

IQ - Math Library

Introduction

In Chapter 4 we discussed the differences between fixed-point and floating-point processors and the influence of hardware support for the computing time of numerical mathematics. In Lab 4 we also benchmarked the performance of the F2833x for fixed-point and floating-point implementations.

The good thing with a Digital Signal Controller, such as the F2833x is that we can decide whether to generate fixed-point code or floating-point machine code, because the hardware of this device supports both worlds. There are not that many controllers in the market, which give us such flexibility!

However, the C2000 family of Digital Signal Controller includes also some other members without the additional support of a floating-point unit. There is one other member, the TMS320F28035, which has an additional hardware "Control Law Accelerator" (CLA), which is also based on floating-point hardware. But for all other members of the C2000 - family, for example the F281x and the F280x groups, we cannot rely on a floating-point hardware.

In Lab 4 we also realized, that the use of a floating-point library always leads to an extended execution time for each mathematical instruction, which involves floating-point data type variables. For a real-time control application, this extended calculation time is not very welcome. And, to make it worse, real-time applications are quite often very cost sensitive, so that a floating-point controller is out of the question.

The question is: Is there a better solution for mathematical tasks running on a controller without hardware support for floating-point variables?

Probably, you can guess the answer. In the case of the Texas Instruments C2000 - family there is.

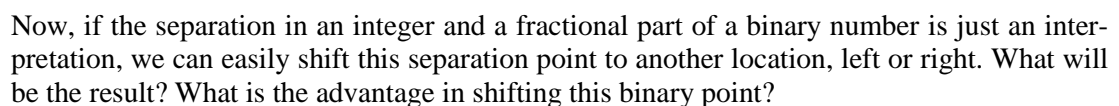
The solution is called "IQ-Math" -library (IQ = "Integer - Quotient"). This library is based purely on the F28x 32-bit fixed-point hardware unit. The "IQ-Math" solution takes advantage of the internal 64-bit resolution of the F28x fixed-point hardware-module. The word "library" might sound like a common C-compilers collection of supporting functions, but the IQ-Math library is different. All "functions" of that library are not "called" like any other library function. Instead, an optimized set of a few machine code lines is directly placed into the translation sequence of the machine code. For such functions the term "intrinsic" is used. The advantage is a very short execution time, because at execution time there is no function call, no return, no context save and context restore. Of course, the code size will grow slightly. However, all IQ-Math functions are optimized and consist of only a few words.

The current version (version 1.5a) of Texas Instruments "IQ-Math" - Library can be found in literature number "SPRC087" at www.ti.com.

Module Topics

IQ - Math Library	17-1
<i>Introduction.....</i>	<i>17-1</i>
<i>Module Topics.....</i>	<i>17-2</i>
<i>The “IQ”-Format.....</i>	<i>17-3</i>
<i>Which IQ-Format is best?</i>	<i>17-5</i>
<i>How do use IQ-Math?.....</i>	<i>17-6</i>
Standard ANSI - C 16-Bit Mathematics	17-7
Standard ANSI - C 32-Bit Mathematics	17-8
32-Bit IQ - Math Approach.....	17-9
<i>IQ - Math Library Functions.....</i>	<i>17-13</i>
<i>IQ- Math Application: Field Orientated Control.....</i>	<i>17-14</i>
Benchmark of IQ - formats	17-17
Benchmark Results	17-20
<i>IQ - Math summary</i>	<i>17-20</i>
<i>Lab 17: IQ - Math based low - pass filter</i>	<i>17-21</i>
Objective.....	17-21
Procedure	17-22
Install IQMath.....	17-22
Open Project	17-23
Build, Load and Run	17-23
Add code for ADC - Initialization	17-23
Build, Load and Run	17-25
Add a sampling buffer	17-26
Build, Load and Run	17-27
Add the low - pass filter code	17-28
MATLAB Filter Coefficient Calculation.....	17-29
Final Build, Load and Run.....	17-31
Benchmark IQ-Math and Floating-Point Filter code	17-32
Summary	17-33

Based on this special F28x IQ - fixed point machine code set, we can qualify the IQ-Math library to be comparable to a solution based on floating-point hardware. Therefore, Texas Instruments calls this library rightly "a virtual floating-point library".



The answer is, we can adjust our number system to have a more dynamic range of the numbers, or we can optimize our number system to give a higher resolution. The term "higher resolution" means that we can reduce the step size between two consecutive numbers in a selected IQ-format. The term "dynamic range" refers to the difference between the most negative and most positive member of a given IQ-scale.

The next slide (Slide 17-3) gives an example, how a 4-bit number can be used in four different IQ-Formats:

IQ – Range and Resolution			
4-bit number:			
Format	Most Negative	Most Positive	Resolution (step size)
I1Q3	1.000 -1	0.111 +0.875	0.001 0.125
I2Q2	10.00 -2	01.11 +1.75	00.01 0.25
I3Q1	100.0 -4	011.1 +3.5	000.1 0.5
I4Q0	1000 -8	0111 +7	0001 1

- Trade - Off between Range and Resolution
- Note: Integer Format (I4Q0) is a subset of IQ-Math

17 - 3

Depending on the requirements of an application and its control code, we can optimize the number system. If we need more dynamic range, we would prefer more integer bits. If we should need more resolution, a format with more fractional bits would make sense. This decision must be made by the software designer. In most cases it is a trade-off between range and resolution, you cannot get both high dynamic range and high resolution!

The decision, which IQ-Format shall be used for a given task, is one of the most important decisions, which must be made by the programmer of a control task. At the end of this chapter, we will inspect a real-world application with a benchmark to select the proper IQ-format.

Note: The last line in Slide 17-3, which shows the I4Q0 - Format, is nothing more than the standard signed integer format for a 4-bit number in 2's-complements. Therefore we can state, that the format "signed integer" is just a subset of a more general IQ - number system!

Of course, in real-world, we do not operate on 4-bit numbers; the typical size for a control application is either 16-bit or even 32-bit numbers. For simplification, we used a 4-bit example in Slide 17-3.

Which IQ-Format is best?

The IQ-number selection is not a static one. The software engineer can decide which part of a code solution should be executed on IQ-Format 'A' and which one on format 'B'.

Example: A software - project might be based by default on I8Q24 - numbers, which gives all numbers a dynamic range of $-128...+127.999$ and a resolution of 2^{-24} . If, for some reason, at a certain point in the control code this dynamic range is not sufficient, we can easily change the number system to another system, for example into I12Q20. Of course, we have to take into account the reduced resolution for that part of the code.

Texas Instruments offers a wide range of libraries, free to download, which are based on IQ-Math. All libraries feature an interface to language C, e.g. you can call any of these IQ-Functions like a common C - function. Please note that all these IQ-functions are "intrinsic" as discussed at the beginning of this chapter. The prototypes of the functions expect input data in a certain IQ-Format, such as "I1Q15" or "I8Q24" and so on. Return values are also delivered in such a format. It is your responsibility to adjust your input- and output-variables to the expected data format. With the help of the previous slides you should now be able to understand these requirements.

The next slide (Slide 17-4) shows a set of libraries, offered by Texas - Instruments Libraries; most of them are based on IQ-Math:

F28x Signal Processing Libraries	
Signal Processing Libraries & Applications Software	Literature #
ACI3-1: Control with Constant V/Hz	SPRC194
ACI3-3: Sensored Indirect Flux Vector Control	SPRC207
ACI3-3: Sensored Indirect Flux Vector Control (simulation)	SPRC208
ACI3-4: Sensorless Direct Flux Vector Control	SPRC195
ACI3-4: Sensorless Direct Flux Vector Control (simulation)	SPRC209
PMSM3-1: Sensored Field Oriented Control using QEP	SPRC210
PMSM3-2: Sensorless Field Oriented Control	SPRC197
PMSM3-3: Sensored Field Oriented Control using Resolver	SPRC211
PMSM3-4: Sensored Position Control using QEP	SPRC212
BLDC3-1: Sensored Trapezoidal Control using Hall Sensors	SPRC213
BLDC3-2: Sensorless Trapezoidal Drive	SPRC196
DCMOTOR: Speed & Position Control using QEP without Index	SPRC214
Digital Motor Control Library (F/C280x)	SPRC215
Communications Driver Library	SPRC183
DSP Fast Fourier Transform (FFT) Library	SPRC081
DSP Filter Library	SPRC082
DSP Fixed-Point Math Library	SPRC085
DSP IQ Math Library	SPRC087
DSP Signal Generator Library	SPRC083
DSP Software Test Bench (STB) Library	SPRC084
C28x FPU Fast RTS Library	SPRC664
C2833x C/C++ Header Files and Peripheral Examples	SPRC530

Available from TI Website ⇒ <http://www.ti.com/c2000>

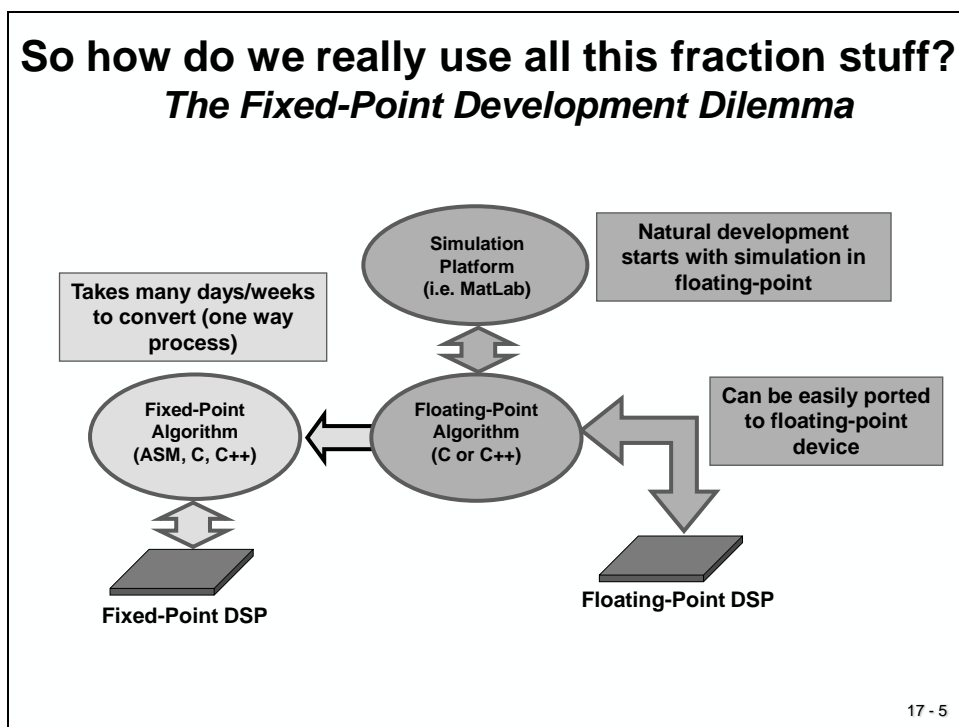
17 - 4

How do use IQ-Math?

Implementing complex digital control algorithms on a Digital Signal Controller (DSC), or any other DSP capable processor, typically we come across the following issues:

- Algorithms are typically developed using floating-point maths
- Floating-point devices are more expensive than fixed-point devices
- Converting floating-point algorithms to a fixed-point device is very time consuming
- Conversion process is one way and therefore backward simulation is not always possible

The diagram below illustrates a typical development scenario in use today:



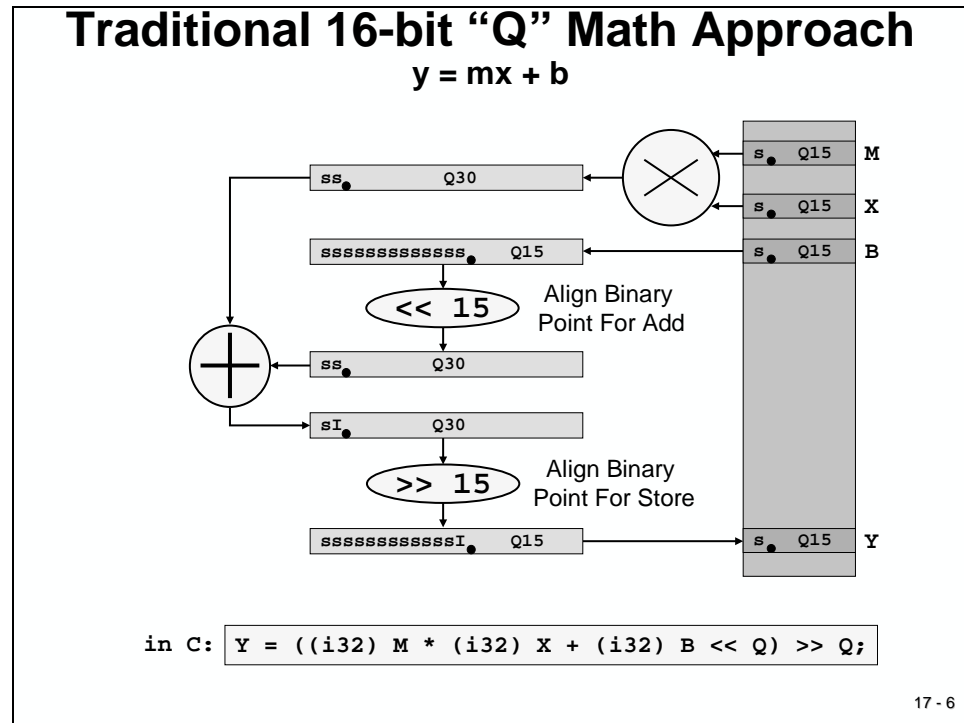
The design may initially start with a simulation (i.e. MATLAB) of a control algorithm, which typically would be written in floating-point math (C or C++). This algorithm can be easily ported to a floating-point device. However, because of the commercial reality of cost constraints, most likely a 16-bit or 32-bit fixed-point device would be used in many target systems.

The effort and skill involved in converting a floating-point algorithm to function using a 16-bit or

32-bit fixed-point device is quite significant. A great deal of time (many days or weeks) would be needed for reformatting, scaling and coding the problem. Additionally, the final implementation typically has little resemblance to the original algorithm. Debugging is not an easy task and the code is not easy to maintain or document.

Standard ANSI - C 16-Bit Mathematics

If the processor of your choice is a 16-bit fixed-point and you do not want to include a lot of library functions in your project, a typical usage of binary fractions is shown next. We assume that the task is to solve the equation $Y = MX + B$. This type of equation can be found in almost every mathematical approach for digital signal processing.



The diagram shows the transformations, which are needed to adjust the binary point in between the steps of this solution. We assume that the input numbers are in I1Q15-Format. After M is multiplied by X , we have an intermediate product in I2Q30-format. Before we can add variable B , we have to align the binary point by shifting b 15 times to the left. Of course we need to typecast B to a 32-bit long first to keep all bits of B . The sum is still in I2Q30-format. Before we can store back the final result into Y we have to right shift the binary point 15 times.

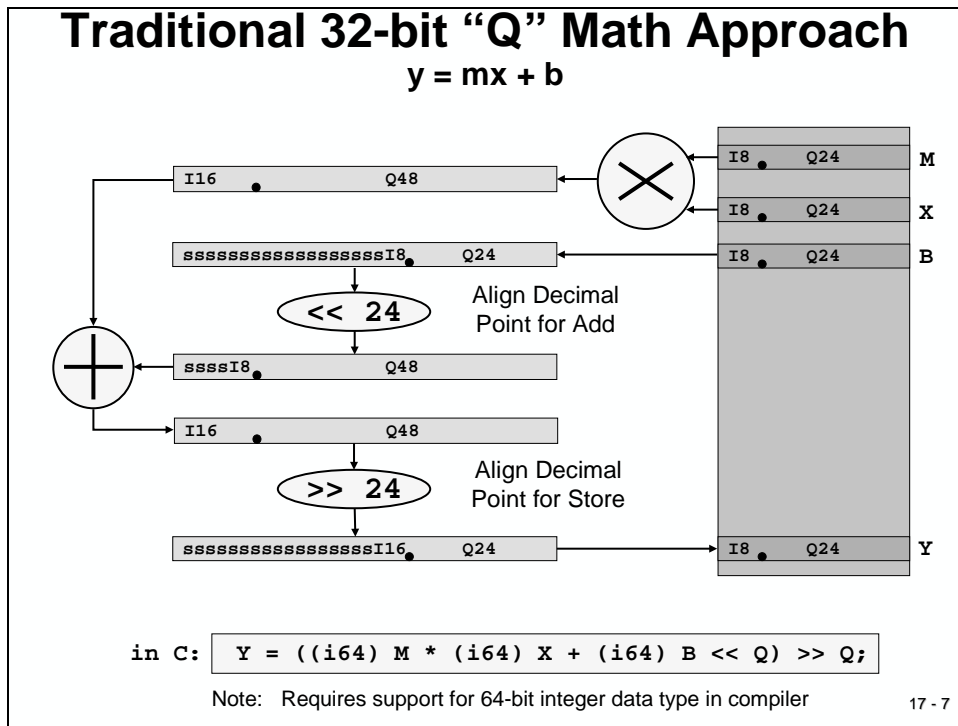
The last line of the slide shows the equivalent syntax in ANSI-C. “i32” stands for a 32-bit integer, usually called ‘long’. ‘Q’ is a global constant and gives the number of fractional bits; in our example Q is equal to 15.

The disadvantage of this Q15 - approach is its limitation of only 16 bits. A lot of projects for digital signal processing and digital control will not be able to achieve stable behavior due to the lack of either resolution or dynamic range.

The F28x as a 32-bit processor can do better - we just have to expand the scheme to 32-bit binary fractions!

Standard ANSI - C 32-Bit Mathematics

The next diagram is an expansion of the previous scheme to 32-bit input values. Again, the task is to solve equation $Y = MX + B$. In the following example the input numbers are in an I8Q24-format.



The big problem with the translation into ANSI-C code is that we do not have a 64-bit integer data type! Although the last line of the slide looks pretty straight forward, we can't apply this line to a standard C-compiler!

What now?

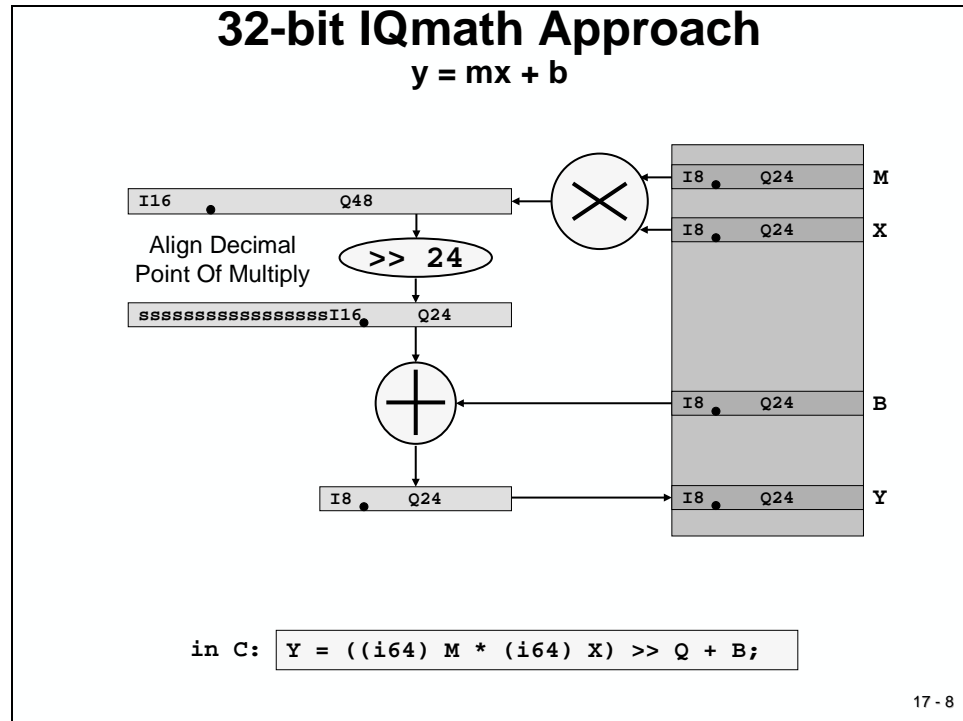
The rescue is the internal hardware arithmetic (Arithmetic Logic Unit and 32-bit by 32-bit Hardware Multiply Unit) of the F28x. These units are able to deal with 64-bit intermediate results in a very efficient way. Dedicated assembly language instructions for multiply and add operations are available to operate on the integer part and the fractional part of the 64-bit number.

To be able to use these advanced instructions, we have to learn about the F28x assembly language in detail. Eventually your professor offers an advanced course in F28x assembly language programming -

OR, just use Texas Instruments "IQ-Math"-library, which is doing nothing more than using these advanced assembly instructions!

32-Bit IQ - Math Approach

The first step to solve the 64-bit dilemma is to refine the last diagram for the 32-bit solution of $Y = MX + B$. As you can see from the next slide, the number of shift operations is reduced to 1. Again, the C-line includes a 64-bit 'long', which is not available in standard C.



The "IQ"-Math approach 'redefines' the multiply operation to use the advantages of the internal hardware of the C28x. As stated, the F28x is internally capable of handling 64-bit fixed-point numbers with dedicated instruction sets. Texas Instruments provides a collection of intrinsic functions, one of them to replace the standard multiply operation by an `_IQmpy(M,X)` -line. Intrinsic means, we do not 'call' a function with a lot of context save and restore; instead the machine code instructions are directly included in our source code.

As you can see from the next slide the final C-code looks much better now without the cumbersome shift operations that we have seen in the standard C approach.

AND: The execution time of the final machine code for the whole equation $Y = MX + B$ takes only 7 cycles; with a 150MHz F2833x, this translates into 46 nanoseconds!

IQmath Approach

Multiply Operation

$$Y = ((i64) M * (i64) X) >> Q + B;$$

Redefine the multiply operation as follows:

$$_IQmpy(M,X) == ((i64) M * (i64) X) >> Q$$

This simplifies the equation as follows:

$$Y = _IQmpy(M,X) + B;$$

C28x compiler supports “_IQmpy” intrinsic; assembly code generated:

```

MOVL    XT,@M
IMPYL   P,XT,@X      ; P  = low  32-bits of M*X
QMPYL   ACC,XT,@X     ; ACC = high 32-bits of M*X
LSL64   ACC:P,#(32-Q) ; ACC = ACC:P << 32-Q
                        ; (same as P = ACC:P >> Q)
ADDL    ACC,@B        ; Add B
MOVL    @Y,ACC        ; Result = Y = _IQmpy(M*X) + B
; 7 Cycles

```

17 - 9

Let us have a closer look to the assembly instructions used in the example above.

The first instruction ‘MOVL XT,@M’ is a 32-bit load operation to fetch the value of M into a temporary register ‘XT’.

Next, ‘XT’ is multiplied by another 32-bit number taken from variable X (‘IMPYL P,XT,@X’). When multiplying two 32-bit numbers, the result is a 64-bit number. In the case of this instruction, the lower 32-bit of the result are stored in a register ‘P’.

The upper 32 bits are stored with the next instruction (‘QMPYL ACC,XT,@X’) in the ‘ACC’ register. ‘QMPYL’ is doing the same multiplication once more but keeps the upper half of the result only. At the end, we have stored all 64 bits of the multiplication in the register combination ACC:P.

What follows is the adjustment of the binary point. The 64-bit result in ACC:P is in I16Q48-fractional format. Shifting it 32-24 times to the left, we derive an I8Q56-format. The instruction ‘ADDL ACC,@B’ uses only the upper 32 Bits of the 64-bit, thus reducing our fractional format from I8Q56 to I8Q24 - which is the same format as we use for B and all our variables!

The whole procedure takes only 7 cycles!

The next slide compares the different approaches. The IQ-Math library also defines a new data type ‘_iq’ to simplify the definition of fractional data. If you choose to use C++ the floating-point equation and the C++ equation are identical! This is possible due to the overload feature of C++. The floating-point multiply operation is overloaded with its IQ-Math replacement - the code looks ‘natural’.

IQmath Approach

It looks like floating-point!

Floating-Point	<pre>float Y, M, X, B; Y = M * X + B;</pre>
Traditional Fix-Point Q	<pre>long Y, M, X, B; Y = ((i64) M * (i64) X + (i64) B << Q)) >> Q;</pre>
“IQmath” In C	<pre>_iq Y, M, X, B; Y = _IQmpy(M, X) + B;</pre>
“IQmath” In C++	<pre>iq Y, M, X, B; Y = M * X + B;</pre>

“IQmath” code is easy to read!

17 - 10

This technique opens the way to generate a unified source code that can be compiled in a floating-point representation as well as into a fixed-point output solution. No need to translate a floating-point simulation code into a fixed-point implementation - the same source code can serve both worlds.

IQmath Approach

GLOBAL_Q simplification

User selects "Global Q" value for the whole application

● GLOBAL_Q

based on the required dynamic range or resolution, for example:

GLOBAL_Q	Max Val	Min Val	Resolution
28	7.999 999 996	-8.000 000 000	0.000 000 004
24	127.999 999 94	-128.000 000 00	0.000 000 06
20	2047.999 999	-2048.000 000	0.000 001

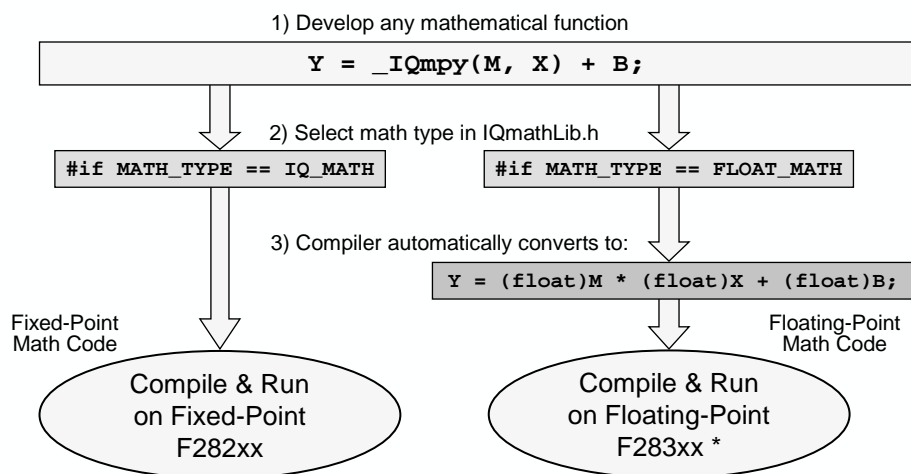
```
#define GLOBAL_Q 18 // set in "IQmathLib.h" file
_iq Y, M, X, B;
Y = _IQmpy(M,X) + B; // all values are in Q = 18
```

The user can also explicitly specify the Q value to use:

```
_iq20 Y, M, X, B;
Y = _IQ20mpy(M,X) + B; // all values are in Q = 20
```

17 - 11

IQmath Provides Compatibility Between Floating-Point and Fixed-Point



All "IQmath" operations have an equivalent floating-point operation

* Can also compile floating-point code on any floating-point compiler (e.g., PC, Matlab, fixed-point w/ RTS lib) 

IQ - Math Library Functions

The next two slides summarize the existing library functions of IQ-Math.

IQmath Library: Math & Trig Functions

Operation	Floating-Point	"IQmath" in C	"IQmath" in C++
type	float A, B;	_iq A, B;	iq A, B;
constant	A = 1.2345	A = _IQ(1.2345)	A = IQ(1.2345)
multiply	A * B	_IQmpy(A, B)	A * B
divide	A / B	_IQdiv(A, B)	A / B
add	A + B	A + B	A + B
subtract	A - B	A - B	A - B
boolean	>, >=, <, <=, ==, !=, &&,	>, >=, <, <=, ==, !=, &&,	>, >=, <, <=, ==, !=, &&,
trig and power functions	sin(A), cos(A) sin(A*2pi), cos(A*2pi) asin(A), acos(A) atan(A), atan2(A, B) atan2(A, B)/2pi sqrt(A), 1/sqrt(A) sqrt(A*A + B*B) exp(A)	_IQsin(A), _IQcos(A) _IQsinPU(A), _IQcosPU(A) _IQasin(A), _IQacos(A) _IQatan(A), _IQatan2(A, B) _IQatan2PU(A, B) _IQsqrt(A), _IQisqrt(A) _IQmag(A, B) _IQexp(A)	IQsin(A), IQcos(A) IQsinPU(A), IQcosPU(A) IQasin(A), IQacos(A) IQatan(A), IQatan2(A, B) IQatan2PU(A, B) IQsqrt(A), IQisqrt(A) IQmag(A, B) IQexp(A)
saturation	if(A > Pos) A = Pos if(A < Neg) A = Neg	_IQsat(A, Pos, Neg)	IQsat(A, Pos, Neg)

Accuracy of functions/operations approx ~28 to ~31 bits

17 - 13

IQmath Library: Conversion Functions

Operation	Floating-Point	"IQmath" in C	"IQmath" in C++
iq to iqN	A	_IQtoIQN(A)	IQtoIQN(A)
iqN to iq	A	_IQNtoIQ(A)	IQNtoIQ(A)
integer(iq)	(long) A	_IQint(A)	IQint(A)
fraction(iq)	A - (long) A	_IQfrac(A)	IQfrac(A)
iq = iq*long	A * (float) B	_IQmpyl32(A, B)	IQmpyl32(A, B)
integer(iq*long)	(long) (A * (float) B)	_IQmpyl32int(A, B)	IQmpyl32int(A, B)
fraction(iq*long)	A - (long) (A * (float) B)	_IQmpyl32frac(A, B)	IQmpyl32frac(A, B)
qN to iq	A	_QNtoIQ(A)	QNtoIQ(A)
iq to qN	A	_IQtoQN(A)	IQtoQN(A)
string to iq	atof(char)	_atolQ(char)	atolQ(char)
IQ to float	A	_IQtoF(A)	IQtoF(A)
IQ to ASCII	sprintf(A, B, C)	_IQtoA(A, B, C)	IQtoA(A, B, C)

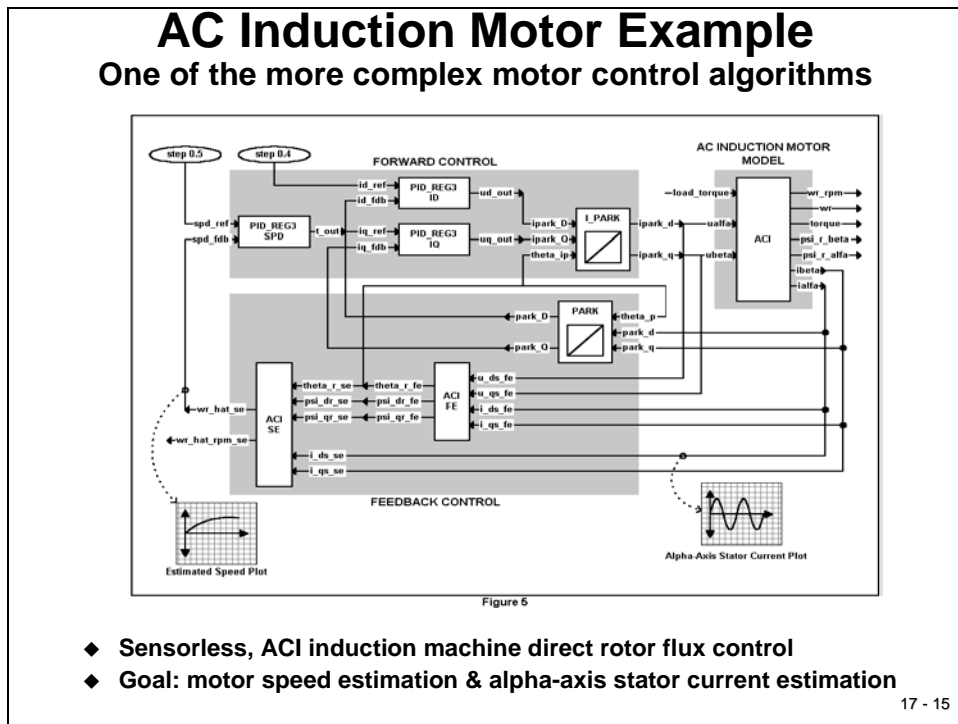
IQmath.lib > contains library of math functions
 IQmathLib.h > C header file
 IQmathCPP.h > C++ header file

17 - 14

IQ- Math Application: Field Orientated Control

The next slides are just to demonstrate the ability of “IQ-Math” to solve advanced numeric calculations in real time. The example is taken from the area of digital motor control. We will not go into the details of the control scheme and we will not discuss the various options to control an electrical motor. If you are a student of an electrical engineering degree you might be familiar with these control techniques. Eventually your university also offers additional course modules with this topic. The field of motor and electrical drive control is quite dynamic and offers a lot of job opportunities.

The next slide is a block diagram of a control scheme for an alternating current (AC) induction motor. These types of motors are based on a three-phase voltage system. Modern control schemes are introduced these days to improve the efficiency of the motor. One principle, called “Field Orientated Control”, incorporates “Space Vector Modulation”, a voltage modulation technique to drive an inverter, which is quite popular today. In fact this theory is almost 70 years old now, but in the past it was impossible to realize a real time control due to the lack of computing power. Now with a controller like the F2833x, it can be implemented!



The core control system consists of three digital PID-controllers, one for the speed control of the motor (“PID_REG3 SPD”), one to control the torque (“PID_REG3 IQ”) and one for the flux (“PID_REG3 ID”). Between the control loops and the motor two co-ordinate transforms are performed (“PARK” and “I_PARK”).

Let us have a look into a standard C implementation of the PARK transform, which converts a 3-D vector to a 2-D vector. For now, it is not necessary to fully understand this transform, just have a look into the mathematical operations involved.

All variables are data type “float” and the functions included are:

- Six multiply operations,
- Two trigonometric function calls,
- An addition and
- A subtraction.

This code can easily be compiled by any standard C compiler and downloaded into a simulation or into any processor, for example the F2833x. It will work, but it will not be the most efficient way to use the F2833x because it will involve floating-point library function calls that will consume a considerable amount of computing time.

AC Induction Motor Example

Park Transform – floating-point C code

```
#include "math.h"

#define TWO_PI 6.28318530717959

void park_calc(PARK *v)
{
    float cos_ang , sin_ang;
    sin_ang = sin(TWO_PI * v->ang);
    cos_ang = cos(TWO_PI * v->ang);

    v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
    v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
}
```

17 - 16

With the “IQ-Math” library we can improve the code for the C28x, as shown at the next slide. Of course, we have to replace all float function calls by “IQ-Math” intrinsic functions.

All variables are now of data type “_iq”, the sine and cosine function calls are replaced by their intrinsic replacements as well as the six multiply operations.

The constant “TWO_PI” will be converted into the standard IQ-format with the conversion function “_IQ()”. This way the number 6.28 will be translated into the correct fixed-point scale before it is used during compilation.

The resulting code will be compiled into a much denser and faster code for the C28x. Of course, a little bit of coding is still needed to convert an existing floating-point code into the “IQ-Math” C- code.

Fortunately, the structure of the two program versions is identical, which helps to keep a development project consistent and maintainable, for both the floating-point and the fixed-point implementations.

AC Induction Motor Example

Park Transform - converting to “IQmath” C code

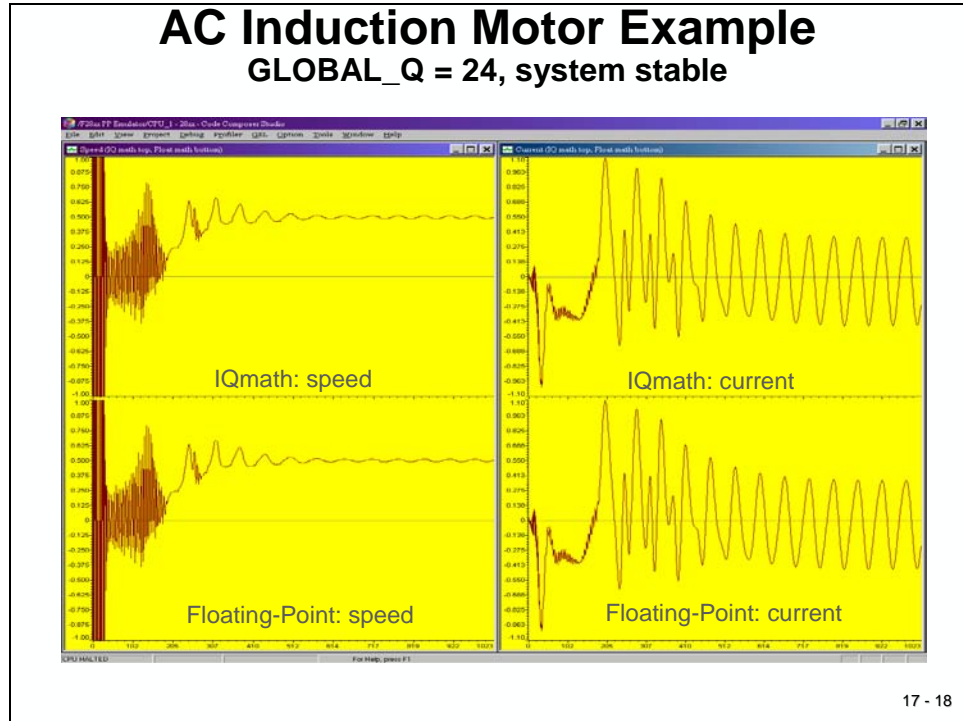
```
#include "math.h"
#include "IQmathLib.h"
#define TWO_PI _IQ(6.28318530717959)
void park_calc(PARK *v)
{
    _iq cos_ang , sin_ang;
    sin_ang = _IQsin(_IQmpy(TWO_PI , v->ang));
    cos_ang = _IQcos(_IQmpy(TWO_PI , v->ang));

    v->de = _IQmpy(v->ds , cos_ang) + _IQmpy(v->qs , sin_ang);
    v->qe = _IQmpy(v->qs , cos_ang) - _IQmpy(v->ds , sin_ang);
}
```

17 - 17

Benchmark of IQ - formats

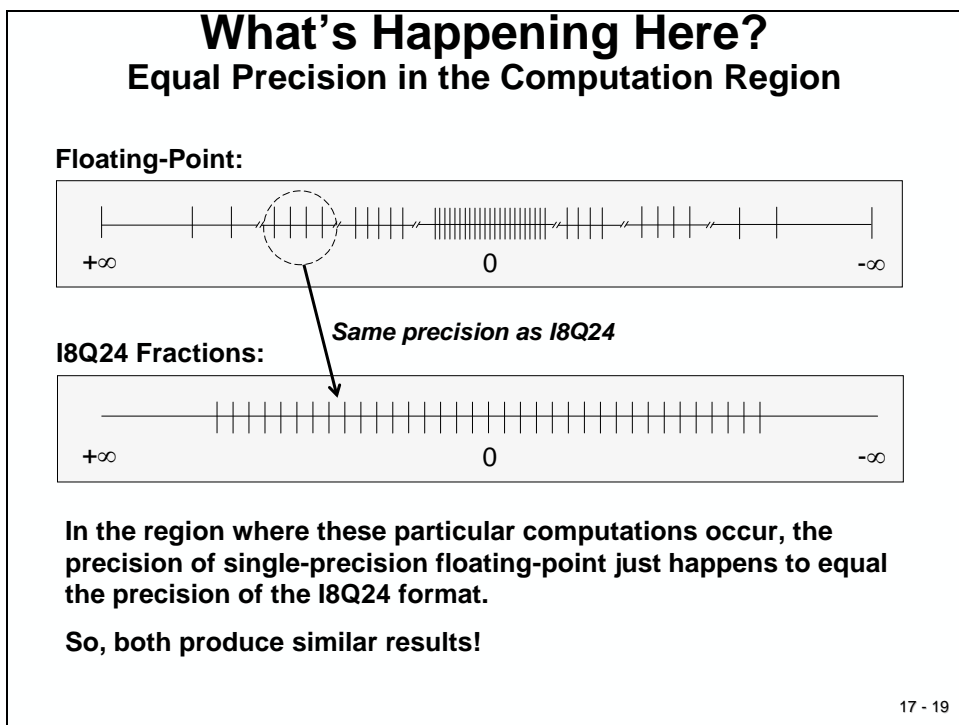
The complete AC-induction motor system was coded using "IQ-Math". Based on analysis of coefficients in the system, the largest coefficient had a value of 33.3333. This indicated that a minimum dynamic range of 7bits (+/-64 range) was required. Therefore, this translated to a GLOBAL_Q value of $32-7 = 25(Q25)$. Just to be safe, the initial simulation runs were conducted with GLOBAL_Q = 24 (Q24) value.



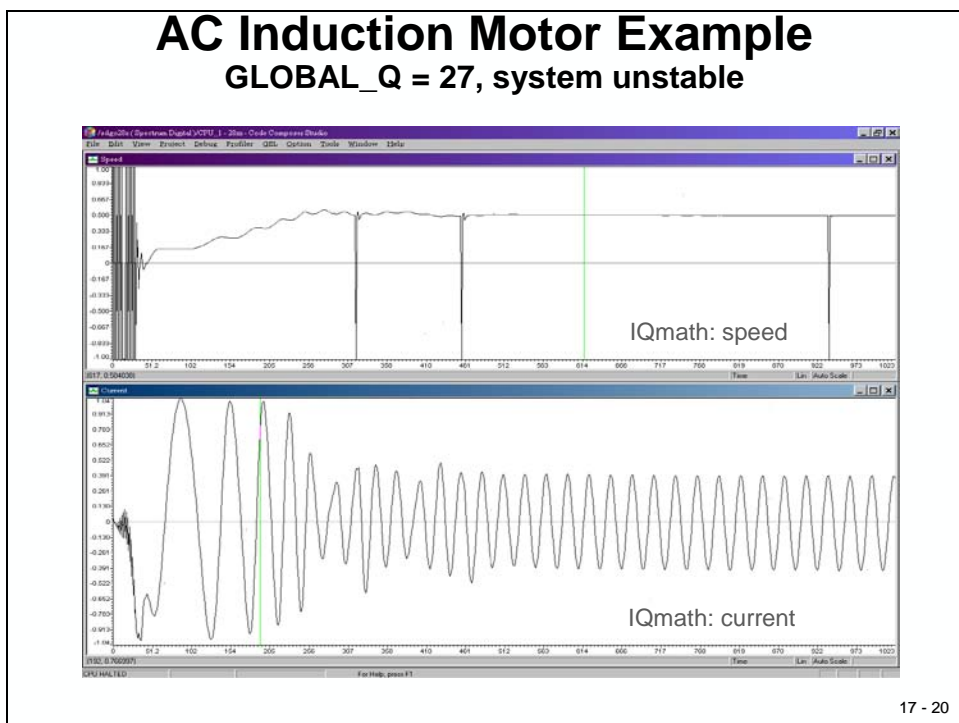
Slide 17-18 compares the results for a floating-point code and an I8Q24-code. The left hand side diagrams show the speed response of the motor to reach a set point. The left hand side diagrams are the measurement results in one of the three phase currents each.

We can say that the results are almost identical; there is no difference in the control loop for a floating-point controller and an I8Q24 fixed-point device. The reason for this similarity is shown at the next slide (Slide 17-19).

In the computation region of numbers, which are used in this first test, the resolution (or precision) of fixed-point and floating-point is equal - and so are the results!

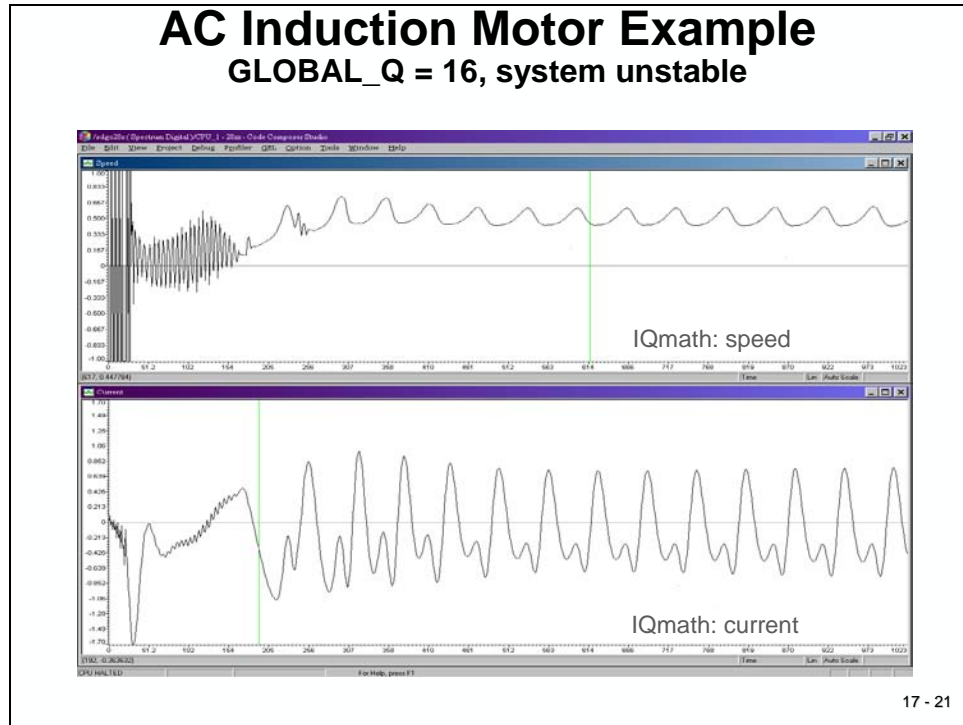


Next, the whole AC induction motor solution was investigated for stability and dynamic behavior by changing the global Q value. With a 32-bit fixed-point data type we can modify the fractional part between 0 bit (“Q0”) and 31 bits (“Q31”). The following slide shows the results for an I5Q27 - math system:



The system becomes unstable with spikes in the speed response, because of the reduced dynamic range and resulting overflows in the numbering scale.

The next slide (Slide 17-21) shows the results for an I16Q16 - solution. Here the system becomes unstable, because of the limited resolution of a step size of the numbers.



All the results are summarized below. As you can see, there is an area, in which all tests led to a stable operating mode of the motor. The two other areas showed an increasing degree of instability, caused by either not enough dynamic range in the integer part or not enough fractional resolution of the numbering system.

AC Induction Motor Example

Q stability range

Q range	Stability Range
Q31 to Q27	Unstable (not enough dynamic range)
Q26 to Q19	Stable
Q18 to Q0	Unstable (not enough resolution, quantization problems)

The developer must pick the right GLOBAL_Q value!

17 - 22

Benchmark Results

Here is a summary of the results for the control code of a "field orientated control" (FOC) of an AC - induction motor:

AC Induction Motor Example Performance comparisons			
Benchmark	C28x C floating-point std. RTS lib (150 MHz)	C28x C floating-point fast RTS lib (150 MHz)	C28x C IQmath v1.4d (150 MHz)
B1: ACI module cycles	401	401	625
B2: Feedforward control cycles	421	371	403
B3: Feedback control cycles	2336	792	1011
Total control cycles (B2+B3)	2757	1163	1414
% of available MHz used (20 kHz control loop)	36.8%	15.5%	18.9%

Notes: C28x compiled on codegen tools v5.0.0, -g (debug enabled), -o3 (max. optimization)
fast RTS lib v1.0beta1
IQmath lib v1.4d

17 - 23

IQ - Math summary

IQmath Approach Summary "IQmath" + fixed-point processor with 32-bit capabilities =	
<ul style="list-style-type: none"> ◆ Seamless portability of code between fixed and floating-point devices <ul style="list-style-type: none"> ◆ User selects target math type in "IQmathLib.h" file <ul style="list-style-type: none"> • #if MATH_TYPE == IQ_MATH • #if MATH_TYPE == FLOAT_MATH ◆ One source code set for simulation vs. target device ◆ Numerical resolution adjustability based on application requirement <ul style="list-style-type: none"> ◆ Set in "IQmathLib.h" file <ul style="list-style-type: none"> • #define GLOBAL_Q 18 ◆ Explicitly specify Q value <ul style="list-style-type: none"> • _iq20 X, Y, Z; ◆ Numerical accuracy without sacrificing time and cycles ◆ Rapid conversion/porting and implementation of algorithms 	
<p><i>IQmath library is freeware - available from TI DSP website</i> <u>http://www.ti.com/c2000</u></p>	

17 - 24

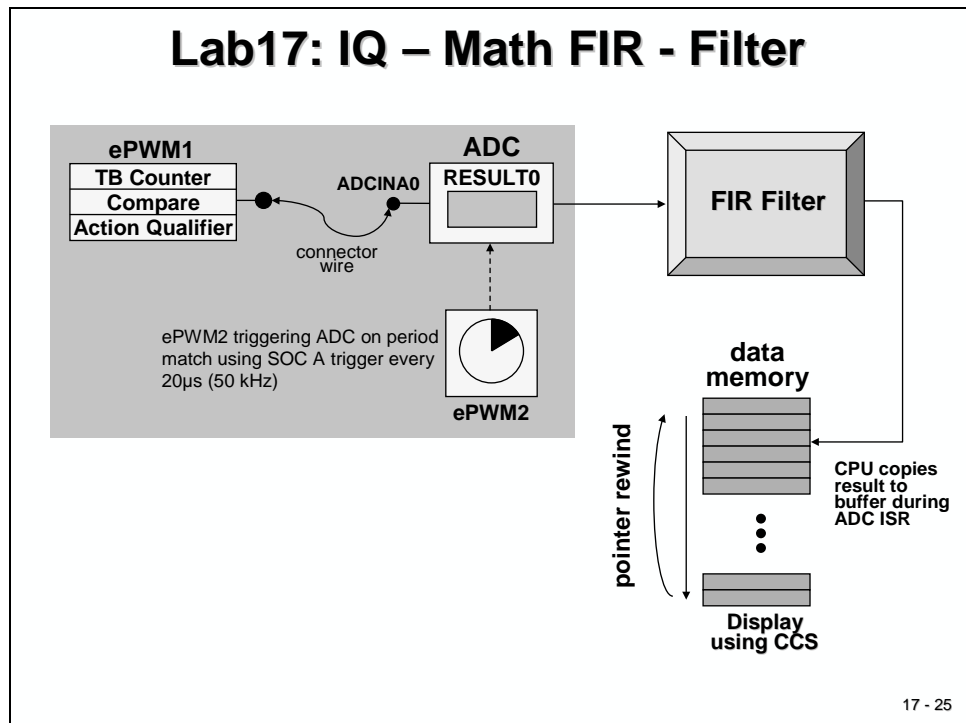
Lab 17: IQ - Math based low - pass filter

Before we start this lab exercise, you should inspect your solution from Lab 7-9 “Sine Wave PWM signal at ePWM1A” (see Chapter 7). In this exercise we already used some IQ-Math functions. Based on the discussion in Chapter 17 you should be able to better understand the details of the calculations, performed in Lab 7-9.

Objective

The objective of this laboratory exercise is to practice using the F2833x and its IQ-Math library.

The hardware diagram of this exercise is shown below (Slide 17-25). We will write code to calculate an FIR-Filter with low-pass characteristics, send our samples through the filter and compare the results in graphical form.



The procedure steps of the exercise are listed in Slide 17-26:

Lab17: IQ – Math FIR - Filter

Objective:

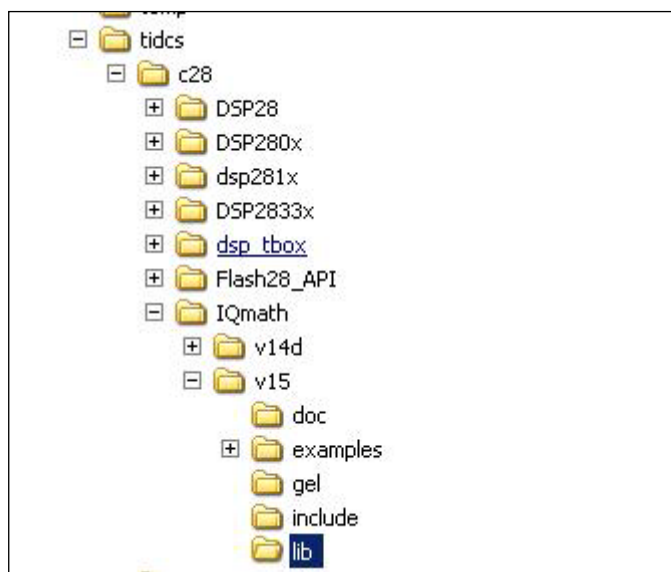
1. Generate a symmetrical 2 kHz PWM signal with 25% duty cycle
2. Measure the signal with ADC at a 50 kHz sample rate
3. Store the ADC results in a circular result buffer
4. Compute the ADC results in a digital finite impulse response (FIR) filter with low-pass characteristics. The Filter is calculated by IQ-Math functions
5. Plot the graphs of the filtered and unfiltered signal with Code Composer Studios graph tool
6. Compare the results

17 - 26

Procedure

Install IQMath

If not already installed on your PC, you will have to install the IQMath library now. The default installation path is "C:\tidcs\c28\IQmath":



If this library is not available on your PC, you will have to install it first. If you are in a classroom and you do not have administrator installation rights, ask your teacher for assistance. You can find the installation file under number "sprc087.zip" in the utility part of this CD-ROM or at the Texas Instruments Website (www.ti.com).

Open Project

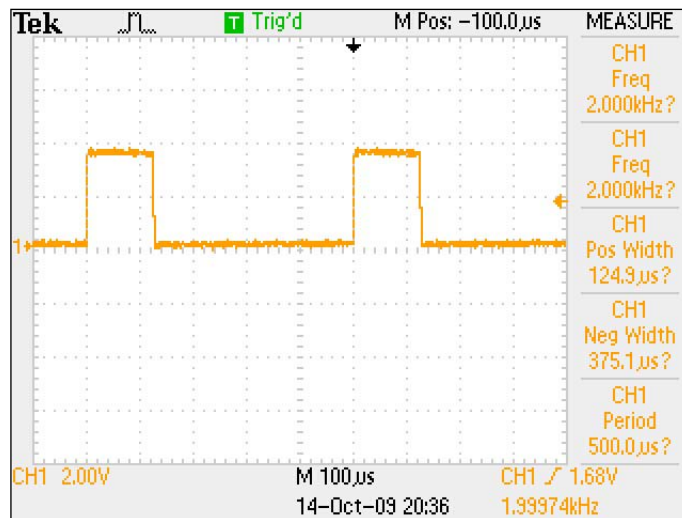
1. For convenience, open the provided project “Lab17.pjt” from C:\DSP2833x\Labs\Lab17.
2. Open the file “Lab17.c” to edit. In the function “Setup_ePWM1A()”, change the frequency of the square wave signal at ePWM1A from 1 kHz to 2 kHz and set the pulse width (duty cycle) to 25%.
3. In function “cpu_timer0_isr()”, delete the code to change the duty cycle (register CMPA). For Lab17 we will use a static or permanent pulse width of 25%.

Build, Load and Run

4. Click the “Rebuild All” button or perform:

Project	→ Build
File	→ Load Program
Debug	→ Reset CPU
Debug	→ Restart
Debug	→ Go main
Debug	→ Run(F5)

5. Use an oscilloscope to measure and verify the 2 kHz-output signal at ePWM1A. Connect your oscilloscope to the Peripheral Explorer Board Header J6-1.



Add code for ADC - Initialization

6. At the end of file “Lab17.c”, add the new function “void Setup_ADC(void)”. Also, define a function prototype at the beginning of “Lab17.c”. In the function “main()”, add a call to “Setup_ADC()”, immediately after the call to “Setup_ePWM1A()”.

In function “Setup_ADC()”, add the following initialization sequence:

- Call function “InitAdc()”. Because this function is defined in the file “DSP2833x_Adc.c”, we also have to add an external function prototype at the beginning of “Lab17.c”.
 - In register “ADCTRL1”:
 - Select “Cascaded Sequencer Mode”
 - Select “No continuous Run”
 - Set prescaler “CPS” to zero
 - In register “ADCTRL2”:
 - disable the ePWM_SOCA_SEQ1 start-option (we will use software start)
 - disable the ePWM_SOCB_SEQ start-option
 - disable the ePWM_SOCB_SEQ2 start option
 - Enable SEQ1 interrupts with every end of sequence (EOS)
 - Set the ADC-clock to HSPCLK / 6. Note: HSPCLK has been initialized in function “InitSysCtrl()” to SYSCLK/2. For a 150 MHz device the resulting ADC clock will be 12.5 MHz.
 - In register “ADCMAXCONV”:
 - Set the number of conversions to “1 conversion per start”
 - In register “ADCCHSELSEQ1”:
 - set field “CONV00” to convert channel ADCINA2
7. In Step 6 we have enabled the ADC to request an interrupt at the end of a conversion. To get this interrupt into the CPU, we also must enable the PIE - unit switch for the ADC and we must provide an interrupt-service routine.

- In function “main()”, locate the line, in which we enabled bit 7 of register PIEIER1. Add a second line to enable also bit 6 of register PIEIER1, which controls the ADC interrupt line:

PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

- Locate the line, in which we loaded variable “PieVectTable” with the address of function “cpu_timer0_isr()”. Now add a second line to load the address of an interrupt service routine (e.g. “adc_isr()”) for the ADC:

PieVectTable.ADCINT = &adc_isr;

- At the beginning of file “Lab17.c”, add a function prototype for the new function:

interrupt void adc_isr(void);

- At the end of file “Lab17.c”, add the new interrupt service function:

```
interrupt void adc_isr(void)  
{  
}
```

We will fill in the code lines in “adc_isr()” in procedure Step 9.

8. To start the ADC we will use our time-base, the CPU-Timer 0. In file “Lab17.c” this timer is still initialized to 100 microseconds from an earlier lab. For the new exercise, we would like to use a sample frequency of 50 kHz or a period of 20 microseconds. Change the line to initialize the CPU-timer 0 to:

```
ConfigCpuTimer(&CpuTimer0,150,20);
```

In the function “cpu_timer0_isr()”, which is triggered by Timer 0 once every 20 microseconds, add a line to start the ADC by software:

```
AdcRegs.ADCTRL2.bit.SOC_SEQ1 = 1;
```

9. In the function “adc_isr()”, which will be triggered at the end of each conversion, add the following 3 lines to re-initialize the ADC for the next conversion:

```
AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;    // Reset SEQ1  
AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;  // Clear INT SEQ1 bit  
PieCtrlRegs.PIEACK.all = 1;         // Acknowledge interrupt to PIE
```

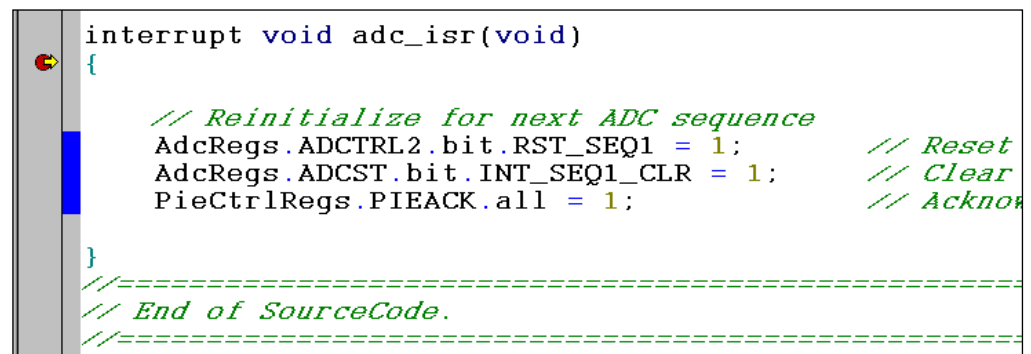
Now we should have completed the framework, which consists of a 2 kHz - signal at ePWM1A, a sampling time base of 50 kHz, generated by CPU-Timer 0 and an ADC, which is triggered by CPU-Timer0 to sample channel ADCINA2. The end of conversion will trigger the interrupt service routine “adc_isr()”. Before we go on to read and store the ADC results in a buffer, let us perform a test, to verify that this framework actually works:

Build, Load and Run

10. Click the “Rebuild All” button or perform:

Project	→ Build
File	→ Load Program
Debug	→ Reset CPU
Debug	→ Restart
Debug	→ Go main

11. Set a breakpoint in the function “adc_isr()” and run the program (F5). If everything works as expected, the breakpoint should be hit:



```
interrupt void adc_isr(void)
{
    // Reinitialize for next ADC sequence
    AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;    // Reset
    AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;  // Clear
    PieCtrlRegs.PIEACK.all = 1;         // Acknow
}
//=====
// End of SourceCode.
//=====
```

Resume a few times the run of the code (F5). The breakpoint should be hit periodically. This proves that our framework is functional.

Remove the breakpoint and run the code. If your oscilloscope is still connected to ePWM1A, it should still show the 2 kHz - signal from procedure step 5. Finally, halt the code (Shift + F5).

Add a sampling buffer

12. Because our computation will be done based on IQ-Math, we need to include the IQ - Math function prototypes. At the beginning of file “Lab17.c”, add a line to include the IQ-Math header file:

```
#include "IQmathLib.h"
```

13. Also at the beginning of file “Lab17.c”, add a macro to define the size of our data buffer, a macro to define the value of 3.0 in default IQ-format, and the data buffer itself as a global variable:

```
#define AdcBufLen    50
#define AdcFsVoltage _IQ(3.0) // ADC full scale voltage
_iq AdcBuf[AdcBufLen];      // ADC results buffer
```

14. In interrupt service function “adc_isr()”:

- Add a static unsigned integer variable “index” and initialize it with zero.
- Read the current sample and store it in array “AdcBuf”:

```
AdcBuf[index] = _IQmpy(AdcFsVoltage, _IQ16toIQ( (_iq)AdcRegs.ADCRESULT0));
```

This line needs an explanation (from right to left). First we read the latest sample from the ADC. The result register format is 16 bit, but the result data are in bits 15 to 4 (left justified). We “interpret” these numbers as an unsigned value between +1 and 0; the term is “binary fractions” or “per-unit”. Next, we convert this I16Q16 - number into the default IQ - format (function “_IQ16toIQ()”). Finally this percentage number is multiplied by the full scale value of 3.0.

- Increment variable “index” and reset it to 0, if it exceeds AdcBufLen.

The whole ISR should now look like this:

```
interrupt void adc_isr(void)
{
    static Uint16 index=0; // index into ADC buffers
    AdcBuf[index] = _IQmpy(AdcFsVoltage, _IQ16toIQ( (_iq)AdcRegs.ADCRESULT0));
    index++; // Increment the index
    if(index == AdcBufLen) index = 0; // Rewind the pointer to begin
    // Reinitialize for next ADC sequence
    AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1; // Reset SEQ1
    AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1; // Clear INT_SEQ1 bit
    PieCtrlRegs.PIEACK.all = 1; // Acknowledge interrupt to PIE
}
```

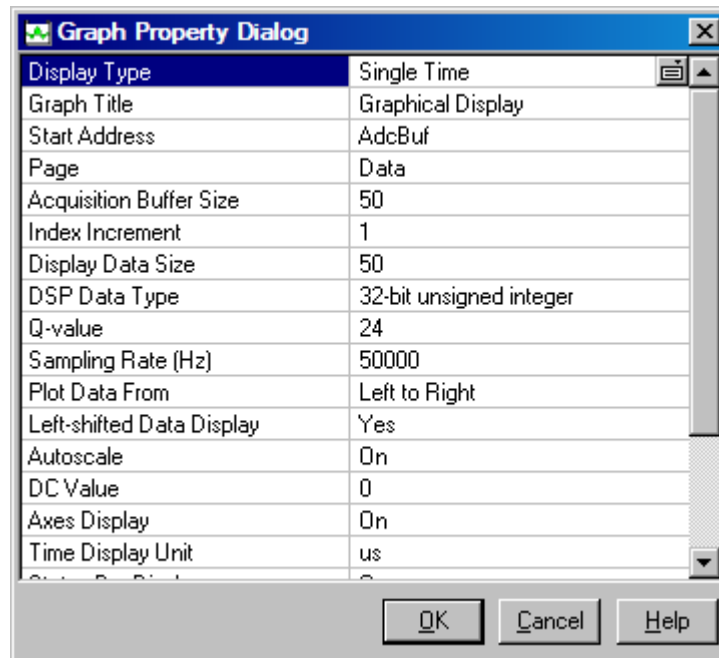
Now it is time to perform a test, whether our sampling system is able to fill the result buffer. Using a wire, connect the Peripheral Explorer Board Header J6-1 (ePWM1A) to Header J13-4 (ADCINA2).

Build, Load and Run

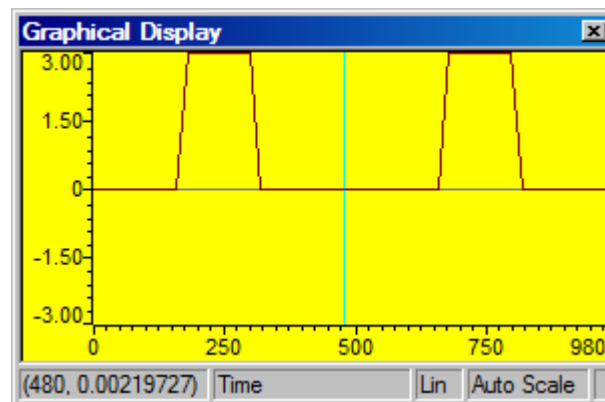
15. Click the “Rebuild All” button or perform:

Project	→	Build
File	→	Load Program
Debug	→	Reset CPU
Debug	→	Restart
Debug	→	Go main
Debug	→	Run

16. Open a Graph - Window (View → Graph → Time/Frequency) and enter the following properties:



The graph should display the sampled data, e.g. the waveform of the 2 kHz signal with 25% pulse width:



Add the low - pass filter code

17. At this point it is time to add the low pass filter code. For convenience, the source code file “**Filter.c**” has been provided. Add this file to the project.
18. Open and inspect file “Filter.c”. This file contains an IQ-Math N-tap single-sample FIR filter function (“IQssfir()”). It calculates:

$$y(k) = \sum_{n=0}^{N-1} a(n) * x(k - n)$$

The code of function “IQssfir()” is based on some basic IQ-Math functions. It is just a simple filter example, and completely un-optimized. The goal with the code was clarity and simplicity, not efficiency. The filtering is done from last tap to first tap. This allows more efficient delay chain updating. The array ‘x’ contains the latest N samples, which are used in the next calculation of y(k). The array ‘a’ contains the filter coefficients and defines the transfer function of the filter.

19. At the beginning of “Lab17.c”, add a function prototype:

```
extern _iq IQssfir(_iq*, _iq*, Uint16);
```

20. Also at the beginning of “Lab17.c”, add 3 new global variables:

```
_iq AdcBufFiltered[AdcBufLen];      // filtered ADC results buffer  
_iq xBuffer[5] = {0,0,0,0,0};      // filter sample buffer  
_iq coeffs[5] = {_IQ(0.0357), _IQ(0.2411), _IQ(0.4465), _IQ(0.2411),  
  _IQ(0.0357)};
```

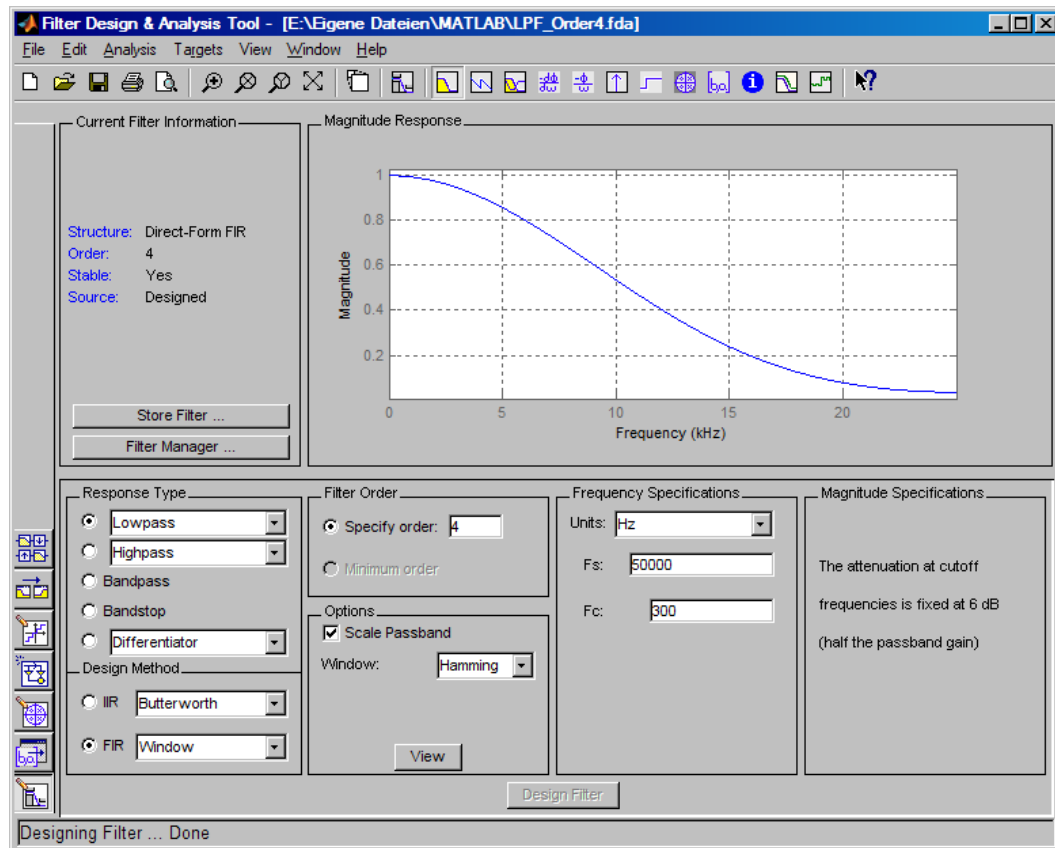
All variables are of type “_iq”, which is a signed 32-bit number with default IQ-format. To inspect or change this default IQ-format, open the file “IQmathLib.h” and search for the definition of constant “GLOBAL_Q”:

```
-----  
// Select global Q value and scaling. The Q value is limited to the  
// following range for all functions:  
//  
30 <= GLOBAL_Q <= 1  
//  
#ifndef GLOBAL_Q  
#define GLOBAL_Q 24  
#endif
```

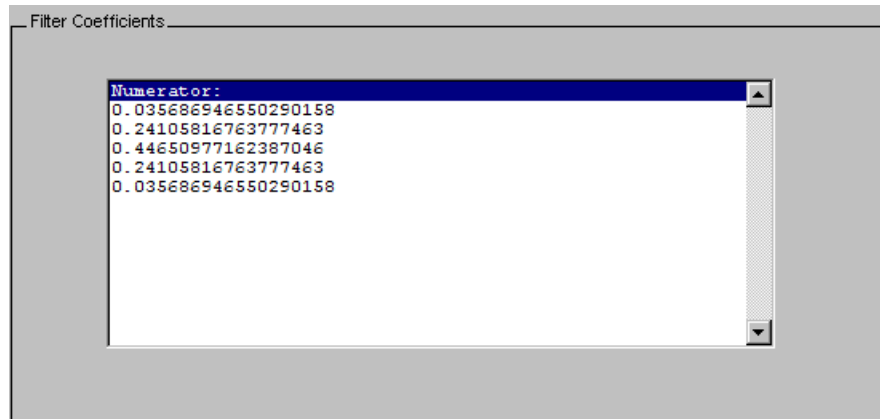
MATLAB Filter Coefficient Calculation

The filter coefficient values, shown above as initialization values for variable “coeffs”, are derived from a MATLAB - filter design tool called “fdatool”. The Filter parameters are:

- FIR - Low pass - Filter 4th order
- Sampling Frequency: 50 kHz
- Corner Frequency: 300 Hz
- Window: Hamming



The resulting filter coefficients are shown next:



21. In the function “adc_isr()”, immediately after the store instruction for the latest ADC-result, add code to update the filter sample buffer and call the filter function “IQssfir()”:

```
xBuffer[0] = AdcBuf[index];
```

```
AdcBufFiltered[index] = IQssfir(xBuffer, coeffs, 5);
```

The whole interrupt service routine “adc_isr()” should now look like:

```
interrupt void adc_isr(void)
{
    static Uint16 index=0;           // index into ADC buffers
    AdcBuf[index] = _IQmpy(AdcFsVoltage, _IQ16toIQ((_iq)AdcRegs.ADCRESULT0));

    *** Call the filter function ***
    xBuffer[0] = AdcBuf[index];      // Add the new entry to the delay
    AdcBufFiltered[index] = IQssfir(xBuffer, coeffs, 5);

    index++;                         // Increment the index
    if(index == AdcBufLen) index = 0; // Rewind the pointer to begin
    // Reinitialize for next ADC sequence
    AdcRegs.ADCCTRL2.bit.RST_SEQ1 = 1; // Reset SEQ1
    AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1; // Clear INT SEQ1 bit
    PieCtrlRegs.PIEACK.all = 1;      // Acknowledge interrupt to PIE
}
```

Also, at the beginning of “Lab17.c”, add an external prototype for the function “IQssfir()”:

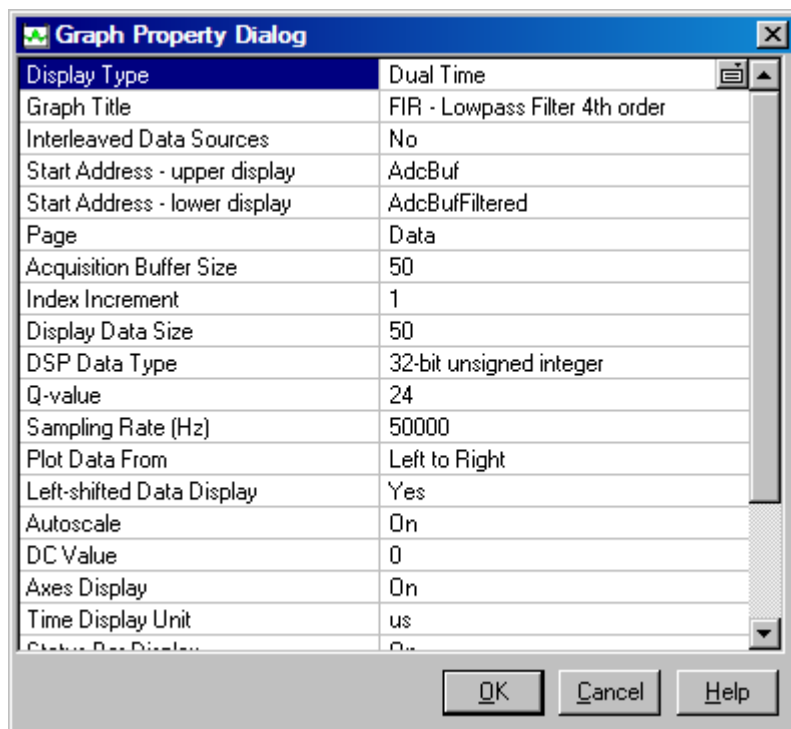
```
extern _iq IQssfir(_iq*, _iq*, Uint16);
```

Final Build, Load and Run

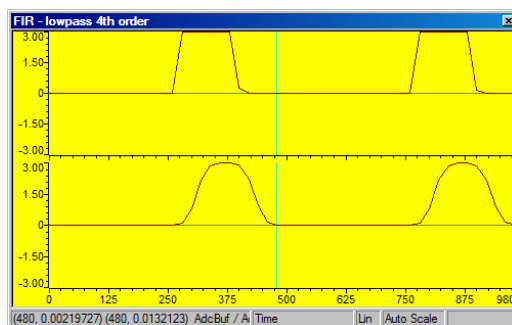
22. Click the “Rebuild All” button or perform:

Project	→	Build
File	→	Load Program
Debug	→	Reset CPU
Debug	→	Restart
Debug	→	Go main
Debug	→	Run

23. Open a Graph - Window (View → Graph → Time/Frequency) and enter the following properties:



The top graph should display the sampled 2 kHz - signal and the bottom graph shows the output of the low-pass filter, stored in buffer “AdcBufFiltered”.



Benchmark IQ-Math and Floating-Point Filter code

Since the F2833x is able to run floating-point code and fixed-point code, we can test the filter code of this exercise, based on floating-point machine code or fixed-point code. The fixed-point code will be done in IQ-Math.

24. Open the file “IQmathLib.h” and verify that the macro “MATH_TYPE” is set to “IQ_MATH”:

```
#ifndef __IQMATHLIB_H_INCLUDED__
#define __IQMATHLIB_H_INCLUDED__

#define    FLOAT_MATH    1
#define    IQ_MATH       0

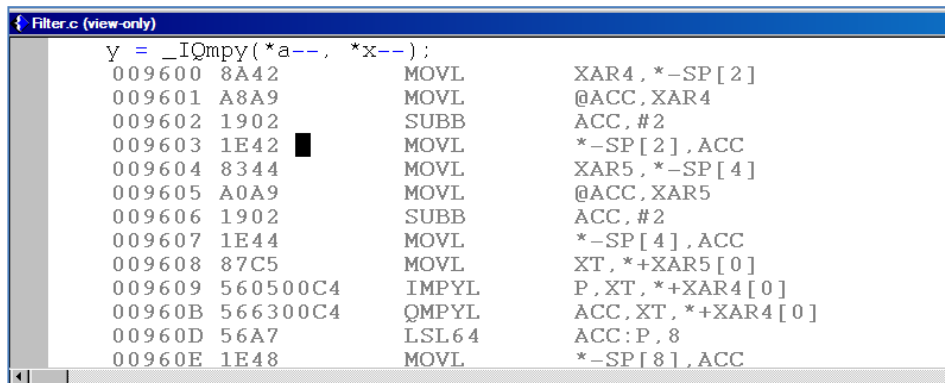
#ifndef    MATH_TYPE
#define    MATH_TYPE     IQ_MATH
#endif
```

This macro will tell the compiler to use IQ-Math instructions for the compilation of function “IQssfir()”.

25. Rebuild, reload and run the project.

Open the graph window shown in procedure Step 23 and verify that it still shows the same two graphs.

Open the file “Filter.c”, right mouse click into the file and select “Mixed Mode”:



```
Filter.c (view-only)
y = _IQmpy(*a--, *x--);
009600 8A42      MOVL      XAR4, *-SP[2]
009601 A8A9      MOVL      @ACC, XAR4
009602 1902      SUBB      ACC, #2
009603 1E42      MOVL      *-SP[2], ACC
009604 8344      MOVL      XAR5, *-SP[4]
009605 A0A9      MOVL      @ACC, XAR5
009606 1902      SUBB      ACC, #2
009607 1E44      MOVL      *-SP[4], ACC
009608 87C5      MOVL      XT, *+XAR5[0]
009609 560500C4   IMPYL     P, XT, *+XAR4[0]
00960B 566300C4   QMPYL     ACC, XT, *+XAR4[0]
00960D 56A7      LSL64     ACC:P, 8
00960E 1E48      MOVL      *-SP[8], ACC
```

Verify that the compiler has actually used IQ - machine code instructions. Look for the instructions “IMPYL”, “QMPYL” and “LSL64”. This code will run on any C2000 family member without floating-point hardware, e.g. F280x, F281x or F2823x devices.

26. Now change the macro “MATH_TYPE” in file “IQmathLib.h” from “IQ_MATH” to “FLOAT_MATH”.

```
#define    FLOAT_MATH    1
#define    IQ_MATH      0

#ifdef    MATH_TYPE
#define    MATH_TYPE    FLOAT_MATH
#endif
```

27. Rebuild, reload and run the project.

Open the graph window shown in procedure Step 23 and verify that it still shows the same two graphs.

Open the file “Filter.c”, right mouse click into the file and select “Mixed Mode”:

```
y = _IQmpy(*a--, *x--);
0095B4 8344      MOVL      XAR5, *-SP[4]
0095B5 A0A9      MOVL      @ACC, XAR5
0095B6 1902      SUBB      ACC, #2
0095B7 1E44      MOVL      *-SP[4], ACC
0095B8 8A42      MOVL      XAR4, *-SP[2]
0095B9 E2AF00C5  MOV32     R0H, *+XAR5[0], UNCF
0095BB A8A9      MOVL      @ACC, XAR4
0095BC 1902      SUBB      ACC, #2
0095BD 1E42      MOVL      *-SP[2], ACC
0095BE E2AF01C4  MOV32     R1H, *+XAR4[0], UNCF
0095C0 E7000008  MPYF32     R0H, R1H, R0H
0095C2 7700      NOP
0095C3 E2030048  MOV32     *-SP[8], R0H
```

Now the compiler has generated floating-point machine code! Instructions “MOV32” and “MPYF32” are using the floating-point hardware unit of the F28335.

Summary

Code written in IQ-Math can be compiled for a fixed-point target (MATH_TYPE = IQ_MATH) or for a floating-point target (MATH_TYPE = FLOAT_MATH). All we have to do is to change one single line in the header-file!

This is an exciting feature of the C2000 - family, because we don't have to modify a single line of code, when we move from a floating-point device to a fixed-point device or vice versa.

blank page