System-Level Programming

17  \( \mu C \)-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?

\[ \mu \text{Controller} := \text{processor} + \text{memory} + \text{peripherals} \]

- In fact, computer system on one chip \( \rightarrow \) System-on-a-Chip (SoC)
- Often usable without additional external components, like e.g., clock generators and memory \( \sim \) cost-efficient system design

Main features are (plentiful) contained input/output components (peripherals)

The distinctions are not fixed: processor \( \leftrightarrow \) µC \( \leftrightarrow \) SoC

- AMD64 CPUs also have included timers, memory (cache), . . .
- Some µC reach the speed of “big processors”
Example ATmega32: Block Circuit Diagram

- **CPU core**
- **Memory**
- **Peripherals**
Peripheral Devices

- **Peripheral device:** hardware component that is located “outside” of the central unit of a computer
- **Traditional (PC):** keyboard, monitor, ...
  (↦physically “outside”)
- **In general:** hardware functions that are not directly mapped into the processor’s instruction set
  (↦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., I²C) protocol.

- **A/D converter**: Component for onetime/continuous discretization of voltage values (e.g., 0–5V → 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.

- **Bus systems**: SPI, RS-232, CAN, Ethernet, MLI, I²C, ...
Peripheral Devices – Registers

Different architectures for accessing I/O registers

- Memory-mapped: (most \( \mu \)C) 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: Registers are integrated in address space; access with memory instructions of the processor (load, store) (most µC)
  - Port-based: Registers are organized in a separate I/O address space; access with special in- and out-instructions (x86-based PCs)

Addresses of the registers are listed in the hardware’s documentation

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3F ($5F)</td>
<td>SREG</td>
<td>I</td>
<td>T</td>
<td>H</td>
<td>S</td>
<td>V</td>
<td>N</td>
<td>Z</td>
<td>C</td>
<td>8</td>
</tr>
<tr>
<td>$3E ($5E)</td>
<td>SPH</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>SP11</td>
<td>SP10</td>
<td>SP9</td>
<td>SP8</td>
<td>11</td>
</tr>
<tr>
<td>$3D ($5D)</td>
<td>SPL</td>
<td>SP7</td>
<td>SP6</td>
<td>SP5</td>
<td>SP4</td>
<td>SP3</td>
<td>SP2</td>
<td>SP1</td>
<td>SP0</td>
<td>11</td>
</tr>
<tr>
<td>$3C ($5C)</td>
<td>OCR0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>86</td>
</tr>
</tbody>
</table>

| $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                  | PIN7   | PIN6   | PIN5   | PIN4   | PIN3   | PIN2   | PIN1   | PIN0   | 68   |
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 $i=1$).
  
<table>
<thead>
<tr>
<th>DDRD7</th>
<th>DDRD6</th>
<th>DDRD5</th>
<th>DDRD4</th>
<th>DDRD3</th>
<th>DDRD2</th>
<th>DDRD1</th>
<th>DDRD0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
</tbody>
</table>

- **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).
  
<table>
<thead>
<tr>
<th>PORTD7</th>
<th>PORTD6</th>
<th>PORTD5</th>
<th>PORTD4</th>
<th>PORTD3</th>
<th>PORTD2</th>
<th>PORTD1</th>
<th>PORTD0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
</tbody>
</table>

- **PINx**  
  **Input Register:** Bit $i$ represents the voltage level at pin $i$ (1=high, 0=low), independent of the data direction of the register.
  
<table>
<thead>
<tr>
<th>PIND7</th>
<th>PIND6</th>
<th>PIND5</th>
<th>PIND4</th>
<th>PIND3</th>
<th>PIND2</th>
<th>PIND1</th>
<th>PIND0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
</tbody>
</table>

Examples: $\leftarrow \text{3–8}$ and $\rightarrow \text{3–12}$ [1]
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:

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

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

Example
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:

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

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

Example
Peripheral Devices – Registers (continued)

- Memory-mapped registers enable convenient access
  - register \(\mapsto\) memory \(\mapsto\) variable
  - all C operators are directly available (e.g., \texttt{PORTD++})

- Syntactically, preprocessor macros simplify the access:

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

  Therefore \texttt{PORTD} is syntactically-equivalent to a \texttt{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:

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

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

Example © klsw System-Level Programming (ST 24) 17 µC-System Architecture – Peripherals – Peripherals
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:

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

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:

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

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

Example

```c
#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 \textit{concurrently} to the software
  \textit{\sim} Value in hardware registers can change \textit{anytime}

- This change in contrast to the \textbf{assumption of the compiler}
  - Access to variables \textbf{only} takes place by the \textbf{currently executed function}
  \textit{\sim} 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
// C code
#define PIND (*(uint8_t*) 0x10)
void foo(void) {
    ...
    if (! (PIND & 0x2)) {
        // button0 pressed
        ...
    }
    if (! (PIND & 0x4)) {
        // button 1 pressed
        ...
    }
}
```

```assembly
// 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
```
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
// C code
#define PIND (*(uint8_t*) 0x10)
void foo(void) {
    ...
    if (! (PIND & 0x2)) {
        // button0 pressed
        ...
    }
    if (! (PIND & 0x4)) {
        // button 1 pressed
        ...
    }
}
```

```assembly
// 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
#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         // button 1 pressed
                     ...
L2:
    ret
```

`PIND` is declared as `volatile`. It is therefore loaded from memory before each test.
**volatile** semantics prevent many code optimizations; in particular the removal of *apparently unnecessary code*

**volatile** can be used to implement active waiting:

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

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

*Attention:* 

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.

volatile semantics prevent many code optimizations; in particular the removal of apparently unnecessary code.

volatile can be used to implement active waiting:

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

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

Attention:
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.
volatile semantics prevent many code optimizations; in particular the removal of apparently unnecessary code.

volatile can be used to implement active waiting:

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

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

**Attention: volatile \(\rightarrow\) $$$

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 $\mu$C pin
- Digital input: voltage level at $\mu$C pin $\mapsto$ bit value
- External interrupt: voltage level at $\mu$C pin $\mapsto$ bit value
  (on voltage change) $\leadsto$ 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)
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.