```c
#include "sfp.h"
#include <stdlib.h>
```

기본 포함해준 헤더파일이다.

```c
float fabs(float a) {
    if (a < 0) {
        return -a;
    }
    else {
        return a;
    }
}

int abs(int a) {
    if (a < 0) {
        return -a;
    }
    else {
        return a;
    }
}
```

Fabs, abs함수를 통해 절댓값을 계산하기 위해 함수를 만들어 주었다.

```c
typedef unsigned short sfp;
#define bias 15 //bias 크기는 2^5-1 -1 (exp bit는 5개)
```

Bias의 크기를 15로 설정하였다.

```c
sfp int2sfp(int input){
    sfp newSfp = 0;
    long long binary = 0, temp_binary,ten =1;
    int E = 0, exp, i, index = 32768, count = -1, bin_count=0, sign=0;
    int frac = 0, frac_i = 1;

    if (input == 0) return 0;//0처리
    if (input > 65504) return 31744; //+무한대 처리
    if (input < -65504) return 64512; //-무한대 처리

    if (input > 0) index /= 32;
    else { //음수 처리
        newSfp += index;
        index /= 32;
        input *= -1;
        sign = 1;
    }

    while (1) { //binary
        binary += input % 2 * ten; //binary라는 int에 10진수로 2진수를 표현 ex) 1101
        ten *= 10;
        bin_count++; // binary가 몇자리 수인지 체크
        input /= 2;
        if (input == 0) {
            break;
        }
    }
```

Input의 크기에 따라, sfp가 담을 수 있는 범위를 넘어가면 무한대 처리를 하였다. 0일 경우에는 단순하게 0을 바로 반환하였다.

음수일 때에는 index를 더함으로써 부호처리를 하였다. Index는 16bit의 맨 앞자리부터 시작하는, 16bit에서 현재 위치를 나타내는 변수이다.

그 후 binary라는 int안에, 마치 2진법인 것 처럼 input을 계산하여 담았다.

처음엔 binary를 int로 설정하였었다. 하지만, rount-to-zero를 해야하는 경우, overflow가 발생하는 것을 발견하여 long long 으로 자료형을 바꾸어주었다.

```
if (bin_count > 11) { //round-to-zero 과정
    while (bin_count != 11) {
        bin_count--;
        binary /= 10;
    }
    if (sign = 1) {
        binary += 1;
    }
}
```

Sfp가 허용가능한 frac범위를 넘어가면 round-to-zero를 진행하였다.

```
temp_binary = binary;

while ((temp_binary /= 10) != 0) { //E 계산
    E++;
}

exp = E + bias; //exp 계산
newSfp += (exp * index); // exp부분 설정
temp_binary = binary;
ten = 1;

while (temp_binary >=10 ) { //자릿수 계산위해 temp 사용, ten을 통해 자릿수 파악
    temp_binary /= 10;
    ten *= 10;
    count++;
}

for (i = 0; i < count+1; i++) { //찾은 자릿수만큼 index 설정
    index /= 2;
}

binary -= ten; // 1.xxx 형태이기에 1 제거

while (binary != 0) { //frac 부분 설정
    frac += binary % 10 * frac_i;
    frac_i *= 2;
    binary /= 10;
}
```

binary자체는 남겨놓기 위해 temp_binary를 사용하여 E를 계산하였고, E를 바탕으로 exp를 계산하였다. 그 전에 설정한 index를 통해 sfp에서의 exp부분의 값을 할당하였다.

그 후 binary에 몇자리의 수가 있는지를 파악하였으며, 그 자릿수만큼 index를 내리게 되었다. 이

는 frac부분을 바로 할당해주기 위함이다. Ten 만큼 제거해주는 이유는 frac은 exp가 00000이 아닌 이상 1.xx의 형태로 존재하기 때문이다. 그 후 binary를 통해 frac부분의 값을 설정해주었다.

```
    newSfp += frac * index;

    return newSfp;
}
```

구한 자릿수인 index를 곱함으로써 sfp에 바로 frac값을 할당할 수 있었다. 여기까지가 int2sfp함수이다.

```
int sfp2int(sfp input){
    int binary = 0, index = 15, ten = 1,two=1, newInt = 0, binary_count=0;
    int E, exp=0;

    if (input == 0) return 0;//0처리
    if (input == 31744) return 2147483647; //+무한대 처리
    if (input == 64512 || (input/1024)%32 == 31) return -2147483647; //-무한대 처리

    while (input % 2 == 0) {//1의 위치 찾는것
        input /= 2;
        index--;
    }

    while (index != 5) { // frac 처리하는것
        binary += input % 2 * ten;
        binary_count++;
        input /= 2;
        ten *= 10;
        index--;
    }
    binary += ten;

    while (index != 0) { // exp 처리하는것
        exp += input % 2 * two;

        input /= 2;
        two *= 2;
        index--;
    }
    E = exp - bias; // E 설정
```

마찬가지로 처음 부분에서 0, 무한대 처리를 해준다. 무한대인 경우에는 TMax와 TMin값을 할당한다. Index는 아까와 같이 16비트에서의 자리를 나타내주는 변수이다. 처음 나오는 frac에서의 1을 찾기 위해 while문을 사용하였다. 그 후 frac을 처리하여 binary에 값을 할당하였다. 남은 input 값에서 exp를 계산하였으며, 구한 exp를 통해 E값을 구하였다.

```
if (E < binary_count - 1) { // round-to-zero 과정
    while (!(E == binary_count - 1)) {
        binary /= 10;
        binary_count--;
    }
}
two = 1;
```

이 때 소수 부분이 존재한다면 rount-to-zero 과정을 진행하였다.

```
    while (binary != 0) { //binary를 통해 int값 설정
        newInt += binary % 10 * two;

        binary /= 10;
        two *= 2;
    }

    if (input == 1) newInt = -newInt; //음수일 경우 -부호 처리

    return newInt;
}
```

그렇게 구한 binary값을 통해 int값으로 변환해주었고, 마지막에 음수일 경우 부호처리를 해주었다. 여기까지가 sfp2int 함수이다.

```
sfp float2sfp(float input){
    sfp newSfp=0;
    int index = 32768, i=0, find_one = 0, integer = 0, integer_bin = 0, integer_count = 0;
    int bin_i = 0, two = 1, ten = 1, frac_i, frac = 0, E = 1, integer_save, exp, sign =0;
    int binary[16];

    if (fabs(input)< 0.000001) return 0;//0처리
    if (input > 65504) return 31744; //+무한대 처리
    if (input < -65504) return 64512; //-무한대 처리

    memset(binary, 0, 16*sizeof(int)); //0으로 초기화

    if (input < 0) { //음수 처리
        input *= -1;
        newSfp += index;
        sign = 1;
    }
    index /= 32;

    if (input >= 1) { // integer부분의 크기가 몇인지 파악
        while (input >= 1) {
            input -= 1;
            integer++;
        }
    }
    integer_save = integer;
```

마찬가지로 처음 부분에서 0과 무한대 처리를 해주었으며, float에서는 int와 달리 binary배열을 사용하였다. 소수점의 위치를 자유자재로 설정하고 파악하기 위함이다. 음수일 경우 음수 처리를 하였으며, 정수값 부분이 존재할 경우 몇인지를 파악하였다. Integer은 추후에 사용하기 위해 integer_save변수를 도입하였다.

```
if (integer != 0) { //integer부분 binary로 변환
    while (integer != 0) {
        integer_bin += (integer % 2) * ten;
        ten *= 10;
        integer /= 2;
        integer_count++;
    }
}
for (i = integer_count - 1; i >= 0; i--) {//변환한 binary를 총 binary에 삽입
    binary[i] = integer_bin % 10;
    integer_bin /= 10;
}
bin_i += integer_count;
```

정수부분을 먼저 binary로 변환하여 배열에 저장하였다.

Integer_count는 정수부분의 binary 자릿수가 몇이나 되는지 파악하는 것이다.

```
while (1) { //소수 부분의 binary 설정 과정
    input *= 2;

    if (input > 1) {
        input -= 1;
        binary[bin_i] = 1;
    }
    else {
        binary[bin_i] = 0;
    }

    if (fabs(input - 1) < 0.001) { // 이진법으로 전환한 것 배열에 저장함
        binary[bin_i] = 1;
        bin_i++;
        break;
    }
    bin_i++;
}
```

그 후, 남은 소수 부분을 배열에 할당하였다.

```
if (bin_i > 10) { //round-to-zero 과정
    while (bin_i != 10) {
        binary[bin_i] = 0;
        bin_i--;
    }

    if (sign == 1) {
        if (binary[bin_i] == 0) {
            binary[bin_i] = 1;
        }
        else {
            binary[bin_i] = 0;
            int i = 9;
            while (binary[i] != 0) {
                binary[i] = 0;
                i--;
            }
            binary[i] = 1;
        }
    }
}
```

Frac이 가질 수 있는 크기인 10만큼을 초과할 경우에 round-to-zero를 진행하였다.

```
while (binary[find_one] != 1) { //if 정수부분 존재하면 그대로 사용, 정수 0이면 1나올때까지 찾음
    find_one++;
    E++;
}
find_one++;
```

binary에서 정수부분이 존재한다면 그대로 사용할 것이고, 정수가 0이라면 1.xxx형태로 만들어주기 위해 find_one 변수를 도입하였다.

```
if (E > 1 || integer_save == 0) { //E값 설정
    E *= -1;
    frac_i = bin_i + E;
}

if (integer_count > 0) { //E값 설정
    E = integer_count - 1;
    frac_i = bin_i - find_one;
}

exp = E + bias; //exp값 설정
newSfp += index * exp; //Sfp에 exp부분 삽입
two = 1;

for (i = 1; i < frac_i; i++) { //자릿수 설정
    two *= 2;
}
index /= two*2;
```

정수값이 존재할 때, 하지않을 때로 구분하여 E값을 설정하였고, E를 통해 exp값을 설정하였다. 구한 exp값을 sfp에 할당하였다. 그 후 frac의 자릿수를 설정하였다.

```
    while (two != 0) { // frac부분 설정
        if (binary[find_one++] == 1) {
            frac += two;
        }
        two /= 2;
    }
    newSfp += index * frac; //Sfp에 frac부분 삽입

    return newSfp;
}
```

남은 frac부분의 값을 설정하고 sfp에 삽입하여 최종 sfp를 반환하였다. 여기까지가 float2sfp함수이다.

```
float sfp2float(sfp input){
    int binary[16];
    int index = 15, two = 1, i=0,ten =1,sign= 0;
    int E, exp = 0, frac_i = 0,point;
    float frac_two = 1.0, newFloat = 0.0;

    memset(binary, 0, 16 * sizeof(int)); //binary 초기화

    if (input > 32768) { //크기를 통해 음수인지 파악
        sign = 1;
    }

    while (index != 5) { // frac 처리하는것
        binary[index] += input % 2;
        input /= 2;
        index--;
        frac_i++;
    }

    binary[index] = 1;
    int find_one = index;

    while (index > 0 ) { // exp 처리하는것
        exp += input % 2 * two;

        input /= 2;
        two *= 2;
        index--;
    }
    E = exp - bias; //E값 설정
    point = find_one + E; // 소수점 위치 파악
    two = 1;
```

마찬가지로 소수점이 사용되기에 binary배열을 사용하였으며, 음수일 경우 sign으로 음수임을 표시하였다. Frac을 처리한 후에 exp를 처리하여 E값을 설정하였다. 1의 값의 위치와 E를 통해 소수점 위치를 파악하여 point변수에 저장하였다.

```
    for (i = point; i >= 0; i--) {//정수부분 삽입
        newFloat += two * binary[i];
        two *= 2;
    }
    frac_two = 0.5;

    for (i = point + 1; i < 16; i++) {//소수부분 삽입
        newFloat += frac_two * binary[i];
        frac_two *= 0.5;
    }

    if (sign == 1) newFloat = -newFloat; //-부호 처리

    return newFloat;
}
```

정수부분을 먼저 계산하고 소수부분을 계산하였다. 아까 구한 sign을 통해, 음수일 경우 음수처리를 하였다. 여기까지가 sfp2float함수이다.

```
sfp sfp_add(sfp a, sfp b){
    int a_exp = (a / 1024) % 32;
    int b_exp = (b / 1024) % 32;
    int exp_change = 0, round= 0, b_save;
    int two = 1,i, frac, a_sign = 0, b_sign = 0, sign = 0;
    sfp result = 0;
    int exp_differ;

    if (a_exp == 31 && a < 32768) { //a가 +무한대일때
        if (b_exp == 31 && b > 32768) { //b도 +무한대가 아니라면
            return 31744 + 1; //NaN 반환
        }
        return 31744; //+무한대 반환
    }
    else if (a_exp == 31 && a > 32768) { //a가 -무한대일때
        if (b_exp == 31 && b < 32768) { //b도 -무한대가 아니라면
            return 31744 + 1; // NaN 반환
        }
        return 64512; //-무한대 반환
    }
    else if ((a_exp == 31 && a / 1024 > 0) || (b_exp == 31 && b / 1024 > 0)) {//둘 중 하나가 NaN이라면
        return 31744 + 1; //NaN 반환
    }
```

a와 b가 무한대, NaN인 경우를 가장 먼저 처리하였다.

```
if (a_exp < b_exp) {
    sfp temp = a;
    a = b;
    b = temp;
    temp = a_exp;
    a_exp = b_exp;
    b_exp = temp;
}
if (a > 32768) a_sign = 1; //크기 통해 부호 파악
if (b > 32768) b_sign = 1; //크기 통해 부호 파악
```

항상 a가 크도록 만들어주기 위해 if문을 도입하여 더 큰 값이 a가 되도록 바꾸어주었다. 그 후 a와 b의 부호를 확인하였다.

```
if ( a_exp>=b_exp ) { // a의 E가 더 크거나 같은 경우
    exp_differ = (a_exp > b_exp) ? a_exp - b_exp : 0; //exp 차이를 구한것
    result += 1024 * a_exp;
    a %= 1024;
    b %= 1024;

    for (i = 0; i < exp_differ; i++) { // 자릿수 설정
        two *= 2;
    }
    b_save = b+1024;

    if (two != 1) { //b의 지수를 a와 맞추기
        b += 1024;
        b /= two;
    }

    if (b_save != b * two) { //round-to-even을 해야하는지 여부 확인
        round = 1;
    }
```

위에서 a가 크도록 바꾸었기 때문에 첫 if문은 항상 실행될 것이다. exp차이를 구하여 얼만큼 지수이동을 하여야 하는지 계산하였다. 그 후 b가 더 작기에 b의 지수를 옮겼다. 옮긴 b를 다시 원래 위치의 값과 비교해보아, 값의 변형이 있었으면 round-to-even을 해야하는 것으로 판정하였다.

```
if (a_sign == 1 && b_sign ==1 ) {//각각의 부호와 경우에 따른 계산법
    frac = a+ b;
    if (frac >= 1024 || two == 1) {
        frac /= 2;
        exp_change += 1024;
    }
    sign = 1;
}
else if (a_sign ==1 && b_sign == 0) {
    frac = abs(a+1024 - b);
    if (a_exp > b_exp) {
        sign = 1;
    }
}
else if (a_sign == 0 && b_sign == 1) {
    frac = abs(a+1024 - b);
    if (b_exp > a_exp) {
        sign = 1;
    }
}
else {
    frac = a + b;
    if (frac >= 1024 || two == 1) { //right-shift과정
        frac /= 2;
        exp_change += 1024;
    }
}
```

각각의 부호에 따른 계산법이다. 특별히 둘 다 양수일 때에는, frac범위를 초과하는 경우 지수를 증가시키고 right-shift과정을 진행하였다.

```
        if ((a_sign == 1 && b_sign==0)|| (b_sign == 1 && a_sign ==0)) {
            if (frac < 1024) { //left-shift 과정
                while (frac < 1024) {
                    frac *= 2;
                    exp_change -= 1024;
                }
            }
            frac -= 1024;
        }
        frac += exp_change;

        if (sign == 1) frac += 32768; //부호 처리
        result += frac;

        if (round == 1 && result%2 == 1) { // round-to-even 처리
                result += 1;
        }
    }

    return result;
}
```

그 후 둘 중 하나라도 음수였던 경우 1.xx형태를 맞추어주기 위해 필요한 경우에 left-shift 과정을 진행하였다. 음수일 경우 음수처리를 하였고, round-to-even을 해야되는 경우에 이를 처리해주어 결과값을 반환하였다. 여기까지가 sfp_add함수이다.

```
char* sfp2bits(sfp result){
    int i, index = 1;
    char *bit = (char*)malloc(sizeof(char) * 17);
    memset(bit, '0', 17*sizeof(char));
    bit[16] = '\0';

    for (i = 15; i >= 0; i--) { //0,1에 따라 문자열에 저장
        if (result % 2 == 0) {
            bit[i] = '0';
        }
        else {
            bit[i] = '1';
        }

        result /= 2;
    }

    return bit;
}
```

Sfp를 문자열로 반환하는 함수이다. 간단하게 1이냐 0이냐에 따라 문자열에 1과 0으로 저장하였다.

```
hj@hj-VirtualBox://media/sf_ubuntu/systemProgram/hw1$ ./hw1
Test 1: casting from int to sfp
int(43) => sfp(0101000101100000), CORRECT
int(15) => sfp(0100101110000000), CORRECT

Test 2: casting from sfp to int
sfp(0101000101100000) => int(43), CORRECT
sfp(0100101110000000) => int(15), CORRECT

Test 3: casting from float to sfp
float(0.343750) => sfp(0011010110000000), CORRECT
float(-2.250000) => sfp(1100000010000000), CORRECT

Test 4: casting from sfp to float
sfp(0011010110000000) => float(0.343750), CORRECT
sfp(1100000010000000) => float(-2.250000), CORRECT

Test 5: Addition
0101000101100000 + 0101000101100000 = 0101010101100000, CORRECT
0101000101100000 + 0100101110000000 = 0101001101000000, CORRECT
0100101110000000 + 0100101110000000 = 0100111110000000, CORRECT
0011010110000000 + 0011010110000000 = 0011100110000000, CORRECT
0011010110000000 + 1100000010000000 = 1011111110100000, CORRECT
1100000010000000 + 1100000010000000 = 1100010010000000, CORRECT
0101000101100000 + 0011010110000000 = 0101000101101011, CORRECT
0101000101100000 + 1100000010000000 = 0101000100011000, CORRECT
0100101110000000 + 0011010110000000 = 0100101110101100, CORRECT
0100101110000000 + 1100000010000000 = 0100101001100000, CORRECT
```

리눅스 환경에서의 실행화면이다.