Practical Case Study A

Operating Systems Programming – 300698
Operating Systems Programming (Advanced) – 300943

1 Introduction

Operating Systems have a need for extremely compact data structures, as often these need to be stored wholly in memory. Examples of this are free memory lists, page tables and disk space bitmaps. This Practical Case Study will develop and refresh your knowledge of bit operations, necessary to manipulate such compact data-structures.

2 Bit Operations

The C programming language provides a full set of bit manipulation operators:

- `&` bitwise and
- `|` bitwise or
- `^` bitwise exclusive or
- `~` ones compliment
- >> right shift
- << left shift

These operators can be used to manipulate values at the bit level. They each treat the values as if they were 'arrays' of bits and applies the operation to each bit in turn. There are four main operations that we perform, setting a bit, unsetting a bit, testing if a bit is set and toggling a bit.

To set a bit we use the *bitwise or* operator. To do this we *bitwise or* our value with a *mask* with the bits we want to turn on set to 1 and all others 0. For example to turn on the third bit our mask would be 00000100_2 (the subscript $_2$ means that the number is expressed in binary, no subscript means decimal).

To unset a bit we use the *bitwise and* operator. To do this we *bitwise and* our value with a *mask* with the bits we want to unset set to 0 and all others 1. For example to unset the sixth bit our mask would be 11011111_2. We can also use the *ones compliment* operator to invert a mask used to set a bit.

To test if a bit is set we use the *bitwise and* operator. To do this we *bitwise and* our value with a *mask* with the bits we want to test set to 1 and all others 0. For example to test if the fifth bit is set our mask would be 00010000_2.

To toggle a bit we use the *bitwise exclusive or* operator. To do this we *bitwise exclusive or* our value with a *mask* with the bits we want to toggle set to 1 and all others 0. For example to toggle the fourth bit our mask would be 00001000_2.

Binary	Hexadecimal	Octal	Decimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	10	8
1001	9	11	9
1010	A	12	10
1011	B	13	11
1100	C	14	12
1101	D	15	13
1110	E	16	14
1111	F	17	15

Table 1: Values for 1-16

We can use the *left shift* operator to construct *masks*, by shifting the value 1 the required number of positions.

The shift operators can also be used, with masking to extract subfields or to encode subfields.

3 Number Systems

In the previous section various masks were presented, however earlier versions of the C programming language don't allow numbers to be expressed in binary. C traditionally allows three styles of number: decimal, octal and hexadecimal. Decimal numbers are the numbers we use every day.

Octal numbers are a base 8 system. To specify to the C compiler that a number is octal you add the prefix `0`. For example the decimal number 23 would be written in octal in a C program as `027`. The advantage of octal numbers is that each digit represents exactly three bits. Octal numbers are often used for specifying permissions in Unix.

Hexadecimal numbers are a base 16 system, the letters a-f or A-F are used for the extra positions. To specify to the C compiler that a number is hexadecimal you add the prefix `0x`. For example the decimal number 23 would be written in hexadecimal in a C program as `0x17`. The advantage of hexadecimal numbers is that each digit represents exactly four bits. Hexadecimal numbers are often used for specifying *masks* and memory addresses.

Table 1 shows equivalent values for the numbers 1-16.

4 Programming Tasks

You should begin by reading the example code carefully. There are a number of files that are included in the download. The `makefile` contains the rules to build all of the questions code. You Simply need to type `make` in the directory that you downloaded the code to to compile it all, and if you have made no changes to the file then running make a second time will say everything is up to date. If you wish to remove all of the compiler generated files then run `make clean`. The code supplied on the vUWS site will compile, it just wont do anything useful until you edit it.

In this task we will use the `uint8_t` type that is defined in the header file `stdint.h`. This type is an unsigned 8 bit integer.

4.1 Print a value in binary

Your first task to to implement the function `print_bits()`. This function takes a single `uint8_t` value as a parameter and prints to `stdout` the value of the parameter in binary, starting with the most significant bit. It must always print out all 8 bits, so the value 1_{10} should be displayed as `00000001` and the value 16_{10} should be displayed as `00010000`. The file `part1.c` is the start of some testing for this function, you may wish to add more to this file if you wish.

4.2 Reverse the bits in a byte

Your next task is to implement the function `reverse_bits()` This function takes a single `uint8_t` value as a parameter and returns the value with its bits reversed. So `10000000` would be transformed to `00000001` and `10101000` would be transformed to `00010101`. The file `part2.c` is the start of some testing for this function, you may wish to add more to this file if you wish.

4.3 Check if a bit is turned on

Your next task is to implement the function `check_bit()` This function takes a two `uint8_t` values as a parameters, the first the value to check and the second the bit that you are iterested in. The function will return true if the bit is turned on in the value and false otherwise.

The file `part3.c` is the start of some testing for this function, you may wish to add more to this file if you wish.

4.4 Toggle a bit in a value

Your next task is to implement the function `toggle_bit()` This function takes a pointer to a `uint8_t` value as the parameter to modify and a `uint8_t` value for the bit oyu are interested in. The function will update the value stored at the pointer toggling the bit you are interested in, i.e. if the bit is on turn it off, if it is off turn it on.

The file `part4.c` is the start of some testing for this function, you may wish to add more to this file if you wish.

4.5 Extract a subset of bits from a value

Your next task is to implement the function `get_subfield()` This function takes a three `uint8_t` value as a parameters, the first is the value, the second is the first bit you are interested in and the third is the last bit you are interested in. You should mask off the bits that you are interested in and then shift them to the least significant position, i.e. the first bit you are interested in should bbecome bit 0.

The file `part5.c` is the start of some testing for this function, you may wish to add more to this file if you wish.

5 Sample Code

This sample code is also available at the unit website. You may not need all the variables declared in this code and you may choose to add more.

5.1 `bits.h`

The headerfile that describes the functions to be implemented in this task, you should not need to modify this file.

```c
#ifndef OSP_BITS_H
#define OSP_BITS_H
/* The above lines prevent multiple inclusion problems */

void print_bits(uint8_t value);

uint8_t reverse_bits(uint8_t value);

uint8_t check_bit(uint8_t value, uint8_t bit);

uint8_t toggle_bit(uint8_t *value, uint8_t bit);

uint8_t get_subfield(uint8_t value, uint8_t start, uint8_t stop);

#endif
```

5.2 bits.c

```
#include<stdio.h>
#include<stdint.h>
#include"bits.h"

void print_bits(uint8_t value)
{

}

uint8_t reverse_bits(uint8_t value)
{

}

uint8_t check_bit(uint8_t value, uint8_t bit)
{

}

uint8_t toggle_bit(uint8_t *value, uint8_t bit)
{

}

uint8_t get_subfield(uint8_t value, uint8_t start, uint8_t stop)
{

}
```

5.3 part1.c

```c
#include<stdio.h>
#include<stdint.h>
#include"bits.h"


int main(int ac, char **av)
{
        uint8_t val=0b10101010;

        print_bits(val);

        return 0;
}
```

5.4 part2.c

```
#include<stdio.h>
#include<stdint.h>
#include"bits.h"


int main(int ac, char **av)
{

        uint8_t val=0;
        uint8_t rev=0;

        for(val=0;val<255;val++)
        {
                printf("val:%u\n",val);
                print_bits(val);
                putchar('\n');
                rev = reverse_bits(val);
                print_bits(rev);
                putchar('\n');
        }

        return 0;
}
```

5.5 part3.c

```
#include<stdio.h>
#include<stdint.h>
#include"bits.h"


int main(int ac, char **av)
{
        uint8_t val=0b10101010;
        uint8_t bit;

        print_bits(val);
        putchar('\n');

        for(bit=0;bit<8;bit++)
        {
                printf("Bit %u is",bit);
                if(check_bit(val,bit))
                {
                        printf(" on\n");
                }
                else
                {
                        printf(" off\n");
                }
        }

        return 0;
}
```

5.6 part4.c

```
#include<stdio.h>
#include<stdint.h>
#include"bits.h"


int main(int ac, char **av)
{

        uint8_t val = 0;
        uint8_t bit;

        for(bit=0;bit<8;bit++)
        {
                printf("val:%u, bit:%u\n",val,bit);
                print_bits(val);
                putchar('\n');
                toggle_bit(&val,bit);
                print_bits(val);
                putchar('\n');
        }

        for(bit=0;bit<8;bit++)
        {
                printf("val:%u, bit:%u\n",val,bit);
                print_bits(val);
                putchar('\n');
                toggle_bit(&val,bit);
                print_bits(val);
                putchar('\n');
        }

        return 0;
}
```

5.7 part5.c

```
#include<stdio.h>
#include<stdint.h>
#include"bits.h"


int main(int ac, char **av)
{
        uint8_t val=0b10101010;

        print_bits(val);
        putchar('\n');

        print_bits(get_subfield(val,0,2));
        putchar('\n');

        print_bits(get_subfield(val,1,3));
        putchar('\n');

        return 0;
}
```

5.8 makefile

Please note that formatting is very important in a makefile, and that the character before rm on line 6 must be a tab character.

```
PROGS=part1 part2 part3 part4 part5

ALL: ${PROGS}

clean:
	rm -f *.o ${PROGS}

part1: part1.o bits.o

part2: part2.o bits.o

part3: part3.o bits.o

part4: part4.o bits.o

part5: part5.o bits.o

bits.o: bits.c bits.h
```