Home brewed code for conversion of single precision float value to string

When doing microcontroller programming one may have come across a task of converting a float value to a string. For example when a float value is to be displayed on LCD or it is to be passed out through a UART port. The simple method is to use sprint() function. This function is exactly same as commonly used printf() function. The only difference is that sprint() pass the output to an array instead of some kind of display.

For example if there is a float value myFloat and it is to be converted and stored in an array myArray[] then sprint can be called as below:

sprint(myArray, “%f”, myFloat);

This is an extremely simple way to convert float to string. But the main disadvantages are the memory consumption and time for execution. For example I have used XC16 compiler with dsPIC33FJ32GS406 controller and used sprintf() to convert a float value to string. It took more than 3100 program memory space and taken more than 3800 micro seconds for execution.

This is an annoying situation as my codes are used in very time critical tasks. Also the program memory space is valuable for my projects. So I decided to check whether it is possible to develop an alternate way to do this. My aim is to write a function that converts a float variable to string. The formatting and other features of sprintf() are not needed. I developed a code which take only 212 program memory space (around 1/15 size) and execute within 552uS (almost 7X speed). I am narrating the code below.

First of all note that I myself studied all the details and standards of single precision float variable. My conclusions may be slightly different from the actual facts.

Limitation of the code

The only limitation of the code is that it store the integer part with 32-bit which means integer part can be up to 4294967295. If a float value with integer part more than 4294967295 is passed then code will return with integer part = 4294967295. This can be a problem in some situation.

Representation of single precision variables

Wikipedia article about it give very detailed presentation about it: http://en.wikipedia.org/wiki/Single-precision_floating-point_format

Float values are take 4 bytes (32 bits) of memory. The allocations of bits are as below:

Bitt 28-31				Bits 24-27				Bits 20-23				Bits 16-19				Bits 12-15				Bits 8-11				Bits 4-7				Bits 0-3			
S	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
S	Exponent							Fraction																							

Lower 23 bits represent the fractional part of the number. Next 8 bits represent the exponent and the highest bit gives the sign.

The value of the above number is given by the formula

$$V = (-1)^S \left(1 + \sum_{i=1}^{23} \boldsymbol{F_{23-i}\, 2^{-i}}\right) \times 2^{E-127} \qquad \text{---------- (1)}$$

There is nothing to worry about this bla-bla-bla equation. I am narrating this below. First we start with the Fractional part.

Fraction

The 23 bit fraction represents the number with ***binary point***. This is similar to ***decimal point***. In our usual decimal notation a number 0.1234 is interpreted as 1/10 + 2/100 + 3/1000 + 4/10000. Similarly in binary fractional notation a number 0.101101 is equal to 1/2 + 0/4 + 1/8 + 1/16 + 0/32 + 1/64.

So our 23 bit fractional number written as $F_1\, F_2\, F_3 \ldots F_{22}\, F_{23}$ is equal to

$$\frac{F_1}{2^1} + \frac{F_2}{2^2} + \frac{F_2}{2^2} + \frac{F_2}{2^2} + \ldots + \frac{F_{22}}{2^{22}} + \frac{F_{23}}{2^{23}}$$

This is the meaning of the red colored part in the equation (1) given above.

Sign

The sign is a single bit value in the highest bit position. If it is 1 it means the number is negative. Otherwise it is a positive number.

Exponent

The exponent is an 8 bit number ranging from 0 – 255. Zero and 255 are used for special meaning. 127 is subtracted from the exponent and two raised to exponent give the value of the multiplicand. So the value of the fraction is $Fraction \times 2^{E-127}$. But in the equation (1) above why we need to add 1 to the fraction?. This is explained below.

Suppose we need to represent 0.25 in single precision float standard. We can write it in many different forms. Three representations are:

1. 0.001000×2^1 Here fraction = 00100000… and Exponent is 1.
2. 0.1000×2^{-1} Here fraction = 10000000… and Exponent is -1.
3. 0.000010×2^3 Here fraction = 000010000… and Exponent is 3.

There are plenty of ways to represent 0.25 in fraction + exponent form. So which way?. If we can represent same number in many ways it is not a scientific way. This obviously means that number of unique numbers we can represent with 4 bytes are less. There is a rule to overcome this.

The rule is that the fractional part should shift right until a 1 is shifted in to the integer part. Whenever we give a right shift the exponent is to be decreased. Thus the representation of 0.25 now becomes:

1.000000×2^{-2} Here fraction = 1.0000000… and Exponent is -2.

So the fractional part will be no longer fraction alone, but will contain an integer part which is always 1. If the integer part is always 1 then why we should waste a memory space for that? Hence the integer part (which is always 1) is hidden and that is why we add a 1 in the equation 1.

------ (A)

The meanings of all the fields are explained. We can now convert the float value to string.

For the easiness of splitting the float value to its components we define a union as below:

```
typedef union
{
        float fValue;
        unsigned char fSplit[4];
        unsigned long f_LongValue;
} FloatSplit;
FloatSplit floatSplit;

#define         floatValue      floatSplit.fValue
#define         fSplit0         floatSplit.fSplit[0]
#define         fSplit1         floatSplit.fSplit[1]
#define         fSplit2         floatSplit.fSplit[2]
#define         fSplit3         floatSplit.fSplit[3]
#define         fExponent       floatSplit.fSplit[3]
#define         fLongValue      floatSplit.f_LongValue
```

The float value that is to be converted to string is to be stored to floatValue before calling the conversion function. Note that when the float value is stored to floatValue it is accessible as 4 bytes with using fSplit0, fSplit1, fSplit2 and fSplit3. It is also accessible as a 32 bit value using the long variable fLongValue.

Functions

The main function is

```
void myFtoa(void)
```

It takes no parameters and returns nothing. Before calling this function we have to load the float number to floatValue. On completion this function store the converted string to printBuff[]. For example if floatValue is 1234.5678 then printBuff[ ] = “0000001234.5678”. printBuff[0] to printBuff[10]

will hold the integer part with leading zeros as needed. If the number is –ve then printBuff[0] will be '-'. printBuff[11] will be the decimal point and remaining space will contain the decimal part. The integer part will be always 11 digits with printBuff[0] holding '-' for negative numbers. If the leading zeros are to be truncated we have to write separate code.

Also the fractional portion may not be exactly the same as we expect. In the above example itself we are expecting fractional part = .5678. But it can be 0.567799… or 0.567800122…. This is due to the slight and inherent accuracy problem of the single precision floating point standard itself.

Procedure of myFtoa()

The first part of the function split the number in to parts. The sign is stored in the 1 bit variable 'fSign'. If this is set it means number is negative. The exponent of the number is stored in 'ftoaExpon' with sign indicated by the flag 'expoSign'. For example in case of exponent = -23 'expoSign' will be 1 and 'ftoaExpon' will be 23. The 23-bit fractional part + the additional 1 (see comment (A) in previous page) will be in the lower 24 bits of the floatValue itself. We can access it by the long variable fLongValue or with three unsigned char fSplit0, fSplit1 and fSplit2.

When the split is completed function myFtoa() call extractParts() which convert the integer part and decimal part and store in 'fIntegerPart' and 'fFractPart[ ]' respectively. Note that fIntegerPart is an unsigned long variable and fFractPart[ ] is a string of unsigned char.

Function myFtoa then store the integer part in lower 11 positions of printBuff[ ] and put a decimal point in printBuff[11] and store the decimal part from printBuff[12] onwards. Note that integer part is always stored as 11 digits and in case of negative numbers printBuff[0] will be '-'. This means the integer part may contain leading zeros which can be truncated with additional code.

Function extractParts()

As stated above this function split the number to integer and decimal parts. When this function get control the float number is in a splited form having the 24 bit fractional part stored in the unsigned long variable fLongValue. (Actually it may contain an integer part and fractional part. So do not confuse by the term fractional part). Note that single precision format contain 23 bit fractional part and adding the hidden 1 make it 24 bits. (See note (A) above for more details). The exponent of the number is stored in ftoaExpon and sign of ftoaExpon is in expoSign. Suppose the number is in this form.

$1.101100\ldots. \times 2^{1}$ Fractional part = 1.101100… Exponent = 1 and sign of exponent = +ve

The exponent 2^{1} means the fractional part is to be left shifted once which make fractional part = 11.01100… It means the integer portion = 11 (which means 3) and real fraction = 0.01100…. So as per the exponent and sign of the exponent we give certain number of right or left shift to the number. First we will discuss with the above given example itself where exponent is positive.

As said above the integer part will be formed by giving left shift and adding each bit to fIntegerPart. When updating fIntegerPart the code take care that it do not overflow above the maximum value that fIntegerPart can hold. The maximum value is 2^{32} and if an overflow happen the function stop forming the integer part and make the integer part = $2^{32} - 1$. This is a limitation of the code.

Forming the decimal portion is done in a special way. Here the fractional part = 0.01100… This means

$fraction = \frac{0}{2} + \frac{1}{4} + \frac{1}{8} + \frac{0}{16} + \cdots$ which means 0.25 + 0.125

For adding the decimal numbers we keep a string 'fTable[ ]'. If fTable[ ] = "50000…" it means 0.5000000. If fTable[ ] = "12500…" it means 0.1250000. First we load fTable with a string corresponding to 0.5000. The function reloadTables() do that along with initializing fFractPart[] to "000000…". Also note that fTable[] and fFractPart[] are loaded with binary values not ASCII. That is, fTable[ ] = "12500…" means fTable[0] = 0x01, fTable[1] = 0x02, fTable[2] = 0x05 … and so on. fTable[0] contain 0x01 not the ASCII of 1. Similarly for fFractPart[].

In our above example the number is 0.01100… which means there is no 0.50000 in the fractional part. So the code make fTable[ ] to half the value = 0.2500000. For this purpose the function halfFloatTable() is used. The fractional part contain a 1 in the 0.25 position. So the function call addFloatTable( ) which will add fTable[ ] to fFractPart[ ]. Now fFractPart[ ] will become "2500000" which means 0.2500000. Next the code call halfFloatTable() to make fTable[ ] = "12500000" which means 0.125000. The code then see a 1 in 1/8 position. So it call addFloatTable( ) again to add fTable[ ] to fFractPart[ ]. Now fFractPart[ ] will become "3750000" = 0.375000.

The code use halfFloatTable() and conditionally call addFloatTable( ) to build up the decimal part step by step in to the array fFractPart[ ].

When the exponent is –ve the method is slightly different:

Suppose the number is

$1.101100 \ldots. \times 2^{-3}$ Fractional part = 1.101100… Exponent = -3

This in effect means 1.101100 shifted right 3 times = 0.001101100. But the code will not do the right shift (the code cannot do that because it will lose the bits from the LSB of the 24 bit fractional part). Instead of that the code call reloadTables() and then call halfFloatTable() 3 times. Then the code build up the decimal part step by step in to the array fFractPart[ ] in a similar way narrated above.