To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

RENESAS

SH2/SH2A Family

Using a Fixed Point Math Library

Introduction

The purpose of this application note is to give examples of using the Fixed Point Math library for the SH2 and SH2A families.

In many applications it is necessary to maintain accuracy when doing math and floating point type calculations are desirable. However, many smaller microcontrollers do not have an FPU available. When this happens, engineers typically resort to fixed point math, which can help maintain the best "dynamic range" while using the Integer math instructions and multiplier hardware of a non-FPU microcontroller unit.

This application note does not review Fixed Point concepts or number ranges; those are left to the Fixed Point Library Users manual (see references section). Rather, this attempts to give examples of real-life uses of fixed point libraries.

NOTE: This application note is not intended for the SH2A families with integrated FPU in the core, although it may be helpful in these devices when receiving fixed point data from other Microcontrollers and/or applications.

Target Device

SH2/SH2A

Contents

1. Demo Requirements 2
2. Building the Code 2
3. Running the Demonstration Code 2
4. Selecting Resolutions for Variables 6
5. Multiple Build Configurations 9
6. Appendix A: Library Usage in HEW 10
7. Glossary 13

1. Demo Requirements

The following items are required in order to build and run this demonstration.

1.1 Required Renesas Development Tools (software)

The following tools and their versions are required for building the demonstration and following the tutorial. Evaluation editions of these tools are all available for download from our website.

- High-performance Embedded Workshop (HEW) Version 4.06.xx.xx (or later)
  http://www.renesas.com/download

  Web Search Keyword: "High-performance Embedded Workshop"
- Renesas SuperH RISC Engine Standard Toolchain Version 9.3. Release 00 (or later)
  http://www.renesas.com/download

  Web Search Keyword: "SuperH Compiler"

1.2 Required Hardware for Demo

The following items are needed for the demo below.

- RSK for SH7286 (includes E10A for Starter Kits).
  http://www.renesas.com/rsk

1.3 Demonstration Software

The Demonstration software is intended for use with the Renesas RSK board. More detailed documentation on the RSK hardware is included with the kit or can be downloaded from the Renesas Web site.

2. Building the Code

The demonstration software can be built by opening the HEW Workspace (FP_Demo.hws) and building the code using the "**Build All**" icon . The resulting 'FP_Demo.abs' file in the 'Debug' directory can then be downloaded using the E10A and HEW.

3. Running the Demonstration Code

The demonstration is setup to use data from the potentiometer connected to ADC Unit 0, channel 0. It is assumed the user has already setup the SH7286 RSK using the QSG and is familiar with the operation of the E10A and the SH Family tool chains. The user should refer to the RSK schematic for additional detail as necessary.

3.1 Overview

The demonstration uses the Potentiometer on the RSK to simulate readings from current sensors and to generate "commanded motor voltages" in a motor control scenario. Since it has only one Potentiometer it cannot run both scenarios at the same time. It uses a static variable named ad_demo to select which scenario to run. When ad_demo is set to logic 0, it uses the potentiometer to simulate commanded motor voltages which are then turned into PWM Sine waves of the correct voltage. If ad_demo is set to logic 1, the potentiometer will be used to simulate the reading from a current sensor and the resulting values displayed in the watch window.

We provided two scenarios to give examples that the resolution and size of variables are determined by the system requirements. The engineer can choose these based on the range of numbers he needs to represent as well as the dynamic range he needs for his control system to operate properly.

IMPORTANT NOTE: The demonstration code calls the floating point library in order to display the actual values in the watch windows. When using the Fixed Point library, it is never required to convert the fixed point values to floating point for your calculations, control loops, etc. There may be some reasons for converting Fixed Point to Floating Point such as displaying fixed point values in easily readable floating point form on the User Interface or sending data to another program/MCU that might need Floating Point results. However, the fixed point versions of the variables are all you need to use in the math application itself.

3.2 Adding Correct Watch Variable

The project built session is saved with the correct watch variable for the demonstration assigned. In the event the session is not correct, add the correct watch variables as shown in Figure 1. Be sure they are set to Auto-update (**R** rather than R)

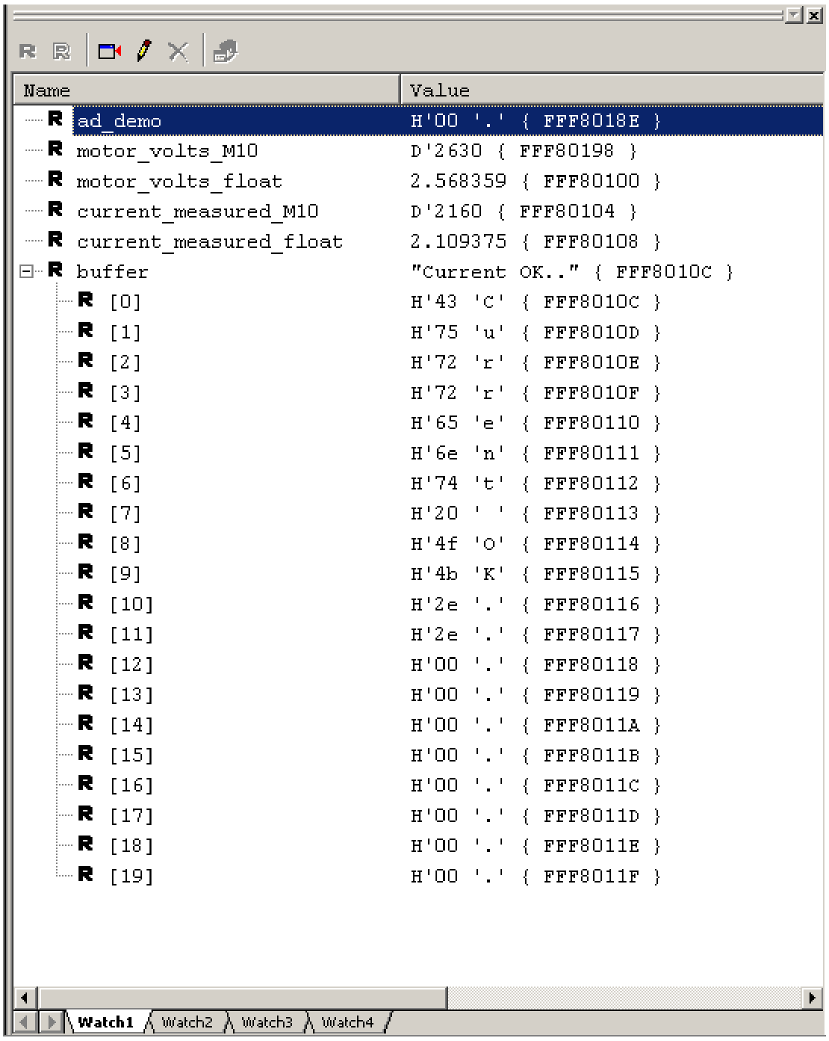

Figure 1: Watch Window

3.3 Simulating Motor Voltage Generation

When ad_demo=0, the demonstration generates a simulated motor voltage out of the PWM channels in complementary mode. For the Motor voltage demonstration we picked a value that could easily be both simulated and verified in the demonstration. For this demo it is set-up to generate simulate Sine wave voltages to the Motor of 0 – 5V so functionality can easily be confirmed with an oscilloscope. By varying the potentiometer (which represents the "commanded voltage"), the motor voltage will go from 0 – 5 Volts. The fixed point value in resolution 2^{10} will be displayed in the watch window in motor_volts_M10. The equivalent floating point will be displayed motor_volts_float. The Motor Volts can be viewed as an output by adding a simple RC filter to the PWM output pin. Select the RC time constant such that it filters the 20 kHz carrier while not filtering the fundamental 50Hz. The scope shots will look as seen in Figure 2 and Figure 3.

NOTE: motor_volts_float is only used for "readability". It would not be used in normal motor control applications, rather you would work with the fixed point version motor_volts_M10.

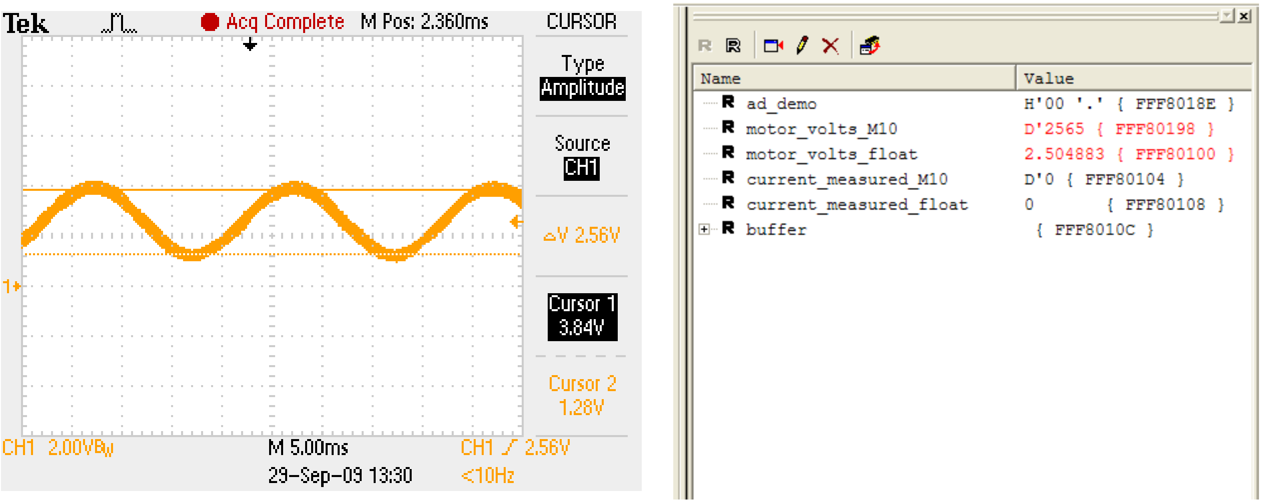

Figure 2: Sine wave (half voltage) generated post RC filter and Associated Watch Values.

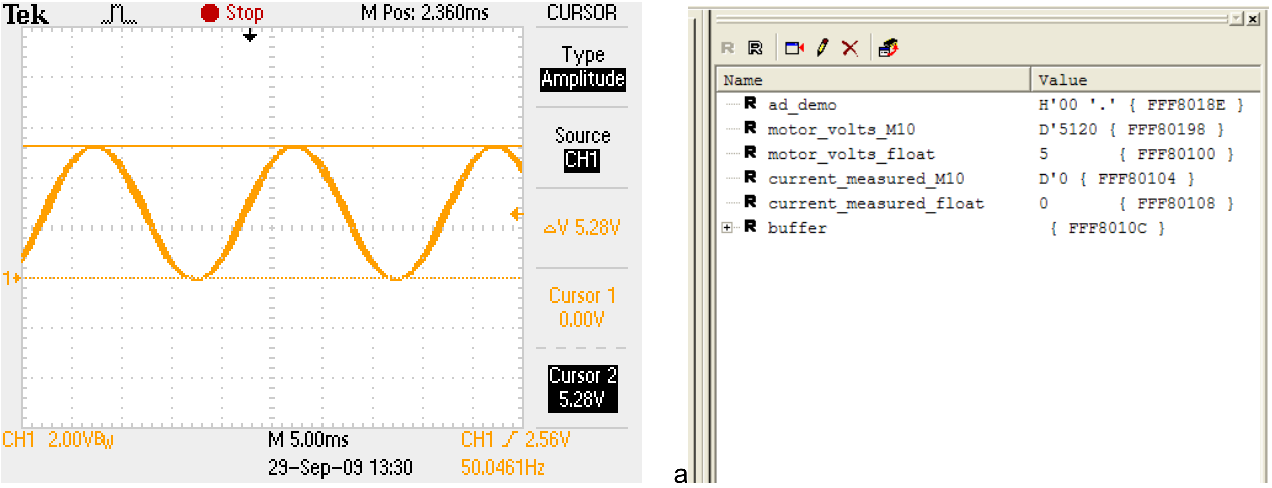

Figure 3: Sine wave (full voltage) generated post RC filter and Associated Watch Values.

3.4 Simulating Current Readings

When ad_demo=1, the demonstration uses the potentiometer to simulate the input from a current sensor. Since the variables are set for auto-update, simply double click the ad_demo entry and change the value to 1 in the dialog box. Once ad_demo equals 1, the potentiometer readings will be reflected in the current_measured_M10. The potentiometer is scaled to simulate a sensor that reads from 0 to 10 amperes full scale in resolution 2^{10}. The floating point value will be displayed in current_measured_float.

In the demonstration, we set the Over current value to 5 Amperes. Note that when you cross 5 Amperes, the string in the buffer goes from "Current OK" to "Over-Current"

NOTE: current_measured_float is only used for "readability". It would not be used in normal motor control applications, rather you would work with the fixed point version current_measured_M10.

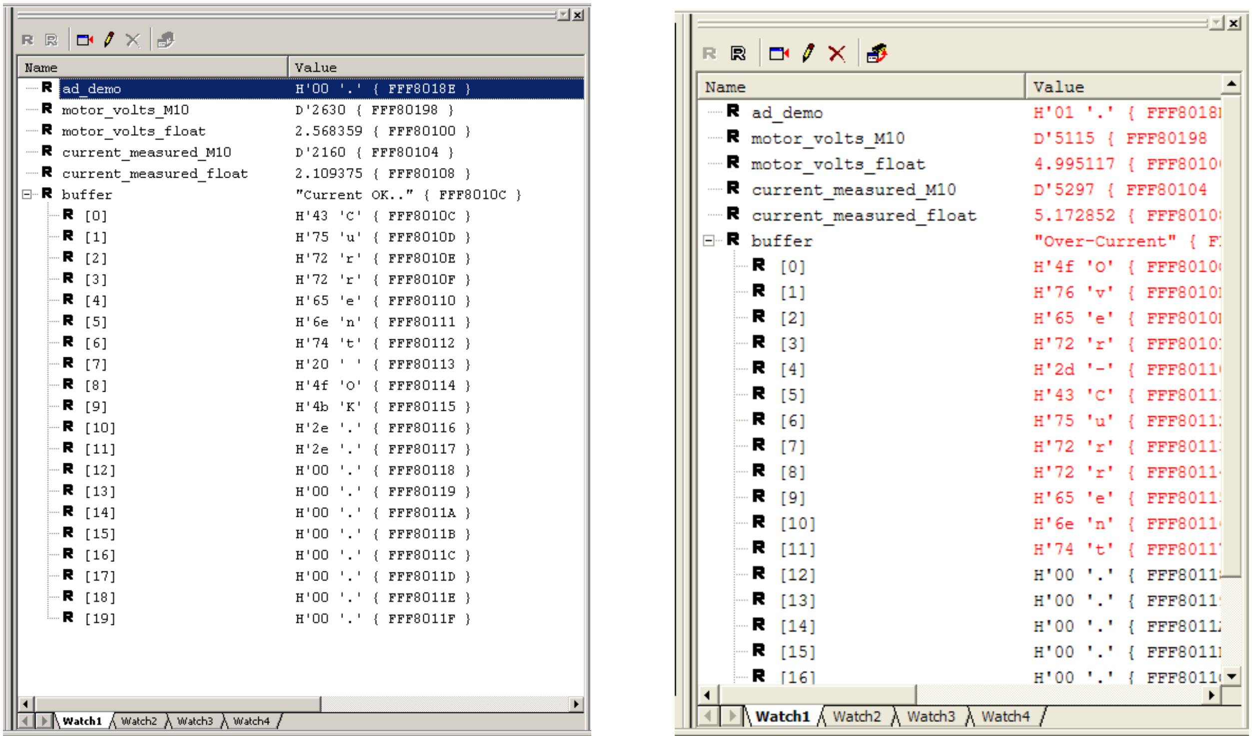

Figure 4: Watch Windows showing Normal and Over Current Detection.

4. Selecting Resolutions for Variables

It is important to select the proper resolutions when using a fixed point library. You can simply select a resolution and live with the resulting rounding error or you can select the maximum resolution, minimize the rounding error and provide the dynamic range required by your application.

In addition, you must be aware that large number may cause overflow errors in your calculations. It is recommended that you check the ranges of the math used on your fixed point numbers, including making sure some of the intermediate values do not cause overflow. Picking resolutions that do not have you "carrying around" excess fractional bits (i.e.bits not needed to meet the accuracy requirements of your system) will reduce the chances of overflowing your results when you add or do scaled multiply operations.

Before we jump into the hardware examples, it might be good to discuss some Fixed Point Basics, starting with notation.

Q*f* - This notation just designates the number of fractional bits (f) and not the size of the number. For example Q5 only says that there are 5 fractional bits, and does not say number of bits. Typically, Signed 16 or 32 bit variables are used.

Q*m.f* – This form define both the magnitude (m) and the Fractional part (f). A number like Q3.28 says that you have 3 integer bits and 28 fractional bits. With 1 sign bit, this fits in a signed 32 bit variable.

fx*m.b* – Similar to Qm.f, it use the second bit number (b) to describe the number of bits, this a number like fx1.6 is fixed point number with 1 magnitude bit and 15 fractional bits.

s:m:f – This describe the whole number, sign (s), magnitude (m) and fractional part (f). So a number like 0:8:0 describe an unsigned 8 bit integer.

Next we will look at some of the basic rules. In fixed point arithmetic we are basically doing exponential arithmetic, so we have to abide the rule for exponential arithmetic.

Adding & Subtracting: to add or subtract fixed point numbers, they must be the same resolution. For example to add a Q5 and a Q7 you must convert one of them. In general, you would convert the lower resolution up to the higher resolution, because it would simply mean shifting in 0's on the LSB end, and thus not affecting the actual answer.

Multiplying: to multiply, you simply multiply the numbers and add the Q factor to determine the resolution of the new number. In general this is not a problem for two reasons, a) most modern MCUs have a hardware multiplier and secondly, most libraries will use the correct intermediate value (for 16x16 = 32 bit result) and then return a normalized number.

Dividing: For most MCUs dividing is an issue since most divides are multi-clock. No way around it, you just need to minimize the number of divides. That being said, if you have a constant that you need to divide by, a one simple performance enhancement is to multiply by its inverse. This is easy to do with fixed point arithmetic.

Scaled Multiplication: This is a little unique. This is used when multiplying numbers of two different resolutions. For example if you want to multiply a number of Q5 and Q7. The resulting would be Q to the 12 and without scaled multiply, the library would not know what resolution to return. So for example suppose you want to multiply Voltage and Current to get power. P=IE, but I is in Q10 and E is in Q5; perfect if you need P in Q15 (probably in a 32 bit signed return value). But suppose you need P as a 16 bit signed result in Q8. You would called a scaled multiply with a scaling value of 5 (i.e. result will be Q12 you need to scale back by Q5 to get result in Q8).

Overflow: Overflow is something that you must contend with, but it is easy to avoid. Multiplying numbers of the same resolution is usually not a problem since they are normalized on return. Adding is a minor issue because in most control systems you multiply and divide. Overflow on One level of addition can be avoided by providing an additional bit. When we are selecting resolution we will always allow 1 bit which we will call "safety margin". This means you can add two numbers of the same resolution without worrying about overflow. If you need to add more than two numbers you will have to scale if your numbers will cause an overflow.

Finally, as a general rule, we will pick the size of variable, then pick the resolution we will need to fit this range into the magnitude of the number. Sometimes this means we will carry extra resolution bits, but this is usually not an issue since the library will multiply and normalize on its return, so it costs us little.

For the following example we will assume a 12 bit ADC with an input voltage range (V_{ref}) range of 5VDC. You can adjust the equations accordingly for 3.3V operation. Also Code in these examples if written for clarity.

Example 1: Voltage Sensor

Assume you need to monitor an incoming DC bus voltage in a lighting control design. To do this you use a simple voltage divider on the DC bus and apply it to the input of the ADC. Since this is a line powered application you need to divide it down by about 100:1. You're resulting schematic is shown in Figure 5 with V_BUS attached to the ADC.

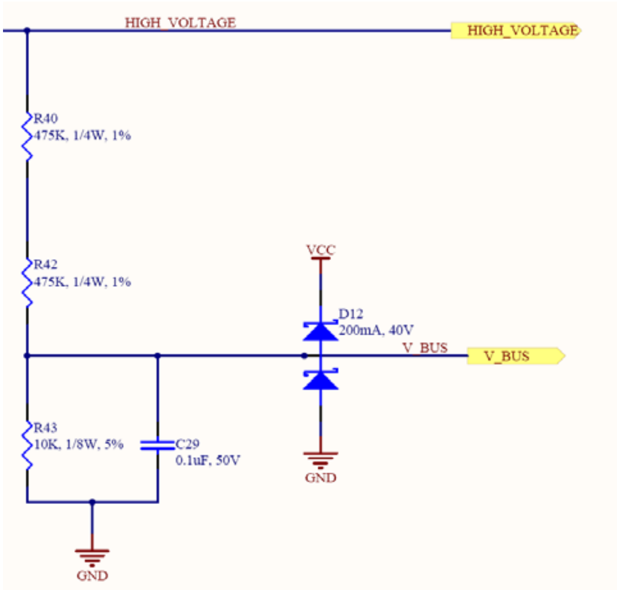

Figure 5: Voltage Monitor

You chose to do all you work with 16 Bit Fixed Point math, but you want to choose a resolution to minimize your error. You must evaluate the range of the number. Therefore maximum HIGH_VOLTAGE measurable is:

$$(Vref * (R40 + R42))/R43 = (5*950K)/10K = 480 \text{ volts}$$

In order to fit this into a 16 bit fixed point representation and cover the range, let's assume 1 bit for sign and one bit for "safety margin". Therefore we will use 14 bit and a resulting resolution of:

$$Log_2(2^{14}/480) = 5.09 \text{ or resolution } 5$$

This means to represent the maximum value of 480 volts, we should use FIX5 resolution. So 480 volts would result in a value of 15360.

Your ADC bit weight would then be 480/4096 or ~0.1171875. Using the same reasoning, we should then use a resolution of:

$Log_2(2^{14}/0.117875) = \sim17.084$ so round down ~ 17, so numbers read from ADC are in 2^{17} so 1 bit is 15360

So suppose we read 2048 from the ADC, intuitively we know that is 240VDC, but let's go through the math.

$(2048*0.117875 \text{ in } 2^{17})/ 2^{12})$ = HIGH_VOLTAGE in 2^5 so your HIGH_VOLTAGE is:

$(2048*15360)/2^{12} = 7680$, but remember, this is 2^5, so your voltage is $7680/2^5 = 240$VDC

Code wise with the Fixed Point library this would look something like this:

```
#define VBUS_ADC_BIT     15360L  // 0.117875 in 2^17
#define BUS_OVERVOLTAGE 12800L  // 400 volts in 2^5

FIX5  high_voltage = 0;         // 0 until read by ADC

void read_bus_voltage(void)
{
FIX17 raw_reading;

    raw_reading = get_AD_data(V_BUS);

// Fixed Point Multiply scaled back by 2^12, results in 2^5 value
    high_voltage = FIX_mul_SCALE12(raw_reading, VBUS_ADC_BIT);

}
```

Example 2: Current Sensor

Assume you need to monitor current in a motor winding in a motor control design. So you chose a high-side DCCT (DC Current Transducer). From the data sheet you see that it outputs a signal at 100mV/A. So your full current range is ±2.5/(100mV/A) or ±25A or a range of 50A.

NOTE: Remember, for these examples a reading of 2048 (1/2 full-scale) is actually 0A since this is a bi-directional current sensor (i.e. current flows and is measured in both directions)

You chose to do all you work with 16 Bit Fixed Point math, but you want to choose a resolution to minimize your error. So you must evaluate the range of the number. The DCCT determines the maximum current we can measure. In order to fit this into a 16 bit fixed point representation and cover the range, let's assume 1 bit for sine and one bit for "safety margin". Therefore we will use 14 bit and a resulting resolution of:

$$\text{Log}_2(2^{14}/50) = 8.35 \text{ so you use resolution } 8$$

This means to represent the maximum value of ±25 volts, we should use FIX8 resolution. So 25A would result in a value of 6400.

Your ADC bit weight would then be 25/4096 (12 bit ADC) or ~0.012207031. Using the same reasoning, we should then use a resolution of:

$\text{Log}_2(2^{14}/0.012207031)$ = TBD so chose 20, so numbers read from ADC are in 2^{20} so 1 bit is 12800

Assume we read 3072 from the ADC; now intuitively we know that this is 12.5A, but let's go through the math.

Subtract 0 current offset, 3072-2048 = 1024

(1024*0.012207031 in 2^{20})/ 2^{12}) = AMPS in 2^8 so your current is:

$(1024*12800)/2^{12} = 3200$, but remember, this is 2^8, so your current is $3200/2^8 = 12.5$A

Code wise with the Fixed Point library this would look something like this:

NOTE: the zero current offset is hard coded in this example. Actual Zero Current Offset could be determined in software at run time.

Comment on Coding Style: In this demo we chose to add a suffix on the variables that are in fixed point format. We chose to do this so we did not have to keep referring back to the declaration to determine its resolution. If we see a variable *xyz_M10*, we know it is in resolution 10. So one example of how this helps keep things straight is in the scaled multiply. If you multiply *abc_M5* and *def_M7* and you are putting it into a variable *xyz_M10*, you know you need to scale back by 2. Some may chose to use prefix such as *FIX10_abc*, either way it is a good idea to do whatever necessary to keep your resolutions correct.

```
#define I_ADC_BIT        12800L     // 0.012207031 in 2^20
#define OVER_CURRENT      5120L     // 20 volts in 2^8
#define ZERO_I_OFFSET     2048L     // Midpoint on 12 bit ADC

FIX8  motor_current = 0;            // 0 until read by ADC

void read_current(void)
{
FIX8  raw_reading;

    raw_reading = get_AD_data(V_BUS);  // 2^20 reading
    raw_reading -= ZERO_I_OFFSET;

// Multiply 2^20 bit weight scaled back by 2^12, results in 2^8 value
    motor_current = FIX_mul_SCALE12(raw_reading, I_ADC_BIT);

}
```

5. Multiple Build Configurations

The demo contains two Build setting. The normal build, simply called “Debug” will run the demo code as described above. The additional session is called Debug_with_vectors.

Debug_with_vectors will include a series of test vectors to be included in the build. These will run through a series of functions calls that attempt to exercise a large number of functions calls. This may be beneficial to software engineers using Fixed Point for the first time, to see the input numbers and the results of the fixed point function call.

NOTE: It is impossible to test all the permutations of the input values possible, so these are included as examples only. Any issues with the fixed point library should be reported to Renesas Tech Support.

6. Appendix A: Library Usage in HEW

This section will show a practical example of how the Filter Library may be integrated into an overall HEW Project.

6.1 Local to Project.

The SH2A Fixed Point Library is simple to add locally to any existing HEW project. To incorporate the Library locally into an existing project, drag and drop the SH2A Fixed Point Library folder into the project directory where you are working. Then simply navigate through the menu Project ➔ Add Files to Project ➔ Library (See Figure 6 through Figure 8). This will make it local to this project. For more information on Library files refer to the HEW User's Manual.

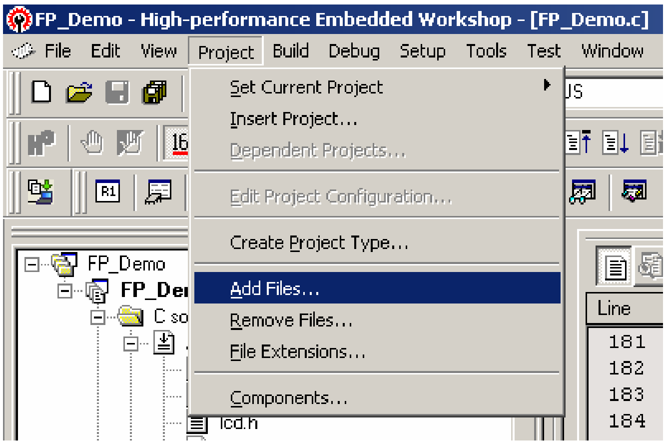

Figure 6: Add File

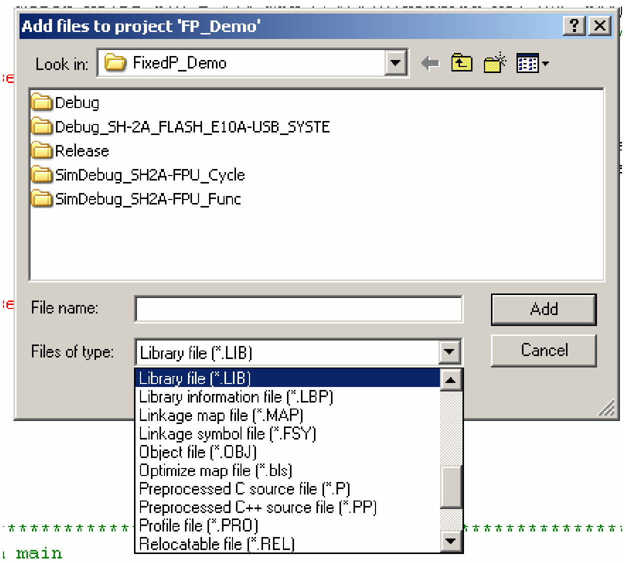

Figure 7: Select LIB type

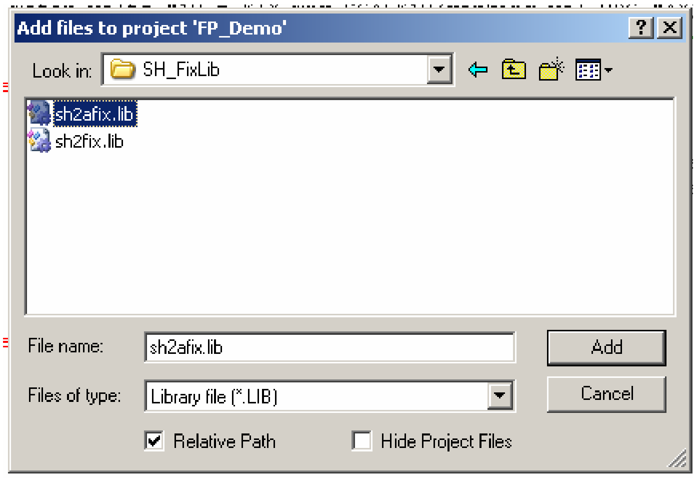

Figure 8: Select Fixed Point Library file

NOTE: Be sure to select the correct library based on device type being used, SH2 or SH2A.

6.2 As Library in Build Environment

The SH2A Fixed Point Library is simple to add as a common library to any existing HEW project. To incorporate the Library from a central library directory into an existing project, navigate through the menu Build ➔ SuperH RISC Engine Standard Toolchain ➔ Link/Library Tab ➔ Library Files ➔ Add (See figures Figure 9 through Figure 11). This project will now always use the .lib file at this directory location when a build is performed. In this example Library is in a folder named UserLib. For more information on Library files refer to the HEW User's Manual.

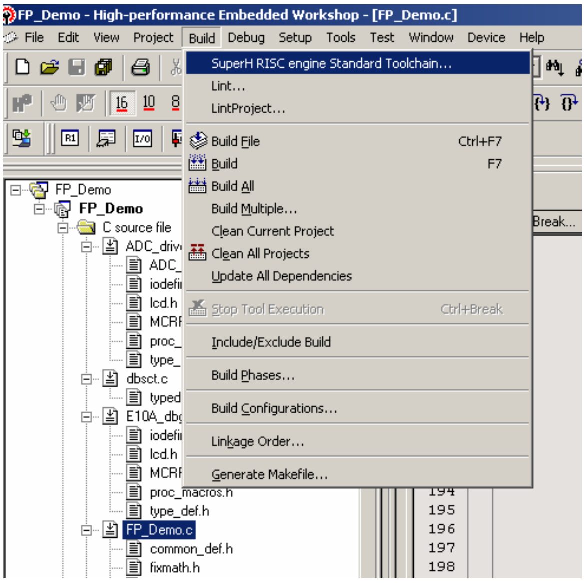

Figure 9: Build Toolchain Menu

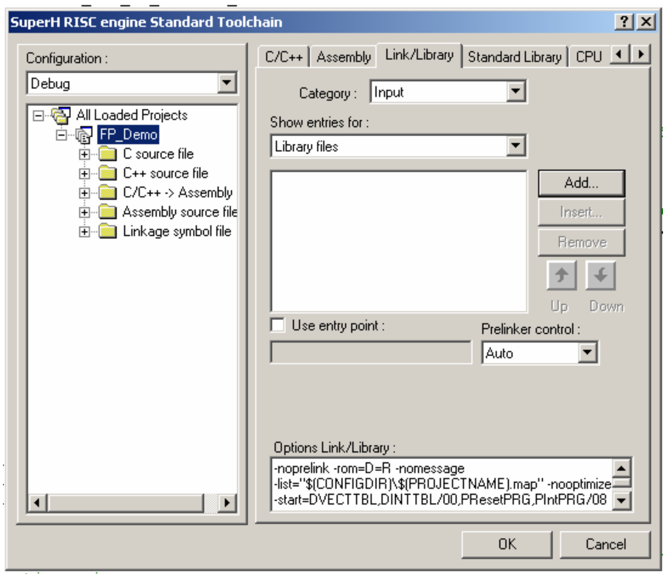

Figure 10: Link/Library Tab

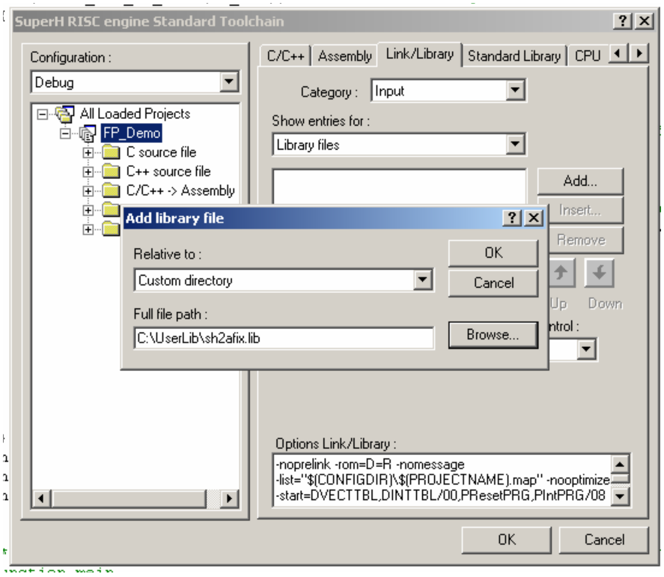

Figure 11:Adding Library Directory

7. Glossary

FPU – Floating Point Unit

QSG – Quick Start Guide

References

Fixed Point Library Users Manual, REJ05J0001-0101

Wikipedia - http://en.wikipedia.org/wiki/Fixed-point_arithmetic

ISO/IEC TR 18037:2008 – Programming Languages - C - Extensions to support Embedded Processors

Website and Support

Renesas Technology Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry
csc@renesas.com

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Sept.01.09	—	First edition issued