Lab 1. Fixed-point Output

This laboratory assignment accompanies the book, Embedded Systems: Real-Time Interfacing to ARM Cortex M Microcontrollers, ISBN-13: 978-1463590154, by Jonathan W. Valvano, copyright © 2014.

Goals
- To introduce the lab equipment,
- To familiarize yourself with Keil uVision4 for the ARM Cortex M processor,
- To develop a set of useful fixed-point output routines.

Review
- "How to program…" section located at the beginning of this laboratory manual,
- Read "Developing C Programs using Metrowerks" at http://users.ece.utexas.edu/~ryerraballi/CPrimer/
- Valvano Chapters 1, 2 and 3 from the book Embedded Systems: Real Time Interfacing,
- **style.pdf**, **style_policy.pdf** and **c_and_h_files.pdf** guide

Starter files
- **Lab1.c fixed.h ST7735_xxx.zip or Printf_UART_xxx.zip** project

Background

The objectives of this lab are to introduce the TM4C123 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 output. 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 add your programs on top of an existing ST7735 or UART driver. Look at the ST7335 or UART examples on the book web site, but also explore the example codes in the TivaWare. Feel free to use either ST7735 or UART as you wish. In the main program you will develop software to test your functions. The **Lab1.c** shows an example test program. You should place the prototypes for the public functions in the header files. The implementations of all functions and any required private global variables should be included in the code files. The **fixed.h** and **fixed.c** files will be used in subsequent labs, whereas software in the main program 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.

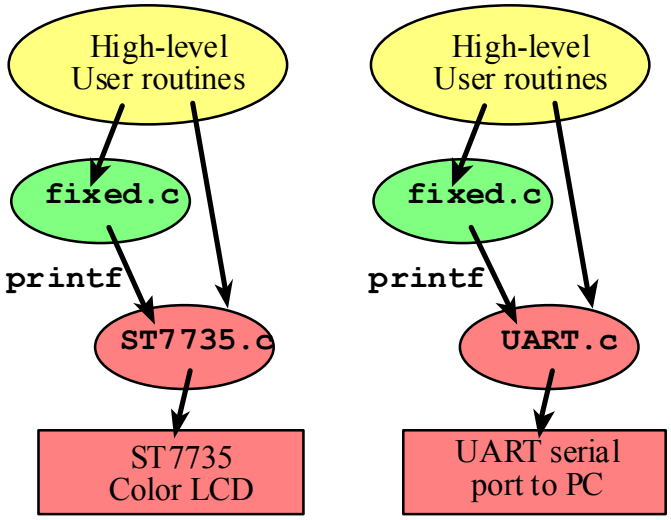

Figure 1.1. If you use printf for character output, the call graph for the system shows your fixed.c software could be used with either the ST7735 color LCD and the UART serial port communication channel with the PC.

In order to use the power and flexibility of the **`printf`** function, you must implement an output function with the following prototype inside the low-level output driver. In this lab, you can ignore the **`*f`** parameter.

However, in future systems you can use it to allow the user to redirect output using **fprintf**. Implementing this function will direct all standard output to a display. The driver will handle at special characters like **CR LF** and **TAB**. You can then add **#include <stdio.h>** and use **printf** to output to whichever output device you wish. Look for the implementation of **fputc** in **ST7735.c** or **UART.c**.

```
// Print a character to ST7735 LCD
int fputc(int ch, FILE *f){
  ST7735_OutChar(ch);
  return 1;
}
```

Even though the ARM Cortex M4 has 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 ARM Cortex M, we typically use 32 bits. If we needed to store a large array of fixed-point numbers in RAM or on disk, we could use 8 or 16 bits to save space. 64-bit 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, and divide. The second part of a fixed-point number is a **fixed constant**, called Δ. This value is fixed. If we wish to change this constant we must perform a design-edit-compile-download-test cycle. 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:

fixed-point number ≡ **I**•Δ

The **resolution** of a number system 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 (Δ). 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.

decimal fixed-point number = $\mathbf{I} \bullet 10^{\mathbf{m}}$ for some fixed integer **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.

binary fixed-point number = $\mathbf{I} \bullet 2^{\mathbf{n}}$ for some fixed integer **n**

When adding or subtracting two fixed-point numbers with the same Δ, we simply add or subtract their integer parts. First, let x ,y, z be three fixed-point numbers with the same Δ, 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.

We will study an example using fixed-point. In this example, we will develop the equations that software could need to implement a digital thermometer. Assume the range of the temperature measurement system is 10 to 50ºC, and the system uses the TM4C 12-bit 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 12-bit ADC analog input range is 0 to +3.3 V, and the ADC digital output varies 0 to 4095. Let **T** be the

temperature to be measured in ºC, $\mathbf{V_{in}}$ be the analog voltage and **N** be the 12-bit digital ADC output, then the equations that relate the variables are

$V_{in} = 3.3 * N/4095$ and $T = 10+(40ºC * V_{in}/3.3V)$

thus

$T = 10+40*N/4095 = 10+ 0.009768 *N$ where T is in ºC

From this equation, we can see that the smallest change in distance that the ADC can detect is about 0.01 ºC. In other words, the temperature must increase or decrease by 0.01ºC 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 50. Because the range is small and known, we can save the temperature data 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 Δ=0.01ºC could be chosen, because it less than the resolution determined by the hardware. We choose a display resolution that provides an accurate and honest representation of the measured data. Table 1.1 shows the performance of the system with the resolution of Δ=0.01ºC. The table shows us that we need to store the variable part of the fixed-point number in a signed or an unsigned 32-bit variable.

T temperature (ºC)	**V_{in}** (V) Analog input	**N** ADC output	**I** (Δ= 0.01ºC) variable part
10.00	0.000	0	1000
10.01	0.001	1	1001
20.00	0.825	1024	2000
30.00	1.650	2048	3000
40.00	2.475	3072	4000
50.00	3.300	4095	5000

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 if performed with 16-bit math

```
I = 1000+(4000*N)/4095;
```

or

```
I = 1000+(4000*N)/4096;
```

because when **N** is greater than 16, **4000*N** exceeds the range of a 16-bit unsigned integer. On the ARM Cortex M there would be no overflow with 32-bit math. If possible, we try to reduce the size of the integers. In this case, the following calculation could also be used.

```
I = 1000+(125*N)/128;
```

You can add one-half of the divisor (64 in this case) to the dividend to implement rounding. In this case,

```
I = 1000+(125*N+64)>>7;
```

The addition of "64" has the effect of rounding to the closest integer. The value 64 is selected because it is about one half of the divisor. For example, when N=1, the calculation (125*1)/128=0, whereas the "(125*1+64)/128" calculation yields the better answer of 1. Shifting executes faster than multiplication or division, so reworking the problem using shifts will provide speed improvements.

No overflow occurs with this equation using unsigned 32-bit math (assuming N ranges from 0 to 4096), because the maximum value of **(125*N+64)** is 511939. Another way to consider overflow is to notice 125 requires 7 bits

to represent. If N is a 12-bit number then 125*N will be a 7+12=19-bit number. If you cannot 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.

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 looses 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+125*(N/128);
```

then the values of I would be only 1000, 1125, 1250, … or 4875. Not all divide operations produce dropout. For example, the calculation `I=1000+(4000*N)/4095;` includes a divide by 4095, but most values of the input 0,1,2,3…,4095 produce a unique output, thus, dropout has not occurred.

In this example, the temperature ranges from 10.00 to 50.00 ºC. This means the integer part (`I`) ranges from 1000 to 5000, and the display output will range from `10.00C` to `50.00C`. Using `printf`, the display algorithm for unsigned decimal fixed point temperature with 0.01-resolution is simple:

1) Display (`I/100`) as an integer with exactly 2 digits
2) Display a decimal point
3) Display `I%100` as an integer with exactly 2 digits
4) Display the units "`C`"

Preparation (do this before your lab period) (do preparation by yourself if you do not have a partner yet)

Preparation is due before the lab period starts and must be uploaded to Canvas. Late preparations will not be accepted. If you own the ST7735 color display we suggest you use it, but either way you may choose to display results either on the ST7735 or the UART. Rather than creating a new project, we suggest you make a copy of an existing ST7735 or UART project and rename it Lab 1. You will need to add your **fixed.c** to the Lab 1 project. "Syntax-error-free" hardcopy listings of your main, **fixed.h**, and **fixed.c** are required as preparation. Doing the preparation before lab allows you to debug with the TA present in the lab. The proper approach to EE445L is to design and edit before lab, and then you can build your hardware and debug your software 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 main program will be used to test the **fixed.c** functions. It is important for you to learn how to use the interactive features of the uVision4 debugger. To improve the reusability of the functions in the **fixed.c** driver, the **fixed.c** functions must NOT call any display functions directly. In other words, output by **fixed.c** functions use only `printf`. Because the driver implements redirection function, `fputc`, `printf` calls will indeed be displayed on the appropriate display.

The first format you will handle is unsigned 32-bit decimal fixed-point with a resolution of 0.01. The full-scale range is from 0 to 999.99. The fixed-point value over 999.99 (integer over 99999) signifies an error. The `Fixed_uDecOut2` function takes an unsigned 32-bit integer part of the fixed-point number and outputs the fixed-point value on the OLED. The specifications of `Fixed_uDecOut2` are illustrated in Table 1.1. In order to make the OLED output pretty it is required that all output commands produce exactly 6 characters. This way the decimal point is always drawn in the exact same location, independent of the number being displayed.

Parameter	Display
0	0.00
1	0.01
99	0.99
100	1.00
999	9.99
1000	10.00

9999	99.99
10000	100.00
99999	999.99
100000	***.**

Table 1.1. Specification for the **Fixed_uDecOut2** *function.*

The second format you will handle is signed 16-bit decimal fixed-point with a resolution of 0.001. The full-scale range is from -9.999 to +9.999. Any integer part outside the range of -9999 to +9999 signifies an error. The **Fixed_sDecOut3** function takes a signed 32-bit integer part of the fixed-point number and outputs the fixed-point value on the display. The specifications of **Fixed_sDecOut3** are illustrated in Table 1.2. In order to make the display output pretty it is required that all output commands produce exactly 6 characters. This way the decimal point is always drawn in the exact same location, independent of the number being displayed.

Parameter	Display
-100000	*.***
-10000	*.***
-9999	-9.999
-999	-0.999
-1	-0.001
0	0.000
123	0.123
1234	1.234
9999	9.999
10000	*.***

Table 1.2. Specification for the **Fixed_sDecOut3** *function.*

The third format you will handle is unsigned 32-bit binary fixed-point with a resolution of 1/256. The full-scale range is from 0 to 999.99. If the integer part is larger than 256000, it signifies an error. The **Fixed_uBinOut8** function takes an unsigned 32-bit integer part of the binary fixed-point number and outputs the fixed-point value on the display. The specifications of **Fixed_uBinOut8** are illustrated in Table 1.3.

Parameter	Display
0	0.00
2	0.01
64	0.25
100	0.39
500	1.95
512	2.00
5000	19.53
30000	117.19
255997	999.99
256000	***.**

Table 1.3. Specification for the **Fixed_uBinOut8** *function.*

In order to make the display output pretty it is required that all output commands produce exactly 6 characters. This way the decimal point is always drawn in the exact same location, independent of the number being displayed. Because of rounding your solution may be different by ±0.01 from the examples in Table 1.3.

You are free to develop a main program that tests your function. You may implement this test in whatever style you wish. You may create your own or edit the starter Lab 1.c. 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)

1. Run 3 or 4 example programs and verify your microcontroller board. The UART projects produce output that can be seen on PuTTY. The SysTick projects will toggle the LEDs. If you bought your board used, ask the TA how to run the tester project.

2. Experiment with the different features of uVision4 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.

3. Debug your software modules (**`fixed.h`**, **`fixed.c`**, file for testing). If you are testing with ST7735, you must wire it up according to the directions in the ST7735.c program. If you are testing with the UART, you will need to run a program like PuTTy or TExaSdisplay on the PC. You can find the COM port using the Windows Device Manager. The serial settings are 115200 bits/sec, no parity, 1 stop, and no flow control.

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)
B) Hardware Design (none for this lab)
C) Software Design (upload **`fixed.h fixed.c`** testing files to Canvas as instructed by your TA)
D) Measurement Data (none for this lab)
E) Analysis and Discussion (1 page maximum). In particular, answer these questions

1) In what way is it good design of fixed.c that there are no arrows directly from the fixed.c module to the low-level display routines in the call graph for your system?

2) Why is it important for the decimal point to be in the exact same physical position independent of the number being displayed? Think about how this routine could be used with the **`ST7735_SetCursor`** command.

3) When should you use fixed-point over floating point? When should you use floating-point over fixed-point?

4) When should you use binary fixed-point over decimal fixed-point? When should you use decimal fixed-point over binary fixed-point?

5) Give an example application (not mentioned in this lab assignment) for fixed-point. Describe the problem, and choose an appropriate fixed-point format. (no software implementation required).

6) Can we use floating point on the ARM Cortex M4? If so, what is the cost?

10 points Extra credit Perform an empirical study to evaluate four implementations on the Cortex M4. Two implements use fixed-point, two use floating-point, two are written in assembly, and two are written in C. For each implementation measure the total execution time. Make conclusions about implementing arithmetic operations on the Cortex M4.

```
// version 1: C floating point
// run with compiler options selected for floating-point hardware
volatile float T;    // temperature in C
volatile uint32_t N; // 12-bit ADC value
void Test1(void){
  for(N=0; N<4096; N++){
    T = 10.0+ 0.009768*N;
  }
}

// version 2: C fixed-point
```

```
volatile uint32_t T;    // temperature in 0.01 C
volatile uint32_t N;    // 12-bit ADC value
void Test2(void){
  for(N=0; N<4096; N++){
    T = 1000+ (125*N+64)>>7;
  }
}
;Version 3 assembly floating point
; run with floating-point hardware active
        AREA    DATA, ALIGN=2
T       SPACE   4
N       SPACE   4
        AREA    |.text|, CODE, READONLY, ALIGN=2
        THUMB
Test3
      MOV R0,#0
      LDR R1,=N    ;pointer to N
      LDR R2,=T    ;pointer to T
      VLDR.F32 S1,=0.009768
      VLDR.F32 S2,=10
loop3 STR R0,[R1]          ; N is volatile
      VMOV.F32 S0,R0
      VCVT.F32.U32 S0,S0   ; S0 has N
      VMUL.F32 S0,S0,S1    ; N*0.09768
      VADD.F32 S0,S0,S2    ; 10+N*0.0968
      VSTR.F32 S0,[R2]     ; T=10+N*0.0968
      ADD R0,R0,#1
      CMP R0,#4096
      BNE loop3
      BX  LR

;version 4, assembly fixed point
        AREA    DATA, ALIGN=2
T       SPACE   4
N       SPACE   4
        AREA    |.text|, CODE, READONLY, ALIGN=2
        THUMB
Test4 PUSH {R4,R5,R6,LR}
      MOV R0,#0
      LDR R1,=N   ;pointer to N
      LDR R2,=T   ;pointer to T
      MOV R3,#125
      MOV R4,#64
      MOV R5,#1000
loop4 STR R0,[R1]          ; N is volatile
      MUL R6,R0,R3         ; N*125
      ADD R6,R6,R4         ; N*125+64
      LSR R6,R6,#7         ; (N*125+64)/128
      ADD R6,R6,R5         ; 1000+(N*125+64)/128
      STR R6,[R2]          ; T = 1000+(N*125+64)/128
      ADD R0,R0,#1
      CMP R0,#4096
      BNE loop4
      POP {R4,R5,R6,PC}
```

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/output parameters when calling a function;
- verify the proper handling of illegal formats;
- demonstrate your software does not crash.

Read the **style.pdf**, **style_policy.pdf** and **c_and_h_files.pdf** from the lab manual website, and be prepared to answer a few questions.

Hints

1) Do not create a new project. Start with a project you know works, make a copy of the project and, then make small changes. It is good practice to make a small change and test it. Once you have some new code that works, make a back-up, 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) It is also good practice to 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 any 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 lab to get a feel for the context of the fixed-point routines you are developing. In particular, you should be able to use your Lab 1 functions in future labs without additional modification.

4) In the following test program, **`Fixed_uBinOut8s`** is the same as **`Fixed_uBinOut8`**, except **`printf`** is replaced by **`sprintf`**.

```
#include <stdio.h>
#include "fixed.h"
// const will place these structures in ROM
const struct outTestCase{       // used to test routines
  unsigned long InNumber;       // test input number
  char OutBuffer[10];           // Output String
};
typedef const struct outTestCase outTestCaseType;
outTestCaseType outTests3[16]={
{     0,  "  0.00" }, //      0/256 = 0.00
{     4,  "  0.01" }, //      4/256 = 0.01
{    10,  "  0.03" }, //     10/256 = 0.03
{   200,  "  0.78" }, //    200/256 = 0.78
{   254,  "  0.99" }, //    254/256 = 0.99
{   505,  "  1.97" }, //    505/256 = 1.97
{  1070,  "  4.17" }, //   1070/256 = 4.17
{  5120,  " 20.00" }, //   5120/256 = 20.00
{ 12184,  " 47.59" }, //  12184/256 = 47.59
{ 26000,  "101.56" }, //  26000/256 = 101.56
{ 32767,  "127.99" }, //  32767/256 = 127.99
{ 32768,  "128.00" }, //  32768/256 = 128
{ 34567,  "135.02" }, //  34567/256 = 135.02
{123456,  "482.25" }, // 123456/256 = 482.25
{255998,  "999.99" }, // 255998/256 = 999.99
{256000,  "***.**" }  // error
};
unsigned int Errors,AnError;
```

```
char Buffer[10];
void main(void){ // possible main program that tests your functions
unsigned int i;
  Errors = 0;
  for(i=0; i<16; i++){
    Fixed_uBinOut8s(outTests3[i].InNumber,Buffer);
    if(strcmp(Buffer, outTests3[i].OutBuffer)){
      Errors++;
      AnError = i;
    }
  }
  for(;;) {} /* wait forever */
}
```

Program 1.1. One approach to software testing