

## Lab 1d. Fixed-point Conversions

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, Second edition, by Jonathan W. Valvano, published by Thomson, copyright © 2006.

### Goals

- To introduce the lab equipment,
- To familiarize yourself with Metrowerks CodeWarrior for the 6812,
- To develop a set of useful fixed-point conversion routines.

### Review

- “How to program...” section located at the beginning of this laboratory manual,
- Read "Developing C Programs using ICC11/ICC12/Metrowerks" at <http://www.ece.utexas.edu/~valvano/embed/toc1.htm>
- Valvano Sections 1.1, 1.5, 1.8, and 2.1 from the book Embedded Microcomputer Systems: Real Time Interfacing,

### Starter files

- `Lab1d.c`

### Background

The objectives of this lab are to introduce the 9S12C32 programming environment and to develop a set of useful fixed-point routines that will be used in the subsequent labs. A **software module** is a set of related functions that implement a complete task. In particular, you will create **Fixed.H** and **Fixed.C** files implement the fixed-point conversion module. An important factor in modular design is to separate the policies of the interface (how to use the software is defined in the **Fixed.H** file) from the mechanisms (how the programs are implemented, which is defined in the **Fixed.C** file.) You will develop a third file, **main.c** containing the main program, which will be used to test the fixed-point routines. You should place the prototypes for the public functions in the **Fixed.H** file. The implementations of all functions and any required private global variables should be included in the **Fixed.C** file. The two files **Fixed.H** and **Fixed.C** will be used in subsequent labs, whereas software in the **main.c** file will only be used in this lab to verify the software is operational. There are two long term usages of the main program. Sometimes we deliver the main program to our customer as an example of how our module can be used, illustrating the depth and complexity of our module. Secondly, the main program provides legal proof that we actually tested the software. A judge can subpoena files relating to testing to determine liability in a case where someone is hurt using a product containing our software.

Because the 6812 has no hardware floating point instructions, we will use fixed-point numbers when we wish to express values in our software that have noninteger values. A **fixed-point number** contains two parts. The first part is a **variable integer**, called **I**. This integer may be signed or unsigned. An unsigned fixed-point number is one that has an unsigned variable integer. A signed fixed-point number is one that has a signed variable integer. The **precision** of a number is the total number of distinguishable values that can be represented. The precision of a fixed-point number is determined by the number of bits used to store the variable integer. On the 6812, we typically use 8 bits or 16 bits. Extended precision can be implemented, but the execution speed will be slower because the calculations will have to be performed using software algorithms rather than with hardware instructions. This integer part is saved in memory and is manipulated by software. These manipulations include but are not limited to add, subtract, multiply, divide, convert to BCD, convert from BCD. The second part of a fixed-point number is a **fixed constant**, called **Δ**. This value is fixed, and can not be changed. The fixed constant is not stored in memory. Usually we specify the value of this fixed constant using software comments to explain our fixed-point algorithm. The value of the fixed-point number is defined as the product of the two parts:

$$\text{fixed-point number} \equiv \mathbf{I} \cdot \mathbf{\Delta}$$

The **resolution** of a number is the smallest difference in value that can be represented. In the case of fixed-point numbers, the resolution is equal to the fixed constant ( $\Delta$ ). Sometimes we express the resolution of the number as its units. For example, a decimal fixed-point number with a resolution of 0.001 volts is really the same thing as an integer with units of mV. When interacting with humans, it is convenient to use **decimal fixed-point**. With decimal fixed-point the fixed constant is a power of 10.

$$\text{decimal fixed-point number} = \mathbf{I} \cdot 10^{\mathbf{m}}$$
 for some fixed integer  $\mathbf{m}$

Again, the integer  $m$  is fixed and is not stored in memory. Decimal fixed-point will be easy to convert to ASCII for display, while **binary fixed-point** will be easier to use when performing mathematical calculations. With binary fixed-point the fixed constant is a power of 2.

$$\text{binary fixed-point number} = \mathbf{I} \cdot 2^n \quad \text{for some fixed integer } n$$

In this example, we will develop the equations that 6812 software could need to implement a digital thermometer. Assume the range of the temperature measurement system is 10 to 40 C, and the system uses the 9S12C32's ADC to perform the measurement. In this example, we will assume the transducer and electronics are linear (in a later lab, we will see that most temperature transducers are nonlinear.) The 10-bit ADC analog input range is 0 to +5 V, and the ADC digital output varies 0 to 1023. Let  $x$  be the temperature to be measured in C,  $V_{in}$  be the analog voltage and  $N$  be the 10-bit digital ADC output, then the equations that relate the variables are

$$V_{in} = 5 * N/1023 = 0.0048876 * N \quad \text{and} \quad x = 10 + (30C * V_{in}/5V) = 10 + 6 (C/V) * V_{in}$$

thus

$$x = 10 + 30 * N/1023 = 10 + 0.0293255 * N \quad \text{where } x \text{ is in C}$$

From this equation, we can see that the smallest change in distance that the ADC can detect is about 0.03 C. In other words, the temperature must increase or decrease by 0.03 C for the digital output of the ADC to change by at least one number. It would be inappropriate to save the temperature as an integer, because the only integers in this range are 10, 11, 12... and 40. Since the 6812 does not support floating point, the temperature data will be saved in fixed-point format. Decimal fixed-point is chosen because the temperature data for this thermometer will be displayed for a human to read. A fixed-point resolution of  $\Delta=0.01$  C could be chosen, because it less than the resolution determined by the hardware. Because the hardware resolution (0.03 C) falls between two possible decimal fixed-point resolutions (0.1 and 0.01 C), there are rationalizations for either format. The 0.1 C format provides a more honest display to the customer. On the other hand, if the system will be performing mathematical calculations on the data (average, slope, differences etc.), then it makes sense to perform calculations using the 0.01 C resolution, so the data-format itself doesn't introduce error. Table 1.1 shows the performance of the system with the resolution of  $\Delta=0.01$  C. The table shows us that we need to store the variable part of the fixed-point number in a signed or an unsigned 16-bit variable.

$x$ temperature in C	$V_{in}$ (V) Analog input	$N$ ADC output	$I$ ( $\Delta=0.01C$ ) variable part	$I = 1000 +$ $(44*N+7)/15$
10.00	0.000	0	1000	1000
10.03	0.005	1	1003	1003
16.00	1.000	205	1600	1601
25.00	2.500	512	2500	2502
40.00	5.000	1023	4000	4001

Table 1.1. Performance data of a microcomputer-based temperature measurement.

It is very important to carefully consider the order of operations when performing multiple integer calculations. There are two mistakes that can happen. The first error is **overflow**, and it is easy to detect. Overflow occurs when the result of a calculation exceeds the range of the number system. The following fixed-point calculation, although mathematically correct, has an overflow bug

$$I = 1000 + (3000 * N) / 1023;$$

because when  $N$  is greater than 21,  $3000 * N$  exceeds the range of a 16-bit unsigned integer. If possible, we try to reduce the size of the integers. In this case, an approximate calculation can be performed without overflow

$$I = 1000 + (44 * N) / 15;$$

You can add one-half of the divisor (7 in this case) to the dividend to implement rounding. In this case,

$$I = 1000 + (44 * N + 7) / 15;$$

The addition of "7" has the effect of rounding to the closest integer. The value 7 is selected because it is about one half of the divisor. For example, when  $N=4$ , the calculation  $(44*4)/15=11$ , whereas the " $(44*4+7)/15$ " calculation yields the better answer of 12.

No overflow occurs with this equation using unsigned 16-bit math (assuming  $N$  ranges from 0 to 1023), because the maximum value of  $44*N$  is 45012. If you can not rework the problem to eliminate overflow, the best solution is to use promotion. **Promotion** is the process of performing the operation in a higher precision. For example, in C we cast the input as **unsigned long**, and cast the result as **unsigned short**

```
I = 1000+(unsigned short)((3000*(unsigned long)N)/1023);
```

Again, you can add one-half of the divisor to the dividend to implement rounding. In this case,

```
I = 1000+(unsigned short)((3000*(unsigned long)N+512)/1023);
```

The above equation will run slowly on a 6812 because there are no instructions to implement 32-bit by 32-bit arithmetic. When speed is important we can implement the calculation in assembly, considering carefully the possibility of overflow. In the following implementation, overflow can not occur in the **emul** instruction because a 16 by 16-bit multiply always fits into a 32-bit product. If we assume the input  $N$  is limited to the range of 0 to 1023, then the **ediv** instruction also can not overflow because the range of  $(3000*N)/1023$  will be 0 to 3000, which always will fit into a 16-bit integer. Similarly, the **leay 1000,y** addition will not overflow because the range of  $1000+(3000*N)/1023$  will be 1000 to 4000

```
ldd N
ldx #3000
emul      ;32-bit Y:D is 3000*N
ldx #1023
ediv      ;16-bit Y is (3000*N)/1023
leay 1000,y
sty I
```

The other error is called **drop out**. Drop out occurs after a right shift or a divide, and the consequence is that an intermediate result loses its ability to represent all of the values. If possible, it is better to divide last when performing multiple integer calculations. If you divided first, e.g.,

```
I=1000+44*(N/15);
```

then the values of  $I$  would be only 1000, 1044, 1088, ... or 3992. Not all divide operations produce dropout. For example, the calculation  $I=1000+(3000*N)/1023$ ; includes a divide by 1023, but all values of the input 0,1,2,3...,1023 produce a unique output, thus, dropout has not occurred.

The display algorithm for unsigned decimal fixed point with 0.01 resolution is simple:

- 1) display  $(I/1000)$  as a single digit value (output a space if  $I/1000$  is zero), (tens digit)
- 2) let  $I = I\%1000$
- 3) display  $(I/100)$  as a single digit value (ones digit)
- 4) display a decimal point
- 5) let  $I = I\%100$
- 6) display  $(I/10)$  as a single digit value (tenths digit)
- 7) display  $(I\%10)$  as a single digit value (hundredths digit)
- 8) display the units space, followed by "C"

When adding or subtracting two fixed-point numbers with the same  $\Delta$ , we simply add or subtract their integer parts. First, let  $x, y, z$  be three fixed-point numbers with the same  $\Delta$ , having integer parts  $I, J, K$  respectively. To perform  $z=x+y$ , we simply calculate  $K=I+J$ . Similarly, to perform  $z=x-y$ , we simply calculate  $K=I-J$ . When adding or subtracting fixed-point numbers with different fixed parts, then we must first convert two the inputs to the format of the result before adding or subtracting. This is where binary fixed-point is more convenient, because the conversion process involves shifting rather than multiplication/division.

In this next example, let  $x, y, z$  be three binary fixed-point numbers with the different  $\Delta$ s. In particular, we define  $x$  to be  $I\cdot 2^{-5}$ ,  $y$  to be  $J\cdot 2^{-2}$ , and  $z$  to be  $K\cdot 2^{-3}$ . To convert  $x$ , to the format of  $z$ , we divide  $I$  by 4 (right shift twice). To convert  $y$ , to the format of  $z$ , we multiply  $J$  by 2 (left shift once). To perform  $z=x+y$ , we calculate

$$K=(I>>2)+(J<<1)$$

For the general case, we define  $x$  to be  $I\cdot 2^n$ ,  $y$  to be  $J\cdot 2^m$ , and  $z$  to be  $K\cdot 2^p$ . To perform any general operation, we derive the fixed-point calculation by starting with desired result. For addition, we have  $z=x+y$ . Next, we substitute the definitions of each fixed-point parameter

$$\mathbf{K} \cdot 2^p = \mathbf{I} \cdot 2^n + \mathbf{J} \cdot 2^m$$

Lastly, we solve for the integer part of the result

$$\mathbf{K} = \mathbf{I} \cdot 2^{n-p} + \mathbf{J} \cdot 2^{m-p}$$

For multiplication, we have  $z = x \cdot y$ . Again, we substitute the definitions of each fixed-point parameter

$$\mathbf{K} \cdot 2^p = \mathbf{I} \cdot 2^n \cdot \mathbf{J} \cdot 2^m$$

Lastly, we solve for the integer part of the result

$$\mathbf{K} = \mathbf{I} \cdot \mathbf{J} \cdot 2^{n+m-p}$$

For division, we have  $z = x/y$ . Again, we substitute the definitions of each fixed-point parameter

$$\mathbf{K} \cdot 2^p = \mathbf{I} \cdot 2^n / \mathbf{J} \cdot 2^m$$

Lastly, we solve for the integer part of the result

$$\mathbf{K} = \mathbf{I} / \mathbf{J} \cdot 2^{n-m-p}$$

Again, it is very important to carefully consider the order of operations when performing multiple integer calculations. We must worry about both overflow and drop out. In particular, in the division example, if  $(n-m-p)$  is positive then the left shift ( $\cdot 2^{n-m-p}$ ) should be performed before the divide ( $/\mathbf{J}$ ). On the other hand, if  $(n-m-p)$  is negative then the divide ( $/\mathbf{J}$ ) should be performed before the right shift ( $\cdot 2^{n-m-p}$ ).

We can use these fixed-point algorithms to perform complex operations using the integer functions of our 6812. For example, consider the following digital filter calculation.

$$\mathbf{y} = \mathbf{x} - 0.0532672 \cdot \mathbf{x}_1 + \mathbf{x}_2 + 0.0506038 \cdot \mathbf{y}_1 - 0.9025 \cdot \mathbf{y}_2;$$

In this case, the variables  $\mathbf{y}$ ,  $\mathbf{y}_1$ ,  $\mathbf{y}_2$ ,  $\mathbf{x}$ ,  $\mathbf{x}_1$ , and  $\mathbf{x}_2$  are all integers, but the constants will be expressed in binary fixed-point format. The value  $-0.0532672$  will be approximated by  $-14 \cdot 2^{-8}$ . The value  $0.0506038$  will be approximated by  $13 \cdot 2^{-8}$ . Lastly, the value  $-0.9025$  will be approximated by  $-231 \cdot 2^{-8}$ . The fixed-point implementation of this digital filter is

$$\mathbf{y} = \mathbf{x} + \mathbf{x}_2 + (-14 \cdot \mathbf{x}_1 + 13 \cdot \mathbf{y}_1 - 231 \cdot \mathbf{y}_2) / 256;$$

A second example comes from the integral term of a PID controller, illustrating that fixed-point need not be either decimal or binary. In this control system, we wish to calculate

$$\mathbf{I}(n) = \mathbf{I}(n-1) + 0.157658 \cdot (\mathbf{X} - \mathbf{X}')$$

Fixed-point is used to approximate  $1.57658$  as  $35/222$  ( $0.157657658$ ). So we can implement this control equation using integer calculations.

$$\mathbf{I}(n) = \mathbf{I}(n-1) + 35 \cdot (\mathbf{X} - \mathbf{X}') / 222$$

### Preparation (do this before your lab period) (if you don't have a partner, do the preparation on your own)

There is no hardware for this lab. Please create a project that contains your three files:

**fixed.h** header file for the fixed-point conversion module  
**fixed.c** implementation file for the fixed-point conversion module  
**main.c** main program that tests the fixed-point conversion module

A "syntax-error-free" hardcopy listing for procedure Part (2) of the software is required as preparation. The TA will check this off at the beginning of the lab period. You are required to do your editing before lab. The debugging will be done during lab. Document clearly the operation of the routines. The comments included in **fixed.h** are intended for the client (programmers that will use your functions.) The comments included in **fixed.c** are intended for the coworkers (programmers that will debug/modify your functions.) Your programs should not perform any SCI input or output; rather you should learn how to use the interactive features of the Metrowerks debugger.

The format is signed 16-bit with a fixed-point constant of 0.01. The full-scale range is from  $-327.67$  to  $+327.67$ . The fixed-point value  $-327.68$  (integer  $-32768$ ) signifies an error has occurred. The **Fixed\_Str2Fix** function should convert an ASCII string into the signed 16-bit integer part of the fixed-point number. The specifications of **Fixed\_Str2Fix** are illustrated in Program 1.1. **Fixed\_Str2Fix** should round to the closest

fixed-point result (e.g., 15.995 rounds up to 16.00 and 16.004 rounds down to 16.00). You may limit the system to 16-bit signed math. In particular, some input like 1.2345678 might be considered illegal because they cause overflow of intermediate results. In the comments of the software, please discuss why you chose your particular implementation method over the other available choices. In other words, it is OK that input such as 1.2345678 returns -32768, but it is inappropriate for your software to return an incorrect value (because of an overflow occurring during an intermediate calculation.) You are free to modify the prototypes in any way you feel is appropriate. You must check for illegal inputs, returning -32768, which is defined as an illegal number. The main program then can decide what to do on an illegal input.

Your second function converts the signed 16-bit integer part of a fixed-point number into an ASCII string. Again, this function also performs no SCI input/output. The input/output specifications for **Fixed\_Fix2Str** are also illustrated in the Program 1.1.

You are free to develop a testing file in whatever style you wish. This main program has three important roles. First, you will use it to test all the features of your program. Second, a judge in a lawsuit can subpoena this file. In a legal sense, this file documents to the world the extent to which you verified the correctness of your program. When one of your programs fails in the marketplace, and you get sued for damages, your degree of liability depends on whether you took all the usual and necessary steps to test your software, and the error was unfortunate but unforeseeable, or whether you rushed the product to market without the appropriate amount of testing and the error was a foreseeable consequence of your greed and incompetence. Third, if you were to sell your software package (**fixed.c** and **fixed.h**), your customer can use this file to understand how to use your package, its range of functions, and its limitations.

### Procedure (do this during your lab period)

Part (1) Experiment with the different features of Metrowerks and its debugger. Familiarize yourself with the various options and features available in the editor/assembler/terminal. Edit, compile, download, and run your project working through all aspects of software development. In particular, learn how to:

- create hard copy listings of your source code;
- open and read the assembly listing files;
- save your source code on floppy disk;
- compile, download, and execute a program.

Part (2) Debug your software module (**fixed.h fixed.c main.c**).

### Deliverables (exact components of the lab report)

- A) Objectives (1/2 page maximum)
- B) Hardware Design (none for this lab)
- C) Software Design (no software printout in the report)
- D) Measurement Data (none for this lab)
- E) Analysis and Discussion (1 page maximum)

### Checkout (show this to the TA)

You should be able to demonstrate correct operation of each routine:

- show the TA you know how to observe global variables and I/O ports using the debugger;
- demonstrate to the TA you know how to observe assembly language code;
- verify proper input/outputs of the I/O functions;
- verify the proper handling of illegal formats;
- demonstrate your software does not crash.

**Your software files will be copied by the TA, and graded for style at a later time.**

### Hints

1) Create a new project first, then make small changes and test. Make backups of the previous versions, so that when you add something that doesn't work, you can go back to a previous working version and try a new approach. Please add documentation that makes it easier to change and use in the future. Your job is to organize these routines to facilitate subsequent laboratories.

2) You must always look at the assembly language created by the compiler to verify the appropriate function. Analyzing the assembly listing files is an excellent way to double-check if your software will perform the intended function. This is especially true when overflow, dropout, and execution speed are important. We have not found many bugs with this compiler. Most reported compiler bugs (my program doesn't do what I want) turn out to be

programmer errors or misunderstanding about the C language. However, if you think you've found a bug, email the source and assembly listing to the TA explaining where the bug is.

3) You may find it useful read to the temperature measurement and calculator labs to get a feel for the context of the fixed-point routines you are developing. In particular, you should be able to use these functions in these labs without additional modification.

```
// const will place these structures in ROM
const struct inTestCase{          // used to test Fixed_Str2Fix
    unsigned char InBuffer[10];  // Input String
    short CorrectAnswer;        // proper result
};
typedef const struct inTestCase inTestCaseType;
inTestCaseType inTests[34]={
{ "0",          0}, // 0.00
{ "1",          100}, // 1.00
{ ".02",        2}, // 0.02
{ "+.1",        10}, // 0.10
{ "+1.",        100}, // 1.00
{ "12.34",     1234}, // 12.34
{ "5.05",      505}, // 5.05
{ "10.70",    1070}, // 10.70
{ "0.0023",    0}, // 0.00
{ "9.994",     999}, // 9.99
{ "9.995",    1000}, // 10.00
{ "14.595",   1460}, // 14.60
{ "14.604",   1460}, // 14.60
{ "19.994",   1999}, // 19.99
{ "19.995",   2000}, // 20.00
{ "-21",       -2100}, // -21.00
{ "-27.67",   -2767}, // -27.67
{ "327.67",   32767}, // 327.67
{ "-327.67", -32767}, // -327.67
{ "-327.69", -32768}, // illegal, too big
{ "328",      -32768}, // illegal, too big
{ "-500",     -32768}, // illegal, too big
{ "327.700", -32768}, // illegal, too big
{ "327.675", -32768}, // illegal, too big
{ "327.8",   -32768}, // illegal, too big
{ "3*2",     -32768}, // illegal, illegal character
{ "3A.769", -32768}, // illegal, illegal character
{ "32.7b9", -32768}, // illegal, illegal character
{ "3..767", -32768}, // illegal, two decimal points
{ "3.2.767", -32768}, // illegal, two decimal points
{ ".",      -32768}, // illegal, no numbers
{ "-+1",    -32768}, // illegal, both signs
{ "1+",     -32768}, // illegal, sign not first
{ "",      -32768}, // illegal, no numbers
};
const struct outTestCase{        // used to test Fixed_Fix2Str
    short InNumber;              // test input number
    unsigned char OutBuffer[10]; // Output String
};
typedef const struct outTestCase outTestCaseType;
outTestCaseType outTests[20]={
{ 0, " 0.00" }, // 0.00
{ 4, " 0.04" }, // 0.04
{ 10, " 0.10" }, // 0.10
{ -20, " -0.20" }, // 0.20
{ 100, " 1.00" }, // 1.00
{ 505, " 5.05" }, // 5.05
{ 1070, " 10.70" }, // 10.70
{ 1234, " 12.34" }, // 12.34
};
```

```
{ -2859, " -28.59" }, // -28.59
{ -2999, " -29.99" }, // -29.99
{ -3000, " -30.00" }, // -30.00
{ -3001, " -30.01" }, // -30.01
{ 6460, " 64.60" }, // 64.60
{ -9999, " -99.99" }, // -99.99
{ 10000, " 100.00" }, // 100.00
{-12345, "-123.45" }, // -123.45
{ 32767, " 327.67" }, // 327.67
{-32767, "-327.67" }, // -327.67
{ -1, " -0.01" }, // -0.01
{ 32768, " ***.***" } // illegal
};
short Input;          // fixed-point resolution 0.001
short Result;        // fixed-point resolution 0.001
unsigned short I;
unsigned short Errors, AnError;
unsigned char Buffer[10];
void main(void){ // possible main program that tests your functions
    Errors = 0;
asm cli
    for(I=0; I<34; I++){
        strcpy((char *)Buffer, (char *)inTests[I].InBuffer); // input test
        Result = Fixed_Str2Fix(Buffer); // convert string to fixed point
        if(Result != inTests[I].CorrectAnswer){
            Errors++;
            AnError = I;
        }
    }
    for(I=0; I<20; I++){
        Input = outTests[I].InNumber;
        Fixed_Fix2Str(Input, Buffer);
        if(strcmp((char *)Buffer, (char *)outTests[I].OutBuffer)){
            Errors++;
            AnError = I;
        }
    }
    for(;;) {} /* wait forever */
}
```

*Program 1.1. One approach to software testing*