

# System-Level Programming

## 11 Preprocessor

**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>





- Before a C source file is compiled, it is processed by the macro preprocessor
  - in the past, a stand-alone program (**CPP** = **C PreProcessor**)
  - nowadays, integrated into compilers
- The CPP edits the source code by **text transformations**
  - automatic transformation (“clean-up” of the source code)
    - comments get deleted
    - lines ending with \ get put back together
    - ...
  - controllable transformations (by the programmer)
    - **preprocessor directives** are evaluated and executed
    - **preprocessor macros** are expanded



- **preprocessor directive** := control expression for the preprocessor

`#include <file>`

**Inclusion:** The contents of *file* are included at this exact place into the token stream.



- **preprocessor directive** := control expression for the preprocessor

`#include` *<file>*

**Inclusion:** The contents of *file* are included at this exact place into the token stream.

`#define` *macro replacement*

**Definition of macros:** Defines a preprocessor macro *macro*. In the following token stream, each occurrence of *macro* will be replaced by *replacement*. *replacement* can be empty.



- **preprocessor directive** := control expression for the preprocessor

`#include` *<file>*

**Inclusion:** The contents of *file* are included at this exact place into the token stream.

`#define` *macro replacement*

**Definition of macros:** Defines a preprocessor macro *macro*. In the following token stream, each occurrence of *macro* will be replaced by *replacement*. *replacement* can be empty.

`#if` *condition*,  
`#elif`, `#else`, `#endif`

**Conditional compilation:** Following lines of code are handed to the compiler or are deleted from the token stream dependent on *condition*.

`#ifdef` *macro*,  
`#ifndef` *macro*

Conditional compilation dependent on the definition of *macro* (e. g., with `#define`).



- **preprocessor directive** := control expression for the preprocessor

`#include <file>`

**Inclusion:** The contents of *file* are included at this exact place into the token stream.

`#define macro replacement`

**Definition of macros:** Defines a preprocessor macro *macro*. In the following token stream, each occurrence of *macro* will be replaced by *replacement*. *replacement* can be empty.

`#if condition,`  
`#elif, #else, #endif`

**Conditional compilation:** Following lines of code are handed to the compiler or are deleted from the token stream dependent on *condition*.

`#ifdef macro,`  
`#ifndef macro`

Conditional compilation dependent on the definition of *macro* (e. g., with `#define`).

`#error text`

**Abort:** The compilation procedure gets aborted with the error message *text*.



- **preprocessor directive** := control expression for the preprocessor

`#include <file>`

**Inclusion:** The contents of *file* are included at this exact place into the token stream.

`#define macro replacement`

**Definition of macros:** Defines a preprocessor macro *macro*. In the following token stream, each occurrence of *macro* will be replaced by *replacement*. *replacement* can be empty.

`#if condition,`  
`#elif, #else, #endif`

**Conditional compilation:** Following lines of code are handed to the compiler or are deleted from the token stream dependent on *condition*.

`#ifdef macro,`  
`#ifndef macro`

Conditional compilation dependent on the definition of *macro* (e. g., with `#define`).

`#error text`

**Abort:** The compilation procedure gets aborted with the error message *text*.

The preprocessor defines an embedded **meta language**. All preprocessor directives (i.e., the meta program) modify the C program (i.e., actual program) prior to actual compilation.



- Simple macro definitions

empty macro (flag)            `#define USE_7SEG`

source code constant        `#define NUM_LEDS (4)`

“inline”-function           `#define SET_BIT(m, b) (m | (1 << b))`

Preprocessor directives are **not** followed by a semicolon!





- Simple macro definitions

empty macro (flag)            `#define USE_7SEG`


source code constant        `#define NUM_LEDS (4)`


“inline”-function           `#define SET_BIT(m, b) (m | (1 << b))`

Preprocessor directives are **not** followed by a semicolon!

- Usage

```
#if NUM_LEDS < 0 || 8 < NUM_LEDS
# error invalid NUM_LEDS           // this line is not included
#endif

void enlighten(void) {
    uint8_t mask = 0, i;
    for (i = 0; i < NUM_LEDS; i++) { // NUM_LEDS --> (4)
        mask = SET_BIT(mask, i);    // SET_BIT(mask, i) --> (mask | (1 << i))
    }
    sb_led_setMask(mask);           // --> 
}

#ifdef USE_7SEG
    sb_show_HexNumber(mask);       // --> 
#endif
}
```



- Function-like macros are indeed no functions!
  - Parameters are not evaluated, rather they are inserted **textually**  
This can lead to **unpleasant surprises**

```
#define POW2(a) 1 << a  
n = POW2(2) * 3
```

<< has lower precedence than \*

~> n = 1 << 2 \* 3



- Function-like macros are indeed no functions!
  - Parameters are not evaluated, rather they are inserted **textually**  
This can lead to **unpleasant surprises**

```
#define POW2(a) 1 << a  
n = POW2(2) * 3
```

<< has lower precedence than \*  
~ n = 1 << 2 \* 3

- Some problems can be avoided by the correct use of brackets

```
#define POW2(a) (1 << a)  
n = POW2(2) * 3
```

~ n = (1 << 2) \* 3



- Function-like macros are indeed no functions!
  - Parameters are not evaluated, rather they are inserted **textually**  
This can lead to **unpleasant surprises**

```
#define POW2(a) 1 << a  
n = POW2(2) * 3
```

<< has lower precedence than \*  
~ n = 1 << 2 \* 3

- Some problems can be avoided by the correct use of brackets

```
#define POW2(a) (1 << a)  
n = POW2(2) * 3
```

~ n = (1 << 2) \* 3

- However, not all

```
#define max(a, b) ((a > b) ? a : b)  
n = max(x++, 7)
```

a++ will be potentially evaluated twice  
~ n = ((x++ > 7) ? x++ : 7)



- Function-like macros are indeed no functions!
  - Parameters are not evaluated, rather they are inserted **textually**  
This can lead to **unpleasant surprises**

```
#define POW2(a) 1 << a
n = POW2(2) * 3
```

<< has lower precedence than \*  
 ~ n = 1 << 2 \* 3

- Some problems can be avoided by the correct use of brackets

```
#define POW2(a) (1 << a)
n = POW2(2) * 3
```

~ n = (1 << 2) \* 3

- However, not all

```
#define max(a, b) ((a > b) ? a : b)
n = max(x++, 7)
```

a++ will be potentially evaluated twice  
 ~ n = ((x++ > 7) ? x++ : 7)

- A possible alternative are real **inline**-functions

C99

- function code gets embedded ~ as efficient as macros

```
inline int max(int a, int b) {
    return (a > b) ? a : b;
}
```

