

# SISTEMI EMBEDDED

The C Pre-processor  
Fixed-size integer types  
Bit Manipulation

Federico Baronti

Last version: 20160302

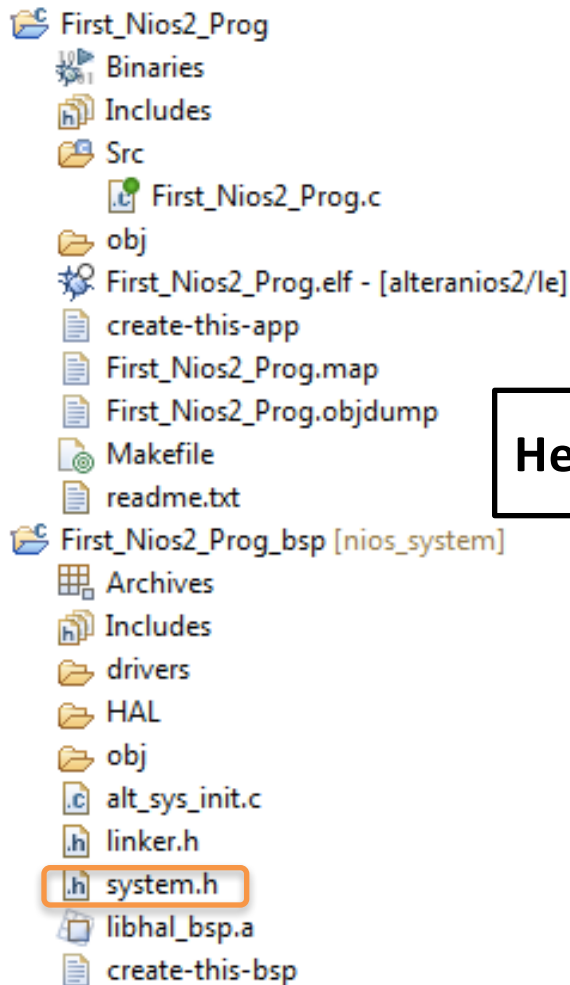
# The C PreProcessor CPP (1)

- **CPP** is a program called by the compiler that processes the **text** of the program before its actual translation
- It basically does the following operations:
  - Includes the content of other files (usually *header* files)
  - Expands the SYMBOLS with their related definitions
  - Includes/Excludes part of the code to the text that will be actually compiled
- These actions are controlled by **directives**
  - A directive is a single code line that starts with **#**
  - You can use the character `\` to go to a new line within the same directive

# The C PreProcessor (2)

- **Inclusion of header files** (files with .h extension that contains only declarations). E.g.  
`#include <stdint.h>`  
`#include "my_header.h"`
- The file `stdint.h` is searched in a standard directory list; `my_header.h` is searched in the same directory as the including source file
- The list of directories searched for header files can be changed with a compiler option

# The C PreProcessor (3)



First\_Nios2\_Prog.c

```
/* ... */  
#include "../FirstNios2_Prog_bsp/system.h"  
/* ... */
```

**Compiling First\_Nios2\_Prog.c (other compiler options omitted)**

```
nios2-elf-gcc -c -o First_Nios2_Prog.o First_Nios2_Prog.c
```

**Header path can be omitted by using *-I*dir compiler option**

First\_Nios2\_Prog.c

```
/* ... */  
#include "system.h"  
/* ... */
```

**Compiling First\_Nios2\_Prog.c (other compiler options omitted)**

```
nios2-elf-gcc -I../FirstNios2_Prog_bsp/ -c -o First_Nios2_Prog.o  
First_Nios2_Prog.c
```

# The C PreProcessor (4)

- **Macro** is a symbol that is replaced with its definition before compilation (it can be followed by one or or more arguments). E.g. of macro def.

```
#define MASK 0xF
```

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

- The instructions:

```
b = a & MASK;
```

```
y = 1 + MAX(10,x);
```

- are expanded by the preprocessor to:

```
b = a & 0xF;
```

```
y = 1 + ((10) > (x) ? (10) : (x));
```

# The C PreProcessor (5)

- **Macro** are largely used in C programming of embedded systems to access peripheral registers.

E.g. of definition:

```
#include "system.h"
```

```
#define RED_LEDS_DATA_REG \
    (*(volatile unsigned int*) RED_LEDS_BASE)
```

```
#define SLIDER_DATA_REG \
    (*(volatile unsigned int*) SLIDER_SWITCHES_BASE)
```

- E.g. of use:

```
RED_LEDS_DATA_REG = SLIDER_DATA_REG;
```

```
/* Show the status of the slider switches on the red leds */
```

# The C PreProcessor (6)

- **The macro** *name\_of\_the\_macro* exists from its definition to the end of the file or when it is undefined using the directive:  
*#undef name\_of\_the\_macro*
- A macro can also be defined with an option passed to the compiler:  
*-D name\_of\_the\_macro=def*
- Do a large use of parenthesis to avoid unintended behaviors when the MACRO is expanded
- **Write macro SYMBOLS with all CAPITAL letters**

# The C PreProcessor (7)

- **Conditional compilation** makes it possible to include/exclude code segments if certain expressions evaluated by the preprocessor are true or false. E.g.

```
#ifdef DEBUG
```

```
    printf("Debug mode enabled\n");
```

```
    /* or any other code that we want to include  
       for debug purposes */
```

```
#endif
```

- `#define DEBUG 1`  
includes the debug code



# The C PreProcessor (8)

- A common use of **conditional compilation** is to avoid multiple inclusions of a header file. To this end, start the header file, say config.h, with:

```
#ifndef CONFIG_H_  
#define CONFIG_H_
```

- and end it with:

```
#endif /* CONFIG_H_ */
```

- After the first inclusion of my\_header.h, the symbol MY\_HEADER\_H is defined. Thus, further inclusions are filtered out by the **conditional compilation directives**

# Integer types

- 2 basic integer types: *char*, *int*
- and some type-specifiers:
  - sign: *signed*, *unsigned*
  - size: *short*, *long*
- The actual size of an integer type depends on the compiler implementation
  - *sizeof(type)* returns the size (in number of bytes) used to represent the *type* argument
  - *sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)...*  
*≤ sizeof(long long)*

# Fixed-size integers (1)

- In embedded system programming **integer size is important**
  - Controlling minimum and maximum values that can be stored in a variable
  - Increasing efficiency in memory utilization
  - Managing peripheral registers
- To increase software portability, fixed-size integer types can be defined in a header file using the *typedef* keyword

# Fixed-size integers (2)

- **C99** update of the **ISO C standard** defines a set of standard names for signed and unsigned fixed-size integer types
  - 8-bit: `int8_t`, `uint8_t`
  - 16-bit: `int16_t`, `uint16_t`
  - 32-bit: `int32_t`, `uint32_t`
  - 64-bit: `int64_t`, `uint64_t`
- These types are defined in the standard-library header file **`stdint.h`**

# Fixed-size integers (3)

- Altera HAL (Hardware Abstraction Layer) also provides the header file **alt\_types.h** (<project\_name\_bsp>/HAL/inc/) with definition of fixed-size integer types:

```
typedef signed char           alt_8;  
typedef unsigned char       alt_u8;  
typedef signed short        alt_16;  
typedef unsigned short     alt_u16;  
typedef signed long         alt_32;  
typedef unsigned long      alt_u32;  
typedef long long           alt_64;  
typedef unsigned long long alt_u64;
```

- These type definitions are used in Altera HAL source files.
- **To increase portability, you'd better code using C99 fixed-size integer types (including the header file stdint.h)**

# Putting into practice

- Write a program that shows on the 7-seg display HEX3-HEX0 the sizes in number of bytes of *long long*, *long*, *short* and *char* integer data types
- Do they match with the definitions of fixed-size integer types in `alt_types.h`?

# Logical operators

- Integer data can be interpreted as **logical values** in conditions (if, while, ...) or in logical expressions:

**= 0, FALSE**

**ANY OTHER VALUE, TRUE**

- **Logical operators:**

<b>AND</b>	<b>&amp;&amp;</b>
<b>OR</b>	<b>  </b>
<b>NOT</b>	<b>!</b>

- Integer data can store the result of a logical expressions: 1 (**TRUE**), 0 (**FALSE**)

# Bitwise operators (1)

- Operate on the bits of the operand/s

<b>AND</b>	<b>&amp;</b>
<b>OR</b>	<b> </b>
<b>XOR</b>	<b>^</b>
<b>NOT</b>	<b>~</b>
<b>SHIFT LEFT</b>	<b>&lt;&lt;</b>
<b>SHIFT RIGHT</b>	<b>&gt;&gt;</b>



# Shift operators

- $A \ll n$ 
  - The result is the bits of  $A$  moved to the left by  $n$  positions and padded on the right with 0
  - It is equivalent to multiply  $A$  by  $2^n$  if the result can be represented
- $A \gg n$ 
  - The result is the bits of  $A$  moved to the right by  $n$  positions and padded on the left with 0 if type of  $A$  is unsigned or with the **MSB of  $A$**  if type is signed
  - It is equivalent to divide  $A$  by  $2^n$

# Bit manipulation (1)

- $\ll$  and  $|$  operands can be used to create expressive binary constants by specifying the positions of the bits equal to 1
  - E.g.  $(1\ll 7) | (1\ll 5) | (1\ll 0) = 0xA1$  (10100001)
  - Better not to use “magic numbers” as 7, 5 and 0. Use instead **symbolic names** to specify bit positions
    - For instance, the **symbolic names** can reflect the function of the bit within a peripheral register
  - $(1\ll x)$  can be encapsulated into a macro:
    - **#define** BIT(x)  $(1\ll(x))$

# Bit manipulations (2)

- Altering only the bits in given positions
  - E.g. bits: 7, 5, 0
  - *#define* MSK = BIT(7) | BIT(5) | BIT(0)
- Clearing bits
  - $A \&= \sim\text{MSK};$
- Setting bits
  - $A |= \text{MSK};$
- Toggling bits
  - $A \wedge= \text{MSK};$

# Bit manipulations (3)

- Testing bits
  - E.g. do something if bit 0 (LSB) of  $A$  is set, regardless of the other bits of  $A$
  - *if* ( $A \& \text{BIT}(0)$ ) {  
    /\* some code here \*/  
}