# *Oversampling Techniques using the TMS320C24x Family*

Literature Number: SPRA461
Texas Instruments Europe
June 1998

# Contents

# Oversampling Techniques using the TMS320C24x Family

**ABSTRACT**

This document describes the theory of oversampling, the hardware and software implementation on the TMS320C240 and important aspects that need to be considered when using oversampling.

The different techniques with their main advantages, limitations and applications are covered in chapter 2. Chapter 3 describes the key implementation blocks that are required to implement oversampling on the TMS320C24x family. This includes a description of the TMS320C240, the Evaluation Module (EVM) and dedicated oversampling hardware as well as the software routines.

# 1. Introduction

Analog to Digital conversion is usually the first step in typical Digital Signal Processing applications. Depending on the type of application there are different requirements on the Analog to Digital Converter (ADC), which could be an external device or an integrated module in the DSP itself. The most important criteria for AD-Converters are the maximum sample rate, the resolution and accuracy.

Oversampling methods are able to increase the resolution by sampling the analog signal at much higher rates. Using the higher rate of sampling and applying suitable digital filters -- implemented in DSP algorithms -- will result in an effective resolution higher than that of the original ADC.

# 2. Oversampling Theory

Oversampling methods for Analog-to-Digital Converters are based on sampling an analog signal at a much higher rate, filtering the samples with a digital low-pass filter and reducing the sample rate by decimation. In some implementations the filtering and decimation process is done in several stages in order to achieve the desired results with a minimum of computing performance.

That higher sampling rate - while maintaining the band of interest – makes it possible to relax the constraints on the analog signal spectrum and increases the resolution of such a subsystem.

## 2.1 Nyquist Theorem

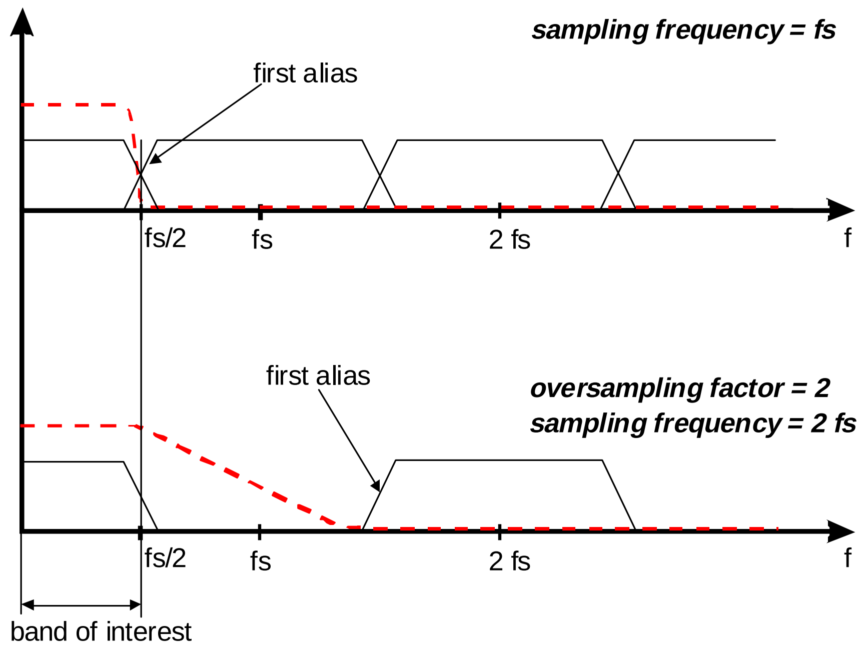
Any analog signal that needs to be digitized has to be band limited and should not contain any frequencies above half the sample frequency $f_s$. Violating the Nyquist Theorem causes aliasing effects and the signal can not be fully reconstructed from the digital series of data.

Therefore analog to digital conversion applications require steep analog low-pass filters for limiting the frequency spectrum of the analog signal. An ideal filter would pass all frequencies lower than $f_s/2$ with an attenuation of 1 and completely block all frequencies above half the sample frequency.

Usually the filter and the sample rate are selected in such a way that the band of interest falls into the frequency range between DC and $f_s/2$.

Using a much higher sample rate lowers the constraints of the low-pass filter. The figure below shows an example, in which the sampling the analog signal is sampled at twice the original sampling frequency. In such a case the required analog filter rolls-off at $f_s/2$ in order to pass the full 'band of interest' and suppresses all significant frequency components above 1.5 $f_s$. Since this violates the Nyquist Theorem a digital filter can be used to limit the frequency band to frequencies below $f_s/2$.

Such an analog filter is much easier to implement than an anti-aliasing filter at a sampling frequency of $f_s$.



## 2.2 Quantization

After filtering, the analog signal is sampled and converted into digital values. Since the digital domain consists only of finite length digital words, which have to represent a continuous signal, the conversion step introduces quantization error.

The maximum error of an ideal quantizer is +/- 0.5 LSB. Since the input range of an N-bit ADC is divided into $2^N$ discrete levels each represented by an N-bit binary word, the ADC's input range and the word width N are a direct measure of the maximum absolute error.

## 2.3 Signal-to-noise ratio (SNR)

In addition, the quantization step can be analyzed in the frequency domain: The number of bits representing the digital value determines the signal-to-noise ratio. Therefore by increasing the signal-to-noise ratio the effective resolution of the conversion will be increased.

Assuming the input signal is an ever-changing signal, the error caused by quantization can be viewed as a white noise signal, that spreads its energy uniformly across the whole bandwidth from DC to one half of the sample rate.

However, it can be shown that for an ideal A/D Converter and an input signal with no, or low, change in the amplitude, the quantization error cannot be described as a white noise source. Therefore, in order to follow this theory the ADC input has to be provided with a continuously changing signal, in order to keep the quantizer output "busy". In practice this can be ensured by providing an additive error signal with at least 1 LSB amplitude at the ADC input. This signal is called a 'dithering signal'.

For a full-scale sinewave signal, the theoretical signal-to-noise ratio (SNR) can be approximated as:

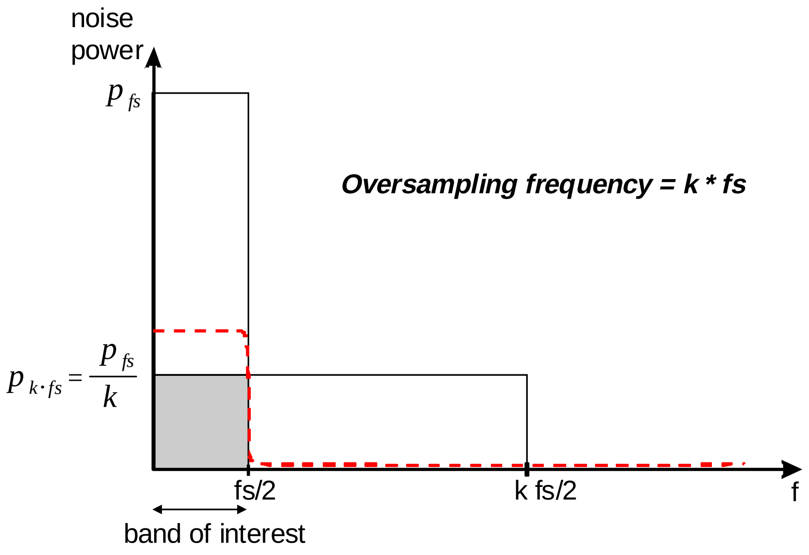
$$SNR_{\max}[db] \approx 6.02 \cdot N + 1.76$$

with N the number of bits representing the digital value.

## 2.4 Improving the Signal-to-noise ratio by adding white noise

As shown above, the quantization noise power is uniformly distributed across the spectrum between DC and half the sample rate. This quantization noise power is independent of the sample rate. When using higher sample rates the noise power is spread over a wider range of frequencies. Therefore the effective noise power density at the band of interest is lower at higher sampling rates.

The figure below illustrates the effective noise power reduction in the frequency band of interest at an oversample rate of k and the resulting sample rate of $k \cdot f_s$.


noise power
$p_{fs}$
Oversampling frequency = k * fs
$p_{k \cdot fs} = \frac{p_{fs}}{k}$
fs/2
k fs/2
f
band of interest

Digital low-pass filters should remove all frequencies above $f_s/2$. The effective resolution is determined by the quality of the digital filter. The remaining noise power beyond $f_s/2$ is a measure for the quantization noise and therefore responsible for a decrease in the signal-to-noise ratio (SNR).

According to the Nyquist theorem a signal can be fully described and reconstructed if it does not contain any frequency components above half the sample rate. In other words, in order to describe a signal with $f_{\max}$ as the maximum frequency the series of the digital values can be reduced to the equivalent of $2 \cdot f_{\max}$ sample rate. This process is called 'decimation'.

In a typical oversampling system using an oversample factor of $k$, the samples are decimated - after or during the filtering - by a factor of $k$. In such cases an ideal low-pass filter and decimator will reduce the quantization noise by the factor of $k$. Since the signal at the band of interest is not affected by the filter, this leads to an enhancement of the signal-to-noise ratio.

The formula for the improved SNR is:

$$SNR_{\max}[db] \approx 6.02 \cdot N + 1.76 + 10 \cdot \log_{10}(k)$$

This leads to following improvements:

| Oversampling factor k | SNR improvement in dB | extra resolution in number of bits |
|---:|---:|:---:|
| 2 | 3 | 0,5 |
| 4 | 6 | 1,0 |
| 8 | 9 | 1,5 |
| 16 | 12 | 2,0 |
| 32 | 15 | 2,5 |
| 64 | 18 | 3,0 |
| 128 | 21 | 3,5 |
| 256 | 24 | 4,0 |
| 512 | 27 | 4,5 |
| 1024 | 30 | 5,0 |
| 2048 | 33 | 5,5 |
| 4096 | 36 | 6,0 |

### 2.4.1 Features and limitations

Oversampling using white noise provides about 3dB - or half a bit - of resolution gain for each doubling of the oversampling rate. For applications where this gain is sufficient this implementation is a good choice.

It requires a minimum of overhead to provide the white noise signal. Sometimes even the internal noise of the AD-Converter is already sufficient and there is no need for an external noise source.

The above method places no limitations on the waveform and can be used in many applications, especially where high oversampling factors can be used.

## 2.5 Improving the Signal-to-noise ratio by adding a triangular signal

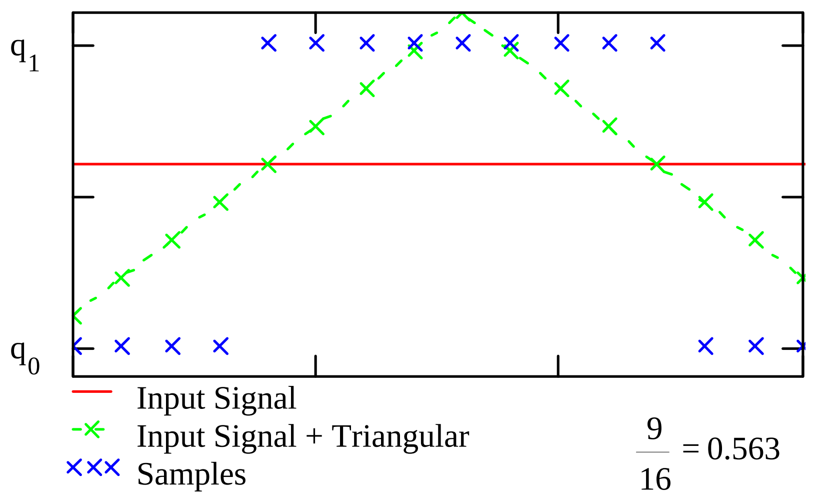
Another approach for achieving a twofold improvement of the resolution is to use a method similar to a Delta-Sigma modulator, by adding a triangular wave to the input signal.

If the input signal is between an upper ($q_1$) and a lower ($q_0$) quantization step, the quantizer would either convert into the upper or the lower value. With the addition of an appropriate triangular signal, the quantizer generates a series of $q_1$ and $q_0$. The ratio between the occurrence of those two values represents the relative position of the input signal between the two quantization steps. Or in other words, averaging the $q_1$ over a given interval determines the position between the upper and lower quantization values.

In order to get the best results from this method the triangular signal should have an amplitude of *n + 0.5 LSB* with *n* being 0, 1, 2…

Due to the high sample rate the changes in the input signal can be viewed as relatively small. The following figure displays an input signal at 0.6 LSB. A typical AD converter would sample the input signal and convert it into 1 in the LSB. When adding the triangular signal and sampling at a higher rate the conversion generates a number of samples with either 0 or 1. The ratio between the occurrences of 0 and 1 describes the actual value between 0 and 1 LSB.

In the figure below, an oversampling factor of 16 is used. This results in a ratio of seven 0- and nine 1-samples. The 9 to 16 ratio results in 0.563 and a lower quantization error than the representation with the 1 value.



In general the following equation applies to a triangular modulated oversampling:

$$SNR_{gain}[db] = 20 \cdot \log_{10}(\frac{k}{2})$$

This leads to the following SNR and extra resolution:

| Oversampling factor k | SNR improvement in dB | extra resolution in number of bits |
|---|---|---|
| 2 | - | 0 |
| 4 | 6 | 1 |
| 8 | 12 | 2 |
| 16 | 18 | 3 |
| 32 | 24 | 4 |
| 64 | 30 | 5 |
| 128 | 36 | 6 |
| 256 | 42 | 7 |
| 512 | 48 | 8 |
| 1024 | 54 | 9 |
| 2048 | 60 | 10 |
| 4096 | 66 | 11 |

### 2.5.1 Features and limitations

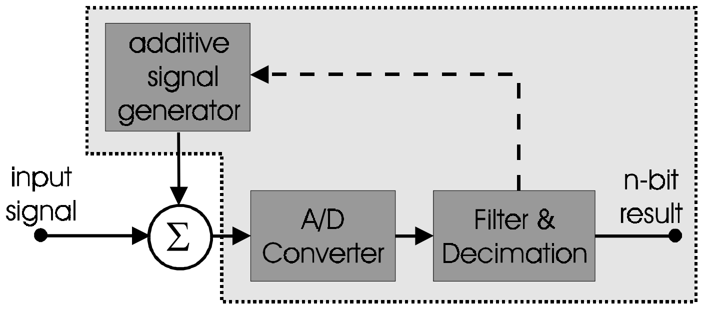
Oversampling using the addition of a triangular signal provides about 6dB - or one bit - of resolution gain for each doubling of the oversampling rate. This is twice the improvement achievable with the oversampling method using white noise as described in chapter 2.4.

However, this approach requires that the signal does not correlate with the triangular signal. If this cannot be ensured the signal should not change more than +/- 0.5 LSB from the expected resolution during an oversample period $k \cdot f_s$.

## 3. TMS320C24x Implementation

As explained in the previous chapter, there is no implementation that fits all needs for all characteristics of input signals, e.g. waveform or frequency. Therefore this chapter describes all the key building blocks to generate an oversampling implementation on the TMS320C240 EVM.

According to the theory, an oversampling AD converter consists of a conventional AD converter that samples the input signal plus an optional additive signal. Filtering and decimation of the series of samples returns the higher resolution result. The figure below illustrates the block diagram.



The devices of the TMS320C24x family includes all components drawn in the dotted block, which simplifies the design of such an oversampling subsystem.

The TMS320C240 device includes two independent 10 bit AD-Converters. The PWM Output signals can be used to generate the additive signal synchronized to the oversampling factor. The Digital Filter and Decimation can be implemented in software running on the TMS320C24x with up to 20 MIPS.

## 3.1 TMS320C24x Architecture

The TMS320C24x branch is a member of the TMS320C2xx family. It combines the TMS320C2xx DSP Core with many peripherals suited for Digital Control System Applications.

### *3.1.1 TMS320C24x CPU*

The TMS320C24x family uses the TMS320C2xx 16-bit core CPU, with a 32-bit wide central arithmetic logic unit and accumulator. Together with the 16-bit x 16-bit parallel multiplier many arithmetic operation can be performed in a single cycle allowing efficient implementations of filters with a 16-bit data width. Eight 16-bit auxiliary registers allow flexible addressing modes.

The TMS320C240 device contains 544 words (16-bit) of on-chip data/program dual-access RAM and 16K words of on-chip program ROM.

### *3.1.2 TMS320C240 Peripherals*

Some of the TMS320C240 peripherals can be utilized for the oversampling subsystem. However, not all of them are used by the application. The remaining peripherals can still be used by other applications or tasks running on the DSP. The table below lists all peripherals of the TMS320C240 device and its usage by the oversampling application.

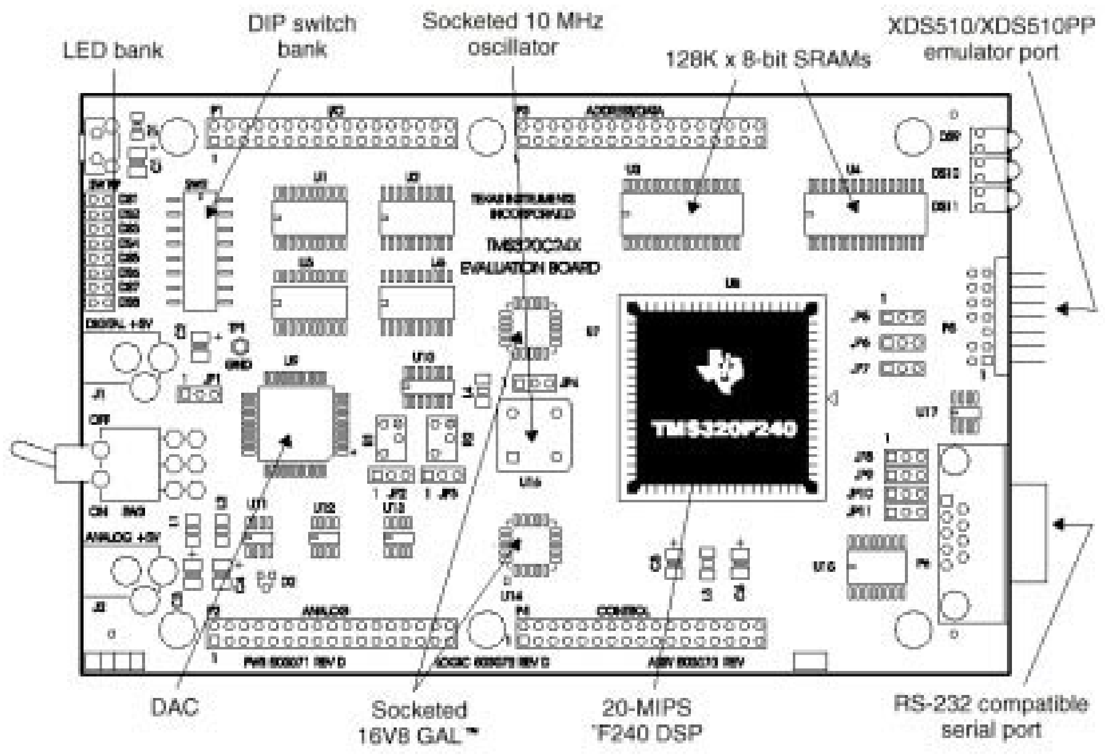
| Peripheral | Used in Application |
|---|---|
| Event manager: | |
| 12 compare/pulse-width modulation (PWM) channels (9 independent) | - |
| Three 16-bit general-purpose timers with six modes, including continuous up counting and continuous up/down counting | 1 timer |
| Three 16-bit full compare units with dead band capability | - |
| Three 16-bit simple compare units | 1 PWM output |
| Four capture units, two of which have quadrature encoder-pulse interface capability | - |
| Dual 10-bit analog-to-digital converter | 1 or 2 ADC, depending on number of channels |
| 28 individually programmable, multiplexed I/O pins | - |
| Phase-locked loop (PLL)-based clock module | x |
| Watchdog timer module with real-time interrupt | x |
| Serial communication interface (SCI) | - |
| Serial peripheral interface (SPI) | - |

| | | |
|---|---|---|
| *Legend:* | - | *not used* |
| | *x* | *not primarily, depends on application* |

## 3.2 TMS320C24x Evaluation Module

The TMS320C24x evaluation module (’C24x EVM) is a digital signal processor (DSP) development package that makes it possible to evaluate the ’C24x family of DSP controllers. It consist of a standalone evaluation board that contains a TMS320C240 device to explore the architecture and operation of the ’C24x CPU and its peripherals.

It includes 128K words of external on-board SRAM, a 4-channel 12-bit digital-to-analog converter (DAC), an RS-232 compatible serial port, a bank of eight I/O memory-mapped DIP switches and LEDs.


LED bank
DIP switch bank
Socketed 10 MHz oscillator
128K x 8-bit SRAMs
XDS510/XDS510PP emulator port
DAC
Socketed 16V8 GAL™
20-MIPS 'F240 DSP
RS-232 compatible serial port
TMS320F240
TMS320C24X EVALUATION BOARD

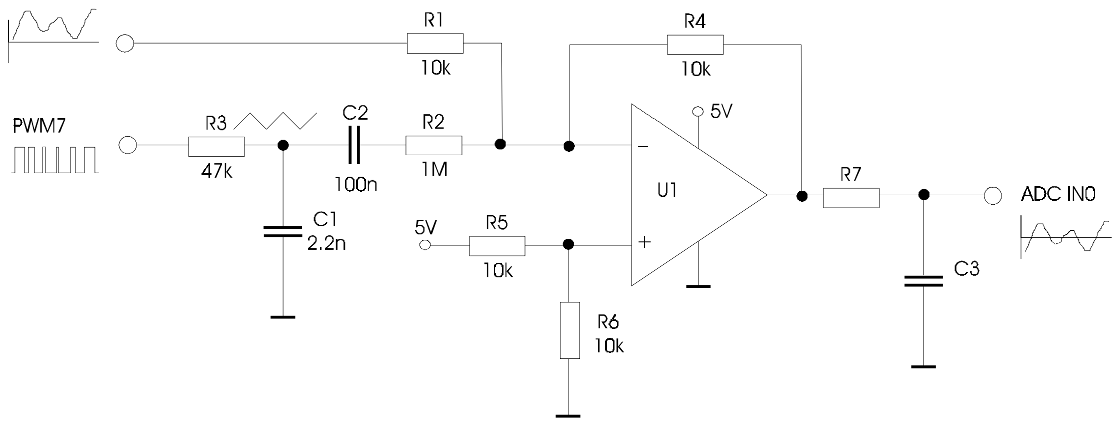
The evaluation board can be controlled and debugged through the board’s JTAG Interface using any Texas Instruments XDS510 emulator.

Four 34-pin headers allow the extension of the EVM board with user hardware. The oversampling daughter board accesses the PWM output signals of the simple compare unit and the AD-Converter’s input and reference signals.

## 3.3 EVM Daughter Board

As shown above, the TMS320C240 includes all peripherals to implement an oversampling subsystem, apart from the analog adder, which combines the input signal with the additive signal derived from the PWM output. Since the EVM board provides most of the C240’s signals over the four extension headers, a daughter board can easily be connected to the C240.

The PWM output signal operates between 0 and 100% duty-cycle. The integrator, consisting of R3 and C1, generates a triangular signal between 0 and 5V which is added to the input signal with a 1:1000 factor derived by R2 and R4. The attenuation of the input signal is 1 (R1/R4). The output signal is fed to any ADC input.



## 3.4 Oversampling Code

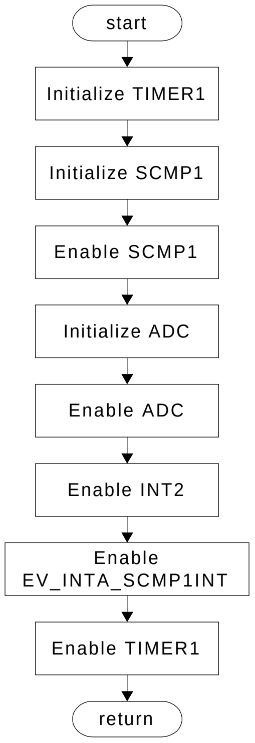
The DSP software can be divided into five major blocks:

- Peripheral Initialization
- Triangular Signal Generation
- Data Acquisition
- Digital Filter
- Decimation
- Interrupt Service Routine

All the routines described here are implemented in C in order to allow easily adaptation to any application need. However, it is obvious that the highest performance can be achieved only when implementing all modules in assembler language, especially the filter and decimation routines.

The source code can be found in appendix A.

### 3.4.1 Peripheral Initialization



Timer 1 is used both by the AD Converter to trigger a new conversion and the Simple Compare Unit's PWM output. The timer is configured for continuous up count with the period register programmed at the rate of the desired AD conversion rate.

Then the simple compare unit 1 is programmed to use timer 1 with the output signal as active high. Since the output pin of the simple compare unit 1 (PWM7/CMP7) is shared with the IOPB0 function of the Digital I/O module, this pin has to be selected for PWM7/CMP7 mode. As a next step the simple compare unit is enabled.

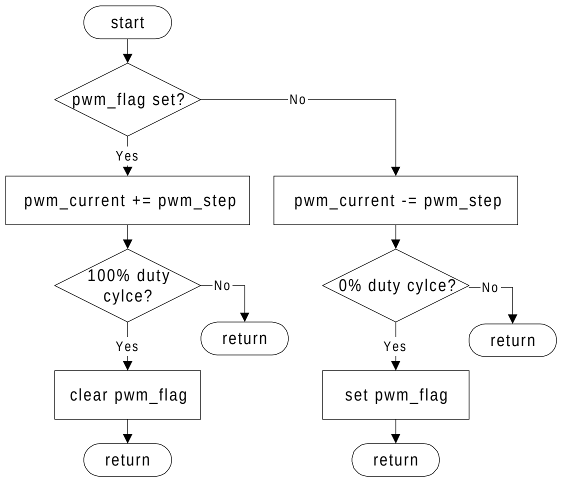
The AD Converter module is initialized to trigger conversions on period match events of timer 1, which gives a continuous sample rate equivalent to the timer's period register value. Again, since some of the ADC channels are shared with the Digital I/O module, it may be necessary to set up certain pins for ADC function. By setting up the ADC Control Registers the ADC is enabled.

In order to generate interrupts constantly from timer 1, CPU interrupt INT2 and the peripheral level interrupt from timer 1 (EV_INTA_T1PINT) are enabled. An interrupt service routine responding on INT2 interrupts will be installed to drive further steps of the signal generation, data acquisition and calculations.

As a last step of the initialization process timer 1 will be started by writing to the T1CON register.

### 3.4.2 Triangular Signal Generation

As described above a triangular signal is the most effective additive signal using oversampling. The PWM output and an analog low pass filter can be used to generate this signal.



The PWM duty cycle should range between 0 and 100% in order to achieve the highest possible resolution. The duty cycle is programmed by setting up the simple compare register (SCMPR1).

In order to get a triangular signal the value in SCMPR1 needs to be incremented/decremented by a certain step value determined by the timer period register ( value in T1PER) and the oversampling factor. The equation for the step value is as follows:

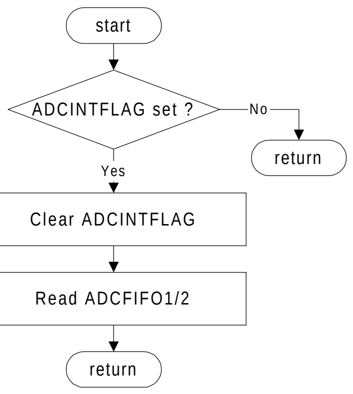
$$pwm\_step = 2 \cdot \frac{pwm\_period}{oversample\_factor}$$

A flag indicates the slope of the triangular signal. It is used to determine whether the value in the compare register should be incremented or decremented by the pwm_step value. pwm_current is the global variable of the value in the compare register.

### 3.4.3 Data Acquisition

The timer ISR calls a routine to get the sampled values from the AD converter. Before reading the values from the ADC's FIFO it checks the corresponding interrupt flag bit in the ADCCTRL1 register in order to check if the previous conversion has been completed.

The read value from the ADC is used as an input for the digital low-pass.



### 3.4.4 Digital Filter

#### 3.4.4.1 Moving Average Filter

Simulation has shown that a moving average filter is the best choice when using a triangular signal as the additive signal. A moving average filter in FIR structure is the best compromise between code complexity and performance in terms of frequency response, phase response and group delay.

The z-transformation for the moving average filter can be expressed with the following equation.

$$H(z) = \frac{1}{N}\sum_{n=0}^{N-1} z^{-n}$$

Processing every sample, the equation for calculating the output sample $y_i$ from the input sample $x_i$ results in the following equation.

$$y_i = y_{i-1} + \frac{x_i - x_{i-N}}{N}$$

This implementation requires a circular buffer for the input array x. The flexible indirect addressing modes of the TMS320C2xx core enable easy implementation of the moving average filter.

The input samples $x_i$ are 10 bit unsigned values representing the values 0 to 1023. The TMS320C24x has a 32 bit wide ALU and can perform arithmetic operations with 16 bit operands in a single cycle. In order to avoid divisions the oversampling factor N is selected as a number representing a power of two, e.g. 32, 64, 128.

In order to achieve an effective resolution of up to 15 bits the filter can be calculated in single precision (16 bit). In such an implementation the 10-bit input values x are shifted 6 bits to the left in order to use the full 16-bit width of the DSP.

Double precision calculation is required once the desired resolution is 16bits and higher. In such cases the filter should be processed in 32 bit data width.

#### 3.4.4.2 FIR Low-pass

When using white noise as the additive signal - or relying on the ADC's noise component - simulation showed that FIRs filter provide the most effective type of filtering. They have a linear phase response and are more stable than IIR filters.

Digital Filter Design Packages provide sufficient help in designing, testing and using FIR low-pass filters. Therefore the FIR code implementation is not covered in this document.

### *3.4.5 Decimation*

The decimation process reduces the final data rate by a factor of k. Typically, the decimation process is combined with the digital filter. If the filter is implemented in an FIR structure there is no need to calculate output samples that are ignored anyway after the decimation, which means that only every k-th output sample needs to be calculated.

### *3.4.6 Interrupt Service Routine*

The interrupt service routine is installed on the CPU's INT2 interrupt. The INT2 is bound to the timer 1 period match event. Since the timer 1 period interrupt is a member of the group A, the interrupt service routine has to read the content of the IVRA register in order to determine the source for the interrupt. If using only one interrupt within the group A there is no need to determine the vector ID and perform different actions based on the interrupt source. However the IVRA register should always be read.

The timer interrupt service routine calls the Triangular_Wave() function which reprograms the PWM duty cycle.

The ADC_Read() function gets the value of the last conversion, while the current conversion is just taking place, since it is started on timer period match - the same source as the interrupt itself.

The ADC value is passed to the digital filter which, in turn, calls the decimation routine.

## 4. Conclusion

Oversampling is a method of achieving higher effective resolutions while lowering the constraints of the analog anti-aliasing filter by implementing the filtering and decimation process digital. The TMS320C240 provides the computing power and the peripheral mix to implement an oversample subsystem with a resolution of up to 15 bits with the integrated 10 bit ADC.

There are different implementation approaches, each with advantages and disadvantages. Depending on the input signal spectrum, the sample rate and the desired resolution, the user can select any of the different methods.

## References

1. TMS320C24x DSP Controller - CPU, System and Instruction Set - Reference Set. Texas Instruments. 1997.
2. TMS320C24x DSP Controller - Peripheral Library and Specific Devices - Reference Set. Texas Instruments. 1997.
3. TMS320C1x/C2x/C2xx/C5x - Assembly Language Tools - User's Guide. Texas Instruments. 1995.
4. TMS320C2x/C2xx/C5x - Optimizing C Compiler - User's Guide. Texas Instruments. 1995.
5. Linear Design Seminar 1995, Texas Instruments
6. A simple approach to digital signal processing, Craig Marven & Gillian Ewers. TI Mentors. Texas Instruments. 1993.
7. Halbleiter-Schaltungstechnik, Ulrich Tietze & Christoph Schenk, Springer Verlag. 1983.
8. Digital Audio Signal Processing, Udo Zoelzer. John Wiley & Sons. 1995.

# Appendix A: TMS320C24x Code for Oversampling

## MAIN.C

```
/*-----------------------------------------------------------------------------------------

       filename:  main.c

         project:  oversampling on C24x

     description:  Main Function

-----------------------------------------------------------------------------------------*/

#include "osample.h"
#include "vectors.h"
#include "initfunc.h"
#include "filter.h"
#include "debug.h"



/* ---------------------------------------------------------------------------

    void init_hardware()

---------------------------------------------------------------------------*/

void init_hardware()
{
   Clock_Init();
}


/* ---------------------------------------------------------------------------

    void init_oversample()


    Performs all initialization steps for the oversampling task.

---------------------------------------------------------------------------*/

void init_oversample()
{
    int period;

    Int_DisableAll();                    /* disable all interrupts */

    period = Init_TriangularWave(1000);      /* 100us sample time   */

    Moving_Avg_Init();

    IO_InitAsOutputIOPC7();

    /* timer #1, pwm_period no prescaler */
    Timer_Init(GP_TIMER1, period, TIMER_PRESCALER1);

    /* spwm #1, active low on GP timer #1 */
    SPWM_Init( 1, SPWM_TYPE_ACTIVE_LOW, GP_TIMER1);

    /* ADC channel #0 */
    ADC_Init(0, ADC_TRIGGER_GP_TIMER1);
    ADC_Enable(0, TRUE);

    Int_EnablePeripheralA( EV_INTA_T1PINT, TRUE);
    Int_Enable( INT2, TRUE);

    Timer_Enable( GP_TIMER1, TRUE);

}
```

```
/* -----------------------------------------------------------------------------

    void main()

    Sets up the hardware and waits...
-----------------------------------------------------------------------------*/

void main()
{
   init_hardware();
   init_oversample();
   for (;;)
   { }
}
```

## ISR_C.C

```
/*-------------------------------------------------------------------------------------

        filename:  isr_c.c

         project:  oversampling on C24x

     description:  Interrupt service routines

-------------------------------------------------------------------------------------*/

#include "filter.h"
#include "initfunc.h"


/*-------------------------------------------------------------------------------------
    Interrupt:   RTI (INT1)
    Function:    Main Interrupt Routine Handler
-------------------------------------------------------------------------------------*/
void interrupt c_int1(void)
{
   asm(" .copy \"c240app.h\"");
   asm(" ldp       #IVRB/128");
   asm(" lacc      SYSIVR");
}


/*-------------------------------------------------------------------------------------
     Interrupt:   (INT2)
     Function:    Timer 1 Interrupt Routine Handler
-------------------------------------------------------------------------------------*/
void interrupt c_int2(void)
{
    asm(" .copy \"c240app.h\"");
    asm(" ldp       #IVRA/128");
    asm(" lacc      IVRA");

    TriangularWave();
    adc_value = ADC1_read();
    Moving_Avg_Filter();
    return;
}

/*-------------------------------------------------------------------------------------
    Interrupt:   Timer 3/c_int3
    Function:    Main Interrupt Routine Handler
-------------------------------------------------------------------------------------*/
void interrupt c_int3(void)
{
   asm(" .copy \"c240app.h\"");
   asm(" ldp       #IVRB/128");
   asm(" lacc      IVRB");

}
```

## INITFUNC.C

```
/*-----------------------------------------------------------------------------------------

        filename:  initfunc.c

         project:  oversampling on C24x

     description:  Functions for initializing the C24x peripherals and EVM and support
                   functions for reading ADC and programming simple compare PWM outputs.

---------------------------------------------------------------------------------------*/


#include "osample.h"
#include "c240.h"

#include "initfunc.h"


unsigned int pwm_step;    /*  pwm_step:
                                  The step size for each incremental stage.
                                  Calculated as ( pwm_period / pwm_oversample)
                          */


unsigned int pwm_half;    /*  pwm_half:
                                  The maximum 'amplitude' of the PWM signal
                                  Calculated as (pwm_step*pwm_oversample) - 1
                                  If pwm_period is a magnitude of pwm_oversample
                                  this value is pwm_period-1
                          */


unsigned int pwm_current; /*  pwm_current:
                                  Current 'value' of the pwm signal. Incremented
                                  by pwm_step on each compare interrupt.
                          */


unsigned char pwm_ramp;       /*  pwm_ramp:
                                  Indicator if the pwm signal is 'counting' up-
                                  or downwards.
                                  Will be changed on pwm_current=0 or
                                  pwm_current=pwm_half
                          */


unsigned int pwm_oversample;  /*  pwm_oversample:
                                      Oversample factor.
                              */
unsigned int pwm_sampletime;  /*  pwm_sampletime:
                                      PWM signal period in 100ns resolution.
                                      Multiply it with 2 and you have the number of timer ticks
                              */

/* ---------------------------------------------------------------------------

  void Init_TriangularWave(int sampletime)

    Initializes the gloabl variables used in TriangularWave().

    The parameter 'sampletime' is the sample time in 0.1 us (= 2 timer ticks @
    20 MIPS)

-----------------------------------------------------------------------------*/

int Init_TriangularWave( int sampletime)
{
    int period;

    pwm_current = 0;                  /* start at zero */
    pwm_ramp = 1;                     /* go up first */

    pwm_oversample = N_OVERSAMPLE;       /* 32 times oversample, hardcoded here   */

    period = sampletime << 1;     /* in timer ticks  */

    pwm_step = period / pwm_oversample;
    pwm_half = ((pwm_step*pwm_oversample)>>1) - 1;

    return period;
}



/* ----------------------------------------------------------------------------

  void TriangularWave()

  Generates Triangular signal on SPWM1. Is called within the SPWM1 ISR

----------------------------------------------------------------------------*/


void TriangularWave()
{
    if (pwm_ramp)  /* ramp up */
    {
        pwm_current+= pwm_step;
        if (pwm_current >= pwm_half)
            pwm_ramp = 0;      /* next time down */
    }
    else
    {
        pwm_current-= pwm_step;
        if (pwm_current <= 0)
            pwm_ramp = 1;      /* next time up */
    }
    SPWM_1_Send(pwm_current);
}



/* ****************************************************************************
         TIMER
****************************************************************************/



/* ----------------------------------------------------------------------------

    void Timer_Init( int timer, unsigned int period, int timer_prescaler)

    Initializes GP timer #timer with 'period' as the value written into
    the period regster and 'timer_prescaler' indicating the prescaler
    value.

      Arguments:    timer: a value of 1, 2, 3 indicating which timer should be
                          configured.

                   period: the value for the period register of GP timer 1

           timer_prescaler: a value of TIMER_PRESCALER1-TIMER_PRESCALER128
                          describing the timer prescaler for GP timer 1

        Note:   Timer and Timer Compare are disabled.

----------------------------------------------------------------------------*/

void Timer_Init( int timer, unsigned int period, int timer_prescaler)
{
    unsigned int timerreg;

    timerreg = 0x7404 + ((timer-1)<<2);      /* get address of TxCON */

                              /* disable counter, clear TENABLE in TxCON */
    *(unsigned *)(timerreg) = TIMER_CONTINOUS_UP | timer_prescaler;
    *(unsigned *)(timerreg-1) = period;      /* write period to TxPR register */
}


/* ---------------------------------------------------------------------------

    void Timer_Enable( int timer, int enable)


    Enables or disables the GP timer specified in the timer parameter.


        Arguments: timer:  a value of 1, 2 or 3 indicating which GP timer
                           should enable or disabled
                   enable: boolsche variable indicating if the function
                           should start or stop the timer

        Note:   Function does not initialize the GP timer, nor does it modify
                other parameters of the timer, but the TENABLE bit.

---------------------------------------------------------------------------*/

void Timer_Enable( int timer, int enable)
{
    int timerreg;

    timerreg = 0x7404 + ((timer-1)<<2);      /* get address of TxCON */
                                         /* disable that counter, TENABLE in TxCON */
    if (enable)
        *(unsigned *)(timerreg) = *(unsigned *)(timerreg) | TIMER_ENABLE;
    else
        *(unsigned *)(timerreg) = *(unsigned *)(timerreg) & 0xffbf;
}



/* ***************************************************************************
         SPWM
****************************************************************************/


/* ---------------------------------------------------------------------------

    void SPWM_Init( int spwm, int spwm_type, int timer)

    Initializes the simple compare unit specified in the 'spwm' parameter to
    use the GP timer as specified in 'timer' parameter.
    The simple compare output pin are configured as described in the
    'spwm_type' parameter.

        Arguments: spwm:   a value of 1, 2 or 3 indicating which simple
                           compare unit should be configured

                spwm_type: a value of SPWM_TYPE_FORCED_LOW,
                           SPWM_TYPE_ACTIVE_LOW, SPWM_TYPE_ACTIVE_HIGH or
                           SPWM_TYPE_FORCED_HIGH describing the desired output
                           of the simple compare unit outputs

        Note:   Function does initialize the Simple Compare Unit #spwm.
                It does not initialize or enable GP Timer1 #timer.

---------------------------------------------------------------------------*/

void SPWM_Init( int spwm, int spwm_type, int timer)
{
    COMCON &= 0x7fff;                    /* disable compare unit */

    if (spwm == 1)                       /* construct SACTR word */
    {                                    /* and set the appropriate bits */
        SACTR &= 0xfffc;
        SACTR |= spwm_type;
        OCRA |= 0x0100;                      /* enable PWM7/CMP7 of shared IO */
    }
    if (spwm == 2)
    {
        SACTR &= 0xfff3;
        SACTR |= (spwm_type << 2);
        OCRA |= 0x0200;                    /* enable PWM8/CMP8 of shared IO */
    }
    if (spwm == 3)
    {
        SACTR &= 0xffcf;
        SACTR |= (spwm_type << 4);
        OCRA |= 0x0400;                    /* enable PWM9/CMP9 of shared IO */
    }

    if (timer==1)
        COMCON &= 0xff7f;
    else
        COMCON |= 0x0080;

    COMCON |= 0x0100;                    /* enable simple compare unit */

    COMCON |= 0x8000;                    /* enable COMPARE unit */
}


/* ----------------------------------------------------------------------------

    void SPWM_1_Send( unsigned int value)


    Sends a value to Simple Compare Unit 1


        Arguments: value:  the value that should be written into the SCMPR1
                           register

    Note: This function is for the sake of easy readability.
--------------------------------------------------------------------------------*/

inline void SPWM_1_Send( unsigned int value)
{
    SCMPR1 = value;
}


/* ----------------------------------------------------------------------------

    void SPWM_2_Send( unsigned int value)


    Sends a value to Simple Compare Unit 2. dito as SPWM_1_Send()

--------------------------------------------------------------------------------*/

inline void SPWM_2_Send( unsigned int value)
{
    SCMPR2 = value;
}


/* ----------------------------------------------------------------------------

    void SPWM_3_Send( unsigned int value)


    Sends a value to Simple Compare Unit 3. dito as SPWM_1_Send()

--------------------------------------------------------------------------------*/

inline void SPWM_3_Send( unsigned int value)
{
    SCMPR3 = value;
}



/* ****************************************************************************
         ADC
*****************************************************************************/


/* ----------------------------------------------------------------------------
```

```
    void ADC_Init( int channel, int mode)


    Initializes the ADC Module, by setting the required registers to sample
    a certain channel and to use a GP timer to trigger the sample event.

    If required (channel 0 or 1) the function activates the shared GPIOx/ADCINx
    signal lines to use the ADCIN0 and ADCIN1 respectively.

    The function expects SYSCLK to run at 10MHz.

    The function will initialize the Event Module to generate the trigger impulse
    for ADC1 by calling the InitADC_TriggerMode( int timer) function.


        Arguments: channel:   ADC is programmed to use this channel as input
                              channel valid range is 0..7 for ADC1 and
                              channel 8..15 for ADC2.

                   mode:      describes the mode the ADC uses to trigger a new
                              conversion.
                              valid modes are:
                                     ADC_TRIGGER_CONTINOUS
                                     ADC_TRIGGER_GP_TIMER1
                                     ADC_TRIGGER_GP_TIMER2
                                     ADC_TRIGGER_GP_TIMER3
                                     ADC_TRIGGER_EXTERNAL

----------------------------------------------------------------------------*/

void ADC_Init( int channel, int mode)
{
    switch (channel)
    {
        case 0: OCRA |= 0x0001;       /* activate muxed ADCIN0 */
                break;
        case 1: OCRA |= 0x0002;       /* activate muxed ADCIN1 */
                break;
        case 9: OCRA |= 0x0004;       /* activate muxed ADCIN9 */
                break;
        case 8: OCRA |= 0x0008;       /* activate muxed ADCIN8 */
                break;
    }

    if (mode == ADC_TRIGGER_CONTINOUS)
    {
        ADCTRL1 = 0x0400;   /* continous mode */
        ADCTRL2 = 0x0003;  /* Prescaler = 10 */
        return;
    }

    if (mode == ADC_TRIGGER_EXTERNAL)
    {
        ADCTRL1 = 0x0000;
         ADCTRL2 = 0x0203; /* Prescaler = 10 */
         OCRB |= 0x0001;     /* activate pin as ADCSOC */
        return;
    }

    /* prepare ADC Control Registers for Event Manager triggered conversion */
    ADCTRL1 = 0x0000;      /* non continous mode */
    ADCTRL2 = 0x0403;      /* Prescaler=10, event manager triggers conversion */

    /* configure timer to generate interrupt on compare */
    ADC_InitTriggerMode(mode);
    return;
}



/* ---------------------------------------------------------------------------

    void ADC_InitTriggerMode( int mode)


    Initializes the Event Manager Module, by setting the required registers
    to derive the ADC's trigger events from the GP timer compare interrupt.
```

```
       Arguments: mode:  a value of ADC_TRIGGER_GP_TIMER1/2/3 indicates
                         which GP timer should be used to trigger ADC
                         conversion


               Following modes are already handled in ADC_1_Init:

               ADC_TRIGGER_CONTINOUS

                         a value of ADC_TRIGGER_CONTINOUS indicates that
                         the ADC should run continously

               ADC_TRIGGER_EXTERNAL

                         a value of ADC_TRIGGER_EXTERNAL indicates that
                         the ADC should be triggered on external events

       Note:   Function does not initialize the GP timer, this should be
               performed in advance. However, the timer is stopped
               temporarily, but restored to the previous value on exit.

----------------------------------------------------------------------------*/

void ADC_InitTriggerMode( int mode)
{
   unsigned int txcon;
   if (mode == ADC_TRIGGER_GP_TIMER1)
   {
       txcon = T1CON;         /* save T1CON */
       T1CON &= 0xffbf;       /* clear TENABLE in T1CON = disable timer */
       GPTCON &= 0xfe7f;      /* clear bit 7-8 in GPTCON */
       GPTCON |= 0x0100;      /* set bit 8 = 1 (=period i-flag)*/
       T1CON = txcon;         /* restore T1CON again */
       return;
   }
   if (mode == ADC_TRIGGER_GP_TIMER2)
   {
       txcon = T2CON;         /* save T2CON */
       T2CON &= 0xffbf;       /* clear TENABLE in T2CON = disable timer */
       GPTCON &= 0xf9ff;      /* clear bit 9-10 in GPTCON */
       GPTCON |= 0x0400;      /* set bit 10 = 1 (=period i-flag)*/
       T2CON = txcon;         /* restore T2CON again */
       return;
   }
   if (mode == ADC_TRIGGER_GP_TIMER2)
   {
       txcon = T3CON;         /* save T3CON */
       T3CON &= 0xffbf;       /* clear TENABLE in T3CON = disable timer */
       GPTCON &= 0xe7ff;      /* clear bit 11-12 in GPTCON */
       GPTCON |= 0x1000;      /* set bit 12 = 1 (=period i-flag)*/
       T3CON = txcon;         /* restore T3CON again */
       return;
   }
}

/* --------------------------------------------------------------------------

   void ADC_Enable( int channel, int enable)


   Enables or disables the ADC1 and ADC2 respectively, depending on the
   channels passed in the 'channel' argument.


       Arguments: channel:   a value between 0 and 15 specifing the
                             channels 1-16
                             0-7 accesses ADC1
                             8-15 accesses ADC2

                  enable:    TRUE or FALSE to enable or disable the
                             appropriate ADC

----------------------------------------------------------------------------*/

void ADC_Enable( int channel, int enable)
{
   if (channel < 8)          /* ADC1 */
   {
       ADCTRL1 &= 0xfff1;            /* mask off bits */
       ADCTRL1 |= (channel<<1);      /* select ADC channel */
       if (enable)
           ADCTRL1 |= 0x0800;        /* enables ADC1 */
       else
           ADCTRL1 &= 0xf7ff;        /* disable ADC1 */
   }
   else
   {
       ADCTRL1 &= 0xff8f;            /* mask off bits */
       ADCTRL1 |= ((channel-8)<<4);  /* select ADC channel */
       if (enable)
           ADCTRL1 |= 0x1000;        /* enables ADC2 */
       else
           ADCTRL1 &= 0xefff;        /* disable ADC2 */
   }
}


/* ----------------------------------------------------------------------------

   int ADC1_read()

   Reads out the FIFO of the ADC1 by checking the interrupt flag.
   If ADC is not valid returns 0xffff.

----------------------------------------------------------------------------*/


int ADC1_read()
{
   unsigned int temp = 0xffff;

   if(ADCTRL1 & 0x0100)
   {
       ADCTRL1 |= 0x0100;
       temp = ADCFIFO1;
   }
   return(temp);
}


/* ----------------------------------------------------------------------------

   int ADC2_read()

   Reads out the FIFO of the ADC2 by checking the interrupt flag.
   If ADC is not valid returns 0xffff.

----------------------------------------------------------------------------*/

int ADC2_read()
{
  volatile unsigned int temp=0xffff;

  if( ADCTRL2 & 0x0100 )
  {
   ADCTRL2 |= 0x0100;
   temp = ADCFIFO2;
  }
  return(temp);
}


/* ****************************************************************************
         Interrupt
*****************************************************************************/

/* ----------------------------------------------------------------------------

   void Int_DisableAll()

   Disables all interrupts and clears all flag registers.

----------------------------------------------------------------------------*/
```

```
void Int_DisableAll()
{
    unsigned int tmp = 0xffff;

    asm(" .copy \"c240app.h\"");
    asm(" setc    INTM"); /* disable all maskable interrupts */

    IFR = tmp;
    EVIFRA = tmp;
    EVIFRB = tmp;
    EVIFRC = tmp;

    tmp = 0x0;
    IMR = tmp;
    EVIMRA = tmp;
    EVIMRB = tmp;
    EVIMRC = tmp;

    return;
}

/* ---------------------------------------------------------------------------

    void Int_Enable( int int_nr, int enable)

    Enables or disables CPU Level interrupts by setting or clearing the
    corresponding INTx bits in the IMR register.

        Parameters:int_nr:    a value of INT1 - INT6 (see #define)

                   enable:    variable to indicate wether the interrupt should
                              be enabled or disabled

-------------------------------------------------------------------------------*/

void Int_Enable( int int_nr, int enable)
{
    asm(" .copy \"c240app.h\"");
    asm(" setc    INTM"); /* disable all maskable interrupts */

    if (enable)
        IMR |= int_nr;
    else
        IMR &= ~int_nr;

    IFR |= int_nr;

    asm(" .copy \"c240app.h\"");
    asm(" clrc    INTM"); /* enable all maskable interrupts */
    return;
}

/* ---------------------------------------------------------------------------

    void Int_EnablePeripheralA(int per_nr, int enable)

    Enables or disables Peripheral Level interrupts by setting or clearing the
    corresponding EVIMRx bits in the EVIMRx register.

        Parameters:int_nr:    any of the EV_INTA_xxxxx values (see #define)

                   enable:    variable to indicate wether the interrupt should
                              be enabled or disabled

        Note: Function works on EVIMRA register only. If you need access to
              EVIMRB and EVIMRC call Int_EnablePeripheralB() and
              Int_EnablePeripheralC() respectively.

              Be sure you call Int_Enable() for the corresponding interrupt
              after the Int_EnablePeripheralX() functions when enabling
              interrupts.
              When disabling interrupts call Int_Enable() before any of the
              corresponding Int_EnablePeripheralX() functions.

-------------------------------------------------------------------------------*/


void Int_EnablePeripheralA(int per_nr, int enable)
```

```
{
    if (enable)
        EVIMRA |= per_nr;
    else
        EVIMRA &= ~per_nr;

    EVIFRA |= per_nr;
    return;
}


/* ----------------------------------------------------------------------------

    void Int_EnablePeripheralB(int per_nr, int enable)

    Same as Int_EnablePeripheralA(), but works on EVIMRB and EVIFRB registers.

----------------------------------------------------------------------------*/

void Int_EnablePeripheralB(int per_nr, int enable)
{
    if (enable)
        EVIMRB |= per_nr;
    else
        EVIMRB &= ~per_nr;

    EVIFRB |= per_nr;
    return;
}

/* ----------------------------------------------------------------------------

    void Int_EnablePeripheralC(int per_nr, int enable)

    Same as Int_EnablePeripheralA(), but works on EVIMRC and EVIFRC registers.

----------------------------------------------------------------------------*/

void Int_EnablePeripheralC(int per_nr, int enable)
{
    if (enable)
        EVIMRC |= per_nr;
    else
        EVIMRC &= ~per_nr;

    EVIFRC |= per_nr;
    return;
}



/* ----------------------------------------------------------------------------

    void Clock_Init()

    Initializes the clock and PLL module for operation with 10 MHz external
    clock

----------------------------------------------------------------------------*/


void Clock_Init()
{
    /* 10 MHz external clock */

    CKCR1 = 0x00BB; /* divide CLK(PLL_IN) by 2, PLL Multiply = 4,
                       ACLK Divider = 10 */
    CKCR0 = 0x00C3;    /* System Clock = CPUCLK/2; ACLK enabled;PLL enabled */
    SYSCR = 0x40C0;

    wsgr = 0x04;
}
```

## FILTER.C

```
/*------------------------------------------------------------------------------------------

        filename:  filter.c

         project:  oversampling on C24x

     description:  Digital Filters

------------------------------------------------------------------------------------------*/

#include "osample.h"

#include "filter.h"
#include "debug.h"



unsigned int adc_array[MAX_OVERSAMPLE];
unsigned int avg_value;
unsigned int avg_count;
unsigned long avg_result = 0;


unsigned int adc_value;


/* ---------------------------------------------------------------------------

  void Moving_Avg_Init()

  Initializes the variables and circular buffer used by the moving average
  filter in the Moving_Avg_Filter() function.

-----------------------------------------------------------------------------*/

void Moving_Avg_Init()
{
    int i;
    for (i=0; i<MAX_OVERSAMPLE;i++)
        adc_array[i] = 0;
    adc_value = 0;

    avg_result = 0;
    avg_value = 0;
    avg_count = 0;

#ifdef _DEBUG
    DbgReset(0);
#endif
}




/* ---------------------------------------------------------------------------

  void Moving_Avg_Filter()

  Performs the moving average filter and decimation.

  Expects the adc_value to contain the latest sample from the ADC (input x)
  and updates every n-th run the output variable avg_result (output y).

-----------------------------------------------------------------------------*/

void Moving_Avg_Filter()
{
    unsigned int adc_temp;

    if (avg_count >= N_OVERSAMPLE)           /* reset index into circular buffer */
    {
        avg_result = avg_value;       /* output every N_OVERSAMPLE-th result */
        avg_count = 0;
    }
#ifdef _DEBUG
    if (adc_value == 0xFFFF)
    {
        error_count++;
        return;
    }
#endif

                /* adc_value and adc_value[i] are left aligned */
    avg_value = avg_value + (((long)adc_value - (long)adc_array[avg_count]) >> N_POWER);

    adc_array[avg_count] = adc_value;    /* put adc_value into circular buffer */
    avg_count++;


#ifdef _DEBUG
    dbg_adc_value[dbg_count] = adc_value;
    dbg_median_result[dbg_count] = avg_result;
    dbg_count++;


    if (dbg_count > DBG_SIZE)
        DbgReset(dbg_count);
#endif

    return;
}
```

## DEBUG.C

```
/*-----------------------------------------------------------------------------------------

        filename:  debug.c

         project:  oversampling on C24x

     description:  Some help functions to trace conditions, values and states.

-----------------------------------------------------------------------------------------*/

#include "debug.h"

unsigned int error_count = 0;
unsigned int dbg_count = 0;
unsigned int dbg_reset_count = 0;


                        /* filled in Moving_Avg_Filter() */
unsigned int dbg_adc_value[DBG_SIZE];        /* value of adc measurement */
unsigned int dbg_median_result[DBG_SIZE];       /* value of median */


void DbgReset(unsigned int dummy)
{
    dbg_count = 0;
    if (dbg_reset_count > 30)
        dbg_reset_count= 0;
    dbg_reset_count++;
}
```

## SUPPORT.C

```
/*-----------------------------------------------------------------------------------------

        filename:  support.c

         project:  oversampling on C24x

     description:  Some support functions

-----------------------------------------------------------------------------------------*/
```

```
#include "c240.h"
#include "support.h"


static int io_mode = 0;

void IO_InitAsOutputIOPC7()
{
   OCRB = OCRB & 0x007F;
   PCDATDIR = PCDATDIR | 0x8000;
   PCDATDIR = PCDATDIR & 0xFF7F;
   io_mode = 0;
}


void IO_TriggerIOPC7()
{
   if (io_mode == 0)
   {
      PCDATDIR = PCDATDIR | 0x80;
      io_mode = 1;
      return;
   }
   PCDATDIR = PCDATDIR & 0xFF7F;
   io_mode = 0;
}


ioport unsigned port00;
ioport unsigned port01;
ioport unsigned port02;
ioport unsigned port03;
ioport unsigned port04;

/* ---------------------------------------------------------------------------

   void DAC_Write0( int value)

   Writes a value to the DAC0 register and updates the DAC.

-------------------------------------------------------------------------- */

void DAC_Write0( int value)
{
   port00 = value;
   port01 = 2 * value;
   port02 = 3 * value;
   port03 = 4 * value;
   port04 = value;       /* dummy write for updating */
   return;
}
```

## OSAMPLE.H

```
/*----------------------------------------------------------------------------------------

        filename:  osample.h

         project:  oversampling on C24x

     description:  Global definitions

---------------------------------------------------------------------------------------*/


#define _DEBUG     1

#define MAX_OVERSAMPLE      512

#define N_POWER                  5
#define N_OVERSAMPLE          32
```

## VECTORS.H

```
/*-----------------------------------------------------------------------------------

       filename:  vectors.h

        project:  oversampling on C24x

    description:  Include this file in the first C file

-----------------------------------------------------------------------------------*/


  asm("       .global _c_int0                   ");
  asm("       .global _c_int1                   ");
  asm("       .global _c_int2                   ");
  asm("       .global _c_int3                   ");

  asm("       .sect \"vectors\"                 ");
  asm("RESET:  b     _c_int0 ; external RESET   ");
  asm("INT1:   b     _c_int1                    ");
  asm("INT2:   b     _c_int2                    ");
  asm("INT3:   b     _c_int3                    ");
  asm("       .text                             ");
```

## INITFUNC.H

```
/*-----------------------------------------------------------------------------------

       filename:  initfunc.h

        project:  oversampling on C24x

    description:  Functions for initializing the C24x peripherals and EVM and support
                  functions for reading ADC and programming simple compare PWM outputs.

-----------------------------------------------------------------------------------*/


/* -------- VARIABLES ------------- */

extern unsigned int pwm_step;
extern unsigned int pwm_half;
extern unsigned int pwm_current;
extern unsigned char pwm_ramp;
extern unsigned int pwm_oversample;
extern unsigned int pwm_sampletime;



/* -------- FUNCTION PROTOTYPES ------------- */

void Timer_Init( int timer, unsigned int period, int timer_prescaler);
void Timer_Enable( int timer, int enable);

void SPWM_Init( int spwm, int spwm_type, int timer);
inline void SPWM_1_Send( unsigned int value);
inline void SPWM_2_Send( unsigned int value);
inline void SPWM_3_Send( unsigned int value);

void ADC_Init( int channel, int mode);
void ADC_InitTriggerMode( int mode);
void ADC_Enable( int channel, int enable);
int ADC1_read();
int ADC2_read();

void Clock_Init();

void Median_Init();
void Median_Reset();
void Median_Filter();
```

```
/* -------- DEFINES ------------- */


#define TRUE   1
#define FALSE  0

#define SPWM_TYPE_FORCED_LOW  0
#define SPWM_TYPE_ACTIVE_LOW  1
#define SPWM_TYPE_ACTIVE_HIGH 2
#define SPWM_TYPE_FORCED_HIGH 3

#define GP_TIMER1             0x0001
#define GP_TIMER2             0x0002
#define GP_TIMER3             0x0003

#define TIMER_CONTINOUS_UP    0x1000
#define TIMER_CONTINOUS_UPDOWN    0x2800

#define TIMER_COMPARE_ENABLE  0x0002
#define TIMER_ENABLE          0x0040

#define TIMER_PRESCALER1      0x0000
#define TIMER_PRESCALER2      0x0100
#define TIMER_PRESCALER4      0x0200
#define TIMER_PRESCALER8      0x0300
#define TIMER_PRESCALER16     0x0400
#define TIMER_PRESCALER32     0x0500
#define TIMER_PRESCALER64     0x0600
#define TIMER_PRESCALER128    0x0700


#define ADC_TRIGGER_CONTINOUS 0
#define ADC_TRIGGER_GP_TIMER1 1
#define ADC_TRIGGER_GP_TIMER2 2
#define ADC_TRIGGER_GP_TIMER3 3
#define ADC_TRIGGER_EXTERNAL  4


#define INT1              0x0001
#define INT2              0x0002
#define INT3              0x0004
#define INT4              0x0008
#define INT5              0x0010
#define INT6              0x0020

#define EV_INTA_PDPINT    0x0001
#define EV_INTA_CMP1INT       0x0002
#define EV_INTA_CMP2INT       0x0004
#define EV_INTA_CMP3INT       0x0008
#define EV_INTA_SCMP1INT  0x0010
#define EV_INTA_SCMP2INT  0x0020
#define EV_INTA_SCMP3INT  0x0040
#define EV_INTA_T1PINT    0x0080
#define EV_INTA_T2CINT    0x0100
#define EV_INTA_T1UFINT       0x0200
#define EV_INTA_T1OFINT       0x0400

#define EV_INTB_T2PINT    0x0001
#define EV_INTB_T2CINT    0x0002
#define EV_INTB_T2UFINT       0x0004
#define EV_INTB_T2OFINT       0x0008
#define EV_INTB_T3PINT    0x0010
#define EV_INTB_T3CINT    0x0020
#define EV_INTB_T3UFINT       0x0040
#define EV_INTB_T3OFINT       0x0080

#define EV_INTC_CAP1INT       0x0001
#define EV_INTC_CAP2INT       0x0002
#define EV_INTC_CAP3INT       0x0004
#define EV_INTC_CAP4INT       0x0008



#define IMR  *(unsigned int*) 0x0004
#define IFR  *(unsigned int*) 0x0006
```

```
ioport unsigned port0ffff;/* waitstate generator */
#define wsgr port0ffff
```

## FILTER.H

```
/*-----------------------------------------------------------------------------------------

        filename:  filter.h

         project:  oversampling on C24x

     description:  Digital Filters

-----------------------------------------------------------------------------------------*/


/* -------- VARIABLES ------------- */

extern unsigned int adc_value;

extern unsigned int avg_value;
extern unsigned int avg_count;
extern unsigned long avg_result;


/* -------- FUNCTION PROTOTYPES ------------- */

void Moving_Avg_Init();
void Moving_Avg_Reset();
void Moving_Avg_Filter();
```

## DEBUG.H

```
/*-----------------------------------------------------------------------------------------

        filename:  debug.h

         project:  oversampling on C24x

     description:  Some help functions to trace conditions, values and states.

-----------------------------------------------------------------------------------------*/

#define DBG_SIZE   6000

/* -------- VARIABLES ------------- */

extern unsigned int error_count;
extern unsigned int dbg_count;
extern unsigned int dbg_reset_count;

extern unsigned int dbg_adc_value[DBG_SIZE];        /* value of adc measurement */
extern unsigned int dbg_median_result[DBG_SIZE];    /* value of median */


/* -------- FUNCTION PROTOTYPES ------------- */

void DbgReset(unsigned int dummy);
```

## SUPPORT.H

```
/*-----------------------------------------------------------------------------------------
```

```
        filename:  support.h

         project:  oversampling on C24x

     description:  Some support functions

-------------------------------------------------------------------------------------*/

/* -------- FUNCTION PROTOTYPES ------------- */

void DAC_Write0( int value);


void IO_InitAsOutputIOPC7();
void IO_TriggerIOPC7();
```

## C240.H

```
#ifndef __C240_H_
#define __C240_H_


/*********************************************************************/
/* Watchodg and Real time Interrupt Control Registers               */
/*********************************************************************/
#define RTICNTR  *(volatile unsigned int*) 0x7021
#define WDCNTR   *(volatile unsigned int*) 0x7023
#define WDKEY    *(volatile unsigned int*) 0x7025
#define RTICR    *(volatile unsigned int*) 0x7027
#define WDCR     *(volatile unsigned int*) 0x7029

/*********************************************************************/
/* pll Clock Registers                                               */
/*********************************************************************/
#define CKCR0    *(volatile unsigned int*) 0x702B
#define CKCR1    *(volatile unsigned int*) 0x702D

#define OCRA     *(volatile unsigned int*) 0x7090
#define OCRB    *(volatile unsigned int*) 0x7092

/*********************************************************************/
/*     Definitions for SCI Module                                    */
/*********************************************************************/

                    /* SCI communications control register    */
#define SCICCR   *(volatile unsigned int*) 0x7050
                    /* SCI control register                   */
#define SCICTL1  *(volatile unsigned int*) 0x7051
                    /* Baud rate select MSB                   */
#define SCIHBAUD  *(volatile unsigned int*)0x7052
                    /* Baud rate select LSB                   */
#define SCILBAUD  *(volatile unsigned int*)0x7053
                    /* Transmitter int. control and status reg*/
#define SCICTL2   *(volatile unsigned int*)0x7054
                    /* Receiver int. control and status reg   */
#define SCIRXST   *(volatile unsigned int*)0x7055
                    /* Receiver data buffer                   */
#define SCIRXEMU  *(volatile unsigned int*)0x7056
                    /* Transmit data buffer                   */
#define SCIRXBUF  *(volatile unsigned int*)0x7057
                    /* Transmit data buffer                   */
#define SCITXBUF  *(volatile unsigned int*)0x7059
                    /* Port control register #2               */
#define SCIPC2    *(volatile unsigned int*)0x705E
                    /* Interrupt priority control register    */
#define SCIPRI    *(volatile unsigned int*)0x705F



/*********************************************************************/
/*     Definitions for ADC Module                                    */
/* structures are computed uncorrectly by the C Compiler v6.60 !    */
/* therefore following approach has to be used                      */
/*********************************************************************/
```

```
#define ADCTRL1   *(volatile unsigned int*) 0x7032   /* ADC Control register 1 */
#define ADCTRL2   *(volatile unsigned int*) 0x7034   /* ADC Control register 2 */
#define ADCFIFO1  *(volatile unsigned int*) 0x7036   /* ADC1  result FIFO */
#define ADCFIFO2  *(volatile unsigned int*) 0x7038   /* ADC2 result FIFO */


/*******************************************************************/
/*      Definitions for EV Module                                 */
/* structures are computed uncorrectly by the C Compiler v6.60 !    */
/*******************************************************************/
#define GPTCON   *(volatile unsigned int*) 0x7400
#define T1CNT    *(volatile unsigned int*) 0x7401
#define T1CMPR   *(volatile unsigned int*) 0x7402
#define T1PR     *(volatile unsigned int*) 0x7403
#define T1CON    *(volatile unsigned int*) 0x7404
#define T2CNT    *(volatile unsigned int*) 0x7405
#define T2CMPR   *(volatile unsigned int*) 0x7406
#define T2PR     *(volatile unsigned int*) 0x7407
#define T2CON    *(volatile unsigned int*) 0x7408
#define T3CNT    *(volatile unsigned int*) 0x7409
#define T3CMPR   *(volatile unsigned int*) 0x740A
#define T3PR     *(volatile unsigned int*) 0x740B
#define T3CON    *(volatile unsigned int*) 0x740C

#define COMCON   *(volatile unsigned int*) 0x7411
#define ACTR     *(volatile unsigned int*) 0x7413
#define SACTR    *(volatile unsigned int*) 0x7414
#define DBTCON   *(volatile unsigned int*) 0x7415
#define CMPR1    *(volatile unsigned int*) 0x7417
#define CMPR2    *(volatile unsigned int*) 0x7418
#define CMPR3    *(volatile unsigned int*) 0x7418
#define SCMPR1   *(volatile unsigned int*) 0x741A
#define SCMPR2   *(volatile unsigned int*) 0x741B
#define SCMPR3   *(volatile unsigned int*) 0x741C

#define CAPCON    *(volatile unsigned int*) 0x7420
#define CAPFIFO   *(volatile unsigned int*) 0x7422
#define CAP1FIFO  *(volatile unsigned int*) 0x7423
#define CAP2FIFO  *(volatile unsigned int*) 0x7424
#define CAP3FIFO  *(volatile unsigned int*) 0x7425
#define CAP4FIFO  *(volatile unsigned int*) 0x7426

#define EVIMRA   *(volatile unsigned int*) 0x742C
#define EVIMRB   *(volatile unsigned int*) 0x742D
#define EVIMRC   *(volatile unsigned int*) 0x742E
#define EVIFRA   *(volatile unsigned int*) 0x742F
#define EVIFRB   *(volatile unsigned int*) 0x7430
#define EVIFRC   *(volatile unsigned int*) 0x7431
#define EVIVRA   *(volatile unsigned int*) 0x7432
#define EVIVRB   *(volatile unsigned int*) 0x7433
#define EVIVRC   *(volatile unsigned int*) 0x7434

#define SYSCR   *(volatile unsigned int*) 0x7018

#define PCDATDIR     *(volatile unsigned int*) 0x709C


#endif
```