Using the ADSP-21990 EZ LITE Board for Implementing a Narrow Band Digital Radio Receiver

by Cătălin Ionescu
Radio Consult SRL
16th Ritmului Str.
Bucharest
73347 Romania
Web: http://www.radioconsult.ro
E-mail: office@radioconsult.ro

© Radio Consult SRL, February 2003

Table of Contents

1. Summary..........3
2. ADSP-21990 EZ LITE Hardware Changes..........3
3. Programming the ADC..........5
4. Programming the SPI for audio output..........8
5. Implemented demodulator structure..........9
   5.1. Quadrature DDS and I and Q mixers..........10
   5.2. Low-pass filters with decimations..........12
   5.3. Computing the arctan derivative..........12

© Radio Consult SRL, February 2003

1 Summary

Although the ADSP-21990 EZ LITE board has not been designed for such applications, the practical performances of the ADC built in the DSP and of the input amplifiers installed on the board recommend it for implementing a digital radio receiver. All needed components are present by default on the ADSP-21990 EZ LITE and the required changes are minimal.

This paper presents an example of digital radio receiver implementation, considering that there is a radio frequency front-end with a final intermediary frequency below 100MHz and an intermediate frequency bandwidth below 1MHz.

2 ADSP-21990 EZ LITE Hardware Changes

The original ADC input amplifier network has a low-pass characteristic with a very low cut-off frequency when compared to the targeted range. As for using the ADC with DMA controlled transfer at maximum possible sampling rate only VIN0 input can be sampled, only the passive network corresponding to its amplifier must be affected.

By soldering a 330pF ceramic capacitor over R29 and removing both C32 and C71 the frequency characteristic will become a high pass one, usable from 100kHz up to 100MHz. Practical tests showed that even signals at 300MHz can be sampled, but only if their level is high enough to compensate the falling characteristic of the amplifiers in AD8044.

The original schematic of the input amplifier of VIN0 is presented in figure 1 and the schematic after the changes are done is presented in figure 2.

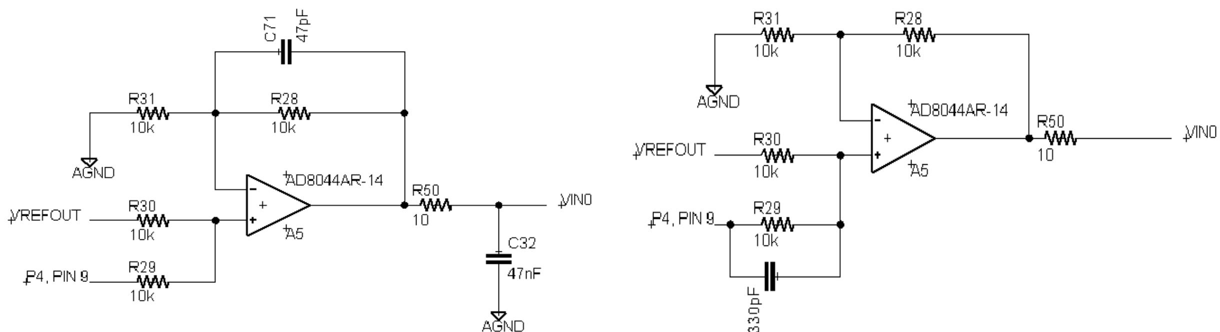

Figure 1 Original schematic of the VIN0 input amplifier

Figure 2 Modified schematic of the VIN0 input amplifier

For determining the actual frequency characteristic after the input amplifier network has been modified, a SMT 06 signal generator produced by Rohde&Schwarz has been used to inject a variable frequency unmodulated signal with an amplitude of 56mV. By measuring the input level at the center of each Nyquist region below 100MHz, the characteristic in figure 3 has been obtained.

The frequency limits for Nyquist regions are given by the following expression

$$f_{min} = f_s \cdot \frac{i-1}{2} \quad ; \quad f_{max} = f_s \cdot \frac{i}{2}$$

$$\textit{where } f_s \textit{ is the ADC sampling rate}$$

The actual frequency of the signal injected in the EZ LITE board stays in the middle of each Nyquist zone, and its value is given by the formula:

$$f = f_s \cdot \frac{i-0.5}{2}$$

On the horizontal axis are displayed the indexes of the Nyquist regions.

The ADC sampling rate used for these tests was 2,343,750Hz.

© Radio Consult SRL, February 2003

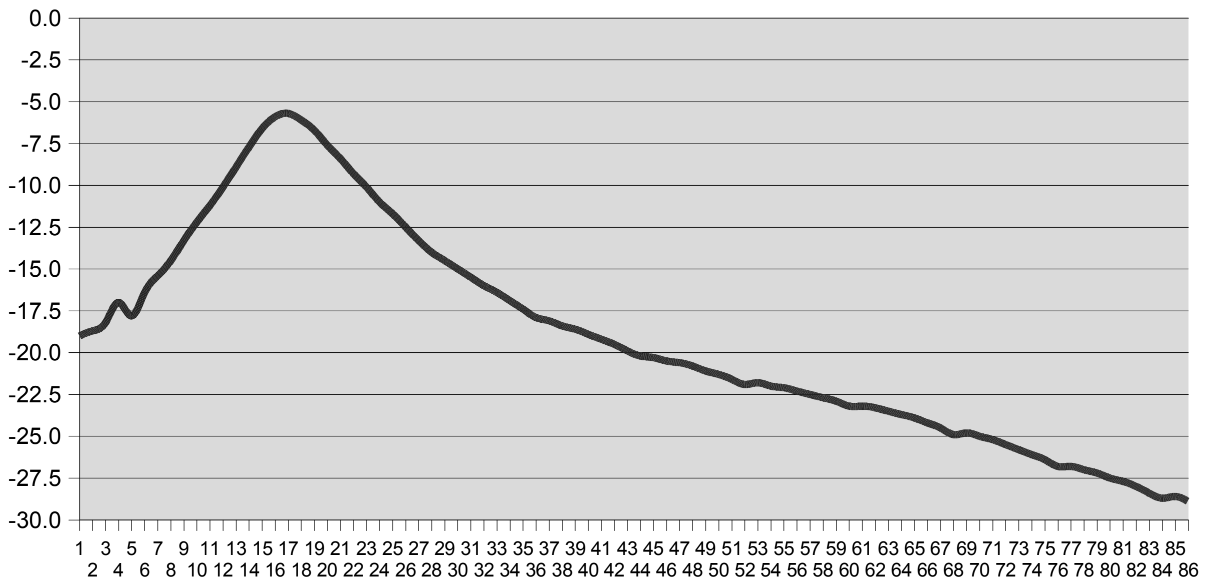

Figure 3 Input frequency characteristic for 56mV signal

The measurements have been done with a dedicated DSP program that captured 2048 samples from the ADC, transferred them using the RS232 connection to the application in the PC that computed the FFT and displayed it as mirrored in the first Nyquist zone.

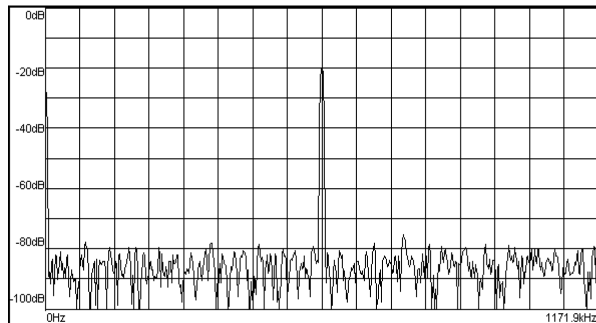

Figure 4 586kHz input signal

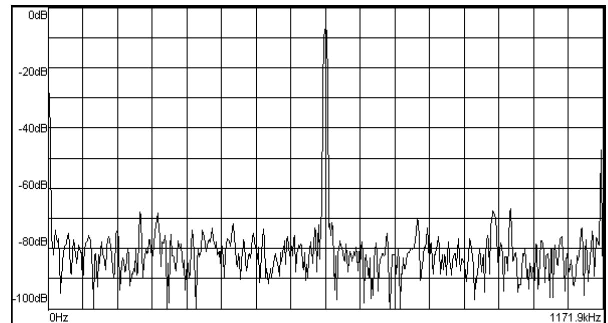

Figure 5 19.336MHz input signal

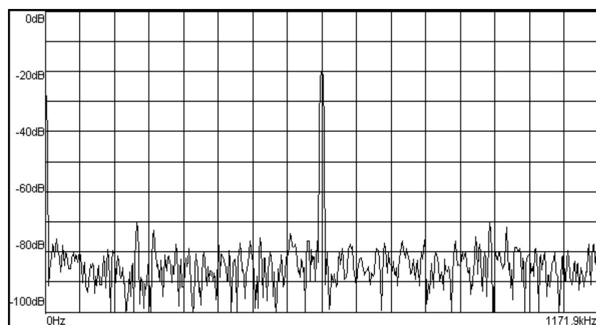

Figure 6 45.117MHz input signal

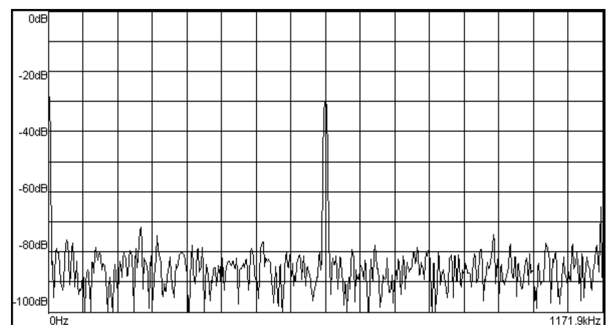

Figure 7 100.195MHz input signal

The block diagram for the platform used in these digital radio tests are shown in figure 8. The modulation signal, containing both voice and music, has been written on an audio CD in order to insure continuous playback, without any possible interference with the software running on the PC. The headphones output from the CD unit has been connected to the modulation input of the generator, having also the possibility of changing the level of the modulation signal by simply controlling the volume button on the CD unit. The signal output from the modulated signal generator has been connected to the VIN0 input pin on ADSP-21990 EZ LITE (P4, pin 9). Two DAC outputs

(P5, pins 2 and 6) have been designated as audio outputs and connected to the Line-in connector of the sound card installed in the PC. The audio output of the sound card can be sent either to headphones or speakers. An extra serial connection has been established between one of the COM ports of the computer and the RS-232 connector on the ADSP-21990 EZ LITE board for samples transfer when studying the input frequency characteristic. The same connection, with properly written software, can be used for controlling various parameters of the implemented demodulators.

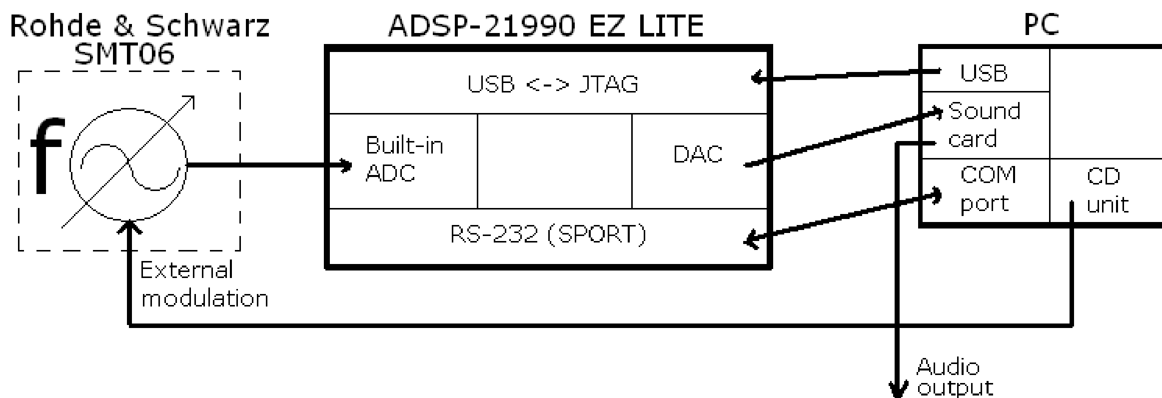

Figure 8 Test structure connections

3 Programming the ADC

In order to be able to maintain a continuous stream of input samples while the demodulator is running, the ADC samples must be copied to the DSP memory through the associated DMA controller. As descriptor based DMA transfers require a significant amount of DSP time when the end of the list of descriptors is reached, auto-buffering is used instead.

The ADC sampling rate is configured to the highest possible value to insure enough input bandwidth for wider transmissions and to accommodate wider input filters without undesired distortions of the demodulated audio due to aliasing of the input signal. The conversion cycle is started through a software command and it will go on as long as the ADC is not stopped, continuously storing samples into the DMA buffer.

Maximum ADC sampling rate is given by the maximum DSP core clock frequency and corresponding peripheral clock frequency. For the 50MHz oscillator on the EZ LITE board, the maximum achievable DSP core clock frequency is 150MHz. Maximum peripheral clock frequency corresponding to this value is 75MHz. As the DSP resetting disables the on-chip PLL used for core clock multiplying, it must be enable through the following sequence of instructions:

```
.section/pm program;
//  Configures the PLL to obtain 150MHz core clock rate and 75MHz peripheral
// clock rate. Even if 160MHz core clock would be desireable, the 50MHz clock
// allows only 150MHz. The sequence of commands first turns the PLL off in
// bypass mode, reprograms the multiplier, turns the PLL back on and exits
// bypass mode. At the end, a delay loop is performed to insure proper clock
// for the rest of the execution.
iopg = Clock_and_System_Control_Page;
//      |---------------- multiplier select
//      |       |--------- bypass PLL multiplier
//      |       ||-------- divide CLKIN/2 in bypass mode
//      |       |||------- CLKOUT enable (CLKOUT = HCLK)
//      |       ||||------ powerdown
//      |       |||||----- core:peripheral clock ratio (if 1 HCLK=CCLK/2)
//      |       ||||||---- stop core clock
//      |       |||||||--- stop all PLL output
//      |       ||||||||-- PLL off
//      |       |||||||||- divide frequency
```

© Radio Consult SRL, February 2003

```
    //      |-----||||||||||
    ar = b#0000010101010010; io(PLLCTL) = ar; nop;
    ar = b#0000011101010010; io(PLLCTL) = ar; nop;
    ar = b#0000011101010000; io(PLLCTL) = ar; nop;
    ar = b#0000011001010000; io(PLLCTL) = ar;
    cntr = 1100;
    do PLLTurnOnLoop until ce;
  PLLTurnOnLoop:
        nop;
```

With the above values, the actual ADC maximum sampling rate is 2,343,750Hz (1/64 of the core clock frequency). With only 64 DSP cycles for each input sample, no demodulation can be performed, so a series of decimations must be done before the actual demodulators. When choosing the decimation factors there are more things to be considered, most important ones being the bandwidth of the transmission to be received and the number of cycles required to perform the low-pass filtering before the actual decimation. A higher decimation factor gives lower execution time, while a lower one gives a wider accepted transmission bandwidth.

Considering as targets AM and FM radio broadcast transmissions, both 9kHz and 300kHz input bandwidths are required. The corresponding I and Q signals need only half of those bandwidths and they are both obtained while decimating, so throughout the demodulator there will be no single sample, but (I,Q) pairs of samples.

By implementing a series of decimations best results are obtained as there is no need for very large filters. A first decimation by 8 gives a sampling rate of 292,969Hz, enough for wide FM side bands. The demodulated FM audio has a bandwidth of 15kHz, so a sampling rate over 30kHz is required. By further decimating the demodulated audio with a factor of 9, the final sampling rate is 32,552Hz, value that must be used by the output DACs, too. For AM the 32,552Hz sampling rate is used.

If there are even narrower transmission modes that must be implemented, further decimations could be done, for example with a factor of 4, obtaining a final sampling rate of 8,138Hz.

The implementation presented in this paper considers all these three decimating factors as the final one brings another advantage – by processing larger blocks of samples, the ADC DMA interrupt overhead and demodulator initializations overhead are reduced to minimum.

The last thing that is very important to correctly demodulating the input signal is the actual size of the ADC DMA buffer. So far the value that should be specified is the product of the three decimation factors, but this is not entirely true. As there are a lot if initializations required by the demodulator for each buffer of input samples, the first sample or even the first two samples are lost as they are overwritten by the newly received ones.

This problem is overcome by simply copying the samples in the ADC DMA buffer into a secondary buffer that the demodulator code actually processes. This buffer copying requires very simple initializations and thus no sample is lost.

The ADC DMA interrupt signals to the main program loop that a new buffer of samples is ready for processing through the ***ADC_NewSamples*** variable. Normally, if there are no new samples to be processed, the main loop of the program just waits for the new buffer to be completely filled.

The ADC DMA interrupts are sent to interrupt 5 to have a high priority, but not higher than the SPI interrupt used for audio samples output. The issue of properly associating the interrupt vectors to the hardware interrupts used is better explained in chapter 4. When an interrupt occurs, the corresponding status flags are cleared and the main loop is signaled that new samples are ready for processing.

The constants and the DSP code used for ADC initializations, conversion starting, interrupt handling and main program loop skeleton are the following:

```
//        Decimation coefficients for various modes - in order to have enough
// processing time, the decimation for FMW will be considered "basic decimation"
// and all others should be multiple of this value for best performance and less
// restrictions. Even more, in order to reduce the program memory occupied by the
// coefficients of the decimation filters, FMW decimation will ALWAYS be applied
// and all other decimations will follow it.
//        For proper algorithm running, the FMW decimation coefficient should be an
// even value and it should be as large as necessary in order to have a
// reasonable execution time for each buffer of samples while providing the
// required result quality.
#define    Decimation_293kHz              8
#define    Decimation_32kHz               9
#define    Decimation_8kHz                4
//  ADC DMA buffer size must be large enough to contain the samples required for
// a single output sample after the highest decimation value. Even more, the
// actual ADC DMA buffer should be twice larger than the theoretical value as the
// DSP will most surely not be able to start processing the samples before at
// least one new sample is written, thus overwriting the samples to be processed.
// The effective code is rather simple due to the bug in the ADSP-219x core that
// causes extra DMA interrupts when the middle of the buffer is encountered.
#define ADC_DMABufferSize (Decimation_293kHz*Decimation_32kHz*Decimation_8kHz)
//  Buffer used to store the converted input samples
    .section/data data1;
    .var ADC_Buffer[ADC_DMABufferSize];
    .var ADC_WorkBuffer[ADC_DMABufferSize];
    .var ADC_NewSamples = 0;

    .section/pm program;
    //  Configures the ADC for DMA controlled sampling of the specified number
    // of channels. The DMA is used in autobuffering mode to avoid extra
    // complexity required by the linked list of descriptors. For conversion
    // start, the software command is chosen as DMA takes care of everything
    // else. When the buffer is filled, an interrupt is issued and the DMA goes
    // back to the first location.
    iopg = ADC_Page;
    //      |---------------- ADCDATSEL - data format select - 0 = bit 0 is OTR
    //      | |-------------- LATCHBSEL - latch B select
    //      | ||------------- LATCHASEL - latch A select
    //      | |||------------ ADCCLKSEL - ADCCLK = HCLK / 2 / ADCCLKSEL
    //      | |||     |------- MODSEL - ADC operating mode select
    //      | |||     |   |--- TRIGSRC - ADC trigger event select
    //      |x|||--|x|-|x|-|
    ar = b#0000001001000111; io(ADC_CTRL) = ar;
    //  Configures the DMA port of the ADC. Autobuffering is enabled using a
    // single buffer.
    ar = 0x0010; io(ADC_CFG) = ar;
    ar = page(ADC_Buffer); io(ADC_SRP) = ar;
    ar = ADC_Buffer; io(ADC_SRA) = ar;
    ar = length(ADC_Buffer); io(ADC_CNT) = ar;
    ar = 0x0015; io(ADC_CFG) = ar;


    .section/pm program;
    //  Starts ADC conversion.
    iopg = ADC_Page; ar = 1; io(ADC_SOFTCONVST) = ar;
    //  Enables the interrupts.
    ena int;
   MainLoop:
    //  Checks for an ADC DMA interrupt.
    ar = dm(ADC_NewSamples); ar = pass ar; if eq jump MainLoop;
    //  Copies the samples from the DMA buffer to the work buffer.
    i0 = ADC_Buffer; l0 = 0; i1 = ADC_WorkBuffer; l1 = 0;
    cntr = length(ADC_Buffer);
    do BufferCopyLoop until ce;
        ar = dm(i0 += 1);
   BufferCopyLoop:
        dm(i1 += 1) = ar;
                                       ........

    //  Resets the ADC DMA interrupt indicator.
    ar = 0; dm(ADC_NewSamples) = ar;
```

```
    //  Goes back to the ADC DMA interrupts waiting loop.
    jump MainLoop;


//  This interrupt is allocated to ADC DMA end-of-transfer as it is almost the
// most important event. When the ADC buffer is filled with new samples, the ADC
// must be immediately disabled and the host should be notified of the event.
    .section/pm IVint5;
    //  The secondary sets of registers are activated to insure all possible
    // interferences to the rest of the code are avoided.
    ena sec_reg; ax0 = iopg; iopg = ADC_Page;
    //  Clears the interrupt flag to insure proper code execution until the
    // next buffer will be filled.
    ar = 0x0003; io(ADC_IRQ) = ar;
    //  Signals the ADC DMA buffer is filled with new values.
    rti (db); dm(ADC_NewSamples) = ar;
    //  Restores the I/O page register.
    iopg = ax0;
```

4 Programming the SPI for audio output

Even if DMA transfers would have been the best solution when considering the amount of time required for treating the interrupts for each transferred word, it has two major disadvantages:

– if a stereophonic demodulator is implemented the two DACs should receive the word loading command simultaneously after 32 bits are transferred, while the DMA usage would cause 2 or 4 load pulses;

– if a monophonic demodulator is implemented and only one audio output is needed, using 16-bit words would simply cause a cycle of 17 SPI clocks and that would require a total decimation in the actual demodulator multiple of 17, thus reducing a lot the possible combinations;

– if a monophonic demodulator is implemented and 8-bit words are used for transferring the audio samples to the DACs, the cycle would take 18 SPI clocks, but two load pulses would be generated, thus making the output completely unusable.

Considering all these, the only solution left is an interrupt based algorithm that uses 8-bit words on the SPI connection, that combined to the stereophonic output give a total cycle of 36 SPI clocks. As the ADC sampling rate must be an integer multiple of the audio samples transmitting cycle, the first two decimations are by factors of 8 and 9, giving a total of 72.

Also, for loading pulses generating, the automatic control through the SPI logic is used, but it is only activated for the first 8-bit word transmitted for each pair of samples, to load into the actual DACs the samples previously transferred. If the interrupt handling routine must contain more code, tests must be done as it possible to need the SPI logic active on the last 8-bit word of a pair of samples. For the other three 8-bit words it is disabled. Care must be taken in the rest of the code when using the PFx flags to avoid generating undesired load pulses to the two DACs.

The SPI interrupts are mapped to interrupt 4 in order to have the maximum possible priority. Even if the first impression would be that the ADC DMA interrupts should have the highest priority, the relatively large input buffer used for the ADC allows significant delays in interrupt handling, while the continuous stream of 8-bit words sent to the two DACs can't be maintained if the SPI interrupts isn't handled as fast as possible. Also because interrupts nesting is not a good solution for such applications, other hardware interrupts sources should be avoided as much as possible. Instead of using interrupts, the periodicity of the ADC interrupts could be used for polling every other source of hardware interrupts.

The constants and the DSP code used for SPI initializations, interrupt handling and samples loading into the output queue are the following:

© Radio Consult SRL, February 2003

```
//  The index of the currently transmitted 8-bit SPI word
    .section/data data1;
    .var DAC_Status = 0;
//  Audio output samples delay line used for storing the audio samples before
// sending them to the DACs
    .section/data data1;
    .var AudioDelayLine[2*ADC_DMABufferSize/Decimation_293kHz/Decimation_32kHz*2];
    .var AudioDelayLineInput =  AudioDelayLine,AudioDelayLineOutput =  AudioDelayLine;


    //  Configures the SPI for usage with the two 4 channels DACs. They will be
    // used for demodulated samples output. In order to keep everything as simple
    // as possible, channels 0 and 4 will be used for Left and Right. This will
    // allow a minimum audio output overhead. Due to the huge amount of time
    // required for data transmission, over 128 DSP clocks, it must be interrupt
    // driven. The first word will be "manually" transmitted and the second one
    // will be handled by the interrupt. A second interrupt will cause immediate
    // SPI transmission ending.
    iopg = SPI0_Controller_Page;
    //      |-------------- SPI module enable
    //      ||------------- open drain data output enable
    //      |||------------ SPI module is master
    //      ||||----------- clock polarity (1 = active-low)
    //      |||||---------- clock phase
    //      ||||||--------- data format (1 = LSB first)
    //      |||||||-------- word length (0 = 8 bits, 1 = 16 bits)
    //      |||||||  |----- enable MISO pin as output
    //      |||||||  ||---- enable slave-select input for master
    //      |||||||  |||--- when RDBRx full: 1 = get more data, 0 = discard
    //      |||||||  ||||-- send zero (1) or last word (0) when TDBRx empty
    //      |||||||  |||||- transfer initiation mode and interrupt generation
    //     x|||||||xx||||||
    ar = b#0001100000000101; io(SPICTL0) = ar;
    //  Configures the SPI baud rate to 2/(Decimation_293kHz*Decimation_32kHz) of
    // peripheral clock. A correction coefficient of 36/32 should be applied as
    // there is one clock cycle missing between the four transmitted bytes, thus
    // giving four missing clock cycles for each set of samples.
    ar = Decimation_293kHz * Decimation_32kHz * 32 / 36 / 2; io(SPIBAUD0) = ar;
    //  Configures PF2 and PF3 for SPI usage and changes them to logic "1".
    ar = 0x0000; io(SPIFLG0) = ar;
    //  Enables the SPI controller.
    ar = io(SPICTL0); ar = setbit 14 of ar; io(SPICTL0) = ar;


    ar = AudioDelayLine; reg(B2) = ar;
      i2 = dm(AudioDelayLineInput); l2 = length(AudioDelayLine);
    ar = mr1 + 0x8000; sr = lshift ar by -4 (lo);
      dm(i2 += 1) = sr0; dm(i2 += 1) = sr0;
    dm(AudioDelayLineInput) = i2;


//  This handler is allocated to the SPI transmit buffer empty interrupt. At the
// first occurence, the word for DAC 0 is transmitted as the one for DAC 4 has
// already been sent. At the second occurence the SPI is completely stopped.
    .section/pm IVint4;
    //  The secondary sets of registers are activated to insure all possible
    // interferences to the rest of the code are avoided.
    ena sec_reg; ena sec_dag; ax0 = iopg; iopg = SPI0_Controller_Page;
    //  Prepares the audio delay line accesing registers.
    ar = AudioDelayLine; reg(B3) = ar;
      i3 = dm(AudioDelayLineOutput); l3 = length(AudioDelayLine);
    //  Computes the next status of the audio samples transmitter.
    ar = dm(DAC_Status); ar = ar + 0x4000; dm(DAC_Status) = ar;
    //  Extracts the next DAC data word and sends its currently required 8bit
    // fragment.
    sr0 = dm(i3 += 1);
    af = ar and 0x4000; se = -8; if ne sr = lshift sr0 (lo);
    io(TDBR0) = sr0;
    //  If the value MSB for DAC 4 must be transmitted, it allows the generation
    // of a load pulse for the two DACs.
    af = ar - 0x4000; sr0 = 0x000C; if ne sr = lshift sr0 (lo);
```

```
// Gives control to the PF2 and PF3 pins to the SPI if needed and restores
// the I/O page register.
af = ar and 0x4000; if ne rti (db); io(SPIFLG0) = sr0; iopg = ax0;
// Sends the new word on the SPI connection.
rti (db); dm(AudioDelayLineOutput) = i3; nop;
```

5 Implemented demodulator structure

The actual demodulators presented in this application note (see figure 9) are the theoretical ones, thus providing much better results than the classical analogical equivalents, if enough precision is used in all computations. As the purpose of this paper is to prove the possibility of using the ADSP-2199x mixed signals DSPs for implementing digital radio receivers, all computations are kept as simple as possible just to be easier to understand.

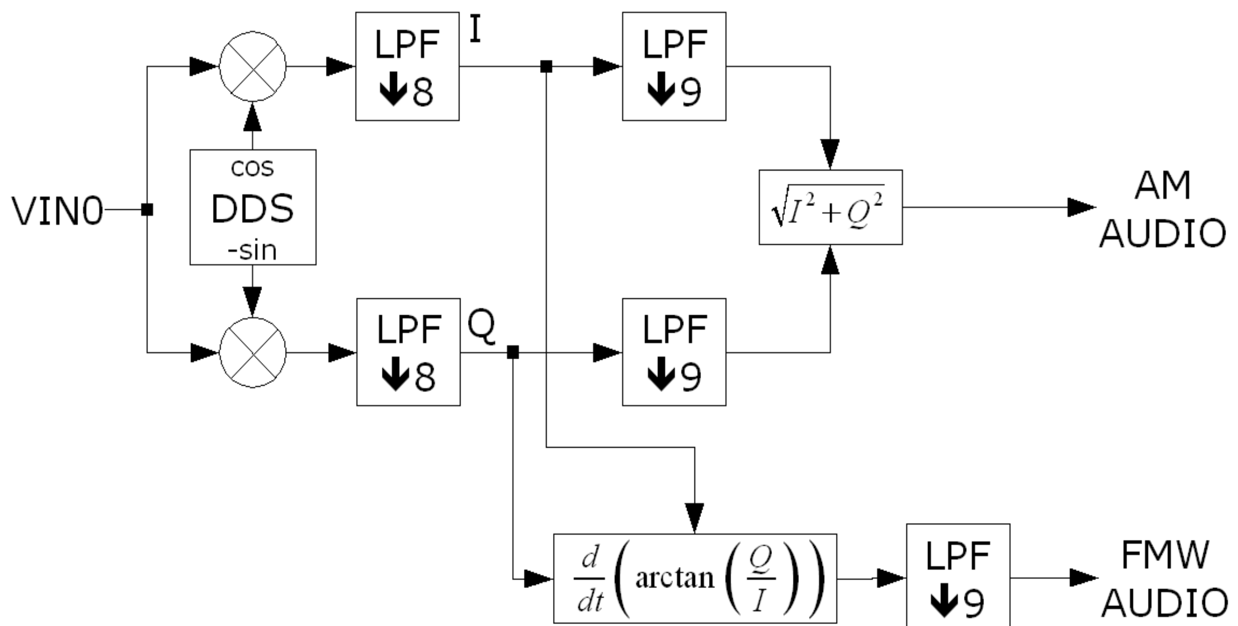

Figure 9 General demodulator structure

As understanding the DSP source code requires good knowledge of assembly code and signal processing and most blocks implemented in the code are very well described in other papers, only the things that are unusual will be presented here, as they are the key to the demodulators implementation in a time effective manner.

5.1 Quadrature DDS and I and Q mixers

Classically, the quadrature oscillator would have a variable holding the sine/cosine phase and would call the sine and cosine computing routines for each input sample. While for low sampling rates this would provide the best quality, the limited number of cycles for each input sample, just 64 in this case, requires a solution that avoids calling twice the 34 cycles sine computing routine.

Of course, the first idea is to use a software replica of a DDS, with a table with sine values large enough to insure the required quality, but also small enough to fit the limited amount of on-chip memory. The upper bits of the sine/cosine phase would simply provide the index to access the table in order to obtain the desired value. But even this technique is too slow for this situation, with over 10 cycles for each input sample.

After refining the implementation of the above, a solution that needs only 7 cycles for each input sample has been established, solution that also implements the two mixers. Apart from the look-up table with sine samples and phase increment values, also two pointers for accessing the sine and cosine values in the same table are used. By having

© Radio Consult SRL, February 2003

an initial difference between the two pointers equal to a quarter of the table size, and by maintaining this difference constant in time, the same table can be successfully used for both sine and cosine values.

```
//  Sine/Cosine look-up table that can be used for generating frequencies in
// 2^(16*k+10) steps - is needed for generating all virtual oscillators in the
// demodulator software with the highest possible speed.
    .section/pm data2;
    .var SineTable[1024] = "SineTable.dat";
//  IF sine and cosine components of the input quadrature oscillator - store the
// IF frequency value as a 32-bit value and the current indexes into the
// sine/cosine look-up table. The difference between the two pointers is always
// the same, equal to a fourth of the number of samples in the look-up table, but
// they are stored both for fastest possible accessing.
//  Also the current phase value and the phase increment are stored. This is a
// completely different technique of accessing the look-up table with minimum
// error.
    .section/dm data1;
    .var IFOscSinePointer,IFOscCosinePointer;
    .var IFOscPhase[2],IFOscPhaseInc[2];


    ar = 0; dm(IFOscPhase+0) = ar; dm(IFOscPhase+1) = ar;
    ar = SineTable + (length(SineTable) / 4); dm(IFOscCosinePointer) = ar;
    ar = SineTable; dm(IFOscSinePointer) = ar;
```

The value of the sine/cosine phase increment is giving the desired frequency through the following formula:

$$IFOscPhaseInc = \frac{IF\ Frequency}{Sampling\ Rate} \cdot 2^{26}$$

The resulting value must be loaded to ***IFOscPhaseInc*** using the 16.16 format. **IF frequency** value is not always the IF frequency of the signal fed to the VIN0 input, and determining its value is presented in another chapter.

The minimum execution time for the software quadrature DDS and the two mixers is obtained if everything is done in a single step. So the input samples are fed directly into the delay line of the first decimation filter immediately after multiplication.

The process is further simplified, and so the speed increased, by using a single array for both decimation FIR delay lines. This is achieved by simply interlacing the elements of the two delay lines and using corresponding values for the Mx DAG registers used when accessing them.

Another improvement is obtained by using **cos** and **sin** instead of **cos** and **-sin** and moving the minus to the convolution corresponding to the **Q** low-pass filter.

```
    ar = Decim_293kHz_Delay; reg(B0) = ar;
      i0 = dm(Decim_293kHz_Pointer); l0 = length(Decim_293kHz_Delay);
    i1 = ADC_WorkBuffer; l1 = 0;
    ar = AudioDelayLine; reg(B2) = ar;
      i2 = dm(AudioDelayLineInput); l2 = length(AudioDelayLine);
    ar = Decim_293kHz_Taps; reg(B4) = ar;
      i4 = Decim_293kHz_Taps; l4 = length(Decim_293kHz_Taps);
    ar = SineTable; reg(B5) = ar;
      i5 = dm(IFOscSinePointer); l5 = length(SineTable);
    reg(B6) = ar; i6 = dm(IFOscCosinePointer); l6 = length(SineTable);

                                   ........

            ar = dm(IFOscPhase+0); ay1 = dm(IFOscPhase+1);
            ax0 = dm(IFOscPhaseInc+0); ax1 = dm(IFOscPhaseInc+1);
            af = ax0 + ar; ar = ax1 + C; m7 = ar;
            cntr = Decimation_293kHz - 1;
            ar = ar + ay1, mx0 = dm(i1 += m0), my0 = pm(i5 += m7);
            do IQ_FMW_293kHz_Multiply_Loop until ce;
                mr = mx0 * my0 (rnd), my0 = pm(i6 += m7);
                af = ax0 + af, dm(i0 += m0) = mr1; ay1 = ar, ar = ax1 + C; m7 = ar;
```

```
          mr = mx0 * my0 (rnd); dm (i0 += m0) = mr1;
      IQ_FMW_293kHz_Multiply_Loop:
          ar = ar + ay1, mx0 = dm(i1 += m0), my0 = pm(i5 += m7);
      mr = mx0 * my0 (rnd), my0 = pm(i6 += m7);
      dm (i0 += m0) = mr1, mr = mx0 * my0 (rnd);
      ar = pass af, ay1 = ar;
      dm(IFOscPhase+0) = ar; dm(IFOscPhase+1) = ay1;

                                  ........

dm(Decim_293kHz_Pointer) = i0;
dm(ADC_Buffer_Pointer) = i1;
dm(IFOscSinePointer) = i5; dm(IFOscCosinePointer) = i6;
```

5.2 Low-pass filters with decimations

In the demodulator block diagram the low-pass filters and the decimations have been represented as a single block as the actual implementation does not allow a different representation. Instead of doing the low-pass filtering and using for the next processing only the samples left after decimation, as theory tells, the low-pass filters are executed only for those output samples that remain after the decimation. This significantly improves the execution time.

Another execution time improvement, but not so significant, is given by simultaneously executing both I and Q low pass filtering and the low-pass FIRs have the same coefficients.

5.3 Computing the arctan derivative

While theory specifies that the ideal FM demodulation is given by

$$\frac{d}{dt}\left(\arctan\left(\frac{Q}{I}\right)\right)$$

thus suggesting that in practice the last arctan result should be stored and substracted from the current one, practice shows that such a solution gives results far from expectations as significant phase information is lost.

The trick is rather simple:

$$\arctan\left(\frac{Q}{I}\right)-\arctan\left(\frac{OldQ}{OldI}\right)=\arctan\left(\tan\left(\arctan\left(\frac{Q}{I}\right)-\arctan\left(\frac{OldQ}{OldI}\right)\right)\right)$$

$$\arctan\left(\frac{Q}{I}\right)-\arctan\left(\frac{OldQ}{OldI}\right)=\arctan\left(\frac{OldI\cdot Q-OldQ\cdot I}{OldI\cdot I+OldQ\cdot Q}\right)$$

and by using it, along with an arctan implementation that computes the Y/X ratio (the equivalent of ***atan2*** defined in C/C++), the results are very accurate.