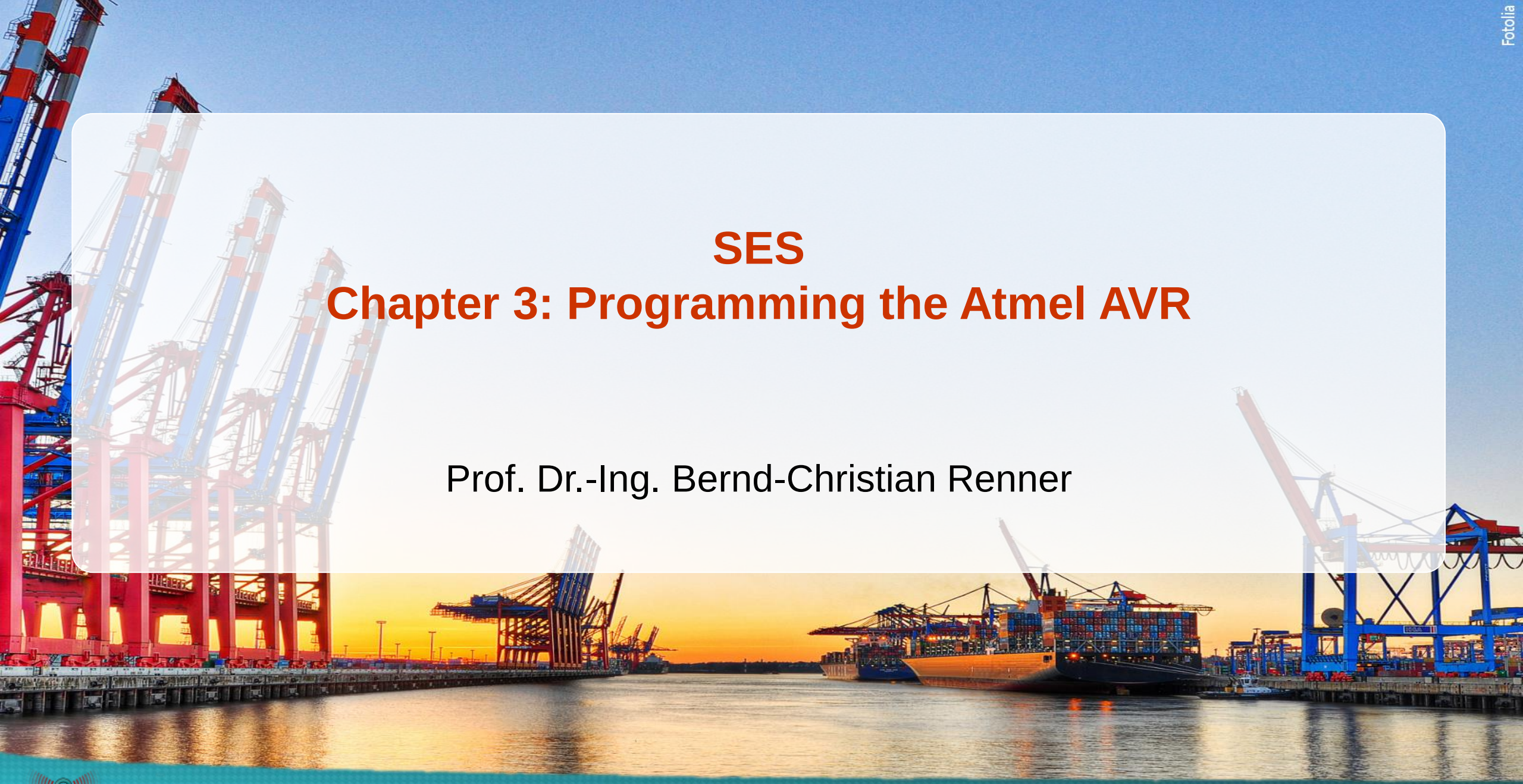


SES
Chapter 3: Programming the Atmel AVR

Prof. Dr.-Ing. Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology
TUHH

Contents

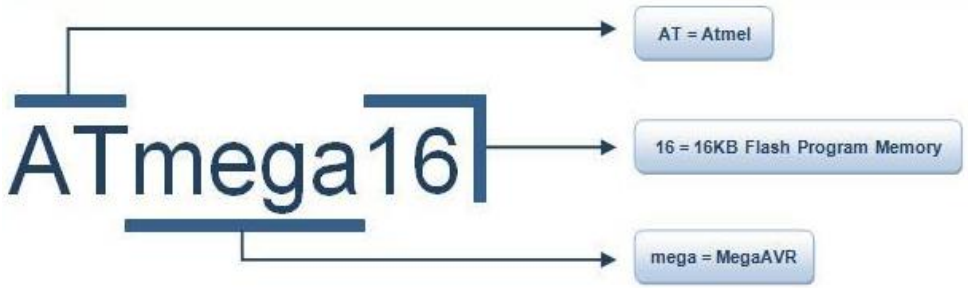
1

AVR Hardware

AVR Hardware

- AVR is a
  - modified Harvard architecture
  - low-power CMOS 8-bit RISC single chip microcontroller
  - developed by Atmel in 1996
  - ATMEL was acquired by Microchip in 2016
- Flash, EEPROM, and SRAM are all integrated on a single chip, removing the need for external memory
- Program instructions are stored in non-volatile flash memory
- Data address space consists of register file, I/O registers, SRAM

AVR microcontrollers

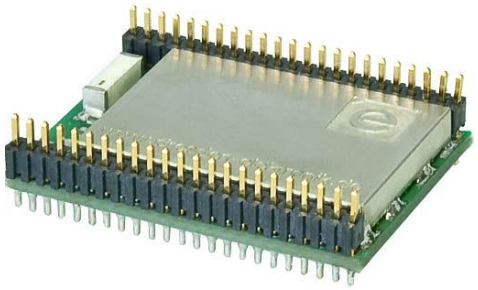


- TinyAVR
  - Less memory, small size
- MegaAVR
  - Very popular, memory upto 256 KB, higher number of inbuilt peripherals and suitable for moderate to complex applications
- XmegaAVR
  - Used commercially for complex applications, requiring large program memory and high speed

Series Name	Pins	Flash Memory	Special Feature
TinyAVR	6-32	0.5-8 KB	Small in size
MegaAVR	28-100	4-256KB	Extended peripherals
XmegaAVR	44-100	16-384KB	DMA, event system included

ATMEL ATmega128RFA1 Features

- SoC based on ATmega1281/AT86RF231 Transceiver
- Real Time Counter (RTC)
  to wake up controller from low power modes
- Six Timer/Counters with compare modes and PWM
- 2 USARTs
- Byte oriented 2-wire Serial Interface
- 8-channel, 10-bit ADC, up to 330 k Samples/s
- Programmable watchdog timer with internal oscillator
- SPI serial port
- IEEE std. 1149.1 compliant JTAG test interface, for accessing the On-chip Debug system and programming
- Six software selectable power saving modes

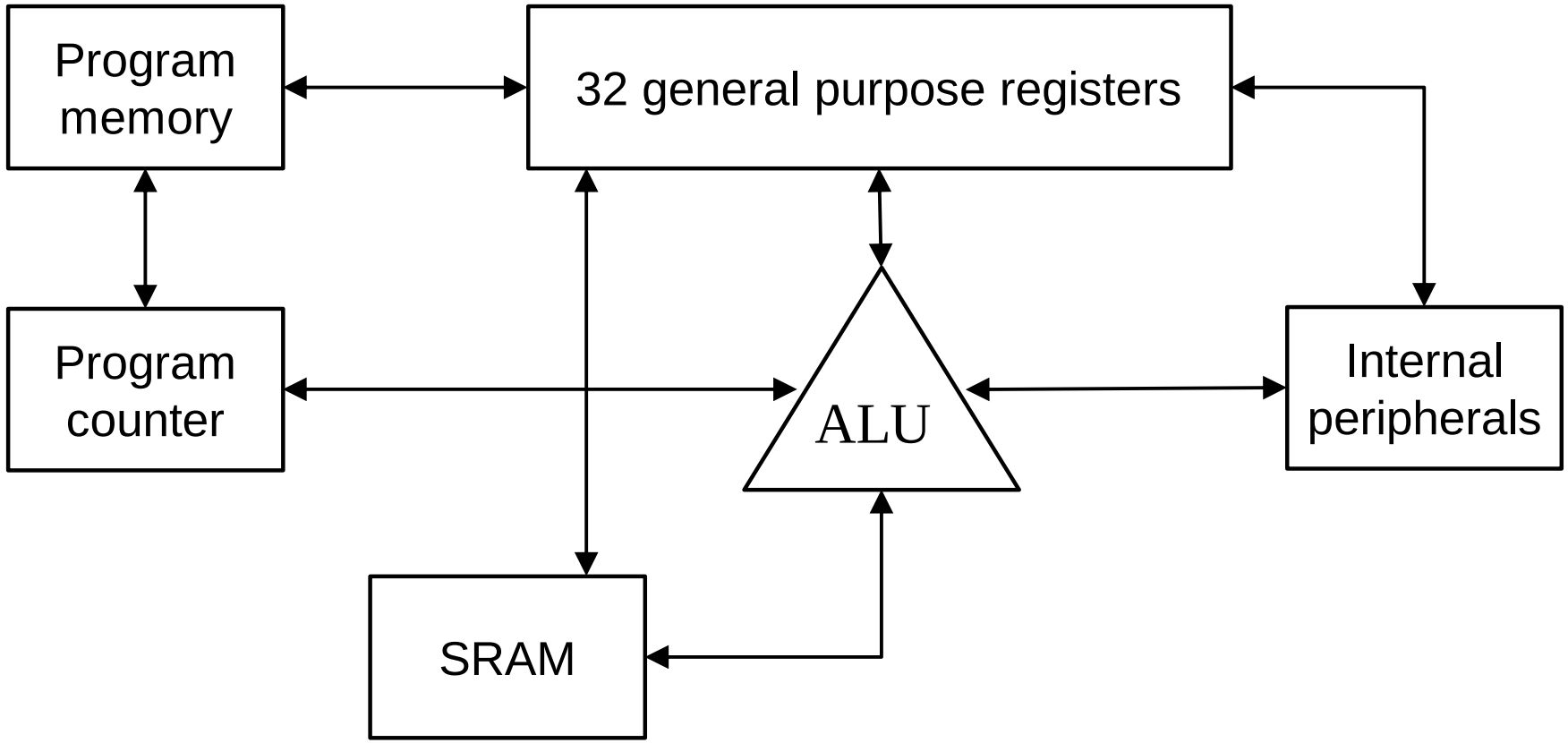


deRFmega128-22A00
Source: Dresden Elektronik

ATmega640/1280/1281/2560/2561 family

Device	Flash	EEPROM	RAM	General Purpose I/O pins	16 bits reso-lution PWM channels	Serial USARTs	ADC Channels
ATmega640	64KB	4KB	8KB	86	12	4	16
ATmega1280	128KB	4KB	8KB	86	12	4	16
ATmega1281	128KB	4KB	8KB	54	6	2	8
ATmega2560	256KB	4KB	8KB	86	12	4	16
ATmega2561	256KB	4KB	8KB	54	6	2	8

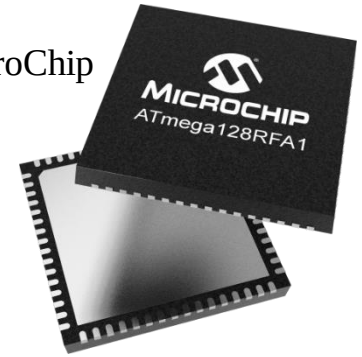
AVR ALU



Internal peripherals: function blocks such as UART (Universal asynchronous receiver and transmitter), Timers, SPI (Serial Peripheral Interface), EEPROM, etc.

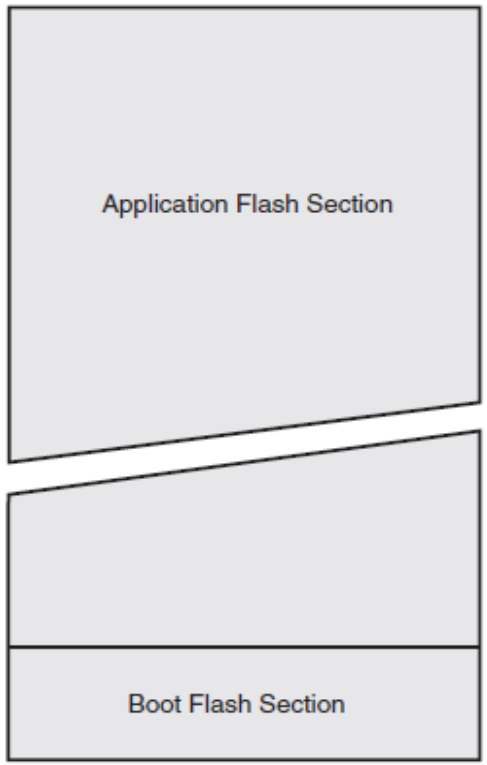
Memory sizes of ATmega128RFA1

Source: MicroChip



Flash	SRAM	EEPROM
128 kB	16 kB	4 kB

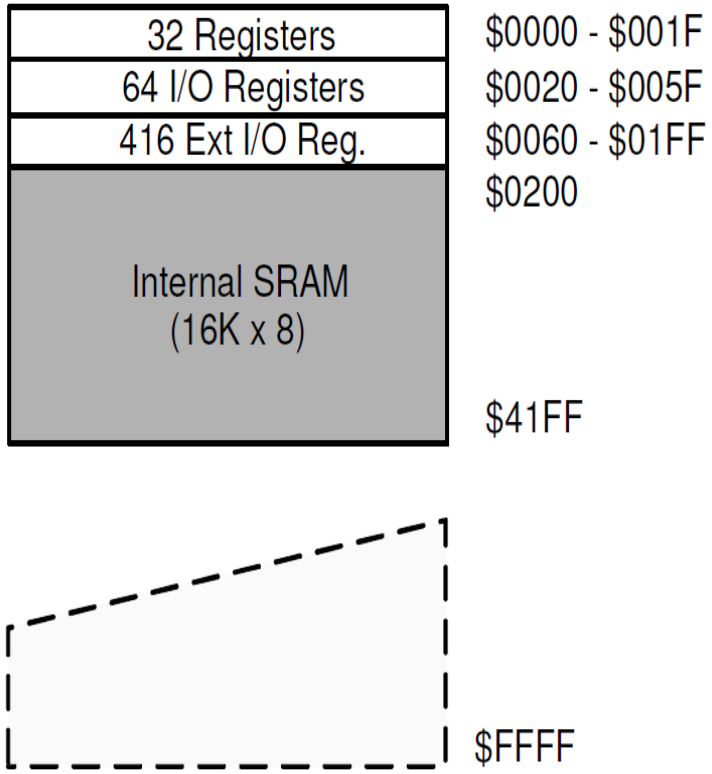
All addresses are 16 Bit



0x0000

0x7FFF/0xFFFF/0x1FFFF

Flash memory has an endurance of at least 10000 write/erase cycles



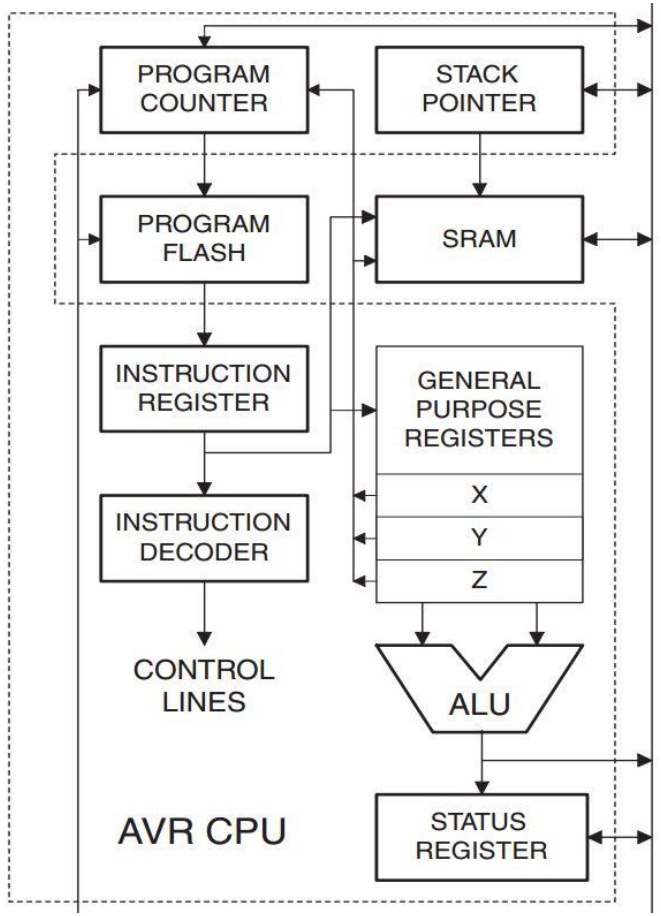
Registers

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

- Registers are special storages with 8 bits capacity
  - least significant bit starts with 0
- Special character of registers, compared to other storage sites, is that
  - they can be used directly in assembler commands
  - operations with content require only a single command word
  - they are connected directly to accumulator of CPU
  - they are source and target for calculations

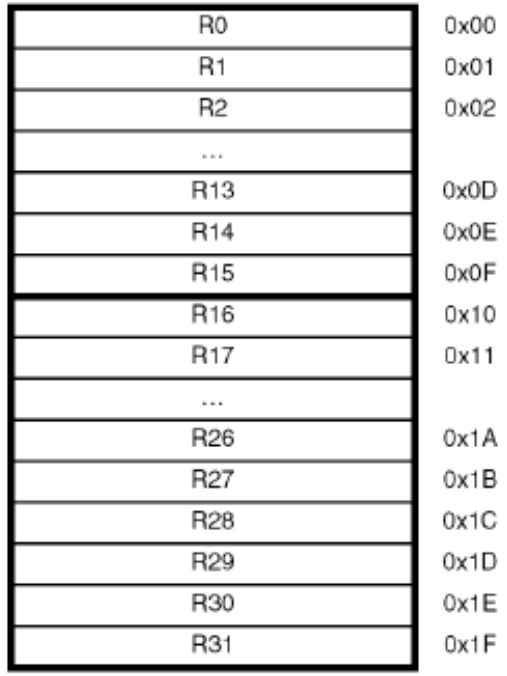
Registers

- SREG – AVR Status Register
  - Bit 7: Global Interrupt Enable
  - Bit 3: Two's Complement Overflow Flag
- Instruction Register
- General Purpose Register File
  - R0, R1, …, R31 with addresses 0x00 to 0x1F
  - Registers R26, ..., R31 have some added functions to their general purpose usage
    - Registers are 16-bit address pointers for indirect addressing of the data space called X, Y, and Z registers
    - Example:
      - R26 and R27 are the X register (low and high byte)
- Stack Pointers (16 Bit, must be set in Reset)
  - SPH – Stack Pointer High
  - SPL – Stack Pointer Low



AVR's General Purpose Registers

- 32 single-byte registers (R0 to R31) mapped into first 32 memory addresses (0000_{16}-$001F_{16}$) followed by the 64 I/O registers (0020_{16}-$005F_{16}$)
- Memory-mapped I/O registers occupy 0060_{16}-$01FF_{16}$
- Actual SRAM starts at 0200_{16}, ends at $41FF_{16}$
- Internal EEPROM memory
  - Not mapped into the micro controller's addressable memory space
  - Can only be accessed the same way an external peripheral device is, using special pointer registers and read/write instructions making EEPROM access much slower than other internal RAM



Registers used by C Compiler GCC

- Call-used registers (R18-R27, R30-R31)
  - May be allocated for local data, can be freely used in assembler subroutines
  - Calling C subroutines can clobber any of them, caller is responsible for saving and restoring
- Call-saved registers (R2-R17, R28-R29)
  - May be allocated for local data
  - Calling C subroutines leaves them unchanged. Assembler subroutines are responsible for saving and restoring these registers. R28:R29 (Y pointer) is used as a frame pointer (points to local data on stack)
- Fixed registers (R0, R1)
  - Never allocated by gcc for local data (R1 is assumed to be always zero in any C code)

AVR's general purpose registers

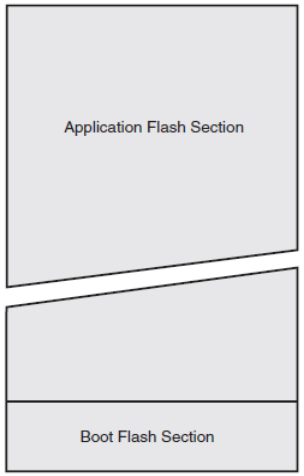
- C-macros defined in <avr/io.h> allow easy access to registers in C-programs
- Examples (simplified):
  - `#define SREG (*(volatile uint8_t *)(0x3F))`
  - `#define TCNT1 (*(volatile uint16_t *)(0x84))`

Working with registers

- Loading a value immediately to register r16,…,r31 (1 cycle)
  - `ldi r28,0xFF  ;load 255 in r28`
- Load an I/O location to a register (1 cycle)
  - `in r16,PIND  ;load PIND in r16`
- Store register in I/O location (1 cycle)
  - `out 0x3e,r29  ;store r29 in 0x3e`
- Exclusive-OR operation (1 cycle) (16 bit instruction)
  - `eor r3,r1  ;store exclusive-OR of r3 and r1 in r3`
- Load direct from RAM (2 cycles) (32 bit instruction)
  - `lds r24,0x0200  ;load byte at 0x0200 into r24`
- Call to subroutine (4 cycles)
  - `call 0x148  ;call subroutine starting at 0x148`

Flash memory

- Program Flash memory space is organized in words of 2 bytes and divided in two sections
  - Boot Program section
  - Application Program section
- Program Flash allows the program memory to be reprogrammed in-system through
  - SPI serial interface, by a nonvolatile memory programmer, or
  - by an on-chip boot loader
- Boot loader can use any interface to download application program into application Flash memory

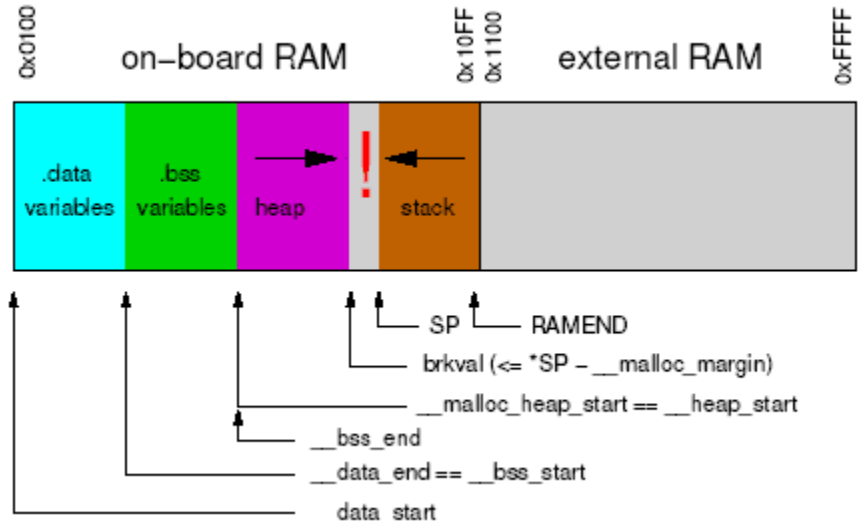


RAM

- avr-gcc arranges data in sections

  .data: static data defined in code
  .bss: uninitialized global or static variables
  .text: actual machine instructions which make up the program
  .eeprom: eeprom variables
  .bootloader: bootloader code

	sections
Flash	.text + .bootloader + .data
SRAM	.data + .bss
EEPROM	.eeprom



- Further sections: .initN, .finiN

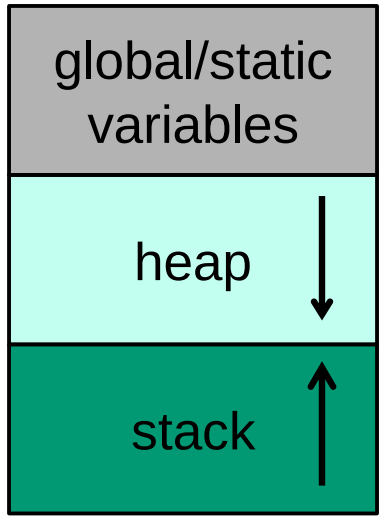
Quiz


```
int z = 11;

int foo(void)
{
    int x, y = 13;
    // …
    x = y + 12 + z;
    // …
    return x;
}
```

The Stack

- Stack: Area of RAM where a program stores temporary data during code execution
- Stack Pointer Register (SPL and SPH) always points to top of stack
- Life span of variables on stack is limited to duration of function
- As soon as function returns, used stack memory is freed
- Types of data stored on stack
  - local variables
  - return addresses (functions, interrupts)
  - function arguments
  - interrupt contexts
- Stack usually grows from higher to lower memory locations



The Stack

- Instructions that use stack pointer

Instruction	Stack Pointer	Description
PUSH	Decremented by 1	Data is pushed onto stack
CALL ICALL	Decremented by 2	Return address is pushed onto stack with subroutine call or interrupt
POP	Incremented by 1	Data is popped from stack
RET RETI	Incremented by 2	Return address is popped from stack with return from subroutine or return from interrupt

The Stack

- If memory area allocated for stack isn't large enough, executing code writes to area allocated below stack
  - Consequence: overflow situation

  - Underestimating stack usage can lead to serious runtime errors (overwritten variables, wild pointers, corrupted return addresses)
  - Overestimating stack usage wastes memory resources
- Experimentally computing stack size:
  - Fill entire amount of memory allocated to stack area with a dedicated constant fill value (e.g. 0xCD)
  - Run program
  - Search stack upwards until a value that is not 0xCD is found (end of used stack)
    - if dedicated value cannot be found, stack has most likely overflowed

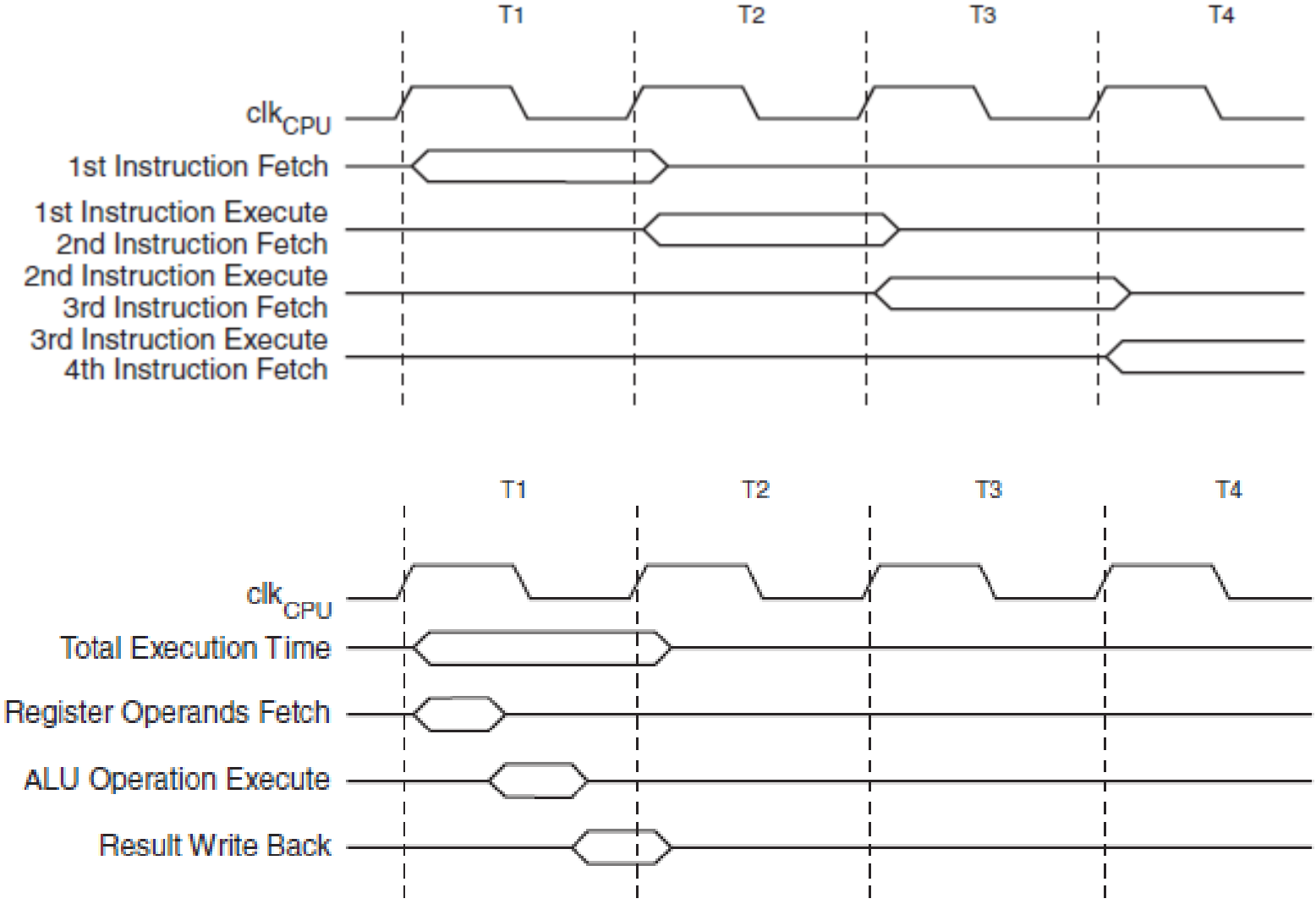
2

Program Execution

Program Execution

- AVR instructions have 16 bit/32 bit format (depending on addressing mode)
- AVRs have a two stage, single level pipeline design
  - Next machine instruction is fetched as current one is executing
  - Instruction being fetched cannot start execution until the one currently being executed is finished
  - Most instructions take just one or two clock cycles
- AVR supports clock speeds up to 20 MHz

Single Cycle ALU Operation - ATmega128



Source: ATMEL

Execution

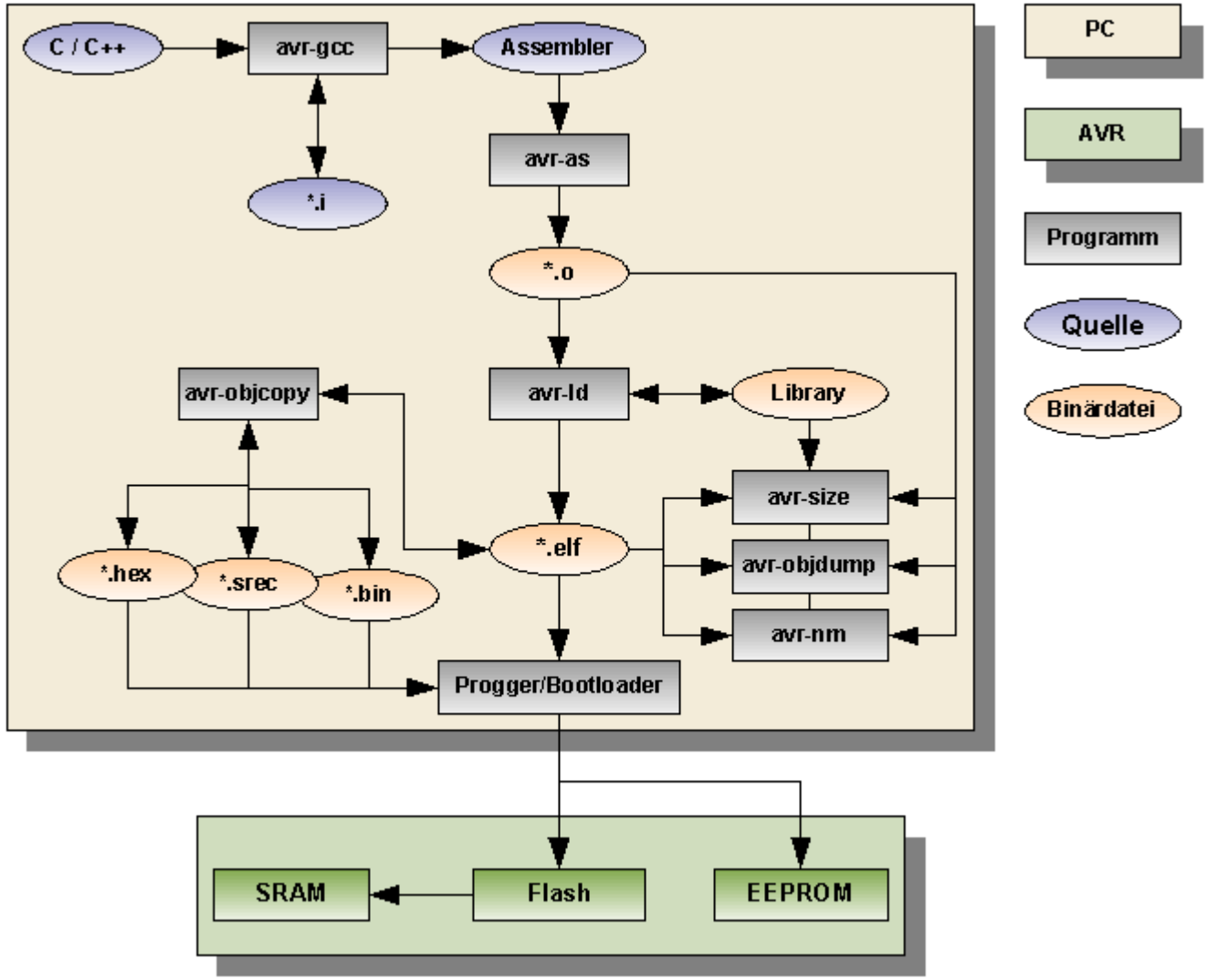
- When power supply rises and processor starts its work, hardware triggers a reset sequence
  - Registers and ports are set to default values
  - Program counter is set to zero (code address $0000), where execution always starts
  - Code is read word wise and executed
- This also happens
  - during an external reset on reset pin and
  - if watchdog counter reaches its maximum count
- Subsections of .text
  - .initN: startup code from reset up through start of main()
  - .finiN: exit code executed after return from main() or call to exit()

GNU Toolchain for Atmel AVR

- Compiler
  - GNU Compiler Collection (or GCC) to compile C or C++
- Assembler, Linker, Librarian, …
  - GNU Binutils is a collection of binary utilities
    - avr-as: The Assembler
    - avr-ld: The Linker
    - avr-ar: Create, modify, and extract from archives (libraries)
    - avr-nm: List symbols from object files
    - many more
- C Library
  - avr-libc is the Standard C Library for AVR 8-bit GCC. It contains many standard and non-standard C routines that are specific and useful for AVR 8-bit MCUs
- Source
  - http://www.atmel.com/tools/atmelavrtoolchainforwindows.aspx

AVR Software Development Process

Source: https://rn-wissen.de/wiki/index.php/Avr-gcc/Interna



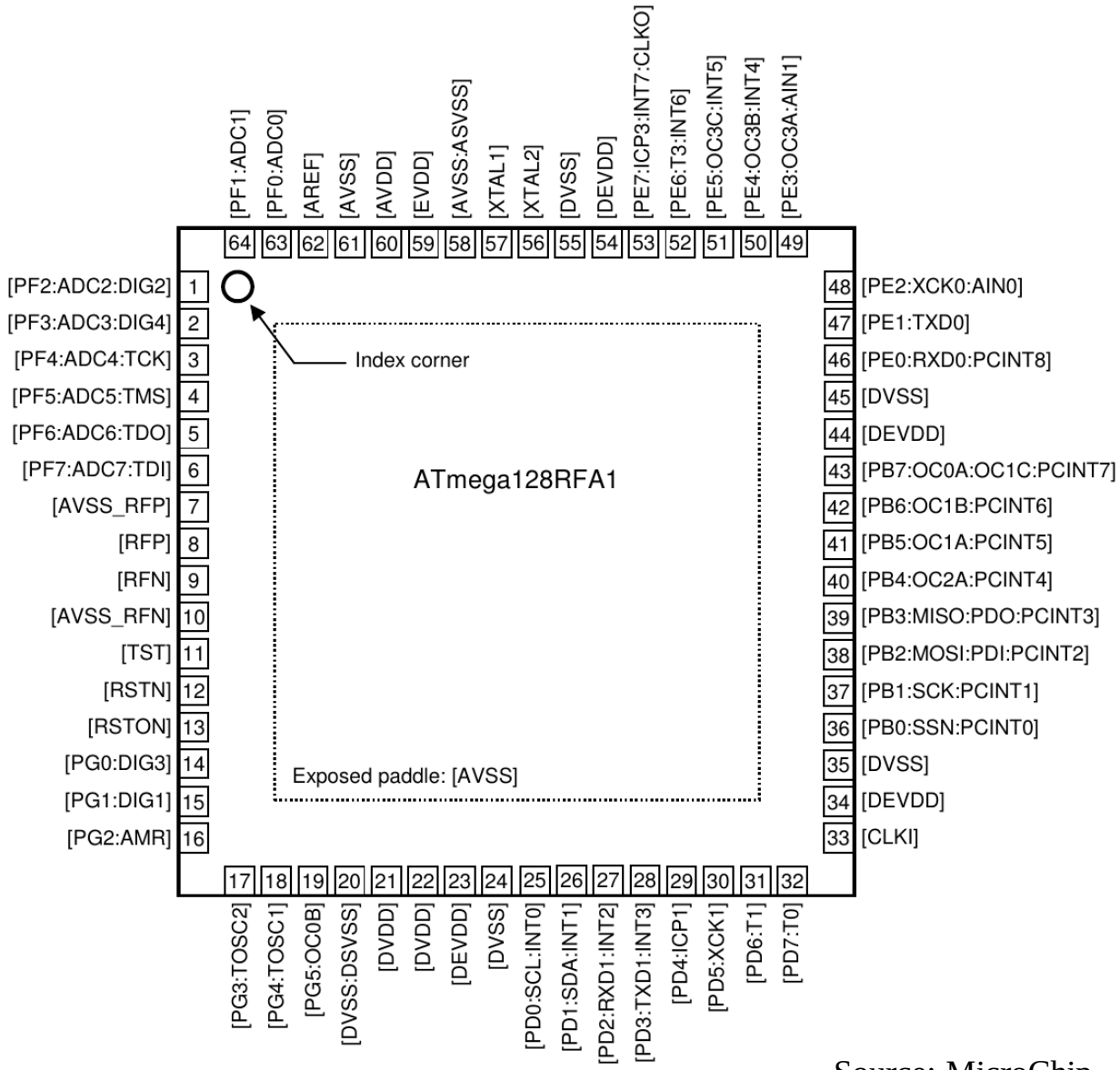
3

Registers and I/O Ports of AVR

AVR I/O Ports

- I/O Port:
  - Interface between CPU and internal and external hard- and software components
- CPU communicates with these components, reads from them or writes to them

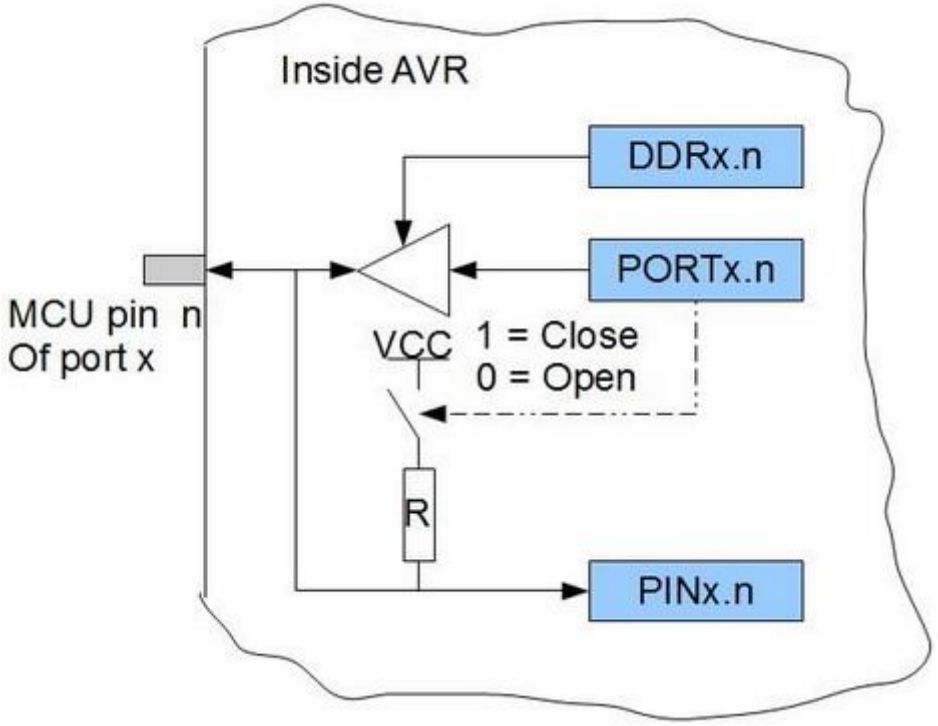
Figure 1-1. Pinout ATmega128RFA1



Source: MicroChip

Digital AVR I/O Pins and Ports

- Ports contain 8 pins (why?)
- Port addresses
  - are independent from type of AVR
  - have convenient aliases defined in provided header files for different AVR
- Naming
  - x is the name of the port {A,B,…,L}
  - n is the corresponding bit {0,1,…7}
- Each port bin corresponds to three bits (in the registers DDR, PORT, and PIN)



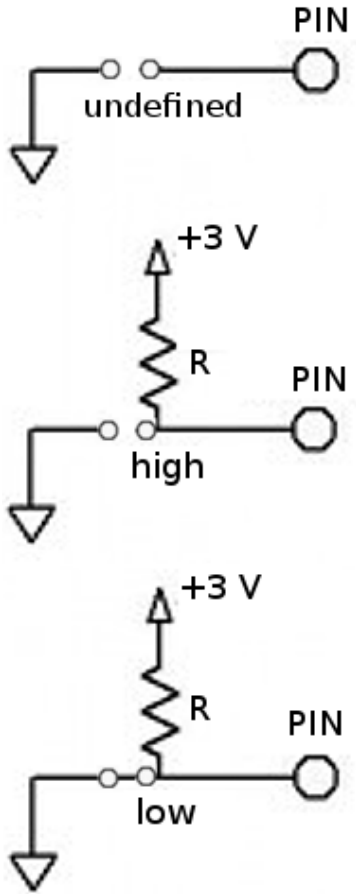
Digital AVR I/O Pins and Ports (II)

- AVR I/O ports have true Read-Modify-Write functionality when used as general digital I/O ports
  - direction of one port pin can be changed independently of others
- Three I/O memory address locations are allocated for each port
  - Data Direction Register DDRx (with bits DDxn) for Portx
    - Selects direction of this pin: 1 for output pin, 0 for input pin
  - Port Input Pins – PINx (read only, with bits PINxn) for Portx
    - Independent of the setting of Data Direction bit DDxn, port pin can be read through PINxn Register bit. 1 if pin "high", 0 if pin "low“
  - Data Register PORTx (with bits Pxn) for Portx
    - Used to access the output pins of Portx. Pins, which are declared input pins (via DDRx) can activate/deactivate pull-up resistors
- Note
  - x is the name of the port {A,B,…,L}
  - n is the corresponding bit {0,1,…7}

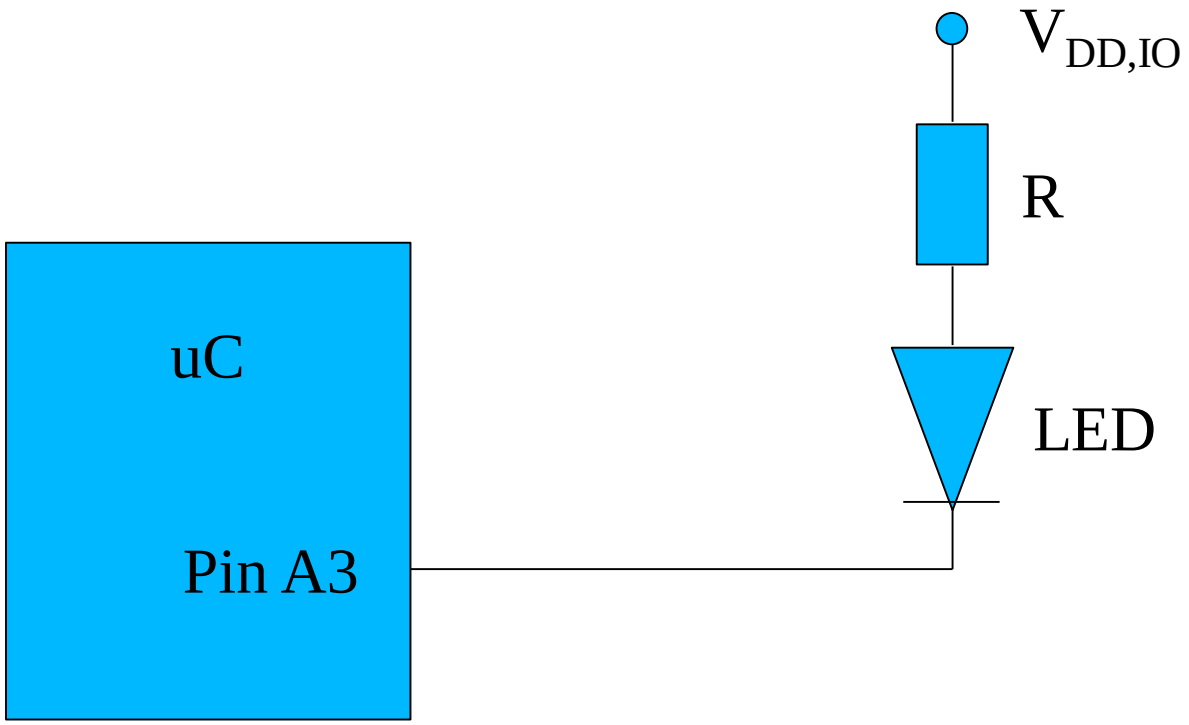
	DDR bit = 1	DDR bit = 0
Port bit = 1	high	pull-up
Port bit = 0	low	floating

Tri-State Logic

- If pin is configured as output pin and PORTxn is written a logic
  - 1 then port pin is driven high (high, $V_{DD,IO}$ voltage)
  - 0 then port pin is driven low (low, 0 V voltage)
- If pin is configured as input pin and PORTxn is written
  - logic 1 then pull-up resistor is activated
  - logic 0 then pull-up resistor is switched off
  - high-impedance for port pin (Hi-Z) if no active input, effectively removing device's influence from rest of circuit
- Pull-ups
  - ensure that inputs to logic systems settle at expected logic levels if external devices (e.g. a button) are disconnected
  - weakly "pull" voltage of wire if it is connected towards its voltage source level when components on the line are inactive



Quiz



AVR Registers and I/O Ports

- Most registers are read/write registers
- To access registers by name include `avr/io.h`
- This file includes a processor specific file
- Reading from or writing to registers is as simple as with variables, always set direction first
- Example: Writing to a register

```
#include <avr/io.h>
...
int main(void) {
   DDRA = 0xFF;
   // pins PA0 to PA7 of PORTA are output pins
   // Better to use logical names
   DDRA = (1 << DDA0) | (1 << DDA1) | (1 << DDA2) | …);
```

Example 1: Reading a Register

```
#include <avr/io.h>
#include <stdint.h>
...
uint8_t foo;
int main(void) {
    foo = PINB;
    // copies status of input pins on PORTB to foo
}
```

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	210
0x08 (0x28)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	29
0x07 (0x27)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	29
0x06 (0x26)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	29
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	209
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	209
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	209
0x02 (0x22)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	28
0x01 (0x21)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	28
0x00 (0x20)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	28

Example 2: LED

Useful tool: https://godbolt.org/
Shows generated code

```
#include <avr/io.h>
#include <util/delay.h>

/** Toggles the red LED of the SES-board */
/** Red LED connected to 2nd pin of port G */
int main (void) {
   DDRG |= (1 << DDG2);     /* Make output pin */
   while (1) {
      _delay_ms(1000);
      PORTG ^= (1 << PG2);  /* Toggle value */
   }
   return 0;
}
```

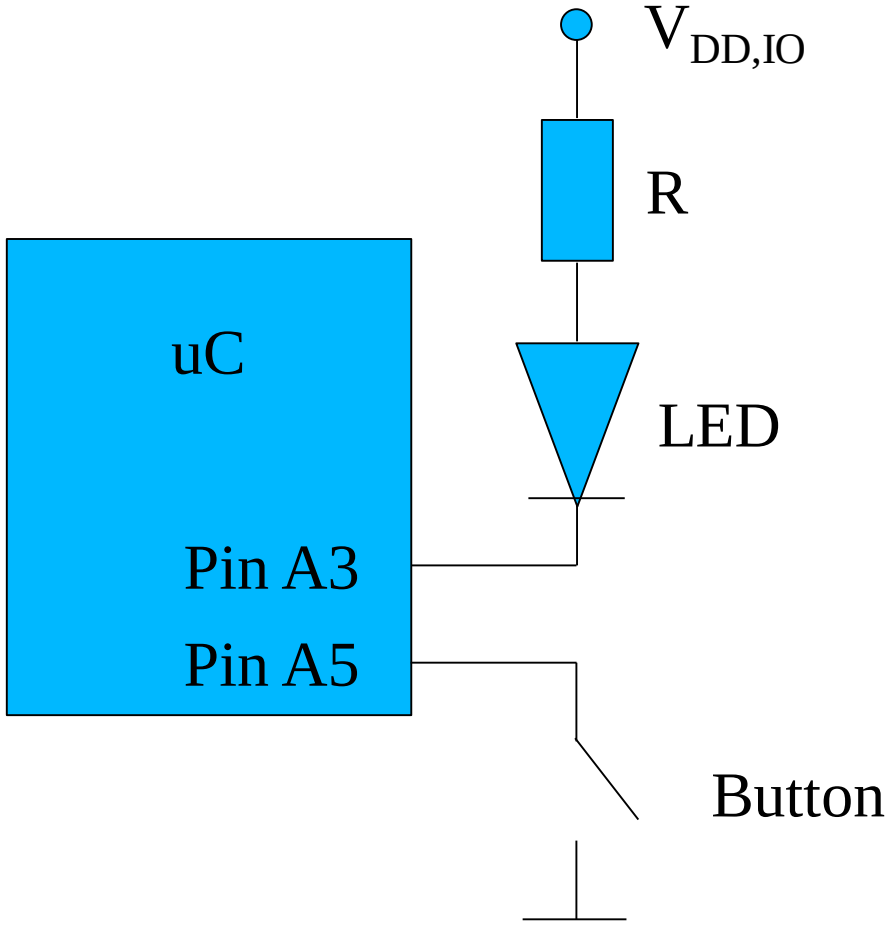
Example 3: Mixed Setup

```
#include <avr/io.h>
uint8_t i;
    ...
/* Define pull-ups and set outputs high */
/* Define directions for port pins */
DDRB = (1<<DDB3)|(1<<DDB2)|(1<<DDB1)|(1<<DDB0);
PORTB = (1<<PB7)|(1<<PB6)|(1<<PB1)|(1<<PB0);

/* Insert nop for synchronization*/
__no_operation();
/* Read port pins */
i = PINB;
```

Pins 0,1,2, and 3 are output pins of Port B
Pins 4,5,6, and 7 are input pins of Port B
Pins 0 and 1 are set high, Pins 6 and 7 have activated pull-ups
Resulting pin values are read back again, a *nop* instruction is included to be able to read back the value recently assigned to pins

Quiz




```
...
// init LED
DDRA  |= (1 << DA3);

// init button
DDRA  &= ~(1 << DA5);
PORTA |= (1 << PA5);

while (1) {
  if (PORTA & (1 << PA5)) {
    PORTA = (1 << PA3);
  } else {
    PORTA = (0 << PA3);
  }
}
```

Registers and I/O Ports of AVR: Overview I

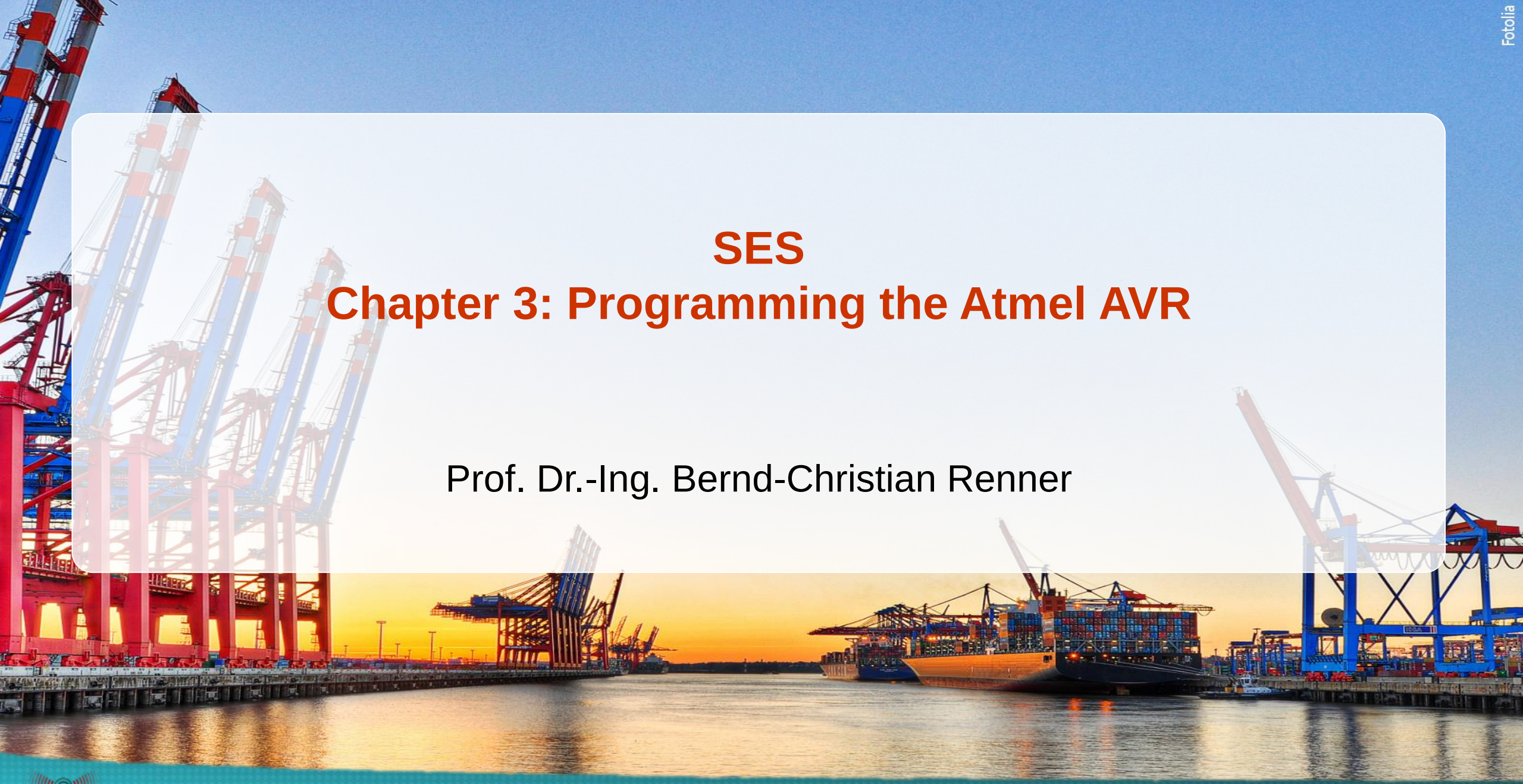
Accumulator	SREG	Status Register	SREG
Stack	SPL/SPH	Stackpointer	SPL/SPH
Ext.SRAM/ Ext.Interrupt	MCUCR	MCU General Control Register	MCUCR
Ext.Int.	INT	Interrupt Mask Register	EIMSK
		Flag Register	EIFR
Timer Interrupts	Timer Int	Timer Int Mask Register	TIMSK
		Timer Interrupt Flag Register	TIFR
8 Bit Timer 0	Timer 0	Timer/Counter 0 Control Register	TCCR0
		Timer/Counter 0	TCNT0
16 Bit Timer 1	Timer 1	Timer/Counter Control Register 1 A	TCCR1A
		Timer/Counter Control Register 1 B	TCCR1B
		Timer/Counter 1	TCNT1
		Output Compare Register 1 A	OCR1A
		Output Compare Register 1 B	OCR1B
		Input Capture Register	ICR1L/H
Watchdog Timer	WDT	Watchdog Timer Control Register	WDTCR

Registers and I/O Ports of AVR : Overview II

EEPROM	EEPROM	EEPROM Adress Register	EEAR
		EEPROM Data Register	EEDR
		EEPROM Control Register	EECR
SPI	SPI	Serial Peripheral Control Register	SPCR
		Serial Peripheral Status Register	SPSR
		Serial Peripheral Data Register	SPDR
UART	UART	UART Data Register	UDR
		UART Status Register	USR
		UART Control Register	UCR
		UART Baud Rate Register	UBRR
Analog Comparator	ANALOG	Analog Comparator Control and Status Register	ACSR
I/O Ports	I/O Ports	Port Output Register	PORTx
		Port Direction Register	DDRx
		Port Input Register	PINx

Summary

- ATmega1281
  - Low-power CMOS 8-bit microcontroller
  - based on the AVR enhanced RISC architecture
- AVR instructions have 16-bit/32-bit word format
- AVR ports have true Read-Modify-Write functionality when used as general digital I/O ports



SES
Chapter 3: Programming the Atmel AVR

Prof. Dr.-Ing. Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology
TUHH