Programming in C
Assignment 1 : Bit Handler

Version: March 2022
Author: Brice Guayrin (b.guayrin@fontys.nl)

Introduction

With this assignment you are tasked to create the initial version of the Bit Handler tool written in C language. The tool provides C programmers with various pre-written methods for handling of bits. Bit Handler is intended to be used e.g. by embedded programmers having to perform operations on arrays of bits. In the first part of the assignment you will use your knowledge of functions, modular programing, strings and bit-wise operations. In the second part you will update the application by using pointers. An additional challenge is also provided as an optional assignment.

Assignment – Part 1

In the first part of the assignment you create a console application in C. The console application is made of the two following modules:

1) The first module is named *bit_handler* and performs bit handling operations. The module consists of a header file for function declarations and a source file for function definitions. The initial version of the module supports the seven functions described below from Table 2 to Table 6.

2) The second module consists of source file *main.c* and is responsible for executing the console application from a terminal window. This module is intended to test every functions of module *bit_handler*. In the context of this assignment, a test consists in calling a function and writing the results to the terminal. The message in the terminal provides a clear evidence that the function operates as expected. Note: later in the semester we will study testing technique called *unit testing*.

Description	Function returns an 8-bit mask with a given size and shift Example: 'size' 5 and 'shift' 2 gives 'mask' 0b01111100
Input argument(s)	uint8_t size uint8_t shift
Return type	uint8_t

Table 1: Function create_mask_for_byte

Description	Functions applies 'mask' to 'byte' depending on 'mode'. Input 'mode' can be "Set" or "Clear". Function returns 'masked_byte' as exemplified below Examples: 'Mode' -> "Set" --- 'byte' b7 b6 b5 b4 b3 b2 b1 b0 'mask' 0 0 1 1 0 0 0 0 'masked_byte' b7 b6 1 1 b3 b2 b1 b0 'Mode' -> "Clear" --- 'byte' b7 b6 b5 b4 b3 b2 b1 b0 'mask' 0 0 0 0 1 1 1 0 'masked_byte' b7 b6 b5 b4 0 0 0 b0
Input argument(s)	uint8_t byte uint8_t mask char mode[]
Return type	uint8_t

Table 2: Function apply_mask_to_byte

Description	The values of 'high_byte' and 'low_byte' are combined. Functions returns 'word' as exemplified below Example: 'low_byte' 0x1F and 'high_byte' 0xC5 gives 'word' 0xC51F
Input argument(s)	uint8_t low_byte uint8_t high_byte
Return type	uint16_t

Table 3: Function combine_bytes_to_word

Description	Function swaps the nibbles in 'byte'. The function returns 'swapped_nibbles' as exemplified below Example: 'byte' 0xDA gives 'swapped_nibbles' 0xAD
Input argument(s)	uint8_t byte
Return type	uint8_t

Table 4: Function swap_nibbles_in_byte

Description	Function toggles a bit in 'byte'. The index of the toggled bit is given by ‘position’. The function returns ‘output_byte’ as exemplified below Example: 'byte' 0b01100100 and position 2 give ‘output_byte’ 0b01100000
Input argument(s)	uint8_t byte uint8_t position
Return type	uint8_t

Table 5: Function toggle_bit_in_byte

Description	Returns the number of toggled bits between 'byte_one' and 'byte_two' Example: 'byte_one' 0b00000101 and 'byte_two' 0b00001010 returns 4
Input argument(s)	uint8_t byte_one uint8_t byte_two
Return type	uint8_t

Table 6: Function compare_bytes

Assignment - part 2

In this part of the assignment you create a second version of the console application developed in the first part. To do so you update the prototype of each function of module *bit_handler*. The new prototypes make use of a pointer to output the result of the function. Table 7 below gives an example of the updated prototype of function *create_mask_for_byte*. Additionally you update *main.c* accordingly to the change in *bit_handler*.

Description	Function returns an 8-bit mask with a given width and shift Example: 'size' 5 and 'position' 1 gives 'mask' 0b00111110
Input argument(s)	uint8_t size uint8_t position uint8_t* mask
Return type	void

Table 7: Updated function create_mask_for_byte

Additional challenge

In this optional assignment you are challenged to create and test three additional functions for the module *bit_handler*. It is only expected to add three functions and corresponding tests to the console application of the second part of the assignment. The prototypes of the three new functions use a pointer to output the value of the processed array of bits.

Description	Function reverses order of bits of 'long_word'. Example: 'long_word' [b31 b30 ... b1 b0] gives ‘word_reversed’ [b0 b1 ... b30 b31]
Input/output argument(s)	To be defined
Return type	void

Table 8: First additional function

Description	Function sorts nibbles in 'long_word' Example: 'long_word' 0x5AB1F79D returns 'sorted_long_word' 0x1579ABDF
Input/output argument(s)	To be defined
Return type	void

Table 9: Second additional function

Description	Swap odd and even bits in 'long_word' Example: 'long_word' [b31 b30 b29 b28... b1 b0] gives 'swapped_bits' [b30 b31 b28 b29 ... b0 b1]
Input/output argument(s)	To be defined
Return type	void

Table 10: Third additional function

Criteria

- The code should compile without errors and a working executable file should be generated
- The working executable should consists of a C console application
- The bit handling functions should be functioning according to the description above
- The bit handling functions should be implemented using bit-wise operators (&, |, ^, ~, <<, >>)
- The bit handling functions should be optimized. Use least possible amount of operations
- All bit handling functions should be tested in the *main.c* file
- Use pre-processor macros for defining constant values when appropriate
- Use meaningful variable names. All variable names should be easy to understand
- Code should be clean. Use proper indentation of code and consistent coding style
- Code should be commented when relevant to increase legibility
- Code should be modular. A module consists of a source and a header file
- Code should be authentic. Student should be able to explain his/her own code and to justify implementation choices. Plagiarism is not accepted
- Use Microsoft Visual Studio Code and C code only
- Parts 1 and 2 of the assignment are mandatory. The additional challenge is optional and gives extra credits if above criteria are met.

Submission

- A zip file containing your two MS Visual Studio Code projects. After submission you will be given a formative indication of your work following the development-oriented scale in canvas.