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.

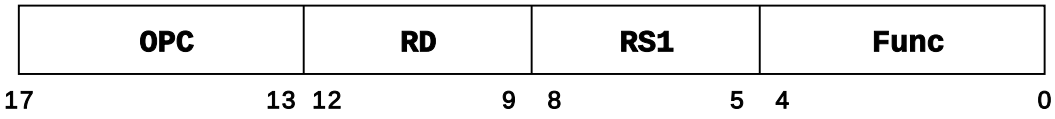


Figure 1-1: R-Type instruction

1.1.2. I-Type

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

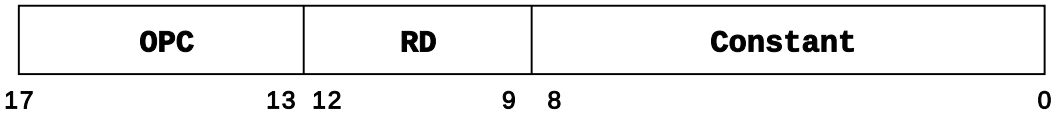


Figure 1-2: I-Type instruction

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.

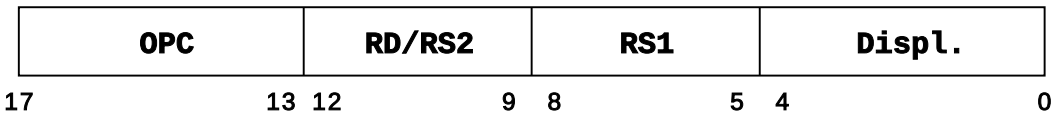


Figure 1-3: M-Type instruction

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.

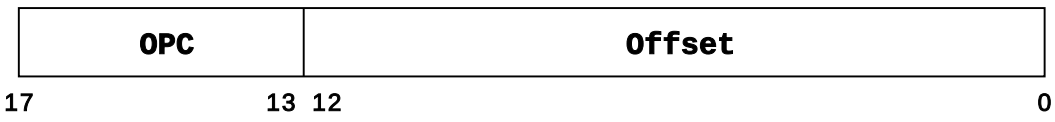


Figure 1-4: J-Type Instruction

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 local 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).

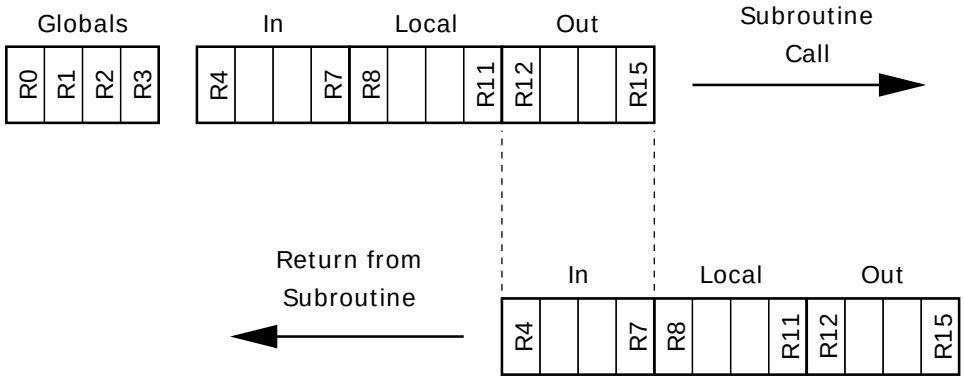


Figure 1-5: SpartanMC register window

1.4. Special Function Registers

For special purposes the SpartanMC contains 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)

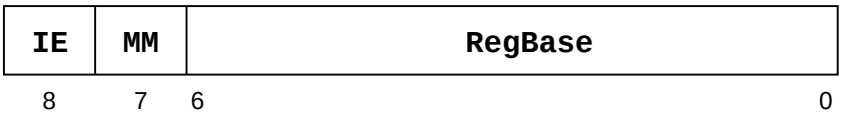


Figure 1-6: Status Register

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)

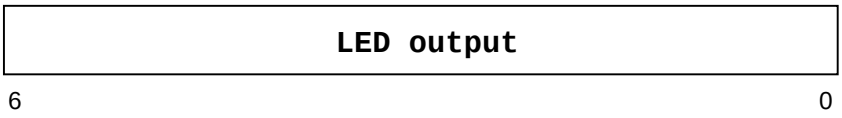


Figure 1-7: LED Register

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)

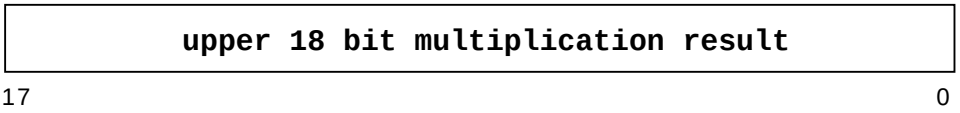


Figure 1-8: MUL Register

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)

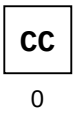


Figure 1-9: CC Register

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)

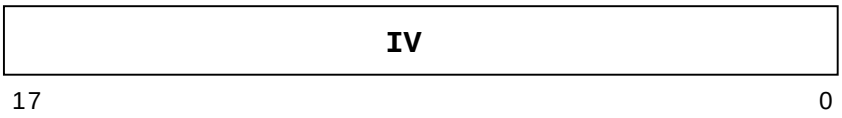


Figure 1-10: IV Register

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.

1.4.7. Hardware Debugging Registers (SFR_DBG_IDX, SFR_DBG_DAT)

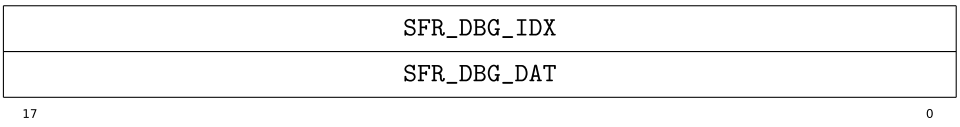


Figure 1-12: DBG Registers

SFR-Name: SFR_DBG_IDX, SFR_DBG_DAT

SFR-Nr.: 6,7

These registers allow indirect addressing of all Hardware Debugging registers

Both registers will be 0 if the Core was synthesized without hardware debugging support

See Hardware Debugging Support for more details on the indirect access

1.5. Instruction Set Details

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

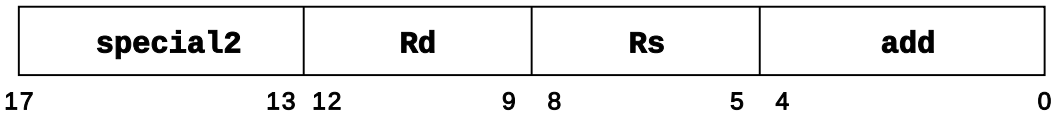
Each of the following pages covers a single SpartanMC instruction. They are organized alphabetically by instruction mnemonic.

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

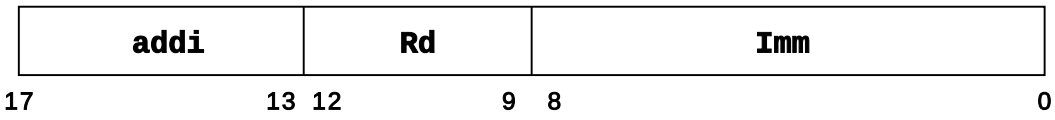
R-Typ

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

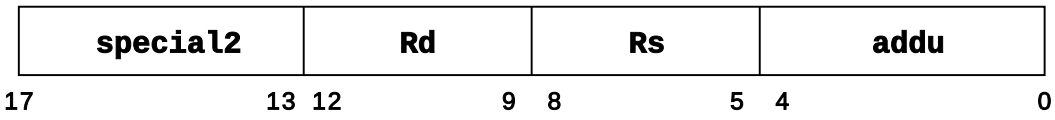
$IR_8^9 \,\#\#\, IR_{8:0}$ performs a sign extension for the 9 bit immediate.

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

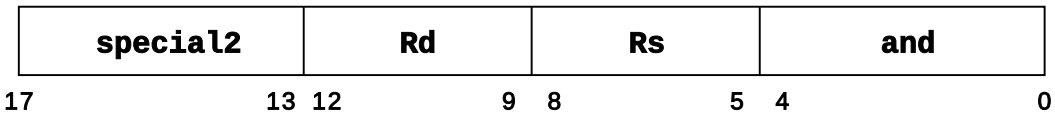
R-Typ

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

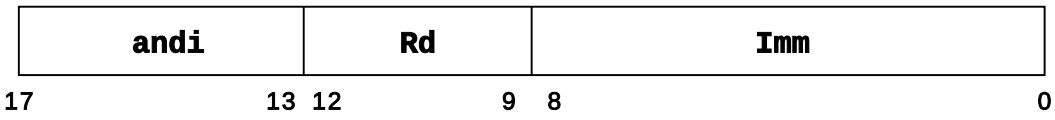
R-Typ

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

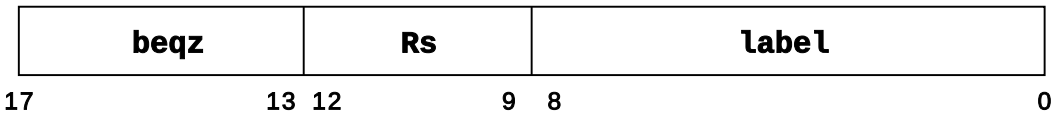
I-Typ

beqz beqz

branch equal zero

Mnemonic

```
beqz        Rs, displacement
```



Pseudocode

```
IF Rs=0; PC ← PC + displacement
```

Delay Slots

1 unconditional delay slot

Description

Sets the program counter to PC + displacement, if GPR Rs equals zero. Note, that in contrast to all other relative jumps/branches the displacement has only a size of 9 bit instead of the usual 13 bit.

Comments

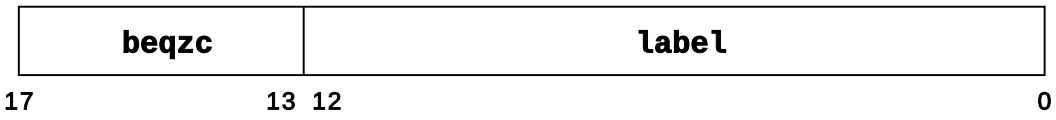
I-Typ

beqzc beqzc

branch equal zero condition bit

Mnemonic

```
beqzc       displacement
```



Pseudocode

```
IF CC=0; PC ← PC + displacement
```

Delay Slots

1 unconditional delay slot

Description

Sets the program counter to PC + displacement if CC has a value of zero.

Comments

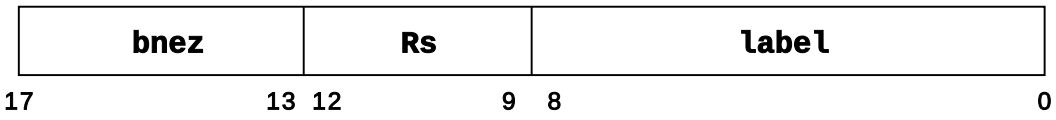
J-Typ

bnez bnez

branch not equal zero

Mnemonic

```
bnez        Rs, displacement
```



Pseudocode

```
IF Rs!=0; PC ← PC + displacement
```

Delay Slots

1 unconditional delay slot

Description

Sets the programm counter to PC + displacement, if GPR Rs is unequal to zero. Note, that in contrast to all other relative jumps/branches the displacement has only a size of 9 bit instead of the usual 13 bit.

Comments

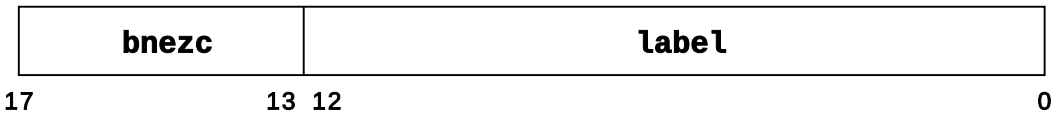
I-Typ

bnezc bnezc

branch not equal zero condition bit

Mnemonic

```
bnezc       displacement
```



Pseudocode

```
IF CC!=0; PC ← PC + displacement
```

Delay Slots

1 unconditional delay slot

Description

Sets the program counter to the PC + displacement if CC is unequal to zero.

Comments

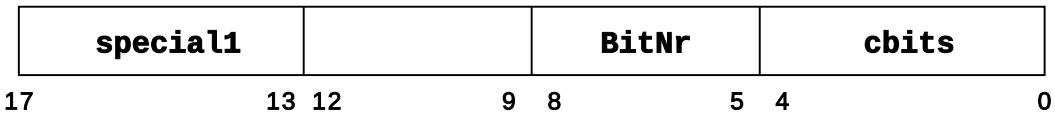
J-Typ

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

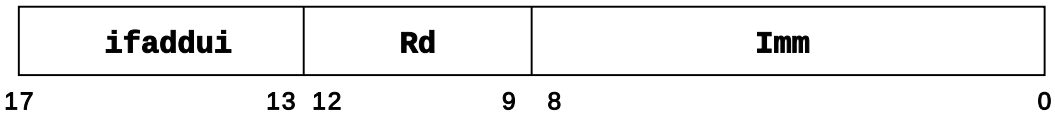
R-Typ

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

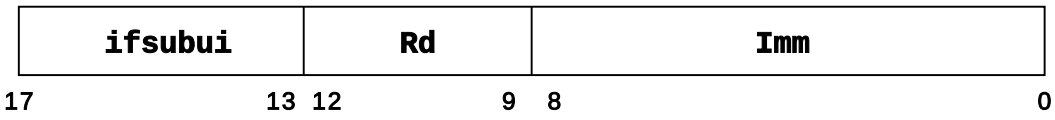
I-Typ

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

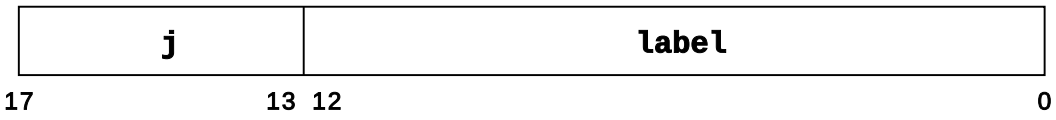
I-Typ

j j

jump

Mnemonic

```
j           displacement
```



Pseudocode

```
PC ← PC + displacement
```

Delay Slots

1 unconditional delay slot

Description

Sets the PC unconditionally to the target address given with the value PC + displacement.

Comments

J-Typ

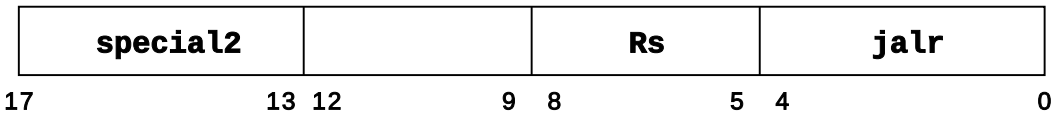
jalr

jalr

jump and link register

Mnemonic

```
jalr        Rs
```



Pseudocode

```
R11 ← PC + 1
PC ← Rs
```

Delay Slots

1 unconditional delay slot

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

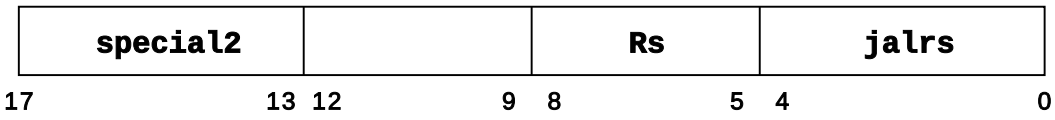
R-Typ

jalrs jalrs

jump and link and shift register window

Mnemonic

```
jalrs       Rs
```



Pseudocode

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

Delay Slots

1 unconditional delay slot

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

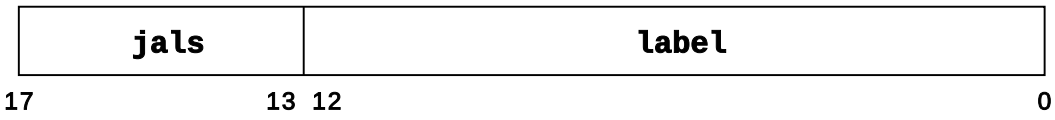
R-Typ

jals jals

jump and link and shift register window

Mnemonic

```
jals        displacement
```



Pseudocode

```
RegBase ← RegBase + 1
R11 ← PC + 1
PC ← PC + displacement
```

Delay Slots

1 unconditional delay slot

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 PC + displacement.

Comments

J-Typ

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

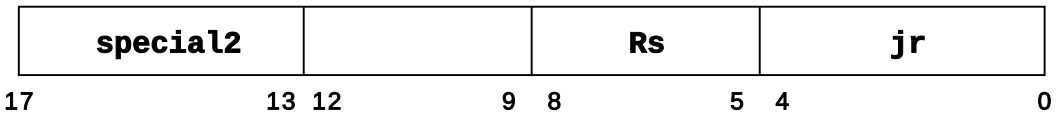
jr

jr

jump register

Mnemonic

```
jr          Rs
```



Pseudocode

PC ← Rs

Delay Slots

1 unconditional delay slot

Description

Set the PC unconditinally to the content of GPR Rs.

Comments

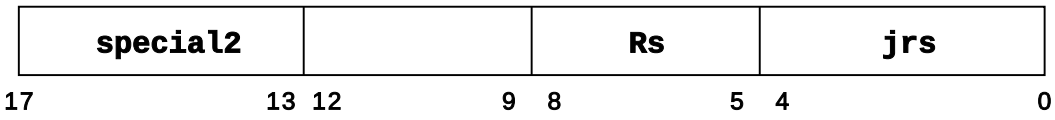
R-Typ

jrs jrs

jump register shift register window (return subroutine)

Mnemonic

```
jrs          Rs
```



Pseudocode

```
PC ← Rs
RegBase ← RegBase - 1
```

Delay Slots

1 unconditional delay slot

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

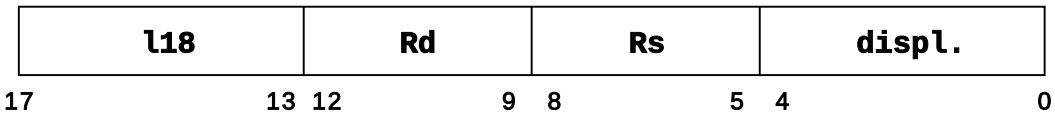
R-Typ

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

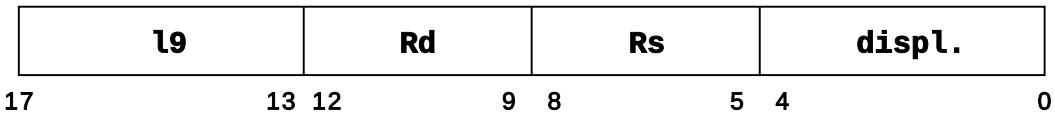
The given address must be even.

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

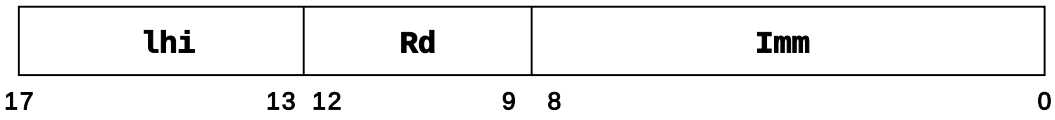
The given address can be even or odd.

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

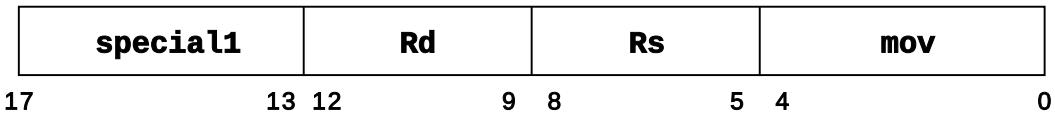
I-Typ

mov mov

move

Mnemonic

```
mov         Rd, Rs
```



Pseudocode

$Rd \leftarrow Rs$

Description

The content of GPR Rs is written to GPR Rd.

Comments

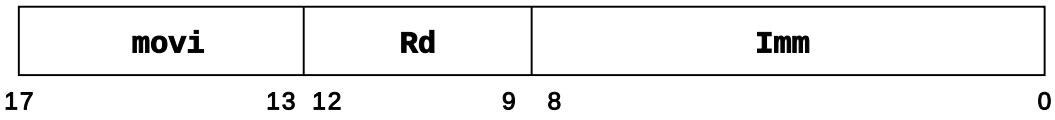
R-Typ

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

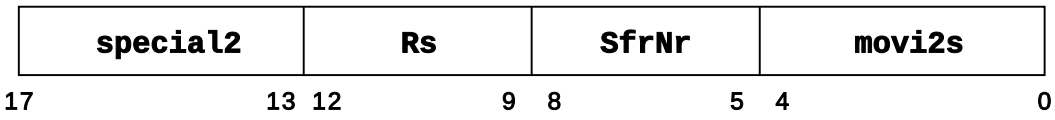
I-Typ

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.

If the destination SFR is SFR_Status, a register window change will only take effect after the next instruction. This is the same behaviour as with delay slots in Function Calls, where the delay slot still uses the old register window.

Comments

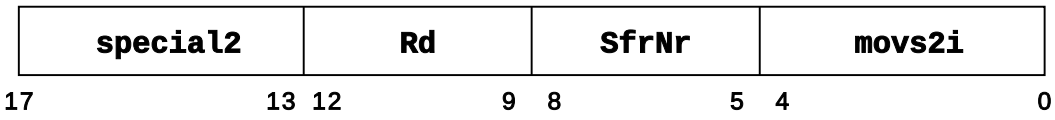
R-Typ

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

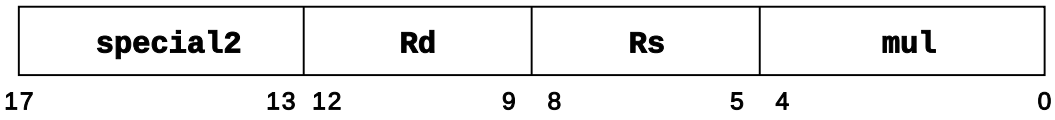
In this instruction code, $IR_{8:5}$holds the number of the SFR which is used as destination register for this instruction. The source register is given in $IR_{12:9}$.

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

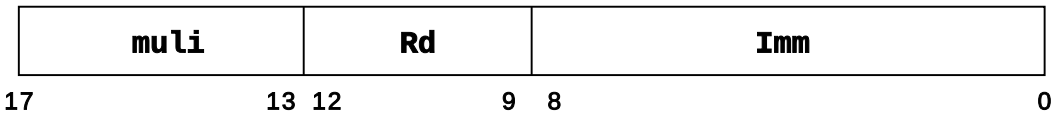
R-Typ

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

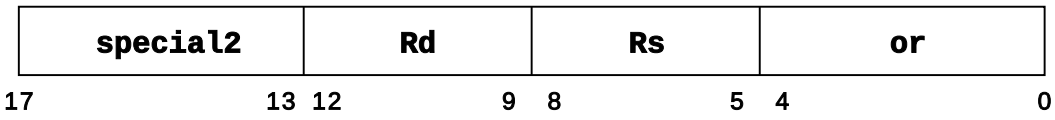
I-Typ

nop nop

no operation

Mnemonic

nop Rd



Pseudocode

Rd ← Rd or Rd

Description

Convenience Instruction. Is really an `or Rd, Rd` , but much more recognizable. Since there are 16 possible no-op combinations, this allows encoding extra information into the nop. This is useful only for debugging purposes, as it allows manipulating bypass logic and makes the instruction more distinguishable in assembler. Using no parameter defaults to 0.

Comments

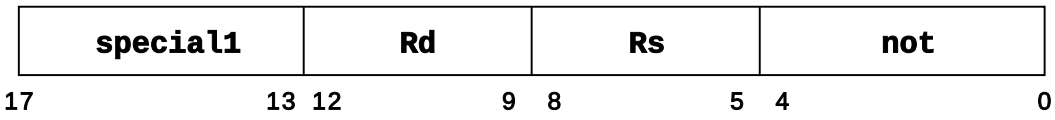
R-Typ

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

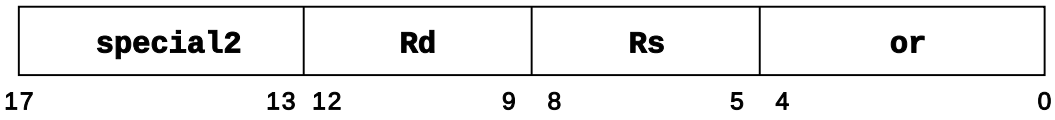
R-Typ

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

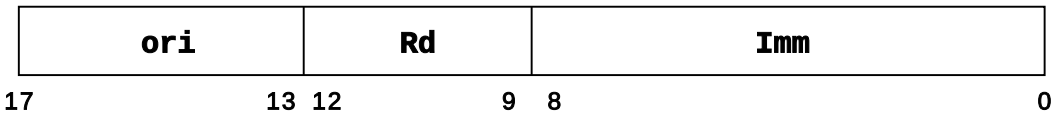
R-Typ

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

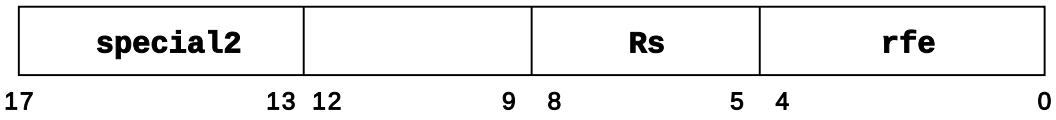
I-Typ

rfe rfe

return from exception

Mnemonic

```
rfe          Rs
```



Pseudocode

PC ← Rs

RegBase ← RegBase - 1

Delay Slots

1 unconditional delay slot

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.

This instruction is internally identical to `jsr` except that this will raise `ir_return` for a single cycle.

Comments

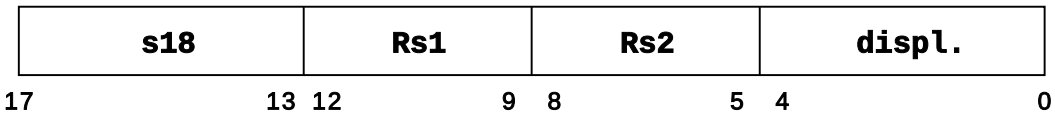
R-Typ

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

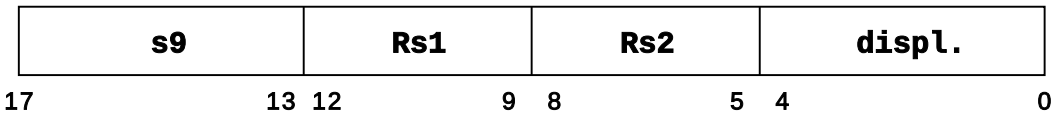
The 18 bit address must be even.

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

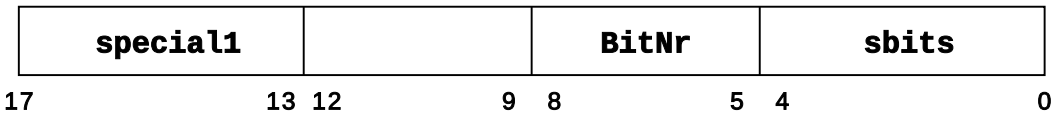
The given address can be even or odd.

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

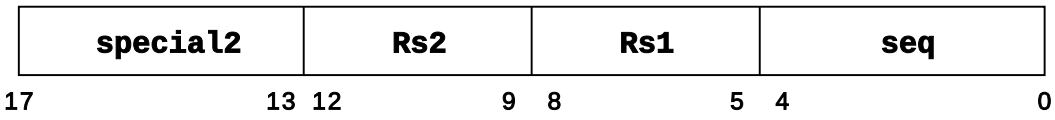
R-Typ

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

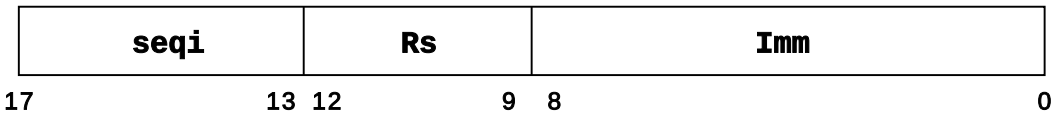
seqi

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

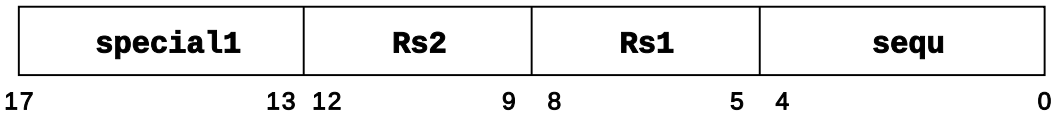
I-Typ

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

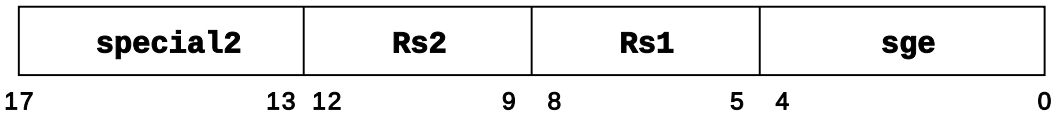
R-Typ

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

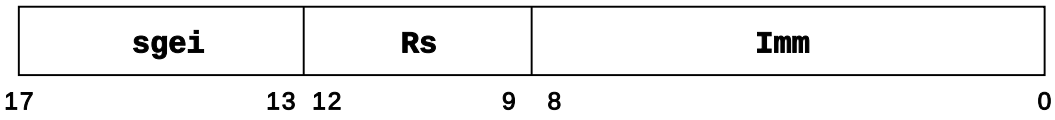
R-Typ

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

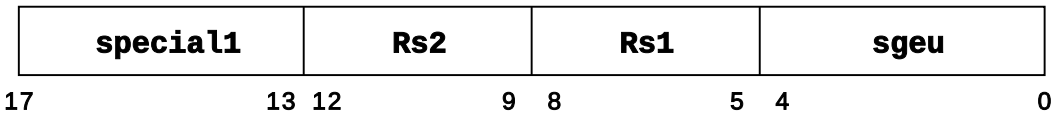
I-Typ

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

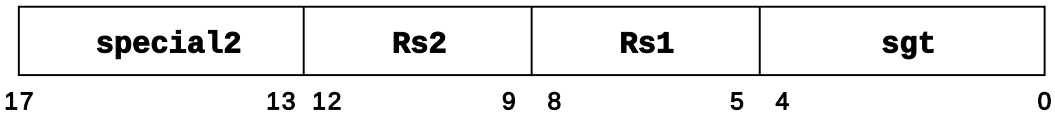
In this instruction -1 = 0x3FFFF is bigger than Ox1FFFF.

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

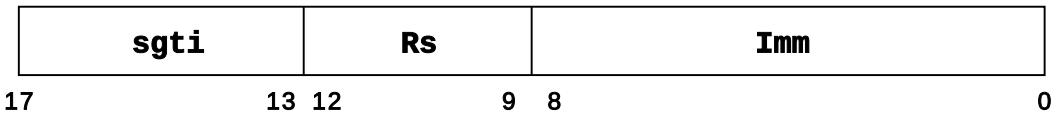
R-Typ

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

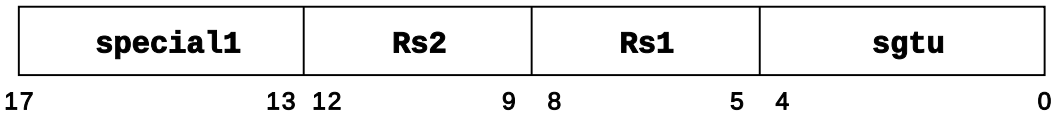
I-Typ

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

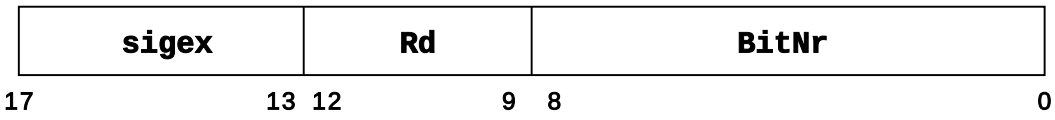
R-Typ

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

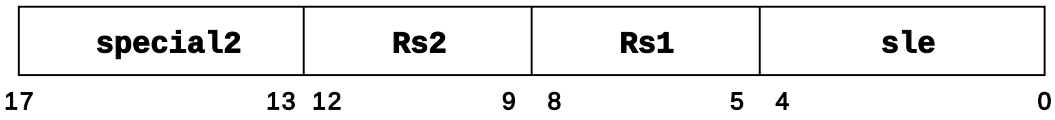
The allowed values for BitNr are 8, 9 or 16. Other values will be treated as 8.

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

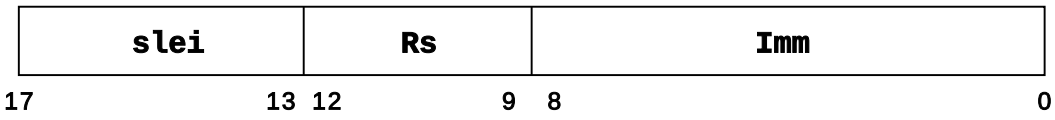
R-Typ

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

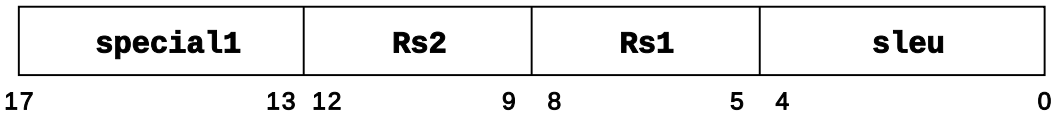
I-Typ

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

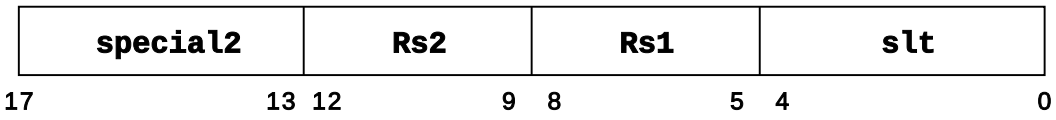
R-Typ

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

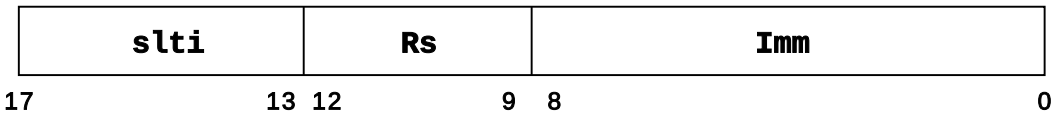
R-Typ

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

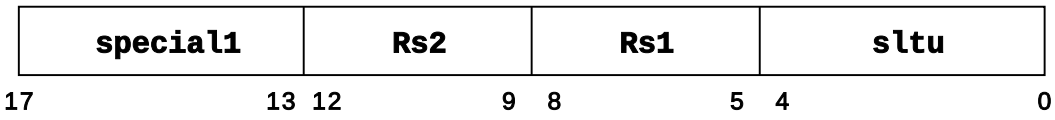
I-Typ

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

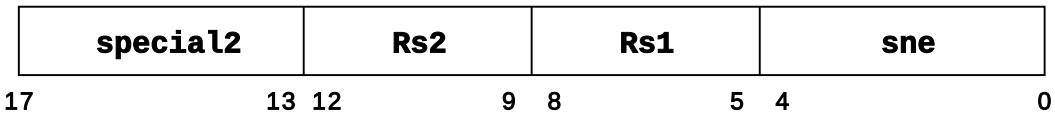
R-Typ

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

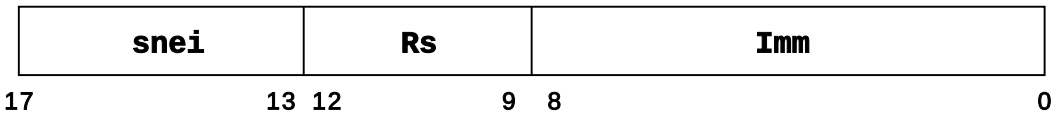
R-Typ

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

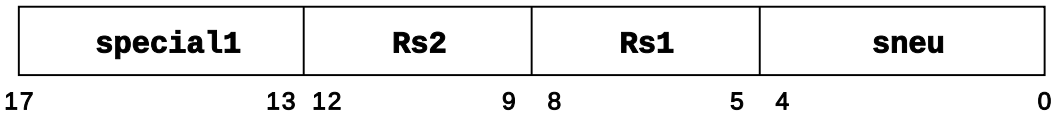
I-Typ

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

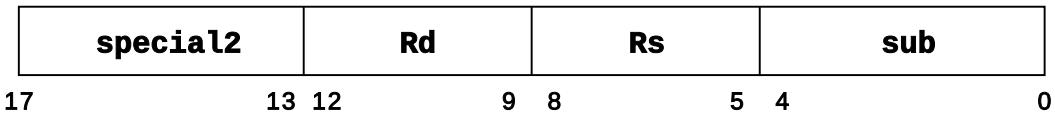
R-Typ

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

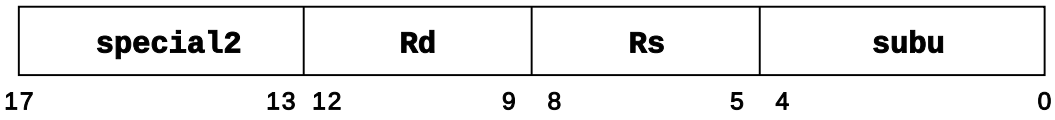
R-Typ

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

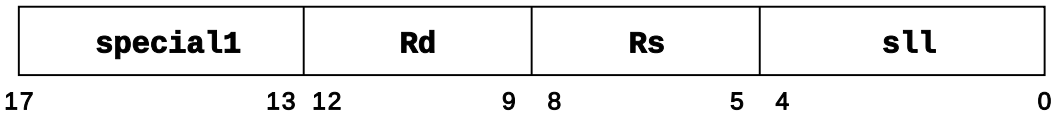
R-Typ

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.

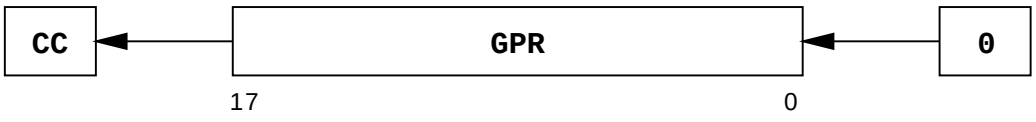


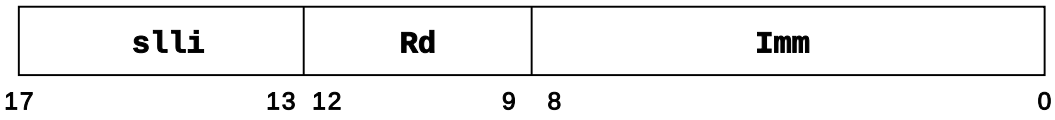
Figure 1-13: shift left logical

slli slli

shift left logical immediate

Mnemonic

```
slli        Rd, Imm
```



Pseudocode

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

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.

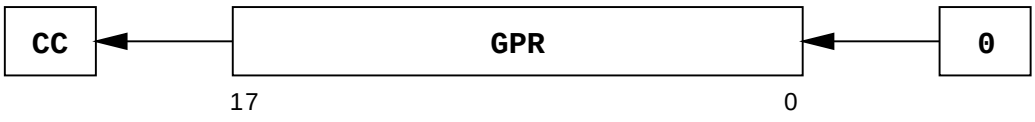


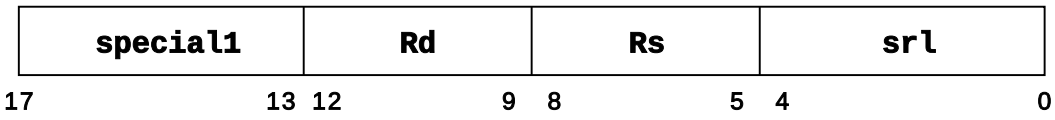
Figure 1-14: shift left logical immediate

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.

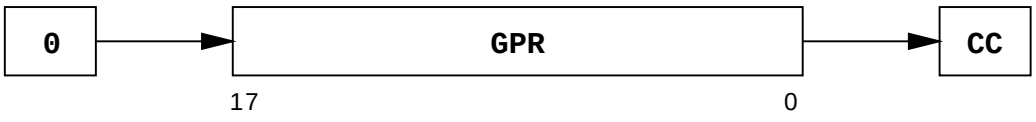


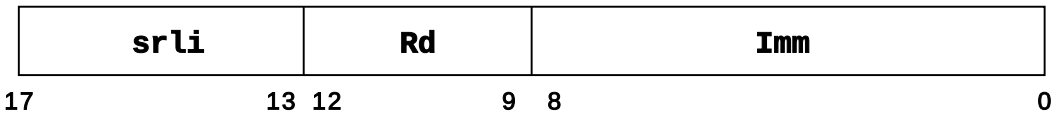
Figure 1-15: shift right logical

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.

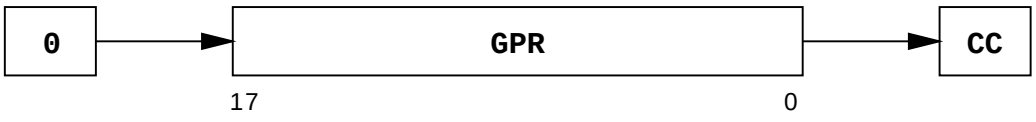


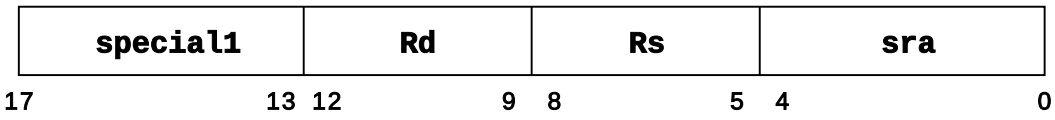
Figure 1-16: shift right logical immediate

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.

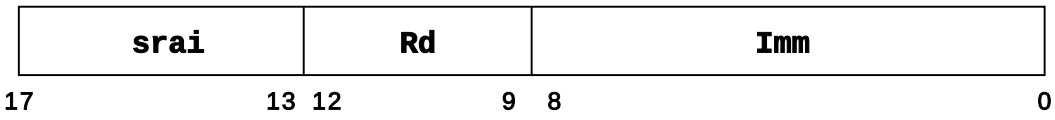
Figure 1-17: shift right arithmetic

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.

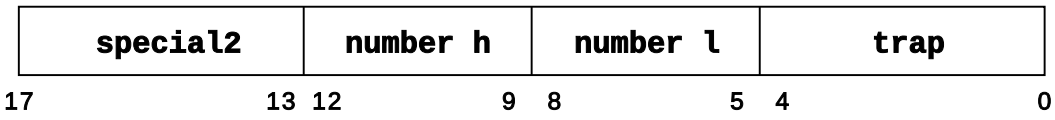
Figure 1-18: shift right arithmetic immediate

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

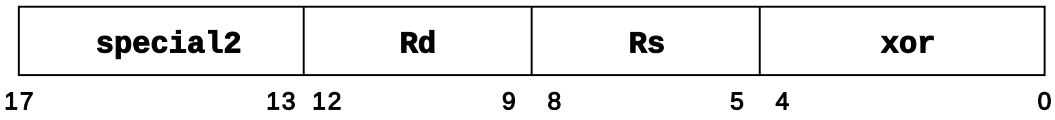
number = number h ## number l

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

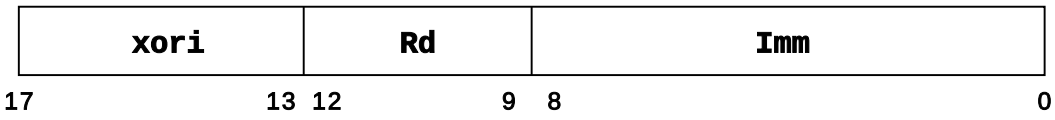
R-Typ

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.

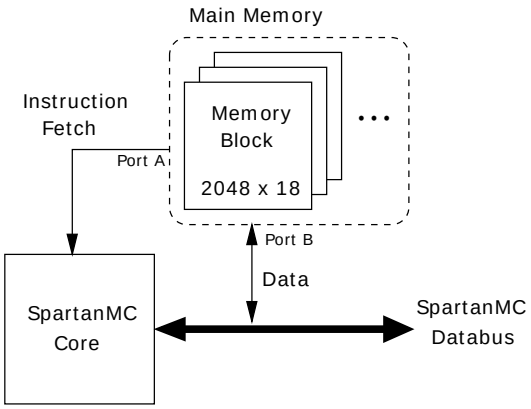


Figure 2-19: Dual ported main memory

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.

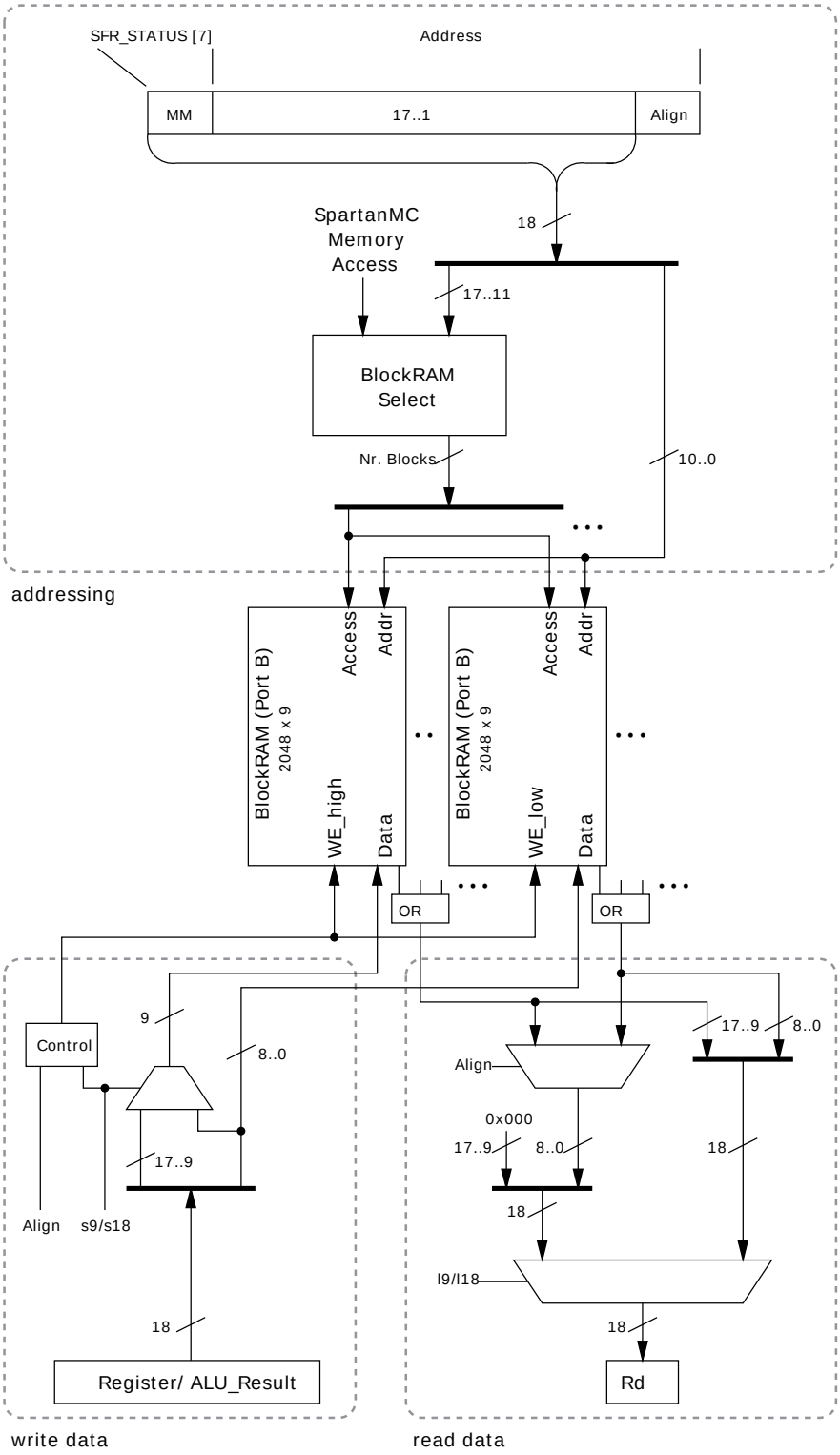


Figure 2-20: Data address management

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.

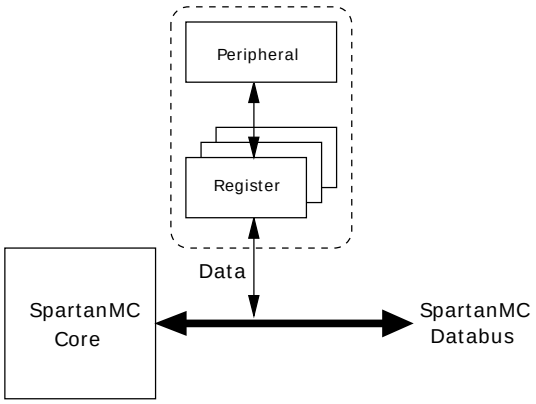


Figure 2-21: Memory mapped registers

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.

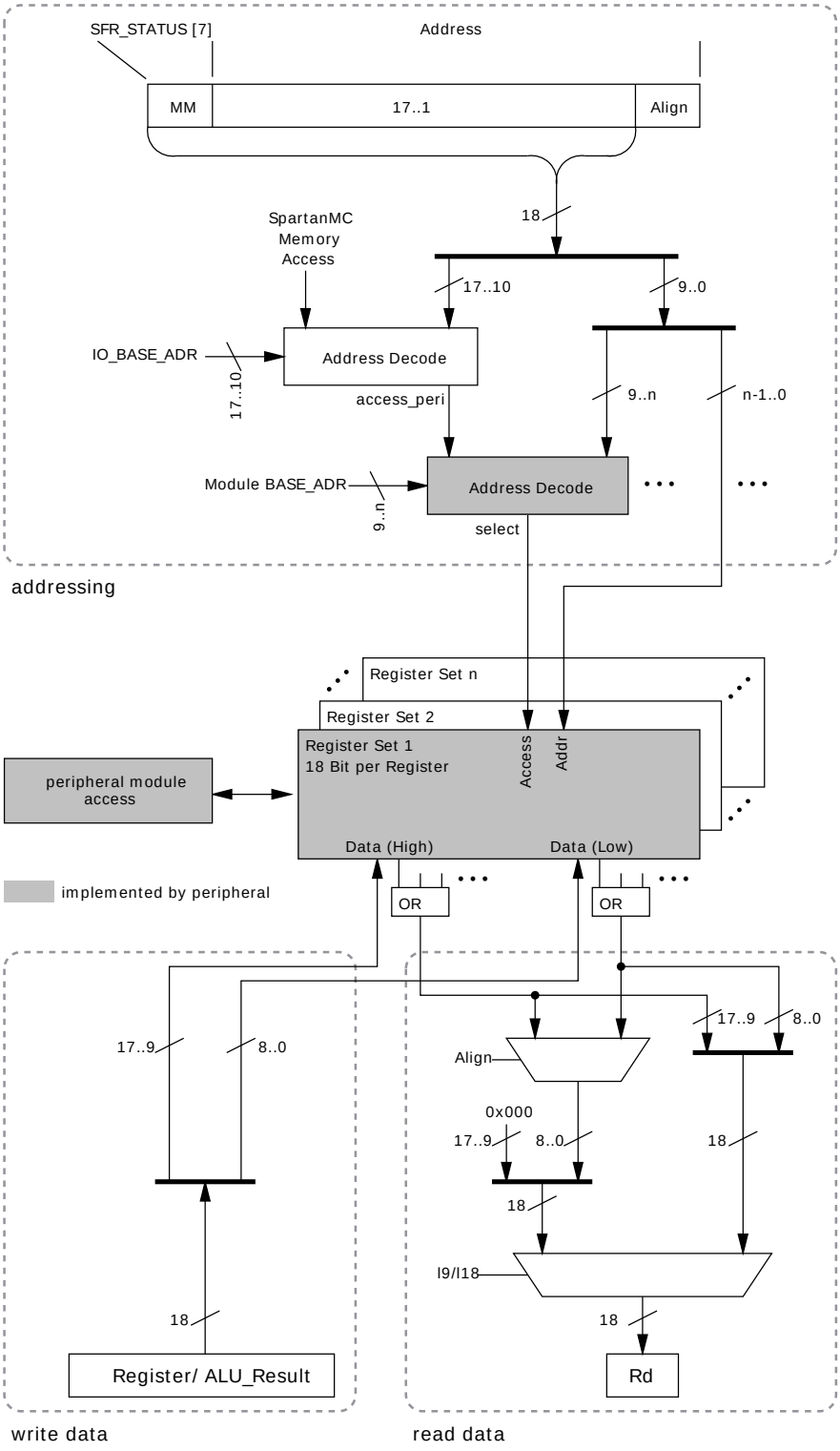


Figure 2-22: Peripheral register address management

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.

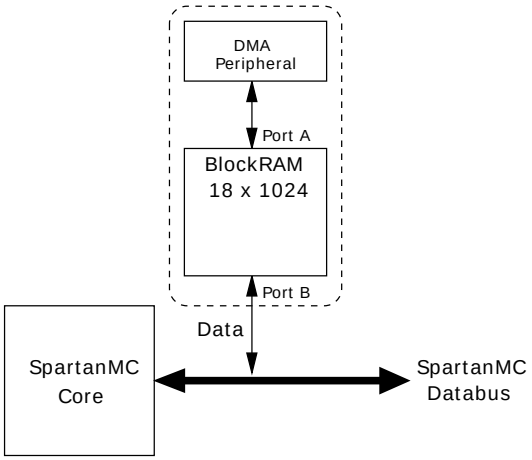


Figure 2-23: DMA with dual ported BlockRAM

Note: Due to the SpartanMC memory management which uses the second port of the BlockRAM as instruction fetch, the processor can not execute code from 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 half word to be written is selected by the store_access_low and store_access_high lines.

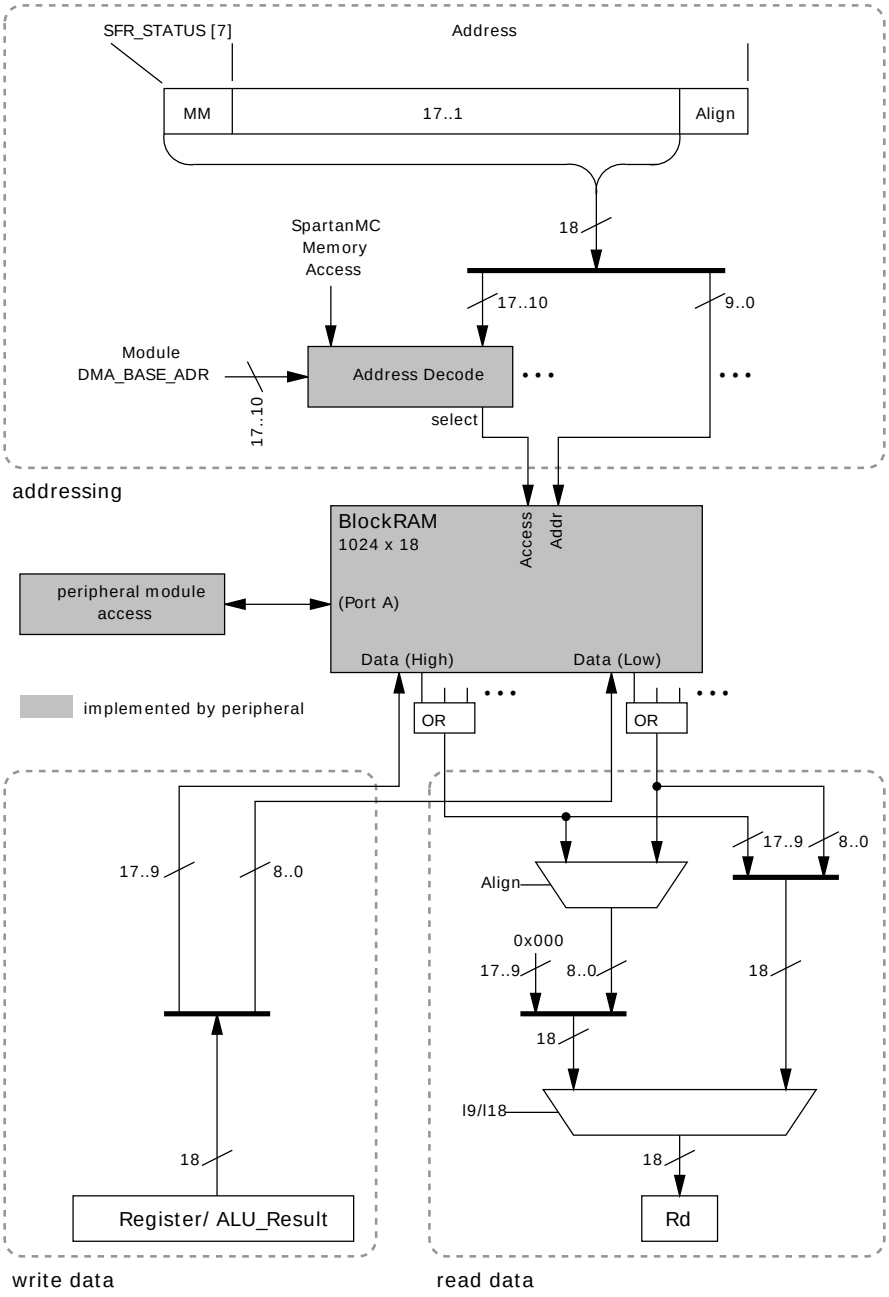


Figure 2-24: DMA address management

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.

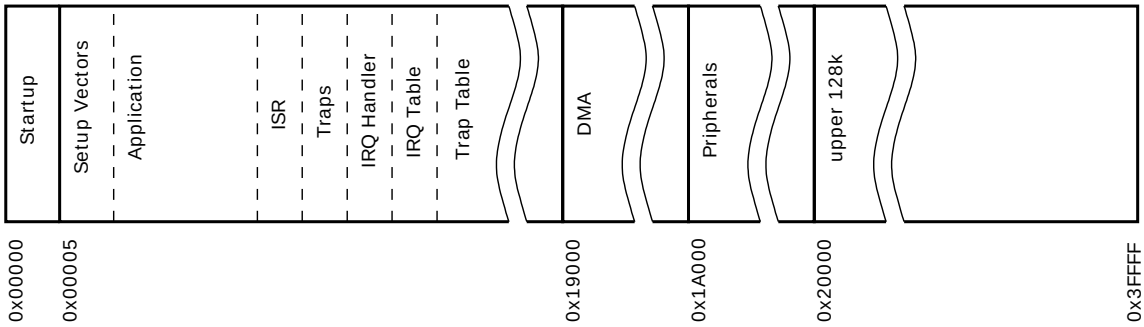


Figure 2-25: Example memory map

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. Performance counter

The performance counter counts clock cycles and certain events that are generated by the SpartanMC core. Counting can be enabled and disabled by software to profile specific sections of code. The performance counter is controlled by two special function registers. They allow starting and stopping the counters as well as configuring the event type that is counted. All counters provide a configurable prescaler. Event counters can detect edges to count events that last longer than one clock cycle.

3.1. Module Parameters

Table 3-4: Performance counter module parameters

Parameter	Default Value	Descripton
PERFORMANCE_COUNTER	0	Specifies whether the SpartanMC core is equipped with a performance counter.
COUNT_CYCLECOUNTERS	2	Specifies the number of cycle counters.
COUNT_EVENTCOUNTERS	2	Specifies the number of event counters.
WIDTH_CYCLECOUNTER	36	Specifies the width of the cycle counter. Possible values are 18 or 36 bits.
WIDTH_EVENTCOUNTER	36	Specifies the width of the event counters. Possible values are 18 or 36 bits.

3.2. Special function registers

Table 3-5: Performance counter special function registers

SFR name	SFR number	Descripton
sfr_pcnt_idx	9	Used to specify the register that is accessed by writing or reading sfr_pcnt_dat. Also contains global counter enable/disable bits.
sfr_pcnt_dat	10	Writing/reading this register results in a write/read of the register specified by sfr_pcnt_idx.

Access to the performance counter registers is provided by two special function registers: **sfr_pcnt_idx** and **sfr_pcnt_dat.** sfr_pcnt_idx is an index register that specifies which register is read or written when sfr_pcnt_dat is read/written. The layout of

sfr_pcnt_idx is shown in the table below. sfr_pcnt_idx also provides two global enable and disable bits. They can be used to enable/disable all counters with only one instruction.

Table 3-6: sfr_pcnt_idx register layout

Bits	Default value	Descripton
0 - 3	0	Register number.
4 - 7	0	Selected cycle/event counter.
8	0	0: Access cycle counter register. 1: Access event counter register.
9 - 15	0	Unused.
10	0	Writing 1 disables cycle counter 3. Always reads as 0.
11	0	Writing 1 enables cycle counter 3. Always reads as 0.
12	0	Writing 1 disables cycle counter 2. Always reads as 0.
13	0	Writing 1 enables cycle counter 2. Always reads as 0.
14	0	Writing 1 disables cycle counter 1. Always reads as 0.
15	0	Writing 1 enables cycle counter 1. Always reads as 0.
16	0	Writing 1 disables cycle counter 0. Always reads as 0.
17	0	Writing 1 enables cycle counter 0. Always reads as 0.

3.3. Performance counter registers

The cycle counter and every event counter has three registers, one configuration register and two registers that contain the counter value. Registers are selected by setting **sfr_pcnt_idx** according to table 3-6. After that the registers can be read or written by reading or writing to **sfr_pcnt_dat** .

Table 3-7: Performance counter registers

Register number	Descripton
0x00	Cycle/event counter configuration register.
0x01	Cycle/event counter value (bits 0-17).
0x02	Cycle/event counter value (bits 18-35).

The cycle counter configuration register is used to configure the cycle counters. It also contains information about the amount of available cycle and event counters.

Table 3-8: Cycle counter configuration register layout

Bits	Default value	Descripton
0	0	Enable counter. 0: Counter disabled. 1: Counter enabled.
1	0	Counter reset. Writing 1 resets the counter to zero. Always reads as 0.
2	0	Interrupt filter enable. Writing 1 disables counting while the processor handles an interrupt.
4-7	0	Prescaler value: 0: No prescaler 1: Prescaler 2 2: Prescaler 4 3: Prescaler 8 ... 15: Prescaler 32768
8	0	Cycle counter overflow flag. Writing 1 will clear the flag.
9-11	1	Number of cycle counters.
12-16	2	Number of event counters.

The event counter configuration register defines the behaviour of the event counters. It is used to set the event that is counted as well as enabling edge detection or a prescaler.

Event counters can be tied to a cycle counter. This is done by setting bit 6 to 1 and bits [4:5] to the number of the cycle counter. When an event counter is tied to a cycle counter it uses the cycle counters enable and reset signals. This means that the event counter is only counting events when the cycle counter is running. Enabling the interrupt filter for a cycle counter will also enable it for all event counters that are tied to that cycle counter.

Table 3-9: Event counter configuration register layout

Bits	Default value	Descripton
0-3	0	Selected event number.
4-5	0	Cycle counter that the event counter is tied to.
6	0	Tie event counter to a cycle cunter.
7	0	Enable event counter. 0: Counter disabled. 1: Counter enabled.
8	0	Reset event counter. Always reads as 0.
9	0	Writing 1 enables the interrupt filter.

Bits	Default value	Descripton
10	0	Edge detect. 1 enables edge detection.
11	0	Event counter overflow flag. Writing 1 will clear the flag.
12-15	0	Prescaler value: 0: No prescaler 1: Prescaler 2 2: Prescaler 4 3: Prescaler 8 ... 15: Prescaler 32768

3.4. Countable events

Table 3-10 lists all countable events. The mnemonics are defined in **perf.h** .

Table 3-10: Countable events

Event number	Mnemonic	Descripton
0	EVENT_STALL	Pipeline stall.
4	EVENT_DWRITE	Data write.
5	EVENT_DREAD	Data read.
6	EVENT_INTR	Interrupt generated.
7	EVENT_NOP	nop instruction executed.
8	EVENT_EXT1	External event line 1.
9	EVENT_EXT0	External event line 0.
10	EVENT_CCNT0_OVF	Cycle counter 0 overflow.
11	EVENT_CCNT1_OVF	Cycle counter 1 overflow.
12	EVENT_CCNT2_OVF	Cycle counter 2 overflow.
13	EVENT_CCNT3_OVF	Cycle counter 3 overflow.

3.5. Example code

The following example code demonstrates the use of the performance counters. In this example a function **do_work(uint36_t i)** is profiled. This function generates **i** read and write accesses to memory. In addition to that an interrupt is generated every 256 clock cycles using a RTI timer. After profiling has finished the result is printed over a UART interface.

The example uses a performance counter with one cycle counter and two event counters. The event counters are tied to the cycle counter.

```
#include <peripherals.h>
#include <moduleParameters.h>
#include <perf.h>
#include <interrupt.h>
#include <stdio.h>

static void do_work(uint36_t i)
{
   volatile int a = 0, b = 0;

   while(i--) {
      b = a;
      asm volatile("nop");
   }

   return;
}

   FILE * stdout = &spartanmc_0_uart_light_0_file;

int main(void)
{
   int i;
   struct perf_conf c;
   struct perf_result r;

   printf("Started!\r\n");

   c.num_ccnt = 1;
   c.num_evcnt = 2;
   c.cycle[0].prescaler = 0;
   c.cycle[0].intr_filter = 1;

   c.event[0].eventnum = EVENT_DREAD;
   c.event[0].prescaler = 0;
   c.event[0].edge_detect = 1;
   c.event[0].tie_ccnt = 1;
   c.event[0].tied_ccnt = 0;

   c.event[1].eventnum = EVENT_INTR;
   c.event[1].prescaler = 0;
   c.event[1].edge_detect = 1;
   c.event[1].tie_ccnt = 1;
   c.event[1].tied_ccnt = 0;
   perf_init(&c);
```

```
   spartanmc_0_rti_0.ctrl = 0x23; // RTI prescaler 256
   interrupt_enable();

   printf("Profiling started: Interrupts enabled.\r\n");
   perf_start(0);
   do_work(1000);
   perf_stop(0);
   perf_read(&r);
   perf_results_printf(&r);
   perf_reset();

   printf("Profiling started: Interrupts disabled.\r\n");
   spartanmc_0_rti_0.ctrl = 1;
   perf_start(0);
   do_work(1000);
   perf_stop(0);
   perf_read(&r);
   perf_results_printf(&r);
   perf_reset();

   printf("Entering endless loop.\r\n\n\n");
   while(1) {
      ;
   }

   return 0;
}

void isr00(void)
{
   // Reset interrupt flag.
   volatile int t = spartanmc_0_rti_0.ctrl;

   return;
}
```

The results received via UART look like this:

```
Started!
Profiling started: Interrupts enabled.
========== Performance counter results ==========
Cycle0: 9086
Event0: 1000
Event1: 39

Profiling started: Interrupts disabled.
========== Performance counter results ==========
```

```
Cycle0: 9007
Event0: 1000
Event1: 0

Entering endless loop.
```

3.6. perf.h header file

```
#include <stdint.h>

#define MAX_CYCLECOUNTERS   4
#define MAX_EVENTCOUNTERS   16

#define EVENT_CCNT3_OVF   13
#define EVENT_CCNT2_OVF   12
#define EVENT_CCNT1_OVF   11
#define EVENT_CCNT0_OVF   10
#define EVENT_EXT0      9
#define EVENT_EXT1      8
#define EVENT_NOP      7
#define EVENT_INTR      6
#define EVENT_DREAD      5
#define EVENT_DWRITE   4
#define EVENT_IREAD      3
#define EVENT_IHIT      2
#define EVENT_IMISS      1
#define EVENT_STALL      0

struct evcnt_conf {
   uint9_t prescaler; // Prescaler value.
   uint9_t eventnum; // Number of the counted event.
   uint9_t tied_ccnt;
   uint9_t edge_detect; // Edge detection enabled?
   uint9_t intr_filter; // Interrupt filtering enabled?
   uint9_t tie_ccnt;
};

struct ccnt_conf {
   uint9_t prescaler;
   uint9_t intr_filter;
};


struct perf_conf {
   uint9_t num_ccnt;
   uint9_t num_evcnt;
```

```
   // Configuration for the cycle counters.
   struct ccnt_conf cycle[MAX_CYCLECOUNTERS];
   // Configuration for the event counters.
   struct evcnt_conf event[MAX_EVENTCOUNTERS];
};

struct perf_result {
   struct {
      uint36_t counter;
      uint9_t overflow;
   } cycles[MAX_CYCLECOUNTERS];

   struct {
      uint36_t counter;
      uint9_t overflow;
   } events[MAX_EVENTCOUNTERS];
};

// Used by AutoPerf.
struct perf_auto_result {
   char *name;
   struct perf_result result;
};

// Sets the global enable bit of cycle counter a to 1.
#define perf_start(a)   asm volatile("movi2s sfr_pcnt_idx,
%0" : : "r"(1<<(17 - 2*(a)) ))
// Sets the global enable bit of cycle counter a to 0.
#define perf_stop(a)   asm volatile("movi2s sfr_pcnt_idx,
%0" : : "r"(1<<(16 - 2*(a)) ))

// Sets the configuration registers according to conf.
int perf_init(struct perf_conf *conf);
// Resets all counters and overflow flags.
void perf_reset(void);
// Reset a specific cycle counter.
void perf_reset_cycle(int n);
// Reset a specific event counter.
void perf_reset_event(int n);
// Reads the results and stores them in res.
void perf_read(struct perf_result *res);
// Uses printf() to print the results.
void perf_results_printf(struct perf_result *res);

// Used by AutoPerf.
#define perf_auto_start() perf_start()
```

```
#define perf_auto_stop(/*int*/a,/*struct perf_auto_result* */
b)\
   do {\
      perf_stop();\
      perf_auto_stop_((a),(b));\
   } while(0)

void perf_auto_init(void);
void perf_auto_stop_(int n, struct perf_auto_result *r);
void perf_auto_print(int size, struct perf_auto_result *r);
```

4. 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.

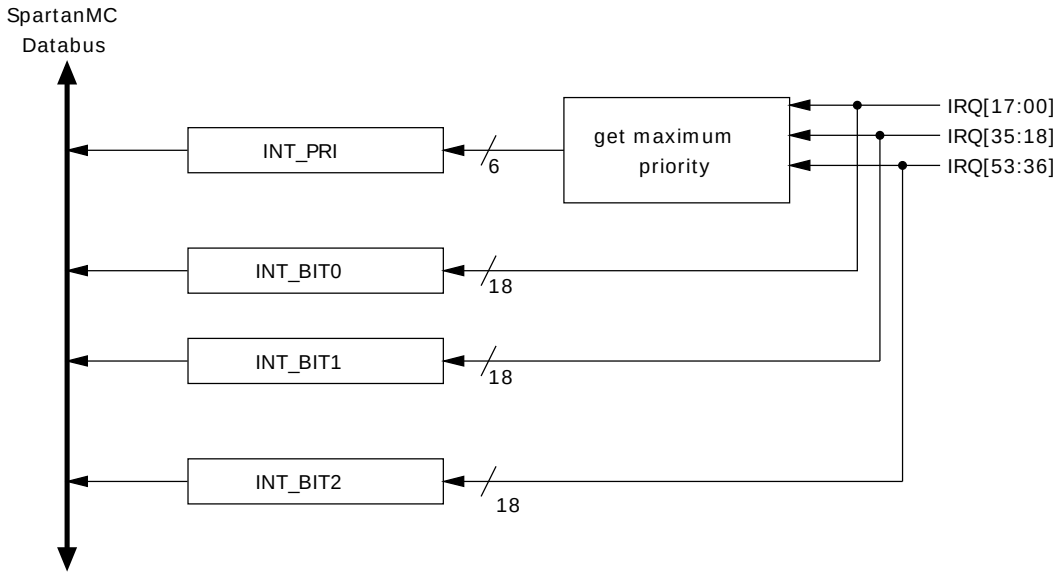


Figure 4-26: IRQ-Ctrl block diagram for IR_SOURCES=54

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.

4.2. Module parameters

Table 4-11: 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.

4.3. Peripheral Registers

4.3.1. 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 4-12: 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 53.

4.3.2. INT_PRI Register

Table 4-13: INT_PRI register layout

Bit	Name	Access	Default	Description
0-7	current max. Priority IRQ-Number	read	0	Register for received Priority IRQ-Number.
8-17		read	x	Not used.

4.3.3. IRQ-Ctrl C-Header for Register Description

```
#ifndef __INTCTRL_H
#define __INTCTRL_H

#ifdef __cplusplus
extern "C" {
#endif
// Number of interrupts (i_bits) is set by jconfig

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

#ifdef __cplusplus
}
#endif

#endif
```

5. 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.

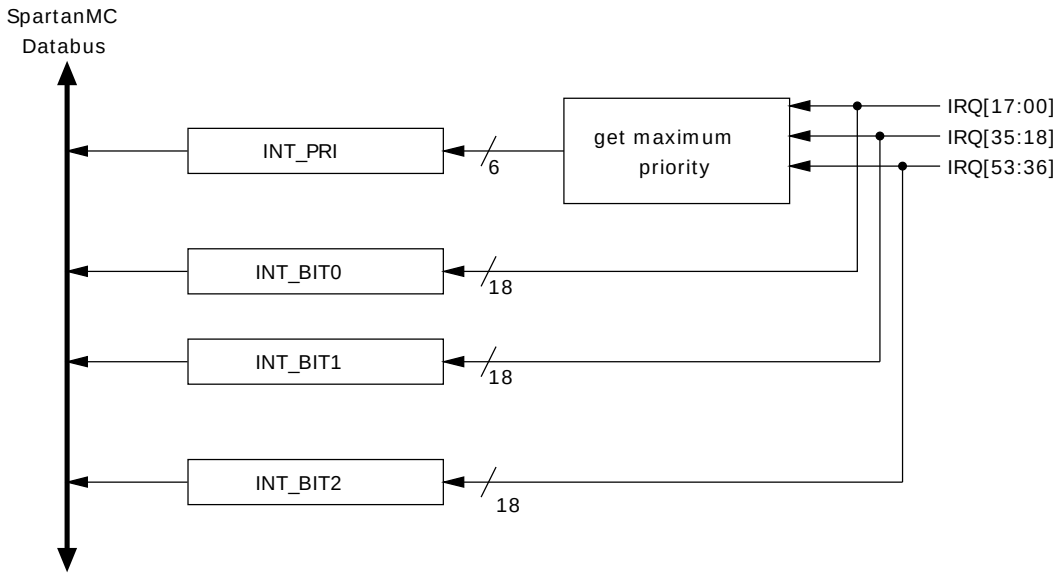


Figure 5-27: IRQ-Ctrl block diagram for IR_SOURCES=54

5.1. 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 interrupt controller additionally requires the bus signals `run_intr`, `intr_return` and `intr_enable`.

5.2. Module parameters

Table 5-14: 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.

5.3. Peripheral Registers

5.3.1. 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 5-15: 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 53.

5.3.2. INT_PRI Register

Table 5-16: INT_PRI register layout

Bit	Name	Access	Default	Description
0-7	current max. Priority IRQ-Number	read	0	Register for received Priority IRQ-Number.

Bit	Name	Access	Default	Description
8-10	Number of currently nested ISR calls	read	0	Register for received Number of currently nested ISR calls.
11-17		read	x	Not used.

5.3.3. IRQ-Ctrl C-Header for Register Description

```
#ifndef __INTCTRLP_H
#define __INTCTRLP_H

#ifdef __cplusplus
extern "C" {
#endif
// Number of interrupts (i_bits) is set by jconfig

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

#ifdef __cplusplus
}
#endif

#endif
```

6. 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.

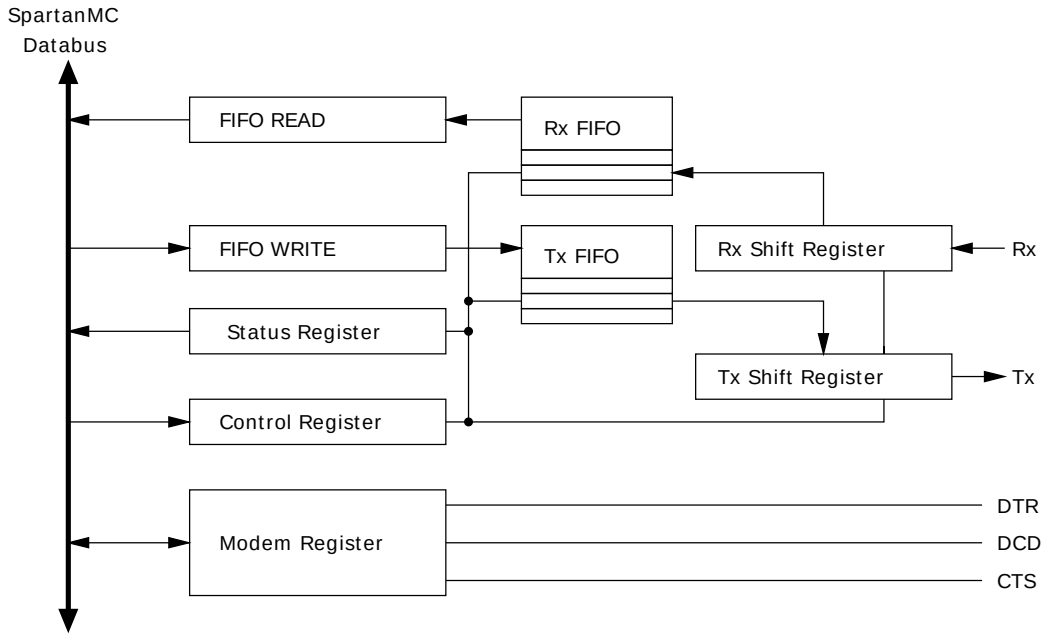


Figure 6-28: UART block diagram

6.1. 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).

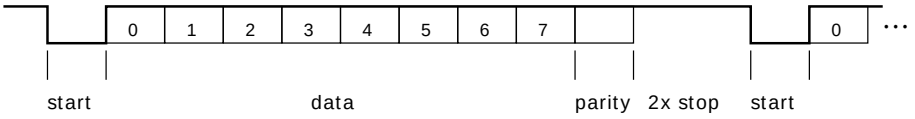


Figure 6-29: UART frame example

6.2. Module parameters

Table 6-17: 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.

6.3. 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.

6.4. Peripheral Registers

6.4.1. 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 6-18: 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.

6.4.2. UART_Status Register

Table 6-19: 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 6-19: UART status register layout

6.4.3. UART_FIFO_READ Register

Table 6-20: 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.

6.4.4. UART_FIFO_WRITE Register

Table 6-21: 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.

6.4.5. 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 6-22: 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 6-22: UART control register layout

6.4.6. 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 6-23: 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 6-23: UART modem register layout

6.4.7. UART C-Header for Register Description

```
#ifndef __UART_H
#define __UART_H

#ifdef __cplusplus
extern "C" {
#endif

#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 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)

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 = status &
Reset Tx Interrupt)
   volatile uint18_t modem;    // write (optional)
} uart_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```

7. 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.

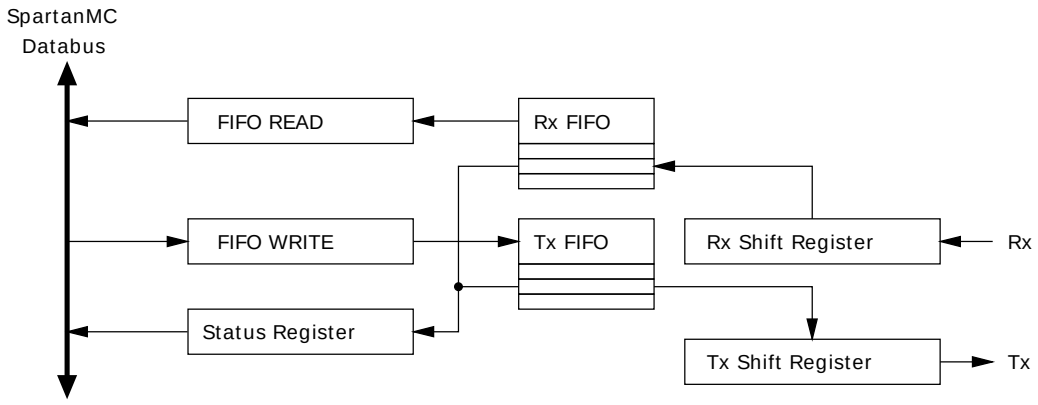


Figure 7-30: UART Light block diagram

7.1. 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).

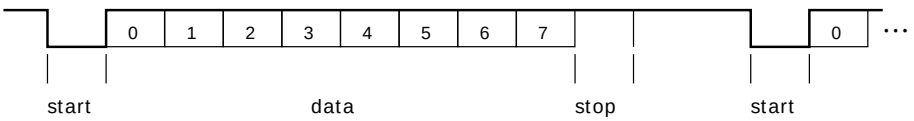


Figure 7-31: UART Light frame

7.2. Module parameters

Table 7-24: 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.

7.3. 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.

7.4. Peripheral Registers

7.4.1. 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 7-25: 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 $UART\_FIFO\_WRITE_0$.

7.4.2. UART_STATUS Register

Table 7-26: 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 7-26: UART status register layout

7.4.3. UART_FIFO_READ Register

Table 7-27: 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.

7.4.4. UART_FIFO_WRITE Register

Table 7-28: 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.

7.4.5. UART C-Header for Register Description

```
#ifndef UART_LIGHT_H_
#define UART_LIGHT_H_

#ifdef __cplusplus
extern "C" {
#endif

#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)

// Interruptfreigabe fuer UART light
#define UART_LIGHT_RXIE      (1<<9)
#define UART_LIGHT_TXIE      (1<<10)


typedef struct {
   volatile uint18_t status;      // read/write = Reset Tx
Interrupt
   volatile uint18_t rx_data;   // read = Reset Rx Interrupt
   volatile uint18_t tx_data;   // write
} 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);

#define declare_UART_LIGHT_FILE(uart) { \
   .base_addr = (void*) uart, \
   .send_byte = (fun_stdio_send_byte) uart_light_send, \
   .receive_byte = (fun_stdio_receive_byte) uart_light_receive,
\
   .receive_byte_nb = (fun_stdio_receive_byte_nb)
uart_light_receive_nb \
}
```

```
void __attribute__((error("stdio_uart_light_open
is no longer supported. Declare a global variable
FILE * stdout = &UART_LIGHT_*_FILE instead")))
stdio_uart_light_open(uart_light_regs_t * uart);

#ifdef __cplusplus
}
#endif

#endif /*UART_LIGHT_H_*/
```

8. Serial Peripheral Interface Bus (SPI)

The SPI is a SpartanMC peripheral device for serial communication using the SPI bus. The SPI enables bit frames to be shifted in and out of the component at programmable speed. The frame width can be changed during runtime so that one single SPI master is able to control multiple different slaves. An SPI master can be connected with up to 15 SPI-slaves which share 3 wires:

- SCLK (*serial clock*)
- MOSI (*master out slave in*)
- MISO (*master in slave out*)

Besides the three shared signals above, there is also one dedicated low-active slave-select-signal SS for each slave. A slave may use the shared wires, only if it has been selected by the master using this select-signal SS. The block diagram below shows the brief structure of the SPI master and its interfaces to the slave and SpartanMC side.

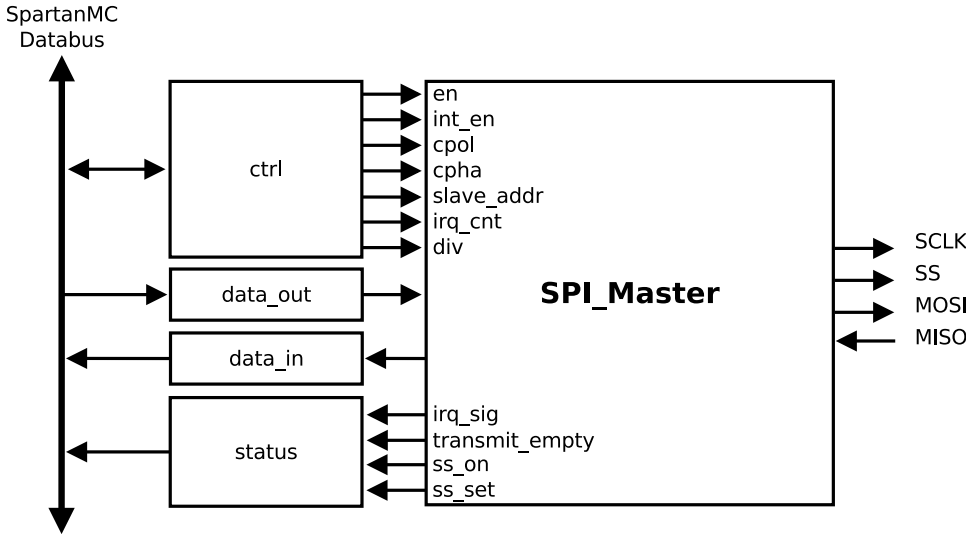


Figure 8-32: SPI block diagram

As shown in this diagram, the SPI master uses four registers on the SpartanMC side, namely `ctrl` , `data_out` , `data_in` and `status` . The former two are writable and the latter two are read-only.The SPI master can be configured by setting the different fields of the `ctrl` register, like the clock divider, the frame width or the slave address etc. After configuration, the data can be sent to the target slave by writing the `data_out` register and the received data can be read from the `data_in` register. Since an external SPI slave works asynchronously with the SpartanMC, the status of the current transmission should be checked to ensure if it has been finished. This can be done by either polling the `TRANS_EMPTY` flag in the `status` register or using the interrupt controller.

8.1. 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 and phase of the clock can be configured by two bits of the control register, namely CPOL and CPHA, in the following way:

If CPOL = 0, the base value of the clock is zero

- if CPHA = 0, the data are read on the rising edge and refreshed on the falling edge
- if CPHA = 1, the data are read on the falling edge and refreshed on the rising edge

If CPOL = 1, the base value of the clock is one

- if CPHA = 0, the data are read on the falling edge and refreshed on the rising edge
- if CPHA = 1, the data are read on the rising edge and refreshed on the falling edge

In other words, the data are always sampled on the first edge of one clock cycle if CPHA = 0, and on the second one if CPHA = 1, regardless of whether the edge is rising or falling. The timing diagram below shows the clock polarity and clock phase according to an example SPI frame of 8 bits.

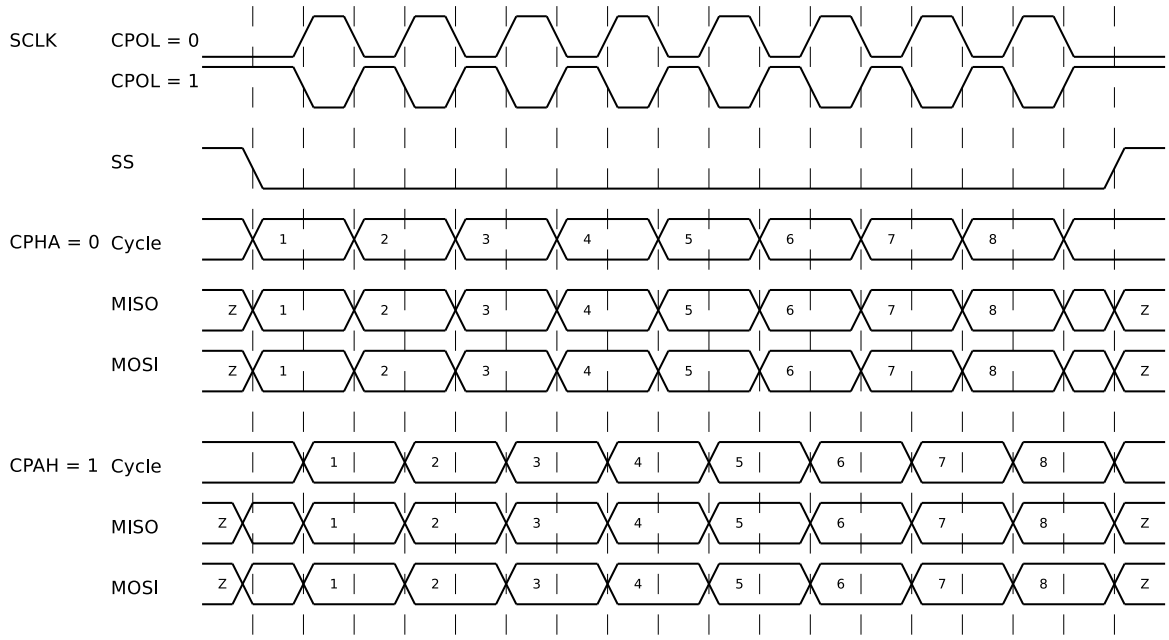


Figure 8-33: SPI frame

Note: By CPHA = 0, the data must be set up one half clock cycle before the first clock edge.

8.2. Module parameter

The SPI master uses only one parameter named `SPI_SS` which gives the number of connected slaves. This parameter can be set in JConfig with respect to the SoC system being built.

Table 8-29: SPI module parameters

Parameter	Default	Descripton
SPI_SS	1	Number of connected slaves

8.3. Peripheral Registers

8.3.1. SPI Register Description

As shown in the table below, the SPI peripheral provides four 18-bit registers which are mapped to the SpartanMC address space.

Table 8-30: SPI registers

Offset	Name	Access	Description
0	spi_control	r/w	Contains the current SPI settings e.g. CPOL, CPHA, clock divider etc.
1	spi_data_out	r/w	Register for outgoing data.
2	spi_data_in	r	Register for incoming data.
3	spi_status	r	Status register.

In the following, the control and status register are described in more details.

8.3.2. SPI Control Register

The table below gives an overview of the layout of the control register:

Table 8-31: SPI control register layout

Bit	Name	Access	Default	Description
0	EN	r/w	0	Enable the master.
1	IRQ_EN	r/w	0	Enable the interrupt.

Bit	Name	Access	Default	Description
2	CPOL	r/w	0	Clock polarity.
3	CPHA	r/w	0	Clock phase.
4-7	SLAVE	r/w	0	Address of the selected slave. 0 deactivates all slave-select-signals 1 activates the first slave-select-signal .... 15 activates the 15th slave-select-signal
8-12	BITCNT	r/w	01000	Number of bits contained in one frame. Only the range from 1 to 18 can be used. Other values will be ignored. 1 : 1-bit SPI frame .... 18 : 18-bit SPI frame
13-15	CLK_DIV	r/w	111	Clock divider. SCLK = (clk_spi) / (4 * divider) 0 : 2^0= 1 1 : 2^1= 2 .... 7 : 2^7= 128
16	CLK_MODE	r/w	0	0=Normal 1=SCLK is chronic active (for itc1407)

Table 8-31: SPI control register layout

8.3.3. SPI Status Register

The following table shows the layout of the status register:

Table 8-32: SPI control register layout

Bit	Name	Access	Default	Description
0	TRANS_EMPTY	r	0	Is set to 1, if the transmission has been finished.
1	IRQ_Sig	r	0	Is set to 1 together with `TRANS_EMPTY` , but will be cleared on a read access to the `data_in` register.
2	SS_ON	r	0	Is set to 1, if the slave address is not equal 0.
3	SS_SET	r	1	Is set to 1, if the slave address has been assigned a new value.

Table 8-32: SPI control register layout

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

```
#ifndef __SPI_H
#define __SPI_H

#ifdef __cplusplus
extern "C" {
#endif

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

//master only functions
int spi_master_activate(spi_master_regs_t *spi,unsigned int
device);      // return = 1 if spi not enable or device<1 or
device>15
void spi_master_deactivate(spi_master_regs_t *spi);
void spi_master_set_div(spi_master_regs_t *spi, unsigned int
div);

//master duplicate functions
unsigned int spi_master_readwrite(spi_master_regs_t *spi,
unsigned int data);
unsigned int spi_master_read(spi_master_regs_t *spi);
void spi_master_write(spi_master_regs_t *spi, unsigned int
data);
void spi_master_enable(spi_master_regs_t *spi);
void spi_master_disable(spi_master_regs_t *spi);
void spi_master_enable_irq(spi_master_regs_t *spi);
void spi_master_disable_irq(spi_master_regs_t *spi);
void spi_master_set_cpol(spi_master_regs_t *spi, unsigned int
cpol);
void spi_master_set_cpah(spi_master_regs_t *spi, unsigned int
cpah);
int spi_master_set_bitcnt(spi_master_regs_t *spi, unsigned int
bitcnt);   // return = bitcnt or -1 if error (bitcnt > 18 or
bitcnt < 1)

//slave duplicate functions
unsigned int spi_slave_readwrite(spi_slave_regs_t *spi,
unsigned int data);
unsigned int spi_slave_read(spi_slave_regs_t *spi);
void spi_slave_write(spi_slave_regs_t *spi, unsigned int
data);
void spi_slave_enable(spi_slave_regs_t *spi);
```

```
void spi_slave_disable(spi_slave_regs_t *spi);
void spi_slave_enable_irq(spi_slave_regs_t *spi);
void spi_slave_disable_irq(spi_slave_regs_t *spi);
void spi_slave_set_cpol(spi_slave_regs_t *spi, unsigned int
cpol);
void spi_slave_set_cpah(spi_slave_regs_t *spi, unsigned int
cpah);
int spi_slave_set_bitcnt(spi_slave_regs_t *spi, unsigned int
bitcnt);      // return = bitcnt or -1 if error (bitcnt > 18 or
bitcnt < 1)

#ifdef __cplusplus
}
#endif

#endif
```

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

```
#ifndef __SPI_MASTER_H
#define __SPI_MASTER_H

#ifdef __cplusplus
extern "C" {
#endif

#include <peripherals/spi_common.h>

// CONTROL
#define SPI_MASTER_CTRL_EN      SPI_CTRL_EN
#define SPI_MASTER_CTRL_INT_EN   SPI_CTRL_INT_EN
#define SPI_MASTER_CTRL_CPOL      SPI_CTRL_CPOL
#define SPI_MASTER_CTRL_CPHA      SPI_CTRL_CPHA
#define SPI_MASTER_CTRL_SLAVE   0x000F0 // 00 0000 0000 1111
0000
#define SPI_MASTER_CTRL_BITCNT   SPI_CTRL_BITCNT
#define SPI_MASTER_CTRL_DIV      0x0E000 // 00 1110 0000 0000
0000
#define SPI_MASTER_CTRL_CLK_MOD   0x10000 // 01 0000 0000 0000
0000

//STATUS COMPATIBILITY SECTION FOR OLD PROJECTS
#define SPI_MASTER_STAT_TRANS_EMPTY SPI_STAT_TRANS_EMPTY
#define SPI_MASTER_STAT_INT      SPI_STAT_INT
#define SPI_MASTER_STAT_SS_ON   SPI_STAT_SS_ON
#define SPI_MASTER_STAT_SS_SET   SPI_STAT_SS_SET
```

```
typedef struct {
   spi_t spi;
} spi_master_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```

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

```
#ifndef __SPI_SLAVE_H
#define __SPI_SLAVE_H

#ifdef __cplusplus
extern "C" {
#endif

#include <peripherals/spi_common.h>

//CONTROL
#define SPI_SLAVE_CTRL_EN      SPI_CTRL_EN
#define SPI_SLAVE_CTRL_INT_EN   SPI_CTRL_INT_EN
#define SPI_SLAVE_CTRL_CPOL      SPI_CTRL_CPOL
#define SPI_SLAVE_CTRL_CPHA      SPI_CTRL_CPHA
#define SPI_SLAVE_CTRL_BITCNT   SPI_CTRL_BITCNT

//STATUS
#define SPI_SLAVE_STAT_DONE      0x00001 // 00 0000 0000 0000
0001
#define SPI_SLAVE_STAT_INT      0x00002 // 00 0000 0000 0000
0010

typedef struct {
   spi_t spi;
} spi_slave_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```

8.3.7. SPI C-Header spi_common.h for Register Description

```
#ifndef __SPI_COMMON_H
#define __SPI_COMMON_H

#ifdef __cplusplus
extern "C" {
#endif

// CONTROL
#define SPI_CTRL_EN      0x00001 // 00 0000 0000 0000 0001
#define SPI_CTRL_INT_EN   0x00002 // 00 0000 0000 0000 0010
#define SPI_CTRL_CPOL      0x00004 // 00 0000 0000 0000 0100
#define SPI_CTRL_CPHA      0x00008 // 00 0000 0000 0000 1000
#define SPI_CTRL_BITCNT   0x01F00 // 00 0001 1111 0000 0000

//STATUS
#define SPI_STAT_TRANS_EMPTY 0x00001 // 00 0000 0000 0000 0001
#define SPI_STAT_INT      0x00002 // 00 0000 0000 0000 0010
#define SPI_STAT_SS_ON   0x00004 // 01 0000 0000 0000 0100
#define SPI_STAT_SS_SET   0x00008 // 10 0000 0000 0000 1000

typedef volatile struct {
   volatile unsigned int spi_control;
   volatile unsigned int spi_data_out;
   volatile unsigned int spi_data_in;
   volatile unsigned int spi_status;
} spi_t;

#ifdef __cplusplus
}
#endif

#endif
```

8.3.8. Basic Usage of the SPI Registers

The structures shown above can be used in a program directly, if `<spi.h>` has been included. They serve as the interface between software and hardware. A programmer can configure and control the SPI master simply using these structures without having to care about any low-level details (e.g. timing) at all. According to several trivial examples, this section illustrates how to use this interface to communicate with the SPI master. First of all, assume that `SPI_MASTER_0` is a pointer which has been assigned the physical address of a SPI master. The registers of the SPI master can be accessed via this pointer.

- **Example 1 : Enable the SPI master or slave**

```
spi_enable(&SPI_MASTER_0);
spi_enable(&SPI_SLAVE_0);
```

- **Example 2 : Set the frame width to 16**

```
err1 = spi_set_bitcnt(&SPI_MASTER_0, 16);
err2 = spi_set_bitcnt(&SPI_SLAVE_0, 16);
/* if (bitcnt > 18) or (bitcnt < 1) then err* = -1 */
```

- **Example 3 : Send the constant value 256 to the slave 1 and 4**

```
/* activate the slave 1 and 4 */
spi_activate(&SPI_MASTER_0, 1);
spi_activate(&SPI_MASTER_0, 4);
/* data written into spi_data_out will be sent */
spi_write(&SPI_MASTER_0, 256);
/* deactivate the slave 1 and 4 */
spi_deactivate(&SPI_MASTER_0);
```

- **Example 4 : Read the received value**

```
/* activate the slave 2 */
spi_activate(&SPI_MASTER_0, 2);
/* data written (256) for read data */
int v = spi_readwrite(&SPI_MASTER_0, 256);
/* deactivate the slave 2 */
spi_deactivate(&SPI_MASTER_0);
```

- **Example 5 : Check IRQ_Sig of the status register**

```
/* wait until the bit has been set */
while(!(SPI_MASTER_0.spi_status&SPI_MASTER_STAT_INT));

/* handle the interrupt here */
```

8.4. 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"

FILE * stdout = &UART_LIGHT_0_FILE;

void main() {
```

```
    unsigned int i;
    printf("\r\nHello SPI_Sample:");

    printf(\r\nEnable the SPI-Core:");
    spi_enable(&SPI_MASTER_0);

    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 : 0x%x\r\n",i,id[i]);
    }
    spi_disable(&SPI_MASTER_0);
    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 sent to a host PC via serial connection. Therefore a UART peripheral is required in the SoC. ID 0 (0x20) specifies the manufacturer type (ST), ID 1 (0x20) specifies device type and ID 2 (0x16) indicates the memory capacity.

```
SpMC loader v20140908

Hello SPI_Sample:
Enable the SPI-Core:
Power-Up the connected SPI-Flash:
Read ID of the SPI-Flash:
ID 0 : 0x20
ID 1 : 0x20
ID 2 : 0x16
```

9. I2C Master

I2C (also referred to as *two-wire interface*) is a serial bus which allows for connection of multiple master devices to multiple slave devices, only using two single bidirectional lines:

- SCL (*serial clock line*)
- SDA (*serial data line*)

Both lines need to be pulled up with resistors. Because of this, both of them remain simply high, if there is no communication between any master and slave. The clock line needs to be driven by a master. Using this clock, the data will be transmitted bit by bit between the master and the corresponding slave over the data line.

The SpartanMC I2C master controller is a quite simple peripheral device which supports basic I2C functions. The following block diagram gives an overview of its structure and interfaces to both the slave and SpartanMC side.

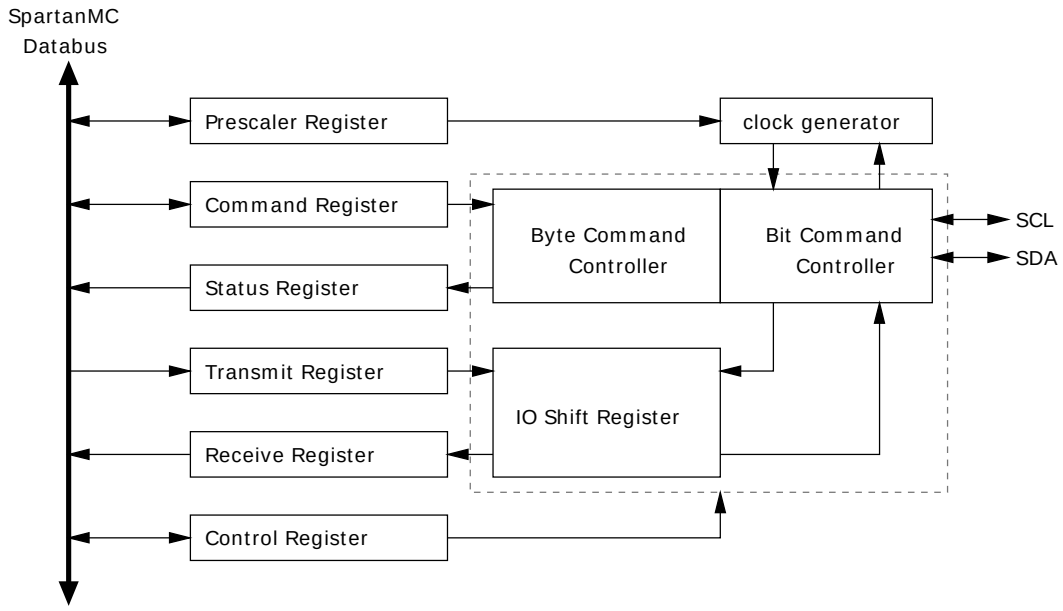


Figure 9-34: I2C block diagram

The I2C master controller can be configured and controlled by setting the writable registers such as the `Control` , `Command` and `Transmit` register on the SpartanMC side with flags, commands, slave addresses or data to be sent. With respect to the current settings, it will operate autonomously, i.e. send/receive data to/from slave. After current operation, the corresponding bits in the `Status` register will be set and the received data will be written in the `Receive` register. Both of the `Status` and `Receive` register are read-only.

9.1. Communication

As mentioned above, both SDA and SCL remain high, if there is no transmission between any master and slave. In this case, the I2C bus is considered as idle and can be used by any master. To start a transmission, SDA is pulled low while SCL remains high. After the start signal, 8-bit data packets will be transferred, one bit on each rising edge of SCL. Since multiple slaves can be attached to the I2C bus, each of them should have a unique 7-bit address so that it can be distinguished from the other slaves. As the first packet, the master should always put the 7-bit address of the target slave and one direction bit on the bus. If the direction bit is 1, the master wants to read data from the slave, otherwise write data to it. After the corresponding slave has received the start packet, it needs to send 1-bit acknowledge back to the master as response. After this handshake, the master can begin reading or writing data. If the current transmission is over, SDA must be released to float high again which is used as stop signal and idle marker. Except for the start and stop signal, the SDA line only changes while SCL is low. The timing diagram below shows an example transmission of two data packets.

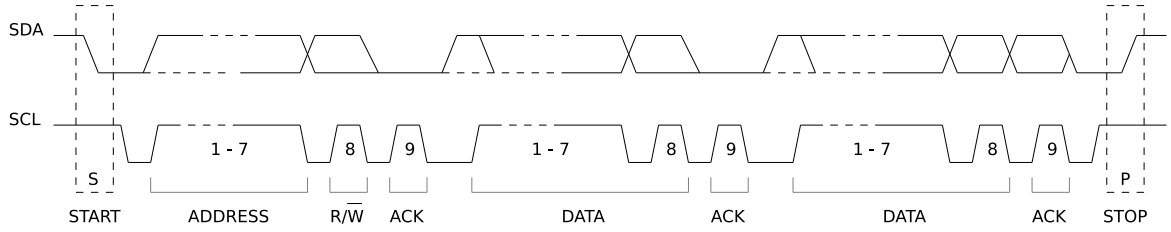


Figure 9-35: SCL, SDA Timing for Data Transmission

Each time after a data packet has been transmitted in one direction, an acknowledge bit needs to be transmitted in the other direction, as shown in the following diagram.

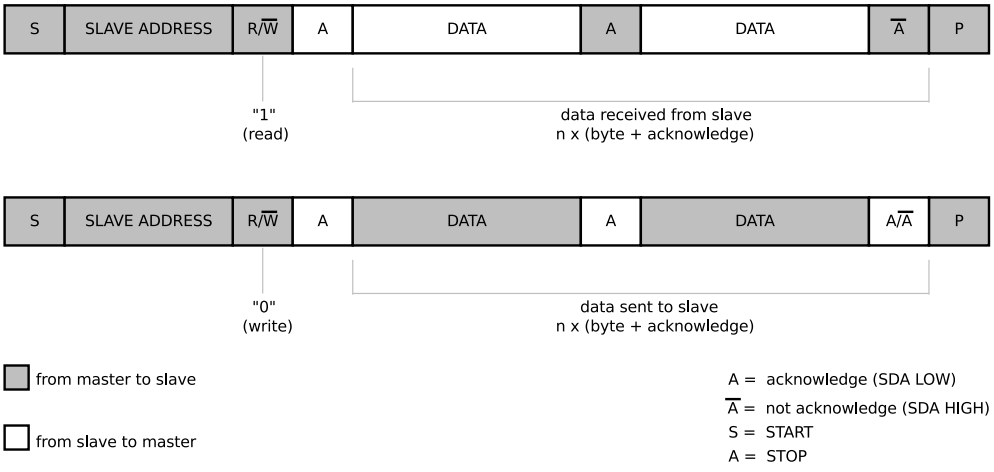


Figure 9-36: I2C Acknowledge

If the transmitter gets a "0" (ACK) as acknowledge, the transmission has succeeded. Otherwise, if it gets a "1", meaning that:

- If the transmitter is master
  - Unknown slave
  - Busy slave
  - Unknown command
- If the transmitter is slave
  - Stop request from the master

9.2. Bus Arbitration

Since multiple masters can be connected to an I2C bus, several of them may start the transmission simultaneously. To overcome this situation, all masters monitor SDA and SCL continuously. If one of them detects that SDA is low while it should actually be high on the next rising edge of SCL, it will stop the current transmission immediately. This process is called *arbitration* and illustrated in the following diagram.

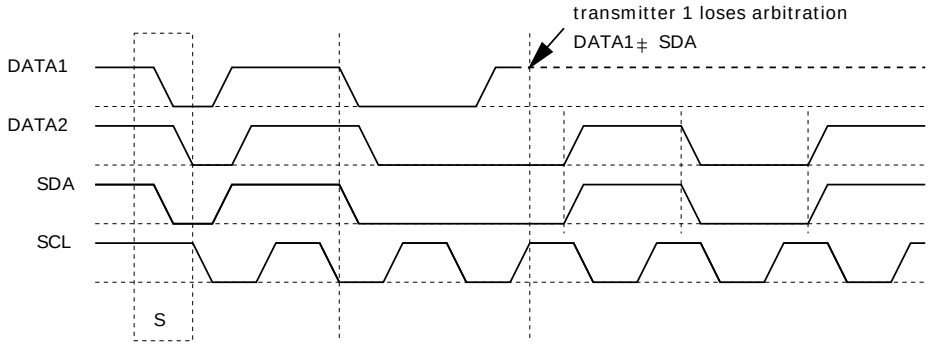


Figure 9-37: I2C Arbitration

9.3. Peripheral Registers

9.3.1. I2C Register Description

The I2C peripheral provides five 18-bit registers which are mapped to the SpartanMC address space. In the following, the layout of each register is described in more detail.

Table 9-33: I2C registers

Offset	Name	Access	Description
0	CONTROL	r/w	Contains a 16-bit clock divider and two enable bits for the I2C master itself and the interrupt controller respectively.
1	TX	w	Contains the current byte to be sent.
2	RX	r	Contains the current recieved byte.
3	COMMAND	w	Used to set I2C commands.
4	STATUS	r	Contains the controller status flags.

9.3.2. CONTROL Register

Table 9-34: I2C control register layout

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

9.3.3. TX Register

Table 9-35: I2C transmit data register layout

Bit	Name	Access	Default	Description
0-7	TX	w	0	Register for data to be sent.
8-17	-	w	0	Not used.

9.3.4. RX Register

Table 9-36: I2C receive data register layout

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

9.3.5. COMMAND Register

Table 9-37: I2C command register layout

Bit	Name	Access	Default	Description
0	IACK	r/w	0	Interrupt acknowledge. If set to 1 the pending interrupt will be cleared.
1-2	-	r/w	0	Not used.
3	ACK	r/w	0	If set to 0, acknowledge (0) will be sent. Otherwise, not acknowledge (1) will be sent.
4	WR	r/w	0	If WR = 1, the data in the TX register will be written to slave.
5	RD	r/w	0	If RD = 1, the RX register will be filled with data from slave.
6	STO	r/w	0	Send stop signal.
7	STA	r/w	0	Send (re-)start signal.
8-17	-	r/w	0	Not used.

Note: If both WR and RD are set to 1 at the same time, the read operation will be carried out.

9.3.6. STATUS Register

Table 9-38: I2C status register layout

Bit	Name	Access	Default	Description
0	IF	r	0	This bit is set to 1 when an interrupt is pending and IEN in Control register has been set. An interrupt occurs, if: • A byte transfer has been completed. • The arbitration has been lost.
1	TIP	r	0	Is set to 1 when a transfer is in progress.
2-4	-	r	0	Not used.
5	AL	r	0	Is set to 1 if the arbitration has been lost.
6	I2C_BUSY	r	0	Is set to 1 after a start signal has been detected and set to 0 after a stop signal has occurred.
7	RX_ACK	r	0	Is set to 1 if a not acknowledge (NAK) has been received.
8-17	-	r	0	Not used.

9.3.7. I2C C-Header i2c_master.h for Register Description

```
#ifndef __I2C_MASTER_
#define __I2C_MASTER_

#ifdef __cplusplus
extern "C" {
#endif

/*
* Definitions for the Opencores i2c master core
*/
// Rückgabewerte für non blocking read
#define   I2C_OK      0
#define   I2C_NO_ACK   1

/* --- Definitions for i2c master's registers --- */

/* ----- Read-write access                                    */

//#define I2C_PRER   0x00   /* Low byte clock prescaler
register */
#define   I2C_CTR      0x00   /* Control
register           */

/* ----- Write-only registers                                 */

#define   I2C_TXR      0x01   /* Transmit byte
register         */
#define   I2C_CR      0x03   /* Command
register           */

/* ----- Read-only registers                                 */

#define   I2C_RXR      0x02   /* Receive byte
register         */
#define   I2C_SR      0x04   /* Status
register             */

/* ----- Bits definition                                    */

/* ----- Control register                                    */

#define   I2C_EN      (1<<16)   /* Core enable
bit:            */
           /*   1 - core is enabled      */
           /*   0 - core is disabled      */
```

```
#define   I2C_IEN      (1<<17)   /* Interrupt enable
bit         */
            /*   1 - Interrupt enabled      */
            /*   0 - Interrupt disabled      */
            /* Other bits in CR are reserved   */

/* ----- Command register bits                              */
#define I2C_STA      (1<<7)   /* Generate (repeated) start
condition*/
#define I2C_STO      (1<<6)   /* Generate stop
condition          */
#define I2C_RD       (1<<5)   /* Read from
slave               */
#define I2C_WR       (1<<4)   /* Write to slave              */
#define I2C_NAK      (1<<3)   /* Acknowledge send to
slave      */
            /*   0 - ACK               */
            /*   1 - NACK               */
#define   I2C_ACK      0
#define I2C_IACK   (1<<0)   /* Interrupt acknowledge         */

/* ----- Status register bits                              */

#define I2C_RXACK   (1<<7)   /* ACK received from
slave         */
            /*   0 - ACK               */
            /*   1 - NACK               */
#define I2C_BUSY   (1<<6)   /* Busy bit                  */
#define I2C_AL       (1<<5)   /* Arbitration lost           */
#define I2C_R_W      (1<<2)   /* last byte read or
write         */
            /*   0 - READ               */
            /*   1 - WRITE               */
#define I2C_TIP      (1<<1)   /* Transfer in
progress          */
#define I2C_IF       (1<<0)   /* Interrupt flag              */

/* bit testing and setting macros                           */

#define ISSET(reg,bitmask)   ((reg)&(bitmask))
#define ISCLEAR(reg,bitmask)   (!(ISSET(reg,bitmask)))
#define BITSET(reg,bitmask)   ((reg)|(bitmask))
#define BITCLEAR(reg,bitmask)   ((reg)|(~(bitmask)))
#define BITTOGGLE(reg,bitmask)   ((reg)^(bitmask))
#define REGMOVE(reg,value)   ((reg)=(value))

typedef   volatile struct {
```

```
   volatile unsigned int   ctrl;   // (r/w)
   volatile unsigned int   txr;   // (r/w)
   volatile unsigned int   rxr;   // (r)
   volatile unsigned int   cmd;   // (r/w)
   volatile unsigned int   stat;   // (r)
} i2c_master_regs_t;

#ifdef __cplusplus
}
#endif

#endif //define __I2C_MASTER
```

9.3.8. Basic Usage of the I2C Registers

The structure shown above serves as interface between hardware and software. It can be used directly in a C program by including the header file `<i2c_master.h>` . This section presents several quite simple examples to illustrate the usage of this register.

First, assume that `I2C_MASTER_0` is a pointer which contains the physical address of an I2C master.

- **Example 1 : Enable the I2C master and set the prescaler to 134**

```
I2C_MASTER_0->ctrl = I2C_EN | 134;
```

- **Example 2 : Send write request to the slave at the address 0x70**

```
I2C_MASTER_0->txr = 0x70 << 1; // or 0xE0
I2C_MASTER_0->cmd = I2C_WR | I2C_STA;
```

- **Example 3 : Check if the current 8-bit packet has been transfered**

```
/* wait as long as TIP is set */
while(I2C_MASTER_0->stat & I2C_TIP);

/* do something here */
```

- **Example 4 : Check if a not acknowledge has been received**

```
if(I2C_MASTER_0->stat & I2C_RXACK)
    return I2C_NO_ACK;
```

- **Example 5 : Write a constant value 0xFF to the slave**

```
I2C_MASTER_0->txr = 0xFF;
I2C_MASTER_0->cmd = I2C_WR;
```

- **Example 6 : Send read request to the slave at the address 0x70**

```
I2C_MASTER_0->txr = (0x70 << 1) + 1; // or 0xE1
```

```
I2C_MASTER_0->cmd = I2C_WR | I2C_STA;
```

- **Example 7 : Read one last packet from the slave**

```
int v;
I2C_MASTER_0->cmd = I2C_RD | I2C_NAK | I2C_STO;
while(I2C_MASTER_0->stat & I2C_TIP);
v = I2C_MASTER_0->rxr;
```

Note: Sometimes, a hardware manufacturer may give an 8-bit slave ID instead of a 7-bit address. This ID is actually equal `address << 1` and implies that the direction bit is 0. Therefore, it can be sent to the slave as write request directly and `ID + 1` can be used as read request.

10. 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 10-39: 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.

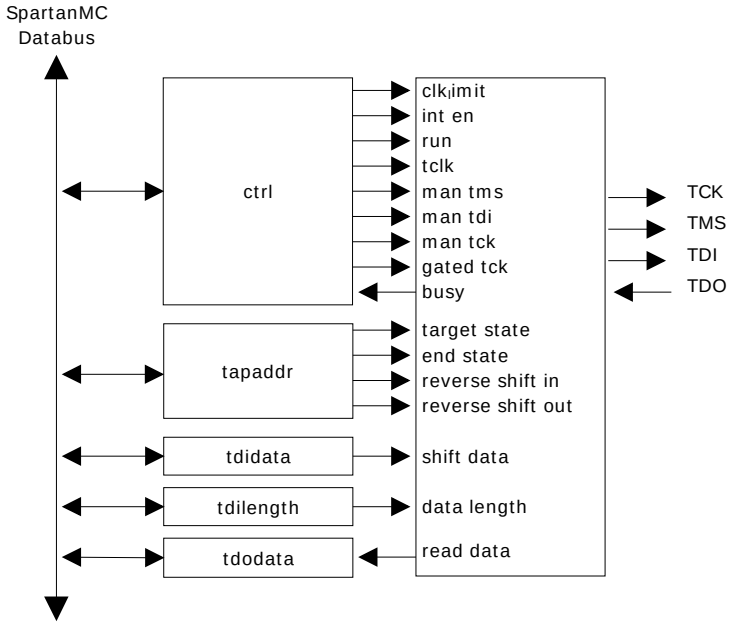


Figure 10-38: JTAG block diagram

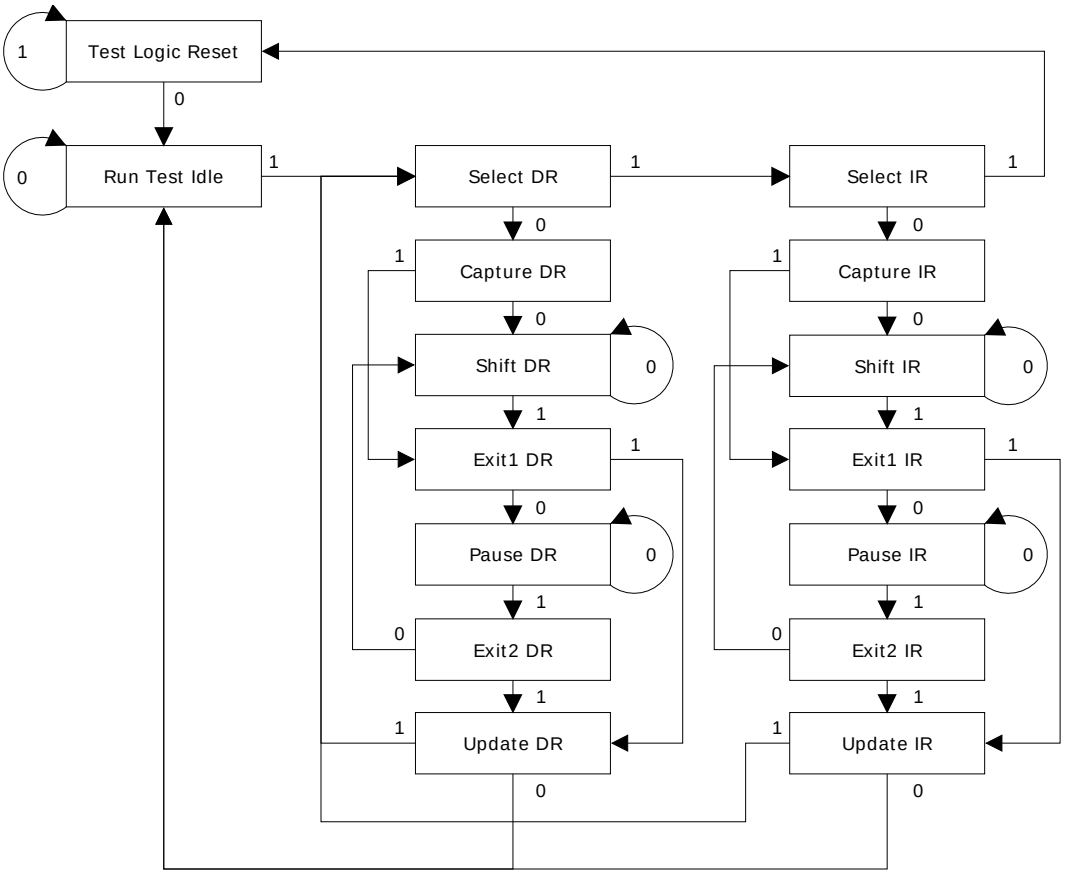


Figure 10-39: JTAG TAP Controller State Machine

10.1. 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.

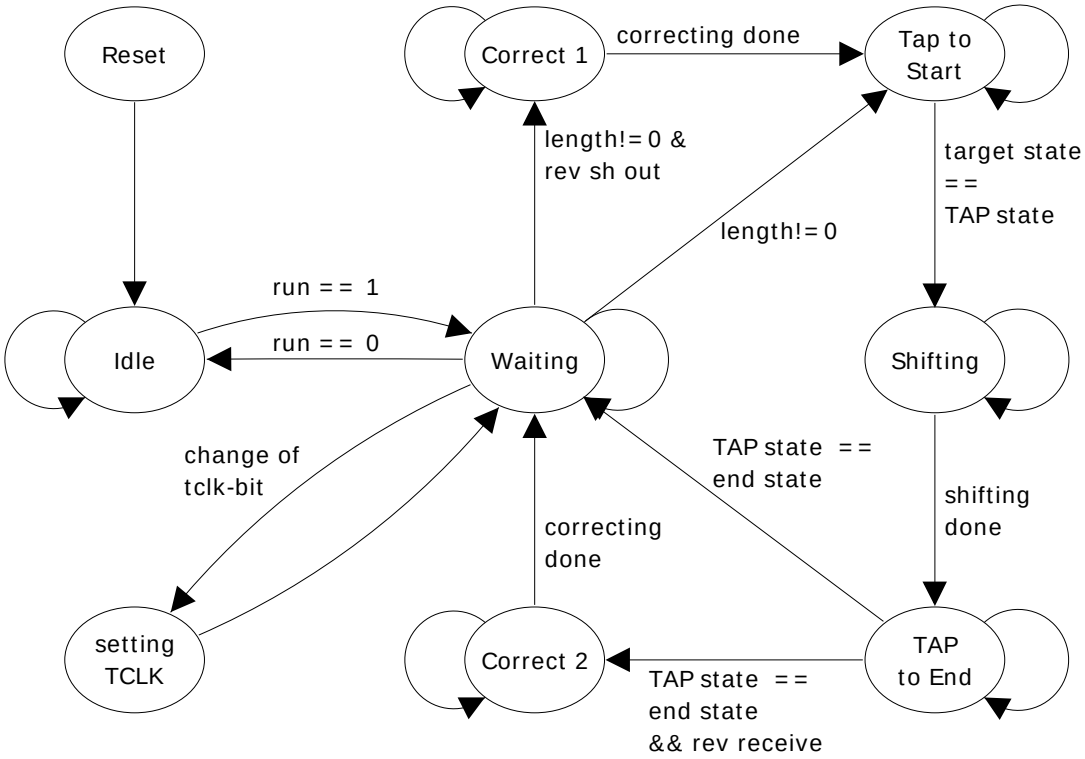


Figure 10-40: JTAG State machine

10.2. Module parameters

This module does not have Synthesis-Parameter

10.3. Peripheral Registers

10.3.1. JTAG Register Description

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

Table 10-40: 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.

10.3.2. JTAG Control Register (ctrl)

Table 10-41: 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 10-41: JTAG control register layout

10.3.3. JTAG TAP Control Register (tapaddr)

Table 10-42: 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 10-42: JTAG TAP control register layout

11. 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.

11.1. Module Parameters

Table 11-43: 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.

11.2. Peripheral Registers

11.2.1. 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 11-44: 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.

11.2.2. PORT_OUT C-Header for Register Description

```
#ifndef __PORT_OUT_H
#define __PORT_OUT_H

#ifdef __cplusplus
extern "C" {
#endif

#define PORT_OUTBIT_0 (1<<0)
#define PORT_OUTBIT_1 (1<<1)
#define PORT_OUTBIT_2 (1<<2)
#define PORT_OUTBIT_3 (1<<3)
#define PORT_OUTBIT_4 (1<<4)
#define PORT_OUTBIT_5 (1<<5)
#define PORT_OUTBIT_6 (1<<6)
#define PORT_OUTBIT_7 (1<<7)
#define PORT_OUTBIT_8 (1<<8)
#define PORT_OUTBIT_9 (1<<9)
#define PORT_OUTBIT_10 (1<<10)
#define PORT_OUTBIT_11 (1<<11)
#define PORT_OUTBIT_12 (1<<12)
#define PORT_OUTBIT_13 (1<<13)
#define PORT_OUTBIT_14 (1<<14)
#define PORT_OUTBIT_15 (1<<15)
#define PORT_OUTBIT_16 (1<<16)
#define PORT_OUTBIT_17 (1<<17)

typedef struct port_out {
   volatile unsigned int data; // (r/w)
   volatile unsigned int oe; // (r/w)
} port_out_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```

12. 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.

12.1. Module Parameters

Table 12-45: 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.

12.2. 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.

12.3. Peripheral Registers

12.3.1. 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 12-46: 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.

12.3.2. PORT_IN C-Header for Register Description

```
#ifndef __PORT_IN_H
#define __PORT_IN_H

#ifdef __cplusplus
extern "C" {
#endif

#ifndef __PORT_IN_F_H

#define PORT_INBIT_0   (1<<0)
#define PORT_INBIT_1   (1<<1)
#define PORT_INBIT_2   (1<<2)
#define   PORT_INBIT_3   (1<<3)
#define PORT_INBIT_4   (1<<4)
#define PORT_INBIT_5   (1<<5)
#define PORT_INBIT_6   (1<<6)
#define PORT_INBIT_7   (1<<7)
#define PORT_INBIT_8   (1<<8)
#define PORT_INBIT_9   (1<<9)
#define PORT_INBIT_10   (1<<10)
#define PORT_INBIT_11   (1<<11)
#define PORT_INBIT_12   (1<<12)
#define PORT_INBIT_13   (1<<13)
#define PORT_INBIT_14   (1<<14)
#define PORT_INBIT_15   (1<<15)
#define PORT_INBIT_16   (1<<16)
#define PORT_INBIT_17   (1<<17)

#endif
```

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

#ifdef __cplusplus
}
#endif

#endif
```

13. 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.)

13.1. Module Parameters

Table 13-47: 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).

13.2. 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.

13.3. Peripheral Registers

13.3.1. 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 13-48: 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.

13.3.2. PORT_BI C-Header for Register Description

```
#ifndef __PORT_BI_H
#define __PORT_BI_H

#ifdef __cplusplus
extern "C" {
#endif

#define PORT_IOBIT_0   (1<<0)
#define PORT_IOBIT_1   (1<<1)
#define PORT_IOBIT_2   (1<<2)
#define PORT_IOBIT_3   (1<<3)
#define PORT_IOBIT_4   (1<<4)
#define PORT_IOBIT_5   (1<<5)
#define PORT_IOBIT_6   (1<<6)
#define PORT_IOBIT_7   (1<<7)
#define PORT_IOBIT_8   (1<<8)
#define PORT_IOBIT_9   (1<<9)
#define PORT_IOBIT_10 (1<<10)
#define PORT_IOBIT_11 (1<<11)
#define PORT_IOBIT_12 (1<<12)
#define PORT_IOBIT_13 (1<<13)
#define PORT_IOBIT_14 (1<<14)
#define PORT_IOBIT_15 (1<<15)
#define PORT_IOBIT_16 (1<<16)
#define PORT_IOBIT_17 (1<<17)

typedef struct port_bi {
   volatile unsigned int data; // (r/w) (reset-interrupt)
   volatile unsigned int ie; // (r/w) (reset-interrupt)
   volatile unsigned int oe; // (r/w)
   volatile unsigned int dir; // (r/w) (reset-interrupt) 1 =
Input
   volatile unsigned int edgsel; // (r/w) (reset-interrupt) 1 =
positiv
   volatile unsigned int ir_stat; // (r)
} port_bi_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```

14. SpartanMC Core Hardware Debugging Support

The SpartanMC Core contains optional Hardware Debugging Support, that allows hardware breakpoints, watchpoints and stepping. Hardware Debugging is enabled on a per-Core basis with the **HARDWARE_DEBUGGING** flag. The number of available break-/watchpoints can be configured from JConfig, as well as the trap indexes that will be called upon registering a hit on either breakpoint, watchpoint or step event.

Breakpoints are placed on single instruction addresses

Watchpoints are placed on Word-Level Memory addresses (each address maps to an 18-bit word). Their configuration allows limiting a watchpoint to lower/upper byte (9-bit) or watching only read or write access.

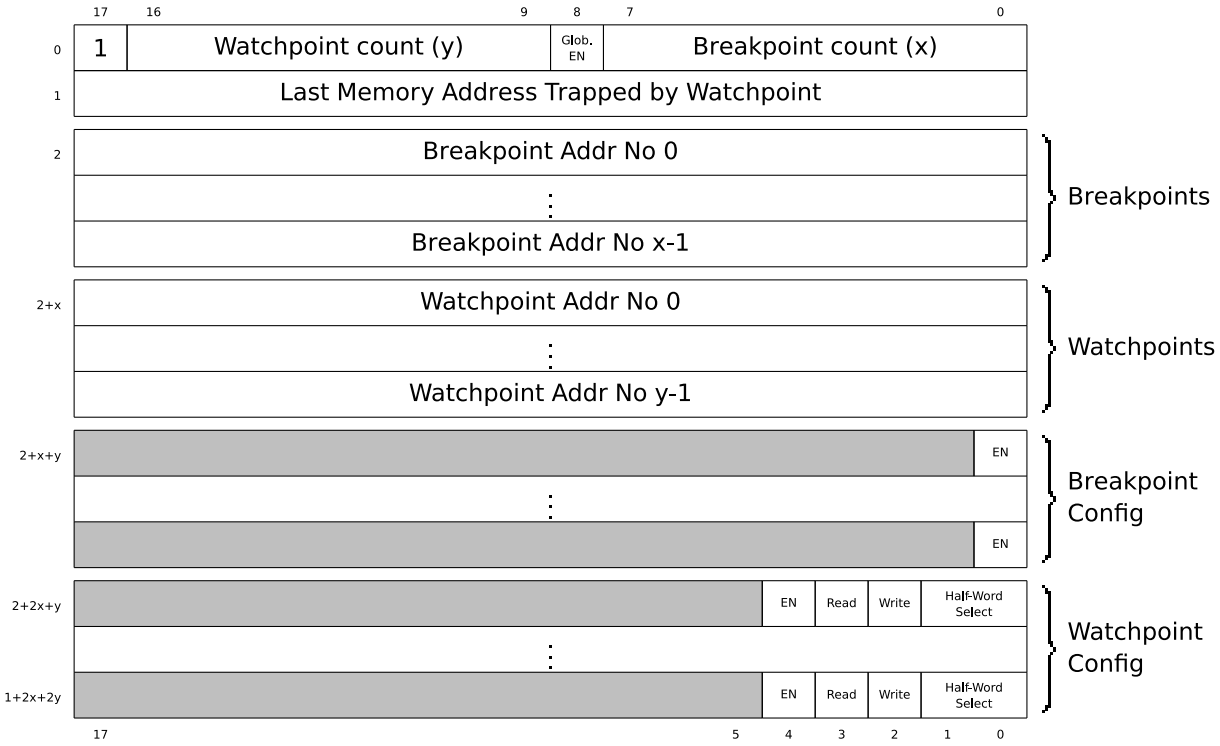


Figure 14-41: Hardware Debugging Registers

14.1. Access

All registers are accessed indirectly by using SFR_DBG_IDX as index, and SFR_DBG_DAT as data.

14.2. Hardware Debugging Status Register (idx 0)

The first register allows detection of the presence of Hardware debugging support (bit 18 is always 1 if present). It also contains the number of break-/watchpoints that were synthesized.

Only writing the following values to the first register has a function

0x00000: turn of Hardware Debugging Functionality globally

0x00001 or 0x00100: turn on Hardware Debugging Functionality globally

0x00002 or 0x00003: Execute a single step upon leaving the current register window *0x00003 combines global enable with single step*

14.3. Hardware Behavior

The main work is done by the "Breakpoint Manager". After a reset it will always be disabled. If it is enabled by writing to the**Hardware Debugging Status Register**, it will internally delay its activation, until the register window has changed. This is so that it can never trap a breakpoint in the breakpoint handler code itself, but consequently means, it *must* be activated from its own function. Additionally, breakpoints on the first instruction after activation, may be ignored. Either, if the manager is in single stepping mode (to force it execute at least 1 instuction) or if the address is that of the last breakpoint. The last breakpoint is initialized to 0.

14.4. Last Trap Register

Deprectated: *This register is still present internally, but not curently exposed via Specialfunction Registers.* This information is convenient to gauge, which breakpoint actually hit, but is irrelevant to the debugger, because if it differs from the actual PC, we will have already missed the breakpoint. Software should not place breakpoints on delay slots.

Contains the exact PC of the instruction that caused a trap last. Read-Only. Accounts for delayed jumps/branches. Accounts for pipeline for watchpoints: It will point to instruction causing the memory access, even though several other instructions have since been queued into the pipeline.

14.4.1. Last Trapped Memory Adress Register (idx 1)

Contains the lower 18 bits of the memory address that caused the latest watchpoint. Read-Only.

15. 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.

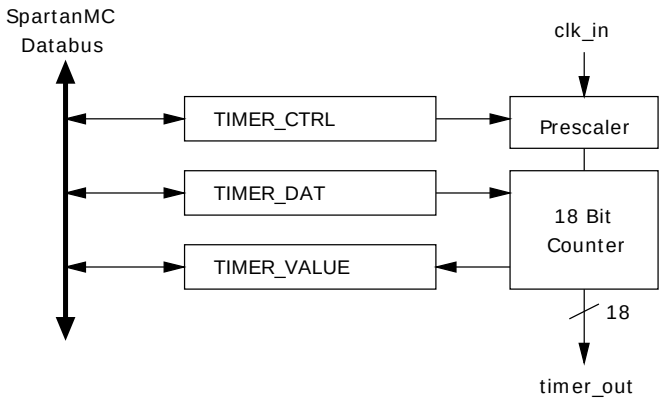


Figure 15-42: Timer block diagram

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.

15.1. Module parameters

Table 15-49: 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.**

15.2. Peripheral Registers

15.2.1. 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 15-50: 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.

15.2.2. TIMER_CTRL Register

Table 15-51: 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 15-51: TIMER_CTRL register layout

15.2.3. TIMER_DAT Register

Table 15-52: TIMER_DAT register layout

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

15.2.4. TIMER_VALUE Register

Table 15-53: 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.

15.2.5. TIMER C-Header for Register Description

```
#ifndef __TIMER_H
#define __TIMER_H

#ifdef __cplusplus
extern "C" {
#endif

#define TIMER_EN      (1<<0)
#define TIMER_PRE_EN   (1<<1)
#define TIMER_PRE_VAL   (1<<2)      // *0 fuer 2^1 bis *7 fuer
2^8
#define TIMER_PRE_2   (TIMER_PRE_VAL * 0)
#define TIMER_PRE_4   (TIMER_PRE_VAL * 1)
#define TIMER_PRE_8   (TIMER_PRE_VAL * 2)
#define TIMER_PRE_16   (TIMER_PRE_VAL * 3)
#define TIMER_PRE_32   (TIMER_PRE_VAL * 4)
#define TIMER_PRE_64   (TIMER_PRE_VAL * 5)
#define TIMER_PRE_128 (TIMER_PRE_VAL * 6)
#define TIMER_PRE_256 (TIMER_PRE_VAL * 7)

typedef struct timer {
```

```
   volatile unsigned int control;
   volatile unsigned int limit;
   volatile unsigned int value;
} timer_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```

16. 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.)

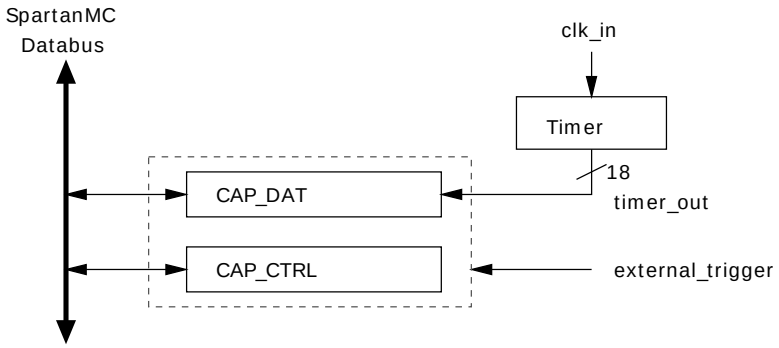


Figure 16-43: Capture module block diagram

16.1. 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.

16.2. Module parameters

Table 16-54: 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.**

16.3. Peripheral Registers

16.3.1. 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 16-55: 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)

16.3.2. CAP_DAT Register

Table 16-56: CAP_DAT register layout

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

16.3.3. CAP_CTRL Register

Table 16-57: 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 16-57: CAP_CTRL register layout

16.3.4. TIMER_CAP C-Header for Register Description

```
#ifndef __CAPTURE_H
#define __CAPTURE_H

#ifdef __cplusplus
extern "C" {
#endif

#define CAPTURE_EN            (1 << 0)
#define CAPTURE_EN_INT         (1 << 1)
#define CAPTURE_EDGE         (1 << 2)

#define CAPTURE_NON            (CAP_EDGE * 0)
#define CAPTURE_FALLING_EDGE   (CAP_EDGE * 2)
#define CAPTURE_RISING_EDGE      (CAP_EDGE * 3)
#define CAPTURE_LOW_LEVEL      (CAP_EDGE * 4)
#define CAPTURE_HIGH_LEVEL      (CAP_EDGE * 5)
#define CAPTURE_ANYTHING_EDGE   (CAP_EDGE * 7)

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

#ifdef __cplusplus
}
#endif

#endif
```

17. 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.)

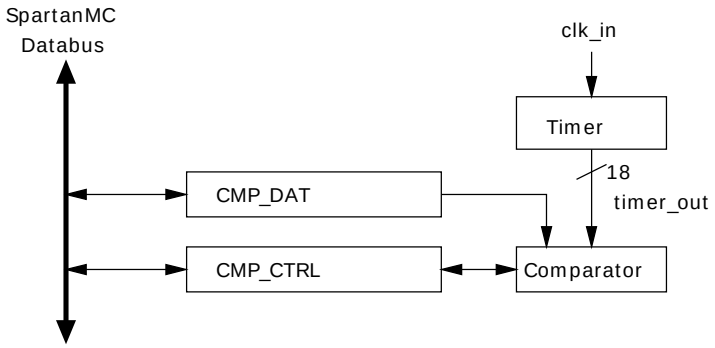


Figure 17-44: Timer compare module block diagram

17.1. 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.

17.2. Module parameters

Table 17-58: 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.**

17.3. Peripheral Registers

17.3.1. 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 17-59: 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.

17.3.2. Compare Control Register

Table 17-60: 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 17-60: CMP_CTRL register layout

17.3.3. Compare Value Register

Table 17-61: CMP_DAT register layout

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

17.3.4. TIMER_CMP C-Header for Register Description

```
#ifndef __COMPARE_H
#define __COMPARE_H

#ifdef __cplusplus
extern "C" {
#endif

#define COMPARE_EN      (1 << 0)   // Compare Enable
#define COMPARE_EN_INT   (1 << 1)   // Compare Interrupt Enable
#define COMPARE_MODE      (1 << 2)   // Mode Bit 0

#define COMPARE_NON_FRQ   (COMPARE_MODE * 0) // Ausgang bleibt
gleich
#define COMPARE_SET_OUT   (COMPARE_MODE * 1) // Ausgang
setzen(=1)
#define COMPARE_CLEAR_OUT (COMPARE_MODE * 2) // Ausgang
zurücksetzen(=0)
#define COMPARE_TOGGLE_OUT (COMPARE_MODE * 3) // Ausgang
negieren
#define COMPARE_NON_IMP   (COMPARE_MODE * 4) // Ausgang bleibt
gleich
#define COMPARE_C0_N1   (COMPARE_MODE * 6) // Ausgang auf 0
wenn Timer = CMP_DAT ist -- Ausgang auf 1 wenn Timer = 0 ist.
```

```
#define COMPARE_C1_N0   (COMPARE_MODE * 7) // Ausgang auf 1
wenn Timer = CMP_DAT ist -- Ausgang auf 0 wenn Timer = 0 ist.

#define COMPARE_EN_OUT   (1 << 5)   // Compare Output Enable
#define COMPARE_VAL_OUT   (1 << 6)   // Compare Output Value

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

#ifdef __cplusplus
}
#endif

#endif
```

18. 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.

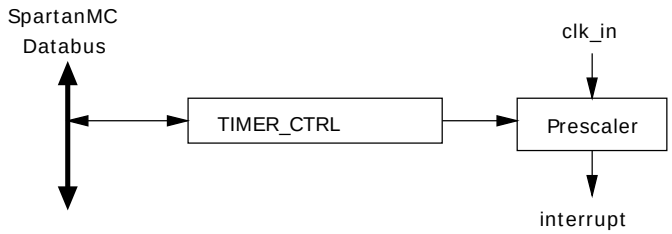


Figure 18-45: Timer RTI block diagram

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

18.1. Interrupts

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

18.2. Module Parameters

Table 18-62: 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.**

18.3. Peripheral Registers

18.3.1. 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 18-63: 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)

18.3.2. RTI_CTRL Register

Table 18-64: 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 18-64: RTI_CTRL register layout

18.3.3. RTI C-Header for Register Description

```
#ifndef __RTI_H
#define __RTI_H

#ifdef __cplusplus
extern "C" {
#endif

#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 ctrl;
} rti_regs_t;

#ifdef __cplusplus
}
#endif
```

```
#endif
```

19. 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.

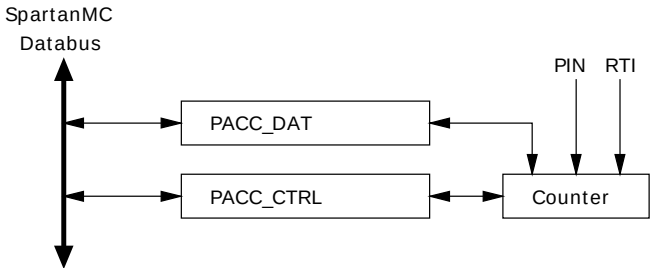


Figure 19-46: Timer Pulse Accumulator block diagram

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).

19.1. Module Parameters

Table 19-65: 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.**

19.2. Peripheral Registers

19.2.1. 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 19-66: 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.

19.2.2. PACC_CTRL Register

Table 19-67: 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 19-67: PACC_CTRL register layout

19.2.3. PACC_DAT Register

Table 19-68: PACC Counter register layout

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

19.2.4. PACC C-Header for Register Description

```
#ifndef __COUNTER_H
#define __COUNTER_H

#ifdef __cplusplus
extern "C" {
#endif

#define COUNTER_EN      (1 << 0)
#define COUNTER_MODE   (1 << 1)

#define COUNTER_INPMODE   (COUNTER_MODE * 0)
#define COUNTER_RTIMODE   (COUNTER_MODE * 1)

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

#ifdef __cplusplus
}
#endif

#endif
```

20. 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.)

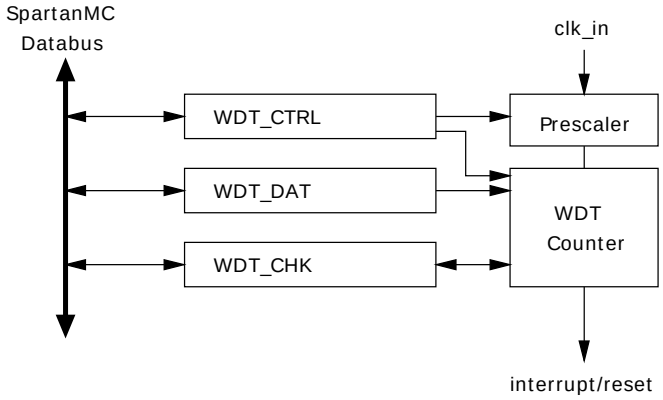


Figure 20-47: Watchdog timer block diagram

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).

20.1. 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 .

20.2. Module Parameters

Table 20-69: 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.

20.3. 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.

20.4. Peripheral Registers

20.4.1. 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 20-70: 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.**

20.4.2. WDT_CTRL Register

Table 20-71: 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 20-71: WDT_CTRL register layout

20.4.3. WDT_DAT Register

Table 20-72: WDT maximum value register layout

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

20.4.4. WDT_CHK Register

Table 20-73: 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.

20.4.5. WDT C-Header for Register Description

```
#ifndef __WATCHDOG_H
#define __WATCHDOG_H

#ifdef __cplusplus
extern "C" {
#endif

#define WATCHDOG_EN         (1<<0)
#define WATCHDOG_PRE_EN      (1<<1)
#define WATCHDOG_PRE_VAL   (1<<2)      // *0 fuer 2^1 bis *7
fuer 2^8

#define WATCHDOG_PRE_2      (WATCHDOG_PRE_VAL * 0)
#define WATCHDOG_PRE_4      (WATCHDOG_PRE_VAL * 1)
#define WATCHDOG_PRE_8      (WATCHDOG_PRE_VAL * 2)
#define WATCHDOG_PRE_16      (WATCHDOG_PRE_VAL * 3)
#define WATCHDOG_PRE_32      (WATCHDOG_PRE_VAL * 4)
#define WATCHDOG_PRE_64      (WATCHDOG_PRE_VAL * 5)
#define WATCHDOG_PRE_128   (WATCHDOG_PRE_VAL * 6)
#define WATCHDOG_PRE_256   (WATCHDOG_PRE_VAL * 7)

#define   WATCHDOG_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)
} watchdog_regs_t;
#ifdef __cplusplus
}
#endif

#endif
```

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

21.1. 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 ausloesen, 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 die drei erzeugten Signale ist nur noch folgende externe Beschaltung notwendig:

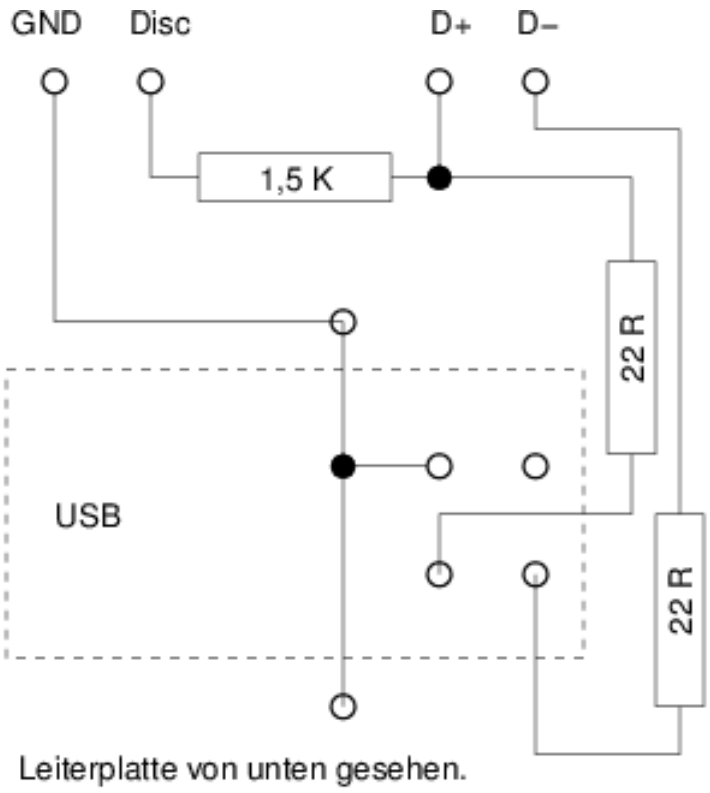


Figure 21-48: Der 1,5K Widerstand(external link) zieht D+ bei Disc=1 auf 3,3V wodurch das Interface im FULL-Speed Mode angemeldet wird.

21.2. Speicherorganisation

Die Basiadresse des USB Moduls liegt oberhalb des Arbeitsspeichers der Konfiguration. Die Adressen (Offset) 0x000 bis 0x07f des DMA-Speichers sind fuer 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 Berich ab 0x280 bleibt unbenutzt. Ist Double Buffering aktiviert werden in aufsteigender Reihenfolge jedem der Endpunkte 0 bis 13 zwei Puffer zugeordnet. Enpunkt 14 und 15 kann nicht verwendet werden! Siehe Adresstabelle. Die aktuelle Implementierung der Hardware kann nur maximal 6 Endpunkte verwalten!

21.3. Konfigurations- und Statusregister

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

Table 21-74: Die aktuelle Implementierung unterstuetzt nur 6 Endpunkte!

21.4. Descriptoren (read only)

Offset	Bemerkung
0x21	Device Descriptor
0x2a	Configuration Descriptor
0x68	Language Descriptor
0x6a	String Descriptor describing manufacturer
0x73	String Descriptor describing product
0x7c	String Descriptor describing serial number

Table 21-75: Descriptoren

21.5. Puffer

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

Table 21-76: Adressen der Puffer

Die aktuelle Implementierung unterstuetzt nur 6 Endpunkte! Die Anordnung der Bytes in den Puffern kann mit dem Parameter NOGAP veraendert werden. Mit NOGAP=0 werden die Bytes in der 9 Bit Anordnung des SpartanMC abgelegt. Diese Anordnung ist fuer die Uebertragung von Zeichenketten sinnvoll. Sollen 16 Bit Werte vom Spartan-MC in dem DMA-Puffer abgelegt werden, dann muessen 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
```

21.6. Bitbelegung der Register

21.6.1. epXc Register

Bit	Bezeichnu	Bedeutung	
6-0	Size	Anzahl zu sendender Bytes -1	Wert mit 0x3f maskieren (0 entspricht 1 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 enhaellt Daten, EP OUT: 1=Puffer leer

Table 21-77: epXc Register

21.6.2. epXs Register (read only)

Bit	Bezeichnung	Bedeutung
6-0	Anzahl empfangender Bytes -1	Anzahl der Bytes -1 nach dem Transfer vom Host
17-7	not used	

Table 21-78: epXs Register (read only)

21.6.3. Globales Steuerregister

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

Table 21-79: Globales Steuerregister

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

Die usb_init.h zur Vorinitialisierung der USB DMA Puffer muss fuer jedes Projekt angepasst werden. Dies betrifft die Intialiesierung der Anzahl von Endpunkten und die Initialisierung der Steuerregister fuer jeden Endpunkt epXc. Ein Beispiel mit 2 Endpunkten ist im PRNG Projekt aus dem Quickguide oder unter "src/doc/users-manual/src/usb11/usb_init.h" zu finden. **Ohne Iniialisierung der Descriptoren ist das Interface nicht funktionsfaehig und darf nicht durch setzen von Bit 17 im Globalen Steuerregister oder durch Aufruf der Funktion `void usb_init(usb11_dma_t *usb, unsigned int delay);` aktivert werden!**

```
#include "peripherals.h"   // I/O-Adessen und Registerstrukturen
aller installierten Interfaces
#include "moduleParameters.h"   // in der Hardwere eingestellte
Parameter aller installierten Interfaces
#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 tx nicht bereit kein Intr. setzen
```

```
   .ep02c = (              USB_CTRL_MODE | USB_CTRL_OUT    ), //
EP2 rx nicht bereit kein Intr. setzen
//   .ep03c = ( USB_CTRL_EN | USB_CTRL_INTR | USB_CTRL_MODE |
USB_CTRL_IN | 8 ), // 8 Byte sind im Puffer
//   .ep04c = ( USB_CTRL_EN | USB_CTRL_INTR | USB_CTRL_MODE |
USB_CTRL_OUT   ),
//   .ep05c = ( USB_CTRL_EN | USB_CTRL_INTR | USB_CTRL_MODE |
USB_CTRL_IN | 8 ), // 8 Byte sind im Puffer
//   .ep06c = ( 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
   },
   .ep05 = {
      .hdr.size      = 0x07,      //poll_interval - hdr.size +1
      .hdr.type      = USB_ENDPOINT_DESCRIPTOR,
      .adr         = {
         .direction   = USB_DIRECTION_IN,
         .number      = 5
      },
      .attr         = USB_ATTRIBUTE_BULK,
      .packet_size_low   = 0x40,
      .packet_size_high   = 0x00,
      .poll_interval      = 0x01
   },
   .ep06 = {
      .hdr.size      = 0x07,      //poll_interval - hdr.size +1
      .hdr.type      = USB_ENDPOINT_DESCRIPTOR,
      .adr         = {
         .direction   = USB_DIRECTION_OUT,
         .number      = 6
      },
      .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)
   .data02           = {0x06030,0x06030,0x06032,0x00000},   //
000002 EP01.1
   .data03           = {0x06030,0x06030,0x06033,0x00000},   //
000003 EP01.2
   .data04           = {0x06030,0x06030,0x06034,0x00000},   //
000004 EP02.1
   .data05           = {0x06030,0x06030,0x06035,0x00000},   //
000005 EP02.2
   .data06           = {0x06030,0x06030,0x06036,0x00000},   //
000006 EP03.1
   .data07           = {0x06030,0x06030,0x06037,0x00000},   //
000007 EP03.2
   .data08           = {0x06030,0x06030,0x06038,0x00000},   //
000008 EP04.1
   .data09           = {0x06030,0x06030,0x06039,0x00000},   //
000009 EP04.2
   .data10           = {0x06030,0x06030,0x06230,0x00000},   //
000010 EP05.1
   .data11           = {0x06030,0x06030,0x06231,0x00000},   //
000011 EP05.2
   .data12           = {0x06030,0x06030,0x06232,0x00000},   //
000012 EP06.1
   .data13           =  {0x06030,0x06030,0x06233,0x00000}   //
000013 EP06.3
};
```

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

```
#ifndef __USB_H
#define __USB_H

#ifdef __cplusplus
extern "C" {
#endif

#include <peripherals/usb11.h>

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

#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*)(((volatile unsigned
char*)usb_base) + ep_num * 4), \
   .data = (volatile unsigned int*)(((volatile unsigned
char*)usb_base) + 0x100 + ep_num * 64), \
   .usb = usb_base, \
   .index = (ep_num) \
}

#define USB_ENDPOINT_DB(usb_base, ep_num) { \
   .ctrl = (volatile unsigned int*)(((volatile unsigned
char*)usb_base) + ep_num * 4), \
   .data = (volatile unsigned int*)(((volatile unsigned
char*)usb_base) + 0x100 + ep_num * 128), \
   .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);
int 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);

#ifdef __cplusplus
}
#endif

#endif /*USB_H_*/
```

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

```
#ifndef __USB11_H
#define __USB11_H

#ifdef __cplusplus
extern "C" {
#endif

//////////////////////////////////////////////////////////////////////
//                                                                  //
// 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
};

/*
// Konfigurationsregister USB-Endpunkt
struct __attribute__((__packed__)) usb_control {
      unsigned en         :1;   //   [17] Enable
      unsigned intr      :1;   //   [16] Interrupt enable
      unsigned mode      :1;   //   [15] Uebertragungsmodus
                              //         1 = synchron
                              //         0 = asynchron
      unsigned control      :1;   //   [14] Kontrollendpunkt
      unsigned out         :1;   //   [13] Endpunkt OUT
      unsigned in         :1;   //   [12] Endpunkt IN
      unsigned bufsel      :1;   //   [11] Pufferauswahl
                              //         1 = oberer
                              //         0 = unterer
      unsigned           :4;   // [10: 7] reserviert
      unsigned size      :7;   // [ 6: 0] Zeichenanzahl EP IN
};

// Konfigurationsregister USB global
struct __attribute__((__packed__)) usb_global {
      unsigned en         :1;   // [17] USB-Interface Enable
      unsigned ien_ep0      :1;   // [16] Interruptfreigabe
fuer EP0
      unsigned iep15      :1;   // [15] Reset Interrupt EP 15
      unsigned iep14      :1;   // [14] Reset Interrupt EP 14
      unsigned iep13      :1;   // [13] Reset Interrupt EP 13
```

```
      unsigned iep12      :1;   // [12] Reset Interrupt EP 12
      unsigned iep11      :1;   // [11] Reset Interrupt EP 11
      unsigned iep10      :1;   // [10] Reset Interrupt EP 10
      unsigned iep09      :1;   // [ 9] Reset Interrupt EP 9
      unsigned iep08      :1;   // [ 8] Reset Interrupt EP 8
      unsigned iep07      :1;   // [ 7] Reset Interrupt EP 7
      unsigned iep06      :1;   // [ 6] Reset Interrupt EP 6
      unsigned iep05      :1;   // [ 5] Reset Interrupt EP 5
      unsigned iep04      :1;   // [ 4] Reset Interrupt EP 4
      unsigned iep03      :1;   // [ 3] Reset Interrupt EP 3
      unsigned iep02      :1;   // [ 2] Reset Interrupt EP 2
      unsigned iep01      :1;   // [ 1] Reset Interrupt EP 1
      unsigned iep00      :1;   // [ 0] Reset Interrupt EP 0
};
*/
// 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
      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;
#ifdef __cplusplus
}
#endif

#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_).

22. SpartanMC CAN Interface -- Controller area network

22.1. 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.

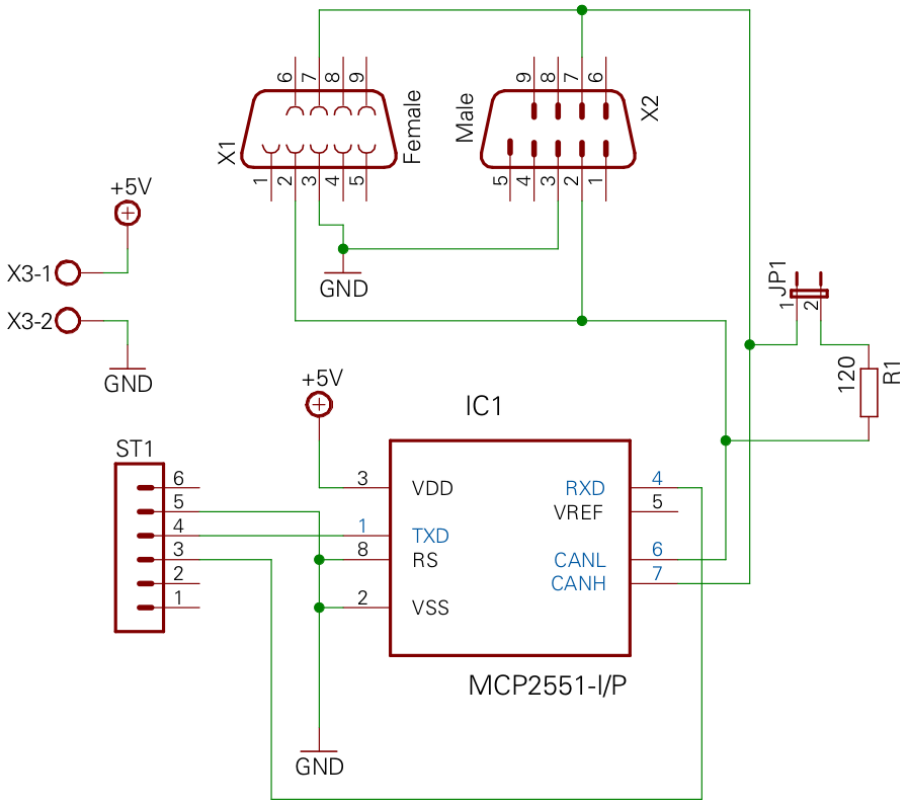


Figure 22-49: Anbindung des CAN-Interface

22.2. 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.

22.3. 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.

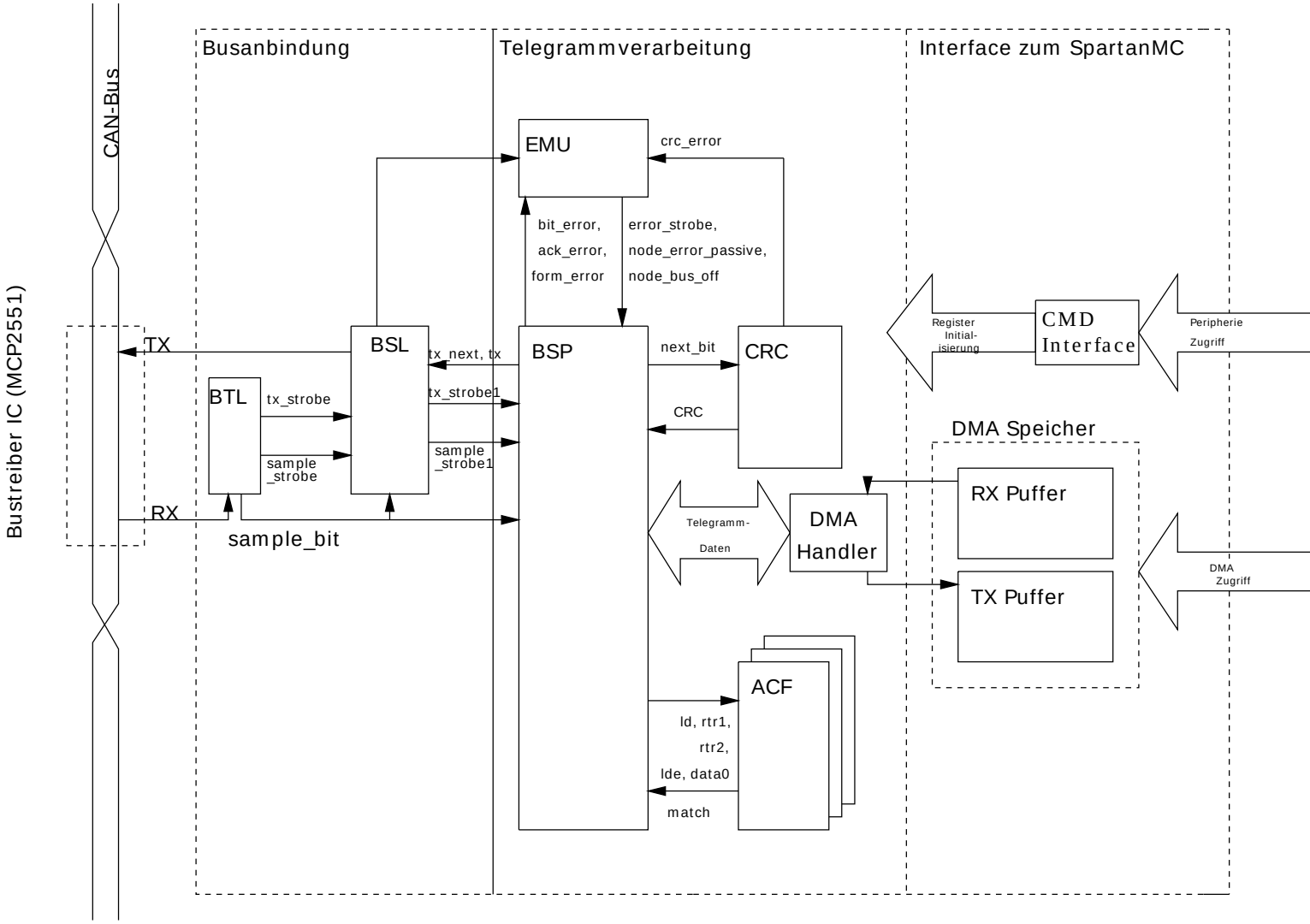


Figure 22-50: CAN-Interface block diagram

22.4. Konfigurations- und Statusregister

Offse	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 01 = Test Mode 10 = Listen Mode 11 = Konfigurations Mode	h3
		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

Offse	Register	Bit	Verilog Name	Bemerkung (Rechte)	Initialwert
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**	
		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

Offse	Register	Bit	Verilog Name	Bemerkung (Rechte)	Initialwert
		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 = 001 = Form-Error 010 = Stuff-Error 011 = ACK-Error 100 = CRC-Error 101 = Bit-Error 110 = 111 = alle anderen	h0
		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	h00

Offse	Register	Bit	Verilog Name	Bemerkung (Rechte)	Initialwert
				1 0111 = 1 1000 = crc_lim 1 1001 = ack 1 1010 = eof 1 1011 = ack_lim 1 1100 = overload 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 22-80: Die aktuelle Implementierung unterstützt maximal 18 Rx Puffer, 18 Tx Puffer und 18 Filter

22.5. 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.
- Mit SJW wird entweder die Periodendauer vergrößert durch **(tseg1 + SJW)** oder verkleinert mit **(tseg2 - SJW)** um den Synchronisationszeitpunkt von Sender und Empfänger wieder anzugleichen.
- 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 22-81: 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

Name	Bedeutung	Wertebeeich
tseg2	Time Segment 2 -2 (= Phase_Segment_2 -2	0 ... 7

Table 22-82: Bedeutung der Bezeichner

22.6. 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
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 22-83: Adressen der Tx Puffer

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

22.7. 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
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 22-84: Adressen der Rx Puffer

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

22.8. 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	hexadezir	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)
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)

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
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
ooo				ooo
0x08F	Tx18 / frei			

Table 22-85: 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

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	
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)
°°°				°°°
0x11F	Rx18 / CRC	2	0x00000	CRC (00 0110 0000 1000 1110) frei

Table 22-86: Rx Puffer

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

```
//#include "peripherals.h"   // I/O-Adessen und
Registerstrukturen aller installierten Interfaces
//#include "moduleParameters.h"   // in der Hardwere
eingestellte Parameter aller installierten Interfaces
#include <can.h>

// Initialisierung der Tx DMA Telegramm Puffer

struct can_dma dma_can_ini
__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_ini.h zur Vorinitialisierung der CAN DMA Puffer ist im CAN Test Project zu finden" ./hw-tests/can-test/atlys/firmware/PCAN-longtime-tx/include".

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

```
#ifndef __CAN_HW_H
#define __CAN_HW_H

#ifdef __cplusplus
extern "C" {
#endif

// 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;
#ifdef __cplusplus
}
#endif

#endif
```

22.8.3. CAN C-Header for C-Funktion

```
#ifndef __CAN_H
#define __CAN_H

#ifdef __cplusplus
extern "C" {
#endif

#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);

#ifdef __cplusplus
}
#endif

#endif   /*CAN_H_*/
```

23. 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.

23.1. 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.

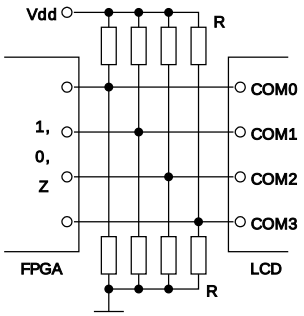


Figure 23-51: Circuit for connecting the LCD

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

23.1.1. Periphal registers

Table 23-87: 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 23-87: 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.

23.1.2. 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.

23.1.3. Module parameters

Table 23-88: 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 23-88: Parameters of the segment display controller

23.2. 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).

23.2.1. Periphal registers

Table 23-89: 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 23-89: Configuration register of the matrix display controller

23.2.2. Assembly of the register REG_DISPLAYSTATUS

Table 23-90: 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 23-90: Register REG_DISPLAYSTATUS

23.2.3. Assembly of REG_TEXT_CHARPOS and REG_TEXT_CURSORPOS

Table 23-91: 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 23-91: Registers REG_TEXT_CHARPOS and REG_TEXT_CURSORPOR

23.2.4. Interrupts

Table 23-92: 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 23-92: Interrupts of the matrix display controller

23.2.5. 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 23-93: 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 23-93: Implemented graphic functions

23.2.6. 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.

23.2.7. Module parameters

Table 23-94: 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 23-94: Parameters of the matrix display controller

24. 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.

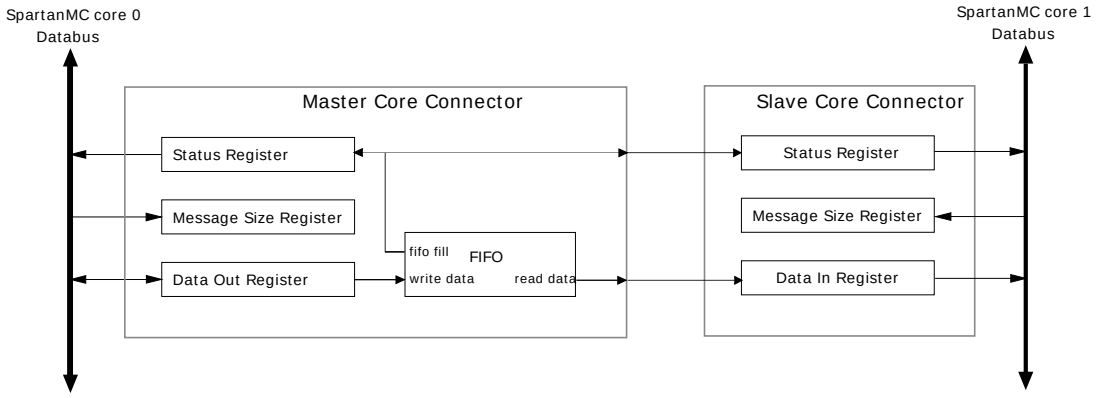


Figure 24-52: Unidirectional core connector

24.1. 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 24-95: Module parameters

24.2. Peripheral Registers

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

24.2.1. 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 or fewer entries than specified by MSG_SIZE are used.
2	FIFO is full or has fewer free entries than specified in the MSG_SIZE register.

Table 24-96: STATUS states

24.2.2. 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.

24.2.3. 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.

24.2.4. DATA_IN Register Description

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

24.3. Usage examples: MPSoC Lib

To communicate at a higher abstraction level the mpsoc library (see spartanmc/include/mpsoc.h) can be used. It offeres various functions for blocking and non-blocking send and receive for all inter core communication peripherals. To use the library add mpsoc to LIB_OBJ_FILES in config-build.mk in the firmware folder.

The _nb specifies non blocking funtions. Meaning that the function sends/receives the data if possible, if not it returns an error code (CORE_CONN_FULL / CORE_CONN_EMPTY). The functions also have a _value addition for sending one value or _data for sending more data, like arrays. The _data functions receive/send data in blocks of 16. In ideal case the buffer depth should be multiples of 16. The message to be send can also be larger than the buffer size. If the core connector is read from both sides at the same time via the _data functions the buffer depth shall ideally be 32.

24.3.1. Minimal send example

```
#include <system/peripherals.h>
#include <peripherals/duplex_core_connector.h>
#include <mpsoc.h>

master_core_connector_regs_t *master= DUPLEX_CORE_CONNECTOR_0;

int main(){
   int a[5]={1,2,3,4,5};
   core_connector_master_send_value(master,5);
   core_connector_master_send_data(master,&a, 5);
   while(1);
}
```

24.3.2. Minimal receive example

```
#include <system/peripherals.h>
#include <peripherals/duplex_core_connector.h>
#include <mpsoc.h>

slave_core_connector_regs_t *slave= DUPLEX_CORE_CONNECTOR_0 +
DUPLEX_CONNECTOR_SLAVE_OFFSET;

int main(){
   int size=core_connector_slave_receive_value(slave);
   int a[size];
   core_connector_slave_receive_data(slave,&a, size);
   while(1);
}
```

25. Concentrator system for multicore systems

The concentrator system implements a N to 1 connection between multiple SpartanMC cores. Several slaves can transmit data to the master core unidirectionaly. An internal round robin arbiter implements the selection of the slaves.

25.1. Module Parameters

25.1.1. Master

Parameter	Default Value	Descripton
BUFFER_SIZE	64	Size of the integrated FIFO buffer.
NUMBER_OF_SLAVES	3	Number of slaves connected to the master.
SOFTWARE_ARBITRATION	0	Defines if direct software arbitration is used instead of the round robin arbiter.

Table 25-97: Master module parameters

25.1.2. Slave

Parameter	Default Value	Descripton
SLAVE_ID	0	Unique ID of the slave.

Table 25-98: Slave module parameters

25.2. Peripheral Registers

25.2.1. Master

The master offers the following registers.

Name	Description
STATUS	Returns if the amount of data depicted by MSG_SIZE is available in the buffer.

Name	Description
MSG_SIZE	Can only be written to check the buffer (see STATUS register).
DATA_OUT	Register to read out the data values and header.
PEEK_DATA	Register to read out the header packet without affecting the fifo's content.
SOFT_ARBIT	Control register to define the slave id that is granted access in case of direct software arbitration.

Table 25-99: Registers

25.2.2. Register usage

First, set MSG_SIZE to "1" to check if there is any data in the buffer. This can be done by reading out the STATUS register. If it returns BUFFER_EMPTY there is no data to be read. Else, the header can be read from the DATA_OUT register. The 9 LSB of the header contain the source slave ID, the MSB of the header contain the message size. According to the message size, data can be read from the DATA_OUT register.

25.2.3. Slave

The slave offers the following.

Name	Description
STATUS	Returns if enough space is left in buffer for amount of data depicted by MSG_SIZE.
MSG_SIZE	Can only be written to check the buffer (see STATUS register).
DATA_IN	Register to write data values and header.
DATA_AVAILABLE	Register to mark the availability of data.

Table 25-100: Registers

25.2.4. Register usage

First, write the message size plus one (for the header) to the MSG_SIZE register and set DATA_AVAILABLE to "1". Then, read the STATUS register to check if there is enough space in the buffer. If the STATUS register contains NO_SEND_PERMISSION, this slave is not permitted to send data. If it contains BUFFER_FULL, the slave is permitted to send data, but there is not enough space in the buffer for the whole message. Use the copy_mem_to_reg_with_small_buffer function then. If it contains BUFFER_AVAILABLE, the whole message can be written to DATA_IN at once. Finally, set DATA_AVAILABLE to "0".

25.3. Usage examples

25.3.1. Register level access

Because of the complex register usage, it is highly recommended to use the C library which can be included by adding "#include <concentrator.h>" at the top of the program. The functions of the library use the high efficiency copy functions of "mpsoc.c".

25.3.2. Slave - sending a packet with the blocking function

Sending a package with 5 values

```
unsigned int data_array[5] = {1, 2, 3, 4, 5};
unsigned int number_of_values = 5;

concentrator_slave_send_data(SLAVE_CONCENTRATOR_0,
&data_array, number_of_values);
```

25.3.3. Slave - sending a packet with the non-blocking function

Sending a package with 5 values

```
unsigned int data_array[5] = { 1, 2, 3, 4, 5 };
unsigned int number_of_values = 5;

concentrator_slave_data_available_request_for_nb(SLAVE_CONCENTRATOR_0,
number_of_values);
while (concentrator_slave_send_data_nb(SLAVE_CONCENTRATOR_0,
&data_array, number_of_values) != CONCENTRATOR_SEND_OK);
```

25.3.4. Master - receiving a packet with the blocking function

```
unsigned int data_array[50];
unsigned int number_of_values = 0, slave_id = 0;

concentrator_slave_send_data(MASTER_CONCENTRATOR_0,&data_array,
&number_of_values, &slave_id);
```

25.3.5. Master - receiving a packet with the non-blocking function

```
unsigned int data_array[50];
unsigned int number_of_values = 0, slave_id = 0;
```

```
while
(concentrator_slave_send_data_nb(MASTER_CONCENTRATOR_0,&data_array,
&number_of_values, &slave_id) != CONCENTRATOR_RECEIVE_OK);
```

26. Dispatcher system for multicore systems

The dispatcher system implements a 1 to N connection between multiple SpartanMC cores. A master can transmit data to several slave cores unidirectionally. Packages can be sent to individual receivers or can be distributed via an internal round robin arbiter or a load balancing arbiter, which chooses the slave with the fewest number of waiting data entries in the buffer.

26.1. Module Parameters

26.1.1. Master

Parameter	Default Value	Descripton
BUFFER_SIZE	64	Size of the integrated FIFO buffer.
NUMBER_OF_SLAVES	3	Number of slaves connected to the master.

Table 26-101: Master module parameters

26.1.2. Slave

Parameter	Default Value	Descripton
SLAVE_ID	0	Unique ID of the slave.

Table 26-102: Slave module parameters

26.2. Peripheral Registers

26.2.1. Master

The master offers the following registers.

Name	Description
STATUS	Returns if enough space is left in buffer for amount of data depicted by MSG_SIZE.

Name	Description
MSG_SIZE	Can only be written to check the buffer (see STATUS register).
DATA_IN	Register to write data values.

Table 26-103: Master registers

26.2.2. Register usage

Write the amount of data to be written plus one (for the packet header) to the MSG_SIZE register. Accordingly, the STATUS register returns BUFFER_AVAILABLE or BUFFER_FULL. If the return value is BUFFER_AVAILABLE, the whole message can be written to the DATA_IN register at once. If the return value is BUFFER_FULL, the message can be written in pieces (e.g. using the copy_mem_to_reg_with_small_buffer function) or the send action can be aborted.

26.2.3. Slave

The slave offers the following registers.

Name	Description
STATUS	Returns if amount of data depicted by MSG_SIZE is available for this slave.
MSG_SIZE	Can only be written to check the buffer (see STATUS register).
DATA_OUT	Register to read out the data values.
MSG_SIZE_IN	Contains the amount of data of the current message.
RECEIVED	Can be written to confirm the complete reception of the message.

Table 26-104: Slave registers

26.2.4. Register usage

Set MSG_SIZE to "1" to check if any data is available for the slave by reading out the STATUS register. If DATA_AVAILABLE is returned, data is available. Then, the MSG_SIZE_IN register can be read to obtain the size of the message to be read. To check whether the full message is present in the buffer, this read out message size can be written to the MSG_SIZE register. If the STATUS register returns DATA_AVAILABLE, the whole message can be read from the DATA_OUT register at once. If the STATUS register returns BUFFER_EMPTY, the message has to be read out in pieces (e.g. using the copy_reg_to_mem_with_small_buffer function). Finally, the RECEIVED register has to be written to confirm the complete reception of the message.

26.3. Usage examples

26.3.1. Register level access

Because of the complex register usage, it is highly recommended to use the C library which can be included by adding "#include <dispatcher.h>" at the top of the program. The functions of the library use the high efficiency copy functions of "mpsoc.c".

26.3.2. Master - sending a packet with the blocking function

Sending a package with 5 values.

```
unsigned int data_array[5] = {1, 2, 3, 4, 5};
unsigned int number_of_values = 5;

//Receiver id = 2
dispatcher_master_send_data(MASTER_DISPATCHER_0, &data_array,
number_of_values, 2);

//Receiver id choosen by round-robin arbiter
dispatcher_master_send_data(MASTER_DISPATCHER_0, &data_array,
number_of_values, DISPATCH_ROUND_ROBIN);

//Receiver id choosen by load-balancing arbiter
dispatcher_master_send_data(MASTER_DISPATCHER_0, &data_array,
number_of_values, DISPATCH_LOAD_BALANCING);
```

26.3.3. Master - sending a packet with the non-blocking function

Sending a package with 5 values.

```
unsigned int data_array[5] = {1, 2, 3, 4, 5};
unsigned int number_of_values = 5;

//Receiver id = 2
while ( dispatcher_master_send_data_nb(MASTER_DISPATCHER_0,
&data_array, number_of_values, 2) != DISPATCHER_SEND_OK);

//Receiver id choosen by round-robin arbiter
while ( dispatcher_master_send_data_nb(MASTER_DISPATCHER_0,
&data_array, number_of_values, DISPATCH_ROUND_ROBIN) !=
DISPATCHER_SEND_OK);

//Receiver id choosen by load-balancing arbiter
```

```
while ( dispatcher_master_send_data_nb(MASTER_DISPATCHER_0,
&data_array, number_of_values, DISPATCH_LOAD_BALANCING) !=
DISPATCHER_SEND_OK);
```

26.3.4. Slave - receiving a packet with the blocking function

```
unsigned int data_array[50];
unsigned int number_of_values = 0;

dispatcher_slave_read_data(SLAVE_DISPATCHER_0, &data_array,
&number_of_values);
```

26.3.5. Slave - receiving a packet with the non-blocking function

```
unsigned int data_array[50];
unsigned int number_of_values = 0;

while ( dispatcher_slave_read_data_nb(SLAVE_DISPATCHER_0,
&data_array, &number_of_values) != DISPATCHER_RECEIVE_OK);
```

27. 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.

27.1. 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.

27.2. 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 27-105: 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
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

27.3. Task management

27.3.1. 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 27-106: 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 27-107: Info about create_task

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

27.3.2. 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 27-108: Parameters of delete_task

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

Table 27-109: Info about delete_task

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

27.3.3. suspend_task

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

Table 27-110: Parameters of suspend_task

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

Table 27-111: Info about suspend_task

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

27.3.4. resume_task

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

Table 27-112: Parameters of resume_task

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

Table 27-113: Info about resume_task

Property	Value
Callable by ISR	No
Internal call depth	3

Property	Value
with active interrupts	2

27.3.5. get_current_task

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

Table 27-114: Info about get_current_task

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

27.3.6. 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 27-115: Info about forbid_preemption

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

27.3.7. 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 27-116: Info about permit_preemption

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

27.3.8. task_yield

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

Table 27-117: Info about task_yield

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

27.4. Semaphores

27.4.1. 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 27-118: Parameters of initialize_semaphore

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

Table 27-119: Info about initialize_semaphore

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

27.4.2. 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 27-120: Parameters of semaphore_down

Parameter	Description
sem	The semaphore

Table 27-121: Info about semaphore_down

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

27.4.3. 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 27-122: Parameters of semaphore_up

Parameter	Description
sem	The semaphore

Table 27-123: Info about semaphore_up

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

27.5. Dynamic memory allocation

27.5.1. malloc

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

Table 27-124: Parameters of malloc

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

Table 27-125: Info about malloc

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

27.5.2. free

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

Table 27-126: Parameters of free

Parameter	Description
ptr	Pointer to the memory block to free.

Table 27-127: Info about free

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

27.6. 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();
   }
}
FILE * stdout = &UART_LIGHT_0_FILE;

void main( void ) {

   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
```

28. Simple technology agnostic clock generator

The Simple technology agnostic clock generator provides a configurable output clock that is generated from an input clock. The input clock can be multiplied and divided to meet the user desired output frequency. It can be used for xilinx and altera projects.

The output frequency is calculated by the formula:

output frequency = (input frequency * MULTIPLY) / DIVIDE

28.1. Module Parameters

Table 28-128: Simple technology agnostic clock generator module parameters

Parameter	Descripton
CLKIN_PERIOD	The duration of one period of the input frequency in nanoseconds.
MULTIPLY	The multiplier for the input frequency. Supported values range from 2 to 32.
DIVIDE	The divider for the input frequency. Supported values range from 1 to 32.

29. Altera Cyclone 4 PLL

The Altera Cyclone 4 PLL clock generator provides up to five configurable output clocks which are generated from an input clock. The input clock can be multiplied and divided to meet the user desired output frequency. The output clocks can also be shifted.

This module aims to provide all important options of the altpll megafunction to the user. Not all combinations of MULTIPLY_BY, DIVIDE_BY and PHASE_SHIFT might be possible. If the synthesis fails with PLL related errors use the Quartus IDE to check if the combination of selected parameters is supported. To do this, create a Quartus project with device EP4CE22F17C6 and select "PLL" from the ip catalog. Configure the pll parameters in MegaWizard according to the settings in jconfig (all output clocks in MegaWizard tab 3 enabled, all parameters in MegaWizard tabs 1 and 2 default) and check if Megawizard says "Able to implement the requested PLL". If not, change the parameters in MegaWizard until the PLL is implementable and apply those settings to jconfig.

The output frequency for each output clock is calculated by the formula:

output frequency = (input frequency * MULTIPLY) / DIVIDE

29.1. Module Parameters

Table 29-129: Cyclone 4 PLL module parameters

Parameter	Descripton
INCLK0_INPUT_FREQUENCY	The frequency on the INCLK0 input in picoseconds.
CLKx_DIVIDE_BY	The divider for the input frequency of output clock x.
CLKx_MULTIPLY_BY	The multiplier for the input frequency of output clock x.
CLKx_DUTY_CYCLE	The duty cycle for output clock x.
CLKx_PHASE_SHIFT	The phase shift for output clock x.

30. Lattice VersaECP5 DevKit PLL

The Lattice VersaECP5 DevKit PLL provides a configurable output clock that is generated from an input clock. The input clock can be multiplied and divided to meet the user desired output frequency.

Because the parameters of the EHXPLLL primitive used on ECP5 devices are not explained in the lattice documentation and their calculation is dark magic hidden in the Lattice Diamond Clarity Designer GUI this module uses several hardcoded PLL implementations as an ugly workaround. The module expects a 100 MHz clock as input which is available on the VersaECP5 board at FPGA pin P3. The implementation to use for the project is chosen by the MULTIPLY parameter.

For most projects this should be sufficient and the use of Clarity Designer can be avoided. If more advanced PLL options are needed, please use the Module ECP5_PLL and get the necessary parameters from Clarity Designer.

The output frequency is calculated by the formula:

output frequency = (input frequency * MULTIPLY) / DIVIDE

30.1. Module Parameters

Table 30-130: Lattice VersaECP5 DevKit PLL module parameters

Parameter	Descripton
MULTIPLY	The multiplier for the input frequency. Supported values range from 4 to 100.
DIVIDE	The divider for the input frequency. Hardcoded to 100.

31. Lattice ECP5 PLL

The Lattice ECP5 PLL clock generator provides up to three configurable output clocks which are generated from an input clock. The input clock can be multiplied and divided to meet the user desired output frequency. The output clocks can also be shifted.

This module aims to provide all important options of the EHXPLLL primitive to the user. Because the parameters of the EHXPLLL primitive used on ECP5 devices are not explained in the lattice documentation and their calculation is dark magic hidden in the Lattice Diamond Clarity Designer GUI it is required to get the correct parameters from Clarity Designer.

To do this, create a Lattice Diamond project with device LFE5UM-45F-8BG381C (FPGA used on the VersaECP5 DevKit) and open Clarity Designer. Select "pll" from the ip catalog. Select "Int_OP" for "Feedback Mode" and 10% tolerance for CLKOP. Enter the desired phase shift and output frequency for CLKOS, CLKOS2 and CLKOS3 and click "calculate". Enter the parameters in jconfig.

For no phase shift, use CLKOS/2/3_DIV - 1 for CLKOS/2/3_CPHASE and 0 for CLKOS/2/3_FPHASE. If you need phase shift, generate the configured pll and open the generated verilog file with the EHXPLLL instance. Enter the CPHASE and FPHASE parameters in jconfig as set in the verilog file.

31.1. Module Parameters

Table 31-131: Lattice ECP5 PLL module parameters

Parameter	Descripton
CLKI_DIV	The Refclk Divider in the Diamond Clarity Designer UI.
CLKFB_DIV	The Feedback Divider in the Diamond Clarity Designer UI.
CLKOP_DIV	The CLKOP Output Divider in the Diamond Clarity Designer UI.
CLKOS_DIV	CLKOS Output Divider in the Diamond Clarity Designer UI.
CLKOS_CPHASE	CLKOS_CPHASE in the verilog file generated by Diamond Clarity Designer. Choose CLKOS_DIV - 1 for no phase shift.
CLKOS_FPHASE	CLKOS_FPHASE in the verilog file generated by Diamond Clarity Designer. Choose 0 for no phase shift.
CLKOS2_DIV	CLKOS2 Output Divider in the Diamond Clarity Designer UI.
CLKOS2_CPHASE	CLKOS2_CPHASE in the verilog file generated by Diamond Clarity Designer. Choose CLKOS2_DIV - 1 for no phase shift.
CLKOS2_FPHASE	CLKOS2_FPHASE in the verilog file generated by Diamond Clarity Designer. Choose 0 for no phase shift.
CLKOS3_DIV	CLKOS3 Output Divider in the Diamond Clarity Designer UI.
CLKOS3_CPHASE	CLKOS3_CPHASE in the verilog file generated by Diamond Clarity Designer. Choose CLKOS3_DIV - 1 for no phase shift.

Parameter	Descripton
CLKOS3_FPHASE	CLKOS3_FPHASE in the verilog file generated by Diamond Clarity Designer. Choose 0 for no phase shift.

32. ChipScope

The ChipScope Pro system from Xilinx is a tool which provides capabilities for on-chip debugging. ChipScope is being integrated into the target design and thereby provides access to internal signals of the design. It features functionality of a logic analyzer, such as advanced trigger configurations for detection of relevant events and means for displaying the recorded data. The ChipScope system can be added and configured directly through the SpartanMC toolchain.

32.1. System Setup

The ChipScope system is composed of individual modules. This allows for extensive customization for the actual needs. The system is setup by adding, configuring and connecting the ChipScope modules, which are the Integrated Controller (ICON) and the Integrated Logic Analyzer (ILA). Every ILA module must be connected to a dedicated control port of the ICON module. These control ports (*CONTROL0...15*) are the only ports of the ICON module. Each ILA core features one control port (*CONTROL*) for connection with the ICON. Besides, the ILA also has multiple signal inputs for signals to measure. These are called trigger ports (*TRIG0...15*). Additionally, there are ports for a clock input (*CLK*), an optional data input (*DATA*) and an optional trigger output (*TRIG_OUT*).

Note: There must be only one ICON Core in the design. While the SpartanMC toolchain allows creating configurations with more than one ICON Cores, the ChipScope Analyzer software does not support the use of multiple ICONs.

32.2. Module Parameters

32.2.1. Integrated Controller (ICON)

Table 32-132: ICON module parameters

Parameter	Description
NUMBER_CONTROL_PORTS	Number of ILA Cores to be connected.
BOUNDARY_SCAN_CHAIN	BSCAN USER scan chain number to be used.

32.2.2. Integrated Logic Analyzer (ILA)

Table 32-133: ILA module parameters

Parameter	Description
SAMPLE_ON	Defines whether data shall be captured on rising or falling edge of the incoming clock signal.
SAMPLE_DATA_DEPTH	Depth of the data buffer.
ENABLE_STORAGE_QUALIFICATION	Enables the use of optional storage qualifiers via the ChipScope Analyzer.
DATA_SAME_AS_TRIGGER	Defines whether the trigger signals shall be captured or if the data to be captured is supplied through an additional data input port.
DATA_PORT_WIDTH	Defines the width of the optional data input port.
NUMBER_OF_TRIGGER_PORTS	Sets the number of available trigger input ports.
MAX_SEQUENCE_LEVELS	The maximum length for sequencing trigger events in the ChipScope Analyzer. A value of 1 means no sequencing.
ENABLE_TRIGGER_OUTPUT_PORT	Enables an additional 1 bit output port, which is activated whenever the active trigger event is detected.
USE_RPMS	Enables the use of relative-placed macro constraints for performance optimized logic placement of the core. It is recommended to leave option enabled. Disable, if constraints can't be fulfilled (may happen for resource intense designs).
TRIGGER_PORT_WIDTH_0...15	Width of the individual trigger ports.
MATCH_UNITS_AMOUNT_0...15	The amount of match units at the corresponding trigger port. The total amount of match units per ILA core may not exceed 16.
COUNTER_WIDTH_0...15	Adds an optional counter to the corresponding trigger port and sets the counters width.
MATCH_TYPE_0...15	Specifies the type of the match units for the corresponding trigger port
EXCLUDE_FROM_DATA_STORAGE_0...15	Excludes the signal at the corresponding trigger port from being stored in case of a trigger event.

Table 32-134: Types of match units

Type	Bit Values	Match Function
Basic	0, 1, X	'=', '<>'
Basic with edges	0, 1, X, R, F, B, N	'=', '<>'
Extended	0, 1, X	'=', '<>', '<', '>=', '<', '<='
Extended with edges	0, 1, X, R, F, B, N	'=', '<>', '<', '>=', '<', '<='

Type	Bit Values	Match Function
Range	0, 1, X	'=', '<>', '<', '>=', '<', '<=', 'in range', 'not in range'
Range with edges	0, 1, X, R, F, B, N	'=', '<>', '<', '>=', '<', '<=', 'in range', 'not in range'

32.3. Usage

All the debugging, e.g. defining trigger events for data capture and evaluating the recorded signals, is done in the Xilinx ChipScope Analyzer. After adding the ChipScope system to a design, completing the build process and loading the design onto the hardware, the Analyzer can be started by running the following command from the project directory:

make chipscope

To connect the Analyzer with the target design, the used connection method has to be chosen in the *JTAG Chain* menu in the Analyzers user interface. After selecting the adequate option, the Analyzer will do a JTAG scan to detect ChipScope on the hardware. After confirming the following dialogs the automatically created project will be loaded and the system is ready to use.

32.3.1. Bus / Pin Names

When debugging modules using Chipscope, one often ends up adding a new module output port with a number of internal signals, so that they are accessible to Chipscope. To display the correct names of the internal signals, one can add a file called *internal_ports.v* in the module's directory, containing the part of the Verilog file where the internal signals are assigned to the output port.

For example, the following file would assign the name *a* to bit 0 of port *debug* , and the bus *b* to bits 1 and 2:

```
assign debug[0] = a;
assign debug[2:1] = b;
```

33. AXI-Bus-Master

33.1. Overview

The Advanced eXtensible Interface Standard (AXI) is a wide-spread burst based protocol for chip-internal communication in SoC. This DMA module allows to exchange data with AXI slaves. It does not provide any peripheral registers, so all control, status and payload data is transmitted via the DMA memory.

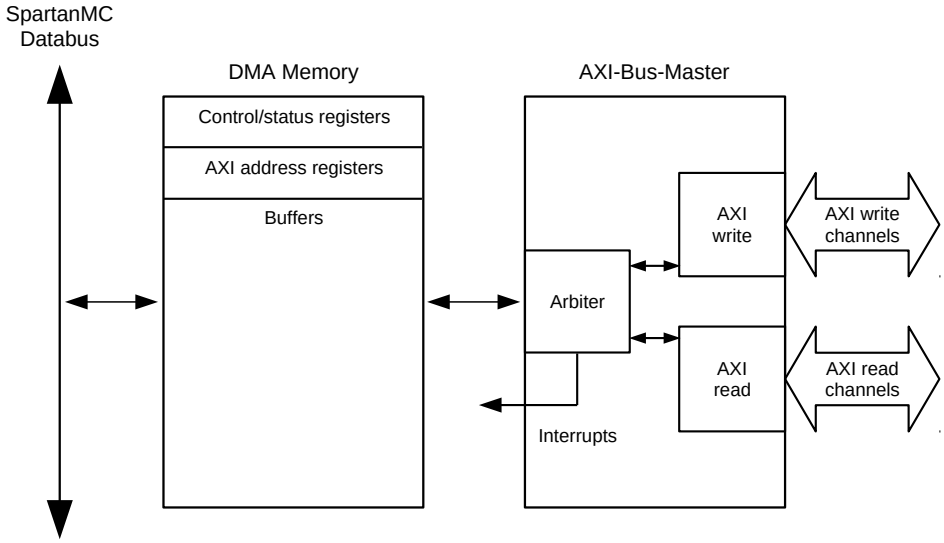


Figure 33-53: AXI-Bus-Master block diagram

33.2. Module parameters

Table 33-135: AXI module parameters

Parameter	Default Value	Description
DMA_BASE_ADR	0x00	Base address of the module's DMA memory. **This parameter is set by jConfig automatically.**
AXI_BUS_WIDTH	16	Width of the AXI read/write data signals. Values of 8, 16 and 32 bit are supported.
DOUBLE_BUFFERING	0	If double buffering is activated, two read and write buffers can be used. Each buffer has its own control and AXI address registers.

Parameter	Default Value	Description
USE_INFER_BRAM	1	Use inferred BRAM instead of macro instantiantion to provide a higher code portability. **When using the inferred BRAM, 8 bit AXI bus width is not supported.**

33.3. DMA Memory Organization

By default, the 16 bit wide DMA memory contains one buffer for read and one buffer for write transactions. Each buffer has its own control/status and AXI address registers. When double buffering is activated, both buffers are divided into two seperate buffers and additional control/status and AXI address registers are enabled. This allows to make better use of the SpartanMC peripheral bus when transmitting big amounts of data. Consider the different maximum burst lengths for single/double buffering.

Offset	Content	Description
0x000	w_ctrl_0	Control and status register of write buffer 0.
0x001	w_ctrl_1	Control and status register of write buffer 1. Only used if double buffering is activated.
0x002	w_ctrl_0	Control and status register of read buffer 0.
0x003	w_ctrl_1	Control and status register of read buffer 1. Only used if double buffering is activated.
0x004	w_addr_0_0	Lower 16 bit of AXI address of write buffer 0.
0x005	w_addr_0_1	Upper 16 bit of AXI address of write buffer 0.
0x006	r_addr_0_0	Lower 16 bit of AXI address of read buffer 0.
0x007	r_addr_0_1	Upper 16 bit of AXI address of read buffer 0.
0x008	w_addr_1_0	Lower 16 bit of AXI address of write buffer 1. Only used if double buffering is activated.
0x009	w_addr_1_1	Upper 16 bit of AXI address of write buffer 1. Only used if double buffering is activated.
0x00A	r_addr_1_0	Lower 16 bit of AXI address of read buffer 1. Only used if double buffering is activated.
0x00B	r_addr_1_1	Upper 16 bit of AXI address of read buffer 1. Only used if double buffering is activated.
0x00C	w_buffer_0	Write buffer 0 with a size of 506 16 bit words. If double buffering is activated, only 252 words can be used.
0x10A	w_buffer_1	Write buffer 1 with a size of 252 16 bit words. Only used if double buffering is activated.
0x206	r_buffer_0	Read buffer 0 with a size of 506 16 bit words. If double buffering is activated, only 252 words can be used.
0x304	r_buffer_1	Read buffer 1 with a size of 252 16 bit words. Only used if double buffering is activated.

Table 33-136: Position of registers and buffers in the DMA memory

Buffering type	8 bit AXI width	16 bit AXI width	32 bit AXI width
single	256	256	253
double	256	252	126

Table 33-137: Maximum burst lengths at different AXI bus widths

33.4. Control Register Organization

Bit	Name	Description
7-0	Burst length	Number of transfers of the AXI transaction.
9-8	Burst type	Burst type of the AXI transaction.
10	valid	Valid bit to initiate the AXI transaction.
12-11	AXI response	The slaves AXI response. Set simultaneously with the done bit.
13	done	Done bit set by the AXI slave when the transaction is finished.
15-14	not used	-

Table 33-138: Control register layout

33.5. Usage

An AXI transaction can be initiated either by setting the AXI address and control registers of the buffer to be used or by using the C-functions that automate the register access for incremental AXI transactions. When transferring big amounts of data, direct register access could reduce control overhead significantly. Besides the "axi_write_sync"/"axi_read_sync" functions, that contain a polling routine, the "axi_write"/"axi_read" functions can be used in combination with the "axi_done" polling function especially to make use of the benefits of double buffering. The module also provides an interrupt signal for each buffer. To reset an interrupt, the done bit in the control/status register needs to be overridden.

Bit	Description
0	Transaction on write buffer 0 finished.
1	Transaction on write buffer 1 finished.
2	Transaction on read buffer 0 finished.
3	Transaction on read buffer 1 finished.

Table 33-139: Interrupt signal structure

33.6. AXI-Bus-Master C-Header for DMA Memory Description

```
#ifndef __AXI_HW_H
#define __AXI_HW_H

#ifdef __cplusplus
extern "C" {
#endif

//Control-register masks
#define AXI_CTRL_BLEN      0x00FF
#define AXI_CTRL_TYPE      0x0300
#define AXI_CTRL_VALID      0x0400
#define AXI_CTRL_RESP      0x1800
#define AXI_CTRL_DONE      0x2000

//Burst-types
#define AXI_CTRL_TYPE_FIXED   0x0000
#define AXI_CTRL_TYPE_INCR   0x0100
#define AXI_CTRL_TYPE_WRAP   0x0200

typedef struct axi {
   //0x00
   volatile unsigned int   w_ctrl_0;
   //0x01
   volatile unsigned int   w_ctrl_1;
   //0x02
   volatile unsigned int r_ctrl_0;
   //0x03
   volatile unsigned int   r_ctrl_1;

   //0x04
   volatile unsigned int      w_addr_0_0;
   //0x05
   volatile unsigned int      w_addr_0_1;
   //0x06
   volatile unsigned int       r_addr_0_0;
   //0x07
   volatile unsigned int       r_addr_0_1;

   //0x08
   volatile unsigned int      w_addr_1_0;
   //0x09
   volatile unsigned int      w_addr_1_1;
   //0x0A
```

```
   volatile unsigned int       r_addr_1_0;
   //0x0B
   volatile unsigned int       r_addr_1_1;

   //0x0C
   volatile unsigned int              w_buffer_0[254];
   //0x10a
   volatile unsigned int              w_buffer_1[252];
   //0x206
   volatile unsigned int   r_buffer_0[254];
   //0x304
   volatile unsigned int   r_buffer_1[252];
} axi_dma_t;

#ifdef __cplusplus
}
#endif

#endif
```

34. Global Firmware Memory

34.1. Overview

The global firmware memory gives multiple SpartanMC cores access to the same Block-RAM-based memory. It stores one firmware, which is executed by every connected core and has to be added as a common module. The global memory module, generates one code memory and for each attached core additionally one data memory.

34.2. Module parameters

Table 34-140: Global firmware memory module parameters

Parameter	Default Value	Description
CORES_COUNT	2	Number of cores to connect
RAMBLOCKS_GLOBAL	4	Ramblocks used for global memory
USE_TWO_PORTS_GLOBAL	0	Enable the use of both ports of the internal block RAMs. Can be faster when the memory is heavily used, but needs more resources.
LOCAL_BASE_ADDR	8192	Base address of the local data memories.
RAMBLOCKS_LOCAL	4	Ramblocks used for local memory
SHOW_MEM_ALL	0	Print all memory accesses during simulation
SHOW_MEM_SINGLE	1	Have one memory print accesses during simulation
CACHE_SIZE	1024	The amount of cache Blocks to be used
OFFSET_LENGTH	1	Length of the caches block offset in bits
CACHE_WAYS	1	N-way set associative cache
INDEX_LENGTH	9	Number of index bits for the cahce.

34.3. Restrictions for connected subsystems

All connected subsystems must use the same firmware. It is not possible to use individual firmware parts, since the instruction fetch from the local memory is entirely disabled.

The subsystems must also have the same memory layout and the same peripherals. Otherwise they would need different code.

35. Router for multicore systems

The router implements a FIFO through which multiple SpartanMC cores are able to communicate. Routers can send, receive and pass messages. Static routing is used. Also the route has to be established from sender to receiver until the first data is transmitted.

35.1. Requirements

Every subsystem should contain only one routermodule. The routerids should be integers starting with 0. The Buffers should contain at least 4 values. The submodule "sender" is only synthesized if the router has at least one output. The submodule "receiver" is only synthesized if the router has at least one input. So the input "0" from the submodule "selector" is connected to the submodule "sender" or the first input. So the output "0" from the submodule "splitter" is connected to the submodule "receiver" or the first output. The TO_DEST_x parameters in jconfig describe to which port the splitter is routing (x is a destination routerid). If the router has a receiver "0" means route to own receiver and "1" to first output port. If the router has no receiver "0" means route to first output port. The maximum allowed messagesize is "buffersize-1". If the firmware at the receiver is not reading the buffer and it filled at a level that there is not enough space for new messages, the messages will not be accepted (returnout=01). There are three signals between two connected routers: data, request and return. If router A wants to send to router B then data (18 bits) and request (1 bit) is connected from A to B , but return (2 bits) is connected from B to A. Every SpartanMC-Core and routermodule has to be driven by the same clock.

Output (Value of TO_DEST_x)	meaning if router has a receiver	meaning if router has no receiver
0	to own receiver	to first output port
1	to first output port	to second output port
2	to second output port	to third output port
3	to third receiver	to fourth output port

Table 35-141: Outputs of splitter (TO_DEST_x)

It is required, that the first 18 bits of datain, the first 1 bit of requestin and the first 2 bits of returnout are connected to the same router and so on.

Routernumber	datain	requestin	returnout
first input	[17:0]	[0]	[1:0]

Routernumber	datain	requestin	returnout
second input	[35:18]	[1]	[3:2]
third input	[53:36]	[2]	[5:4]
fourth input	[71:54]	[3]	[7:6]

Table 35-142: Input bits

It is required, that the first 18 bits of dataout, the first 1 bit of requestout and the first 2 bits of returnin are connected to the same router and so on.

Routernumber	datain	requestin	returnout
first output	[17:0]	[0]	[1:0]
second output	[35:18]	[1]	[3:2]
third output	[53:36]	[2]	[5:4]
fourth output	[71:54]	[3]	[7:6]

Table 35-143: Output bits

35.2. Module Parameters

Parameter	Default Value	Descripton
ROUTER_ID	0	The id of the rounter. Should be x if the router is part of subsystem_x. (0,1,2,..)
DEPTH_OUTPUT_BUFFER	32	Amount of 18bit Values stored in the sender fifo. Should be greater or equal 4.
DEPTH_INPUT_BUFFER	32	Amount of 18bit Values stored in the receiver fifo. Should be greater or equal 4.
AMOUNT_OF_INPUTS	0	Amount of inputs (from other routers)
AMOUNT_OF_OUTPUTS	0	Amount of outputs (to other routers)
TO_DEST_0	0	Which output should the splitter use for messages to router 0.
TO_DEST_1	0	Which output should the splitter use for messages to router 1.
TO_DEST_2	0	Which output should the splitter use for messages to router 2.

Table 35-144: Module parameters

35.3. Java routing tool

There is a java tool reading your jconfig.xml, using Dijkstra's algorithm, creating a .dot file for graph creation using graphviz and telling you what to insert in jconfig in the TO_DEST_x fields.

To use this tool some naming conventions have to be satisfied. Every output a router has has to be connected to a net (Add gluelogic -> internal -> net) in the same subsystem. This net needs to have a specific name. Every dataout and requestout net has to be named net_dataout_x and net_requestout_x, where x is the number of the splitter output the net is connected to (0,1,2,... if the router has no receiver, 1,2,3,... if the router has a receiver). Every returnout net has to be named net_returnout_y, where y is the number of the selector output the net is connected to(0,1,2,... if the router has no sender, 1,2,3,... if the router has a sender).

To use the tool go to the project rootfolder. There the jconfig file should be named "jconfig.xml". Enter "make routing" in the terminal. There will be 4 files created.

File	Content
routingtable_debugoutput.txt	Contains what the tool has detected and errors (if any)
routingtable_FromViaTo.txt	Contains what connections the tool has detected. Notation: from_routerid:splitter_output_port:to_routerid
routingtable_graph.dot	Use "neato -Tsvg routingtable_graph.dot -o routingtable_graph.svg -Gstart=rand" to create a svg graph.
routingtable_text.txt	The output of Dijkstra's algorithm. This is what you should enter in jconfig. All fields not listed in this file should be "0". "0" has several meanings: not reachable, to own receiver(if the router has one), or to first output port(if therouter has no receiver). In fact packages for not reachable destinations will be forwarded. You are responsible that this will never happen.

Table 35-145: output from 'make routing'

35.4. Developer information

see $SPARTANMC_ROOT/src/doc/users-manual/src/router/*

Return wire	Meaning
00	Default. Receiver, or a transport router on the path has not accepted or denied yet.
01	Denied. Either a transport router, or the receiving router is busy. Or maybe the buffer od the receiving router has not enough free entries.
10	Accepted. Every transport router and the receiving router can handle the request. A dedicated line is established from submodule sender to submodule receiver. Receiver buffer has enough free entries. Ready for transfer. This state will be keept until the transmission is finished.
11	Finish. This signal is sended one clock period after sending is completed (state=10, request switches from 1 to 0). After sending one clock cycle, signal will switch to 00 again.

Table 35-146: meanings of return bits

35.5. Peripheral Registers

The router modules have three registers each for message transfer.

35.5.1. Router C-Header for Register description

```
#ifndef __router_H
#define __router_H

#ifdef __cplusplus
extern "C" {
#endif

typedef struct {
   volatile unsigned int data;
   volatile unsigned int free_entries;
   volatile unsigned int data_available;
} router_regs_t;

//router->data (router_start_addr + 0)
//router->free_entries (router_start_addr + 1)
//router->data_available (router_start_addr + 2)

#ifdef __cplusplus
}
#endif

#endif
```

35.5.2. data Register Description

The data register can be written for sending, or read for receiving messages.

35.5.3. free_entries Register Description

The free_entries register is only used for sending messages. It contains how many buffer entries are free. A new message should only be transfered to the sender's buffer via (data register) if there are enough free buffer entries to store the whole message including the header. (router->free_entries >= messagesize+1)

35.5.4. data_available Register Description

The data_available register is only used for receiving messages. it contains how many buffer entries are used. If it is greater than zero, a new header package is stored in the buffer (because new messages always start with a header). It can be read an now router_read.c can extract the messagesize from the header. The message is only transmitted, if the whole message is in the receiver's buffer. (router->data_available >=1) (router->data_available >= messagesize)

35.6. Usage examples

There are the wrappers router_check_data_available.c, router_read.c and router_send_data.c. Only the use of this wrappers is documented here.

35.6.1. router_check_data_available

```
#include <system/peripherals.h>
#include <router.h>

void main() {
   unsigned int dataavailable;
   dataavailable = router_check_data_available(ROUTER_0);
   while(1);
}
```

35.6.2. router_read

```
#include <system/peripherals.h>
#include <router.h>

void main() {
   unsigned int data[10] = {3,3,3,3,3,3,3,3,3,3};
   unsigned int msgsize, source;
   router_read(ROUTER_0, &data, &msgsize, &source);
   while(1);
}
```

35.6.3. router_send_data

```
#include <system/peripherals.h>
#include <router.h>

void main() {
   //router_send_data (router, data, source, msgsize, dest)
   unsigned int data[10] = {193,1,2,3,4,5,6,7,8,9};
   router_send_data (ROUTER_0, &data, 10, 10, 5);
   while(1);
}
```

36. DVI output

The DVI peripheral allows outputting video data to an attached Chrontel CH7301C (as is present on the SP605 eval board). Because the memory needed to store the images is very large, Block RAMs would be too small. Therefore, Data is sourced from an external DRAM, using the ddr_mcb_sp6 module.

36.1. Module Parameters

Most parameters are concerned with the size of the visible area and the blanking intervals. The values needed are dependent on the connected monitor. The monitor's supported resolutions and blanking intervals can be read via a separate I2C connection. An overview of the parameters can be seen in the following figure. Note that the order of back porch and front porch may seem to be swapped (e.g. *front* porch comes *after* visible area). This is not the case, because they are named in relation to the Sync Interval.

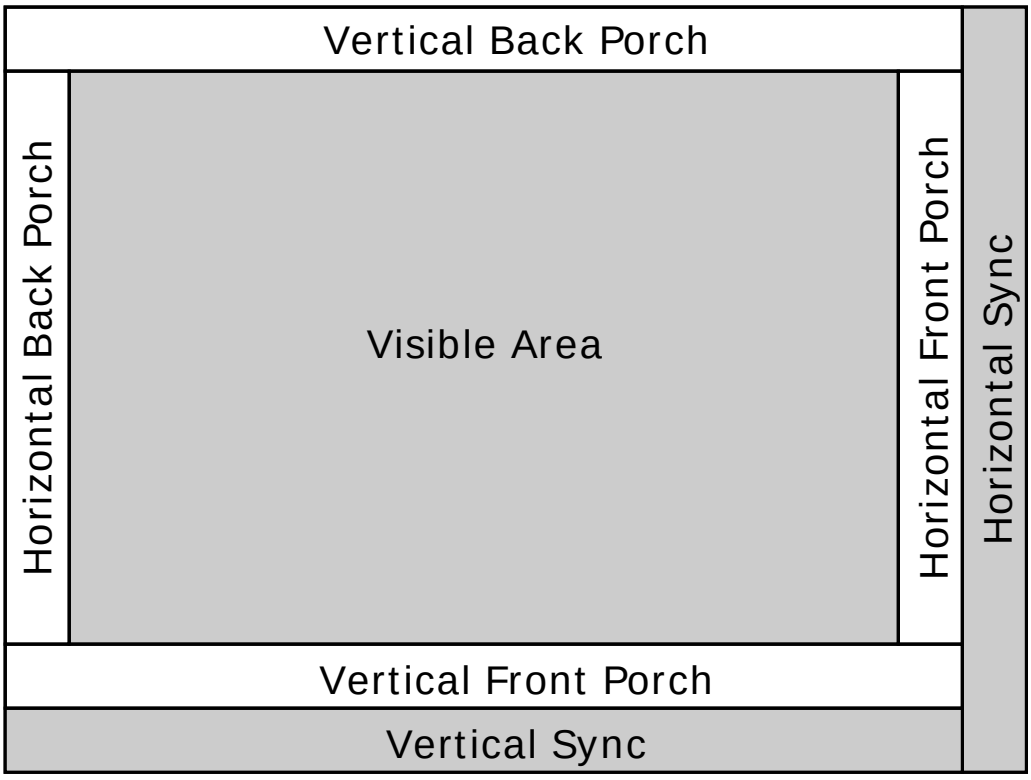


Figure 36-54: Sync intervals

The total number of pixels/lines output in horizontal/vertical directions is the sum of back porch, visible area, front porch and sync. The total number of pixels per frame is the product of the total number of horizontal pixels by the total number of vertical lines. The number of frames per second is the the frequency of the pixel clock divided by the total number of pixels per frame.

Normally, both the horizontal sync and vertical sync signals are active high. However, some monitors support reduced blanking time modes, where one or both of these lines may need to be inverted.

Parameter	Default Value	Descripton
H_VISIBLE	1280	Horizontal visible pixels
H_FRONT_PORCH	48	Duration of horizontal front porch
H_SYNC	112	Duration of horizontal sync
H_BACK_PORCH	248	Duration of horizontal back porch
V_VISIBLE	1024	Vertial visible pixels
V_FRONT_PORCH	1	Duration of vertical front porch
V_SYNC	3	Duration of vertical sync
V_BACK_PORCH	38	Duration of vertical back porch
H_SYNC_INVERT	0	Invert horizontal sync flag
V_SYNC_INVERT	0	Invert vertical sync flag
COLOR_MODE	RGB_24	The color mode to use. See memory layout for details

Table 36-147: Module Parameters

36.2. Peripheral Registers

36.2.1. Enable Register Description

The enable signal can enable or disable the output.

Table 36-148: Enable register

Offset	Name	Access	Description
0	Enable	read/ write	Enable video output

36.3. Memory Layout

Only a single frame can be read from memory, which always has to start at address zero. Therefore, it is not possible to use double buffering. The pixels are arranged row by row from top to bottom. Within the row, they are ordered from left to right. Only the visible area is stored.

Note: The select memory layout has to be set in the Chrontel chip via the dedicated I2C connection as well.

Depending on COLOR_MODE, pixels either occupy two or four bytes:

36.3.1. RGB Color Mode

Every pixel consists of 32 Bits, of which 8 are unused. The remaining 24 Bits provide 8 Bit per color channel.

Table 36-149: Pixel Data in RGB mode

Offset	Name
7-0	Blue
15-8	Green
23-16	Red
31-24	Unused

This corresponds to the chip's input data format 0.

36.3.2. YCRCB Color Mode

Every pixel consists of 16 Bits. Every pixel contains luminance information (Y), while color is the same for two pixels. Even pixels contain the Blue color (Cb), while odd ones contain the red color (Cr).

Table 36-150: Pixel Data in YCrCb mode

Offset	Name
7-0	Luminance (Y)
15-8	Color (Cb for even pixels, Cr for odd)

This corresponds to the chip's input data format 4.

37. Ethernet

The ethernet controller is split into three modules: MDIO, Ethernet TX and Ethernet RX. Different modules may be connected to different SpartanMC cores.

Currently, only MII is supported. GMII is downward compatible to MII. If the MDIO registers are set up so that only 100 MBit connections are negotiated, this controller can be used. RGMII and RMII have to be adapted to GMII/MII externally.

37.1. MDIO

MDIO is the Ethernet standard's Management interface. It is used to set configuration registers. The bus can address up to 32 Ethernet Phys. There are only 32 registers addressable per Phy. Current Phys provide many more than that. To access those, indirect addressing schemes have to be utilized.

MDIO consists of a clock line and a bidirectional data line. On some FPGA architectures, one bidirectional signal cannot be fed into multiple FPGA pins. Since different phys may be connected to different pins, a WIDTH parameter is provided to generate multiple connections.

37.1.1. Module parameters

Table 37-151: MDIO module parameters

Parameter	Default Value	Description
Width	1	Number of MDIO data lines

37.1.2. Module Registers

Table 37-152: MDIO registers

Offset	Name	Access	Description
0	MDIO Data	read/ write	Contains the Data
1	MDIO Address	read/ write	Contains both Phy and Register address and the status.

Writing to the MDIO data register starts a transmission. This causes the ready flag in the MDIO address register to go low until the transmission is finished.

37.1.3. MDIO Data Register

Writing to this register starts a transmission.

Table 37-153: MDIO data register layout

Bit	Name	Access	Default	Description
0-15	DATA	read/ write	0	Data that was read in the previous transaction / data to write. In case a read transaction is desired (write enable is low), data written to this part of the register is irrelevant.
16	unused	read	0	
17	Write Enable	write	0	If set to 1 during a write to the register, the transmission that is started is a write transmission. Otherwise it is a read transmission.

Table 37-153: MDIO data register layout

37.1.4. MDIO Address Register

Table 37-154: MDIO address register layout

Bit	Name	Access	Default	Description
0-4	Register Address	read/ write	0	The Phy's register that the next transmission should target
5-9	Phy Address	read/ write	0	The Phy that the next transmission should target
10-16	unused	read	0	
17	Ready	read	0	Indicates that no transmission is currently running.

Table 37-154: MDIO address register layout

37.2. Ethernet TX

This module handles the TX part of MII. It does not have any parameters.

37.2.1. DMA memory

Memory is organized as a ring buffer. Every entry starts with the length in bytes, stored as an 18 Bit int, followed by an Ethernet frame. Note that the frame should not contain the frame checksum (FCS), as it is appended by hardware.

As soon as the memory location pointed at by the DMA data offset register is not zero (either caused by the memory address being written to, or the register being changed), the frame is transmitted. Therefore, the length may only be written after the frame contents have been written.

After transmitting a packet, the hardware updates the offset register to point to the next address after the frame. To avoid sending an unintended packet, this memory location should be initialized to zero before writing the original packet's length.

37.2.2. Module Registers

Table 37-155: Ethernet TX registers

Offset	Name	Access	Description
0	Status/Control	read/ write	Status flags
1	DMA data offset	read/ write	Contains the current data pointer into the DMA memory
2	Interrupt	read/ write	Interrupt signal
3	Packet count	read	Number of packets sent since reset

37.2.3. Status/Control Register

Table 37-156: Ethernet TX status/control register layout

Bit	Name	Access	Default	Description
0	IrqEn	read/ write	0	Interrupt enable
1	Tx Available	read	0	A frame is available to be transmitted
2	Tx Transmitting	read	0	A frame is currently being transmitted
3-17	unused	read	0	

Table 37-156: Ethernet TX status/control register layout

37.2.4. DMA data offset

Table 37-157: Ethernet TX DMA offset register layout

Bit	Name	Access	Default	Description
0-9	DMA data offset	read/ write	0	The offset within the DMA memory that the next packet is at. This is incremented by the hardware if a frame has been transmitted
10-17	unused	read	0	

Table 37-157: Ethernet TX DMA offset register layout

37.2.5. Interrupt Register

Table 37-158: Ethernet TX Interrupt register layout

Bit	Name	Access	Default	Description
0	Register Address	read/ write	0	Interrupt flag. Write to acknowledge interrupt.
1-17	unused	read	0	

Table 37-158: Ethernet TX Interrupt register layout

37.2.6. Packet count Register

Table 37-159: Ethernet TX packet count register layout

Bit	Name	Access	Default	Description
0-17	Packet count	read	0	The number of packets that have been transmitted since reset

Table 37-159: Ethernet TX packet count register layout

37.3. Ethernet RX

This module handles the RX part of the MII interface

37.3.1. DMA memory

Memory is organized as a ring buffer. Every entry starts with the length in bytes, stored as an 18 Bit int, followed by an Ethernet frame. The Frame check sum (FCS) is part of the received frame and visible to the software. The address of the oldest package currently in memory is stored in the DMA offset register.

The hardware takes care that the length field is zeroed before a valid frame has been received. After a frame has been fully received and the CRC checked, its length is prepended to the data in the memory.

When a package is not needed any more, it must be explicitly discarded. This is done by setting the offset register to the first address after the frame. If the memory is full, all subsequent packages are dropped, until the software discards enough packages in memory.

Note: The hardware writes to the memory in bytes. The frame's length is therefore written as two half-words in two consecutive cycles. Polling code like this

```
int bytes = ETHERNET_RX_DMA.x[ETHERNET_RX.offset];
if (bytes) {
/* Process a frame of length <bytes> */
}
```

is incorrect! If the hardware has written one of the length's half words in the previous cycle and is writing the second half word while the memory is being read, a **wrong value** will be read as the length. When discarding the packet, the offset will then be set to an incorrect value that probably does not line up with the start of a new frame, and frame data will be interpreted as the length field. Only a reset can recover from this error.

Instead, first read the value and check if it is zero, then read it again and use that as the length of the frame:

```
if (ETHERNET_RX_DMA.x[ETHERNET_RX.offset]) {
int bytes = ETHERNET_RX_DMA.x[ETHERNET_RX.offset];
/* Process a frame of length <bytes> */
}
```

The interrupt flag is only raised after both parts of the length have been written, so it is not necessary to read the length twice when using interrupts.

Note: Frames may start near the end of the DMA memory and continue at the start of the memory. Care must be taken to read at the correct addresses. It is best to use the provided functions.

37.3.2. Module parameters

Table 37-160: Ethernet RX module parameters

Parameter	Default Value	Description
MAC_ADDRESS_HIGH	0x0010A4	Top three byte of the MAC address
MAC_ADDRESS_LOW	0x7BEA80	Lower three byte of the MAC address

37.3.3. Module Registers

Table 37-161: Ethernet RX registers

Offset	Name	Access	Description
0	Control	read/ write	Control flags
1	DMA data offset	read/ write	Contains the current data pointer into the DMA memory
2	Interrupt	read/ write	Interrupt signal
3	Packet count	read	Number of packets received since reset
4	CRC error count	read	Number of packets with incorrect CRC sums received since reset
5	MAC Register 1	read	Low 16 Bit of MAC. Provides access to the configured value.
6	MAC Register 2	read	Middle 16 Bit of MAC. Provides access to the configured value.
7	MAC Register 3	read	High 16 Bit of MAC. Provides access to the configured value.

37.3.4. Control Register

Table 37-162: Ethernet RX control register layout

Bit	Name	Access	Default	Description
0	Promiscuous mode	read/ write	0	Receive every frame, regardless of destination MAC. Takes precedence over Ignore Broadcast
1	IrqEn	read/ write	0	Interrupt enable
2	Ignore Broadcase	read/ write	0	Do not receive broadcast frames

Bit	Name	Access	Default	Description
3	Ignore CRC errors	read/ write	0	Also receive frames with invalid frame check sum. Independent of MAC address matching
4-17	unused	read	0	

Table 37-162: Ethernet RX control register layout

37.3.5. DMA data offset

Table 37-163: Ethernet RX DMA offset register layout

Bit	Name	Access	Default	Description
0-9	DMA data offset	read/ write	0	The offset within the DMA memory that the next packet is at. This is incremented by the hardware if a frame has been transmitted
10-17	unused	read	0	

Table 37-163: Ethernet RX DMA offset register layout

37.3.6. Interrupt Register

Table 37-164: Ethernet RX interrupt register layout

Bit	Name	Access	Default	Description
0	Register Address	read/ write	0	Interrupt flag. Write to acknowledge interrupt.
1-17	unused	read	0	

Table 37-164: Ethernet RX interrupt register layout

37.3.7. Packet count Register

Table 37-165: Ethernet RX packet count register layout

Bit	Name	Access	Default	Description
0-17	Packet count	read	0	The number of packets that have been transmitted since reset

Table 37-165: Ethernet RX packet count register layout

38. Simulation using ModelSim

SpartanMC projects can be easily simulated using ModelSim.

38.1. Creating a simulation directory

Inside the project, run `make newsim +path=<path>` to create a simulation directory in the specified subdirectory. A testbench and a start script are automatically generated

38.2. Customizing the simulation

The generated file *testbench.v* is the test bench. By default, it instantiates the toplevel module, connects all its inputs and outputs to regs / wires as appropriate and generates clocks. In most cases, you need to edit it to generate input signals like reset or outputs from peripherals.

In *testbench.fdo* , you can customize the simulation run time and other variables.

38.3. Starting ModelSim

Inside the simulation directory, you can start ModelSim by running `vsim -do testbench.fdo` . Note that the firmware gets compiled automatically.

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 – DEBUGGING(3)

Library

The library contains the debugging stack, that talks to GDB, for SpartanMCs without Hardware Debugging Support use "debugging_soft" (Only Memory Breakpoints supported), when Hardware Support is present, use "debugging" to make use of the additional features and remove support for memory breakpoints

changes to firmware/config-build.mk:

add "debugging" or "debugging_soft" to the LIB_OBJ_FILES property

add "debugging_traptable" to the LIB_AS_FILES property or provide your own version of that trap table

changes to codefile containing main():

#include <debugging.h>

add FILE * debugIO = &<PERI>_FILE; as a global variable with <PERI> being the name of the peripheral to use for debugging IO.

call debugging_initialize() as early as possible, the program will wait here for directions from the debugging host. (if no explicit uart port has been assigned to the debugger, call after stdout has been set up)

Hardware Support

Hardware support enables stepping, hardware breakpoints and watchpoints

enable the boolean flag HARDWARE_SUPPORT for the desired SpartanMC core

Default trap mapping should be fine, number of break-/watchpoints can be configured

SYMBOLS

DEBUGGER_UART_BASE

Base address of UART peripheral used for data transfer. Currently, the only supported UART hardware is *uart_light* .

Hooks

The Debugging Stack defines some hooks to ease integration with other peripherals. Simply define the functions as listed below.

void debuggerAfterStop();

Gets called when the program gets stopped.

void debuggerBeforeContinue();

Gets called when the program is about to be resumed.

When using the debugging stack together with memory guards, they should probably be disabled while the program is stopped. Otherwise, GDB cannot read the instruction memory. Code to do this may look like this:

```
int debuggerMemguard;
void debuggerAfterStop() {
   debuggerMemguard = memguard_is_enabled(&MEMGUARD_0);
   memguard_enable(&MEMGUARD_0, 0);
}
void debuggerBeforeContinue() {
   memguard_enable(&MEMGUARD_0, debuggerMemguard);
}
```

LIMITATIONS

Debugging information provided by the toolchain was faulty and corrupt. This has been fixed somewhat. GCC generates valid and comprehensive debugging information that can be read by *spartanmc-objdump* . Trying to load it in GDB however was not tested yet and may or may work.

Building GDB

If following the current setup manual, GDB (*spartanmc-gdb*)is part of a standard setup.

GDB Usage

start **spartanmc-gdb --baud 115200**

connect to the target device with **target remote /dev/ttyUSBx**

Alternatively, you can do both in a single command with **spartanmc-gdb --baud 115200 --ex="target remote /dev/ttyUSBx"**

Depending on which version of the debugging library you are using you will have access to either "break" or "hbreak", "watch", "stepi"

Skipping Initial Breakpoint manually

Use any terminal program to send **$c#67** to continue execution.

MANPAGE – SPMC-LOADER(1)

NAME

spmc-loader – update firmware on SpartanMC systems

SYNOPSIS

spmc-loader*PORT SYSTEM_ID SPH_FILE*

DESCRIPTION

Overview

Modifies program memory content on the current SpartanMC system by uploading an updated version of the firmware image using a serial port connection (UART).

In normal use cases *spmc-loader* will be invoked by *make upload* . All options described below will be set to proper values corresponding to the current project in that case.

Upload process

The upload process uses a serial UART connection to transfer new program memory content to the SpartanMC processor core of the target system. To enable the ability to receive bytes and store them into memory, a special startup routine need to be compiled into your initial firmware image. This is accomplished by specifying *startup_loader* instead of *startup* in the list of library objects. See *startup_loader* for details on the upload process.

OPTIONS

PORT Specifies the serial port the UART of the target system is connected to.

SYSTEM_ID 5-digit hexadecimal number of the form 0x12345 identifying the hardware design currently present on the FPGA. This number is sent to the target device and must match the number stored within your current hardware design during synthesis. The actual value is determined by calculating a checksum over all hardware synthesis parameter values present in the design. If both numbers do not

match, the firmware upload is aborted. This mechanism prevents uploading a firmware image to the wrong hardware platform.

SPH_FILE Specifies the *.sph file to upload to the processors program memory.

SEE ALSO

startup_loader spartanmc-project sph

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 FILE set in stdout.

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

39. Scriptinterpreter for jConfig

In the current version of jConfig, the software to configure SpartanMC-SoC, a scriptinterpreter for Lua is added. The Luascripts are used for two things: First to do things automatically when the configuration is changed or build. Second for a Terminal in jConfig.

A script can be triggered automatically with various actions from the user. Four actions are implemented. First when a module is added, second before a module is removed, third when a parameter from a module changes and fourth when the configuration is build. The class *LuaScriptPlugin* in jConfig offers more options to trigger a script if wanted e.g *onFirmwareChanged* . In the terminal you write normal luacode, that means you can write whole functions. You need to use the send button to execute your script you write in the terminal.

You can write both sorts of scripts yourself, if you wish to do so. The scripts are located in the directory with the module.xml and have the name of the action that trigger that script (add, remove, parameterchanged). If the module.xml has a prefix, the lua script also needs one. (This only counts for the already implemented triggeractions). When the script is triggered the element, which triggered the script *(currentElement)* and the subsystem this element is in *(currentSubsytem)* are saved in variables. The onBuilding file is in the same directory as the macro file.

39.1. Methods for the Luascripts

There are 21 extra methods to use in the Luascripts, which are the interface to libjconfig, the library of jConfig.

1. **newdoc(LuaString targetname):** Makes a new document with the target(name).
2. **open(LuaString pathtofile):** Opens the file with the path, workingdirectory + pathtofile.
3. **saveas(LuaString savelocation):** Sets the save location to workingdirectory + savelocation and saves the document.
4. **save():** Saves the document to the determined save location.
5. **config():** Returns the current system configuration as a stream.
6. **getTargetName():** Returns the name of the current target hardware.
7. **contains(LuaString modultypeï¿½):** Returns the number of elements in the configuration with the given module type.
8. **getParent(LuaValue moduleï¿½):** Returns the parent of the given module.
9. **getModule(LuaString moduletype, LuaValue subsystemï¿½ï¿½, LuaInteger numberï¿½):** Returns the module from the given type in the given subsystem. If the parameter subsystem is a boolean, it looks for modules not within a subsystem. When there is more than one element of the same module type in the same section

you can determine which one you want with the number parameter. The number is always the spot in the list not the number in the name.

10. **get(LuaString moduletype, LuaInteger numberï¿½):** Returns the module from the type and numberplace form the hole configuration. This method can be used for loops, if you want all from the same type, or as a short version for *getModule()* , but *getModule()* is more save to use when the configuration is big.
11. **getSubsystem(LuaInteger number):** Returns the subsystem in the spot of the list with the number.
12. **add (LuaString moduletype, LuaValue subsystemï¿½ï¿½ or LuaBoolean inConfig):** adds an element with the module type in the subsystem or in the configuration, if the second parameter is a boolean. SpartanMC cores are automatically added in the configuration. The method returns the added element.
13. **remove(LuaValue module):** Removes the given module.
14. **connectBus(LuaValue bus, LuaValue module1, module2ï¿½):** Connects the bus from the module1 with the one from module2.
15. **connectPort(LuaString port1, LuaString module1, LuaValue port2, LuaValue module2ï¿½):** Connects two ports.
16. **connectPin (LuaString pin, LuaString port, LuaValue moduleï¿½):** Connects a port with a pin.
17. **partialConnection (LuaValue port1, LuaString module1, LuaValue port2, LuaString module2, LuaInteger start1, LuaInteger start2, LuaInteger width):** Connects a port with a pin or with an other port, when not the whole width from the ports are used. If you don't pass start1 or start2 and the port is an input , the next free Connection after the highest taken Connection is used. When it is an output the start is 0. And if you don't pass the width parameter the entire width form the smaller port is used. port2 is a pin if module2 is NIL, not given.
18. **dissconnect(LuaString element, LuaValue moduleï¿½):** Disconnects any given pin/port/bus from the given module.
19. **dissconnectAll(LuaValue moduleï¿½):** Disconnects all ports from the module from everything.
20. **setParam(LuaString parameter , LuaValue value, LuaValue moduleï¿½):** Sets the parameter from the given module to the given value.
21. **getParam(LuaString parameter, LuaValue moduleï¿½):** Returns the value of the parameter from the given module

ï¿½ If you don't pass this parameter, the *currentelement* or *currentsubsystem* are used.

ï¿½ You can choose if you pass the subsystem or only the spot in the list the subsystem has. Beginning with 1.

ï¿½ If you don't pass this parameter, the first found Element is returned

39.1.1. Macros

The macros are functions which use the methods above. They are used to make the configuration with the terminal faster. All macros are in the file in the directory $SPARTANMC_ROOT/src/javaTools/scriptInterpreter/scripts/macro.lua and are loaded with the start of jConfig. You can extend this file if you wish.

addinall(LuaString moduletype): Adds an element in every subsystem with the given module type.

setinall(LuaString parameter, LuaValue value, LuaString moduletype): Sets the same parameter in every module with the given type to the same value.

getfromall(LuaString parameter, LuaString moduletype): Prints the value the parameter in every module with the given type.

39.2. Scripts

The already existing script, you can and should complete them if you wish.

39.2.1. If a component is added

1. clock
2. core_connector
3. dispatcher
4. konzentrator
5. SpartanMC-core
6. uart/uart_light
7. uart_selector

40. microStreams

microStreams is a Cetus based tool, that allows the transformation of the source code form a single threaded application to multi-core streaming pipeline program. It is evident that constructing processing pipelines is only useful for repetitive tasks whose throughput shall be increased. Thus, this approach is very well applicable for microprocessors running repetitive tasks on bare metal. So instead of trying to run several instances of this application and count on data parallelism, microStreams extracts different steps as dependent tasks of the application, constructing a data pipeline. However, an additional parallelization in form of a superscalar pipeline is additionally possible.

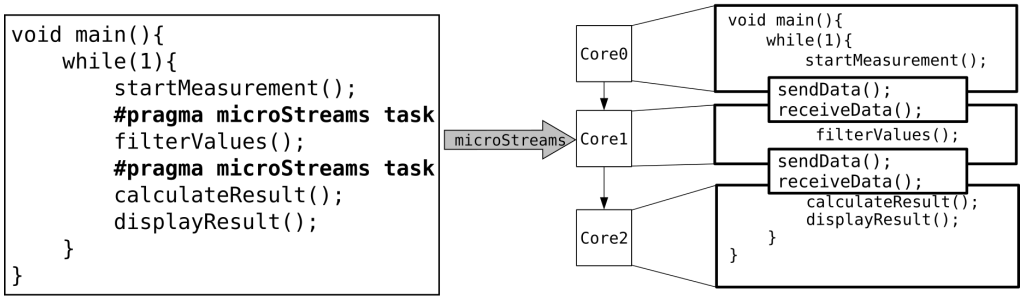


Figure 40-55: Parallelizing Source-Code with microStreams

40.1. Usable Pragmas

In order to successfully transform the application, the user has place few pragma annotations to indicate different tasks in the application. The possible annotations for microStreams are in the form of `#pragma microstreams task [optionals]` . While the [optionals] tag can hold one or more (space separated) of the follwing values:

- `in(variable_name, other_variable)` : Sets the variables listed in the brackets as forced input variables for the task. It is in practice rarely necessary to use this argument, since the variables are in most cases automatically recognized.
- `out(variable_name, other_variable)` : Sets the variables as forced output variables of the task. For details see the previous in optional
- `replicate(nr_of_replications)` : Replicate this task as many times as specified. The tasks will then work in parallel and the replicated tasks can handle new workloads while the original task is still busy processing.
- `end` Ends a task pragma explicitly. Otherwise tasks are ended by scope ends or new task pragmas. This can not be used in combination with other [optionals]

As shown in the example in the following Listing. Pragmas can be placed before function calls, loops or ordinary statements. The created tasks code will reach until the next microStreams task pragma, the end of the scope where the pragma was placed in or until an explicit task end pragma is reached. However the task end pragma is rarely necessary.

```
#pragma microstreams task
void foo(int x){
    int ret = 4;
    #pragma microstreams task
    ret++;
    return 4;
}

void main() {
    int c[10], i;
    #pragma microstreams task
    for(i = 0; i < 10; i++) {
       c[i]=i*i;
    }
    sum(&c);
    #pragma microstreams task
    print(&c);
    #pragma microstreams task end
}
```

40.2. Processing Pipeline

A processing pipeline is automatically created from the pragmas if the data dependencies allow it. microStreams will split the source code at the declared pragmas and create a dependency graph. The dependency reflects the usage of common variables in the tasks. Based on the dependency graph a communication infrastructure between tasks is created to exchange the non exclusive variables. Each task will be mapped to one processing core and a hardware communication infrastructure with simple FIFO buffers is designed. An example for such a system can be seen in the following Figure. It was demonstrated that this methodology is applicable despite of exchanging big data arrays between the tasks in http://ieeexplore.ieee.org/document/7518530/ .

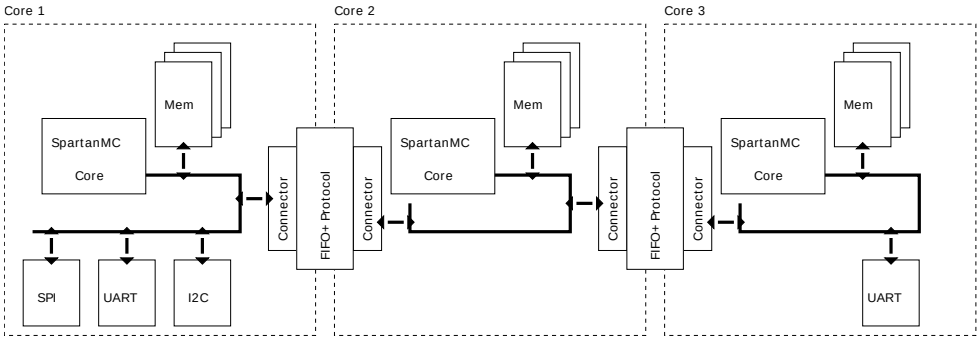


Figure 40-56: A sample SpartanMC based multi-core system consisting of three cores and several peripheral components

40.3. Performace Evaluation

In addition to the automated firmware splitting the tool also offers evaluation techniques to help users judge the created design. The tool can create an environment to measure the time spend for task execution and data transmission to the next task with cycle counters. With those measurements the user can determine if the pipeline stages are balanced and see the communication overhead. The performace evaluation uses performance coutners for each core, and the results of each core will be send to the first core which contains the so called base task via a global memory.

40.4. Created Files

As output, microStreams delivers several firmware files and a HW system description. The system description (hardware.xml) can be read into the system builder (jConfig) via the import flag and additionally selecting a target flag specifing for which FPGA Board to build. Note, that the import is able to add all necessary components and mostly connect them properly. However the configuration of the components such as for example RAM Blocks for memory is set to a default value and should be revised manually (based on the single core design). A good starting point is to set the RAM amount of each core to the one of the single core design and then start reducing and checking if the system still works.

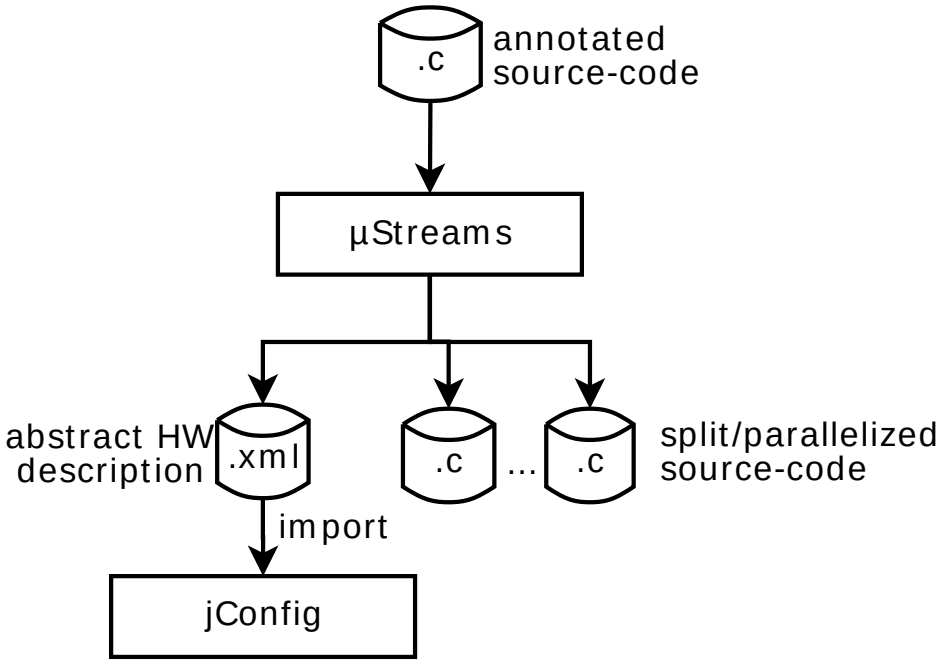


Figure 40-57: microStreams toolflow

40.5. Commandline Options

microStreams can be called in a project directory with: make microstreams +args="[options]" +firmware="firmware folder to analyze" . The following options are possible:

- `-a, --autolibspartanmc` : Automatically detect SpartanMC lib folder. Default: false.
- `-e, --evaluate` Add constructs for a performance measurement evaluation on each core. Default: false
- `-h, --help` Show this help message Default: false
- `-i, --inspect` Shows the generated model with the tasks and task dependencies in a graphical window. Default: false
- `-l, --libs` Give path to additional libraries i.e. header files. Paths are separated by comma. Default: []
- `-g, --no-global-mem` Tells weather we want to move variables to global memory or not. Creates HW and SW. Default: false
- `-p, --project` Should point to the directory where the spartanmc project to be modified resides. Default: .
- `-s, --sim` Generate hardware for a simulated target. Default: false

41. microStreams - AutoPerf & SerialReader

AutoPerf is a simple but quite useful tool, based on parts of microStreams, for profiling applications. AutoPerf expects the source code of an application to profile as input and injects calls to the cycle counter of the SpartanMC processor into it. Those calls will be injected before and after function calls, loops and successive code blocks, as shown in the following listing. By default only the body of the main function will be profiled. With a pragma (`#pragma autoperf`) ahead of another function or inside another functions body the user can profile different parts of the firmware as needed. After running the instrumented code on the device, the cycle counter results can be dumped via UART. The program AutoPerf-SerialReader can read the UART output, separate system out and performance results and write the performance-counter report into a CSV file. This report contains an exact application profile showing which part of the source code took how many cycles to execute. An example of such an output can be seen in the following table. The report contains a field specifying the location of the measurement, the source code line where the measurement was started in the original source (not in the modified program) and the execution time for this step. The application profile is an important step to choose a good pragma placement with microStreams.

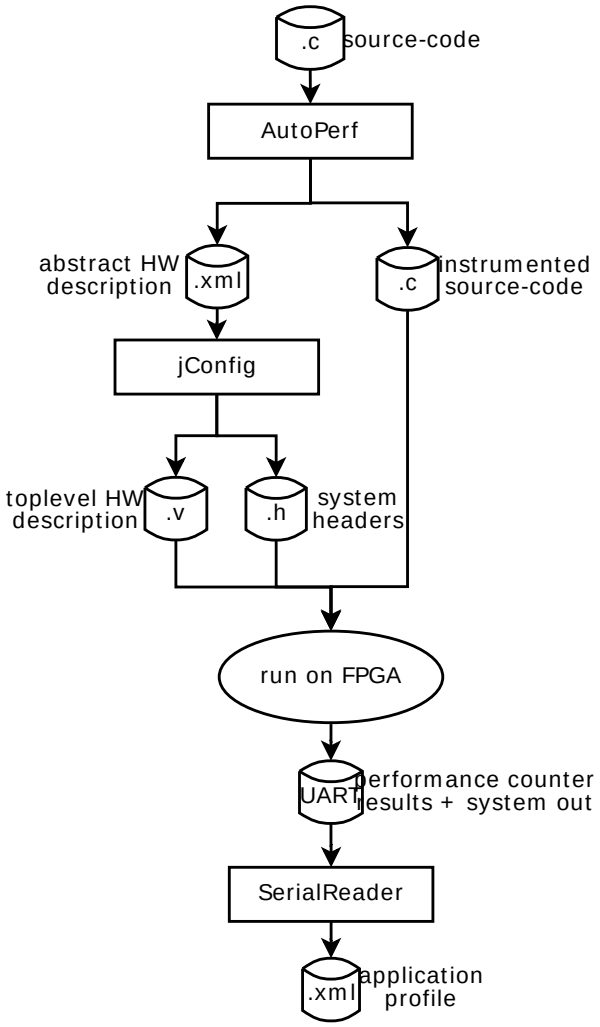


Figure 41-58: AutoPerf workflow

Input Source-Code:

```
void main() {
    int c[10], i;
    for(i = 0; i < 10; i++) {
        c[i]=i*i;
    }
    sum(&c);
    print(&c);
```

Instrumented Source-Code:

```
void main() {
    perf_auto_init();
    perf_auto_start();
    int c[10], i;
    perf_auto_stop(0, perf_results);
    perf_auto_start();
    for(i = 0; i < 10; i++) {
```

```
        c[i]=i*i;
    }
    perf_auto_stop(1, perf_results);
    perf_auto_start();
    sum(&c);
    perf_auto_stop(2, perf_results);
    perf_auto_start();
    print(&c);
    perf_auto_print(3, perf_results);
```

Source File	Function	Code line	Cycles
main.c	main	2	2
main.c	main	3	215
main.c	main	6	385
main.c	main	7	196

Table 41-166: Performance Report

41.1. Commandline Usage

AutoPerf can be called in a project directory with: make autoperf +args="[options]" +firmware="firmware folder to analyze" . The following options are possible:

- `-h, --help` Show this help message Default: false
- `--loopOpt` Switches Mode to profiling pragma ustreams loopOpt autoperf statements Default: false
- `--profile-arrays` Switches the report of array sizes on Default: false
- `-p, --project` Should point to the directory where the spartanmc project to be modified resides. Default: .

AutoPerf-SerialReader can be called in a project directory with: make serialreader +args="[options]" . The following options are possible:

- `-s, --array-size-dump` Specify the output File for the dumped array sizes. Default: arrays.csv
- `-a, --autoperf` Same as '-t autoperf'. Overwrites -t Default: false
- `-h, --help` Show this help message Default: false
- `-l, --loopOpt` Same as '-t loopopt'. Overwrites -t Default: false
- `-m, --microstreams` Same as '-t microstreams'. Overwrites -t Default: false
- `-o, --output` Specify the output File for the results. Default: results.csv
- `-p, --port` Set the serial port to listen to Default: /dev/ttyUSB0

- `-t, --target` Set the target/ Or from which application the performance results are generated Default: autoperf Possible Values: [microstreams, autoperf, loopopt]

42. LoopOptimizer

LoopOptimizer is a programm used to divide loops into multiple parts. These parts can be executed on multiple cores by running MicroStreams afterwards. AutoPerf can be used to improve the processing of loops.

42.1. Preparing your firmware

Before you run LoopOptimizer, you need to set LoopPragmas in your firmware. LoopPragmas need to be set directly in front of all loops that should be transformed. A LoopPragma looks like this: `#pragma microStreams loopOpt [number of splits] [optional parameters]` .

Possible parameters for `[number of splits]` (only one parameter at once):

- Number n: Direct specification by setting the number of splits.
- `maxCycles(` [Number as an Integer] `)` : Passes LoopOptimizer that none of the transformed loops should run longer than [Number as a Integer] Cycles. Needs AutoPerf Data to execute this.

Possible Parameters for `[optional parameters]` (multiple parameters possible at once):

- `autoperf` : Specifies that there is AutoPerf data available for this loop. Simultaniosly also serves as a Pragma for AutoPerf when running it in loopOpt mode.
- `distribution(` [n numbers seperated by comma] `)` : Passes LoopOptimizer to split the loops in a way that the runtime of the first transformed loop in relation to the runtime of all loops is approximatley equal to the relation between the first number of distribution() and the sum of all numbers. Can be executed with or without AutoPerf data (assumes equal runtime of all iterations and statements for the latter).
- `iterationVariable(` [name of a variable] `)` : Specifies which variable of the loop is the iterationVariable. Usually only necessary if LoopOptimizer terminated with a error referencing to this Pragma.
- `useSplitting` / `useFission` : Passes LoopOptimizer to use Splitting or Fission on this loop. Usually LoopOptimizer takes a good decision which shoule be used automatically. Splitting means that each transformed loops executes a part of all iterations, Fission means that each transformed loop executes a part of the body independent from the rest of the body.

42.2. Executing LoopOptimizer

Run `make loopOptimzer +firmware="<path>" +args="<args>"` to execute loopOptimizer. A new firmware will be created in the project folder with the transformed loops. The following parameters in `args` are possible:

- `-t, --taskpragma` : Passes LoopOptimizer to automatically set `#pragma microStreams task` between all transformed loops. See MicroStreams chapter for details.
- `--profile` [path]: Passes the Path to an AutoPerf profile that has been previously made by running AutoPerf in LoopOpt mode.
- `-p, --project` [path]: Specifies the path to the current project if not executed in this folder.
- `-a, --autolibspartanmc` : Specifies that LoopOptimizer should evaluate the `$SPARTANMC_ROOT` variable to find the SpartanMC Root location.
- `-l, --libs` [libaries]: Passes additional Libaries needed for executing the firmware, separated by comma.
- `-h, --help` : Show help message.

42.3. Example Workflow of LoopOpt

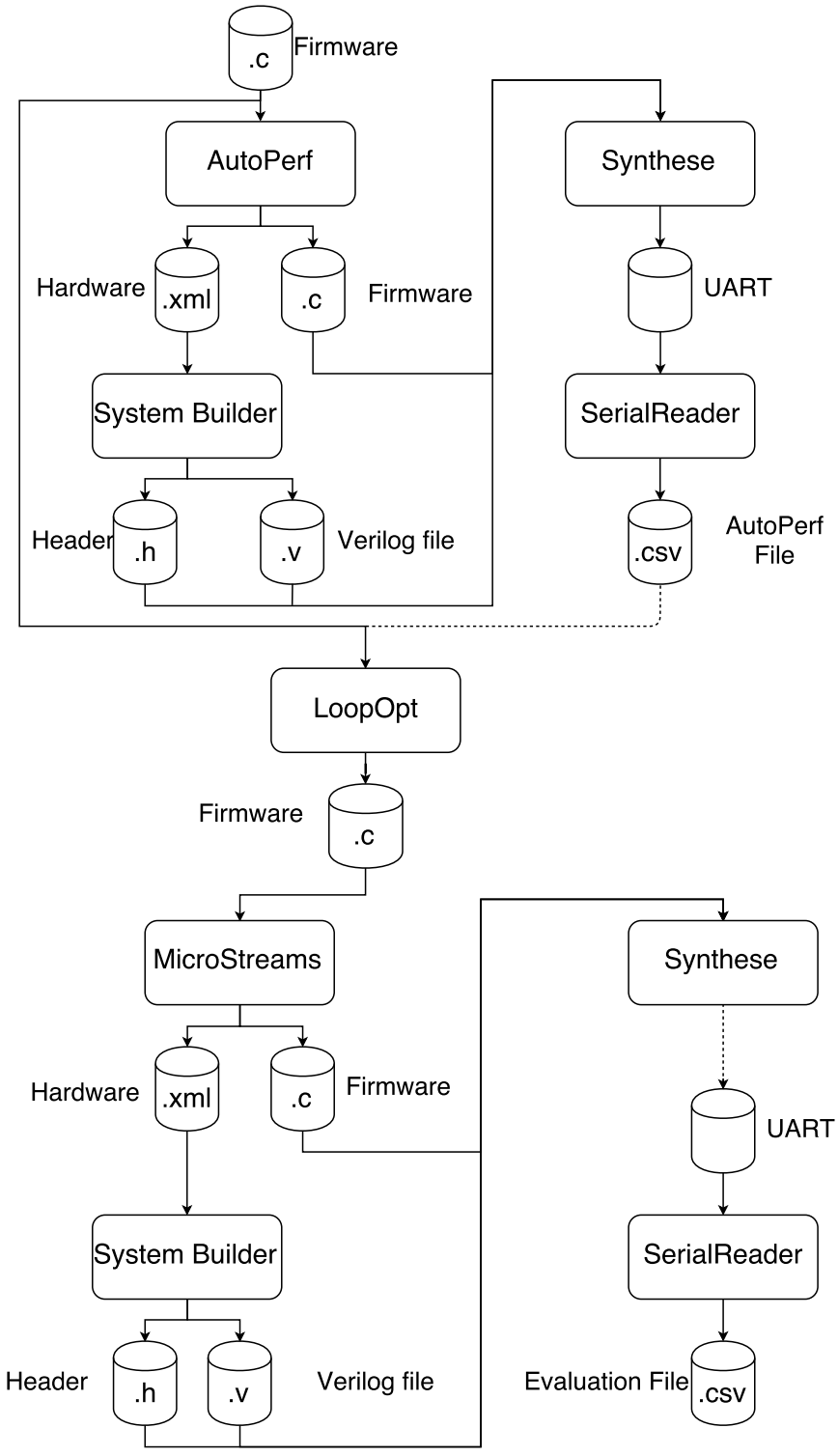


Figure 42-59: Beispielhafter Workflow eines einfachen Projekts

43. AutoStreams

AutoStreams is a java tool which finds a performance-optimal pipeline division for a sequential C program. It inserts task pragma annotations (`#pragma microstreams task`) to mark split points in the source code. The modified firmware can be used by microStreams in a next step to create the multi-core firmwares. For automatic pipelining, an AutoPerf performance profile is needed which contains the measured clock cycles of the individual statements in the code. Moreover, two result text files of an automated performance analysis of the core connector peripherals are required to estimate send and receive cycles for variables to be sent at a certain split point.

Pipelining can be performed in two different ways: either by specifying a maximum cycles count / time for each pipeline stage or by specifying a fixed count of stages for which the fastest possible pipeline is to be found.

43.1. Prerequisites

Apart from the input firmware / project, three files are needed for automatic pipelining:

- **performance profile:** AutoPerf adds start and stop statements to the input firmware. The modified firmware can then be programmed by the board. A string representation of the measured cycles is printed to the UART interface, which can be listened to by the AutoPerf SerialReader in order to write the corresponding CSV file (the performance profile). Usually, it has the name "results.csv".
- **send cycles text file / receive cycles text file:** The results of a performance analysis of the core connector transmission behaviour. Contains the measured send / receive cycles for ascending data sizes (in SpartanMC words) to be sent over the core connector, using the associated C functions. Currently contains results for data sizes between 1 and 133 words. The script for the automated measurement and for the creation of the text files is presently located under "tests/integrationtests/performance_analysis_core_connector". The file names of the results are "sendTimes.txt" and "receiveTimes.txt".

43.2. Usage and Command Line Options

AutoStreams uses the following command line options:

- `-p , --project` : Path to the SpartanMC project folder containing the firmware to modify. Default: current directory
- `--profile` : File path of the performance profile (CSV file), absolute or relative to SpartanMC project directory. Default: "results.csv" in the project or firmware directory.

- `--core-conn-perf-results` : Directory which contains the two result text files of the core connector performance analysis, absolute or relative to SpartanMC project directory. Default: Project directory.
- `-f , --clock-frequency` : Used SpartanMC clock frequency in MHz.
- `-m , --fpga-model` : FPGA model to determine the maximum frequency from.
- `-c , --cycles` : Maximum cycles for each pipeline stage.
- `-t , --max-pipeline-time` : Maximum execution time for each pipeline stage in ns. (Note: Cycles have priority if both -t and -c are specified.)
- `--stage-count` : Desired count of pipeline stages. (Can also be combined with -t / -c).
- `--connect-all` : Ensure that all stages of the resulting pipeline are connected by core connectors for synchronization. (Only insert pragmas at points where variables are transmitted from one core to another.)
- `-r , --report` : Write a detailed report (text file) of the created pipeline and the used configuration to the folder of the created firmware.
- `-ra , --report-all` : Like -r, but with information about (almost) all theoretically possible pipelines matching the specified requirement. Useful for comparing the resulting pipeline arrangement with other possible pipelines.

43.3. Example

Input firmware:

```
void main() {
    changeStreet(&addr);
    swapEntries(&bigArray);
    changeStreet(&addr);
    changeStreet(&addr);
    swapEntries(&bigArray);
}
```

Performance profile:

```
main.c:main:changeStreet(( & addr)):29;1113
main.c:main:swapEntries(( & bigArray)):30;1271
main.c:main:changeStreet(( & addr)):31;1113
main.c:main:changeStreet(( & addr)):32;1113
main.c:main:swapEntries(( & bigArray)):33;1271
```

Resulting AutoStreams firmware, using the argument `--stage-count 2` :

```
void main() {
    changeStreet(&addr);
    swapEntries(&bigArray);
    changeStreet(&addr);
#pragma microstreams task
    changeStreet(&addr);
```

```
    swapEntries(&bigArray);
}
```

Resulting AutoStreams firmware, using the argument `--stage-count 4` :

```
void main() {
    changeStreet(&addr);
#pragma microstreams task
    swapEntries(&bigArray);
#pragma microstreams task
    changeStreet(&addr);
    changeStreet(&addr);
#pragma microstreams task
    swapEntries(&bigArray);
}
```