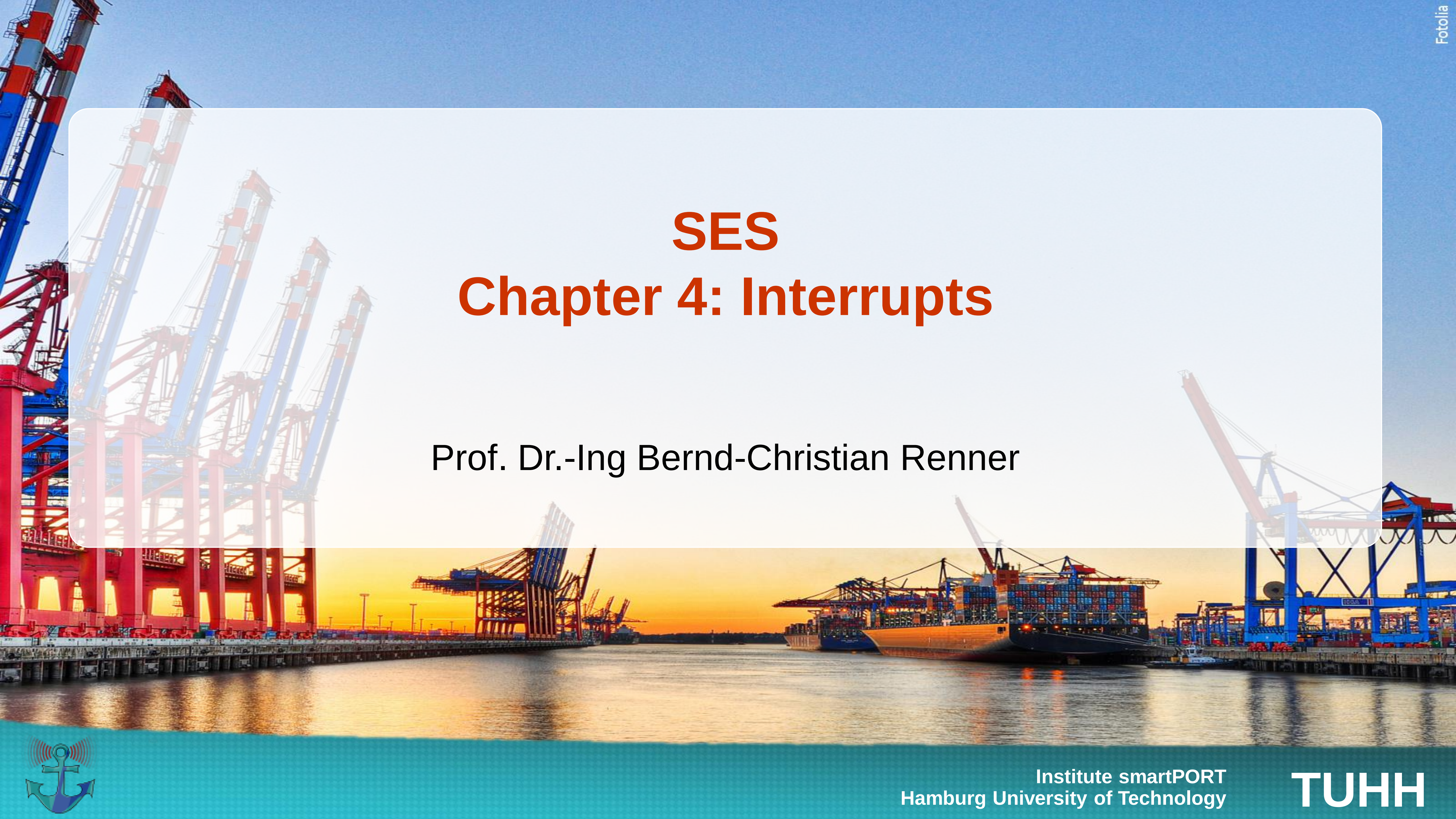


SES
Chapter 4: Interrupts

Prof. Dr.-Ing Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology

TUHH

Contents

1

Event-driven Systems

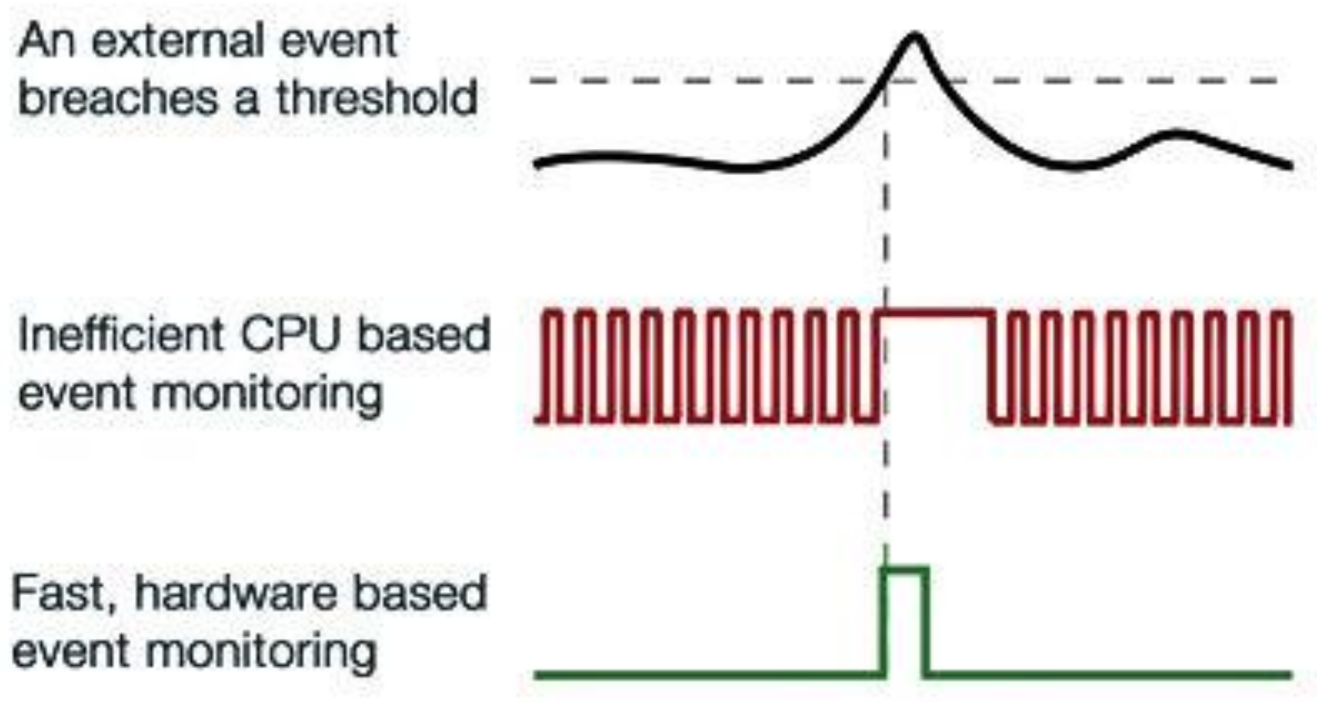
Example: Push Button Handling

Alternatives

1. Polling: Reading the button input regularly
   - If other tasks have to be done as well (e.g. read an RS232 input) there will be a delay between reads of button
   - As long as delays are small compared to speed of input change then no button presses will be missed
   - If however some long calculation are executed then a button press could be missed while processor is busy
2. Asynchronous Solution: Hardware interrupt
   - By using a hardware interrupt driven button reader the calculation could proceed with all button presses captured

Polling vs. Interrupts

- As most events occur asynchronously, embedded systems should be able to react **only** when thresholds are reached and without the need to waste resources by periodically polling sensors


An external event breaches a threshold
Inefficient CPU based event monitoring
Fast, hardware based event monitoring

Event-Driven Systems

- Embedded systems are often realized by event-driven systems
- Programming paradigm in which the program’s flow is determined by events, e.g., soft- or hardware signals
- An **interrupt** is
  - an asynchronous signal to the processor indicating the need for attention or
  - a synchronous event in software indicating the processor the need for a change in execution

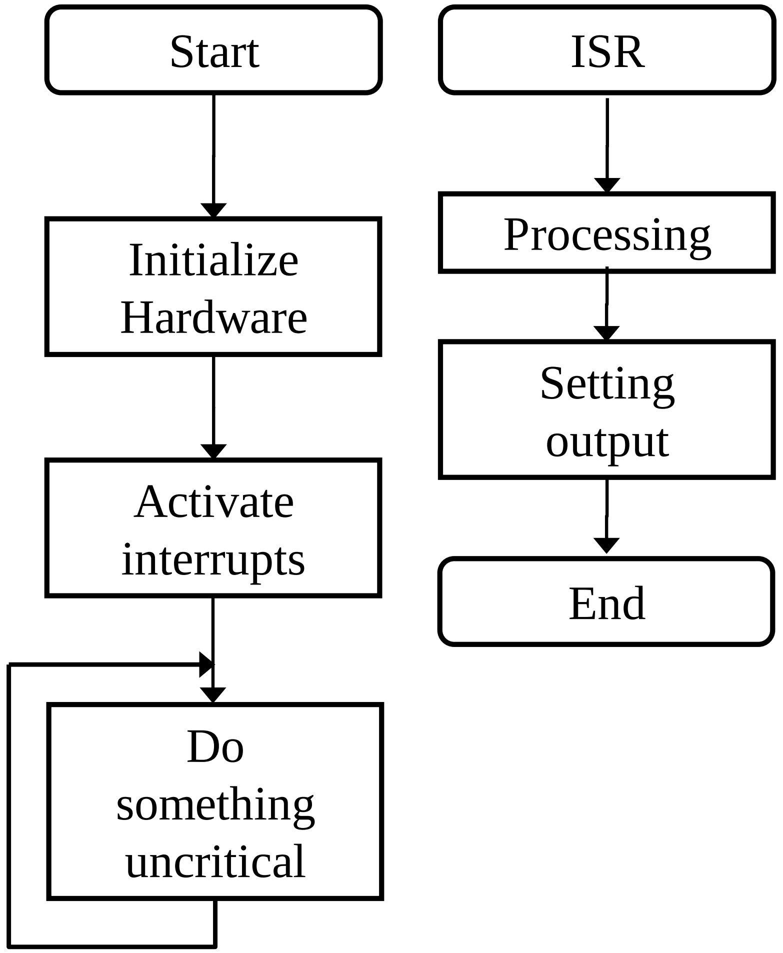
2

Interrupts

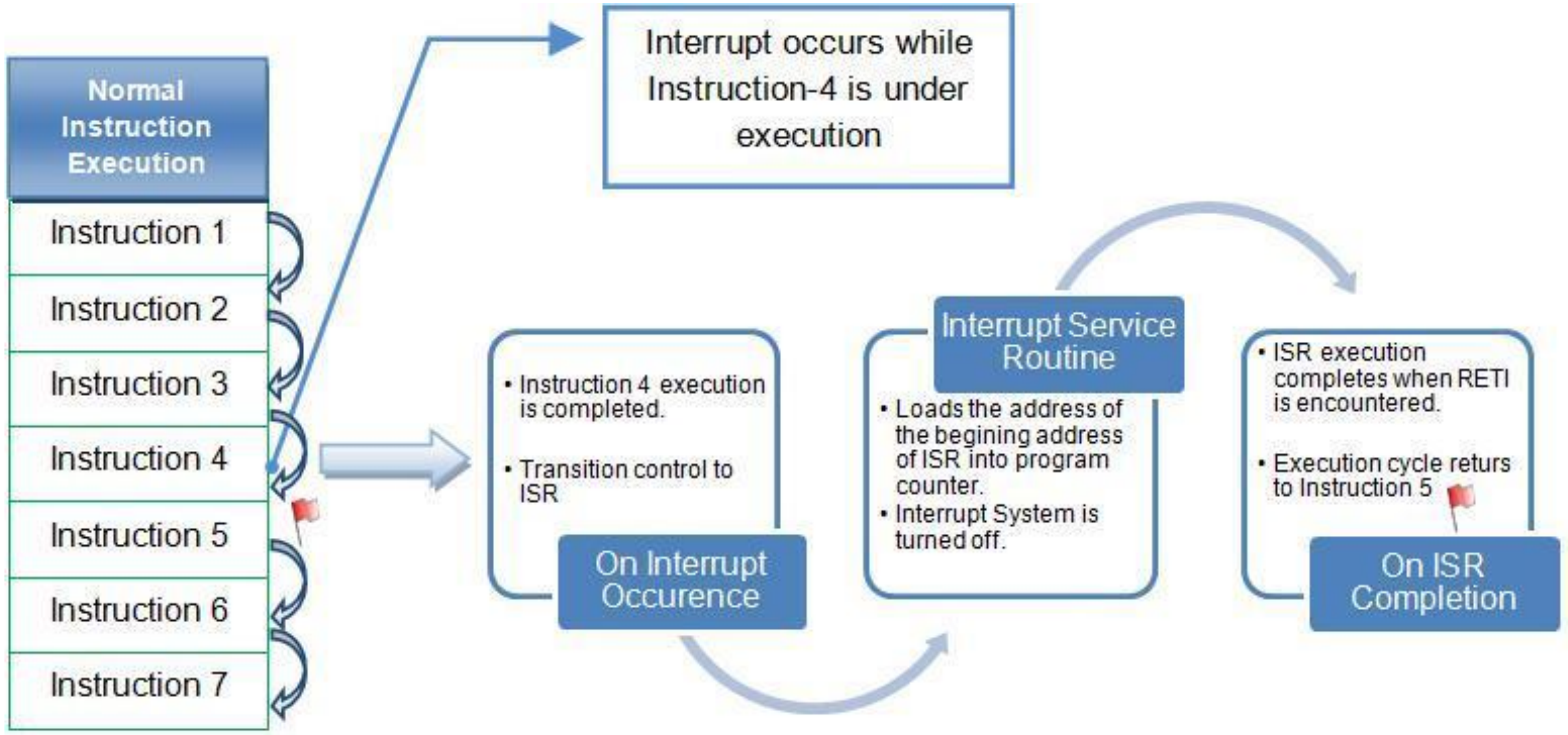
Overview

- Interrupts
  - cause processor to save its state of execution, and begin execution of an **interrupt handler** or an **interrupt service routine** (ISR)
  - are a way to avoid wasting processor's valuable time in polling loops, waiting for external events
- Software interrupts
  - are usually implemented as instructions causing a context switch to an interrupt handler similar to a hardware interrupt
- An act of interrupting is referred to as an **interrupt request** (IRQ)

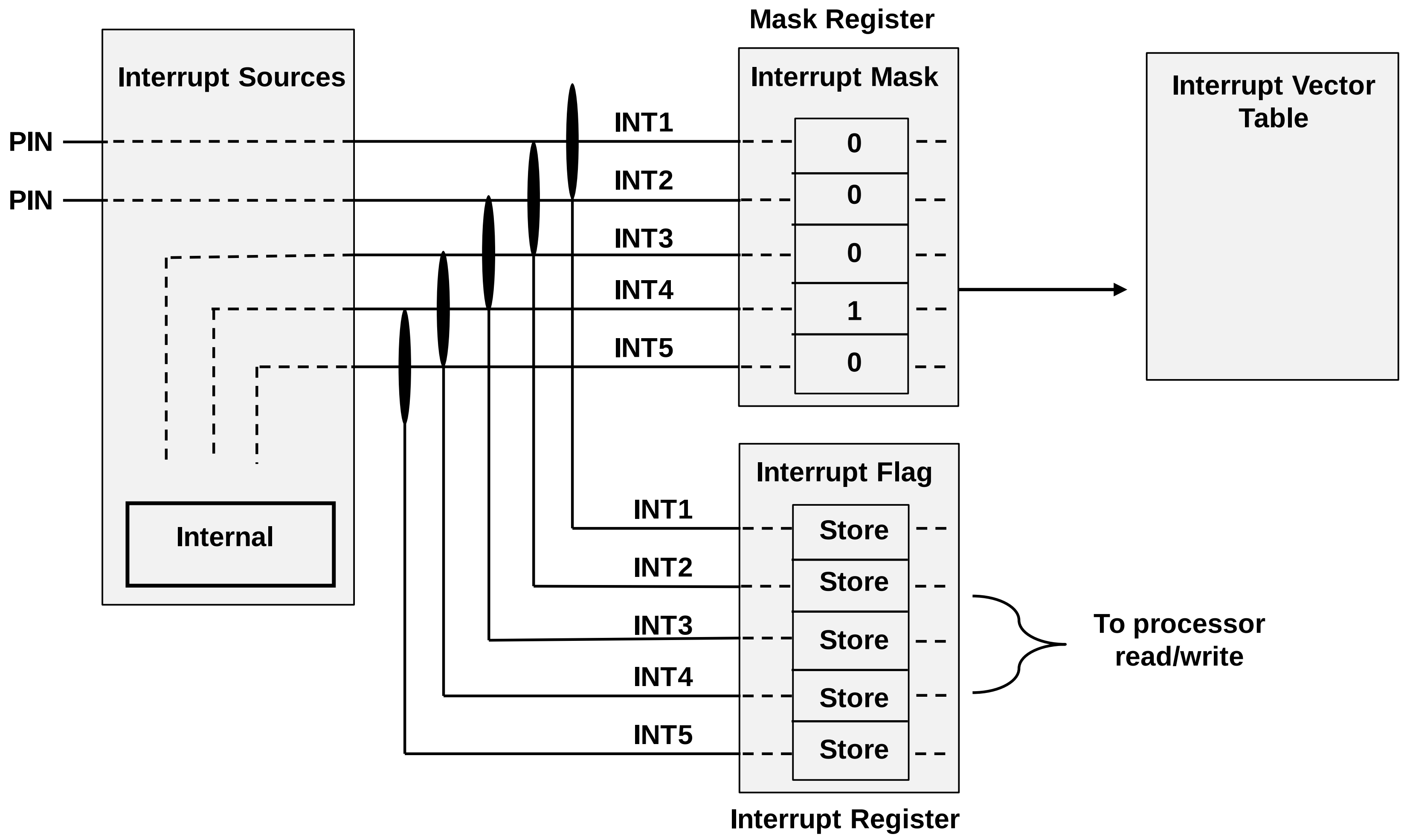
Interrupt-based programming style



Interrupts


Normal Instruction Execution
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
Instruction 7
Interrupt occurs while Instruction-4 is under execution
Instruction 4 execution is completed.
Transition control to ISR
On Interrupt Occurence
Interrupt Service Routine
Loads the address of the begining address of ISR into program counter.
Interrupt System is turned off.
ISR execution completes when RETI is encountered.
Execution cycle returs to Instruction 5
On ISR Completion

A Basic Interrupt System



A Basic Interrupt System

- Interrupt sources: Signals starting an interrupt
  - External pins, where level or a rising/falling edge of an input triggers interrupt
  - Internal peripheral interrupts; e.g., timer fired, ADC completed
- Interrupt flags
  - Each hardware interrupt source has associated interrupt flag
  - Whenever interrupt is triggered the corresponding interrupt flag is set (a bit within an interrupt register)
  - Processor can read from/write to interrupt flag register, reading from it to find out which interrupts occurred and writing to it to clear flags
  - Interrupt flag is usually reset by hardware upon starting ISR

A Basic Interrupt System

- Interrupt mask
  - Interrupt mask has a set of bits identical to interrupt flag register
  - Setting any bit (unmasking) lets corresponding signal source generate an interrupt - causing the processor to execute ISR
  - When a mask’s bit is clear (masked) ISR is not activated for that signal source but interrupt flag register is still set by signal source
    - This allows to detect hardware activities, e.g. by polling interrupt register
    - Interrupt flags of masked interrupts are not automatically reset, so always reset interrupt flag before unmasking
- Interrupt vector table
  - Interrupt vector is a location in memory with addresses of ISRs
  - Whenever an unmasked interrupt occurs program execution starts from address contained in interrupt vector
  - Lowest ROM addresses are by default defined as Interrupt Vectors
  - Interrupt vector table can also be located in boot section

AVR Interrupt Vector Table

Number	Address	Source	Definition
1	$0000	RESET	external pin, power-on reset, watchdog reset
2 - 9	$0002, $0004, …, $0010	INT0 – INT7	external interrupt request 0 ... 7
13	$0018	WDT	watchdog time-out interrupt
26	$0032	USART0 RX	USART0 Rx complete
27	$0034	USART0 UDRE	USART0 data register empty
28	$0036	USART0 TX	USART0 Tx complete
31	$003C	EE READY	EEPROM ready

- Atmega128RFA1 reserves 76 interrupts, the lower the address the higher is the priority level
- By default AVR-Interrupts do not cascade, they are delayed, i.e. the are executed after completion of the first ISR, nesting is possible (manually enable interrupts in ISR or naked)

General Setup of Interrupt Vector Addresses

Address Labels	Code	Comments
$0000	jmp RESET	; Reset Handler
$0002	jmp INT0	; IRQ0 Handler
$0004	jmp INT1	; IRQ1 Handler
$0018	jmp WDT	; Watchdog Timeout Handler
$0032	jmp USART0_RXC	; USART0 RX Complete Handler
$0034	jmp USART0_UDRE	; USART0 UDR Empty Handler
$0036	jmp USART0_TXC	; USART0 TX Complete Handler
$003C	jmp EE_RDY	; EEPROM Ready Handler

AVR addresses words (2 Bytes) in ROM, so jmp instruction requires 4 bytes (2^16 addresses each two bytes = 128 KB addressable), hence two addresses have distance 2

Interrupt Handling

- An interrupt leaves the machine in a well-defined state
  - Program Counter (PC) is saved in a known place
  - All instructions before the one pointed to by PC have fully executed
  - No instruction beyond the one pointed to by PC has been executed
  - Execution state (e.g. register values) of instruction pointed to by PC is known
- Types of external interrupts
  - Level-triggered
    - interrupt is generated whenever level of interrupt source is asserted indicated by a high (1), or low level (0), of interrupt request line
  - Edge-triggered
    - interrupt is generated when an asserting edge of interrupt source is asserted, either a falling (1 to 0) or a rising edge (0 to 1)

Types of Interrupts

- **Maskable interrupt**
  Hardware interrupt that may be ignored by setting a bit in interrupt mask register's (IMR) bit-mask
- **Non-maskable interrupt**
  Hardware interrupt that lacks an associated bit-mask, it can never be ignored
  - e.g. non-recoverable hardware errors (memory fault), reset, timers, especially watchdog timers
  - reserved for highly critical interrupts

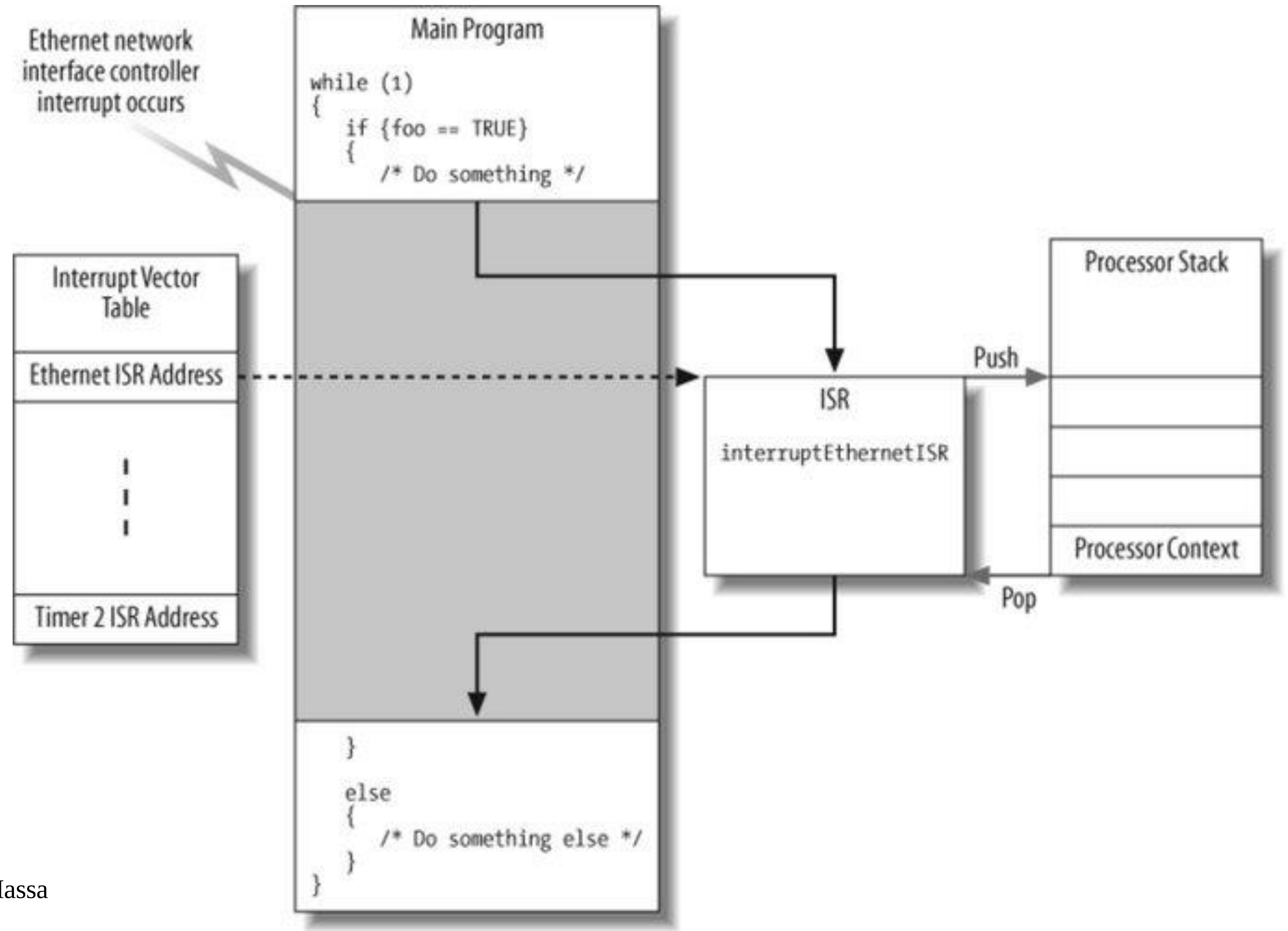
SEI/CLI - Global Interrupt Flag

- SEI sets Global Interrupt flag (I) in SREG
  - Instruction following SEI will be executed before any pending interrupts
  - Question: If I flag is set using another instruction, will this also wait one instruction? How can this be tested? Answer in lab!
- CLI clears Global Interrupt flag (I) in SREG
  - Interrupts will be immediately disabled
  - No interrupt will be executed after CLI instruction, even if it occurs simultaneously with CLI instruction

Multiple Interrupts at One Time

- If several interrupts are active at one time, a priority scheme is used to sort out the source to be serviced
- AVR
  - The lower the address, the higher the priority level
  - Delayed Interrupts AVR: interrupts occurring during an ISR execution are served after of completion of ISR (by priority)
- The processor
  - scans for all active interrupt sources (the highest priority wins)
  - disables all further interrupts by clearing the I bit in the status register
  - gets the address of the ISR for that source and calls that ISR
- After a RETI instruction (*return from interrupt instruction*) (C compiler includes this instruction at end of ISR), the I bit is set so other interrupts can be serviced

Interrupt Service Routine (ISR)



```
while (1)
{
   if {foo == TRUE}
   {
      /* Do something */
   }
   else
   {
      /* Do something else */
   }
}
```

Quelle: M. Barr & A. Massa

Interrupt Service Routine (ISR)

- ISRs must be installed for all interrupts, even if particular interrupts are not used (default ISR)
- ISR is responsible for
  - saving the processor's context
  - acknowledging the interrupt
  - restoring the processor context
- Compilers automatically generate code to save context when ISR is entered, and to restore context when ISR is exited
- This may be done in hard- or software

ISRs with GCC (for AVR)

- Two options
  - `void ISRname (void) __attribute__((signal,used));`
  - `void ISRname (void) __attribute__((interrupt,used));`
- AVR: `ISRname` is a pre-defined name of an ISR
  Examples:
  - `TIMER1_OVF_vect` – Timer1 overflow
  - `ADC_vect` – ADC conversion complete
- Library provides macros for ISR-definition and ISR name
  - `avr/interrupt.h: ISR(ISRname)`, uses `signal`
  - `avr/io.h:` names of ISRs
- `signal` and `interrupt`
  - For both, the compiler creates entry and exit sequences for an interrupt handler
  - ISR defined as `interrupt` (re-)enables interrupts on entry

GCC Keyword `__attribute__`

- Allows to specify special attributes when making a *declaration*
  Function definition cannot have `__attribute__` keyword.
- Keyword is followed by an attribute specification inside double parentheses
- Attribute `used`
  - Function is assumed to be used, even if not called explicitly (which is the case for ISRs)
- Attribute `naked`
  - indicates that specified function does not need entry and exit sequences, only basic `asm` statements can safely be included in naked functions

ISR Specifics

- ISRs should be as short and simple as possible, at most as long as shortest period of releasing events
- ISRs are difficult to debug
- When using interrupts, normal program code should be **reentrant**, i.e. satisfy the following conditions:
  - Variables used by ISR and main program (i.e. shared variables) must be used in an atomic way
  - It does not call non-reentrant functions
  - It does not use the hardware in a non-atomic way
    - Example: Some devices require two I/O operations to access it

3

Race Conditions

3. Race Conditions

- Volatile global variables allow sharing data between ISR and function main
- **Race condition:**
  Situation where outcome varies depending on precise order in which instructions of main code and ISR are executed
- Example:

```
gIndex--;
```

Generated Code:

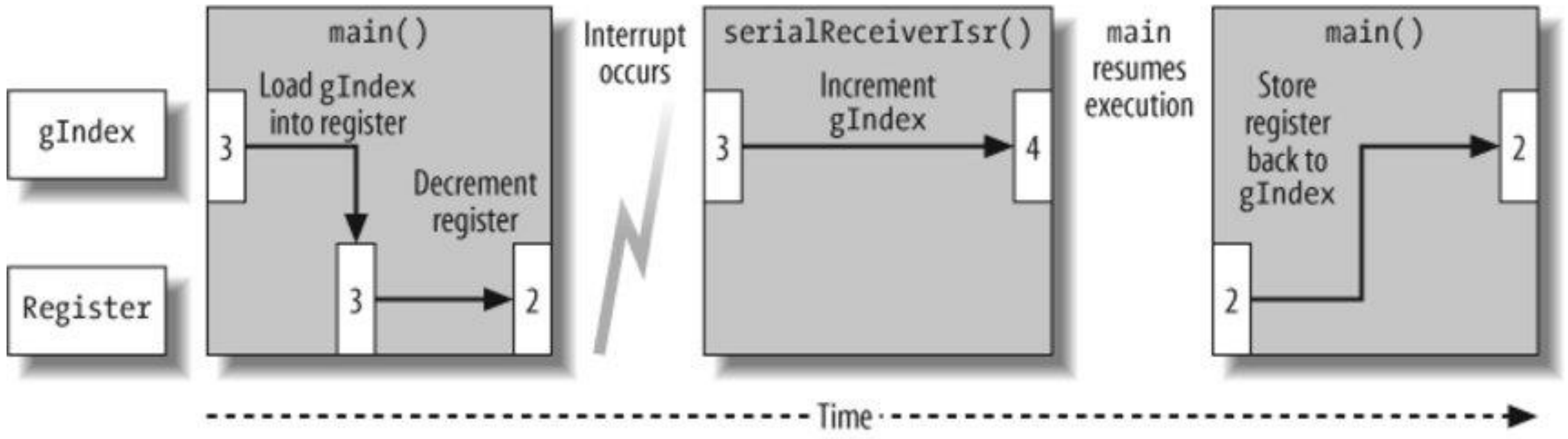
```
LOAD gIndex into a register;
DECREMENT register value;
STORE register value back into gIndex;
```

```
volatile int gIndex = 0;

interrupt void serialReceiveISR(void) {
   /* Store receive character in memory buffer */
   gIndex++;
}

int main(void) {
   while (1) {
      if (gIndex > 0) {
          /* reads character from memory buffer */
          gIndex--;
      }
   }
}
```

Race Conditions

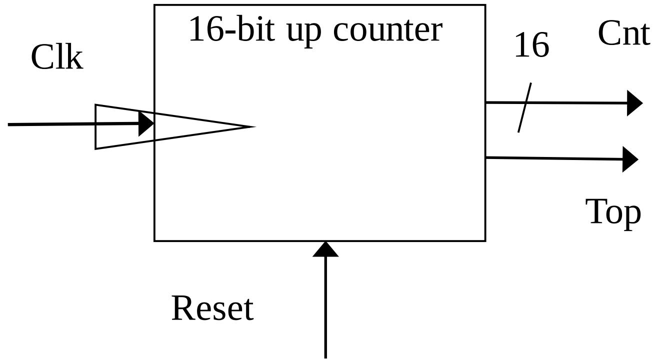


Problem: The decrement code is not executed atomically
Impact: Variable gIndex has wrong value

Critical Section:
Code that must be executed without interruptions

Example for Race Condition

- Timer implementation
  - Interrupt is triggered when 16 bit timer overflows
  - ISR `timer()` increments a global variable `timer_hi` (which maintains number of times hardware counted to 65536)
  - Function `timer_read` returns current "time" as 4 byte value

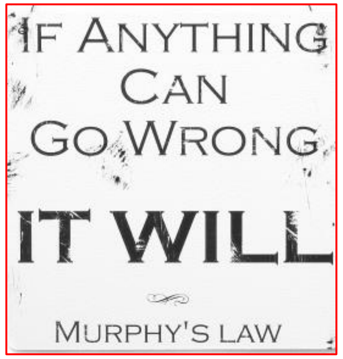


```
volatile uint16_t timer_hi = 0;

ISR timer() {
    ++timer_hi;
}
```

```
uint32_t timer_read() {
    uint16_t low, high;
    low = Cnt;
    high = timer_hi;
    return ((uint32_t) high << 16) + low;
}
```

Example



- Note `timer_read()` and `timer()` are executed asynchronously
- Possible race condition I
  - Initially `timer_hi` has value 0x0000
  - `timer_read()` reads value 0xFFFF from `Cnt`
  - Before reading `timer_hi`, hardware increments `Cnt` to 0x000
  - Overflow triggers interrupt, ISR `timer()` is run, `timer_hi` is now 0x0001
  - ISR returns, `timer_read()` continues, it concatenates the new value 0x0001 with previously read value 0xFFFF and returns 0x1FFFF, a hugely incorrect value
- Probability of this race condition is small but nonzero

Example

- Possible race condition II
  - Suppose `timer_read()` is called during a time when interrupts are disabled (e.g. if another ISR needs time)
  - At start time of `timer_read()` counter `Cnt` has value 0xFFFF and `timer_hi` has value 0x0000
  - Before anything happens `Cnt` is incremented to 0x0000
  - With interrupts off the pending interrupt gets deferred
  - `timer_read()` returns 0x00000 instead of the correct 0x10000, or the reasonable 0xFFFF

- Option 1:
  - Stop timer before attempting to read it (e.g. mask timer interrupt) and restart it at end of function `timer_read()`
  - No chance of overflow putting lower/upper halves out of sync
  - Problem: Time gets lost, gets worse if another interrupt occurs during time span with no counting
- Option 2:
  - Read variable `timer_hi`, then hardware timer `Cnt`, and then reread `timer_hi`, if values for `timer_hi` differ, interrupt has occurred
  - Iterate until two variable reads are equal
  - Problem: Function may loop several times, i.e., execution time is non-deterministic

- Option 3
  - Disable interrupts around the reads, i.e. at start of `timer_read()`
  - Prevents ISR from gaining control and changing `timer_hi` after `Cnt` was read
  - Problem:
    - Suppose `Cnt` is at 0xFFFF, and `timer_hi` is 0x0000
    - Before code does anything, overflow occurs but rollover is missed
    - Function returns 0x0000 instead of 0x1000, or even a reasonable 0xFFFF

- Solution
```
uint32_t timer_read() {
   uint16_t low, high;
   disable_interrupts;
   low = Cnt;
   high = timer_hi;
   if (Top) {
      ++high;
      low = Cnt;
   }
   enable_interrupts;
   return ((uint32_t) high << 16) + low;
}
```
- Question: What about `timer_hi`?

- Solution
  - Interrupts are off
  - New test to see if overflow has taken place
    - `Top` indicates whether an overflow occurred between turning interrupts off and reading low halve of time from device
  - If an overflow occurred:
    - `high` is incremented manually
    - `Cnt` is reread
  - Minor Problem
    - Latency is increased, but function is entirely deterministic

- Problem
  - If function `timer_read()` is called from a place where interrupts are disabled, it returns after turning them back on
  - Consequence: Context is changed in a very dangerous manner
- Solution
  - Save and restore context

Example

```
uint32_t timer_read()
{
    uint16_t low, high;
    save_interrupt_state;
    disable_interrupts;
    low = Cnt;
    high = timer_hi;
    if (Top) {
        ++high;
        low = Cnt;
    }
    restore_interrupt_state;
    return ((uint32_t) high << 16) + low;
}
```

Atomic Access

- Certain operations (e.g. writing variables larger than one byte) need to be done without interruption, i.e., *atomically*
- Option: Place operation between cli() and sei()
- Best AVR option: Store SREG in variable

  -
    ```
     uint8_t sreg = SREG;
    cli();                  /* Disable interrupts */
    /* … atomic code here */
    SREG = sreg;            /* Restore global interrupt flag */
    ```
  - Usable in case of nested cli/sei calls!
  - Example: Suppose in cli/sei section another cli/sei call appears, then after nested sei Interrupts are allowed, this is not intention of programmer

AVR

SREG ≡ AVR Status Register
Bit 7 – I: Global Interrupt Enable
TCNTn = Timer/Counter register

- Atomic read of the 16 bit TCNTn register

```
uint16_t TIM16_ReadTCNTn()
{
  uint16_t i;

  uint8_t sreg = SREG; /* Save global interrupt flag */

  cli();               /* Disable interrupts */

  i = TCNTn;           /* Read TCNTn into i */
  SREG = sreg;         /* Restore global interrupt flag */

  __asm__ __volatile__ ("" ::: "memory");
  return i;
}
```

AVR

- Atomic write of the 16 bit TCNTn register content

```
void TIM16_WriteTCNTn(uint16_t i)
{
  uint8_t sreg = SREG; /* Save global interrupt flag */

  cli();               /* Disable interrupts */

  TCNTn = i;           /* Set TCNTn to i */

  SREG = sreg;         /* Restore global interrupt flag */
  __asm__ __volatile__ ("" ::: "memory");
}
```

To implement atomic access to multi-byte objects, consider using the macros from <util/atomic.h>

Memory Barriers

- Optimizing compilers are free to reorder statements or make function calls inline if resulting *net effect* is the same
- As long as all volatile reads and writes are to same addresses and in same order and write the same values, program is correct
- Memory barriers ensure that all variables are flushed from registers to memory before statement, and then re-read after statement

  ```
  __asm__ __volatile__ ("" ::: "memory");
  ```
- Purpose: Ensure that there are no variables *cached* in registers, so that it is safe to change registers' content
- Effect: Operations issued prior to barrier are guaranteed to be performed before operations issued after barrier

4

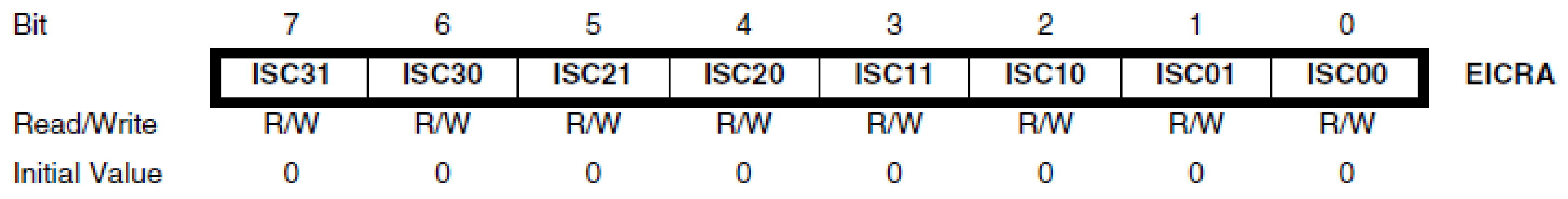
ATmega1281 Interrupts

ATmega1281 Interrupts

- Internal interrupts (Timers, ADC, UART, Reset)
- External interrupts via certain processor input pins
- General interrupt setup
  1. Configuration of interrupt
  2. Implementation of ISR
- Note: Status register is not automatically stored when entering ISR, nor restored when returning from an ISR. This must be handled by software!
  - But there is help!
- When ATmega exits from an interrupt, it will always return to main program and execute one more instruction before any pending interrupt is served

ATmega1281 External Interrupts

- External Interrupts
  - are triggered by INT7:0 pin
  - either by a falling/rising edge or a low level
  - Control registers: EICRA (INT3:0) and EICRB (INT7:4)
- EICRA contains control bits for interrupt sense control (ISC)



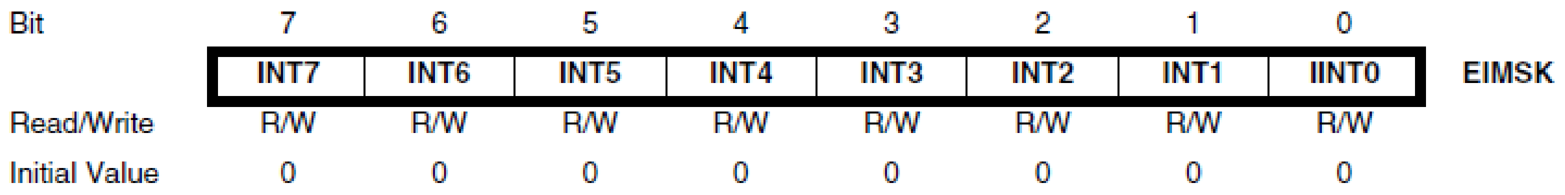
Bit	7	6	5	4	3	2	1	0	
	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

ISCn1	ISCn0	Description for INTn
0	0	Low level generates interrupt request
0	1	Any edge generates asynchronously interrupt request
1	0	Falling edge generates asynchronously interrupt request
1	1	Rising edge generates asynchronously interrupt request

ATmega1281 External Interrupts

- Procedure
  - Select type of interrupt by setting ICSn bits
  - Enable particular interrupt
  - Enable interrupts globally
- When changing ISCn bit, an interrupt can occur, therefore
  - First disable INTn by clearing its Interrupt Enable bit in EIMSK
  - Then change ISCn bit
  - Finally, clear INTn interrupt flag by writing a logical 1 (this causes flag to become 0!) to its Interrupt Flag bit (INTFn) in EIFR Register before interrupt is re-enabled (flag may be 1 from previous interrupt that was masked)

Interrupt Setup for External Interrupt 2



Bit	7	6	5	4	3	2	1	0	
	INT7	INT6	INT5	INT4	INT3	INT2	INT1	IINT0	EIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

```
/* interrupt setup for external interrupt 2 (INT2) */
EIMSK &= ~(1 << INT2);                       /* disable interrupt to prevent
                                                accidental call during setup */
EICRA |=  (1 << ISC21);                      /* trigger falling edges of signal */
EICRA &=  ~(1 << ISC20);
EIFR  |=  (1 << INTF2);                      /* clear pending interrupt (INT2) */
EIMSK |=  (1 << INT2);                       /* enable interrupt */

/* enable interrupts globally (from avr/interrupt.h) */
sei();
```

EIMSK ≡ External Interrupt Mask
EICRA/B ≡ External Interrupt Control Registers
EIFR ≡ External Interrupt Flag Register

Clearing Interrupt Flags

- Resetting a bit can be done with Read-modify-write cycle:
  1. read value into register
  2. AND with bitmask to clear a bit
  3. write value back to original location
- Problem: Not atomic, i.e. can be interrupted!!
- Interrupt registers have a special semantic
  - Writing a logical 0 to a particular bit does not change the bit
  - Writing a logical 1 to a particular bit sets it to 0
  - A bit cannot be set to 1 by software, only by hardware
- Writing a logical 1 with instruction OUT resets bit in atomic style
  - Seems illogical at first, since bit position already carries a logical 1 when reading it, but …
- Often interrupt handlers automatically clear interrupt flag bit during interrupt processing
  - by just calling the handler or by reading a particular hardware register that will normally happen anyway when processing the interrupt (e. g. USART)

ATmega1281 Interrupts - Handling

- AVR library offers ISR macro to take care of generic part of interrupt handling
- Nested interrupts are only possible, if interrupts are enabled inside ISR
- AVR lib has predefined names for all available ISRs
  - Interrupt vector names use the suffix `_vect`

```
#include <avr/interrupt.h>

ISR(XXX_vect) {
    // user code here
}
```

```
/* interrupt handler for external interrupt 2 */
ISR(INT2_vect) {
    /* state is saved automatically by the macro and
     * interrupts are disabled */

     user code here

    /* state is restored automatically by the macro,
     * interrupts are enabled */
}
```

- Examples:
  - `ADC_vect`: ADC conversion complete,
  - `INT2_vect`: External Interrupt Request 2

ATmega1281 Interrupts - Handling

- ISR macro saves registers and processor state of current program and prepares interrupt routine and disables all interrupts
  - clears interrupt flag in SREG
- At its end, it issues a RETI (return form interrupt), so that processor registers are restored
- If unexpected interrupt occurs (interrupt is enabled but no handler installed), then the default action is to reset device by jumping to reset vector

  Alternative:

```
#include <avr/interrupt.h>

ISR(BADISR_vect) {
    // user code here
}
```

Communication: ISR and main

- Data shared between ISR and main() must be both
  - global in scope and
  - volatile or accessed in atomic blocks
- This program will most likely freeze forever in while loop
  (depending on optimizer)

```
# include <avr / interrupt .h>
int myValue = 0;
ISR (SomeVector_vect) {
    myValue++;
}

int main (void) {
    setupInterrupts();
    while (myValue == 0) { /* wait for interrupt to change myValue */ }
    turnOnLED();
}
```

Summary

- For an ISR to be called, three conditions are required
  1. Global Interrupts Enable bit (I) must be set in register SREG. It defaults to being cleared on power up, so we need to set it
  2. The individual interrupt source's enable bit must be set. Each interrupt source has a separate interrupt enable bit in related peripheral's control registers
  3. Condition for interrupt must be met
- By default, on AVR devices, interrupts are themselves not interruptible
- Data shared between ISR and main program must be both volatile and global in scope in C
- Interrupt source flag is usually cleared when ISR executes
  - some interrupt flags are cleared when a particular register is read
  - Example: USART receive complete flag

5

Time-triggered Systems

Overview

- In event-triggered systems:
  - Interrupt events may get lost if they occur during an ISR execution
  - Simultaneous events add to system’s complexity and reduce ability to predict behavior of an event-triggered system
- Example: Hospital doctor must look after needs of seriously ill patients overnight, with support of nursing staff
  - Event-triggered solution:
    Doctor arranges for one of nursing staff to waken her, if there is a significant problem with a patient
  - Time-triggered solution:
    Doctor sets alarm clock to ring every hour. When alarm goes off, she will get up and check in turn each patient

Time-Triggered System

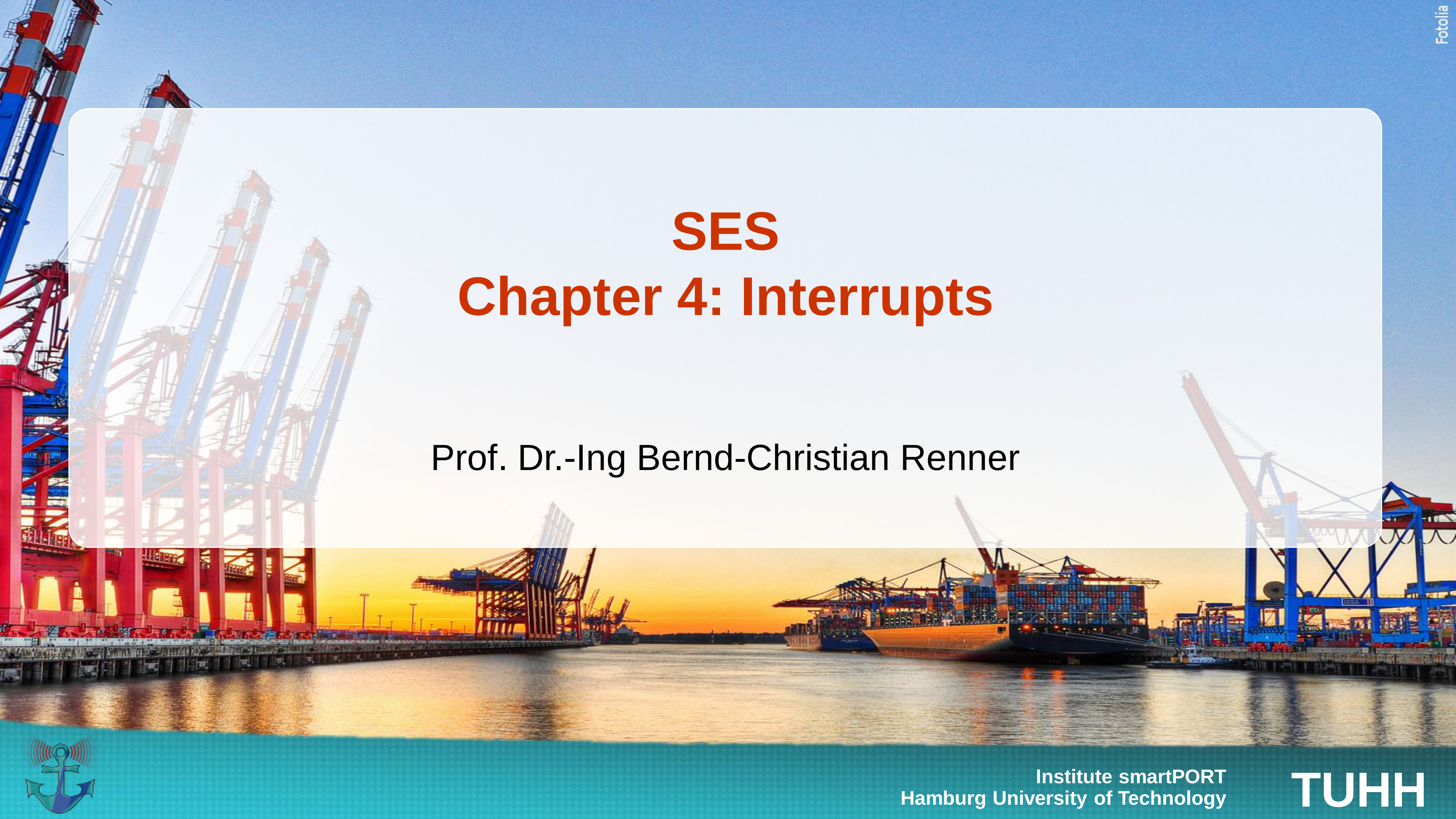
- Definition
  - all activities are carried out at certain points in time known a priori
  - no interrupts except by timer
  - schedule computed off-line → complex sophisticated algorithms are used
  - deterministic behavior at run-time
  - interaction with environment through polling
- Time-triggered techniques are widely used in automotive and aircraft industry
  - Reason: Very predictable behavior (useful in safety-related applications)

Properties

- In an entirely time-triggered system, the temporal control structure of all tasks is established a priori by off-line support-tools
- Task-Descriptor List (TDL) contains the cyclic schedule for all activities
- Schedule considers required precedence and mutual exclusion relationships among tasks such that an explicit coordination of tasks by operating system at run time is not necessary
- Dispatcher is activated by the synchronized clock tick
  - It reads TDL and performs the action that has been planned for this instant

Summary

- Interrupt:
  - Signal that stops the current program forcing it to execute another program immediately
- Benefit of hardware interrupt
  - Processor time used efficiently and not wasted polling input ports
- Event- vs. time-triggered systems
- Some processors provide a wake-from-sleep interrupt
  - This lets processor go into low power mode, where only interrupt hardware is active (useful if system is running on batteries)



SES
Chapter 4: Interrupts

Prof. Dr.-Ing Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology

TUHH