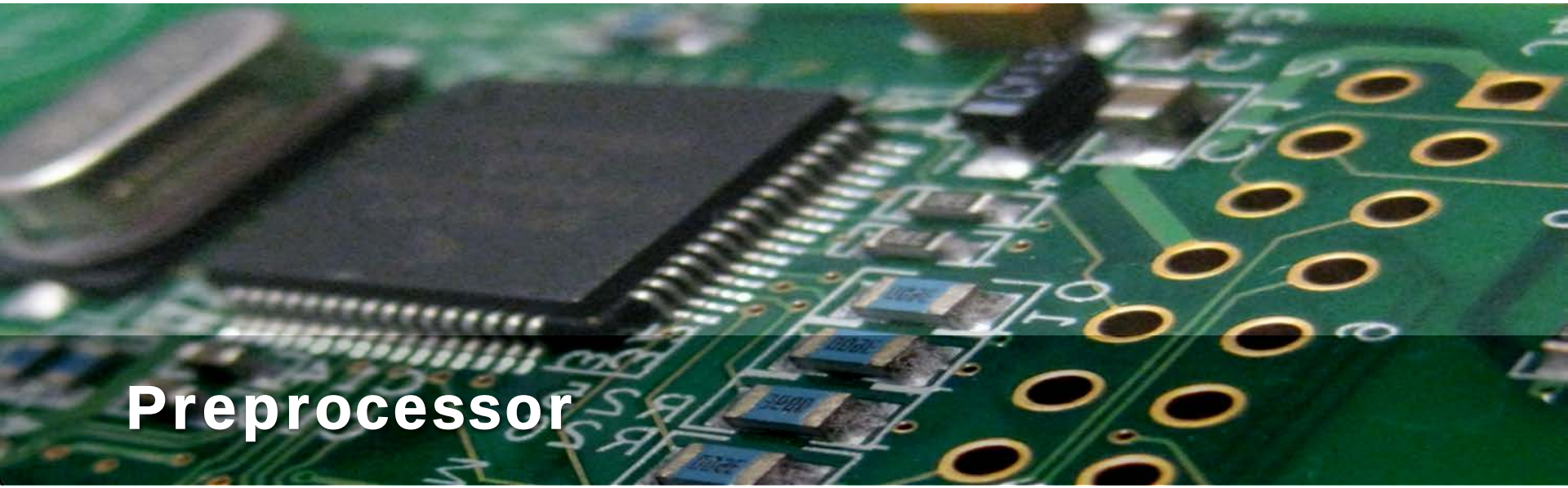
Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN
Technik & Architektur



Preprocessor

"There is always some prep work upfront."

Prof. Erich Styger
erich.styger@hslu.ch
+41 41 349 33 01

Scriptum:
ANSI-C, Exploring Embedded C

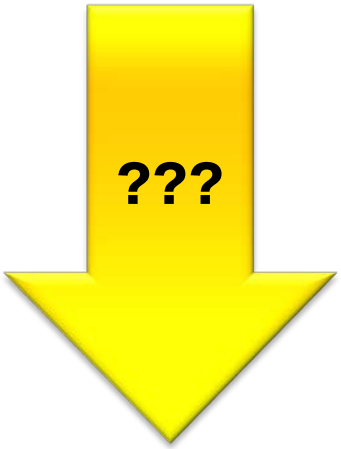
Learning Goals

- Problem: Understanding Macros

- Compiler

- Aspects of
  - Reusability
  - Optimization
  - Debugging
  - Maintenance

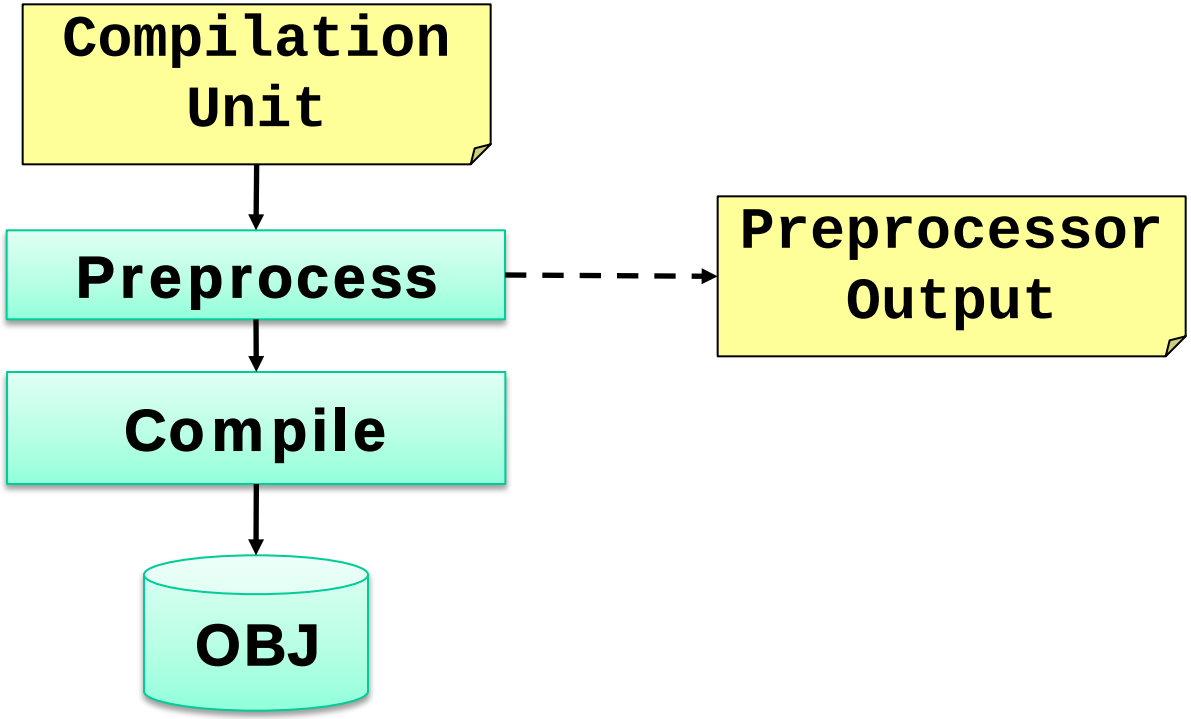
- Macros
  - Usage
  - Pitfalls



C/C++ Macros/#define

- Definition of a macro
- Compiler is replacing macros textually

```
#define BLUE   0
#define RED    1
#define YELLOW 2
```



Textual Replacement

```
int ChangeColor(int color) {
  if (color == BLUE) {
    return RED;
  }
}
```

```
#define BLUE   0
#define RED    1
#define YELLOW 2
```

```
int ChangeColor (int color) {
  if (color == 0) {
    return 1;
  }
}
```

Why Macros?

- Names instead of ‚magic‘ numbers
  `#define DELAY_TIME_MS 10`
- Configuration
  `#define DEBUG_ME 1`
- Portability
  `#define ENABLE_INTERRUPTS  __asm(“CPSIE”)`
- Optimization

Knowing what you are doing
&
Be cautious with whatever you do

Traps & Pitfalls

```
#define INCI(i) {int a=0; i++;}

void main(void) {
  int a = 0, b = 0;
  INCI(a);
  INCI(b);
  printf("a is now %d, b is now %d\n", a, b);
}
```

```
void main(void) {
  int a = 0, b = 0;
  {int a=0; a++;};
  {int a=0; b++;};
  printf("a is now %d, b is now %d\n", a, b);
}
```

a is now 0, b is now 1

Traps & Pitfalls

```
#define PRE_DELAY   5
#define POST_DELAY  2
#define DELAY       PRE_DELAY + POST_DELAY
```

```
return totalDelay(int nofIterations) {
  return nofIterations * DELAY;
}
```

n * 5 + 2

Traps & Pitfalls

```
#define PRE_DELAY   5
#define POST_DELAY  2
#define DELAY       (PRE_DELAY + POST_DELAY)
```

```
#define PRE_DELAY   (5*3)
#define POST_DELAY  (2+5)
#define DELAY       ((PRE_DELAY) + (POST_DELAY))
```

```
return totalDelay(int nofIterations) {
  return nofIterations * (((5*3)) + ((2+5)));
}
```

Example: Function Call

```
typedef enum {
  LED_0 = (1<<0), /*!< Bit0 of port for LED0 */
  LED_1 = (1<<1), /*!< Bit1 of port for LED1 */
  LED_2 = (1<<2), /*!< Bit2 of port for LED2 */
  LED_3 = (1<<3)  /*!< Bit3 of port for LED3 */
} LED_Set;

#define LED   (*((uint32_t*)0x1000))

void LED_On(LED_Set Leds) {
  LED |= Leds;
}

void main(void) {
  ...
  LED_On(LED_0|LED_1|LED_2|LED_3);
  ...
}
```

```
00 push  {r7}
02 sub   sp, sp, #12
04 add   r7, sp, #0
06 mov   r3, r0
08 strb  r3, [r7, #7]
0a mov   r3, #4096
0e mov   r2, #4096
12 ldr   r1, [r2]
14 ldrb  r2, [r7, #7]
16 orrs  r2, r2, r1
18 str   r2, [r3]
1a adds  r7, r7, #12
1c mov   sp, r7
1e ldr   r7, [sp], #4
22 bx    lr
24
```

```
00 movs  r0, #15
02 bl    LED_On
06
```

Code Size: 0x24+0x06 = 42 Bytes!

Inlining with Macros

```
/* led.h */
#define LED_On(leds) ((LED)|=leds)
```

```
void Test(void) {
  LED_On(LED_0|LED_1|LED_2|LED_3);
}
```

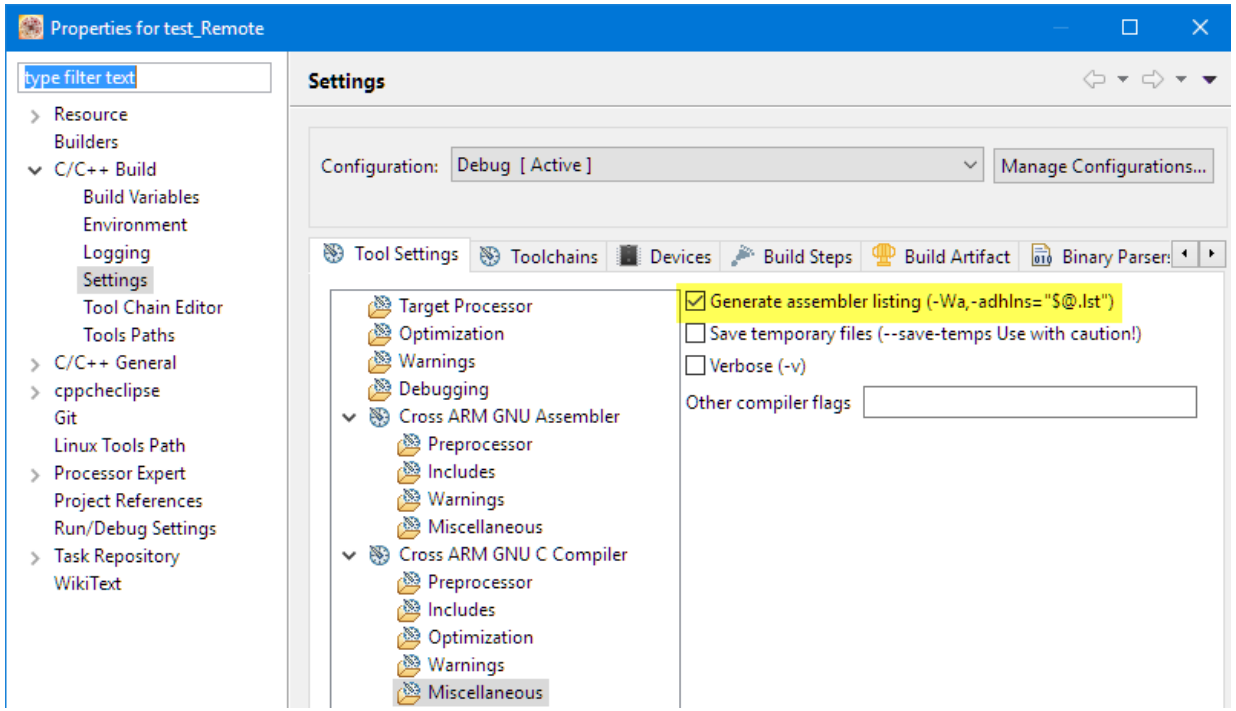
```
00 4FF48053 mov   r3, #4096
04 4FF48052 mov   r2, #4096
08 1268     ldr   r2, [r2]
0a 42F00F02 orr   r2, r2, #15
0e 1A60     str   r2, [r3]
10
```

16 Bytes Code!

- Pros
  - Faster code
  - Smaller Code
- Cons
  - Interface
  - Encapsulation
  - Debugging

Generating Listing File

- Generates *.lst for each *.c in output (Debug) folder

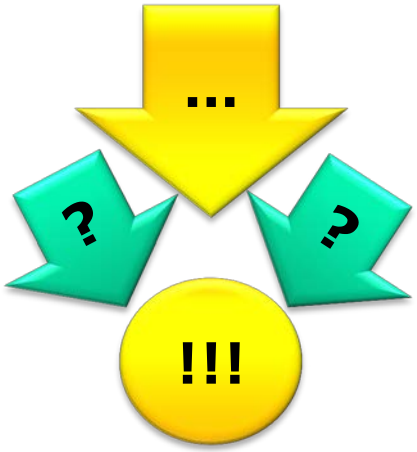


Summary

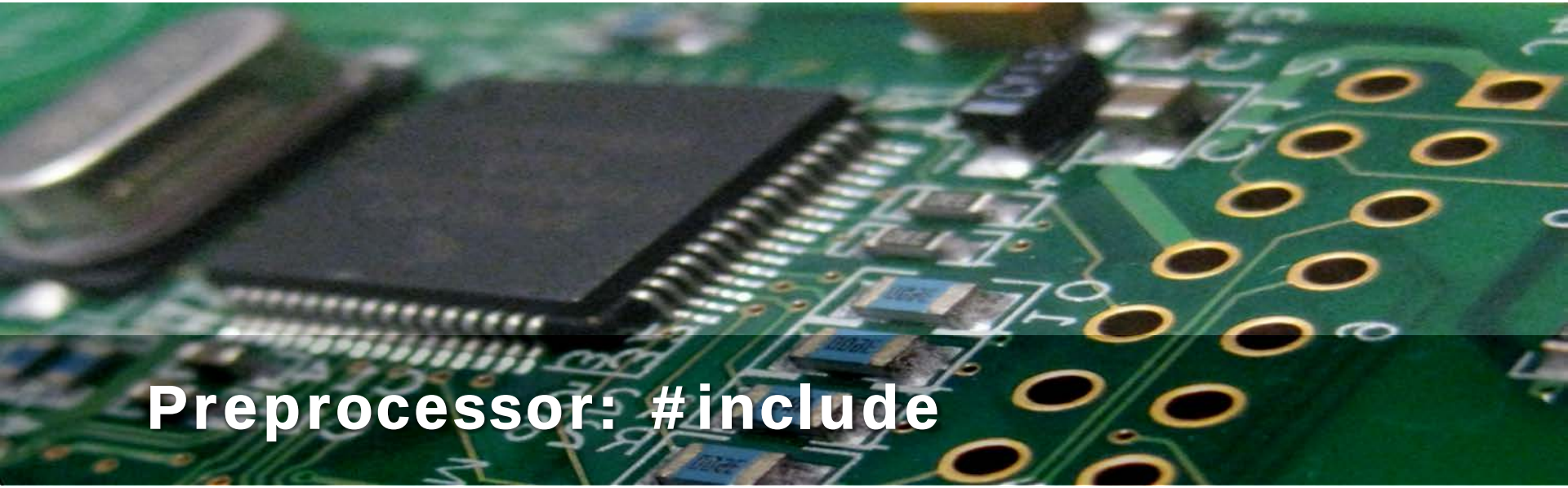
- *Problem: Using and understanding macros*

- Macros
  - textually replaced by preprocessor
  - Pros and Cons
  - Traps & Pitfalls (Parenthesis!)

- Inlining (efficiency!)
  - ➔ Other approach: inlining with compiler (`inline`)



Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN
Technik & Architektur



Preprocessor: #include

"I thought I did the right thing?…"

Prof. Erich Styger
erich.styger@hslu.ch
+41 41 349 33 01

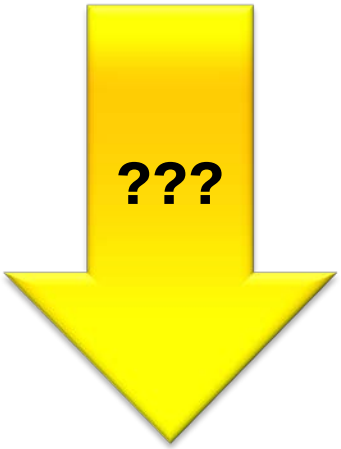
Learning Goals

- Goal
  - Understanding Compiler Includes

- Header File for Interface/Declaration
- Source File for Implementation

- Mechanics of #include
- Guarding
- What and Where

- Common Rules



Header / Source, what is where?

- Declaration: Name
- Definition: Memory allocation
- Convention:
  - *.c: Implementation File, Definition
  - *.h: Interface/Header File: external Declarations

```c
/* drv.c */
#include "drv.h"

int DRV_global = 7;
static int v;

void DRV_Init(void) {
  v = 3;
  DRV_global += v;
}
```

```c
/* drv.h */

#ifndef __DRV_H_
#define __DRV_H_

extern int DRV_global;

void DRV_Init(void);

#endif /* __DRV_H_ */
```

```c
/* main.c */

#include "drv.h"

void main(void) {
  DRV_Init();
  DRV_global++;
}
```

#include Directive

- Textual inclusion of files
- Result is ‘compilation unit’

```
/* b.h */
extern int b;
```

```
/* a.h */
extern int a;
```

```
/* m.h */
#include “b.h”
```

```
/* m.c */
#include “m.h”
#include “a.h”
void foo(void) {}
```

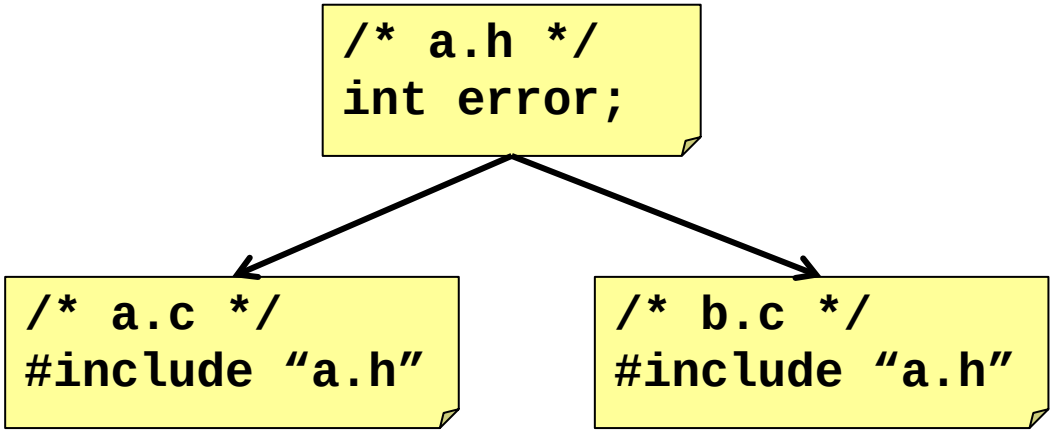
Preprocessor

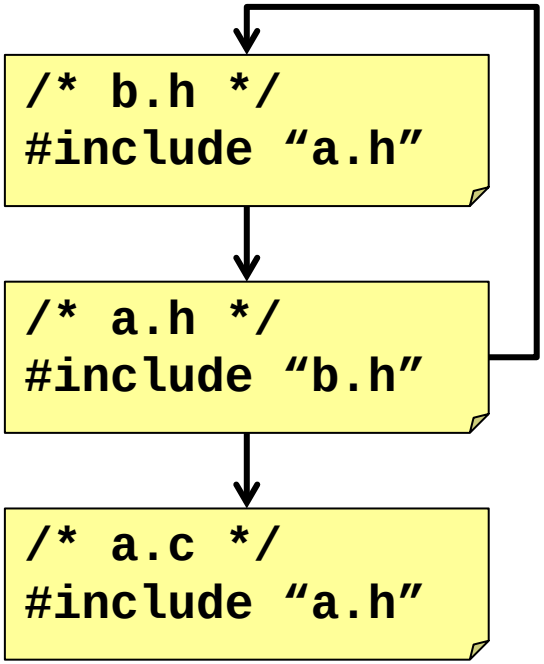
Compilation Unit

```
/* m.c */
/* m.h */
/* b.h */
extern int b;
/* a.h */
extern int a;
void foo(void) {}
```

#ifndef - #define - #endif

- Protection against
  - multiple declarations/definitions
  - Recursive includes





#ifndef - #define - #endif

- ‘Protection’ symbol
  - Avoid name conflicts!
  - Convention: __<*FileName*>_H_
  - Double Underscore: ‘reserved names’

```
/* platform.h */
#ifndef __PLATFORM_H_
#define __PLATFORM_H_
  #define PL_HAS_LED (1)
#endif /* __PLATFORM_H_ */
```

```
/* led.h */
#ifndef __LED_H_
#define __LED_H_
  void LED_On(void);
#endif /* __LED_H_ */
```

```
#include “platform.h”
#include “led.h”
void foo(void) {
  LED_On();
}
```

What and Where? Self-Containment

- As a developer, you could do whatever you want. BUT:…
- Include your own interface too!
- Header file should be ‘self contained’
  - Users of the interface should only need to include that interface
  - Using an interface/header file shall not depend on include order

```
/* led.c */
…
```

```
/* led.h */
#include “BitIO.h”
#define LED_On() \
   BitIO_BitOn()
```

```
/* main.c */
#include “LED.h”
void main(void) {
  LED_On();
}
```

```
/* led.c */
#include “BitIO.h”
#include “LED.h”
void LED_On(void) {
  BitIO_BitOn();
}
```

```
/* led.h */
void LED_On(void);
```

```
/* main.c */
#include “LED.h”
void main(void) {
  LED_On();
}
```

What not to do…

- 'Reducing' includes in the wrong place is a bad thing

```
/* pwm.h */
…
```

```
/* led.h */
#include "pwm.h"
…
```

```
/* pwm.c */
…
```

```
/* led.c */
…
```

```
/* main.c */
#include "led.h"
void main(void) {
  LED_On();
  PWM_Off();
}
```

```
/* led.h */
…
```

```
/* pwm.h */
…
```

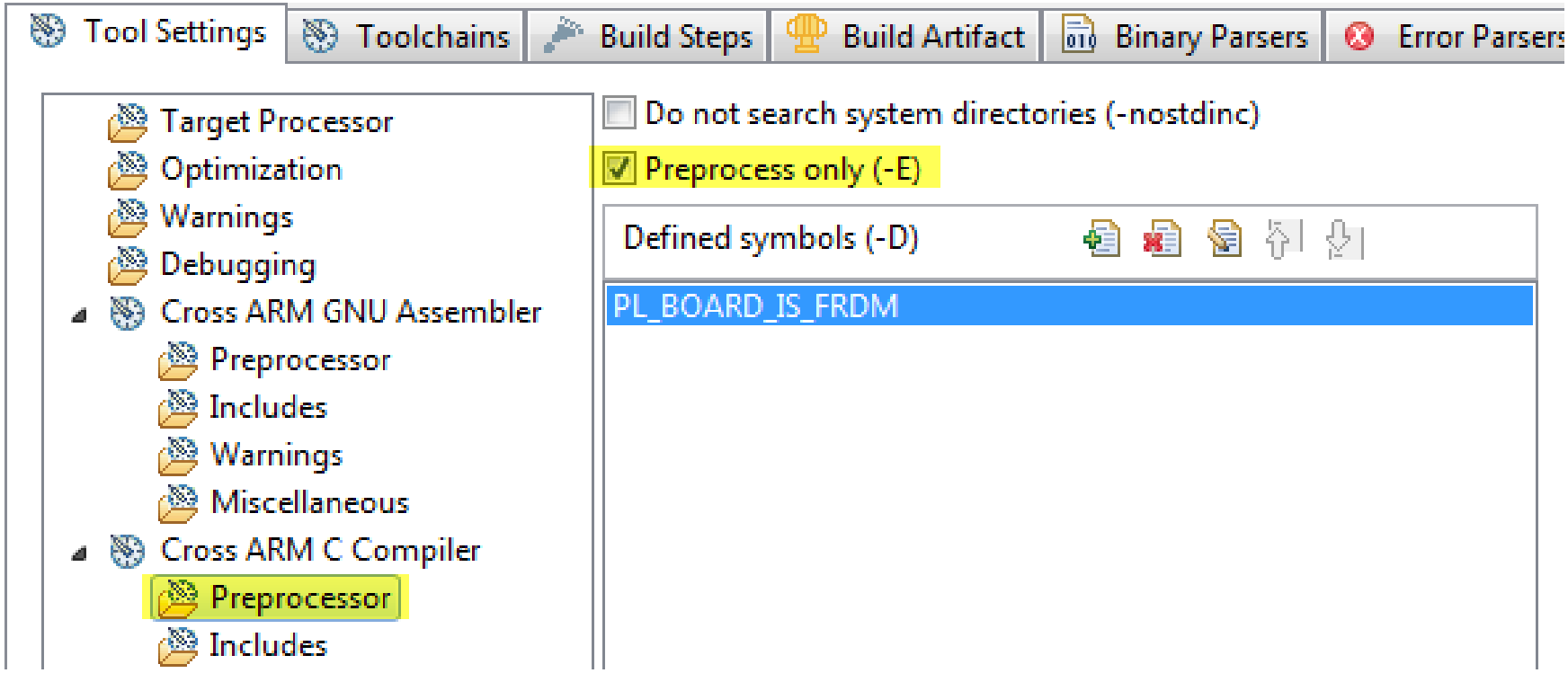
```
/* led.c */
…
```

```
/* pwm.c */
…
```

```
/* main.c */
#include "led.h"
#include "pwm.h"
void main(void) {
  LED_On();
  PWM_Off();
}
```

Generating Preprocessor Output

- Enable –E
- Compiler does not produce object files!
  - *.o are preprocessor output (text files)



Preprocessor Output

- Extension *.o given by build system
- Open as text files
- Link phase will fail!

main.o

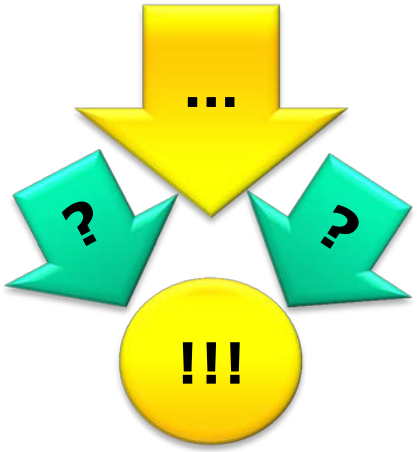
```
# 1 "../Sources/main.c"
# 1 "C:\\Users\\tastyger\\Data\\HSLU\\Vorlesung\\INTRO_HS2014\\git\\INTRO_HS2014\\Projec
#define __STDC__ 1
# 1 "../Sources/main.c"
#define __STDC_VERSION__ 199901L
# 1 "../Sources/main.c"
#define __STDC_HOSTED__ 1
# 1 "../Sources/main.c"
#define __GNUC__ 4
# 1 "../Sources/main.c"
#define __GNUC_MINOR__ 8
# 1 "../Sources/main.c"
#define __GNUC_PATCHLEVEL__ 0
# 1 "../Sources/main.c"
#define __VERSION__ "4.8.0"
# 1 "../Sources/main.c"
#define __ATOMIC_RELAXED 0
# 1 "../Sources/main.c"
```

Summary

- Information hiding
  - Only expose in interface/header file what is needed
  - Not more, not less

- Guard Header Files with #ifndef - #define - #endif
  - Use a guard define name with low chance of conflict

- Header file shall include only what is needed in the header file itself
- Source file includes the interfaces which are used for the implementation



Lab: Preprocessor

- *Problem: Macro Debugging*

- Preprocessor
  - Generate Preprocessor Listing
  - Inspect Preprocessing Listing
- Compiler Listing File
  - Generate Listing
  - Inspect Listing