System-Level Programming

17 μC-System Architecture – Peripherals

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

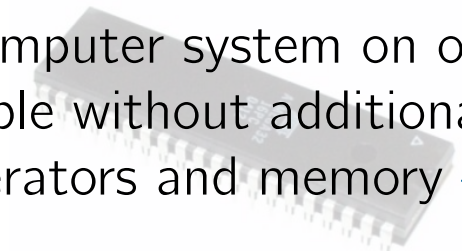
Lehrstuhl für Informatik 4
Systemsoftware

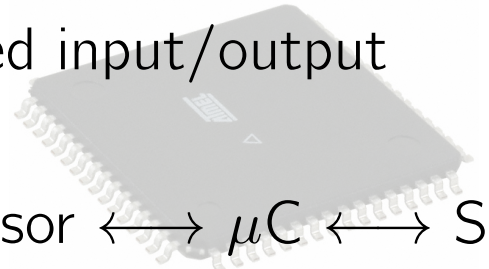
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2024

http://sys.cs.fau.de/lehre/ss24

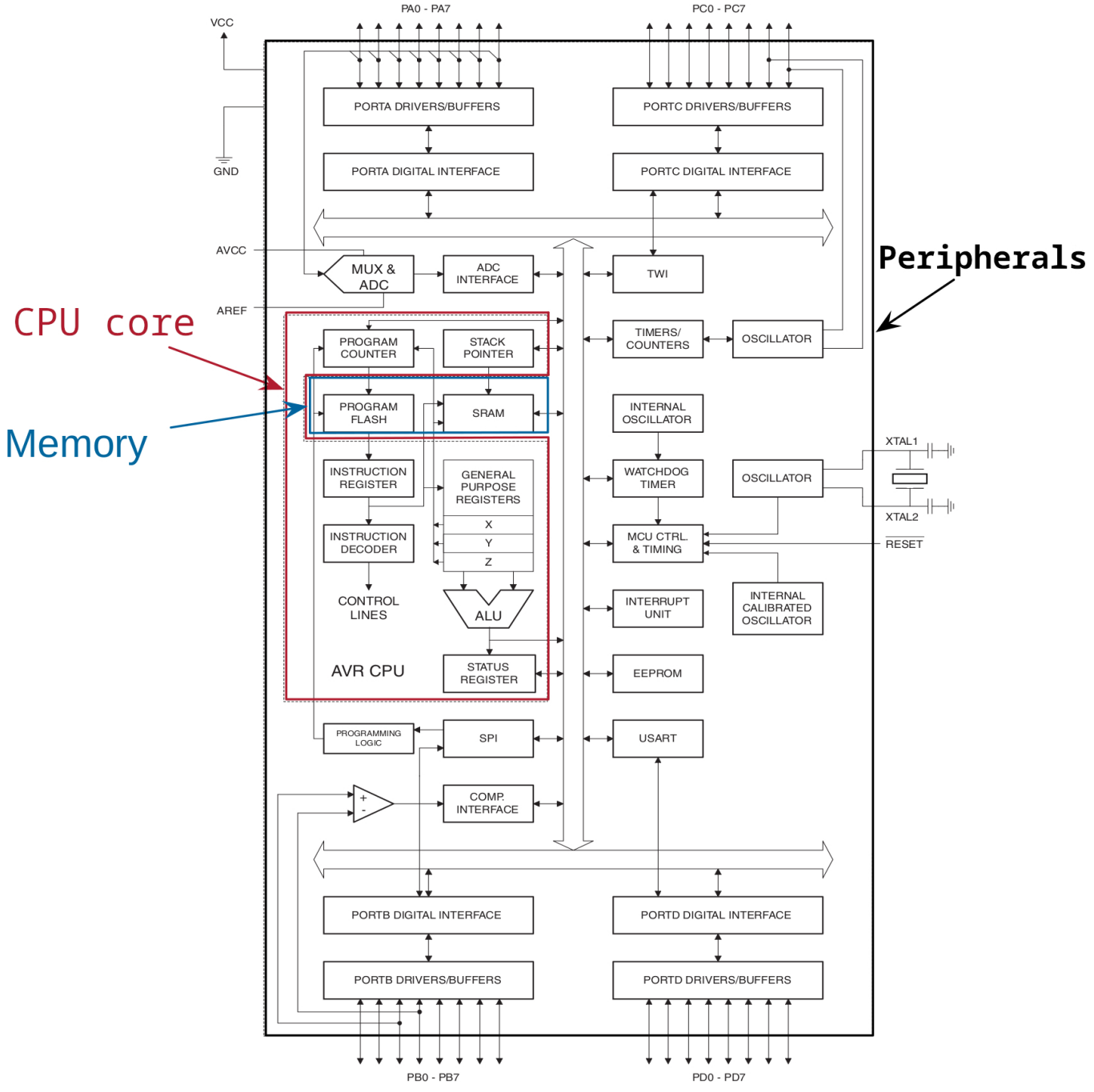
What is a μController?





- **μController** := processor + memory + **peripherals**
  - In fact, computer system on one chip $\longrightarrow$ *System-on-a-Chip* (SoC)
  - Often usable without additional external components, like e. g., clock generators and memory $\rightsquigarrow$ cost-efficient system design
- Main features are (plentiful) contained input/output components (peripherals)
- The distinctions are not fixed: processor $\longleftrightarrow$ μC $\longleftrightarrow$ SoC
  - AMD64 CPUs also have included timers, memory (cache), ...
  - Some μC reach the speed of "big processors"

Example ATmega32: Block Circuit Diagram



Peripheral Devices

- **Peripheral device:** hardware component that is located "outside" of the central unit of a computer
  - Traditional (PC): keyboard, monitor, ...
    ($\mapsto$ physically "outside")
  - In general: hardware functions that are not directly mapped into the processor's instruction set
    ($\mapsto$ logically "outside")
- Peripheral components are addressed via I/O registers
  - Control registers: instructions to control/query state of peripheral is encoded by bit patterns (e. g., `DDRD` for ATmega)
  - Data registers: required for exchange of data
    (e. g., `PORTD`, `PIND` for ATmega)
  - Registers are often only available as *read-only* or *write-only*

Peripheral Devices: Examples

- Selection of typical peripheral devices of a μController
  - **Timer/Counter** — Counting registers that are incremented with a defined frequency (timer) or by external signals (counter) and that trigger an interrupt at a configurable counting value.
  - **Watchdog timer** — Timer that has to be written to regularly otherwise a RESET is triggered ("dead man's button").
  - **(A)synchronous serial interface** — Component for serial (bit-wise) exchange of data with a synchronous (e. g., RS-232) or asynchronous (e. g., I^2C) protocol.
  - **A/D converter** — Component for onetime/continuous discretization of voltage values (e. g., 0–5V $\mapsto$ 10-bit integer).
  - **PWM generators** — Component for generating pulse-width–modulated signals, pseudo-analog (D/A) output.
  - **Ports** — Groups of usually 8 ports which can be set to GND or Vcc and whose states can be monitored. $\hookrightarrow$ 17–16
  - **Bus systems** — SPI, RS-232, CAN, Ethernet, MLI, I^2C, ...

Peripheral Devices – Registers

- Different architectures for accessing I/O registers
  - Memory-mapped: (most μC) — Registers are integrated in address space; access with memory instructions of the processor (`load`, `store`)
  - Port-based: (x86-based PCs) — Registers are organized in a separate I/O address space; access with special `in`- and `out`-instructions

Peripheral Devices – Registers

- Different architectures for accessing I/O registers
  - Memory-mapped: (most μC) — Registers are integrated in address space; access with memory instructions of the processor (`load`, `store`)
  - Port-based: (x86-based PCs) — Registers are organized in a separate I/O address space; access with special `in`- and `out`-instructions
- Addresses of the registers are listed in the hardware's documentation

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
$3F ($5F)	SREG	I	T	H	S	V	N	Z	C	8
$3E ($5E)	SPH	–	–	–	–	SP11	SP10	SP9	SP8	11
$3D ($5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	11
$3C ($5C)	OCR0	Timer/Counter0 Output Compare Register								86
$12 ($32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	67
$11 ($31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	67
$10 ($30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	68

[1]

Peripheral Devices – Registers (continued)

- For every port x, three registers are defined (example for $x = D$)

- **DDRx** **Data Direction Register:** Determines for every pin i whether it is used as input (bit i=0) or output (bit i=1).

7	6	5	4	3	2	1	0
DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- **PORTx** **Data Register:** If pin i is configured to be an output, bit i determines the voltage level (0=GND sink, 1=Vcc source). If pin i is configured to be an input, bit i activates the internal pull-up resistor (1=active).

7	6	5	4	3	2	1	0
PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- **PINx** **Input Register:** Bit i represents the voltage level at pin i (1=*high*, 0=*low*), independent of the data direction of the register.

7	6	5	4	3	2	1	0
PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Examples: ↪ 3–8 and ↪ 3–12 [1]

Peripheral Devices – Registers (continued)

- Memory-mapped registers enable convenient access
  - register $\mapsto$ memory $\mapsto$ variable
  - all C operators are directly available (e. g., `PORTD++`)
- Syntactically, preprocessor macros simplify the access:

```
#define PORTD (*(volatile uint8_t *)   0x12    )
```


Peripheral Devices – Registers (continued)

- Memory-mapped registers enable convenient access
  - register $\mapsto$ memory $\mapsto$ variable
  - all C operators are directly available (e. g., `PORTD++`)
- Syntactically, preprocessor macros simplify the access:

```
#define PORTD (*(volatile uint8_t *)   0x12    )
                                       ^^^^
                                       address: int
```

Peripheral Devices – Registers (continued)

- Memory-mapped registers enable convenient access
  - register $\mapsto$ memory $\mapsto$ variable
  - all C operators are directly available (e. g., `PORTD++`)
- Syntactically, preprocessor macros simplify the access:

```
#define PORTD (*(volatile uint8_t *)   0x12    )
```

`0x12`: `address: int`

`(*(volatile uint8_t *) 0x12 )`: `address: volatile uint8_t *(cast ↪ 7–32)`

Peripheral Devices – Registers (continued)

- Memory-mapped registers enable convenient access
  - register $\mapsto$ memory $\mapsto$ variable
  - all C operators are directly available (e. g., `PORTD++`)
- Syntactically, preprocessor macros simplify the access:

```
#define PORTD (*(volatile uint8_t *)   0x12    )
```

`0x12`: address: int

`(volatile uint8_t *)`: address: volatile uint8_t *(cast ↪ 7–32)

`(*(volatile uint8_t *) ...)`: value: volatile uint8_t (dereferencing ↪ 13–5)

Peripheral Devices – Registers (continued)

- Memory-mapped registers enable convenient access
  - register $\mapsto$ memory $\mapsto$ variable
  - all C operators are directly available (e. g., `PORTD++`)
- Syntactically, preprocessor macros simplify the access:

```
#define PORTD (*(volatile uint8_t *)   0x12    )
```

`0x12`: `address: int`

`(volatile uint8_t *)0x12`: `address: volatile uint8_t *`(cast $\hookrightarrow$ 7–32)

`*(volatile uint8_t *)0x12`: `value: volatile uint8_t` (dereferencing $\hookrightarrow$ 13–5)

Therefore `PORTD` is syntactically-equivalent to a `volatile uint8_t`-variable that is stored at address `0x12`.

Peripheral Devices – Registers (continued)

- Memory-mapped registers enable convenient access
  - register $\mapsto$ memory $\mapsto$ variable
  - all C operators are directly available (e. g., `PORTD++`)
- Syntactically, preprocessor macros simplify the access:

```
#define PORTD (*(volatile uint8_t *)   0x12    )
```

`0x12`: address: int

`(volatile uint8_t *)`: address: volatile uint8_t *(cast ↪ 7–32)

`(*...)`: value: volatile uint8_t (dereferencing ↪ 13–5)

Therefore `PORTD` is syntactically-equivalent to a `volatile uint8_t`-variable that is stored at address `0x12`.

- Example

```
#define PORTD (*(volatile uint8_t *) 0x12)

PORTD |= (1<<7);           // set D.7
uint8_t *pReg = &PORTD;    // get pointer to PORTD
*pReg &= ~(1<<7);          // use pointer to clear D.7
```

Access to Registers and Concurrency

- Peripheral devices operate concurrently to the software
  ⇝ Value in hardware registers can change anytime
- This change in contrast to the assumption of the compiler
  - Access to variables only takes place by the currently executed function
    ⇝ Variables are temporarily stored in registers

Access to Registers and Concurrency

- Peripheral devices operate concurrently to the software
  ⇝ Value in hardware registers can change anytime
- This change in contrast to the assumption of the compiler
  - Access to variables only takes place by the currently executed function
    ⇝ Variables are temporarily stored in registers

```
// C code
#define PIND \
    (*(uint8_t*) 0x10)
void foo(void) {
    ...
    if (! (PIND & 0x2)) {

        // button0 pressed
        ...
    }
    if (! (PIND & 0x4)) {

        // button 1 pressed
        ...
    }
}
```

```
// Resulting assembly code

foo:
   lds   r24, 0x0010 // PIND->r24
   sbrc  r24, 1      // test bit 1
   rjmp L1
   // button0 pressed
   ...
L1:
   sbrc r24, 2       // test bit 2
   rjmp  L2

   ...
L2:
   ret
```

Access to Registers and Concurrency

- Peripheral devices operate concurrently to the software
  ⇝ Value in hardware registers can change anytime
- This change in contrast to the assumption of the compiler
  - Access to variables only takes place by the currently executed function
    ⇝ Variables are temporarily stored in registers

```
// C code
#define PIND \
    (*(uint8_t*) 0x10)
void foo(void) {
    ...
    if (! (PIND & 0x2)) {

        // button0 pressed
        ...
    }
    if (! (PIND & 0x4)) {

        // button 1 pressed
        ...
    }
}
```

```
// Resulting assembly code

foo:
   lds   r24, 0x0010 // PIND->r24
   sbrc  r24, 1      // test bit 1
   rjmp L1
   // button0 pressed
   ...
L1:
   sbrc r24, 2       // test bit 2
   rjmp  L2

   ...
L2:
   ret
```

PIND is not again loaded from memory. The compiler assumes the value in r24 to still be accurate.

The `volatile` Type Modifier

- **Solution:** Declare variable as `volatile` (*"transient, changeable"*)
  - Compiler minimizes the time, the variable is held in registers
    - ⇝ Value is read *immediately before* use
    - ⇝ Value is written *immediately after* modification

The `volatile` Type Modifier

- **Solution:** Declare variable as `volatile` (*"transient, changeable"*)
  - Compiler minimizes the time, the variable is held in registers
    - ⇝ Value is read *immediately before* use
    - ⇝ Value is written *immediately after* modification

```
// C code
#define PIND \
    (*(volatile uint8_t*) 0x10)
void foo(void) {
    ...
    if (! (PIND & 0x2)) {

        // button0 pressed
        ...
    }

    if (! (PIND & 0x4)) {

        // button 1 pressed
        ...
    }
}
```

```
// Resulting assembly code


foo:
   lds  r24, 0x0010 // PIND->r24
   sbrc r24, 1      // test bit 1
   rjmp L1
   // button0 pressed
   ...
L1:
   lds  r24, 0x0010 // PIND->r24
   sbrc r24, 2      // test bit 2
   rjmp L2

...
L2:
   ret
```

`PIND` is declared as `volatile`. It is therefore loaded from memory before each test.

The `volatile` Type Modifier (continued)

- `volatile` semantics prevent many code optimizations;
  in particular the removal of apparently unnecessary code
- `volatile` can be used to implement active waiting:

```
// C code                                    // Resulting assembly code
void wait(void) {                            wait:
    for(uint16_t i=0; i<0xffff; i++);            // compiler has optimized
                                                 // "unneeded" loop
}                                                ret
```

The `volatile` Type Modifier (continued)

- `volatile` semantics prevent many code optimizations;
  in particular the removal of apparently unnecessary code
- `volatile` can be used to implement active waiting:

```
// C code                                          // Resulting assembly code
void wait(void) {                                  wait:
    for(uint16_t i=0; i<0xffff; i++);                  // compiler has optimized
                                                       // "unneeded" loop
}                                                      ret
```

`volatile`!

The `volatile` Type Modifier (continued)

- `volatile` semantics prevent many code optimizations;
  in particular the removal of apparently unnecessary code
- `volatile` can be used to implement active waiting:

```
// C code
void wait(void) {
    for(uint16_t i=0; i<0xffff; i++);
}
```

volatile!

```
// Resulting assembly code
wait:
    // compiler has optimized
    // "unneeded" loop
    ret
```

Attention: volatile $\mapsto$ **$$$**

The use of `volatile` causes considerable runtime penalties

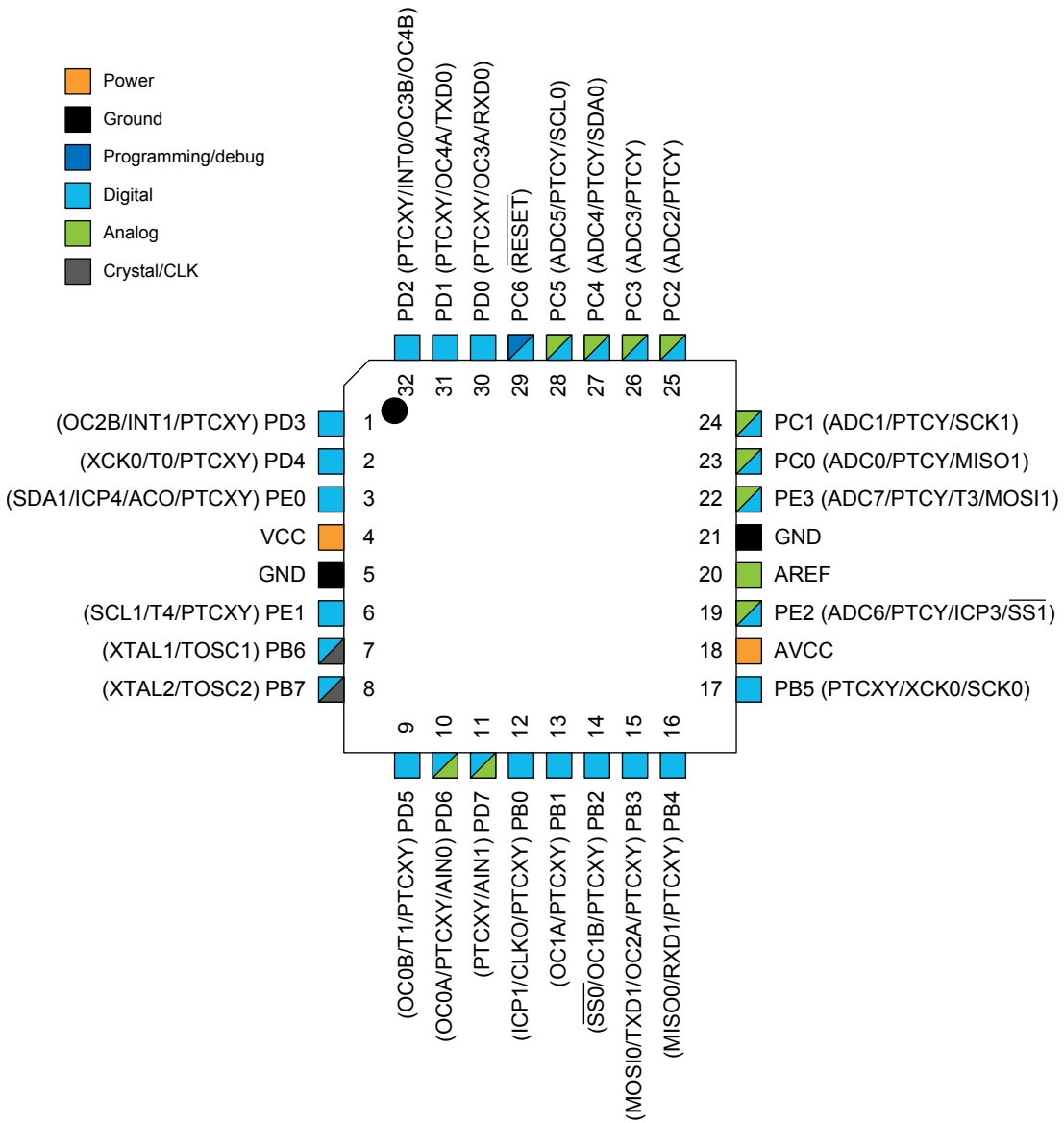
- Values cannot be stored in registers any longer
- Most code optimizations cannot be performed any longer

Rule: Use `volatile` only in **justified scenarios**

Peripheral Devices: Ports

- **Port** := group of (usually 8) digital inputs/outputs
  - Digital output: bit value $\mapsto$ voltage level at μC pin
  - Digital input: voltage level at μC pin $\mapsto$ bit value
  - External interrupt: voltage level at μC pin $\mapsto$ bit value
    (on voltage change) $\rightsquigarrow$ processor executes interrupt program
- This function is usually configurable per pin
  - Input
  - Output
  - External interrupt (only for some inputs)
  - Alternative functions (pin used by another device)

Example ATmega328PB: Port/Pin Assignment



For reasons of cost efficiency nearly every pin is assigned twice. The configuration of the respective functionality takes place in software.

E. g., pins 23–24 are configured as ADCs at the SPiCBoard to connect potentiometer and photo sensor.

Those pins are therefore not available for PORTC.

References

[1] *ATmega328PB 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*. Atmel Corporation. Okt. 2015.