SISTEMI EMBEDDED
AA 2011/2012

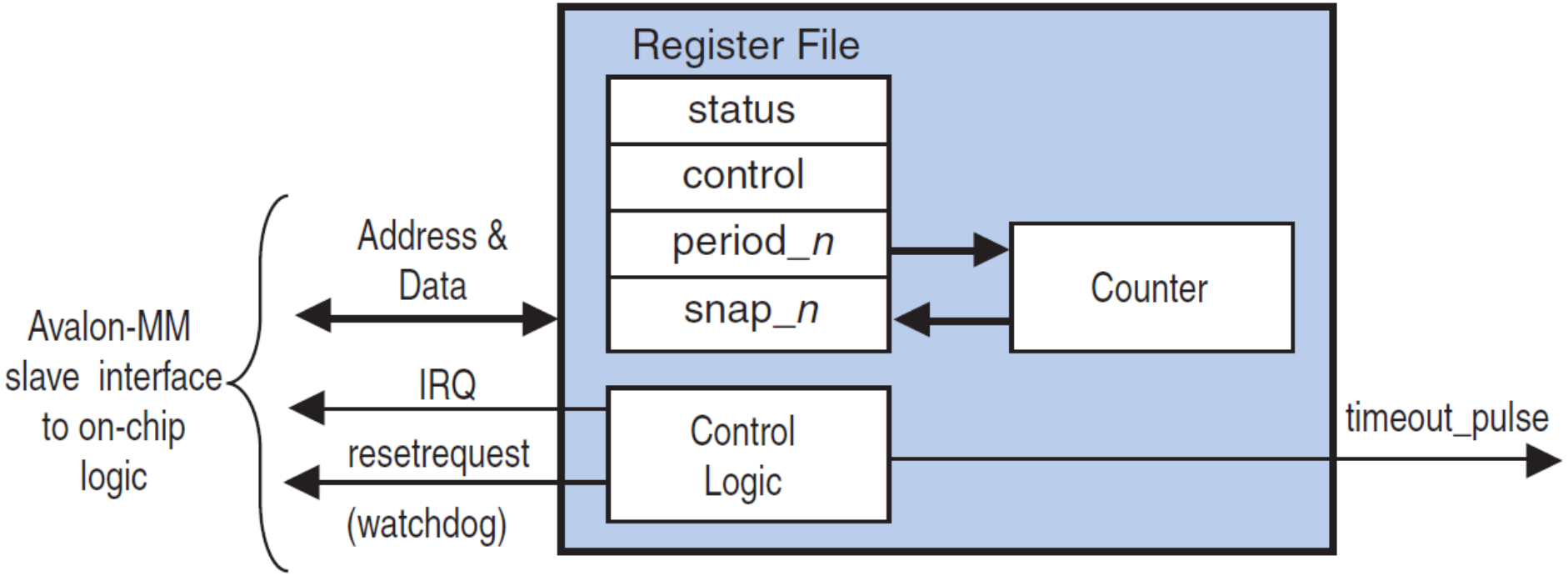
SOPC Nios II
Interval Timer Core

Interval timer core (1)

- 32-bit and 64-bit counters
- Control bits to start, stop, and reset the timer
- Two count modes: count down once and continuous count-down
- Count-down period register
- Option to enable or disable the interrupt request (IRQ) when timer reaches zero
- Optional watchdog timer feature that resets the system if timer ever reaches zero
- Optional periodic pulse generator feature that outputs a pulse when timer reaches zero
- Compatible with 32-bit and 16-bit processors
- Device driver available in the HAL system library

Interval timer core (2)

- **Block diagram**
  - 6x (32-bit counter) or 10x (64-bit counter)
    16-bit registers (certain registers may not be present depending on the configuration)
  - Compatible w/ 16- and 32-bit processors



Interval timer core (3)

- Nios II processor writes the core's control register to:
  - Start and stop the timer
  - Enable/disable the IRQ
  - Specify count-down once or continuous count-down mode
- A processor reads the status register for information about current timer activity
- A processor can specify the timer period by writing a value to the period registers
- An internal counter counts down to zero, and whenever it reaches zero, it is immediately reloaded from the period registers
- A processor can read the current counter value by first writing to one of the snap registers to request a coherent snapshot of the counter, and then reading the snap registers for the full value
- When the count reaches zero, one or more of the following events are triggered:
  - If IRQs are enabled, an IRQ is generated
  - The optional pulse-generator output is asserted for one clock period
  - The optional watchdog output resets the system

Interval timer core (4a)

- Instance configuration using SOPC Builder MegaWizard
  - **Timeout period**: determines the initial value of the period registers; can be changed if depending on the writeable period option
  - **Counter size**: 32- or 64-bits
  - **Hardware options**: 3 pre-set configurations
    - Simple periodic interrupt
    - Full-featured
    - Watchdog
    - Or custom

Interval timer core (4b)

- **Register option**
  - Writeable period
  - Readable snapshot
  - Start/Stop control bits
- **Output signals**
  - Timeout pulse (1 clock wide)
  - System reset on timeout (watchdog)

Interval timer core (4d)

- Watchdog configuration:
  - Set the Timeout Period to the desired "watchdog" period
  - Turn off Writeable period
  - Turn off Readable snapshot
  - Turn off Start/Stop control bits
  - Turn off Timeout pulse
  - Turn on System reset on timeout (watchdog)

Interval timer core (4e)

- **Watchdog behaviour:**
  - It comes out of reset stopped
  - It can be started by writing a 1 to the control register's START bit. Once started, the timer can never be stopped
  - If the internal counter reaches zero, the watchdog timer resets the system by generating a pulse on its *resetrequest* output.
  - To prevent the system from resetting, the processor/program must periodically reset the timer's count-down value by writing one of the period registers (the written value is ignored)
  - If the processor fails to access the timer because, for example, software stopped executing normally, the watchdog timer resets the system and returns the system to a defined state

Interval timer core (5a)

- **Register map**

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	*(1)*					RUN	TO
1	control	RW	*(1)*			STOP	START	CONT	ITO
2	periodl	RW	Timeout Period – 1 (bits [15:0])						
3	periodh	RW	Timeout Period – 1 (bits [31:16])						
4	snapl	RW	Counter Snapshot (bits [15:0])						
5	snaph	RW	Counter Snapshot (bits [31:16])						

Interval timer core (5b)

- **Status register**

Bit	Name	R/W/C	Description
0	TO	RC	The TO (timeout) bit is set to 1 when the internal counter reaches zero. Once set by a timeout event, the TO bit stays set until explicitly cleared by a master peripheral. Write zero to the status register to clear the TO bit.
1	RUN	R	The RUN bit reads as 1 when the internal counter is running; otherwise this bit reads as 0. The RUN bit is not changed by a write operation to the status register.

Interval timer core (5c)

- **Control register**

Bit	Name	R/W/C	Description
0	ITO	RW	If the ITO bit is 1, the interval timer core generates an IRQ when the status register's TO bit is 1. When the ITO bit is 0, the timer does not generate IRQs.
1	CONT	RW	The CONT (continuous) bit determines how the internal counter behaves when it reaches zero. If the CONT bit is 1, the counter runs continuously until it is stopped by the STOP bit. If CONT is 0, the counter stops after it reaches zero. When the counter reaches zero, it reloads with the value stored in the period registers, regardless of the CONT bit.
2	START *(1)*	W	Writing a 1 to the START bit starts the internal counter running (counting down). The START bit is an event bit that enables the counter when a write operation is performed. If the timer is stopped, writing a 1 to the START bit causes the timer to restart counting from the number currently stored in its counter. If the timer is already running, writing a 1 to START has no effect. Writing 0 to the START bit has no effect.
3	STOP *(1)*	W	Writing a 1 to the STOP bit stops the internal counter. The STOP bit is an event bit that causes the counter to stop when a write operation is performed. If the timer is already stopped, writing a 1 to STOP has no effect. Writing a 0 to the stop bit has no effect. If the timer hardware is configured with **Start/Stop control bits** off, writing the STOP bit has no effect.

Interval timer core (5d)

- **period_n Registers**
  - The period_n registers together store the timeout period value
  - The internal counter is loaded with the value stored in these registers whenever one of the following occurs:
    - A write operation to one of the period_n register
    - The internal counter reaches 0
  - The timer's actual period is one cycle greater than the value stored in the **period_n registers**
  - Writing to one of the period_n registers stops the internal counter, except when the hardware is configured with Start/Stop control bits off
  - When the hardware is configured with Writeable period disabled, writing to one of the period_n registers causes the counter to reset to the fixed Timeout Period specified at system generation time

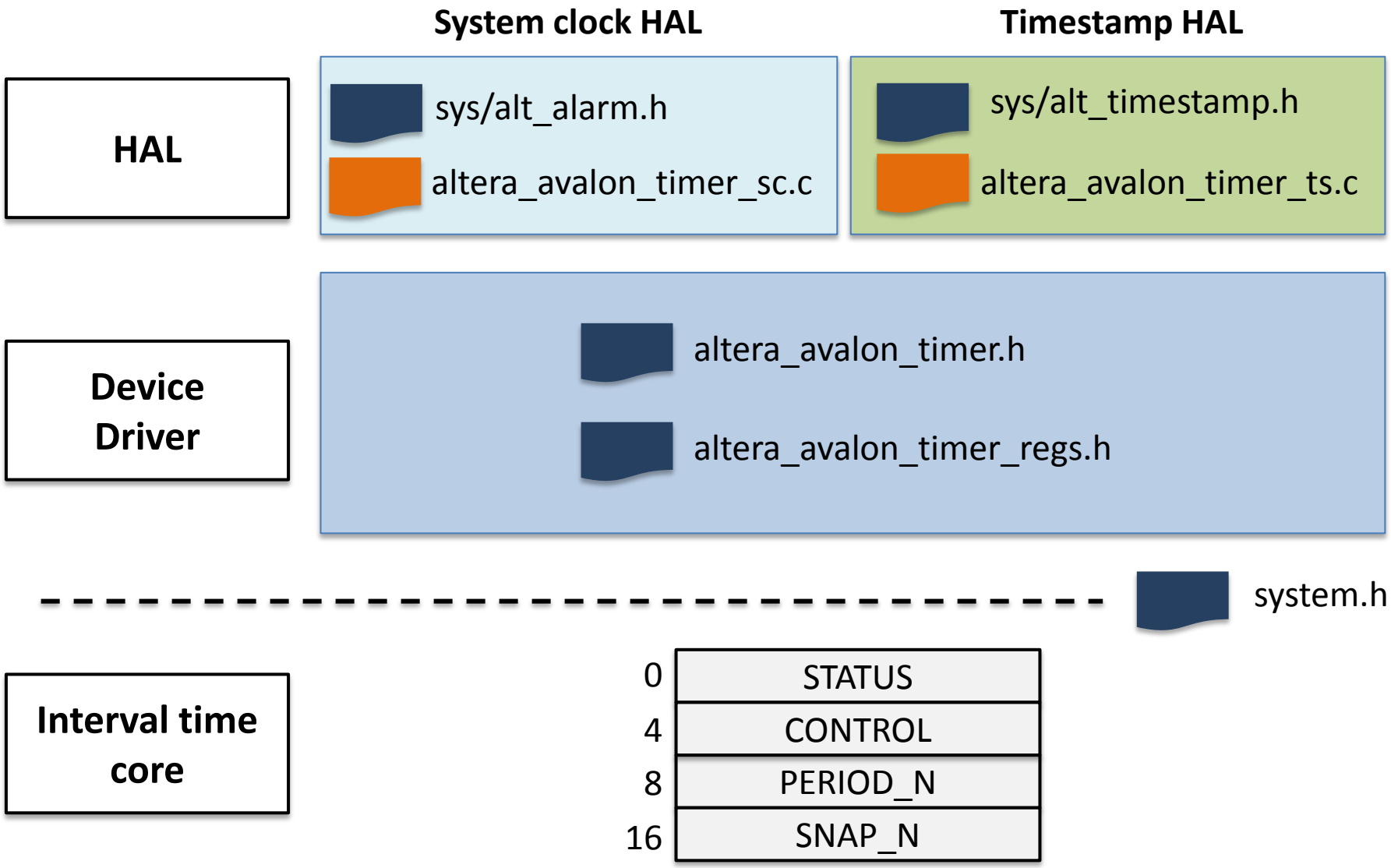
Interval timer core (5e)

- **snap_n registers**
  - A master peripheral may request a coherent snapshot of the current internal counter by performing a write operation (write-data ignored) to one of the snap_n registers
  - When a write occurs, the value of the counter is copied to the snap_n registers

Interval timer core (6)

- **Interrupt Behaviour**
  - The interval timer core generates an IRQ whenever the internal counter reaches zero and the ITO bit of the control register is set to 1
  - Acknowledging the IRQ in one of two ways:
    - Clear the TO bit of the status register
    - Disable interrupts by clearing the ITO bit of the control register
    - Failing to acknowledge the IRQ produces an undefined result

Software programming model (1)



Software programming model (2)

- The device model of the interval timer can be chosen through the **BSP editor**
- This property is recorded in **system.h**

```
/*
 * hal configuration
 *
 */

#define ALT_MAX_FD 32
#define ALT_SYS_CLK INTERVAL_TIMER
#define ALT_TIMESTAMP_CLK none
```

System clock HAL

- Useful for scheduling periodic tasks
  - Can generate the **system tick**
  - The **period of the system tick** is a multiple of the Timeout period of the interval timer
- **Basic HAL functions:**
  - int alt_alarm_start(alt_alarm* alarm, alt_u32 nticks, alt_u32 (*callback) (void* context), void* context);
  - void alt_alarm_stop(alt_alarm* alarm);
  - alt_u32 alt_ticks_per_second(void);
  - alt_u32 alt_nticks(void);
  - See the HAL API Reference for how to use these functions!

Timestamp HAL

- Useful for measuring interval times with high resolution (period of the interval timer clock!)
  - The interval timer peripheral must have the period_n register which is set to the maximum value by the relevant HAL
- **Basic functions:**
  - int alt_timestamp_start(void);
  - alt_u32 alt_timestamp(void);
  - alt_u32 alt_timestamp_freq(void);
  - See the HAL API Reference for how to use these functions!

Putting into practice

- Using the Timestamp HAL profile the wait_ms() function
- Using the System clock HAL make a LED blinking with a given period

References

- Altera, “Embedded Peripherals IP User Guide,” ***ug_embedded_ip.pdf***
  - 28. Interval Timer
- Altera “Nios II Software Developer’s Handbook,” ***n2sw_nii5v2.pdf***
  - Chapters 6. Developing Programs Using Hardware Abstraction Layer