Interrupts

General

COFFEE core currently supports connecting eight external interrupt sources directly. If coprocessors are not connected the four inputs reserved for coprocessor exception signalling can be used as interrupt request lines giving possibility to connect twelve sources. An external interrupt handler can be connected to expand the number of sources even further.

If internal interrupt handler is used, the priorities between sources can be set by software, with external handler, priorities will be fixed according to table below. Note that priorities for coprocessor exceptions/interrupts are always set by software.

Internal exception handler has synchronization circuitry allowing signals to be directly connected to the core. If an external handler is used, synchronization is bypassed in order to reduce signalling latency. See interface document.

Status signals are provided to give feedback about the status of the latest interrupt request. Interrupt sources can be masked individually and disabled or enabled all at once using di and ei –instructions.

All interrupts are vectored. The address of an interrupt service routine can be the corresponding vector directly (see interrupt registers) or a combination of the vector and an offset given externally.

Table 1, Interrupt priorities ***if external handler is used****, 0 - highest.*

Priority	Name
software controlled	coprocessor number 0 exception/interrupt
	coprocessor number 1 exception/interrupt
	coprocessor number 2 exception/interrupt
	coprocessor number 3 exception/interrupt
15	external interrupt 0
15	external interrupt 1
15	external interrupt 2
15	external interrupt 3
15	external interrupt 4
15	external interrupt 5
15	external interrupt 6
15	external interrupt 7

Interrupt interface modes

Two interfacing modes are supported: *external handler* and *internal handler* The mode is selected by EXT_HANDLER –signal. A summary is given in the table below. Note that an external handler usually allows priorities between sources to be set quite freely. In this case an external handler sees a fixed priority between the lines it is driving. The user may see whatever configuration.

table 2, Interrupt interface modes

mode/ EXT_HANDLER state	request signal timing	interrupt vector calculation	priorities
internal handler / LOW	asynchronous	BASE address directly [1]	set by software (see configuration registers)
external handler / HIGH	synchronous	BASE(31 downto 12) & OFFSET & "0000" [2]	fixed between lines (usually configurable via external handler)

[1] BASE address is set by software. See CCB configuration registers.
[2] 8 bit OFFSET provided by an external handler, & means concatenation. Coprocessor exceptions/interrupts do not use OFFSET.

Signalling an interrupt

An interrupt request is signalled by driving a high pulse on one of the interrupt lines. The timing of the pulse depends on the mode: whether an external handler is used or not. The timing of the coprocessor interrupt/exception lines is fixed. See interface –document about the timing details. Each interrupt line has a pulse detection ciruitry and an interrupt request gets through when that circuitry sees a pulse, that is, after seeing a falling edge. If an external handler is used, the offset should be driven simultaneously with the request line, see interface –document.

Once detected, a request is saved in a register called INT_PEND, which is visible to the software. After this it has to go through the priority resolving and masking stage. The following conditions have to be true for a pending request to get through:

- interrupts enabled: IE –bit in processor status register (PSR) must be high.
- Interrupt mask register has to have a high bit ('1') in the corresponding position.
- No interrupts with higher priority are pending or in service.
- No exceptions on pipeline (see document about exceptions)

Once a request gets through, the processor starts execution of an interrupt service routine as soon as possible: pipeline is executed to a point where it is safe to switch to interrupt service routine. This takes 1 – 3 cycles depending on the contents of the pipeline. When a service routine is started the corresponding bit in INT_SERV –register is set. At the same time, the processor drives a pulse to INT_ACK –output in order to signal to an external handler that the latest request got through and is now in service. *This is the earliest point where a new request from the same source can be accepted.*

The latency from asserting an external interrupt signal to the moment when control of the core detects the signal is multiple cycles. The latency also depends on the mode of operation of the interrupt interface. Latency is calculated from the falling edge of the EXT_INTERRUPT –signal. Different cases are shown in the table below.

Table 3, Interrupt signalling latency

Interface type	**signal synchronization**	**pulse detection**	**priority check and masking**	**total cycles**
asynchronous (EXT_HANDLER low)	2 clock cycles	1 clock cycle or less depending on timing of the EXT_INTERRUPT –signal.	1 clock cycle	4
synchronous (EXT_HANDLER high)	-		1 clock cycle	2

Priority resolving

A priority for a particular source is set by writing a four bit value in a field reserved for that source in the EXT_INT_PRI or COP_INT_PRI –register. Priority can have any value between 0 and 15, zero being the highest priority.

Whether the priority is fixed (external handler used) or set by software, priority resolving works the same way. If multiple interrupts are signalled simultaneously, the one with the highest priority (lowest number) will be served first. Note that for coprocessor exceptions/interrupts the priority can allways be set by software.

If multiple sources have the same priority, resolving is performed internally in the following order (COP0_INT having the highest priority):
COP0_INT, COP1_INT, COP2_INT, COP3_INT,
EXT_INT0, EXT_INT1, EXT_INT2, EXT_INT3,
EXT_INT4, EXT_INT5, EXT_INT6, EXT_INT7.

If the same interrupt that is currently in service, is signalled, the interrupt service routine is restarted as soon as it has finished (of course assuming there’s no interrupt requests with higher priority pending). A request with higher priority can interrupt the current service routine if interrupts have been re-enabled with *ei* –instruction (nesting of interrupts).

Switching to an interrupt service routine

The following steps are taken when switching to an interrupt service routine:

- return address is saved to hardware stack (a special logic structure to allow fast switching)
- processor status register (PSR) is saved to hardware stack
- condition register CR0 is saved to hardware stack.
- The start address of an interrupt service routine is calculated(see table 2) and placed to program counter.
- signal INT_ACK is pulsed (***except with coprocessor exceptions/interrupts!***).
- The bit corresponding to the interrupt source is set high in INT_SERV –register.
- The bit corresponding to the interrupt source is cleared from INT_PEND –register.
- Further interrupts are disabled by setting IE bit low in PSR
- Processor status: user mode and, instruction decoding are set according to control registers INT_MODE_IL and INT_MODE_UM. (If superuser –mode is set, register set 2 is selected as default for reading and writing)
- Execution of the interrupt service routine in question is started.

Returning from an interrupt service routine

An interrupt service routine ***has to execute a reti*** –instruction in order to resume program execution where it was interrupted. This causes the following things to happen:

- Processor status is restored from the hardware stack
- CR0 is restored from the hardware stack.
- Program counter is restored from the hardware stack.
- signal INT_DONE is pulsed (***except with coprocessor exceptions/interrupts!***).
- The INT_SERV bit is cleared.
- Interrupts are enabled if they were enabled before entering the service routine. (There is a possibility that di –instruction is executed just before entering the service routine, but after a request got through in which case the interrupt is served but interrupts will be disabled on return)

In chapter Tricks it is explained how to clear a pending interrupt request without executing a service routine.

Internal interrupt handler control & status registers

Bit positions and interrupt sources are associated as follows:
(INT_MODE_IL, INT_MODE_UM, INT_MASK, INT_SERV, INT_PEND)

Bit 11 – EXT_INT7,
Bit 10 – EXT_INT6,
...
Bit 4 – EXT_INT7,
Bit 3 – COP3_INT,
...
Bit 0 – COP0_INT.

Table xx, Internal interrupt handler registers (in CCB)

offset	mnemonic	width	description	notes
02h	COP0_INT_VEC	32	Co-processor 0 interrupt service routine start address.	should be properly aligned.
03h	COP1_INT_VEC	32	Co-processor 1 interrupt service routine start address.	
04h	COP2_INT_VEC	32	Co-processor 2 interrupt service routine start address.	
05h	COP3_INT_VEC	32	Co-processor 3 interrupt service routine start address.	
06h	EXT_INT0_VEC	32	External interrupt 0 service routine base address.	
07h	EXT_INT1_VEC	32	External interrupt 1 service routine base address.	
08h	EXT_INT2_VEC	32	External interrupt 2 service routine base address.	
09h	EXT_INT3_VEC	32	External interrupt 3 service routine base address.	
0ah	EXT_INT4_VEC	32	External interrupt 4 service routine base address.	
0bh	EXT_INT5_VEC	32	External interrupt 5 service routine base address.	
0ch	EXT_INT6_VEC	32	External interrupt 6 service routine base address.	
0dh	EXT_INT7_VEC	32	External interrupt 7 service routine base address.	
0eh	INT_MODE_IL	12	Instruction decoding mode flags for interrupt routines.	See registers – document: PSR
0fh	INT_MODE_UM	12	User mode flags for interrupt routines.	
10h	INT_MASK	12	Register for masking external and cop interrupts individually. A low bit (‘0’) means blocking an interrupt source, a high bit enables an interrupt.	
11h	INT_SERV	12	Interrupt service status bits (active high).	Read only. See chapter Tricks.
12h	INT_PEND	12	Pending interrupt requests(active high).	
13h	EXT_INT_PRI	32	Bits 31 downto 28 : INT 7 priority Bits 27 downto 24 : INT 6 priority ... Bits 7 downto 4 : INT 1 priority Bits 3 downto 0 : INT 0 priority	0 – highest priority 15 – lowest priority Priorities for external interrupts can only be set if internal handler is used.
14h	COP_INT_PRI	16	Bits 15 downto 12 : COP3 priority Bits 11 downto 8 : COP2 priority Bits 7 downto 4 : COP1 priority Bits 3 downto 0 : COP0 priority	

Tricks & notes

Notes

- di –instruction itself can be interrupted! It is guaranteed that instructions between **di, ei** –pair cannot be interrupted but an interrupt can take place between di and the following instruction.

Clearing a pending interrupt without running the service routine

This concerns only clearing a pending request from the internal handler register!
The ability to clear bits in the INT_PEND –register directly would lead to situations where an external interrupt handler would not know the real status of the latest interrupt request because INT_ACK -signal would never go high for these 'canceled' interrupts. This kind of inconsistency is not acceptable and that's why INT_PEND is a read only register.

If there is a need to 'cancel' a request it can be done as follows (If internal CCB is mapped to protected memory area, superuser mode is needed):

- Interrupts should be disabled during these operations!
- Save the current value in the interrupt vector register of the int source in question.
- Replace the old vector with a new one which points to a dummy routine (remember OFFSET, if external handler is present) which executes reti – instruction only (and maybe some acknowledge instructions for external handler).
- Set the interrupt source to highest priority and make sure that no other source shares the same priority (of course save old values).
- Set mask bit for the interrupt source in question (save old value of INT_MASK)
- enable interrupts
- poll the INT_PEND register, disable interrupts when the bit in question is low.
- Restore vector and priorities.
- Continue normally

Do not do this!

- Do not change interrupt priorities while in interrupt service routine if you use nested interrupts (unless you are 100% sure that a new request from a source cannot arise before a service routine is finished). In extreme cases this can lead to hardware stack overflow if interrupt nesting level is twelve and priorities are changed so that multiple requests from a single source can be active simultaneously. Normally an interrupt service routine cannot be interrupted by a new request from the same source because of priority resolving.