```
/*************************************************************************************
                   DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                                  IMPERIAL COLLEGE LONDON

                      EE 3.19: Real Time Digital Signal Processing
                           Dr Paul Mitcheson and Daniel Harvey

                                  PROJECT: Frame Processing

                             ********* ENHANCE. C **********
                              Shell for speech enhancement

        Demonstrates overlap-add frame processing (interrupt driven) on the DSK.

 *************************************************************************************
                              By Danny Harvey: 21 July 2006
                              Updated for use on CCS v4 Sept 2010
 ************************************************************************************/
/*
 *  You should modify the code so that a speech enhancement project is built
 *  on top of this template.
 */
/**************************** Pre-processor statements ******************************/
//  library required when using calloc
#include <stdlib.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

/* Some functions to help with Complex algebra and FFT. */
#include "cmplx.h"
#include "fft_functions.h"

// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

#define WINCONST 0.85185            /* 0.46/0.54 for Hamming window */
#define FSAMP 8000.0                /* sample frequency, ensure this matches Config for AIC */
#define FFTLEN 256                  /* fft length = frame length 256/8000 = 32 ms*/
#define NFREQ (1+FFTLEN/2)          /* number of frequency bins from a real FFT */
#define OVERSAMP 4                  /* oversampling ratio (2 or 4) */
#define FRAMEINC (FFTLEN/OVERSAMP)  /* Frame increment */
#define CIRCBUF (FFTLEN+FRAMEINC)   /* length of I/O buffers */
#define NNOISEBLOCK 4               /* Number of noise blocks */
#define NOISELEN 316                /* Number of frames for the noise window */
#define NOISEBLOCKLEN (NOISELEN/NNOISEBLOCK)    /* Number of frames for each noise block */

#define OUTGAIN 16000.0             /* Output gain for DAC */
#define INGAIN  (1.0/16000.0)       /* Input gain for ADC  */
// PI defined here for use in your code
#define PI 3.141592653589793
#define TFRAME (FRAMEINC/FSAMP)       /* time between calculation of each frame */
#define LAMBDA 0.05
#define TAU 0.032
#define ALPHA 20
#define ALPHA_FILTER 4
#define MAXOVERSUBTRACTION 1.5
#define MINOVERSUBTRACTION 1
#define OVERSUBTRACTIONCUTTOFF (FFTLEN/2) /* Only first 32 frequency bins are modified*/
#define LOWSNR 100

/******************************** MODE SELECTORS **********************************/
//#define FILTERED
#define FILTERED_POWER
#define FILTERED_NOISE
//#define OVERSUBTRACTION
```

```c
/**************************** MODE SPECIFIC DEFINITIONS *****************************/
#if defined(FILTERED) || defined(FILTERED_POWER)
/* Constants for the use in the low pass filter optimisation */
#if defined(ALPHA)
#undef ALPHA
#endif
#define ALPHA ALPHA_FILTER
#endif

#if defined(OVERSUBTRACTION)
#define OVERSUBTRACTIONDEC ((MINOVERSUBTRACTION-MAXOVERSUBTRACTION)*OVERSUBTRACTIONCUTTOFF/FFTLEN)
#endif

/********************************* MODE HELPERS ************************************/
/* These prevent mutual exclusion errors (e.g. FILTERED and FILTRED_POWER cannot both
 *  be active */
#if defined(FILTERED_POWER) && defined(FILTERED)
#undef FILTERED
#endif
/******************************* Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
             /**********************************************************************/
             /*   REGISTER             FUNCTION                  SETTINGS         */
             /**********************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                   */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                   */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                   */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                   */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off       */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on       */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit                */\
    0x008d,  /* 8 SAMPLERATE Sample rate control        8 KHZ-ensure matches FSAMP */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                    */\
             /**********************************************************************/
};

// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;

float *inBuffer, *outBuffer;        /* Input/output circular buffers */
float *inFrame, *outFrame;          /* Input and output frames */
float *inWin, *outWin;              /* Input and output windows */
float inGain, outGain;              /* ADC and DAC gains */
complex *cBuffer;                           /* Buffers for calculation */
float *magnitude[OVERSAMP], *minMagnitude, *inFiltered, *noiseFiltered;     /* Buffers to store minimum noise
     */
int *lowSNR;
float cpuFrac;                      /* Fraction of CPU time used */
volatile int ioPtr=0;              /* Input/ouput pointer for circular buffers */
volatile int framePtr=0;           /* Frame pointer */
volatile int noiseBlockPtr=0;       /* Noise block pointer */
volatile int frameIndex=0;
float filterConst, remFilterConst;

 /****************************** Function prototypes *******************************/
void initHardware(void);        /* Initialize codec */
void initHWI(void);            /* Initialize hardware interrupts */
void ISR_AIC(void);             /* Interrupt service routine for codec */
void processFrame(void);       /* Frame processing routine */
void toComplex(complex *out, float *in, int length);
void toReal(float *out, complex *in, int length);
float min(float a, float b);
float max(float a, float b);
float clamp(float v, float min, float max);
float sqr(float x);
void spectralSubtraction(int m);
void filterInputs(int k);
```

```
/********************************** Main routine ************************************/
void main()
{

    int k; // used in various for loops

/*  Initialize and zero fill arrays */

    inBuffer    = (float *) calloc(CIRCBUF, sizeof(float)); /* Input array */
    outBuffer   = (float *) calloc(CIRCBUF, sizeof(float)); /* Output array */
    inFrame     = (float *) calloc(FFTLEN, sizeof(float));  /* Array for processing*/
    outFrame    = (float *) calloc(FFTLEN, sizeof(float));  /* Array for processing*/
    inWin       = (float *) calloc(FFTLEN, sizeof(float));  /* Input window */
    outWin      = (float *) calloc(FFTLEN, sizeof(float));  /* Output window */
    cBuffer         = (complex *) calloc(FFTLEN, sizeof(complex)); /* Complex Buffer for calculation */
    minMagnitude        = (float *) calloc(FFTLEN, sizeof(float));
    for (k=0;k<FFTLEN;k++)
        minMagnitude[k] = FLT_MAX;                  /* Initialise the minimum magnitudes to be infinity */
    for(k=0;k<NNOISEBLOCK;k++)
        magnitude[k] = (float *) calloc(FFTLEN, sizeof(float)); /* Create arrays for storing noise */
    inFiltered = (float *) calloc(FFTLEN, sizeof(float));   /* Array for low passed c */
    noiseFiltered = (float *) calloc(FFTLEN, sizeof(float));
    lowSNR = (int *)calloc(FFTLEN, sizeof(int));

    filterConst = exp(-TFRAME/TAU);
    remFilterConst = 1-filterConst;

    /* initialize board and the audio port */
    initHardware();

    /* initialize hardware interrupts */
    initHWI();

/* initialize algorithm constants */

    for (k=0;k<FFTLEN;k++)
    {
        inWin[k] = sqrt((1.0-WINCONST*cos(PI*(2*k+1)/FFTLEN))/OVERSAMP);
        outWin[k] = inWin[k];
    }
    inGain=INGAIN;
    outGain=OUTGAIN;


    /* main loop, wait for interrupt */
    while(1)    processFrame();
}

/********************************** init_hardware() *********************************/
void initHardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

    /* Function below sets the number of bits in word used by MSBSP (serial port) for
    receives from AIC23 (audio port). We are using a 32 bit packet containing two
    16 bit numbers hence 32BIT is set for  receive */
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

    /* Configures interrupt to activate on each consecutive available 32 bits
    from Audio port hence an interrupt is generated for each L & R sample pair */
    MCBSP_FSETS(SPCR1, RINTM, FRM);

    /* These commands do the same thing as above but applied to data transfers to the
    audio port */
    MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
    MCBSP_FSETS(SPCR1, XINTM, FRM);


}
/********************************** init_HWI() **************************************/
```

```
void initHWI(void)
{
    IRQ_globalDisable();            // Globally disables interrupts
    IRQ_nmiEnable();                // Enables the NMI interrupt (used by the debugger)
    IRQ_map(IRQ_EVT_RINT1,4);       // Maps an event to a physical interrupt
    IRQ_enable(IRQ_EVT_RINT1);      // Enables the event
    IRQ_globalEnable();             // Globally enables interrupts

}

/******************************** process_frame() ***********************************/
void processFrame(void)
{
    int k, m;
    int io_ptr0;

    /* work out fraction of available CPU time used by algorithm */
    cpuFrac = ((float) (ioPtr & (FRAMEINC - 1)))/FRAMEINC;

    /* wait until io_ptr is at the start of the current frame */
    while((ioPtr/FRAMEINC) != framePtr);

    /* then increment the framecount (wrapping if required) */
    if (++framePtr >= (CIRCBUF/FRAMEINC)) framePtr=0;

    /* save a pointer to the position in the I/O buffers (inbuffer/outbuffer) where the
    data should be read (inbuffer) and saved (outbuffer) for the purpose of processing */
    io_ptr0=framePtr * FRAMEINC;

    /* copy input data from inbuffer into inframe (starting from the pointer position) */

    m=io_ptr0;
    for (k=0;k<FFTLEN;k++)
    {
        inFrame[k] = inBuffer[m] * inWin[k];
        if (++m >= CIRCBUF) m=0; /* wrap if required */
    }

    /************************ DO PROCESSING OF FRAME  HERE *************************/
    /* Convert input frame to frequency domain */

    toComplex(cBuffer, inFrame, FFTLEN);
    fft(FFTLEN, cBuffer);

    // If we are at the beginning of a noise frame, load the current FFT in
    if(frameIndex == 0)
    {
        for (k=0;k<FFTLEN; k++)
        {
            filterInputs(k);
            magnitude[noiseBlockPtr][k] = inFiltered[k];
            // Do a full compute of the noise over 10 seconds as the current block may have held the minimum
            minMagnitude[k] = min(magnitude[0][k], magnitude[1][k]);
            minMagnitude[k] = min(minMagnitude[k], magnitude[2][k]);
            minMagnitude[k] = min(minMagnitude[k], magnitude[3][k]);
            spectralSubtraction(k);
        }
    }
    else
    {
        // If we aren't in the beginning of a noise frame, compare the fft and load the minimum in
        for (k=0; k<FFTLEN; k++)
        {
            filterInputs(k);
            magnitude[noiseBlockPtr][k] = min(magnitude[noiseBlockPtr][k], inFiltered[k]);
            // Compare th currnt minimum with the previous minimum
            minMagnitude[k] = min(minMagnitude[k], magnitude[noiseBlockPtr][k]);
            spectralSubtraction(k);
        }
    }

    // Convert frequency frame to time domain

    ifft(FFTLEN, cBuffer);
```

```
    toReal(outFrame, cBuffer, FFTLEN);

    if(++frameIndex >= NOISEBLOCKLEN)
    {
        frameIndex = 0;
        if (++noiseBlockPtr >= NNOISEBLOCK) noiseBlockPtr=0;
    }


    /********************************************************************************/

    /* multiply outframe by output window and overlap-add into output buffer */

    m=io_ptr0;

    for (k=0;k<(FFTLEN-FRAMEINC);k++)
    {                                           /* this loop adds into outbuffer */
        outBuffer[m] = outBuffer[m]+outFrame[k]*outWin[k];
        if (++m >= CIRCBUF) m=0; /* wrap if required */
    }
    for (;k<FFTLEN;k++)
    {
        outBuffer[m] = outFrame[k]*outWin[k];   /* this loop over-writes outbuffer */
        m++;
    }
}

void filterInputs(int k)
{
#if defined(FILTERED)
    inFiltered[k] = (remFilterConst*cabs(cBuffer[k])) + (inFiltered[k]*filterConst);
#elif defined(FILTERED_POWER)
    inFiltered[k] = sqrtf(fabs((remFilterConst*sqr(cabs(cBuffer[k]))) + (sqr(inFiltered[k])*filterConst)));
#else
    inFiltered[k] = cabs(cBuffer[k]);
#endif
}

void spectralSubtraction(int m)
{
    float alphaValue, snr, tempValue;
#if defined(FILTERED_NOISE)
    // TODO gt correct filterConst (currently using same as input)
    noiseFiltered[m] = (remFilterConst*minMagnitude[m]) + (noiseFiltered[m]*filterConst);
#else
    noiseFiltered[m] = minMagnitude[m];
#endif
    tempValue = noiseFiltered[m]/inFiltered[m];
#if defined(OVERSUBTRACTION)
    if(lowSNR[m] != 0)
        alphaValue = ALPHA*clamp((OVERSUBTRACTIONDEC*m)+MAXOVERSUBTRACTION, MINOVERSUBTRACTION, 
    MAXOVERSUBTRACTION);
    else
        alphaValue = ALPHA;
#else
    alphaValue = ALPHA;
#endif
    cBuffer[m] = rmul(max(LAMBDA, 1-(alphaValue*tempValue)), cBuffer[m]);
#if defined(OVERSUBTRACTION)
    snr = sqr(cabs(cBuffer[m]))/sqr(noiseFiltered[m]);
    lowSNR[m] = (int)(snr < LOWSNR);
#endif
}
/*************************** INTERRUPT SERVICE ROUTINE  *****************************/

// Map this to the appropriate interrupt in the CDB file

void ISR_AIC(void)
{
    short sample;
    /* Read and write the ADC and DAC using inbuffer and outbuffer */

    sample = mono_read_16Bit();
    inBuffer[ioPtr] = ((float)sample)*inGain;
```

```
        /* write new output data */
    mono_write_16Bit((int)(outBuffer[ioPtr]*outGain));

    /* update io_ptr and check for buffer wraparound */

    if (++ioPtr >= CIRCBUF) ioPtr=0;
}

/******************************** HELPER FUNCTIONS *********************************/

void toComplex(complex * out, float * in, int length)
{
    int i;
//#pragma MUST_ITERATE (FFTLEN, FFTLEN);
    for (i=length-1; i>=0; i--)
        out[i] = cmplx(in[i],0);
}

void toReal(float * out, complex * in, int length)
{
    int i;
//#pragma MUST_ITERATE (FFTLEN, FFTLEN);
    for (i=length-1; i>=0; i--)
        out[i] = in[i].r;
}

float min(float a, float b)
{
    // Hopefully the compiler will have good optimisation for ternaries
    return a < b ? a : b;
}

float max(float a, float b)
{
    // Hopefully the compiler will have good optimisation for ternaries
    return a > b ? a : b;
}

float clamp(float v, float min, float max)
{
    // Hopefully the compiler will have good optimisation for ternaries
    return v <= min ? min : v >= max ? max : v;
}

float sqr(float x)
{
    return x*x;
}

/*********************************************************************************/
```