_216 0x36000
#define CAN_BAUDRATE_PRESC_217 0x36400
#define CAN_BAUDRATE_PRESC_218 0x36800
#define CAN_BAUDRATE_PRESC_219 0x36C00
#define CAN_BAUDRATE_PRESC_220 0x37000
#define CAN_BAUDRATE_PRESC_221 0x37400
#define CAN_BAUDRATE_PRESC_222 0x37800
#define CAN_BAUDRATE_PRESC_223 0x37C00
#define CAN_BAUDRATE_PRESC_224 0x38000
#define CAN_BAUDRATE_PRESC_225 0x38400
#define CAN_BAUDRATE_PRESC_226 0x38800
#define CAN_BAUDRATE_PRESC_227 0x38C00
#define CAN_BAUDRATE_PRESC_228 0x39000
#define CAN_BAUDRATE_PRESC_229 0x39400
#define CAN_BAUDRATE_PRESC_230 0x39800
#define CAN_BAUDRATE_PRESC_231 0x39C00
#define CAN_BAUDRATE_PRESC_232 0x3A000
#define CAN_BAUDRATE_PRESC_233 0x3A400
#define CAN_BAUDRATE_PRESC_234 0x3A800
#define CAN_BAUDRATE_PRESC_235 0x3AC00
#define CAN_BAUDRATE_PRESC_236 0x3B000
#define CAN_BAUDRATE_PRESC_237 0x3B400
#define CAN_BAUDRATE_PRESC_238 0x3B800
#define CAN_BAUDRATE_PRESC_239 0x3BC00
#define CAN_BAUDRATE_PRESC_240 0x3C000
#define CAN_BAUDRATE_PRESC_241 0x3C400
#define CAN_BAUDRATE_PRESC_242 0x3C800
#define CAN_BAUDRATE_PRESC_243 0x3CC00
#define CAN_BAUDRATE_PRESC_244 0x3D000
#define CAN_BAUDRATE_PRESC_245 0x3D400
#define CAN_BAUDRATE_PRESC_246 0x3D800
#define CAN_BAUDRATE_PRESC_247 0x3DC00
#define CAN_BAUDRATE_PRESC_248 0x3E000
#define CAN_BAUDRATE_PRESC_249 0x3E400
#define CAN_BAUDRATE_PRESC_250 0x3E800
#define CAN_BAUDRATE_PRESC_251 0x3EC00
#define CAN_BAUDRATE_PRESC_252 0x3F000
#define CAN_BAUDRATE_PRESC_253 0x3F400
```

```
//#define CAN_BAUDRATE_PRESC_254 0x3F800 // nicht zulaessig,
da noch 2 addiert wird
//#define CAN_BAUDRATE_PRESC_255 0x3FC00 // nicht zulaessig,
da noch 2 addiert wird

//#define CAN_SYNC_JUMP_WIDTH_0 0x00000 // nicht zulaessig
#define CAN_SYNC_JUMP_WIDTH_1 0x00100
#define CAN_SYNC_JUMP_WIDTH_2 0x00200
#define CAN_SYNC_JUMP_WIDTH_3 0x00300

//#define CAN_TIME_SEGMENT1_00 0x00000 // nicht zulaessig
//#define CAN_TIME_SEGMENT1_01 0x00008 // nicht zulaessig
//#define CAN_TIME_SEGMENT1_02 0x00010 // nicht zulaessig
//#define CAN_TIME_SEGMENT1_03 0x00018 // nicht zulaessig
#define CAN_TIME_SEGMENT1_04 0x00020
#define CAN_TIME_SEGMENT1_05 0x00028
#define CAN_TIME_SEGMENT1_06 0x00030
#define CAN_TIME_SEGMENT1_07 0x00038
#define CAN_TIME_SEGMENT1_08 0x00040
#define CAN_TIME_SEGMENT1_09 0x00048
#define CAN_TIME_SEGMENT1_10 0x00050
#define CAN_TIME_SEGMENT1_11 0x00058
#define CAN_TIME_SEGMENT1_12 0x00060
#define CAN_TIME_SEGMENT1_13 0x00068
#define CAN_TIME_SEGMENT1_14 0x00070
#define CAN_TIME_SEGMENT1_15 0x00078
#define CAN_TIME_SEGMENT1_16 0x00080
#define CAN_TIME_SEGMENT1_17 0x00088
#define CAN_TIME_SEGMENT1_18 0x00090
#define CAN_TIME_SEGMENT1_19 0x00098
#define CAN_TIME_SEGMENT1_20 0x000A0
#define CAN_TIME_SEGMENT1_21 0x000A8
#define CAN_TIME_SEGMENT1_22 0x000B0
#define CAN_TIME_SEGMENT1_23 0x000B8
#define CAN_TIME_SEGMENT1_24 0x000C0
#define CAN_TIME_SEGMENT1_25 0x000C8
#define CAN_TIME_SEGMENT1_26 0x000D0
#define CAN_TIME_SEGMENT1_27 0x000D8
#define CAN_TIME_SEGMENT1_28 0x000E0
#define CAN_TIME_SEGMENT1_29 0x000E8
#define CAN_TIME_SEGMENT1_30 0x000F0
#define CAN_TIME_SEGMENT1_31 0x000F8

#define CAN_TIME_SEGMENT2_0 0x00000
#define CAN_TIME_SEGMENT2_1 0x00001
#define CAN_TIME_SEGMENT2_2 0x00002
#define CAN_TIME_SEGMENT2_3 0x00003
```

```
#define CAN_TIME_SEGMENT2_4 0x00004
#define CAN_TIME_SEGMENT2_5 0x00005
#define CAN_TIME_SEGMENT2_6 0x00006
#define CAN_TIME_SEGMENT2_7 0x00007

// 4. error_cnt Register
#define CAN_RX_ERROR_MASK 0x3fe00
#define CAN_TX_ERROR_MASK 0x001FF

// 10. error_code Register
#define CAN_ERROR_WARNING 0x20000
#define CAN_ERROR_DIRECTION 0x10000
#define CAN_ERROR_CAPTURE_MASK 0x07000
#define CAN_ERROR_FSM_MASK 0x001F0
#define CAN_NODE_ERROR_PASSIVE 0x00008
#define CAN_NODE_BUS_OFF 0x00004
#define CAN_RX_BUFFER_FULL_Q 0x00002
#define CAN_SEND_FAILED_Q 0x00001

// 11. BAUDRATE

// f = can_clk / ((BRP+2)*(1+tseg1+tseg2+2)) = 28000 kHz /
((0+2)*(1+10+1+2)) = 28000 kHz / (2*14) = 1000 kHz
// (100 - 87,5)% = 12,5% ‹(tseg2+2)% = (tseg2+2)/
(1+tseg1+tseg2+2)*100%
// (100 - 75,0)% = 25,0%› (tseg2+2)% = (tseg2+2)/
(1+tseg1+tseg2+2)*100%

#define CAN_BAUDRATE_1000k CAN_BAUDRATE_PRESC_000|
CAN_SYNC_JUMP_WIDTH_1|CAN_TIME_SEGMENT1_10|
CAN_TIME_SEGMENT2_1 // 21,4%
#define CAN_BAUDRATE_500k CAN_BAUDRATE_PRESC_000|
CAN_SYNC_JUMP_WIDTH_2|CAN_TIME_SEGMENT1_21|
CAN_TIME_SEGMENT2_4 // 21,4%
#define CAN_BAUDRATE_250k CAN_BAUDRATE_PRESC_002|
CAN_SYNC_JUMP_WIDTH_2|CAN_TIME_SEGMENT1_21|
CAN_TIME_SEGMENT2_4 // 21,4%
#define CAN_BAUDRATE_125k CAN_BAUDRATE_PRESC_006|
CAN_SYNC_JUMP_WIDTH_2|CAN_TIME_SEGMENT1_21|
CAN_TIME_SEGMENT2_4 // 21,4%
#define CAN_BAUDRATE_100k CAN_BAUDRATE_PRESC_006|
CAN_SYNC_JUMP_WIDTH_3|CAN_TIME_SEGMENT1_26|
CAN_TIME_SEGMENT2_6 // 22,9%
#define CAN_BAUDRATE_50k CAN_BAUDRATE_PRESC_014|
CAN_SYNC_JUMP_WIDTH_3|CAN_TIME_SEGMENT1_26|
CAN_TIME_SEGMENT2_6 // 22,9%
```

```
#define CAN_BAUDRATE_20k CAN_BAUDRATE_PRESC_038|
CAN_SYNC_JUMP_WIDTH_3|CAN_TIME_SEGMENT1_26|
CAN_TIME_SEGMENT2_6 // 22,9%
#define CAN_BAUDRATE_10k CAN_BAUDRATE_PRESC_078|
CAN_SYNC_JUMP_WIDTH_3|CAN_TIME_SEGMENT1_26|
CAN_TIME_SEGMENT2_6 // 22,9%
#define CAN_BAUDRATE_5k CAN_BAUDRATE_PRESC_158|
CAN_SYNC_JUMP_WIDTH_3|CAN_TIME_SEGMENT1_26|
CAN_TIME_SEGMENT2_6 // 22,9%


// I/O-Register Strucktur des CAN-Interface
typedef struct can {
volatile unsigned int work;
volatile unsigned int choose_config;
volatile unsigned int bustiming;
volatile unsigned int error_cnt;
volatile unsigned int warn_state;
volatile unsigned int acf_select;
volatile unsigned int acf_ac1;
volatile unsigned int acf_am1;
volatile unsigned int acf_ac2;
volatile unsigned int acf_am2;
volatile unsigned int error_code;
volatile unsigned int tx_succ;
volatile unsigned int rx_succ;
volatile unsigned int tx_buffer;
volatile unsigned int rx_buffer;
} can_regs_t;

// Datenpuffer Strucktur des CAN-Interface im DMA Puffer
typedef struct can_dma {
volatile unsigned int data_tx00[8];
volatile unsigned int data_tx01[8];
volatile unsigned int data_tx02[8];
volatile unsigned int data_tx03[8];
volatile unsigned int data_tx04[8];
volatile unsigned int data_tx05[8];
volatile unsigned int data_tx06[8];
volatile unsigned int data_tx07[8];
volatile unsigned int data_tx08[8];
volatile unsigned int data_tx09[8];
volatile unsigned int data_tx10[8];
volatile unsigned int data_tx11[8];
volatile unsigned int data_tx12[8];
volatile unsigned int data_tx13[8];
volatile unsigned int data_tx14[8];
```

```
volatile unsigned int data_tx15[8];
volatile unsigned int data_tx16[8];
volatile unsigned int data_tx17[8];

volatile unsigned int data_rx00[8];
volatile unsigned int data_rx01[8];
volatile unsigned int data_rx02[8];
volatile unsigned int data_rx03[8];
volatile unsigned int data_rx04[8];
volatile unsigned int data_rx05[8];
volatile unsigned int data_rx06[8];
volatile unsigned int data_rx07[8];
volatile unsigned int data_rx08[8];
volatile unsigned int data_rx09[8];
volatile unsigned int data_rx10[8];
volatile unsigned int data_rx11[8];
volatile unsigned int data_rx12[8];
volatile unsigned int data_rx13[8];
volatile unsigned int data_rx14[8];
volatile unsigned int data_rx15[8];
volatile unsigned int data_rx16[8];
volatile unsigned int data_rx17[8];
} can_dma_t;

#endif
```

20.8.3. CAN C-Header for C-Funktion

```
#ifndef __CAN_H
#define __CAN_H

#include ‹peripherals/can.h›
#include ‹bitmagic.h›

void can_set_work(struct can *can_p, unsigned int wert);
void can_set_acf(struct can *can_p, unsigned int data1,
unsigned int maske1, unsigned int data2, unsigned int maske2,
unsigned int acf_sel);

#endif /*CAN_H_*/
```

21. Display Controller

The display controller is a periphal SpartanMC device for driving several types of displays. It is possible to control either a segment based or a pixel based display. For resource optimization both parts are seperated into independent controller modules selectable from the jConfig device menu. For a segment based LCD, a special circuit is required for connecting the SpartanMC FPGA to the device (see later). The pixel based display requires a circuit including elements for controlling the backlight and contrast voltage. Below, both controller parts are described in detail, starting with the segment dispaly controller.

21.1. Controller for segment based displays

This module makes it possible to control a segment based display with a user defined number of digits and segments. The required memory for storing the digit's segment assignments is already included (its content depends on the defined settings). From those segment assigments the signals for the display's multiplex driving are generated. In case the display is a liquid crystal display (LCD), the differing signal sequence for driving LCDs is generated accordingly. This requires a corresponding circuit for connecting the display as shown in the figure below. There, the micro controller's output is connected in the center of a voltage divider with two equal resistors. The divider itself is connected to operating voltage and ground.

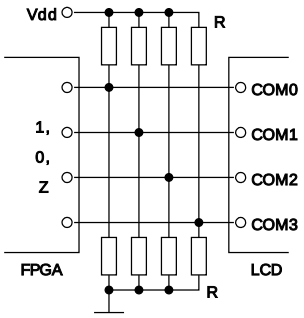


Figure 21-51: Circuit for connecting the LCD

If the display is not connected by using the given circuit, it will be damaged permanently! Because the dc voltage is not excluded.

21.1.1. Periphal registers

Table 21-79: Configuration registers of the segment display controller

Offset	Name	Description	Access	Initialization
0	REG_ENABLE	De-/Activates the display	read/ write	0x00000
1	REG_ADDR	Contains the adress of the digit to read from or write to.	read/ write	0x00000
2	REG_DATA	Contains the segment assignments according to the given digit address	read/ write	0x00000

Table 21-79: Configuration registers of the segment display controller

For accessing the segment memory the adress register must first be set to the correct digit number. Afterwards its current content can be read from or written to the data register.

21.1.2. Memory layout

The memory layout is slightly unconventional. This is caused by the required flexibility for configuring the number of segments/commons. Hence the data word is divided into the number of parts the display has commons. This allows the controller to compute the segment data sequentially for each common cycle instead of picking the required segment information out of the hole data word (which is quite complex to realize dynamically in hardware). The exact segment order depends on the used display. The first part of the data word (starting with bit 0) contains all assignments for the segments to be driven at common cycle 0, the next part for common cycle 1 and so on. For a 14 segment display with 4 commons the data word would look like "mPnd lcke gbjf haiS", where each letter represents one segment according to the typical segment order of such a display.

21.1.3. Module parameters

Table 21-80: Parameters of the segment display controller

Parameter	Description
SYSTEM_FREQUENCY	Clock the system is currently driven by (for example 25 MHz)

Parameter	Description
SEGMENT_FREQUENCY	Specifies the frequency for driving a single segment (see display's specification)
NUMBER_OF_COMMONS	Number of the display's common connections (number of anodes for LED displays)
NUMBER_OF_SEGMENTS	Number of segment connection for the whole display
NUMBER_OF_DIGITS	Number of the display's equal digits
BIT_PER_DIGIT	Width of a single memory word inside the segment memory (usually equal the the number of segments per digit)
IS_LCD	Set to 1 for a LCD, 0 for a LED display

Table 21-80: Parameters of the segment display controller

21.2. Controller for pixel based displays

This part of the display controller allows the driving of almost any pixel based displays. Therefore it provides the required memories. Depending on the module's configuration the operation of a graphic and a text mode is possible whereas both modes run independly. Inside the text mode a blinking cursor is displayed whose appearance can be defined by the user. The required codepage for converting the character codes to according pixel data can also be changed by the user. By default the codepage is initialized during the synthesis of the design with the "codepage 437", known from the original IBM PC. This initialization is defined in the given user constraint file (UCF). Inside the graphic mode there are several hardware accelerated functions for accessing the video memory (e.g. SetPixel or Line).

21.2.1. Periphal registers

Table 21-81: Configuration register of the matrix display controller

Offset	Name	Description	Access	Initialization
0	REG_DISPLAY_STATUS	Status register of the whole controller (see below)	read/ write	0x00100
1	REG_TEXT_COLOR	Foreground color of the displayed text. This color must come from the display's color space.	read/ write	0x0000F (white)
2	REG_TEXT_BGCOLOR	Background color of the displayed text	read/ write	0x00000

Offset	Name	Description	Access	Initialization
3	REG_TEXT_CHARPOS	Contains the coordinates of the currently drawed character. This may be important in relation to the CharLine interrupt described later.	read	0x00000
4	REG_TEXT_CURSORPOS	Position of the blinking cursor in text mode	read/ write	0x00000
5	REG_GRAPH_COORDSELECT	Register for addressing a single coordinate for the graphic functions	read/ write	0x00000
6	REG_GRAPH_COORDVALUE	Value of the selected coordinate	read/ write	0x00000
7	REG_GRAPH_COLOR	Color for a graphic function. This is a color coming from the color space of the memory (color LUT)	read/ write	0x00000

Table 21-81: Configuration register of the matrix display controller

21.2.2. Assembly of the register REG_DISPLAYSTATUS

Table 21-82: Register REG_DISPLAYSTATUS

Bit	Name	Description	Access	Initialization
0	STATUSBIT_DISP_BACKLIGHT	Turns the backlight on or off.	read/ write	0
1	STATUSBIT_DISP_ON	De-/activates the display.	read/ write	0
2	STATUSBIT_TEXTMODE	De-/activates the text mode, if implemented.	read/ write	0
3-5	STATUSBIT_FUNCTION_SELECT	Selects a hardware accelerated graphic function.	read/ write	000
6	STATUSBIT_FUNCTION_FLAG1	Optional flag for the graphic functions	read/ write	0
7	STATUSBIT_FUNCTION_DRAW	Starts the selected graphic function with the given parameters. After	read/ write	0

Bit	Name	Description	Access	Initialization
		setting the bit, it will be reset in the next clock cycle.		
8	STATUSBIT_FUNCTION_READY	Indicated if the currently selected graphic function is ready.	read	1
9	STATUSBIT_OVERLAY	De/activates the overlay	read/ write	0

Table 21-82: Register REG_DISPLAYSTATUS

21.2.3. Assembly of REG_TEXT_CHARPOS and REG_TEXT_CURSORPOS

Table 21-83: Registers REG_TEXT_CHARPOS and REG_TEXT_CURSORPOR

Bit	Name	Description
0 to 8	Y	Y coordinate in the text mode
9 to 17	X	X coordinate in the text mode

Table 21-83: Registers REG_TEXT_CHARPOS and REG_TEXT_CURSORPOR

21.2.4. Interrupts

Table 21-84: Interrupts of the matrix display controller

Interrupt	Description
charLine_ir	Indicates that the drawing of one charcter's pixel line in text mode has completed. This makes it possible to change the color settings for the following character lines for example.
graphFunctionReady_ir	This interrupt is triggered when a graphic function gets ready.

Table 21-84: Interrupts of the matrix display controller

21.2.5. Coding of the graphic functions

The following coding is defined in the file "display_graph_common.v" and should be changed there if required.

Table 21-85: Implemented graphic functions

Number	Function	Macro name
0	invalid	-
1	SetPixel	FUNCTION_SETPIXEL
2	Line	FUNCTION_LINE
3	CopyRect	FUNCTION_COPYRECT
4	GetPixel	FUNCTION_GETPIXEL
5	FillRect	FUNCTION_FILLRECT

Table 21-85: Implemented graphic functions

21.2.6. Memory layouts

Codepage The codepage is a continous memory containing the pixel data of every displayable character. One memory word contains a single pixel line of a character. Thus one character requires a certain number of memory words depending on the configuration (by default 16). The memory offset O of one character is calculated by O = C * H, where C is the character code an H the configured number of lines per character.

Text cursor The layout of the cursor memory is equal to the one of the codepage but contains space for only one character.

Graphic memory Depending on the configuration one word of the graphic memory contains data for several pixels. Therein the MSB represents the most left and the LSB the most right pixel (almost like little endian). Usually the user has no need to access the graphic memory directly since there are functions like SetPixel and GetPixel.

Color LUT The color look up table (color LUT) converts the reduced color space inside the graphic memory to the display's color space. Thus the offset inside the color LUT represents a color of the graphic memory. From this offset the color LUT offers the corresponding color for the display.

21.2.7. Module parameters

Table 21-86: Parameters of the matrix display controller

Name	Description
BASE_ADDR	Base address on which the controller communicates with the SpartanMC
GRAPHMEM_BASE_ADDR	Base address on which the graphic memory is connected. The memory should be placed behind the space for I/O devices in any case. This is caused through its size, where otherwise some I/O device may be activated accidentally.

Name	Description
TEXTMEM_BASE_ADDR	Base address of the text memory
CODEPAGE_BASE_ADDR	Base address of the codepage
COLOR_LUT_BASE_ADDR	Base address of the color LUT
CURSORMEM_BASE_ADDR	Base address of the text cursor's memory
SCREEN_WIDTH	Display's width in pixels
SCREEN_HEIGHT	Display's height in pixels
DATA_WORD_WIDTH	Width of one data word transmitted to the display
BIT_PER_PIXEL	Color depth of one pixel on the display
BIT_PER_MEMORY_PIXEL	Color depth of one pixel in the graphic memory
TIMING_INIT_CLOCKS_TO_VCON_ON	Number of clocks to wait after a reset until the display's contrast voltage is activated.
TIMING_INIT_CLOCKS_TO_DISPOFF	Number of clocks to wait after a reset until the display itself is activated.
TIMING_CL1_CLOCKS_AFTER_RESET	Number of clocks to wait after a reset until a row cycle begins.
TIMING_CL1_CLOCKS_HIGH	Number of clocks the row clock is high.
TIMING_CL1_CLOCKS_LOW	Number of clocks the row clock is low.
TIMING_CL1_BEGIN_HIGH_LOW	Indicates whether a row on the display starts with the falling (1) or the rising edge (0) of the row clock.
TIMING_ROW_BREAK_DELAY	Number of clocks to wait after transmitting one row before the transmission of the next row starts.
TIMING_SCREEN_FINISH_DELAY	Number of clocks to wait after transmitting one complete screen before the transmission of the next screen continues.
TIMING_CL2_CLOCKS_AFTER_RESET	Number of clocks to wait after a reset until a data clock cycle begins.
TIMING_CL2_CLOCKS_HIGH	Number of clocks the data clock is high.
TIMING_CL2_CLOCKS_LOW	Number of clocks the data clock is low.
TIMING_FRAMESTART_CLOCKS_AFTER_RESET	Number of clocks to wait after a reset until the framestart cycle begins.
TIMING_FRAMESTART_CLOCKS_HIGH	Number of clocks the framestart signal is high.
TIMING_FRAMESTART_CLOCKS_LOW	Number of clocks the framestart signal is low.
TIMING_INVERT_CLOCKS_AFTER_RESET	Number of clocks to wait after a reset until the invert signal cycle begins.
TIMING_INVERT_CLOCKS_TOGGLE	Number of clocks after the invertsignal is inverted.
CODEPAGE_CHAR_WIDTH	Width of one character in pixles
CODEPAGE_CHAR_HEIGHT	Height of one character in pixels

Name	Description
CODEPAGE_SIZE	Number of characters in the codepage
FPGA_BRAM_SIZE	Number of words the used Block RAM can save
USE_TEXT_MODE	Defines whether the text mode is included in synthesis.
USE_GRAPH_MODE	Defines whether the graphic mode is included in synthesis.
USE_GRAPHFUNCTION_SETPIXEL	Defines whether the graphic function "SetPixel" is included in synthesis.
USE_GRAPHFUNCTION_LINE	Defines whether the graphic function "Line" is included in synthesis.
USE_GRAPHFUNCTION_COPYRECT	Defines whether the graphic function "CopyRect" is included in synthesis.
USE_GRAPHFUNCTION_GETPIXEL	Defines whether the graphic function "GetPixel" is included in synthesis.
USE_GRAPHFUNCTION_FILLRECT	Defines whether the graphic function "FillRect" is included in synthesis.

Table 21-86: Parameters of the matrix display controller

22. Core connector for multicore systems

The core connector implements a simple FIFO through which two SpartanMC cores are able to communicate. Therefor the modules master core connector as data transmitter and slave core connector as data receiver are available for unidirectional communication. The module duplex core connector provides an interface for bidirectional communication containing one master and one slave core connector.

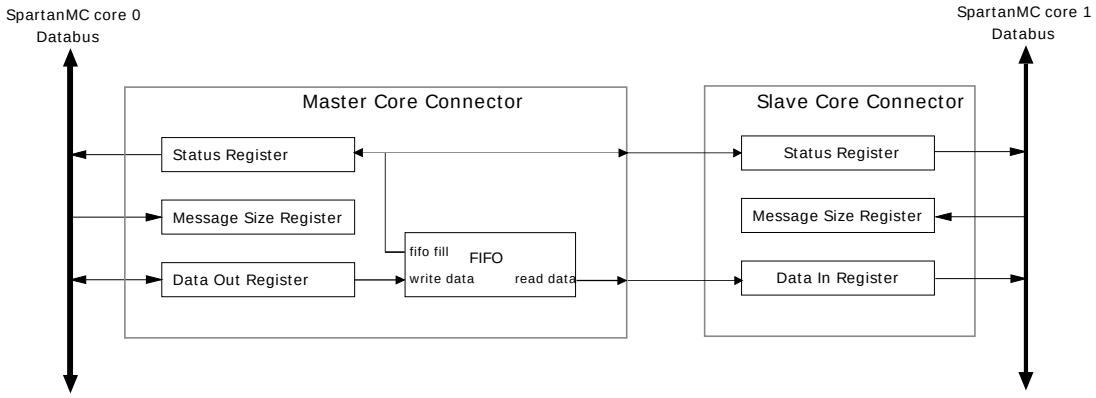


Figure 22-52: Unidirectional core connector

22.1. Module Parameters

Parameter	Default Value	Descripton
FIFO_WIDTH	18	FIFO width of the connector. It is highly recommendet to use the default value for optimal utilization of the FPGA structures.
FIFO_DEPTH	16	Amount of buffer registers used.

Table 22-87: Module parameters

22.2. Peripheral Registers

The slave and master core connectors have three registers each for message transfer:

22.2.1. STATUS Register Description

The status register tells if the FIFO is empty or full and if there is enough space left for another message. The possible return arguments are as follows:

Value	Description
0	FIFO has at least MSG_SIZE entries free.
1	FIFO is empty/No messages are stored.
2	FIFO is full or has fewer free entries than specified in the MSG_SIZE register.

Table 22-88: STATUS states

22.2.2. MSG_SIZE Register Description

In the message size register should be written the size of the message to be written/read. It is needed by the status register to correctly signal whether the FIFO is full or empty. It does not influence the written or read data. Set to 1 by default.

22.2.3. DATA_OUT Register Description

This register is used by the master core connector to write data in. The data is handed to the slave core connectors data in register.

22.2.4. DATA_IN Register Description

This register is used by the slave core connector to read data from the master core connector.

22.3. Usage examples

22.3.1. Register level access

For sending data the registers can directly be accessed through simplification of a basic c wrapper. The following code snipplets will clarify the basic usage.

22.3.2. Master core connector C-Header for Register description

```
#ifndef __MASTER_CORE_CONNECTOR_H
#define __MASTER_CORE_CONNECTOR_H
#define MASTER_FIFO_FULL 0x2
#define MASTER_FIFO_EMPTY 0x1

typedef struct {
   volatile unsigned int status;
   volatile unsigned int msg_size;
   volatile unsigned int data_out;
} master_core_connector_regs_t;

#endif
```

22.3.3. Slave core connector C-Header for Register description

```
#ifndef __SLAVE_CORE_CONNECTOR_H
#define __SLAVE_CORE_CONNECTOR_H
#define SLAVE_FIFO_FULL 0x2
#define SLAVE_FIFO_EMPTY 0x1

typedef struct {
   volatile unsigned int status;
   volatile unsigned int msg_size;
   volatile unsigned int data_in;
} slave_core_connector_regs_t;

#endif
```

22.3.4. Master and Slave core connector usage at C-Level

```
master_core_connector_regs_t *master= MASTER_CORE_CONNECTOR_0;
slave_core_connector_regs_t *slave= SLAVE_CORE_CONNECTOR_0;
while(i < 18 && (*master).status < 2 ){
```

```
   (*master).data_out=i;
   i++;
   if((*slave).status==2){
       while((*slave).status!=1){
           printf("data received %u \n",(*slave).data_in);
       }
   }
}
```

22.3.5. Duplex core connector usage at C-Level

```
duplex_core_connector_regs_t *duplex= DUPLEX_CORE_CONNECTOR_0;
master_core_connector_regs_t *master= duplex;
slave_core_connector_regs_t *slave= duplex
+DUPLEX_CONNECTOR_SLAVE_OFFSET;
```

22.3.6. C library

To communicate at a higher abstraction level the mpsoc interface is offered. It offeres various functions for blocking and non-blocking send and receive.

22.3.7. MPSoC API usage

```
#ifndef __MPSOC_H
#define __MPSOC_H

#define CORE_CONN_OK 0
#define CORE_CONN_FULL -1
#define CORE_CONN_EMPTY -2

void spmc_wait_free(void *conn, const unsigned int count);
void spmc_send(void *conn, const unsigned int value);
int spmc_send_nb(void *conn, const unsigned int value);
void spmc_send_block(void *conn, const void *data, unsigned
int count);
int spmc_send_block_nb(void *conn, const void *data, unsigned
int count);

void spmc_wait_data(void *conn, const unsigned int count);
unsigned int spmc_receive(void *conn);
int spmc_receive_nb(void *conn, unsigned int *value);
void spmc_receive_block(void *conn, const void *data, unsigned
int count);
```

```
int spmc_receive_block_nb(void *conn, const void *data,
unsigned int count);

#endif
```

23. Real Time Operating System

The SpartanMc project contains a small Real Time Operating System that can be used to easily run multiple tasks in parallel. The tasks are scheduled based on their priority and can be synchronized by using semaphores.

23.1. Concepts

To avoid having to save all registers a task uses, every task is assigned a part of the register file. The size of this part has to be set at task creation by specifying the number of call levels in the task. Additionally, every task is assigned a part of memory for its stack.

Note: Neither stack nor registers are range checked. It is up to the application programmer to ensure that a task does not overwrite data outside of its assigned resources.

Scheduling is based on priorities. When a task with higher priority than the currently running task gets ready, it is immediately scheduled.

Note: Do not switch tasks from within an ISR when using the complex interrupt controller. In that case the end of the ISR is not correctly signaled to the interrupt controller, which makes it ignore any future interrupts of the same or lower priority.

Note: The RTOS is not currently compatible with *make bitgen* . You always have to run *make all* to compile new firmware.

23.2. Preparing the Firmware

To link the RTOS into the firmware, edit the file *config-build.mk* in the firmware directory. In the list of libraries, remove *startup* and add *rtos* . If interrupt support is needed, also add *rtos_interrupt* .

The main source file needs to define these *int* variables to tell the RTOS how to initialize the main task:

Table 23-89: Needed variables for initialization of RTOS

Variable	Description
main_task_priority	Priority of the main task
main_task_max_call_level	Maximum call level in the main task

Variable	Description
main_task_stack_size	Maximum stack size in the main task
isr_max_call_level	Maximum call level in interrupt service routines
isr_max_stack_usage	Maximum stack size in interrupt service routines

23.3. Task management

23.3.1. create_task

```
task_t create_task( void ( *entry ) (void * param), void
*param, uint18_t priority, uint18_t max_call_level, size_t
stack_size )
```

Create a new task. Returns a representation of the newly created task or *NULL* if the operation failed due to insufficient memory or insufficient free space in the register file.

Table 23-90: Parameters of create_task

Parameter	Description
entry	Entry Point of the task to create
param	Parameter to call *entry* with
priority	Priority of the task to create
max_call_level	Maximum call level for the task to create. Note that interrupts occurring during the execution of the task also need to be counted. The entry function itself is not to be counted in this value.
stack_size	Stack size of the task to create

Table 23-91: Info about create_task

Callable by ISR	No
Internal call depth	5
with active interrupts	4

23.3.2. delete_task

```
void delete_task( task_t task )
```

Deletes a task. If a task wants to delete itself, its memory is not freed immediately but later when the idle task is scheduled.

Table 23-92: Parameters of delete_task

Parameter	Description
task	The task to delete, as returned by create_task

Table 23-93: Info about delete_task

Callable by ISR	No
Internal call depth	5
with active interrupts	4

23.3.3. suspend_task

`void suspend_task( task_t task )`
Suspend a task. It can be resumed by calling resume_task.

Table 23-94: Parameters of suspend_task

Parameter	Description
task	The task to suspend, as returned by create_task

Table 23-95: Info about suspend_task

Callable by ISR	No
Internal call depth	3
with active interrupts	2

23.3.4. resume_task

`void resume_task( task_t task )`
Resume a previously suspended task.

Table 23-96: Parameters of resume_task

Parameter	Description
task	The task to resume, as returned by create_task

Table 23-97: Info about resume_task

Callable by ISR	No
Internal call depth	3
with active interrupts	2

23.3.5. get_current_task

`task_t get_current_task( )`
Return a representation of the current task.

Table 23-98: Info about get_current_task

Callable by ISR	Yes
Internal call depth	1
with active interrupts	1

23.3.6. forbid_preemption

`void forbid_preemption( )`
Forbid the preemption of the current task to mark critical sections. Event tasks with higher priority than the current task will not get scheduled until permit_preemption is called. Multiple calls to forbid_preemption are allowed to be able to nest critical sections. To allow preemption again, permit_preemption has to be called the same number of times.

Table 23-99: Info about forbid_preemption

Callable by ISR	No
Internal call depth	1
with active interrupts	1

23.3.7. permit_preemption

`void permit_preemption( )`
Permit the preemption of the current task. To allow preemption again, permit_preemption has to be called the same number of times as forbid_preemption.

Table 23-100: Info about permit_preemption

Callable by ISR	No
Internal call depth	1
with active interrupts	1

23.3.8. task_yield

`void task_yield( )`
Change to another task of the same priority if one is available.

Table 23-101: Info about task_yield

Callable by ISR	Yes
Internal call depth	3
with active interrupts	1

23.4. Semaphores

23.4.1. initialize_semaphore

`task_t create_task( semaphore_t *sem, uint18_t value )`
Initializes a semaphore. This function has to be called before using a semaphore for the first time. It cannot be used to change a semaphore's value while it is in use.

Table 23-102: Parameters of initialize_semaphore

Parameter	Description
sem	Pointer to the semaphore to initialize
value	The semaphore's initial value

Table 23-103: Info about initialize_semaphore

Callable by ISR	Yes (?)
Internal call depth	2
with active interrupts	2

23.4.2. semaphore_down

`void semaphore_down( semaphore_t *sem)`
Reduces the semaphore's value by one. If it already is zero, block the task until another task calls semaphore_up.

Table 23-104: Parameters of semaphore_down

Parameter	Description
sem	The semaphore

Table 23-105: Info about semaphore_down

Callable by ISR	No
Internal call depth	3
with active interrupts	2

23.4.3. semaphore_up

`void semaphore_up( semaphore_t *sem)`
Increases the semaphore's value by one. If there are threads blocked by this semaphore, wake one of them.

Table 23-106: Parameters of semaphore_up

Parameter	Description
sem	The semaphore

Table 23-107: Info about semaphore_up

Callable by ISR	No
Internal call depth	3
with active interrupts	1

23.5. Dynamic memory allocation

23.5.1. malloc

`void *malloc( size_t size)`
Allocate a block of memory.

Table 23-108: Parameters of malloc

Parameter	Description
size	the number of 9-bit-words to allocate.

Table 23-109: Info about malloc

Callable by ISR	No
Internal call depth	4
with active interrupts	3

23.5.2. free

`void free( void *ptr)`
Free a previously allocated block of memory.

Table 23-110: Parameters of free

Parameter	Description
ptr	Pointer to the memory block to free.

Table 23-111: Info about free

Callable by ISR	No
Internal call depth	4
with active interrupts	3

23.6. Example Code

This example code creates two threads that both output to an uart_light.

The hardware for this example is the same as in the quickstart guide: processor core, sysclk, and uart_light.

```
#include <system/peripherals.h>
#include <uart.h>
#include <stdio.h>
#include <rtos.h>

int main_task_priority = 1, main_task_max_call_level =
10, main_task_stack_size = 200, isr_max_call_level = 10,
isr_max_stack_usage = 200;

static void hello_task (void * param) {
   while (1) {
       printf("hello from task %d\n",param);
       task_yield();
   }
}

void main( void ) {
   stdio_uart_light_open(UART_LIGHT_0);

   create_task(hello_task, 1, 1, 5, 100);
   create_task(hello_task, 2, 1, 5, 100);

   suspend_task(get_current_task());
}
```

The line for linked libraries in *config-build.mk* looks like this:

```
LIB_OBJ_FILES:=rtos peri
```

MANPAGE – SPARTANMC(7)

NAME

spartanmc – Toolkit for easy implementation of custom SoCs (System-on-Chip) on Xilinx FPGAs

SYNOPSIS

Global targets:

make cablesetup [**+group**=*GROUP*]

make integrity_check +target=*PLATFORM*

make man [*MANPAGE*]

make newproject +path=*PATH*

make reconfigure

make setup

make unconfigure

make [**what**]

Project targets:

DESCRIPTION

The SpartanMC-SoC-Kit is a set of tools for implementing FPGA-based SoCs. The implementation process does not require knowledge about hardware description languages such as Verilog or VHDL.

Based on the 18-bit SpartanMC microprocessor core the SoC-Kit provides a toolchain to allow easy implementation of a custom System-on-Chip (SoC) on a Xilinx FPGA. The frontend used is GNU *make* to invoke a number of underlying backend tools.

To compose an SoC, the GUI-based system builder *jConfig* will generate a set of hardware source code and configuration files based on the users configuration choices. The SoC then is synthesized from this set of files by invoking the corresponding tools from Xilinx ISE Suite. The whole process is driven by *make* which finally generates a bitfile that can be downloaded to your target FPGA.

The configuration also includes system firmware which is written in C and will be embedded into the design during synthesis. An optional bootloader allows for later update of the software components without re-synthesizing the hardware.

MAKE TARGETS

All steps to design a SoC are triggered by *make*. This section describes all available operations implemented as make targets.

Global targets

Global targets are available from the installation directory of the SpartanMC-SoC-Kit. This directory is called *SPARTANMC_ROOT* .

cablesetup Creates a rules file (*.rules) understood by udev to ensure proper operation of the Xilinx programming tool *impact*. The rules will make sure our system loads the correct USB-firmware to enable the cable driver to detect your cable. If your system has restricted USB access, the option *GROUP* specifies which user group is granted access to the USB programming cable. If ommitted, the default group 'xilinx' is used. After running this target you have to copy the generated file to the proper place for udev-rules in order to take effect.

integrity_check Performs an integrity check on the SpartanMC installation. This will run an automated sequence covering most of the functionality of the SpartanMC-SoC-Kit. The sequence will start with calling **make unconfigure** to get a clean installation directory. The next actions cover all setup steps followed by the creation of a test project. Finally, this project will be synthesized. If *PLATFORM* specifies any other value than 'nohw', the design will be implemented on the corresponding target platform. Otherwise, the sequence will be complete after bitfile generation. To get a list of supported platforms, call **make integrity_check** without any option. If the described sequence completes, the SoC-Kit most likely is properly installed and configured. If not, there may be a configuration problem or a functional issue concerning the toolchain. The test sequence will abort with an error message in that case.

man Displays the SpartanMC manpage denoted by *MANPAGE*. Ommitting *MANPAGE* is equal to **make man spartanmc** and will show this manpage.

reconfigure Runs **configure** with the same options and relevant environment variables as the last time the configure was explicitely invoked via the command line.

setup Builds or updates all required components of the SpartanMC-SoC-Kit from the corresponsing sources. Components that any other make targets depend on are automatically built when invoking that target (e.g. manpages are generated from the users manual sources when invoking **make man**).

unconfigure Removes all files generated by **make** and **configure**. After running **make unconfigure** , your SpartanMC installation will be left in the same state as just after a fresh install.

what Shows a list containing all currently available targets and a short description. The availability of some targets may depend on your host system configuration or the current state of your SpartanMC installation or current project.

SEE ALSO

MANPAGE – SPARTANMC-HEADERS(7)

NAME

spartanmc-headers – SpartanMC header files for firmware development

DESCRIPTION

The various functions implemented in the SpartanMC C library (see *spartanmc-libs*) are defined in a number of header files located at **spartanmc/include/**. This path is part fo the standard include path of the SpartanMC-GCC. The following sections describe use and organization of the header files.

LIBRARY HEADER FILES

The following header files define general support functions and macros as well as support functions for access peripheral components:

bitmagic.h	Macros for bit manipulation such as set/clear/toggle bit at a certain memory location.
ddr.h	Support functions for the MCB (Memory Controller Block) in Xilinx devices (mcb_peri_interface). Requires *libperi*.
interrupt.h	Support functions for interrupt controller peripheral (intctrl, intctrl_p). Requires *libinterrupt* or *libinterrupt_p*.
led7.h	Support functions for a 7-Segment Display connected to the SFR_LEDS special function register of the processor core. Requires *libperi*.
mul_high.h	Support function to access the high order word (bits 19-36) of a multiplication.
sleep.h	Function to delay execution by a certain number of clock cycles.
stdint.h	Integer type definitions.
stdio.h	Standard input/output functions such as printf.
stdio_uart.h	Support functions to use UART with stdio functions.
stepper.h	Support functions for stepper motor peripheral (microstepper). Requires *libperi*.
string.h	String and memory operations.
uart.h	Support functions for UART peripherals (uart, uart_light). Requires *libperi*.
usb.h	Support functions for USB11 peripheral (usb11). Requires *libperi*.

GENERATED PROJECT HEADER FILES

To allow easy access to the hardware components of any generated system there are a number of header files generated by the system builder *jConfig*. For details, see **peripherals.h** and **hardware.h**.

FILES AND DIRECTORIES

spartanmc/include/	Header files for peripheral support functions
spartanmc/include/peripherals/	Header files defining structures and bit constants for peripheral register access

SEE ALSO

peripherals.h(3), **hardware.h**(3), *spartnmc-libs*(7)

AUTHORS

Written by Markus Vogt

MANPAGE – HARDWARE.H(3)

NAME

hardware.h – Header file populating hardware implementation parameters for low level hardware access

SYNOPSIS

#include <system/hardware.h>

DESCRIPTION

The project specific generated header file **hardware.h** populates a number of key-value pairs reflecting all synthesis parameters passed to the hardware implementation process. This allows the firmware to be aware of certain features and parameters concerning the hardware platform it runs on.

The actual keys available depend on the system configuration specified in the system builder *jConfig*. Basically, each value choosen at the tab *Parameters* of each hardware component is mapped to a **#define** using the form

#define <KEY> <VALUE>.

Refer to the hardware documentation for details about the actual parameters defined by a certain hardware component.

KEYS

The list below gives an overview about the general format of available keys populated by **hardware.h**.

SB_<instance_name>_<parameter_name>

Value of hardware parameter <parameter_name> of module instance <instance_name>. E.g., the base address (parameter *BASE_ADR*) of module *uart_0* would be **SB_UART_0_BASE_ADR**.

SBI_REGION_<instance_name>_<suffix>

Provides information about the address space occupied by a certain hardware module. This only applies to processor instances and peripheral modules implementing DMA. **<suffix>** can be one of **MIN_ADDR**, **MAX_ADDR** or **BYTES** respectively giving the lower

or upper address boundaries or the number of bytes occupied by the components memory.

SBI_VERSION System builder version, which is currently 2.

SBI_CORE_ID 18-bit hexadecimal checksum of the current hardware design. Used to match a given firmware binary against a certain hardware design when using the bootloader (see *spmc-loader*(1)).

i_bits The number of interrupt lines provided by the interrupt controller (*intctrl* or *intctrl_p*), if any. When no interrupt controller is present, the value for **i_bits** is 0.

VALUES

The form a value is represented depends on the parameters value type as defined by the system builder *jConfig* or read from the respective module description. All values are in fact mapped to integer constants. To deal with float and string values, symbolic constants are used.

Integer values

Decimal and hexadecimal integer values are represented straight forward as shown in *jConfig* (e.g. **23**, **0x42**). Binary numbers are represented using their respective hexadecimal notation.

Float values

Float value parameters defined by the system builder are represented by symbolic constants of the form **SBFLOAT_<int>_<frac>**. E.g, the float number **2.68** woule become **SBFLOAT_2_68**. Note that this technique only allows for test of equality regarding a certain parameter. Performing real float arithmetics is not possible, which rather is limited by the fact that the SpartanMC currently does not support floating point in any way.

Boolean values

Boolean values are represented by integers of value **0** or **1** respectively standing for **false** or **true**. Constants of the form **SBBOOL_...** map the symbolic representation for each boolean parameter to their respective numeric values **0** or **1**. E.g., a boolean parameter with the symbolic meaning of **YES** or **NO** will provide the constants **#define SBBOOL_YES 1** and **#define SBBOOL_NO 0**. This allows you to use symbolic constants similar to as shown in the system builder when testing for values of boolean parameters.

String values

String values are mapped to symbolic constants of the form **SBSTRING_<value>**, where **<value>** is replaced by the original string value in upper case. All characters not allowed in a constant name are replaced by a underscore (_). E.g., the value of parameter VENDOR_STRING = "TU

Dresden" of component *usb11_0* would become **#define SB_USB11_VENDOR_STRING SBSTRING_TU_DRESDEN**. The symbolic constant **SBSTRING_TU_DRESDEN** is mapped to an arbitrary unique hexadecimal value.

FILES

<project_dir>/system/hardware.h	Header file to include for access to hardware parameters
<project_dir>/system/<subsystem_name>/hardware.h	Actual header file defining hardware parameters for the respective subsystem. Included by **hardware.h** depending on the actual subsystem the firmware is built for.

SEE ALSO

peripherals.h(3)

AUTHORS

Written by Markus Vogt

MANPAGE – PERIPHERALS.H(3)

NAME

peripherals.h – Project header file for access to peripheral components

SYNOPSIS

#include <system/peripherals.h>

DESCRIPTION

For any SpartanMC project, the system builder *jConfig* generates a header file providing the interface to all peripheral components present in your system. Variables pointing to the respective I/O and/or DMA memory base addresses will be automatically provided for each peripheral instance.

For a particular peripheral instance the name of the variable will be the respective identifier as shown in the system builder *jConfig* converted to UPPER CASE (e.g. *UART_LIGHT_0*). Each such variable will be a typed pointer tailored to the register I/O space of the particular peripheral. In case a component offers DMA space you will get another variable named <PERIPHERAL_NAME>_DMA pointing to the peripherals DMA base address.

The header files defining the respective variable types can be found at **spartanmc/include/peripherals/** (see section below for details). All files required for your systems peripherals will be automatically included via **peripherals.h**.

IMPLEMENT CUSTOM PERIPHERALS

For each type of peripheral component a header file is required at **spartanmc/include/peripherals/** declaring the particular data types (e.g. a struct) for register I/O and DMA access. The header files name must be the same as the peripherals hardware type.

If you add a custom peripheral component to the SpartanMC-SoC-Kit make sure you provide the corresponding header file. For a component named e.g. *my_peri* a file named **my_peri.h** is required. Within this file the following type declarations are expected to be found:

typedef ... my_peri_regs_t; /* register space access */

typedef ... my_peri_dma_t; /* DMA space access */

Note that in case your peripheral does not implement registers or DMA space the respective type declaration may be ommitted. Basically, the interface to a peripheral component may be a pointer to an unsigned integer. In that case, the type definitions may look like the following:

typedef unsigned int my_peri_regs_t; /* register space access */

typedef unsigned int my_peri_dma_t; /* DMA space access */

Note that there is no explicit pointer- or array-like declaration. The point where the pointer comes in is at the variable instatiation in the generated header file.

To interface more complex peripherals it is wise declaring a structure with descriptive names for the particular registers. Additionally to the type declaration the header file may define bit constants to simplify bit wise access to the registers.

High level support functions operating on the peripherals registers and DMA space should be defined in arbitrary named header files located at **spartanmc/include**. The resepective implementation of such functions should be part of *libperi*, but could virtually be implemented in any other library.

FILES

<project_dir>/system/peripherals.h	Header file to include for access to generated peripheral variables
<project_dir>/system/<subsystem_name>/peripherals.h	Actual header file defining peripheral variables for the respective subsystem. Included by **peripherals.h** depending on the actual subsystem the firmware is built for.

SEE ALSO

hardware.h(3), *spartanmc-libs*(7)

AUTHORS

Written by Markus Vogt

MANPAGE – SPARTANMC-LIBS(7)

NAME

spartanmc-libs – SpartanMC software libraries and library build system

DESCRIPTION

The SpartanMC processor core comes with a number of supporting C libraries. The library functions are available for the firmware on the target system. Some library code is essential such as startup code or interrupt handling routines. The remainder of the library offers functions to the user to simplify access to peripheral components.

To use functions from a certain library, the corresponding header file must be included (see *spartanmc-headers*) in your source code and the linker must be told to include the library into the firmware binary. The latter is achieved by adding the library name to the list of link libraries in *config-build.mk* in the firmware directory. Note that pointer variables for access to peripheral registers are available by including *peripherals.h* (see **peripherals.h**).

To optimize the size of the resulting firmware binary, each library function is implemented in a separate source file. This allows the linker to remove all functions that are never called.

LIBRARIES

The following libraries are currently available for the SpartanMC processor core:

libc System calls like printf, sleep, etc. Automatically linked with each project.

libgcc Support function for the compiler. Automatically linked with each project.

libinterrupt Required interrupt handler and support functions for default interrupt controller (intctrl). Could not be used together with **libinterrupt_p**.

libinterrupt_p Required interrupt handler and interrupt support functions for interrupt controller with priority (intctrl_p). Could not be used together with **libinterrupt**.

libperi Support functions for various peripheral components such as USB, UART and others (also see **peripherals.h**).

libstartup Default startup code for the processor core. This library is included by default for each new project. Could be replaced by **libstartup_loader**.

libstartup_loader Startup code with support for firmware update via UART using a boot loader. For more details, see *spmc-loader* and *startup_loader*. Could not be used together with **libstartup**.

librtos Real Time Operating System. Can not be used together with**libstartup**.

librtos_interrupt Interrupt support for**librtos**. Can not be used together with **libstartup**.

IMPLEMENTING NEW FUNCTIONS

Extending an existing library

To add a function to an existing library, create a new file in the corresponding source folder in **spartanmc/lib_obj/src/<library_name>/**. To get a smaller binary through link time optimization, make sure to implement each function in a separate source file. The type of source file can be either C (*.c) or Assembler (*.s).

To make your new function known to the compiler, create or edit a header file in **spartanmc/include/**. Your implemented function can use other library functions. See section *Library build system* below on how to specify dependences between libraries.

Creating a new library

An entirely new library is create in a seperate folder named **spartanmc/lib_obj/src/<library_name>/**. As described above, create source files in that new folder to implement your library functions.

The new library must be included into the build system. Edit the file *spartanmc/lib_obj/Makefile* for that purpose. See below for details on the library build system.

The library build system

All source files for one certain SpartanMC library are placed in a dedicated directory. The location of the source files is **spartanmc/lib_obj/src/<library_name>**. Possible source file types are C (*.c) and assembler (*.s).

The library build process is controlled by the makefile **spartanmc/lib_obj/Makefile**, where the following variables are of interest:

LIBS List of libraries to build. Each name specified must correspond to a source directory **spartanmc/lib_obj/src/<library_name>**. All library names are specified without the prefix *lib*.

OBJ_DIRS List of directories with additional object files. These files are compiled but not explicitely bundled into a library archive file

(*.a). Other library functions can use the resulting objects as dependences. This is useful for helper functions which could not explicitely associated to a certain library.

DEPS_<library_name>

Specifies dependences for each library, if required. Each object file specified here is included in the library archive (*.a) additionally to the original library code. Valid objects are either from another library or from a directory specified via variable **OBJ_DIRS** (see above).

FILES AND DIRECTORIES

spartanmc/lib_obj/src/<library_name>	Source code for all functions of library <library_name>
spartanmc/lib_obj/Makefile	Makefile controlling the library build process

SEE ALSO

spartanmc-headers(3), *spmc-loader*(1), *startup_loader*(3)

AUTHORS

Written by Markus Vogt

MANPAGE – STARTUP_LOADER(3)

NAME

startup_loader – startup system library with support for updating of program memory content

SYNOPSIS

Link with *-lstartup_loader*

Can not be used together with *-lstartup*

DESCRIPTION

Overview

Provides startup code that allows for replacement of the processors program memory without re-synthesizing the hardware design.

The tool *spmc-loader* implements the mechanism described below on host side to support firmware upload.

Data format

The SPH file format is used to transfer the binary image to the target. For more information, see *sph*.

Upload process

The upload process is started after system reset when requested by the host. When linked with *-lstartup_loader* the startup routine will check for such request and enter the upload routine. Otherwise, the program currently present in memory will be executed as usual.

The initial request is followed by a handshake mechanism to avoid accidental corruption of memory when the hosts UART is transmitting some other unrelated data during system reset. If the handshake fails at any stage, the loader routine exits. Normal program execution follows in that case.

After completing the upload process the target system requires another reset to start execution of the uploaded program.

SYMBOLS

LOADER_UART_BASE

Base address of UART peripheral used for data transfer. The only supported UART hardware currently is *uart_light* .

LIMITATIONS

The upload process can update all memory regions including DMA areas with the following limitations:

The loader code itself cannot be updated. The startup code placed before the loader code (at lower adresses) must not change in size. Both cases will be detected ba the upload routine and reported to the user.

SEE ALSO

spartanmc-project sph

MANPAGE – PRINTF(3)

NAME

printf – formatted string output

SYNOPSIS

#include <stdio.h>

void printf(const char * *s* **);**

void printf(const char * *s* **, void *** *arg1* **);**

void printf(const char * *s* **, void *** *arg1* **, void *** *arg2* **);**

DESCRIPTION

The function **printf**() produces formatted string output according to the format specifications found in the standard C documentation with respect to the limitations described below. It expects its argument *s* to be a null-terminated character string followed by up to two arguments serving as input values for the output format conversion. The argument *s* must not be NULL. Output is sent to the device currently selected for stdio operations (make sure to read **stdio_*_open**(3) and **stdio.h**(3)).

Return Value

This function returns nothing.

Conversion specifiers

The following standard conversion specifiers are supported (see standard C printf documentation for details):

d,**u**	Decimal number in signed (**s**) or unsigned (**u**) notation
x,**X**	Hexadecimal number using lower or upper case notation
o	Octal number
s	String
%	Percent ('%') character

The following additional non-standard conversion specifiers are supported:

b Interprets the given argument as *unsigned int* and produces an output string in binary notation.

Flags, field width, precision, length modifiers

The only supported *flag* is **0** for leading zeroes. *Field with* is supported with a maximum value of **18**. Specifying greater values may lead to undefined behaviour. *Precision* and *length modifiers* are not supported.

EXAMPLE

```
#include <stdio.h>
/* to be completed */
```

SEE ALSO

(to be completed)

AUTHORS

Written by Markus Vogt

MANPAGE – SPH(5)

NAME

sph – SpartanMC hex file format

DESCRIPTION

The *sph* file format is a simple line oriented ASCII representation of memory content. An *sph* file consists of one 5-digit hexadecimal number per line.

Each number represents the 18-bit contents of a memory cell. The upper 6 bits of the most significant nibble (Bits 19-24) are ignored. The format does not permit any other content like comments or whitespace except the line breaks.

Per file format, no addressing information is supported. Data usually is interpreted as single contiguous block of memory starting at address 0x00000.

SEE ALSO

startup_loader spartanmc-project sph