Macro
and
Bit Manipulation

Objectives

- Macro
- Bitwise Operations
- Bit Fields

Macro

- Macro definition
- Object-like Macros
- Function-like Macros
- Stringification and Concatenation
- Undefining and Redefining Macros

Macro Definition

What is macro

A *macro* is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro

Macro is defined using #define preprocessor directive in the C language.

When to use

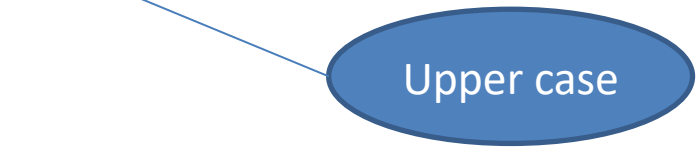
When creating constants that represent numbers, strings or expressions.

Macro classification

- Predefined macro
- User-defined macro

Object-like macros

#define MACRO_NAME macro's body



- Give symbolic names to numeric constants
- The macro's body end at the end of the #define line
- Single line macro:

```
#define SIZE 10
```

- Multiple line macro:

```
#define NUMBERS     1, \
                    2
```

Function-like macros

#define macro_name(list of parameters) macro's body

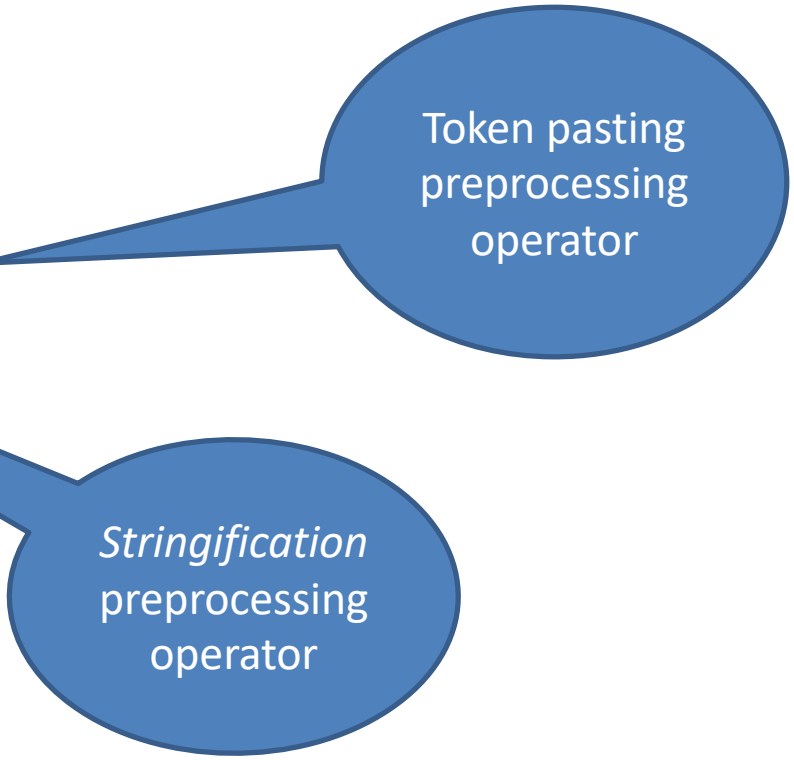

```
#define min(X, Y) ((X) < (Y) ? (X) : (Y))

x = min(a, b); ==> x = ((a) < (b) ? (a) : (b));

y = min(1, 2); ==> y = ((1) < (2) ? (1) : (2));
```

```
extern void foo(void);

#define foo() /* optimized inline version */

#define f   ()      callback()

...

foo();  → ?

funcptr = foo;  → ?

f() → ?
```

?

Stringification and *token pasting*

```
struct command {
    char *name;
    void (*function) (void);
};
#define COMMAND(NAME)  { #NAME, NAME ##
_command }
struct command commands[] = {
    COMMAND (quit),
    COMMAND (help),
};
struct command commands[] = {
    { "quit", quit_command },
    { "help", help_command },
};
```



Undefining and Redefining Macros

```
#ifdef  TRUE

#undef  TRUE

#define TRUE 1

#endif
```

BITWISE OPERATION (1)

Symbol	Operation
&	bitwise AND
|	bitwise inclusive OR
^	bitwise XOR (eXclusive OR)
<<	Left shift
>>	Right shift
~	bitwise NOT (one's complement) (unary)

bit a	bit b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

bit a	~a
0	1
1	0

BITWISE OPERATION (2)

Shift operations

A <<3

A >> 3

BITWISE OPERATION (3)

```
#include <stdio.h>
void main() {
   unsigned int a = 60;      /* 60 = 0011 1100 */
   unsigned int b = 13;      /* 13 = 0000 1101 */
   int c = 0;
   c = a & b;      /* 12 = 0000 1100 */
   printf("Line 1 - Value of c is %d\n", c );
   c = a | b;      /* 61 = 0011 1101 */
   printf("Line 2 - Value of c is %d\n", c );
   c = a ^ b;      /* 49 = 0011 0001 */
   printf("Line 3 - Value of c is %d\n", c );
   c = ~a;         /*-61 = 1100 0011 */
   printf("Line 4 - Value of c is %d\n", c );
   c = a << 2;     /* 240 = 1111 0000 */
   printf("Line 5 - Value of c is %d\n", c );
   c = a >> 2;     /* 15 = 0000 1111 */
   printf("Line 6 - Value of c is %d\n", c );
}
```

OUTPUT:

Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15

Bit Fields - 1

- This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

```
struct {
   unsigned int widthValidated;
   unsigned int heightValidated;
} status;
```

Bit Fields - 2

```
struct {
unsigned int widthValidated : 1;
unsigned int heightValidated : 1;
} status;
```

- The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.
- If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes.

Bit Fields - 3

```c
#include <stdio.h>
#include <string.h>
/* define simple structure */
struct {
  unsigned int widthValidated;
  unsigned int heightValidated;
} status1;
/* define a structure with bit fields */
struct {
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status2;
int main( ) {
  printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
  printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
  return 0;
}
```

OUTPUT:

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration - 1

- The declaration of a bit-field has the following form inside a structure:

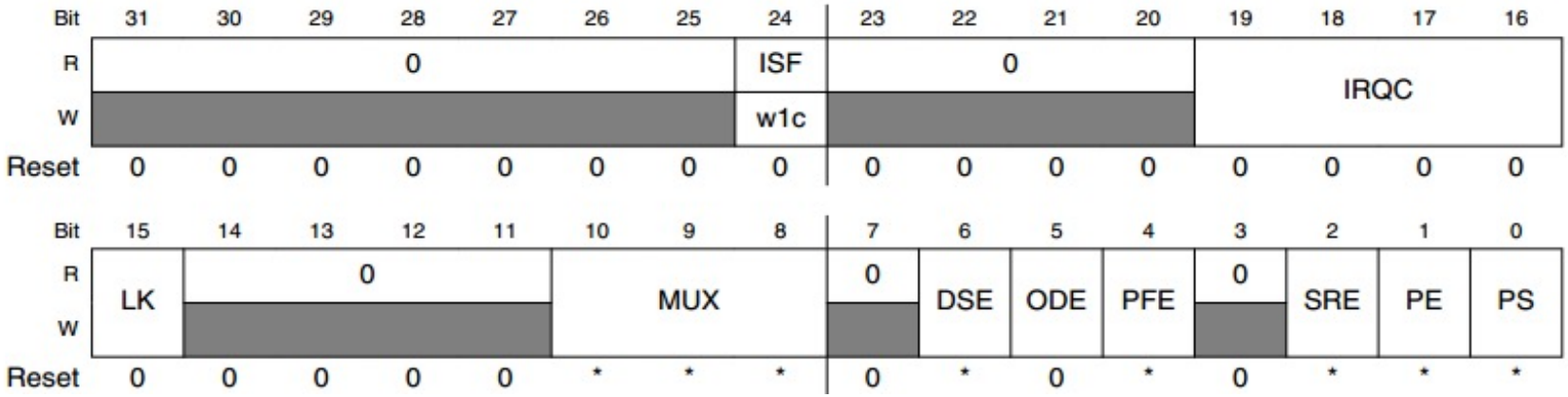
```
struct {
    type [member_name] : width ;
};
```

Bit Field Declaration - 2

- The following table describes the variable elements of a bit field:

Sr.No.	Element & Description
1	**type** An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
2	**member_name** The name of the bit-field.
3	**width** The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

Quiz(1)



Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0							ISF	0				IRQC			
W								w1c								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	LK	0				MUX			0	DSE	ODE	PFE	0	SRE	PE	PS
W																
Reset	0	0	0	0	0	*	*	*	0	*	0	*	0	*	*	*

The figure show the description of register PCR.

1. Write macros to define MASK and SHIFT location of each bit field.
2. Write macro to set IRQC to 3

Quiz(2)

- Write macro to convert 32bit value from big endian to little endian form

Thank you

Q&A