Bitwise Logical Operations I

- Complement operator $\sim$:

$$\sim (00000000) == (11111111).$$

- AND operator $\&$:

$$(01010101)\&(10101011) == (00000001).$$

- Inclusive OR $|$:

$$(01010101)|\,(10101011) == (11111111).$$

- Exclusive OR $^\sim$:

$$(01010101)^\sim(10101011) == (11111110).$$
Bitwise Logical Operations II

- **Left Shift \( << \):**

  \[(00000010) \ll 2 = (00001000).\]

- **Right Shift \( >> \):**

  \[(00001000) \gg 2 = (00000010).\]
struct {
    unsigned Yorn : 1;
    unsigned status : 2;
} bit fields;

declares a structure variable that contains 2 bit field members: Yorn is 1 bit, status is 2 bits.

bit fields.Yorn = 1; (only 0 and 1 can be used for assignment value)
bit fields.status = 2; (0,1,2,3 can be used for assignment value)
The following example shows different integer types which are use in bit manipulations.

```c
#include <stdio.h>
void main(void)
{
    unsigned ui, xi;
    unsigned short sui;
    unsigned long lui;

    printf("\n Please enter an unsigned int: ");
    scanf("%u", &ui);
    printf(" = %u in decimal and = %x in hex. \n", ui,ui);
    printf("\n Please enter an unsigned int in hex: ");
```
Example of various numerical types II

```c
scanf("%x", &xi);
printf(" = %u in decimal and = %x in hex. \n", xi, xi);

printf(" short and long unsigned ints: ");
scanf("%hu%lu", &sui, &lui);
printf(" short in hu = %hu in hx = %hx\n", sui, sui);
printf(" long in lu = %lu in lx = %lx\n", lui, lui);

printf(" Error: short unsigned in hi format = %hi\n", sui);
printf(" Error: long unsigned in li format = %li\n", lui);
}
```
Note also the library stdint, where there are types of the form `intN_t` or `uintN_t`. In other words, you can explicitly show the number of bits you are going to use. Usually, $N$ is 8, 16, 32 or 64.
Decoding an Internet address

In the next example the program reads a 32-bit Internet address in the form used to store addresses internally and print it in the four-part dotted form that we customarily see.

Thus input could be fa1254b9 in hexadecimal form and the program prints 250.18.84.185.

We use a common technique in bit manipulations: mask. Using bitwise operations, mainly &, and a mask we can take part of a bit string. For example, if we have a bit string

\[ b = 10101010101010101010101010101010, \]

and a mask

\[ m = 11111111000000000000000000000000, \]
then

\[ m \& b \]

forms a bit string

\[ 10101010000000000000000000000000. \]

Thus we have the first eight bits as in \( b \) and the rest are zero.

- The mask in the example is needed, if a machine is a 64-bit machine. If all the machines were 32-bit machines, the mask would not be needed.
#define BYTEMASK 0xffL /* L to make a long integer */
#include <stdio.h>

void main( void )
{
    unsigned long ip_address;
    unsigned f1, f2, f3, f4;

    printf("Please enter an IP address as 8 hex digits: ");
    scanf("%lx", &ip_address);
    printf("You have entered %08lx\n", ip_address);

    f1 = ip_address >> 24 & BYTEMASK;
    f2 = ip_address >> 16 & BYTEMASK;
}
f3 = ip_address >> 8 & BYTEMASK;
f4 = ip_address & BYTEMASK;

printf("The IP address in standard form is: ");
printf("%i.%i.%i.%i \n\n", f1, f2, f3, f4 );
}
In modern cryptography, before the actual encryption a plain text is shuffled in many ways in order to reduce regularities. The following program permutes the bytes of a plain text. This is done using a fixed permutation.

Consider a 16-bit block of a plain text. Divide that block into four unequal-length fields of 3, 5, 4, and 4, respectively, and permute those fields such that

- the first field becomes the second,
- the second the fourth,
- the third the first,
- the fourth the third.
#include <stdio.h>

#define AE 0xE000
#define BE 0x1F00
#define CE 0x00F0
#define DE 0x000F

unsigned short encrypt (unsigned short n);
void main ( void )
{
    short in;
    unsigned short crypt;
    printf("\n Enter a short integer to encrypt: ");
    scanf("%hi", &in);
    /* Cast the int to unsigned before calling encrypt. */
crypt = encrypt( (unsigned short) in);

    printf("\n The input number in base 10 is: %hi \n"
    " The input number in hexadecimal is: %hx \n\n"
    " The encrypted number in base 10 is: %hu \n"
    " The encrypted number in base 16 is: %hx \n\n",
    in, in, crypt, crypt );
}
unsigned short encrypt (unsigned short n )
{
    unsigned short a, b, c, d;

    a = (n & AE) >> 4; /* Isolate bits 0:2, shift to 4:6 */
    b = (n & BE) >> 8; /* Isolate bits 3:7, shift to 11:15 */
    c = (n & CE) << 8; /* Isolate bits 8:11, shift to 0:3 */
    d = (n & DE) << 5; /* Isolate bits 12:15, shift to 7:10 */

    return c | a | d | b;
}

- An artist has a studio with a high sloping ceiling containing skylights. Outside, each skylight is covered with louvers that can be opened fully under normal operation to let the light in or closed to protect the glass or keep heat inside the room at night.

- The louvers are opened and closed by a small computer-controlled motor with two limit switches that sense when the skylight is fully open or fully closed. To open the skylight, one runs the motor in the forward direction until the fully open limit switch is activated. To close the skylight, one runs the motor similarly in the reverse direction. To know the current location of the skylight, one simply examines the state of the limit switches.
The motor is controlled by a box with relays and other circuitry for selecting its direction, turning it on and off, and sensing the state of the limit switches. The controller box has an interface to the computer through a multifunction chip using a technique known as *memory-mapped I/O*. This means that when certain main memory addresses are referenced, bits are written or read from the multifunction chip, rather than real, physical memory.

In this program, we assume that the multifunction chip interfaces with the computer through two memory addresses: 0xffff7100 refers to an 8-bit data register (DR) and 0xffff7101 refers to an 8-bit data direction register (DDR). Each bit of the data register can be used to send data either from the chip to the program or vice versa. Data flows from chip to program through a bit if the corresponding bit of the DDR is 0 and from program to chip if the corresponding bit is 1.
Certain bits of the DR then are wired directly to the skylight controller box as shown in the specification below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>In/Out</th>
<th>Purpose</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>out</td>
<td>Motor direction</td>
<td>0=forward</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1=reverse</td>
</tr>
<tr>
<td>1</td>
<td>out</td>
<td>Motor power</td>
<td>0=off</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1=on</td>
</tr>
<tr>
<td>2</td>
<td>in</td>
<td>Fully closed louver sensor</td>
<td>0=not fully closed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1=fully closed</td>
</tr>
<tr>
<td>3</td>
<td>in</td>
<td>Fully open louver sensor</td>
<td>0=not fully open</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1=fully open</td>
</tr>
</tbody>
</table>

We start to write declarations for the skylight controller:

We use four bits to communicate with the multifunction chip; two are used by the program to receive status information from the chip and two are used to send control instructions to the chip. The leftmost four bits in
the chip registers will not be used in this application. Therefore, the bitfield type declaration used to model a register begins with unnamed field for the four padding bits, followed by named fields for two status bits and two control bits.

```c
typedef struct REG_BYTE {
    unsigned int :4;
    unsigned int fully_open :1;
    unsigned int fully_closed :1;
    unsigned int motor_power :1;
    unsigned int motor_direction :1;
} reg_byte;

typedef volatile reg_byte * device_pointer;
```
The key word **volatile** means that something outside the program (in this case, the controller box) may change the value of a register byte at unpredictable times. We supply this information to the C compiler so that its code optimizer does not eliminate any assignments or reads from the location that, otherwise, would appear redundant.

Next follows the definitions of the codes for the multifunction chip:

```c
enum power_values {motor_off = 0, motor_on = 1};
enum direction_values {motor_forward = 0, motor_reverse = 1};
char * position_labels[] = {
    "fully closed", "partially open",
    "fully open"};
typedef enum {fully_closed, part_open, fully_open} position;
```
A Device Controller VI

We have defined three enumerated types to give symbolic names to the various switch settings and status codes. An array of strings is defined to allow easy output of the device status.

Two of the enumerations are not within a `typedef`. They are used simply to give names to codes. A series of `#define` commands could be used for this purpose, but `enum` is better because it is shorter and it lets us group the codes into sets of related values.

Next we want a pointer variable `DR` to point at the address of the data register byte on the multifunction chip. We write its memory-mapped address as a hex literal, cast it to the appropriate pointer type, and store it in `DR`. The keyword `const` after the type name means that `DR` always points at this location and can never be changed.

A bitmask is used when the program is started to initialize the data direction register. The rightmost two bits are used to send control
information to the chip from the program, while the other two will be read by the program to check the chip’s status.

device_pointer const DR = (device_pointer)0xffff7100;
device_pointer const DDR = (device_pointer)0xffff7101;
const reg_byte DDR_mask = {0,0,1,1};
const reg_byte DR_init = {0,0,0,0};

The main program uses the following functions:

position skylight_status ( void );
void open_skylight ( void );
void close_skylight ( void );

And the main program is:
void main (void )
{
    char choice;
    char * menu[] = {"O: Open skylight", "C: Close skylight", 
                    "R: Report position", "Q: Quit"};

    *DDR = DDR_mask;
    *DR = DR_init;

    for (; ; ) {
        choice = toupper( menu_c( " Select operation:" , 4 , menu) );
        if (choice == 'Q') break;
        switch (choice) {
        case 'O': open_skylight(); break;
        case 'C': close_skylight(); break;
        case 'R': /* Report on position */
        }
printf( "Louver is \%s\n",
    position_labels[skylight_status()] );
    break;
    default: puts("Incorrect choice, try again.");
    }
    }
    puts(" Skylight controller terminated.\n");
}

We use functions

    menu_c()     and     toupper()

in the main program. The former displays a menu and reads a selection. The latter is used to recognize both lower-case and upper-case letters. We do not show code for these interface functions.
position skylight_status( void )
{
    if (DR->fully_closed) return fully_closed;
    else if (DR->fully_open) return fully_open;
    else return part_open;
}
void open_skylight( void )
{
reg_byte dr = {0, 0, motor_on, motor_forward };

if (DR->fully_open) return;

*DR = dr;

while (!(DR->fully_open)); /* delay until open */

    dr.motor_power = motor_off;

    *DR = dr;
}
A Device Controller XII

```c
void close_skylight( void )
{
  reg_byte dr = { 0, 0, motor_on, motor_reverse };

  if (DR->fully_closed) return;

  *DR = dr;

  while (!(DR->fully_closed)); /* delay until closed */

  dr.motor_power = motor_off;
  *DR = dr;
}
```