Introduction and Background

Infinite Impulse Response Basics

Chapter 4 of the course text deals with infinite impulse response (IIR) digital filters. IIR filters often result from a desire to represent a traditional analog filter (Butterworth, Chebyshev, or elliptical) in discrete-time form. As a simple motivational example, consider the RC lowpass filter shown in Figure 1. We may choose to implement this in the discrete-time domain using either the impulse invariant approach or via a bilinear transformation. For details on these two approaches consult a DSP text, such as Oppenheim & Schafer\(^1\). In general we desire approaches to convert any \(H_c(s)\) into \(H(z)\). Let us first recall the general form of a discrete-time IIR system and consider filter topologies that will lead to efficient real-time implementation.

General IIR Form

When a general IIR filter is obtained from say MATLAB’s \texttt{fdatool}, the result is a coefficient set that starts out in direct form, that is we have

\[
H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{1 + a_1 z^{-1} + \cdots + a_N z^{-N}}
\]

where \(N\) is the filter order. The corresponding difference equation which must be implemented in real-time, for a direct-form based filter topology, is

\[
y[n] = - \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]
\]

---

Direct Form I

An IIR filter has feedback, thus from DSP theory we recall the general form of an $N$-order IIR filter is

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{r=0}^{M} b_r x[n-r].$$

(3)

By $z$-transforming both sides of (3) and using the fact that $H(z) = Y(z)/X(z)$, we can write

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}}$$

(4)

IIR filters can be implemented in a variety of topologies, the most common ones, direct form I, II, cascade, and parallel, will be reviewed below.

Direct Form I

Direct implementation of (3) leads to the following structure of Figure 2. The calculation of $y[n]$ for each new $x[n]$ requires the ordered solution of two difference equations

$$v[n] = \sum_{k=0}^{M} b_k x[n-k]$$

(5)

$$y[n] = v[n] - \sum_{k=1}^{N} a_k v[n-k]$$

Figure 2: Direct form I structure for IIR filter implementation.
**Direct Form II**

A more efficient direct form structure can be realized by placing the feedback section first, followed by the feedforward section. The first step in this rearrangement is that of Figure 3.

![Figure 3](image)

**Figure 3:** Rearrange the direct form I structure to place the feedback terms first.

The final direct form II structure is shown below in Figure 4.

![Figure 4](image)

**Figure 4:** Direct form II structure.

The ordered pair of difference equations needed to obtain $y[n]$ from $x[n]$ is

\[
\begin{align*}
    w[n] &= x[n] - \sum_{k=1}^{N} a_k w[n-k] \\
    y[n] &= \sum_{k=0}^{M} b_k w[n-k]
\end{align*}
\]  

(6)
Cascade Form

Since the system function, \( H(z) \), is a ratio of polynomials, it is possible to factor the numerator and denominator polynomials in a variety of ways. The most popular factoring scheme is as a product of second-order polynomials, which at the very least insures that conjugate pole and zeros pairs can be realized with real coefficients

\[
H(z) = \prod_{k=1}^{N_s} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}} = \prod_{k=1}^{N_s} H_k(z),
\]

where \( N_s \approx \lceil (N+1)/2 \rceil \) is the largest integer in \( (N+1)/2 \). A product of system functions corresponds to a cascade of biquad system blocks is shown in Figure 5. The \( k \)th biquad can be implemented using a direct form structure (typically direct form II), as shown in Figure 6. The corresponding biquad difference equations are

\[
w_k[n] = y_{k-1}[n] - a_{1k}w_k[n-1] - a_{2k}w_k[n-2]
\]

(8)

\[
y_k[n] = b_{0k}w_k[n] + b_{1k}w_k[n-1] + b_{2k}w_k[n-2].
\]

(9)

The cascade of biquads is very popular in real-time DSP, is supported by the MATLAB signal processing toolbox, and will be utilized in example code presented later.

Filter s-Domain to z-Domain Transformation

Impulse Invariant Method

A simple and natural way to obtain a discrete-time implementation of an analog filter is to sample the impulse response \( h_c(t) \), i.e., let
\[ h[n] = h_c(nT). \]  
\( \text{(10)} \)

In the frequency domain the analog frequency response, \( H_a(j\Omega) = H_a(\Omega) = H_c(j\Omega) \), becomes the discrete-time frequency response, \( H(e^{j\omega}) \), via the ideal sampling theory result

\[ H(e^{j\omega}) = \sum_{k = -\infty}^{\infty} H_a\left(\frac{\omega}{T} + \frac{2\pi k}{T}\right). \]  
\( \text{(11)} \)

From Figure 7 we see that if \( H_a(\Omega) \) is not bandlimited, aliasing shows up in \( H(e^{j\omega}) \).

**Figure 7:** Frequency response of an impulse invariant filter is likely to have some aliasing when compared to the analog prototype.

Applying this to the RC lowpass filter example introduced earlier, we have

\[ h[n] = Th_c(nT) \]

\[ = \frac{T}{RC}e^{-nT/(RC)}u[n] \]

\( \text{(12)} \)

\[ = \frac{T}{RC}(e^{-T/(RC)})^n u[n] \]

In difference equation form we now have

\[ y[n] = e^{-\frac{T}{RC}}y[n-1] + \frac{T}{RC}x[n] \]  
\( \text{(13)} \)

and the pole-zero plot of Figure 8.

**Figure 8:** Pole-zero plot of RC low-pass filter following impulse invariant transformation.
In the $z$-domain the filter takes the form

$$H(z) = \frac{T/(RC)}{1 - e^{-T/(RC)}z^{-1}}, \quad \text{ROC: } |z| > e^{-T/(RC)}. \quad (14)$$

**Bilinear Transformation Method**

A drawback of the impulse invariant approach is that aliasing of the analog filter’s frequency response can occur unless certain filter roll-off conditions are met. Basically, any portion of the analog filter frequency response that extends above the folding frequency, $f_s/2$, folds into the principle alias band that runs over $[0, f_s/2]$. The bilinear transform approach avoids through a frequency warping transformation which makes the analog frequency axis $0 \leq f < \infty$ is first mapped to the frequency interval $0 \leq f < f_s/2$, before being converted to the discrete-time domain.

It turns out that the impulse invariant technique used the many-to-one mapping

$$z = e^{sT}. \quad (15)$$

To correct the aliasing problem we first employ a one-to-one mapping,

$$s' = \frac{2}{T} \tanh \left( \frac{sT}{2} \right), \quad (16)$$

which compresses the entire $s$-plane into a strip as shown in Figure 9.

![Figure 9: Remapping](image)

Following the compression mapping we convert to the $z$-plane as before, except this time there is nothing that can alias. The complete mapping from $s$ to $z$ is

$$s = \frac{2}{T} \left( \frac{1 - z^{-1}}{1 + z^{-1}} \right) \quad (17)$$

or in reverse

$$z = \frac{1 + \frac{T}{2}s}{1 - \frac{T}{2}s} \quad (18)$$

The frequency axis mapping is of the form

---

*Filter s-Domain to z-Domain Transformation*
The basic filter design equation is

\[ H(z) = H_a(s) \bigg|_{s = \frac{2(1 - z^{-1})}{T(1 + z^{-1})}} \]  

The basic filter design equation is

\[ \Omega = \frac{2}{T} \tan \left( \frac{\omega}{2} \right) \]  

or

\[ \omega = 2 \tan^{-1}(\Omega T/2) \]  

In a practical design in order to preserve desired discrete-time critical frequencies, such as the passband and stopband cutoff frequencies, we use frequency prewarping

\[ \Omega_i = \frac{2}{T} \tan \left( \frac{\omega_i}{2} \right) \]  

where \( \omega_i \) is a discrete-time critical frequency that must be used in the design of an analog prototype with corresponding continuous-time critical frequency \( \Omega_i \). The frequency axis compression imposed by the bilinear transformation can make the transition ratio in the discrete-time domain smaller than in the continuous-time domain, thus resulting in a lower order analog prototype than if the design was implemented purely as an analog filter. Given a ratio of polynomials in the \( s \)-domain, or an amplitude response specification, we can proceed to find \( H(z) \). The resulting transformation is known as the bilinear transform and takes the form

\[ s = \frac{2}{T} \cdot \frac{1 - z^{-1}}{1 + z^{-1}}. \]  

For a given \( s \)-domain filter prototype, \( H_c(s) \), the transformation produces \( z \)-domain system function of the form

\[ H(z) = H_c \left( \frac{2}{T} \cdot \frac{1 - z^{-1}}{1 + z^{-1}} \right). \]  

In the case of the RC lowpass filter example, we have

\[ H(z) = \frac{1}{1 + \frac{2RC}{T} \left( \frac{1 - z^{-1}}{1 + z^{-1}} \right)} = \frac{T}{2RC} \cdot \frac{1 + z^{-1}}{1 + \frac{T}{2RC} \cdot z^{-1}} \]  

\[ = \frac{T/(2RC)}{1 + T/(2RC)} \cdot \frac{1 + z^{-1}}{1 - \left( \frac{T}{2RC} \right)^{-1}} \]  

The difference equation is
The pole-zero plot is shown in Figure 10.

\[
y[n] = \frac{2RC}{2RC + T} y[n-1] + \frac{T}{2RC + T} \{x([n] + x[n-1])\} \quad (26)
\]

Figure 10: Pole-zero plot of RC lowpass filter following bilinear transformation.

**MATLAB Support**

The good news is that MATLAB fully supports the design of IIR digital filters from analog prototypes. There are core functions to design filters at the command line as well as the GUI tool `fdatool`.

IIR filters like FIR filters, are typically designed with amplitude response requirements in mind. As an example consider the bandpass filter amplitude response mask of Figure 11. Note that the sampling rate for this design is \(f_s = 32 \text{ kHz}\). In this example we will use `fdatool` as shown in Figure 12 to design a float coefficients filter using second-order direct form II sections, as described by equations (7)–(9).

Figure 11: Bandpass filter design amplitude response requirements with \(f_s = 32 \text{ kHz}\).
Writing C Coefficient Files

To make it easier to port filter design results from MATLAB to CCS an m-file that generates C style header files in cascade form, `sos_C_header.m`, is available in the Lab 5 ZIP package. This function works with the filter object you export from `fdatool` as shown in Figure 13. The function help is:

```matlab
function sos_C_header(Hq,mode,filename);
% sos_C_header(Hq,mode,filename): Used to create a C-style header file
% containing filter coefficients. This reads Hq filter objects assuming a
```

Figure 12: Using `fdatool` to design the bandpass filter.

Figure 13: Exporting an IIR filter object from `fdatool` for use in `sos_C_header`. 
Currently this program only supports float format, but support for fixed i.e., decimal or hex format would be easy to add if we knew how best to scale the coefficients. Using the \texttt{H\_flt\_BPF} object exported to the workspace in Figure 13, the header file is generated using:

\begin{verbatim}
>> sos\_C\_header(H\_flt\_BPF,'float','iir\_sos\_fltcoeff.h');
\end{verbatim}

The resulting header file is:

\begin{verbatim}
//define number of 2nd-order stages
#define STAGES 6

float b[STAGES][3] = {                           /*numerator coefficients */
    {0.600137829781, 0.656616736634, 0.600137829781}, /*b01, b11, b21 */
    {0.600137829781, -0.249529890259, 0.600137829781}, /*b02, b12, b22 */
    {0.464017748833, 0.591067475089, 0.464017748833}, /*b03, b13, b23 */
    {0.464017748833, -0.311957630267, 0.464017748833}, /*b04, b14, b24 */
    {0.280241042376, 0.501121859665, 0.280241042376}, /*b05, b15, b25 */
    {0.280241042376, -0.435716777653, 0.280241042376}  /*b06, b16, b26 */
};

float a[STAGES][2] = {                           /*denominator coefficients*/
    {-0.084689199924, 0.902873456478},    /*a11, a21 */
    {0.80284518453, 0.911375284195},     /*a12, a22 */
    {0.689005255699, 0.73233838940},     /*a13, a23 */
    {-0.030385173857, 0.71173991585},     /*a14, a24 */
    {0.463345170021, 0.563192009926},    /*a15, a25 */
    {0.150731906295, 0.548954963684}     /*a16, a26 */
};

float scalevalue = 1.000000000000;    /* final output scale value */
//define number of 2nd-order stages
#define STAGES 6

float b[STAGES][3] = {                           /*numerator coefficients */
    {0.600137829781, 0.656616736634, 0.600137829781}, /*b01, b11, b21 */
    {0.600137829781, -0.249529890259, 0.600137829781}, /*b02, b12, b22 */
    {0.464017748833, 0.591067475089, 0.464017748833}, /*b03, b13, b23 */
    {0.464017748833, -0.311957630267, 0.464017748833}, /*b04, b14, b24 */
    {0.280241042376, 0.501121859665, 0.280241042376}, /*b05, b15, b25 */
    {0.280241042376, -0.435716777653, 0.280241042376}  /*b06, b16, b26 */
};

float a[STAGES][2] = {                           /*denominator coefficients*/
    {-0.084689199924, 0.902873456478},    /*a11, a21 */
    {0.80284518453, 0.911375284195},     /*a12, a22 */
    {0.689005255699, 0.73233838940},     /*a13, a23 */
    {-0.030385173857, 0.71173991585},     /*a14, a24 */
    {0.463345170021, 0.563192009926},    /*a15, a25 */
    {0.150731906295, 0.548954963684}     /*a16, a26 */
};

float scalevalue = 1.000000000000;    /* final output scale value */
//define number of 2nd-order stages
#define STAGES 6

float b[STAGES][3] = {                           /*numerator coefficients */
    {0.600137829781, 0.656616736634, 0.600137829781}, /*b01, b11, b21 */
    {0.600137829781, -0.249529890259, 0.600137829781}, /*b02, b12, b22 */
    {0.464017748833, 0.591067475089, 0.464017748833}, /*b03, b13, b23 */
    {0.464017748833, -0.311957630267, 0.464017748833}, /*b04, b14, b24 */
    {0.280241042376, 0.501121859665, 0.280241042376}, /*b05, b15, b25 */
    {0.280241042376, -0.435716777653, 0.280241042376}  /*b06, b16, b26 */
};

float a[STAGES][2] = {                           /*denominator coefficients*/
    {-0.084689199924, 0.902873456478},    /*a11, a21 */
    {0.80284518453, 0.911375284195},     /*a12, a22 */
    {0.689005255699, 0.73233838940},     /*a13, a23 */
    {-0.030385173857, 0.71173991585},     /*a14, a24 */
    {0.463345170021, 0.563192009926},    /*a15, a25 */
    {0.150731906295, 0.548954963684}     /*a16, a26 */
};

float scalevalue = 1.000000000000;    /* final output scale value */

To utilize these 2-D arrays of filter coefficients you need a corresponding filter algorithm. Equations (8) and (9) implement the second-order section using the rows of a [] [] and b [] [] respectively. The actual C-code description is described in the next section.

**Cascade of Biquad Sections C Code Implementation**

A C program, that implements a cascade of biquad sections is \texttt{ISRs\_sos\_iir\_float\_c}. The Lab5.zip contains a complete CCS project which includes the filter coefficient header file \texttt{iir\_sos\_fltcoeff.h}. Portions of this file are given below (full listing can be found at the end of this document):

\begin{verbatim}
#include "iir\_sos\_fltcoeff.h"   //coefficients in decimal format
/* add any global variables here */
float dly[STAGES][2];        //buffer for delay samples
\end{verbatim}
CodecDataIn.UINT = ReadCodecData(); // get input data samples
input = CodecDataIn.Channel[ LEFT]; // filter the left channel
xRight = CodecDataIn.Channel[ RIGHT];

//********** Input Noise Testing ************
//Generate left and right noise samples
//input = ((short)rand_int())>>2;
xRight = ((short)rand_int())>>2;

//********** Input Noise Testing ************

/* add your code starting here */
//Biquad section filtering stage-by-stage
//using the float accumulators 'wn' and 'result'.
for (i = 0; i < STAGES; i++)
{
    //2nd-order LCCDE code
    wn = input - a[i][0] * dly[i][0] - a[i][1] * dly[i][1]; //eqn. 8
    result = b[i][0]*wn + b[i][1]*dly[i][0] + b[i][2]*dly[i][1]; //eqn. 9
    //Update filter buffers for stage i
    dly[i][1] = dly[i][0];
dly[i][0] = wn;
    input = result; /*in case we have to loop again*/
}

//result *= scalevalue; //Apply cascade final stage scale factor

CodecDataOut.Channel[ LEFT] = (short) result; /* scaled L output */
// Copy Right input directly to Right output with no filtering
CodecDataOut.Channel[RIGHT] = (short) xRight; /* scaled R output */
/* end your code here */

WriteCodecData(CodecDataOut.UINT); // send output data to port

The C implementation of the biquad difference equations follows directly from (8) and (9). In particular the array dly [STAGES] [2] holds the filter state (memory/history) information for each stage of the cascade. Ideally this array should be initialized to zeros. The feedforward filter coefficients for each cascade section are contained in the array b [STAGES] [3], where

\[
\begin{align*}
b[i][0] &= b_{i0}, \quad i = 0, 2, \ldots, N_s - 1 \\
b[i][1] &= b_{i1}, \quad i = 0, 2, \ldots, N_s - 1 \\
b[i][2] &= b_{i2}, \quad i = 0, 2, \ldots, N_s - 1
\end{align*}
\]

The feedback filter coefficients for each cascade section are contained in the array a [STAGES] [2]:
\[ a[i][0] = a_{i1}, i = 0, 2, \ldots N_s - 1 \]
\[ a[i][1] = a_{i2}, i = 0, 2, \ldots N_s - 1 \]

Note that you do not need to store \( a_{i0} \) since this coefficient is by definition unity. Each section takes in \( \text{input} \) and produces output in \( \text{result} \). You set \( \text{input} = \text{result} \) at the end of each biquad loop to allow the sample to propagate to the next section until looping is complete. The filter in this example was tested with the on-board AIC3106 codec using a sinusoidal input from the Analog Discovery. Noise testing can also be implemented, as described in Lab 4. You will need to uncomment the noise generator as follows:

```c
//Generate left and right noise samples
input = ((short)rand_int())>>2;  // scale by 1/4
xRight = ((short)rand_int())>>2;  // scale by 1/4
```

Continuing the bandpass example started earlier, Analog Discovery network analyzer is used to obtain the frequency response results shown in Figure 14. Focusing on the magnitude response only, you see that the stopband is indeed down 50 dB. The passband cutoff frequencies are close to the design value of Figure 12. The filter has a positive passband gain of about 0.2 dB. This can of course be changed using a gain scale constant.
Expectations

When completed, submit a lab report which documents code you have written and a summary of your results. Screen shots from the scope and any other instruments and software tools should be included as well. I expect lab demos of certain experiments to confirm that you are obtaining the expected results and knowledge of the tools and instruments.

Problems

1. In this first problem you will implement a specific IIR design to meet certain amplitude response requirements. The filter topology will be a cascade of second-order sections that follows from the example given earlier in the this lab document. Test your design using the network analyzer and the white noise/PC sound card test approach. Here you will use the function sos_C_header() to create a cascade of biquad sections C header file and then

![Figure 14: Bandpass filter frequency response measured using the Analog Discovery.](image)
use the program `ISRs_sos_iir_float.c` (both in the Lab 5 ZIP package) to implement the filter in real-time. Design your filter to meet specific amplitude response requirements given in Figure 15, using an *elliptic lowpass filter prototype*.

\[ |H_a(F)|_{dB} \]

\[ f_s = 32 \text{ kHz} \]

**Figure 15:** Elliptic lowpass filter design requirements.

a) Design the filter in FDA tool and export the design to the MATLAB workspace.

b) Plot the filter theoretical magnitude response in dB versus the analog (continuous-time) frequency axis. This means in \( H(e^{j\omega}) \) you make the substitution \( \omega = 2\pi(f/f_s) \). Overlay the amplitude response design requirements.

c) Obtain the frequency response magnitude in dB from the network analyzer. Compare critical frequencies of the measured response to the theoretical response.

d) Obtain the frequency response magnitude in dB using the white noise approach described in Lab 4. The capture sampling rate should be 44.1 or 48 ksps to help insure that no coloration is introduced by the capture signal processing. By normalizing the passband gain to 0 dB you should be able to overlay the theoretical response directly in MATLAB.
2. Repeat Problem 1 for the type II Chebyshev bandstop filter design shown in Figure 16.

![Figure 16: Bandstop filter design in fdatool.](image)

Repeat parts (a) and (b) and either part (c) or (d).

3. In this problem you will design and implement a second-order tunable frequency bandpass filter. The filter has system function

\[ H_{BP}(z) = \frac{1 - \alpha}{2} \cdot \frac{1 - z^{-2}}{1 - \beta (1 + \alpha) z^{-1} + \alpha z^{-2}} \]  

where

\[ \beta = \cos(\omega_0) = \cos\left(2\pi f_0 / f_s\right) \]

\[ \alpha = \frac{1 - \sin\left(2\pi \Delta f / f_s\right)}{\cos\left(2\pi \Delta f / f_s\right)} \]  

with \( f_0 \) the center frequency, \( \Delta f \) the 3 dB bandwidth, and \( f_s \) the sampling rate in Hz.

a) Show that under the constraint \( a_0 = 1 \) you can write
\[ H_{BP}(z) = \frac{b_0 + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \]  

with

\[ b_0 = \frac{1 - \alpha}{2}, \quad b_1 = 0, \quad b_2 = -\frac{1 - \alpha}{2}, \]
\[ a_0 = 1, \quad a_1 = -\beta(1 + \alpha), \quad a_2 = \alpha \]  

b) Verify using Matlab or Python that theoretical frequency response magnitude is as shown in Figure 17. Also verify that the pole-zero plot of the second-order filter, for the case \( f_0 = 4 \) kHz, \( \Delta f = 1 \) kHz and \( f_s = 32 \) kHz, is as shown in.

![Figure 17: Second-order BPF frequency response magnitude in dB.](image-url)
c) Implement the design on the OMAP-L138 board with $f_s = 32 \text{kHz}$ and with the filter tuning parameters $f_0$ and $\Delta f$ as globally defined parameters that can be controlled via GEL sliders. Obtain the experimental frequency response in dB for one of the cases shown in Figure 17.

The $\sin()$ and $\cos()$ functions are available for float arguments by first including math.h and then calling the functions $\text{sinf()}$ and $\text{cosf()}$ respectively.

```c
#include <math.h>

// TWO_PI_OVER_FS = 2*pi/32000
#define TWO_PI_OVER_FS 0.00019634954084936207
```

...  
// Inside ISR  
beta = cosf(TWO_PI_OVER_FS*f0); // f0 in kHz  
// etc  
...

The cascade of second-order sections is not needed for this implementation. You should just implement the difference equation directly as a direct from structure, i.e.,

$$y[n] = b_0x[n] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2]$$

$$y = b_0x + b_2x_{old2} - a_1y_{old1} - a_2y_{old2}$$

where you will need to maintain state variables, that is past values of the input and output, such as $x_{old1}$, $x_{old2}$, $y_{old1}$, and $y_{old2}$. They will need to be updated at the end of the filtering algorithm before leaving the ISR, i.e.,
// Inside the ISR with xold1, xold2, yold1, yold2 global
y = b0*x + b2*xold2 -a1*yold1 - a2*yold2;
xold2 = xold1;
xold1 = x;
yold2 = yold1;
yold1 = y;

Demo your working OMAP-L138 tunable bandpass filter to the lab instructor. You may want to play music through the filter as an application example.

Code Listings

// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011
// Modified by Mark Wickert February 2012 to include GPIO ISR start/stop postings

///////////////////////////////////////////////////////////////////////
// Filename: ISRs_sos_iir_float
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
///////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "iir_sos_fltcoeff.h"   //coefficients in decimal format

// Function Prototypes
long int rand_int(void);

// Data is received as 2 16-bit words (left/right) packed into one 32-bit word. The union allows the data to be accessed as a single entity when transferring to and from the serial port, but still be able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
    Uint32 UINT;
    Int16 Channel[2];
} CodecDataIn, CodecDataOut;

/* add any global variables here */
float dly[STAGES][2];       //buffer for delay samples
interrupt void Codec_ISR()

///////////////////////////////////////////////////////////////////////
// Purpose: Codec interface interrupt service routine
///////////////////////////////////////////////////////////////////////
// Input: None
// Returns: Nothing
// Calls: CheckForOverrun, ReadCodecData, WriteCodecData
// Notes: None
///////////////////////////////////////////////////////////////////////
{
    /* add any local variables here */
    WriteDigitalOutputs(1); // Write to GPIO J15, pin 6; begin ISR timing pulse
    float xRight;
    //Define filter variables
    int i;
    float wn, input;
    float result = 0; //initialize the accumulator

    if(CheckForOverrun()) // overrun error occurred (i.e. halted DSP)
        return; // so serial port is reset to recover

    CodecDataIn. UINT = ReadCodecData(); // get input data samples

    input  = CodecDataIn. Channel[ LEFT]; // filter the left channel
    xRight = CodecDataIn. Channel[ RIGHT];
    //*********** Input Noise Testing ***************
    //Generate left and right noise samples
    //input = ((short)rand_int())>>2;
    //xRight = ((short)rand_int())>>2;
    //******************************************************************************

    /* add your code starting here */
    //Biquad section filtering stage-by-stage
    //using the float accumulators 'wn' and 'result'.
    for (i = 0; i < STAGES; i++)
    {
        //2nd-order LCCDE code
        wn = input - a[i][0] * dly[i][0] - a[i][1] * dly[i][1]; //eqn. 8
        result = b[i][0]*wn + b[i][1]*dly[i][0] + b[i][2]*dly[i][1]; //eqn. 9
        //Update filter buffers for stage i
        dly[i][1] = dly[i][0];
        dly[i][0] = wn;
        input = result; /*in case we have to loop again*/
//result *= scalevalue; //Apply cascade final stage scale factor

CodecDataOut.Channel[ LEFT] = (short) result; /* scaled L output */
// Copy Right input directly to Right output with no filtering
CodecDataOut.Channel[RIGHT] = (short) xRight; /* scaled R output */
/* end your code here */

WriteCodecData(CodecDataOut.UINT); // send output data to port
WriteDigitalOutputs(0); // Write to GPIO J15, pin 6; end ISR timing pulse

//White noise generator for filter noise testing
long int rand_int(void)
{
    static long int a = 100001;

    a = (a*125) % 2796203;
    return a;
}

sos_C_header

function sos_C_header(Hq,mode,filename);
%        sos_C_header(Hq,mode,filename): Used to create a C-style header file
%        containing filter coefficients. This reads Hq filter objects assuming a
%        direct-form II cascade of second-order sections is present
%        
%        Hq = quantized filter object containing desired coefficients
%        mode = specify 'float' (plan for fixed and hex to be added later
%        file_name = string name of file to be created

%Check to see what type of Hq object we have
if strcmp(Hq.FilterStructure,'Direct-Form II, Second-Order Sections') == 0,
    disp('Wrong structure type, no file written.')
    disp(['Type found is: ' Hq.FilterStructure ' not Direct-Form II, Second-Order
Sections!'])
    return
end

dimSOS = size(Hq.sosMatrix);
Ns = dimSOS(1); % Number of biquad sections

num = zeros(Ns,3);
den = zeros(Ns,3);
for i=1:Ns,
```matlab
num(i,:) = Hq.sosMatrix(i,1:3);
num(i,:) = num(i,:)*Hq.ScaleValues(i);
den(i,:) = Hq.sosMatrix(i,4:6);
end

if length(Hq.ScaleValues) == Ns+1
    scalevalue = Hq.ScaleValues(Ns+1);
else
    scalevalue = 1.0;
end

fid = fopen(filename,'w'); % use 'a' for append
fprintf(fid,'//define number of 2nd-order stages
');
fprintf(fid,'#define STAGES %d
',Ns);

kk = 1;
switch lower(mode)
case 'float'
    fprintf(fid,'float b[STAGES][3] =    {                            
        /*numerator coefficients */
        for i=1:Ns,
            if i==Ns
                fprintf(fid,'{%15.12f, %15.12f, %15.12f}',...
                    num(i,1),num(i,2),num(i,3));
            else
                fprintf(fid,'{%15.12f, %15.12f,...
                    %15.12f},',num(i,1),num(i,2),num(i,3));
            end
            fprintf(fid,' /*b0%1d, b1%1d, b2%1d  */
',i,i,i);
        end
        fprintf(fid,'};
        float a[STAGES][2] =    {                            
        /*denominator coefficients*/
        for i=1:Ns,
            if i==Ns
                fprintf(fid,'{%15.12f, %15.12f} ',den(i,2),den(i,3));
            else
                fprintf(fid,'{%15.12f, %15.12f},',den(i,2),den(i,3));
            end
            fprintf(fid,'                  /*a1%1d, a2%1d  */
',i,i);
        end
        fprintf(fid,' */b0%1d, b1%1d, b2%1d */\n',i,i,i);
    end
    fprintf(fid,'});
    fprintf(fid,'float scalevalue =  %15.12f;', scalevalue);
else
    fprintf(fid,' */ final output scale value */\n');
end
otherwise
    disp('Unknown mode!')
```
end
fprintf(fid,'/***************************************************************\n'); fclose(fid);