This course has already ended.

Eighth round tasks as a `ZIP file.`

This round focuses on the bit-level manipulations, which is often necessary for programming, for example, low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization

Bitmasks¶

Bit masks are used to operate on selected bit or set of bits. State of bits can be investigated using the bitwise AND operator, and bit masks can be combined using the bitwise OR operator. The result of bitwise AND operation is true (non-zero) only if at least one of the indicated bits is set.

The below example investigates state of a few bits, and converts the highest four bits in variable a into integer.

```
#include <stdio.h>

int main(void) {
    unsigned char a = 0x36;  // 00110110
    if (a & 0x2) {
        // Second bit is set, true in this case
    }
    if (a & 0x1) {
        // First bit is set, not true in this case
    }
    if (a & 0xf0) {  // are any of the highest four bits set?
        // convert the highest four bits to integer
        int b = (a & 0xf0) >> 4; // b == 3
        printf("b: %x\n", b);
    }
}
```

Only if the "if" conditions of the above program are true, block of statements inside if loop will be executed.

Programs can use **flags** to represent some state in a system. A flag can either be on or off, i.e., it can be naturally represented with a single bit. Such presentation is also very efficient where space matters: a single char - type variable can store 8 different flags, because it consists of 8 bits. Bitwise operations can be used to evaluate and set the state of individual flags.

The following example operates an imaginary file, that can have four types of permissions separately for the file owner and a "group". The permissions are presented by flags that can either be on or off, in different combinations. A single unsigned char is sufficient for representing read, write, execute and delete permissions, separately for an owner and a group.

```
#include <stdio.h>

typedef unsigned char MyFlags;

// Owner permissions
const MyFlags CanRead = 0x1;
const MyFlags CanWrite = 0x2;
const MyFlags CanExecute = 0x4;
const MyFlags CanDelete = 0x8;

// Group permissions
const MyFlags GroupCanRead = 0x10;      // CanRead << 4
const MyFlags GroupCanWrite = 0x20;     // CanWrite << 4
const MyFlags GroupCanExecute = 0x40;   // CanExecute << 4
const MyFlags GroupCanDelete = 0x80;    // CanDelete << 4

typedef struct {
    const char *name;
    MyFlags perms;
} File;

int main(void) {
    File fileA;
    fileA.name = "File 1";
    fileA.perms = CanRead | CanWrite; // can read and write, but not execute
    printf("flags 1: %02x\n", fileA.perms);

    if (fileA.perms & CanRead) {
        printf("reading is possible\n");
    }
    if (fileA.perms & GroupCanRead) {
        // Group cannot read, so we can't get here
        printf("group reading is possible\n");
    }
    fileA.perms |= GroupCanWrite; // now also group can write
    printf("flags 2: %02x\n", fileA.perms);

    // zeroing CanWrite and GroupCanWrite
    fileA.perms &= ~(CanWrite | GroupCanWrite);

    // print the final state of flags
    printf("flags 3: %02x\n", fileA.perms);
}
```

Task 8.4: Bit operations¶

Objective: In this task, you will learn how to do how to do some basic operations with bitfields and such.

The program deals with the array, which is consist of several bytes. Your task is to implement functions that manipulate individual bits in the array. Please read **Notes** below before starting implementation.

Your task is to implement the following functions:

a) Basic operations

In the following functions, *data* parameter indicates start of the array and *i* parameter denotes position of the bit in the input array.

Implement the following functions which manipulates **bit** of input array:

- `void op_bit_set(unsigned char* data, int i)` – sets a bit in input **data**.
- `void op_bit_unset(unsigned char* data, int i)` – resets a bit in input **data**.
- `int op_bit_get(const unsigned char* data, int i)` – returns value 0, if bit value in **i** is zero and returns value 1, if bit value in **i** is one.

b) Print a byte

Implement function `void op_print_byte(unsigned char b)`, which prints one unsigned char's binary representation.

c) Get a sequence

Implement function `unsigned char op_bit_get_sequence(const unsigned char* data, int i, int how_many)`, that separates a maximum of 8 bits long binary number from the array and returns it. **i** and **data** have same meaning as above. **how_many** indicates how many bits need to be counted from the i (max. 8). If how_many is less than 8, the most significant bits of the returned number is left with zeros. In this task, you may want to take advantage of function implemented in (a).

Notes:

- In this task, bit 0 is the most significant bit. It is also assumed that **unsigned char** is exactly 8 bits (1 byte). Thus, for example, bit 8 is a leftmost bit in second unsigned char byte and bit 17 is the second highest bit in the third unsigned char byte. Thus, examining the number 170 (0xAA hexadecimal format, 10101010 in binary), the most significant bit, ie bit 0 has a value of 1.
- If you find yourself implementing some sort of helper array of characters or integers, then you are doing something wrong.
- Bit 5 from an array is not array[5].
- If given binary data is `1110 0101 1111 0011 0001 1110 0100 1111`, and the `op_bit_get_sequence` function is called with index 20, and how_many of 5, return value should be 28, i.e. 5 first bits of the number `1110 0100 1111` from the 20th index i.e. `1 1100`. The return value should then be `0001 1100` (`0001 1100` = 28).

Points **30 / 30** My submissions **3** ▾ Deadline Friday, 30 July 2021, 19:59
To be submitted alone

⚠ This course has been archived (Saturday, 31 December 2022, 20:00).

Bit Operations

Select your files for grading

bits.c

Choose File No file chosen

Submit

Task 8.5: TCP header¶

Objective: Practice binary operations.

Low layer communication protocols aim to utilize the available space in communication efficiently, to reduce the amount of needed network traffic. We will take a look at the TCP protocol as an example of this.

You can find information about the TCP protocol and TCP header here. To implement this task, you will need information about the structure of the TCP header, as described in the wikipedia page linked above.

You can read the TCP header diagram as follows: each row in the diagram represents 4 bytes (32 bits) in the header. The byte count is shown on the top of the diagram, and on the left side of the diagram, and by adding these together you will what is the byte offset from the beginning of the packet for the particular header field. For example, the *checksum* field can be found at 16th and 17th byte. If you think the TCP header as a byte array, you will find the *checksum* field at `tcp_hdr[16]` and `tcp_hdr[17]`.

Additional tips:

- Pay attention to the precedence of C operators regarding the binary operations. The bit shift operators `<<` and `>>` are evaluated before arithmetic operatos (such as '+' and '-'). Use parentheses as needed.

a) Parse header¶

Implement the following functions that each read and return one field from the TCP header. In order to parse the fields, you will need to find the respective bits in the TCP header, and leverage bitwise operations to represent the value as a single integer return parameter.

All functions get the pointer to the beginning of TCP header as a parameter.

- `getSourcePort` returns the source port. This is a 16-bit value, but you will need to process each byte separately. I.e.: if header[0] is 0xac, and header[1] is 0xde, you will need to return integer 0xacde. For this you will need to apply bitwise operations.
- `getDestinationPort` returns the destination port. The same note applies here as above regarding handling of 16-bit values.
- `getAckFlag` returns the value of ACK flag (either 0 or 1). N.B. ACK is in byte 13.
- `getDataOffset` returns the length of the header (i.e., the data offset)

b) Write header¶

Implement the following functions to produce parts of a TCP header. Each function gets an integer value as a parameter, that should be placed in right location in TCP header.

- `setSourcePort` that sets the source port field in TCP header as indicated in the *port* parameter. You will need to handle each byte separately: the most significant eight bits are placed at header[0], and least significant eight bits at header[1].
- `setDestinationPort` that sets the destination port field as indicated in the *port* parameter. The byte order is as described above.
- `setAckFlag` that sets the ACK bit on or off depending on whether the function argument is 0 (off) or non-zero (on).
- `setDataOffset` that sets the data offset field.

In each function, only the given TCP header field can change, and all other parts of the header must remain unchanged.

Points **20 / 20** My submissions **15** ▾ Deadline Friday, 30 July 2021, 19:59
To be submitted alone

⚠ This course has been archived (Saturday, 31 December 2022, 20:00).

TCP Header

Select your files for grading

tcpheader.c

Choose File No file chosen

Submit

Task 8.6: XOR cipher¶

Objective: to get familiar with bitwise operation with longer data types.

Implement function **confidentiality_xor** that gets a 32-bit encryption key ('key') as argument, the encrypted data ('data') along with its length ('len') as arguments. The length is indicated as the number of 32-bit blocks the data contains. The function should implement a simple encryption: each 32-bit data buffer will be encrypted using the XOR-bitwise operation using the encryption key. The function does not need to allocate memory, i.e., it operates on the data buffer directly. **Note:** the data should be handled as 32-bit unsigned integers (`uint32_t` data type in stdint.h header). More information in Wikipedia.

Implement also function **confidentiality_xor_shift** that, in addition to encrypting the data, will modify the encryption key after each 32-bit block. The function works otherwise similarly as the above one, but after each operation it shifts the bits in the key one step left. At each shift, the most significant (leftmost) bit it transferred to the other end, i.e. to represent the least significant (rightmost) bit. The modified key will be used on next encryption block, after which it is again modified, and so on.

Points **20 / 20** My submissions **2** ▾ Deadline Friday, 30 July 2021, 19:59
To be submitted alone

⚠ This course has been archived (Saturday, 31 December 2022, 20:00).

XOR Cipher

Select your files for grading

xorcipher.c

Choose File No file chosen

Submit