C28x FIR - Filter

Introduction

This chapter looks into one of the most common applications for Digital Signal Processors: "Digital Filters". As we have seen before, there are two basic classes of computing: "off – line" and "real – time". This is also valid for digital signal processing. Because we are dealing with the C28x controller, which is designed for embedded control, calculations are usually done in a real time environment. For a digital filter, this means that all internal processing of the current state of a system must be finished <u>before</u> a next input value is sampled. Again, computing time is most precious! The faster we calculate the algorithm of a digital filter, the more samples we can take from an input channel. It also means that the frequency of the input signal that is to be processed depends directly on the efficiency of the controller.

We will start with some mathematical basics of Digital Filters, but we will not go too much into the theoretical background. To learn more about the mathematics behind Digital Filters, you will have to join other courses at your university. The Texas Instruments C6000- Teaching CD-ROM is highly recommended to learn more about the design of digital filters. Beginning with chapter 14 of this CD, you will be introduced to techniques for filter coefficient estimation and to basic approaches of windowing. Although this C6000 CD is based on the C6000 family the chapters about Filters and Fourier Transform are also valid for the C28x.

Next we will look into the structure of "Finite Impulse Response (FIR)"-filters and their properties. Because of their simplicity and stability, these types of filters are often used in digital signal processing. We will calculate some examples for low-pass (LPF) and high-pass (HPF) – FIR-filters before we look into a C implementation of such a filter algorithm for the C28x. The use of IQ-Math C-functions leads to a much faster execution then using standard ANSI-C implementation. As we have already seen, IQ-Math is a unique feature of the C28x, which is based on its internal hardware units.

If the computing speed of the IQ-Math implementation is still not fast enough, we can do better! By switching to Assembly Language coding and by using Texas Instruments Digital Filter Library (sprc082.zip), we can bring the C28x to its top speed. By means of two examples, we will have a look into the implementation of a FIR in Assembly Language and into the usage of the C-callable library function "FIR16", provided by TI.

Finally we will perform a laboratory experiment using a FIR-Filter. After we generate a 2 kHz square wave signal, we will sample it back into the C28x by means of the internal ADC. We will use the samples to calculate a low-pass function with a 4th order FIR-Filter in real-time. Code Composer Studio's Graph Tool will be used to visualize the behaviour of both the unfiltered and filtered signal in parallel.

Module Topics

Basics of Digital Filter Theory

Time Domain Equation

The following equation for a Linear Time-Invariant (LTI) system is the starting point to derive a representation of a Digital Filter:

Basics of Digital Filter Theory

- ◆ **Digital Filter Algorithms are probably the most used numerical operations of a Digital Signal Processor**
- ◆ **Digital Filters are based on the common difference equation for Linear Time-Invariant (LTI) – systems:**

$$\sum_{m=0}^{N-1} a_m \cdot y[n-m] = \sum_{k=0}^{N-1} b_k \cdot x[n-k]$$

- • **y(n) = output signal**
- • **x(n) = input signal**
- • **a_m, b_k = coefficients**
- • **N = number of coefficients (order of system)**
- ◆ **Normalized to $a_0 = 1$ we derive the basic equation in time domain:**

$$y(n) = \sum_{k=0}^{N-1} b_k \cdot x[n-k] - \sum_{m=1}^{N-1} a_m \cdot y[n-m]$$

14 - 2

It is an equation for discrete input (x (n)) and output (y (n)) signals that are processed by a transforming system. The properties of the transformation are expressed by coefficients (a_m and b_k). Terms like x[n-k] or y[n-m] are used to express the status of the input and output signal k or m times before the current sample time n. For causal systems, all samples before time t = 0 are zero.

These types of equations represent the modification of the output signal y(n) on a time base – we call it "Time Domain Representation". The smallest amount of time that is used in these equations is the sampling period. After the input signal is sampled, the continuous time scale is replaced by a sequence of numbers. According to Shannon's sampling theorem, the sampling frequency must be at least twice as fast as the highest frequency component of the input signal.

The calculation of the normalized equation for y(n) can be visualized graphically, as shown with the next slide:

Time Domain Chart of a Digital Filter
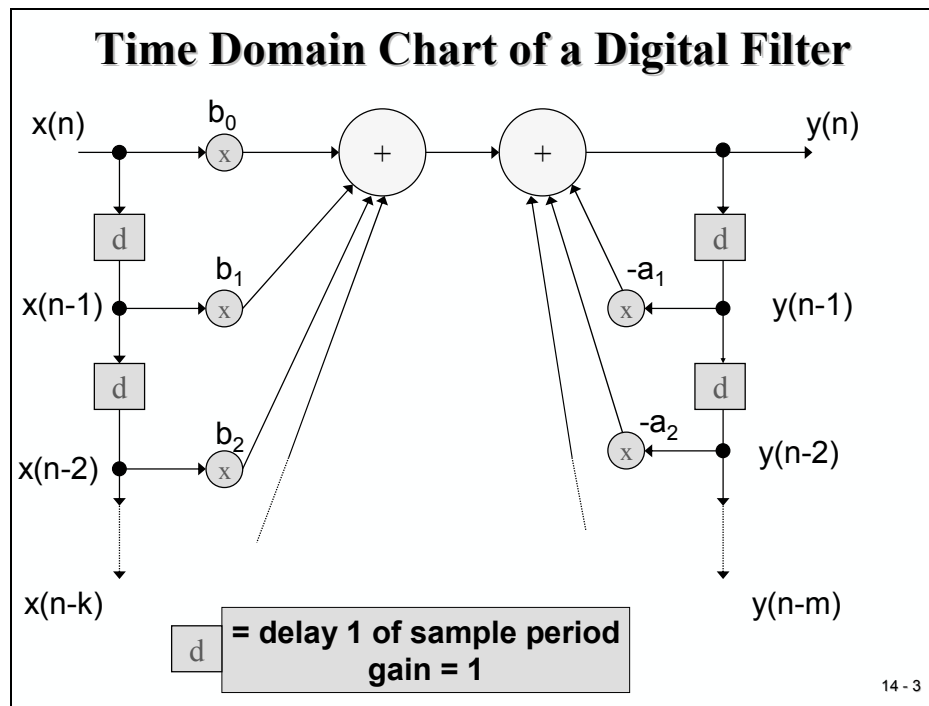
This flow can be used to calculate y(n) from the current input sample x(n) and samples of the input signal, taken one (x(n-1)), two (x(n-2)) or k (x(n-k)) samples before. We call this part the "forward" section of the calculation. If we include the status of the output signal delayed by one period (y(n-1)), two periods (y(n-2)) up to k periods (y(n-k)) into the calculation of the new value of y(n), we add a "feedback" section to the computing scheme.

To translate this flowchart into a computer program, we would have to store not only the current input x(n) and output y(n), but also information about their previous states. How do we code this in a programming environment? Usually, with two arrays that are big enough to store all the previous states of x and y. This type of array is usually called a "buffer". It functions as delay-line, hence the term "Delay-Line-Buffer".

So what happens when the code has calculated a new value of y(n)? Obviously, y(n) must be presented to the outside world as a new result of our calculation. Fine, but what is next? To perform a calculation in real-time, the code must read the next sample from input signal x and store it at buffer position x(n). But x(n) is still occupied by the sample from one period earlier! Before we can store the new sample in x(n), the code must move all entries in array x to the next position, x(n) to x(n-1), x(n-1) to x(n-2) and so on. During this procedure the oldest sample will be discarded. At the output side, the code has to shift all y – values in a similar manner.

Consider the sequence of shift operations! In practice, we have to shift the second oldest first, followed by the next oldest. If not, we fill the entire buffer with x(n)!

Frequency Domain Equation

The second interpretation of the behavior of a LTI-system is done in terms of frequency – the "Frequency Domain" - Equation.

The basic operation to transfer a time discrete signal in frequency domain is called "Z-Transformation" (ZT). The transformation follows these rules:

Transfer Function of a Digital Filter

◆ **The Z-Transform of the original input signal x(n) is defined as:**

$$ZT\{x(n)\} = \underline{X}(z) = \sum_{n=0}^{\infty} x(n) \cdot z^{-n}$$

◆ **with** $z = e^{pT} \text{ and } p = \sigma + j\omega$ p = complex angular frequency

◆ **One property of the Z-Transform is that the ZT of a time-shifted signal is equal to the ZT of the original signal except of a factor z^{-k}:**

$$ZT\{x(n-k)\} = z^{-k} \cdot \underline{X}(z)$$

14 - 4

The slide shows how the series of discrete input samples x (n) is converted into a complex series $\underline{X}$(z). Instead of representing the signal as sequence "number over time" we can represent the signal as sequence "complex number over frequency".

One important property of the ZT is that the ZT of a time shifted input signal x (n-k) is identical to the ZT of the non-time shifted signal x(n), except for a multiplier z^{-k}. This feature reduces the workload to calculate the complex series for $\underline{X}$(z) dramatically.

How do we use this ZT to convert the time-domain equation for an LTI-system into its frequency representation? Well, we have to apply the ZT to both sides of the time-domain equation, shown at the next slide:

Transfer Function of a Digital Filter

◆ **Z-Transform is applied to both sides of the time domain equation of a Digital Filter :**

$$ZT\left\{y(n)+\sum_{m=1}^{N-1}a_m\cdot y[n-m]\right\}=ZT\left\{\sum_{k=0}^{N-1}b_k\cdot x[n-k]\right\}$$

$$\underline{Y}(z)+\sum_{m=1}^{N-1}a_m z^{-m}\underline{Y}(z)=\sum_{k=0}^{N-1}b_k z^{-k}\underline{X}(z)$$

$$\underline{Y}(z)\left[1+\sum_{m=1}^{N-1}a_m z^{-m}\right]=\underline{X}(z)\left[\sum_{k=0}^{N-1}b_k z^{-k}\right]$$

14 - 5

The final equation is called "Transfer Function" of the Digital Filter. It is a frequency domain representation of the influence that is exerted to an input signal by the Digital Filter.

Transfer Function of a Digital Filter

◆ **Finally we derive the Transfer Function of a Digital Filter of order N in frequency domain:**

$$\underline{H}(z)=\frac{\underline{Y}(z)}{\underline{X}(z)}=\frac{\displaystyle\sum_{k=0}^{N-1}b_k z^{-k}}{1+\displaystyle\sum_{m=1}^{N-1}a_m z^{-m}}$$

14 - 6

For a given spectrum of input frequencies $\underline{X}(z)$ the transfer function defines the shape of the output spectrum $\underline{Y}(z)$. Complex frequency numbers are represented as magnitude and phase per frequency line.

In a similar way as we have seen for the time-domain flow, we can draft a calculation scheme for the transfer function in the frequency domain. Each delay-unit in the time domain is replaced by a multiplication by complex number z^{-1}. The basic principle to calculate a new $\underline{Y}(z)$ is similar to the time-domain flow, except the complex multiplication. The algorithm still needs two arrays to store $\underline{X}(z)$ and $\underline{Y}(z)$, except that they have to now use complex numbers with a real and imaginary part for each number. The Frequency Domain Calculation of the frequency response of a system is normally used to analyse an incoming signal for its frequency components.

Frequency Domain Flow of a Digital Filter

Multiply by z^{-1} in Frequency Domain =
Unit delay by one sample period in Time Domain

14 - 7
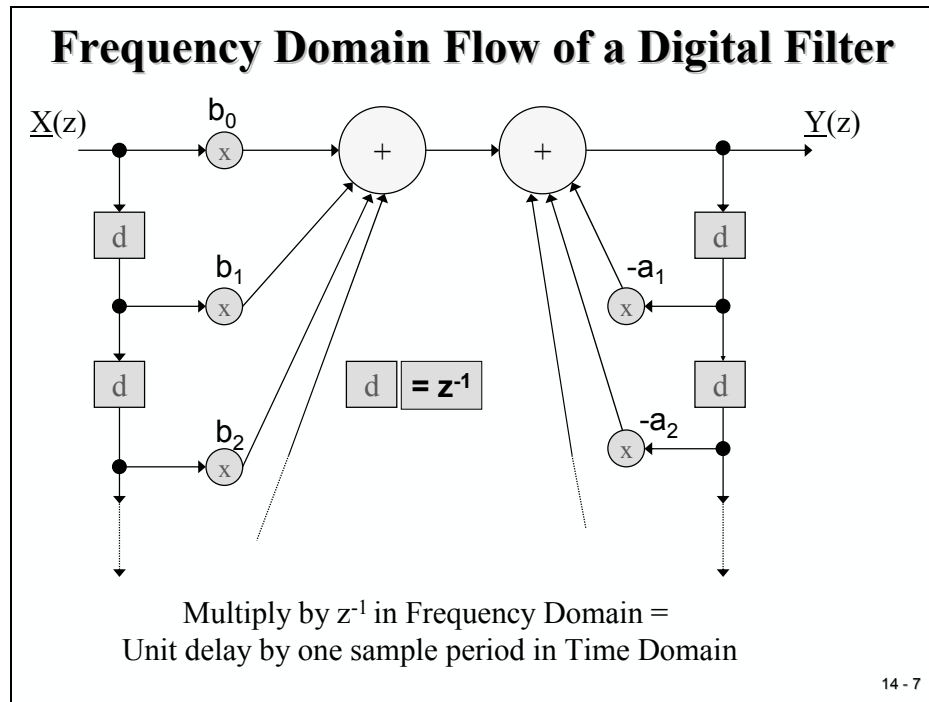
Finite Impulse Response Filter

If a Digital Filter does not have any feedback components (all $a_m = 0$), we call this system a "Finite Impulse Response" (FIR). It can be shown that the response of such a system to a single input impulse will eventually vanish.

Finite Impulse Response (FIR) - Filter

◆ **If all feedback coefficients a_m are equal to zero we derive the equation system for a "Finite Impulse Response (FIR)" – Filter:**

$$\underline{H}(z) = \frac{\underline{Y}(z)}{\underline{X}(z)} = \sum_{k=0}^{N-1} b_k z^{-k}$$ Frequency Domain

◆ **and:**

$$y(n) = \sum_{k=0}^{N-1} b_k x[n-k]$$ Time Domain

14 - 8

If feedback components exist, the system is called an "Infinite Response Filter" (IIR).

Infinite Impulse Response (IIR) - Filter

◆ **If coefficients a_m are present we call this type of filter "Infinite Impulse Response (IIR). In this case the equation with feedback part must be used for the filter calculation.**

◆ **Obviously the "feedback" terms $a_m*y(n-m)$ deliver some amount of energy back into the calculation.**

◆ **Under particular circumstances this feedback system will respond to an input impulse infinite – hence the name.**

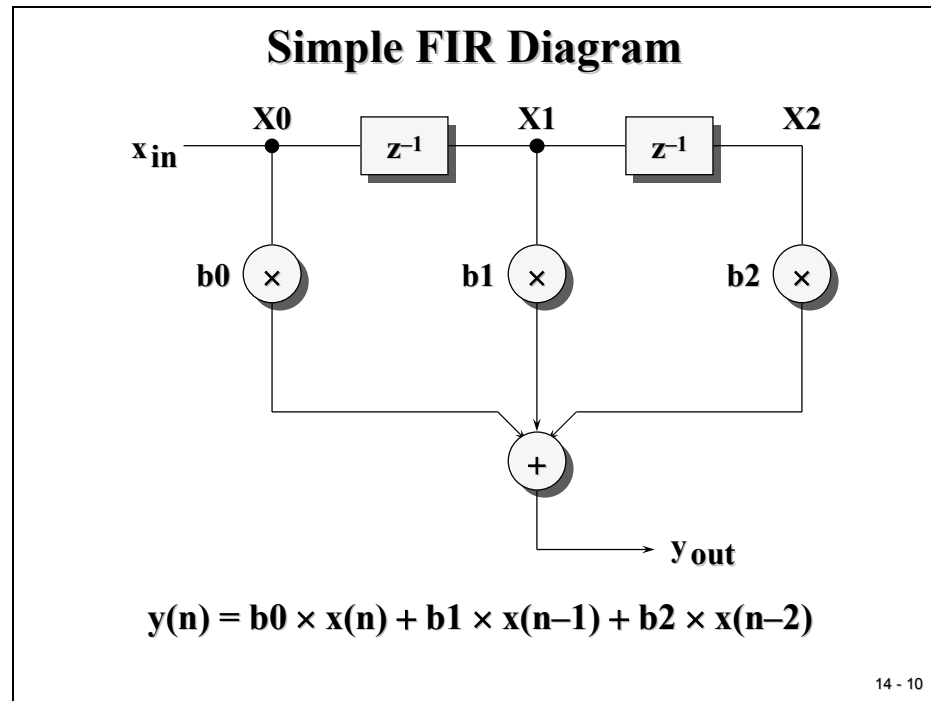$$\underline{H}(z) = \frac{\underline{Y}(z)}{\underline{X}(z)} = \frac{\sum_{k=0}^{N-1} b_k z^{-k}}{1 + \sum_{m=1}^{N-1} a_m z^{-m}}$$ **IIR – Filter**

$$y(n) = \sum_{k=0}^{N-1} b_k \cdot x[n-k] - \sum_{m=1}^{N-1} a_m \cdot y[n-m]$$

14 - 9

Properties of a FIR - Filter

<div style="border:1px solid">

Simple FIR Diagram

X0 **X1** **X2**

x_{in} ———•——— z^{-1} ———•——— z^{-1} ———•

b0 ⊗ **b1** ⊗ **b2** ⊗

⊕

→ y_{out}

$$y(n) = b0 \times x(n) + b1 \times x(n-1) + b2 \times x(n-2)$$

14 - 10

</div>

One typical property of the FIR-Transfer Function is its periodicity by 2π:

<div style="border:1px solid">

Properties of a FIR Filter

◆ **Replacing z by it's original definition:**

$$z = e^{pT} = e^{(\sigma + j\omega)T}$$

disregarding σ (loss – less filter) and normalizing to T=1:

$$\underline{H}(z)\big|_{z=e^{j\omega}} = \underline{H}(\omega) = \sum_{k=0}^{N-1} b_k e^{-jk\omega}$$

◆ **Since $e^{-j2\pi k} = 1$:**

$$\underline{H}(\omega + 2\pi) = \sum_{k=0}^{N-1} b_k e^{-jk(\omega+2\pi)} = \sum_{k=0}^{N-1} b_k e^{-jk\omega} e^{-j2\pi k} = \underline{H}(\omega)$$

◆ **FIR filters have a periodic frequency response of 2π !**
◆ **We need to limit the spectrum!**

14 - 11

</div>

FIR Examples

Let us calculate the frequency response of the following filter system. As the diagram shows it lacks feedback components – it is of FIR-type. It is a first-order filter, the filter coefficients are

- $b_0 = +0.5$

- $b_1 = +0.5$

What will the magnitude of the output signal look like? Which frequency components will pass the filter, which one will be damped?

<div style="border:1px solid black; padding:1em;">

FIR – Example 1

x(n) ○— Z^-1 —→ x(n-1)

0.5 —⊗   0.5 —⊗

b₀ = 0.5   b₁ = 0.5   (+) → y(n)

● **Frequency Response ?**
● **Type of Filter ?**

$$\underline{H}(z) = b_0 z^0 + b_1 z^{-1}$$

$$\underline{H}(z) = 0.5(1 + z^{-1})$$

$$\underline{H}(j\omega) = 0.5(1 + e^{\frac{-j2\pi f}{f_A}})$$

$$z = e^{pT}; \; p = \sigma + j\omega; \; \omega = 2\pi f; \; T = \frac{1}{f_A}$$

f_A = sampling frequency

$$\underline{H}(j\omega) = 0.5(1 + \cos(2\pi \frac{f}{f_A}) - j\sin(2\pi \frac{f}{f_A}))$$

$$\left|\underline{H}(j\omega)\right| = \sqrt{\text{Re}^2 + \text{Im}^2}$$
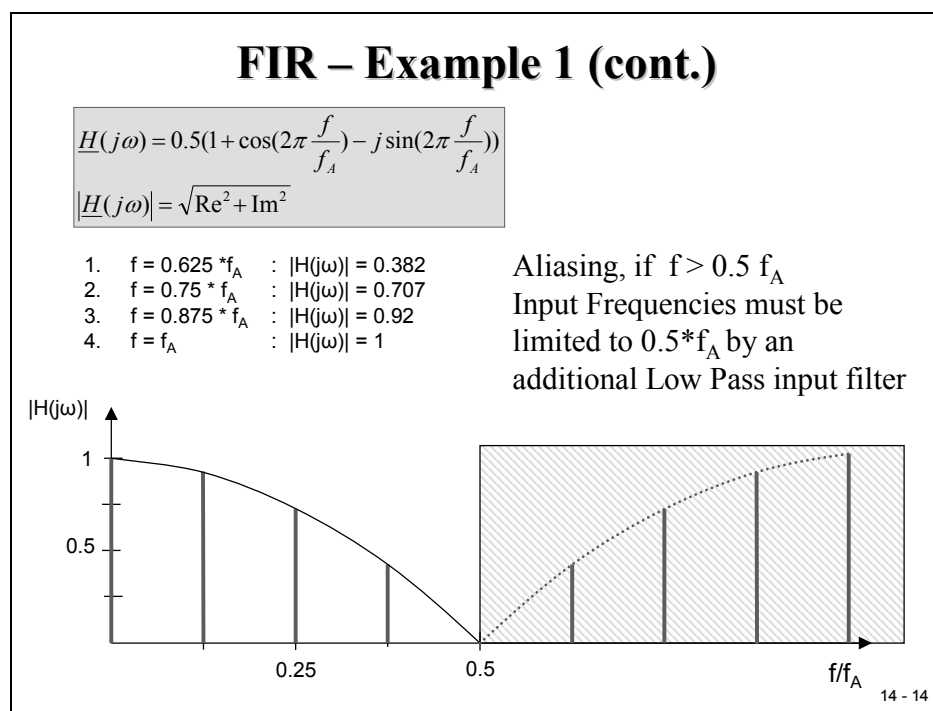
14 - 12

</div>

If we calculate the magnitude for frequencies f = 0, f = f_A/8, f= f_A/4, f =f_A*3/4 and f = f_A/2 we get:

$$\left|H(j\omega)\right| = 1, 0.92, 0.707, 0.382 \text{ and } 0$$

The graph is shown next. Low frequencies are amplified by 1, the more we approach 0.5*f_A, the more the magnitude is damped and finally reaches 0. This is a low-pass filter!

FIR – Example 1 (cont.)

$$\underline{H}(j\omega) = 0.5(1 + \cos(2\pi \frac{f}{f_A}) - j\sin(2\pi \frac{f}{f_A}))$$

$$\left| \underline{H}(j\omega) \right| = \sqrt{\text{Re}^2 + \text{Im}^2}$$

1. f = 0   : |H(jω)| = 1
2. f = 0.125 * f_A : |H(jω)| = 0.92
3. f = 0.25 * f_A : |H(jω)| = 0.707
4. f = 0.375 * f_A : |H(jω)| = 0.382
5. f = 0.5 * f_A : |H(j ω)| = 0

**Filter – Type: Low-pass**

14 - 13
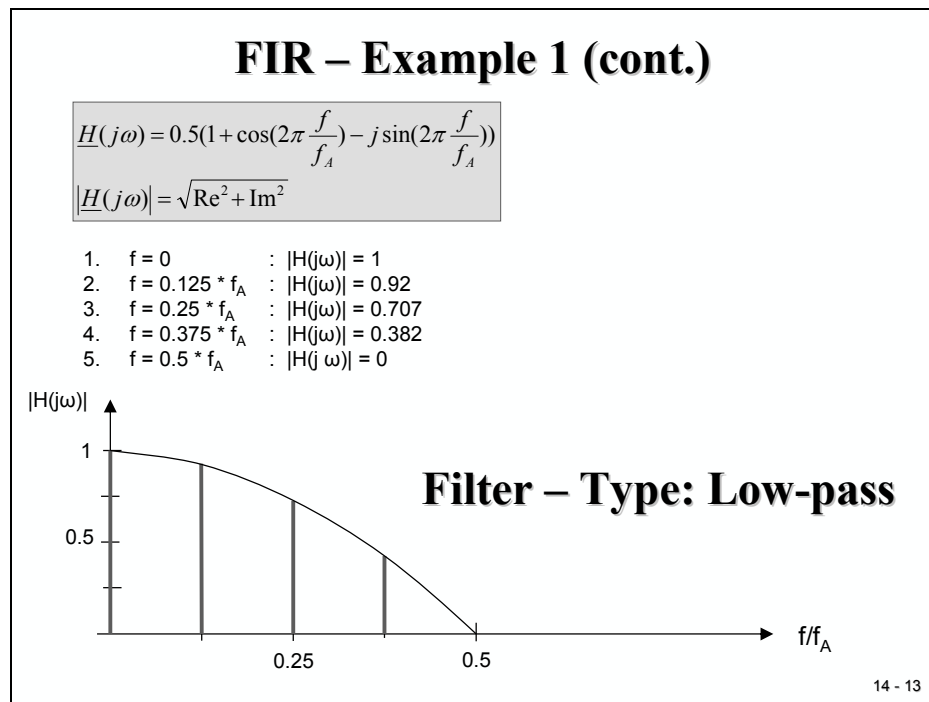
What happens, if the input frequency f goes beyond $f_A/2$, violating the SHANNON-theorem? The magnitude rises from 0 to 1, introducing false ("aliased") frequency components!

FIR – Example 1 (cont.)

$$\underline{H}(j\omega) = 0.5(1 + \cos(2\pi \frac{f}{f_A}) - j\sin(2\pi \frac{f}{f_A}))$$

$$\left| \underline{H}(j\omega) \right| = \sqrt{\text{Re}^2 + \text{Im}^2}$$

1. f = 0.625 *f_A : |H(jω)| = 0.382
2. f = 0.75 * f_A : |H(jω)| = 0.707
3. f = 0.875 * f_A : |H(jω)| = 0.92
4. f = f_A : |H(jω)| = 1

Aliasing, if f > 0.5 f_A
Input Frequencies must be
limited to 0.5*f_A by an
additional Low Pass input filter

14 - 14

The solution to suppress any frequencies that violate the sampling theorem is to introduce an anti-aliasing filter before the samples are taken. This way, no frequency component beyond $f_A/2$ will disturb the digital processing. The anti-alias filter is an analogue low-pass filter.
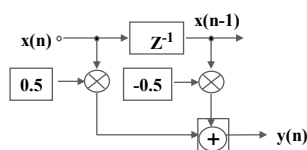
FIR – Example 1 (cont.)

◆ **Solution: Use an anti-aliasing filter at input to limit all input frequencies to $f_A/2$.**

$$x(t) \rightarrow \boxed{\text{Analogue Anti-Aliasing}} \rightarrow \boxed{\text{ADC}} \xrightarrow{x[n]} \boxed{\text{FIR}} \rightarrow y[n]$$
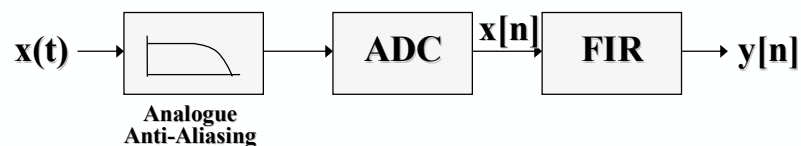
14 - 15

In the next example for an FIR-Filter of order 1 we only change coefficient b_1 from +0.5 into -0.5.
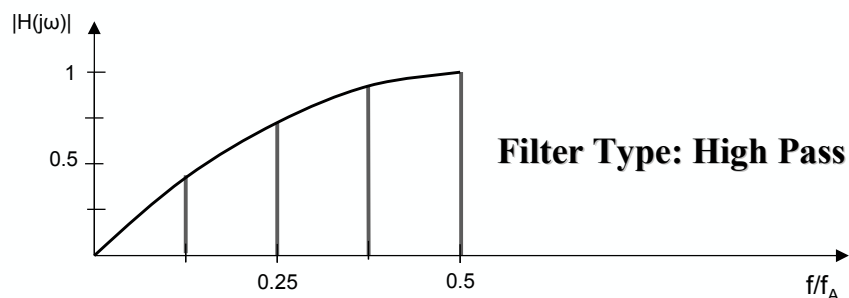
FIR – Example 2

$$b_0 = 0.5 \quad b_1 = -0.5$$

● **Frequency Response ?**
● **Type of Filter ?**

Note : We only changed b_1 from +0.5 to **-0.5** !

$|H(j\omega)|$

Filter Type: High Pass

14 - 16

What happens? The digital FIR-filter now damps frequencies around f=0 and amplifies an input frequency of $f = f_A/2$ by 1. The filter type has changed from low-pass into high-pass. By modifying coefficients we can change the behaviour of the filtering system. Compare this feature with an analogue filter, where you would have to change resistors or capacitors with a soldering iron.

If the modification of filter coefficients is done in real-time by the controller code itself, we call this an "Adaptive Filter".

We can also use digital filters to non-technical topics. The next example of a 2nd order FIR-Filter shows how the average stock price per week is calculated using the "moving average calculation":
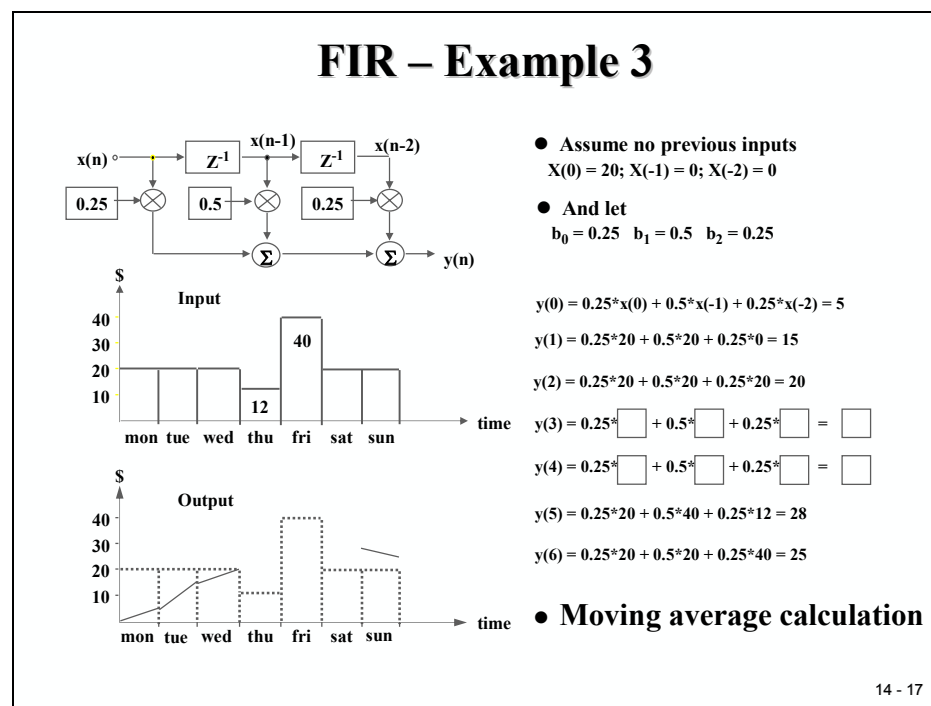

The input is the daily stock price; the output is the moving average. Due to the nature of causally determined systems, we assume that we do not have previous inputs when we start on Monday (time = zero). Of course a broker would know the prices from last week and would request us to add them to our calculation – technically speaking, the stock market is a non causal system, probably practically too.

Anyway, if our calculation advances to Wednesday everybody will be pleased with our results of a moving average.

Let's go back to more technical issues and leave the stock market brokers without further DSP-support.

FIR Implementation in C

Before we proceed to the implementation of a FIR filter using the C28x, let us recall one important step between the periodic calculations. The sample buffer must be prepared to include the latest sample value at the start of the buffer by shifting all elements by one position. Quite often this is done in ascending order:
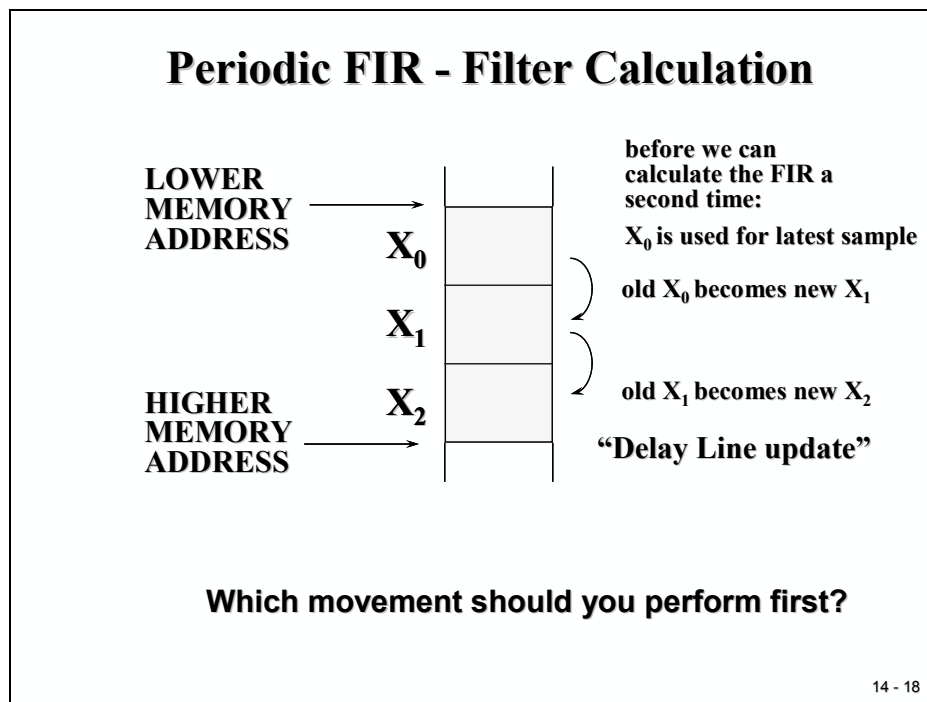
The following code is taken from a Texas Instruments example to implement FIR-Code for the C28x in IQ-Math-format. You have seen that this fixed-point math's is much better adapted to the C28x than any standard ANSI-C solution – in terms of computing power.

The name of the function "IQssFIR" relates to "IQ – single source FIR" – only one stream of input numbers is computed. Input parameters are two pointers to the array of input samples and to the coefficients and the number of taps – that's order minus 1.

The processing is based on the data type "_iq" which is defined in "IQmathLib.h". The return parameter is the new output value y(k), also in "_iq"-format.

When you inspect the code, you will notice that it operates from back to front, placing the two pointers to the end of the two buffers and post-decrementing them after any single multiplication. The shift operation on the delay line is done immediately after the current tap has been processed with the help of a temporary pointer "xold".

The accumulation is done with a simple add-operation using local variable y.

C sourcecode FIR - IQmath

FIR Filter Implementation in C

```
/****************************************************************
* Function: IQssfir()
* Description: IQmath n-tap single-sample FIR filter.
*
*        y(k) = a(0)*x(k) + a(1)*x(k-1) + ... + a(n-1)*x(k-n+1)
*
* DSP: TMS320F2812, TMS320F2811, TMS320F2810
* Include files: DSP281x_Device.h, IQmathLib.h
* Function Prototype: _iq IQssfir(_iq*, _iq*, Uint16)
* Useage: y = IQssfir(x, a, n);
* Input Parameters: x = pointer to array of input samples
*                   a = pointer to array of coefficients
*                   n = number of coefficients
* Return Value: y = result
* Notes:
* 1) This is just a simple filter example, and completely
*    un-optimized. The goal with the code was clarity and
*    simplicity,
*    not efficiency.
* 2) The filtering is done from last tap to first tap.  This
*    allows
*    more efficient delay chain updating.
****************************************************************/
```

14 - 19

FIR Filter Implementation in C

```
_iq IQssfir(_iq *x, _iq *a, Uint16 n)
{
Uint16 i;                  // general purpose
_iq y;                     // result
_iq *xold;                 // delay line pointer

/*** Setup the pointers ***/
a = a + (n-1);             // a points to last coefficient
x = x + (n-1);             // x points to last buffer element
xold = x;                  // temporary buffer

/*** Last tap has no delay line update ***/
y = _IQmpy(*a--, *x--);

/*** Do the other taps from end to beginning ***/
for(i=0; i<n-1; i++)
{
 y = y + _IQmpy(*a--, *x);         // filter tap
 *xold-- = *x--;           // delay line update
}
return(y);
}
```

14 - 20

FIR Implementation in Assembly Language

Although the previous example of a C based implementation of a FIR algorithm used IQ-Math function calls to calculate the next value for the output, there is still headroom to optimize the FIR code for the C28x. As we have seen from the C example, basic mathematical operations in this algorithm are multiply instructions for each "coefficient" and "sample" in the delay chain and add operations to sum the partial products. To prepare the next filter calculation cycle, all sample values have been shifted by one position after they have been processed.

DSP's have a unique group of assembly instructions that take advantage of the parallel hardware units: "Arithmetic Logic Unit (ALU)" – for the sum operation and "Hardware Multiplier (MUL)" - for the multiplication. Thanks to the Harvard Architecture of DSP's, two operands can be read simultaneously – one from the data bus and the other from the program-bus.

The assembly language instruction set supports these types of operations in single clock cycle with the "Multiply and Accumulate" (MAC) instruction. In case of the C28x, two groups of assembly instructions are available:

- MAC   for 16-bit and 32-bit operands

- DMAC   for 16-bit operands

The DMAC – "Dual Mac" instruction takes advantage of the 32-bit width of the internal busses and processes four 16-bit operands (two coefficients and two samples) in a single cycle.

## FIR Filter Implementation in ASM

◆ **FIR Filter Optimization:**

> ➢ Previous C solution is a generic one, coded in standard ANSI-C, can be compiled for every microprocessor or embedded microcomputer. Works well.

> ➢ But is not optimized for Digital Signal Processors like the C28x. In case more computing power for the real time calculation of a FIR is needed, one can take advantage of internal parallel hardware resources of a DSP.

> ➢ ASM-coding of a FIR allows to reduce the number of clock cycles needed to calculate one loop of the FIR algorithm.

> ➢ A new Addressing Mode is used to avoid the shift operations of the delay-line for input samples: "Circular Addressing Mode"

> ➢ Describe the Circular Addressing function

> ➢ Implement FIR filters using Circular Addressing Mode

14 - 21

Circular Addressing Mode

Knowing that MAC or DMAC will accelerate FIR-code the last portion to be optimized is the shift operation of samples, after they have been processed. Assembly language addressing modes of operands include one particular mode that is used to avoid these shift operations altogether. We won't dive too deep into assembly language programming yet, but to explain this addressing mode, let us take an example.
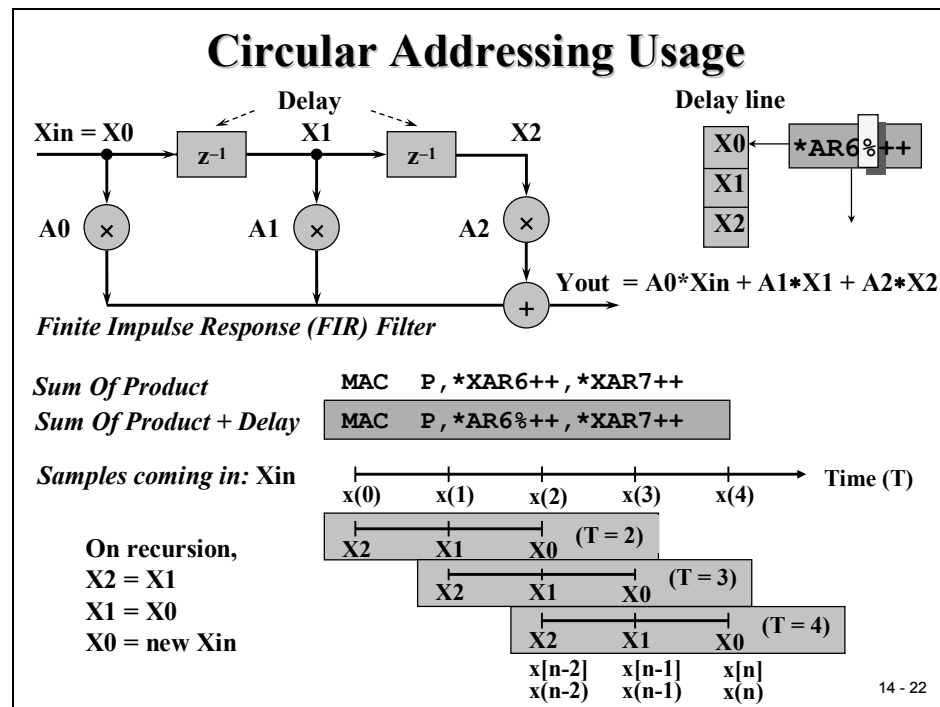
The new addressing mode is called "Circular Addressing Mode" and it is coded with the percentage (%) –sign in front of the pointer register name:

The instruction

MAC P,*XAR6++,*XAR7++

- Adds the previous product (stored in P) to the pre-accumulated sum in register ACC

- Multiplies the data memory operand, pointed to by XAR6, by the program memory operand, pointed to by XAR7

- Post-increments the two pointers XAR6 and XAR7.

If we introduce a new syntax:

MAC P,*AR6%++,*XAR7++

The first pointer XAR6, which points to the sample array in data memory, is used in a circular fashion. Once the pointer has reached the end of the array it will store the next value at start of the buffer. We call this "Circular Buffer".

The next slide explains this with a six element buffer, which is used to store the name of month. If the last space in buffer was used, the month "July" will replace the oldest entry "January"
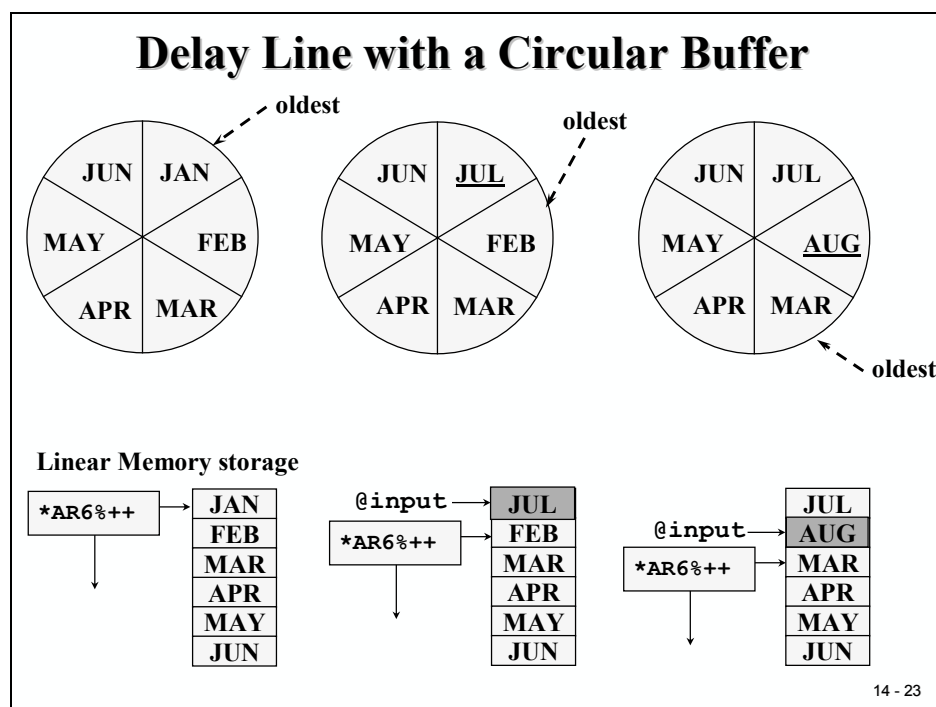
Before we use this circular addressing mode, register XAR7 must be aligned to point to the first element of the coefficient array.

XAR7 must be initialized to point to the start of the circular buffer. This start address of the circular buffer must be aligned to a 256-word boundary (8 LSB's = 0000 0000), which is usually done with the help of a linker command file instruction (see next slide).

The lower 8 bits of register XAR1 are used to specify the size of the circular buffer. This is an implicit usage of register XAR1 by the circular addressing mode; it is not shown in the assembly code!
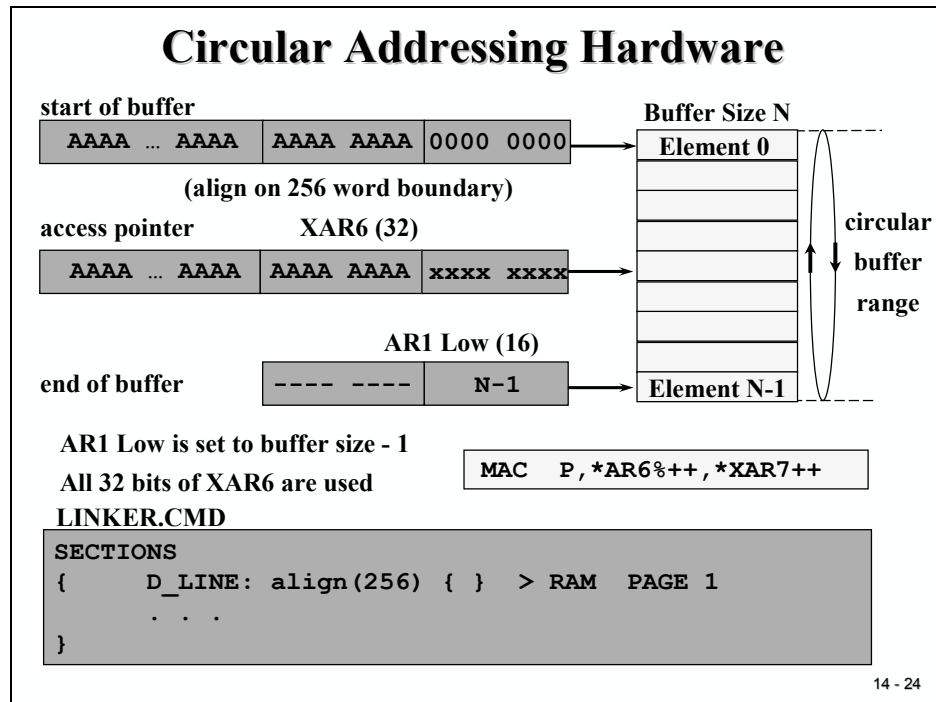
FIR- Filter Code

The next slide is an assembly language implementation of a 3rd order FIR filter (4 taps). It can be adapted to higher orders by changing the constant "TAPS" and the number of coefficients in array "tbl". All operands are 16-bit wide in Q15-format.

The directive ".usect" defines un-initialized data memory of length "TAPS" and assigns it to symbol "xn". The linker command file will connect this section "D_LINE" to physical memory. The directive ".data" defines initialized code memory and assigns it to symbol "tbl". The four coefficients in this array are in I1Q15-format.

The directive ".text" opens the code-section for assembly instructions. After setting core op-mode bits (SXM, OVM, PM) the two pointers XAR6 and XAR7 are initialized to point to symbol "xn" and "tbl" respective. AR1 is loaded with the buffer size.

Next, a new sample is read from the external ADC at data memory address "0:adc" and stored at first place in the circular buffer. We also could have used the internal ADC. The '%++' operator will set the circular buffer pointer to its next element.

After clearing register ACC, P and OV ("ZAPA") the following two instructions will do the entire filter work. The repeat instruction ("RPT") tells the C28x to repeat the following instruction #number plus 1 times, in our case twice. The instruction "DMAC" will perform two 16x16-bit MAC operations, reading and processing two members of "xn" and "tbl" per cycle. The 3rd order FIR filter will be calculated in two clock cycles.

Finally the two halves of the result are added using the instruction "ADDL ACC:P" and the new result is loaded in I1Q15-format to an external DAC at address "0:dac".

FIR Filter – Dual MAC - Operation

```
TAPS  .set   4                        ; FIR – Order +1
xn    .usect "D_LINE",TAPS            ; sample array in I1Q15

      .data                           ; FIR – Coeffs in I1Q15
tbl   .word  32768*707/1000          ; 0.707
      .word  32768*123/1000          ; 0.123
      .word  32768*(-175)/1000       ; -0.175
      .word  32768*345/1000          ; 0.345

      .text
FIR:  SETC  SXM                       ; 2's complement math
      CLRC  OVM                       ; no clipping mode
      SPM   1                         ; fractional math
      MOVL  XAR7,#tbl                 ; coefficient pointer
      MOVL  XAR6,#xn                  ; circular buffer pointer
      MOV   AR1,#TAPS-1               ; buffer offset
      MOV   *XAR6%++,*(0:adc)         ; get new sample ( x(n) )
      ZAPA                            ; clear ACC,P,OVC
      RPT   #(TAPS/2)-1               ; RPT next instr.(#+1)times
||    DMAC  ACC:P,*XAR6%++,*XAR7++    ; multiply & accumulate 2pairs
      ADDL  ACC:P                     ; add even & odd pair-sums
      MOV   *(0:dac),AH               ; update output ( y(n) )
      RET
```

14 - 25

Circular Addressing Summary

Buffer Size

◆ **Up to 256 words**

◆ **Break larger arrays into <= 256 word blocks.**

Buffer Alignment

◆ **Always align on 256-word boundaries, regardless of size. Unused space can be used for other purposes.**

◆ **Let the linker assign addresses. Link largest blocks first.**

Usage

◆ **XAR6 is the only circular pointer.**

◆ **AR1 must be set to the size minus one (0 - 255).**

◆ **Pointer update is post-increment by one (*XAR6%++) .**

◆ **32-bit access causes post-increment by two. Make sure XAR6 and AR1 are even to avoid jumping past end of buffer.**

14 - 26

Texas Instruments C28x Filter Library

So for some types of applications, it seems to make sense to code in Assembly Language. But, as usual, there is no need to re-invent the wheel. Better than developing your own code is – use a library. Texas Instruments is offering a variety of libraries for free, one of them us dedicated to Digital Filters.

Texas Instruments C28x Filter Library

- ◆ **MATLAB script to calculate Filter Coefficients for FIR and IIR, includes windowing**
- ◆ **Filter Modules:**
    - ◆ **FIR16:**   **16-Bit FIR-Filter**
    - ◆ **IIR5BIQ16:**   **Cascaded IIR-Filter (16bit-biquad)**
    - ◆ **IIR5BIQ32:**   **Cascaded IIR-Filter (32bit-biquad)**

- ◆ **C-callable Assembly ("CcA") Functions**
    - ◆ **Adapted to internal Hardware features of the C28x**
    - ◆ **Uses the Dual –MAC instruction**
    - ◆ **Interface according to ANSI-C standard**

Available from TI-web as document "sprc082.zip"

14 - 27

To make it easier to use this library in a C-based program environment, all library functions are equipped with an interface structure. Thus any library function can be called like an ordinary C subroutine.

An important step in designing a Digital Filter is to calculate Filter Coefficients. This task involves a lot of theoretical background. Without this knowledge, you won't be able to profile the set of coefficients for a given transfer function. As recommended earlier, join additional courses at your university to understand the math behind Digital Signal Processing.

The library package includes also a MATLAB script to calculate filter coefficients, including windowing techniques.

MATLAB Filter Script

The MATLAB filter script allows you to initialize essential parts of the transfer function, like sampling frequency, filter type, order, type of window and corner frequency.

MATLAB Filter Script

FIR Filter Design Example: Low-pass Filter of Order 50
LPF Specification:
FIR Filter Order : 50
Type of Window : Hamming
Sampling frequency : 20KHz
Filter Corner Frequency : 3000Hz

ezFIR FILTER DESIGN SCRIPT
Input FIR Filter order(EVEN for BS and HP Filter) : 50
Low Pass : 1
High Pass : 2
Band Pass : 3
Band Stop : 4
Select Any one of the above Response : 1
Hamming : 1
Hanning : 2
Bartlett : 3
Blackman : 4
Select Any one of the above window : 1
Enter the Sampling frequency : 20000
Enter the corner frequency(Fc) : 3000
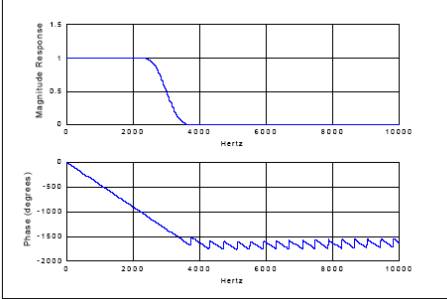Enter the name of the file for coeff storage : lpf50.dat

14 - 28

The output of the MATLAB calculation is a list of coefficients and a graph of magnitude and phase of the filter response.

MATLAB Filter Script

MATLAB – Output File for Filter Coefficients:

```
#define FIR16_COEFF {\
9839,-2219809,-1436900,853008,3340889,3668111,-896,\
-5963392,-8977456,-3669326,8585216,18152991,13041193,\
-8257663,-30867258,-31522540,131,45285320,64028535,\
25231269,-58654721,-124846025,-94830542,68157453,\
320667626,551550942}
```


14 - 29

FIR16 Library Function

FIR16 is one of the library functions of sprc082. It processes a single stream of input samples in Q15-format into a new output value of the same format. One instance of this function occupies 52 words of code memory and processing time is 350 ns with a 150 MHz C28x.
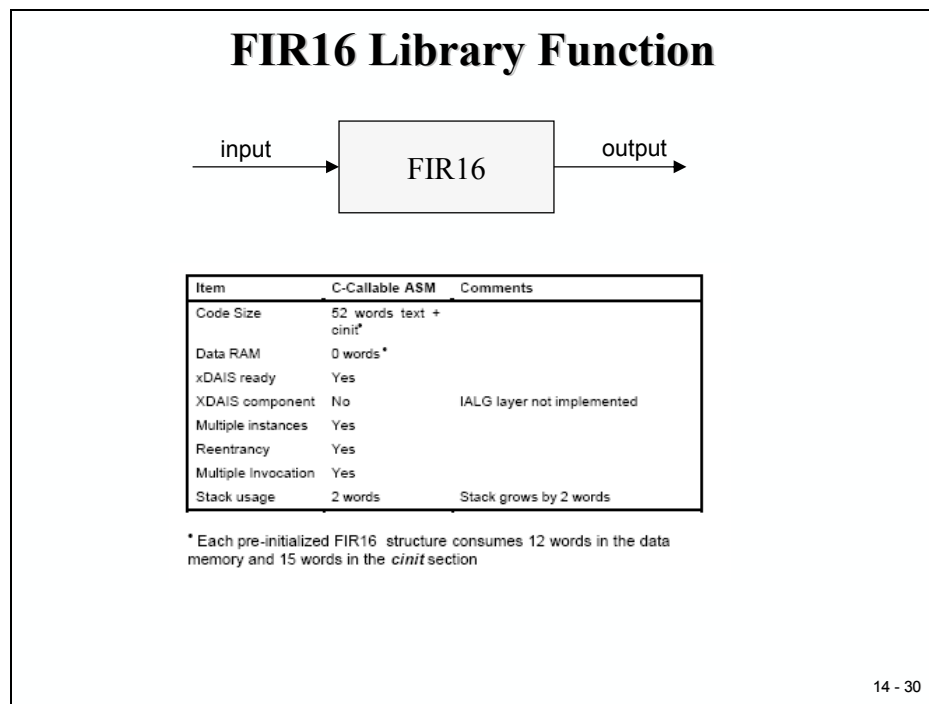
The format of the FIR16 functions object structure is shown next. Interface parameters are

Input:

- 2 pointers to coefficients and samples

- Order of filter

- 1 pointer to an initialize function of the filter

- 1 pointer to the calculation function

Output:

- 1 new filter output value.

To guarantee the ability to operate with more than 1 instance of the filter, you should call the function by its function parameters (init, calc) only.

FIR16 Library Function

Object Definition:

```
typedef struct {
        long *coeff_ptr;        /* Pointer to Filter coeffs  */
        long *dbuffer_ptr;      /* Delay buffer pointer      */
        int cbindex;            /* Circular Buffer Index     */
        int order;              /* Order of the Filter       */
        int input;              /* Latest Input sample       */
        int output;             /* Filter Output             */
        void (*init)(void *);   /* Pointer to Init function   */
        void (*calc)(void *);   /* Pointer to calc function  */
        }FIR16;
```

coeff_ptr:	Pointer to the Filter coefficient array.
dbuffer_ptr:	Pointer to the Delay buffer.
cbindex:	Circular buffer index, computed internally by initialization function based on the order of the filter.
order:	Order of the Filter. Q0-Format, range 1 – 255
input:	Latest input sample to the Filter. Q15-Format (8000-7FFF)
output:	Filter output value. Q15-Format (8000-7FFF)

14 - 31

The next slide is an example for the usage of the filter function. An instance of FIR16, called "lpf" has been defined in a DATA_SECTION "firfilt". All accesses to parameters and functions are made by this instance.

FIR16 Library Usage Example

```
#define FIR_ORDER 50            /* Filter Order */

#pragma DATA_SECTION(lpf, "firfilt");
FIR16 lpf = FIR16_DEFAULTS;
#pragma DATA_SECTION(dbuffer,"firldb");
long dbuffer[(FIR_ORDER+2)/2];

const long coeff[(FIR_ORDER+2)/2]= FIR16_LPF50;
main()
{
    lpf.dbuffer_ptr=dbuffer;
    lpf.coeff_ptr=(long *)coeff;
    lpf.order=FIR_ORDER;
    lpf.init(&lpf);
}
void interrupt isr20khz()
{
    lpf.input=xn;
    lpf.calc(&lpf);
    yn=lpf.output;
}
```

14 - 32

Lab 14: FIR – Filter for a square-wave signal

Objective

<div style="border:1px solid">

Lab 14: LP -Filter of a square wave

Objective:

◆ **Generate a square wave signal of 2 KHz at EVA-T1PWM**

◆ **Asymmetric PWM , duty cycle 50%**

◆ **Use T1-Compare Interrupt Service to serve the watchdog**

◆ **Wire - Connect T1PWM to ADC-input ADCIN2**

◆ **Sample the square wave signal at 50KHz**

◆ **Sample period generated by EVA-Timer 2**

◆ **Store samples in buffer "AdcBuf"**

◆ **Filter the input samples with a FIR – Low pass 4th order**

◆ **Store filtered output samples in buffer "AdcBufFiltered"**

◆ **Visualize "AdcBuf" and "AdcBufFiltered" graphically by Code Composer Studio's Graph Tool**

14 - 33

</div>

The lab experiment consists of four parts:

- First we will generate a 2 kHz square wave signal at output T1PWM. At the Zwickau adapter board this signal can be connected to a loudspeaker (JP3 closed). Or, use a scope to visualize the signal.

- Second we will feedback this signal back into one channel of the internal ADC and store the digital samples in a data memory buffer "AdcBuf".

- Next, we call a Low-Pass Filter of FIR-Type with order 4 to wave-shape the signal edges. The coefficients were calculated with MATLAB as:

    o 1/16, 4/16, 6/16, 4/16 and 1/16

    o The sampling frequency is set to 50 kHz

- The filtered numbers will be stored in "AdcBufFiltered"

- Finally we will use Code Composer Studios graphical tool to visualize the contents of "AdcBuf" and "AdcBufFiltered". We will take advantage of the real time debug capabilities to display the data without interrupting or delaying the C28x while it is running!
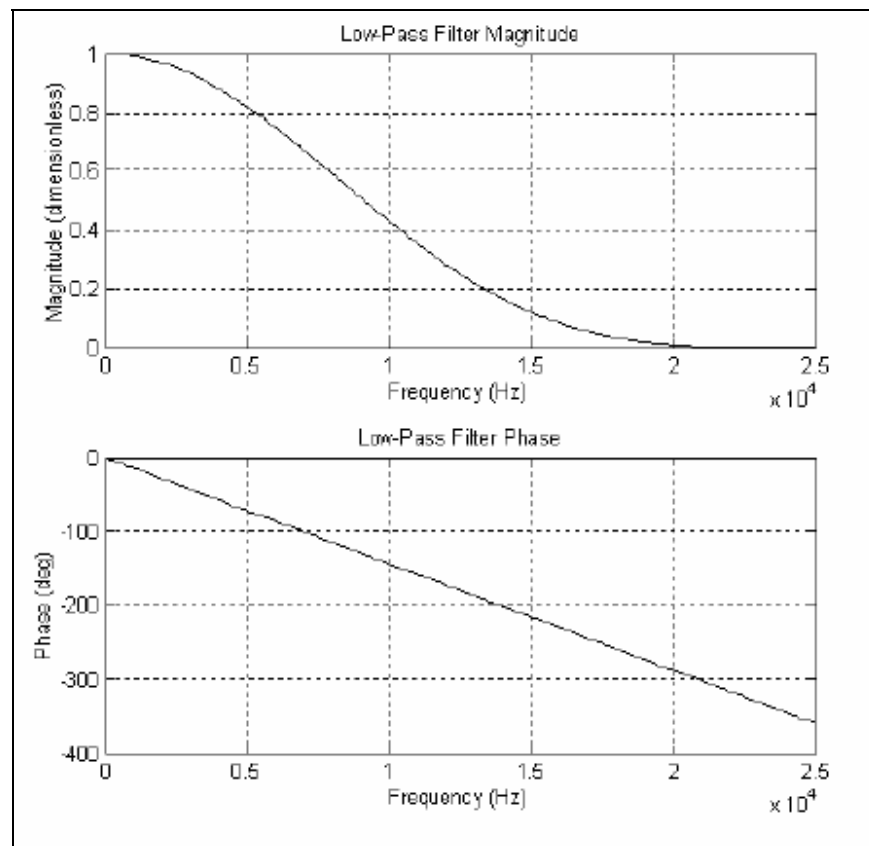
Procedure

Open Files, Create Project File

1. Create a new project, called **Lab14.pjt** in E:\C281x\Labs.

2. Open the file Lab5.c from E:\C281x\Labs\Lab5 and save it as Lab14.c in E:\C281x\Labs\Lab14.

3. Add the source code file to your project:
   - **Lab14.c**

4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:

   - **DSP281x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:

   - **F2812_EzDSP_RAM_lnk.cmd**

From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:

- **F2812_Headers_nonBIOS.cmd**

From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:

- **DSP281x_PieCtrl.c**

- **DSP281x_PieVect.c**

- **DSP281x_DefaultIsr.c**

From *C:\ti\c2000\cgtoolslib* add:

- **rts2800_ml.lib**

Project Build Options

5. Setup the search path to include the peripheral register header files. Click:

   Project → Build Options

   Select the Compiler tab. In the preprocessor Category, find the Include Search Path (-i) box and enter:

   **C:\tidcs\C28\dsp281x\v100\DSP281x_headers\include;
   ..\include; C:\tidcs\C28\IQmath\cIQmath\include**

6. Setup the stack size: Inside Build Options select the Linker tab and enter in the Stack Size (-stack) box:

   400

   Close the Build Options Menu by Clicking <**OK>**.

Modify Source Code

7. Open Lab14.c to edit: double click on "Lab14.c" inside the project window. First we have to remove the parts of the code that we do not need any longer. We will not use the CPU core timer 0 in this exercise; therefore we do not need the prototype of interrupt service routine "cpu_timer0_isr()". Instead, we need a new ISR for EVA-Timer1-Compare-Interrupt. Add a new prototype interrupt function: "interrupt void T1_Compare_isr(void)".

8. We do not need the variables "i","time_stamp" and frequency[8]" from Lab5 - delete their definition lines at the beginning of the function "main".

9. Next, modify the re-map lines for the PIE entry. Instead of "PieVectTable.TINT0 = & cpu_timer_isr" we need to re-map:

PieVectTable.T1CINT = &T1_Compare_isr;

10. Delete the next two function calls: "InitCpuTimers();" and "ConfigCpuTimer(&CpuTimer0, 150, 50000);" and add an instruction to enable the EVA-Timer1-Compare interrupt. According to the interrupt chapter this source is wired to PIE-group 2 , interrupt 5:

PieCtrlRegs.PIEIER2.bit.INTx5 = 1;

Also modify the set up for register IER into:

IER | = 2;

11. Next we have to initialize the Event Manager Timer 1 to produce a PWM signal. This involves the registers "GPTCONA", "T1CON", "T1CMPR" and "T1PR".

For register "GPTCONA" it is recommended to use the bit-member of this predefined union to set bit "TCMPOE" to 1 and bit field "T1PIN" to "active low".

For register "T1CON" set

- The "TMODE"-field to "counting up mode";

- Field "TPS" to "divide by 1";

- Bit "TENABLE" to **"disable timer";**

- Field "TCLKS" to "internal clock"

- Field "TCLD" to "reload on underflow"

- Bit "TECMPR" to "enable compare operation"

12. Remove the 3 lines before the while(1)-loop in main:

- "CpuTimer0Regs.TCR.bit.TSS = 0;"

- "i = 0;"

- "time_stamp = 0;"

and add 4 new lines to initialise T1PR, T1CMPR, to enable GP Timer1 Compare interrupt and to start GP Timer 1:

EvaRegs.T1PR = 37500;

EvaRegs.T1CMPR = EvaRegs.T1PR/2;

EvaRegs.EVAIMRA.bit.T1CINT = 1;

EvaRegs.T1CON.bit.TENABLE = 1;

What is this number 37500 for? Well, it defines the length of a PWM period:

$$f_{PWM} = \frac{f_{CPU}}{T1PR \cdot TPS_{T1} \cdot HISCP}$$

with $TPS_{T1}=1$, HISCP = 2, f_{CPU} = 150MHz and a desired f_{PWM} = 2 kHz we derive: T1PR = 37500!

T1CMPR is preloaded with half of T1PR. Why's that? Well, in general T1CMPR defines the width of the PWM-pulse. Our start-up value obviously defines a pulse width of 50%.

13. Modify the endless while(1) loop of main! We will perform all activities out of GP Timer 1 Compare Interrupt Service. Therefore we can delete almost all lines of this main background loop, we only have to keep the watchdog service:

> **while(1)**
>
> **{**
>
> > **EALLOW;**
> >
> > **SysCtrlRegs.WDKEY = 0xAA;**
> >
> > **EDIS;**
>
> **}**

14. Rename the interrupt service routine "cpu_timer0_isr" into "T1_Compare_isr". Remove the line "CpuTimer0.InterruptCount++;" and replace the last line of this routine by:

> **PieCtrlRegs.PIEACK.all = PIEACK_GROUP2;**

Before this line add another one to acknowledge the GP Timer 1 Compare Interrupt Service is done. Remember how? The Event Manager has 3 interrupt flag registers "EVAIFRA","EVAIFRB" and "EVAIFRC". We have to clear the T1CINT bit (done by setting of the bit):

> **EvaRegs.EVAIFRA.bit.T1CINT = 1;**

Build and Load

15. Click the "Rebuild All" button or perform:

> **Project → Build**

and watch the tools run in the build window. If you get syntax errors or warnings debug as necessary.

16. Load the output file down to the DSP Click:

    File → Load Program and choose the desired output file.

Test

17. Reset the DSP by clicking on:

    Debug → Reset CPU   followed by
    Debug → Restart   and
    Debug → Go main.

18. When you now run the code the DSP should generate a 2 kHz PWM signal with a duty cycle of 50% on T1PWM. If you have an oscilloscope you can use jumper JP7 (in front of the loudspeaker) of the Zwickau Adapter board to measure the signal.

    If your laboratory can't provide a scope, you can set a breakpoint into the interrupt service routine of T1 Compare at line "PieCtrlRegs.PIEACK.all = PIEACK_GROUP2; Verify that your breakpoint is hit periodically, that register T1PR holds 37500 and register T1CMPR is initialized with 18750. Use the Watch Window to do so.

    Do not continue with the next steps until this point is reached successfully! Instead, go back and try to find out what went wrong during the modification of your source code.
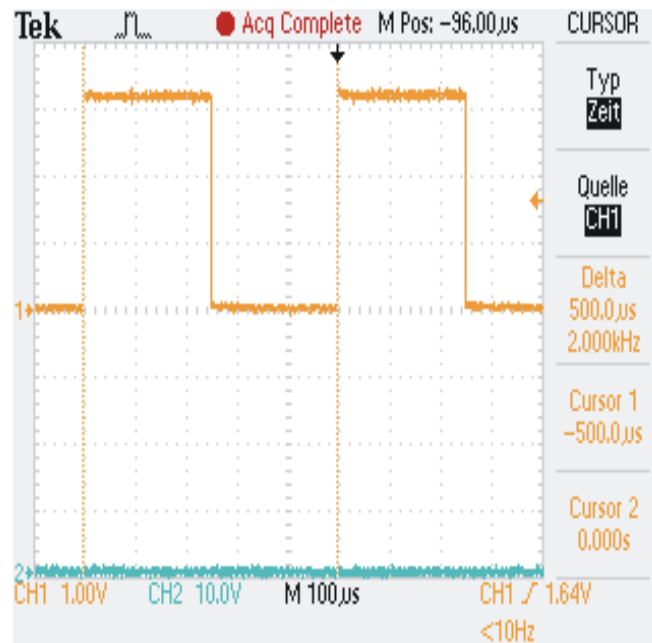
End of Lab 14 Part 1

Feedback the Signal into ADC

19. Three files have been provided to this lab to add the ADC functionality. Add the two files "Adc.c", "Adc_isr.c" and "filter.c" to your project.

20. In function "InitSystem" of Lab14.c enable the ADC-clock:

    SysCtrlRegs.PCLKCR.bit.ADCENCLK = 1;

21. In "main", just after the call of "InitPieVectTable()" add a call to initialize the ADC:

    InitAdc();

    This function will setup the ADC to one conversion per trigger. ADCIN2 will be converted by SEQ1 out of Event Manager A trigger. An interrupt will be requested with every end of sequence. Inspect the code of "InitAdc()".

22. Next, we have to connect the ADC interrupt to a new function: "ADC_FIR_INT_ISR()". This function is defined in the new source code file "Adc_isr". All we have to do is to replace the entry in the PieVectTable by this new address. Look in "main" and locate the line, where we already overload the PieVectTable with T1_Compare_isr. Add a new line:

    PieVectTable.ADCINT = &ADC_FIR_INT_ISR;

    The new function "ADC_FIR_INT_ISR" is not declared yet in "Lab14.c". Therefore we have to add a new prototype statement at begin of "Lab14.c":

    interrupt void ADC_FIR_INT_ISR(void);

    Register IER must be modified to enable INT1 (ADC) and INT2 (T1-Compare):

    IER |= 3;

Set up ADC sample period (Timer 2)

23. EVA-Timer 2 will be used to generate the sample period for the ADC. Each period event of T2 will trigger a start of an ADC sequence automatically, if we enable this option:

    EvaRegs.GPTCONA.bit.T2TOADC = 2;

    Add this line in front of the while(1)-loop of "main".

    Before the code enters the while(1)-loop we have to initialize EVA-Timer2 to produce a sample period of 50 kHz. Register T2CON defines the operating mode. Let's select:

    - Continuous up – Mode

    - Timer – Prescaler : 1

- Enable Timer (TENABLE = 1)

- No Timer Compare Operation Enable

Register T2PR must define the time period. According to:

$$f_{PWM} = \frac{f_{CPU}}{T2PR \cdot TPS_{T2} \cdot HISCP}$$

and with a given 150MHz CPU frequency, HISCP =2, TPST2 = 1 and 50 KHz as output frequency we derive:

T2PR = 1500

Add the necessary instructions for T2PR and T2CON!

Connect T1PWM to ADCIN2

24. Connect T1PWM (eZdsp Pin P8 -15) to ADCIN2 (eZdsp Pin P9 -6) with a wire or a 1000 Ohm resistor provided by your laboratory technician.

Caution: Be careful when connecting pins while the eZdsp is powered on. By plugging the wire into wrong pins you can damage the board!

To be safe, ask your technician for assistance before you connect anything!

Build, Load and Test

25. Finalize the Project and prepare a test: :

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

If all code was modified correct, the 2 kHz-Signal should still be auditable at the loudspeaker.

To verify that our ADC sampling is operating as expected, place a breakpoint at begin of the ADC's interrupt service routine ("ADC_FIR_INT_ISR") in file "Adc_isr.c". If you run the code in real time (F5), it should hit the breakpoint periodically.
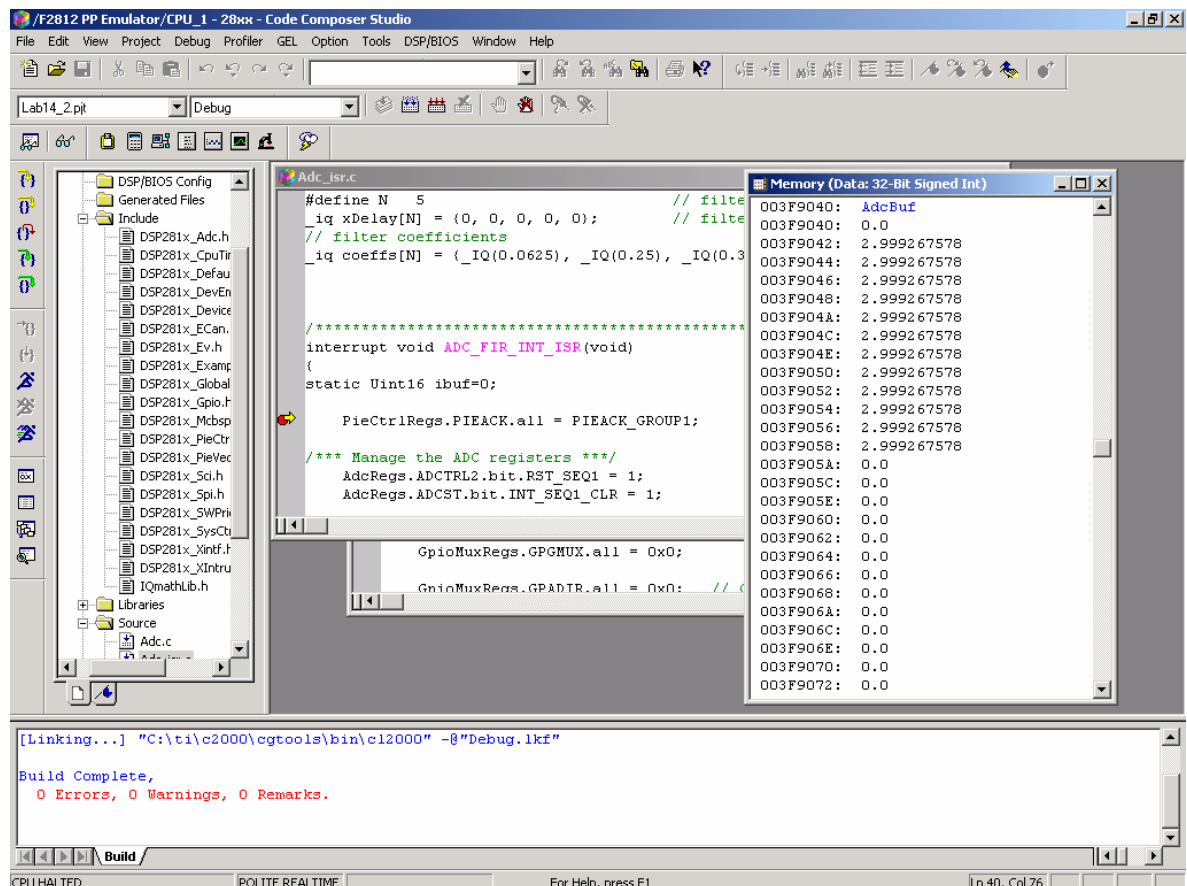
Inspect and Visualize the FIR

26. Let us inspect the code in this ISR. The sample (AdcRegs.ADCRESULT0) is stored in a buffer "AdcBuf" after it is scaled to 3.0V and converted into an IQ-number. The default value is I8Q24, defined in "IQmathLib.h".

    We can display the content of this buffer in a memory window:

    View → Memory

    - Address: AdcBuf
    - Q-Value: 24
    - Format: 32_Bit Signed Int
    - Page: Data

27. Back to "ADC_FIR_INT_ISR". After the sample is stored in AdcBuf, it is also placed as latest sample in a filter-array "xDelay". Then a function "IQssfir" is called and its return value is stored in a new buffer "AdcBufFiltered". Obviously, the return value is the output signal from the FIR-Filter.

28. Now inspect the filter-function "IQssfir" in "filter.c". Input parameters are the samples, the coefficients and number of taps. The filter implementation is a C-based solution with no optimization. The difference to the solution that was presented at the beginning of the chapter is that it is a tailored solution for the C28x IQ-Math Library, running much faster than any ANSI-C solution with float variables.

    You have also learned about Texas Instruments Filter Library. Knowing that these library functions are based on an assembly language implementation we could move on and increase the speed of the FIR-calculation further by replacing the "IQssfir"-function with one from the library.

CCS Graphical Tool

29. Now let's visualize both the square wave and the filtered signal. CCS has a build in tool to visualize the content of an area of code or data memory graphically. We can use this tool to plot the content of "AdcBuf" and "AdcBufFiltered":

    View ➔ Graph ➔ Time/Frequency

    Select the properties:

    - Display Type :   Dual Time

    - Start Address upper display: AdcBuf

    - Start Address lower display: AdcBufFiltered

    - Page: Data

    - Acquisition Buffer Size: 50

    - Display Data Size: 50

    - DSP Data Type: 32-bit signed integer

    - Q-Value   24

    - Sampling Rate: 50000

    - Time Display Unit: μs

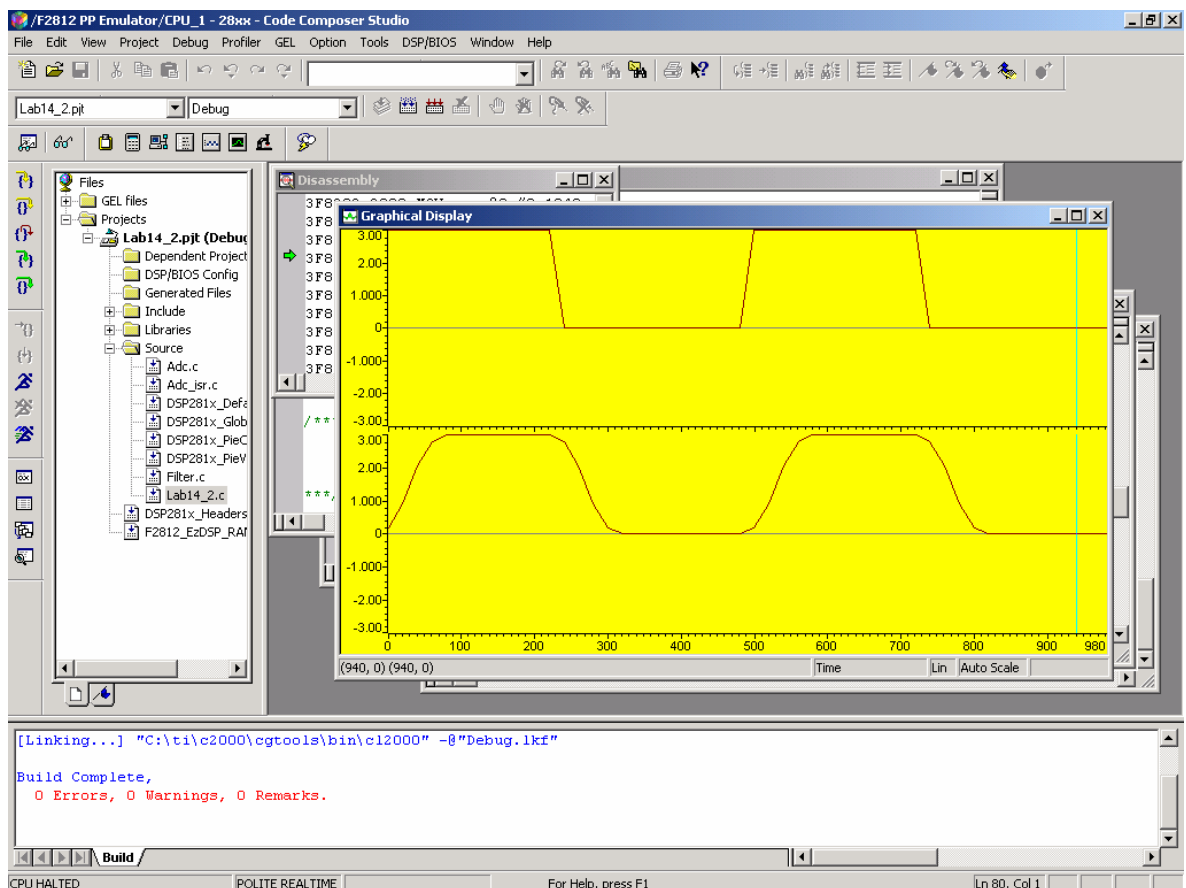   When you close the property window with <OK> a Graphical Display with a yellow background should pop up.

30. Enable Real Time Mode:

   Debug ➔ Real Time Mode

   Click right in the graph window and select "Continuous Refresh"

31. Run the code in Real Time (F5)

The Graph window should look like this:

End of Lab 14

This page has intentionally been left blank.