Introduction to programming with the TMS320C6711 DSK

Mathias Johansson

2003-11-07

Abstract

This document provides an introductory guide to programming the TMS320C6711 DSP Starter Kit from Texas Instruments. It describes the basic building blocks in typical DSP applications, such as accessing the inputs and outputs of the onboard codec and the PCM3003 audio daughter card, FIR and IIR filtering, and using the onboard switches and light-emitting diodes. A code library is provided which can be manipulated and rebuilt to suit the user's needs.

Contents

1 Introduction **2**

2 Overview of the code library **2**

3 Common files that are used in all projects **3**

4 Creating a new project **4**

4.1 Using BSL for accessing dip switches and LEDs 5

4.2 Using assembly language functions from DSPLib 5

4.3 Introduction to debugging 5

5 Introduction to the accompanying projects **6**

5.1 The **loop_PCM3003** project 7

5.2 The **FIR_PCM3003** project 7

5.3 The **FIR2_PCM3003** project 8

5.4 The **IIR** project 8

5.5 The **IIR_adjustable** project 9

5.6 The **loop_8k** project 9

5.7 The **FIR2_8k**, **IIR_8k**, and **IIR_adjustable_8k** projects 10

6 More information **24**

1 Introduction

The Texas Instruments TMS320C6711 DSP starter kit consists of a DSP board with an onboard mono codec (AD535) sampling at a frequency of 8000Hz, three user-defined dip switches and LEDs (light-emitting diodes), and the accompanying software CCS (Code Composer Studio). Programming of the DSP is carried out in CCS, which includes a compiler and various functions for debugging. The compiled code is loaded to the DSP and run from CCS. The DSK board includes 16MB (megabytes) of SDRAM and 128kB (kilobytes) of flash ROM. The C6711 DSP has a clock frequency of 150MHz. For more information on the DSP and the starter kit, see [1]. In addition to the onboard codec, which is of limited use, TI provides an audio daughter card, PCM3003, which gives up to 73kHz sampling frequency.

In this introductory guide, the aim is to provide the reader with enough knowledge in order to quickly get started with programming his/her own applications. We will not discuss programming in detail, nor optimization or assembly language coding. Instead, the focus is on getting working (and readable) code up and running as quickly as possible. The reader is assumed to have basic working knowledge of C programming and digital signal processing, but little or no experience with DSP programming.

The majority of the code described here has been adapted from the examples provided by Chassaing in his book [1]. Some of the code has been built on previous work by Pär Dahlman.

2 Overview of the code library

The following application programs are provided with this guide:

- **loop_8k**, **loop_PCM3003** – input and output to the onboard codec and the audio daughter card, respectively
- **FIR_PCM3003** – General FIR program using circular buffers and the daughter card
- **FIR2_8k**, **FIR2_PCM3003** – General FIR program optimized with assembler routines to facilitate longer filters (using the onboard codec and the audio daughter card, respectively)
- **IIR_adjustable_8k**, **IIR_adjustable** – Simple second order IIR filter with possibility to set and adjust pole/zero locations in real-time (using the onboard codec and the audio daughter card, respectively)
- **IIR_8k**, **IIR** – General IIR filtering (using the onboard codec and the audio daughter card, respectively).

The programs are located in a directory with their respective names as given above. In CCS, a *project* is always created for each application. For instance, in order to open the **loop_PCM3003** project, open the project file **loop_PCM3003.pjt** in the **loop_PCM3003** library. In the leftmost window of CCS, all the files that are used in the project are listed. (Notice that if your installation of CCS is not equivalent to the one used by this author, CCS will ask you to locate certain files. More on this in the following section.)

3 Common files that are used in all projects

For initializing the DSK and the input/output devices, a number of files are used in all projects described here. There are three sets of files, one that is common to all projects, one for use with the onboard AD535 codec, and one for use with the PCM3003 audio daughter card. We will mainly work with the PCM3003, as the onboard codec has proven somewhat bug prone. It has turned out that the code written for the AD535 codec does not work appropriately with all DSK cards. Evidently there is some low-level bug in the DSK's interaction with the onboard codec.

The following source files are used in all projects:

- **vectors_11.asm** – defines interrupt mappings
- **c6xdsk.cmd** – linker command file (determines where code sections are located in memory)
- **c6xdsk.h** – definitions for the DSK board, registers, etc.
- **c6xinterrupts.h** – interrupt definitions.

The files that are common to all projects using the PCM3003 audio daughter card are the following:

- **c6xdskinit6_pcm.c** – initializes the DSK board for use with the PCM3003 audio card
- **c6xdskinit6_pcm.h** – corresponding header file.

Finally, the following files are used in all projects using the onboard codec:

- **c6xdskinit6.c** – initializes the DSK board for use with the onboard codec
- **c6xdskinit6.h** – corresponding header file.

All of these files are located in the **common files** directory. In addition to these source files, if your application involves using the LEDs or the DIP switches, you need to install a library called BSL (Board Support Library) which provides easy access to onboard peripherals. More information regarding this library is given in a later section.

4 Creating a new project

In many cases the easiest way of creating your own project is to copy one of the projects accompanying this guide and modify it to suit your application. On opening an existing project which has been copied from one computer to another, you will typically be asked to provide a few file locations which are different from the original computer setting.

The required files reside in your CCS home directory, which typically is **C:/ti/**. This may however be different on your computer. Locate this directory, and, in the following, denote it as **TI_DIR**.

You will at least need to specify the location of the following library file:

- **rts6701.lib**, providing run-time support tools. This file is located in **TI_DIR/c6000/cgtools/lib/**.

You may also have to change certain include paths in the build options, which can be accessed under the Project menu.

For projects that involve the DSK's dip switches or LEDs, see the section on BSL below.

If instead you wish to create a new project, use the following steps.

1. Create a new folder with the name of the project
2. Copy all the required .c, .asm, .cmd, and .h files as listed above from **common files** to your new directory
3. In CCS, create a new project (**Project** → **New**). Give it the name you just specified and make sure that you use the project directory you just created.
4. Add the necessary files to the project by clicking **Project** → **Add Files to Project**. Add the .c, .asm and .cmd files that you copied into the project directory (the .h files need not be added at this stage). Also, include the runtime support library **TI_DIR/c6000/cgtools/lib/rts6701.lib**
5. Include .h files by clicking **Project** → **Scan All Dependencies**
6. Write your code!

The code is compiled by running **Project** → **Rebuild All** and loaded to the DSK board by **File** → **Load Program**. Run it by clicking **Debug** → **Run**.

In some cases you may want to change the build options, which can be found under the menu choice **Project** → **Build Options...**. A typical change may be to specify some additional search path for the preprocessor or the linker.

4.1 Using BSL for accessing dip switches and LEDs

If your project makes use of dip switches and/or LEDs, as in **loop_8k** you need to install the Board Support Library (BSL), a preliminary version of which is included as a zip-file in the **common files** directory. Unpack this file in the directory (which you have to create) **TI_DIR/c6000/BSL_preliminary**. A manual which describes BSL can be found in **TI_DIR/docs/pdf/spru432a.pdf**.

Add the BSL library file **bsl6711dsk.lib** located in
TI_DIR/c6000/BSL_preliminary/preliminary_bsl/lib,
and the chip support library (CSL) file **TI_DIR/c6000/bios/lib/csl6711.lib** to the project via **Project → Add Files to Project**.

Additionally, you must tell the compiler which chip-set and board you are using, as well as the search path to then BSL include directory. This is done by adding the following to the compile instructions (in **Project → Build Options...**):
-i"C:\ti\c6000\BSL_preliminary\preliminary_bsl\include"
-d"CHIP_6711" -d"BOARD_6711DSK"
assuming that your TI home directory is **C:\ti**. You may also have to change the include path for the linker (accessible from the build options dialog) to point to the BSL include directory.

For information on how to use dip switches and LEDs within your program, see the section describing **loop_8k**.

4.2 Using assembly language functions from DSPLib

DSPLib is a collection of hand-optimized assembly routines provided by Texas Instruments. A manual can be found in **TI_DIR/docs/pdf/spru402.pdf**. The library includes routines for FIR and IIR filtering, FFT, adaptive filters, correlation, matrix manipulations, etc. In the **FIR2_PCM3003** and the **FIR2_8k** projects we show how to use a general FIR filter routine from **DSPLib**.

In order to use **DSPlib** it must first be installed. Check if there is a directory called **TI_DIR/c6000/dsplib** on your computer. If the directory is not there, follow the manual's instructions on installation.

Once installed, you must add the library file
TI_DIR/c6000/dsplib/dsplib62x.lib
to your project (**Project → Add Files to Project**). Then, in your .c-file include the header file for the function you intend to use. For an example, see the **FIR2_PCM3003** project.

It can be noted that the filtering routines provided with **DSPlib** are generally block based and may therefore not always be practical to use.

4.3 Introduction to debugging

CCS provides a number of debugging utilities. Some of them are accessible from the Debug menu, others can be found using the Profiler menu. The most useful ones are often the ones found under the View menu. The View menu provides a number of ways to access and monitor variables during run-time. In the construction of the code for this guide, the author mainly used simple debugging methods such as

- **View → Quick Watch** – lets you specify a variable and see its value. Right-click on the Watch window to refresh the value of the variable
- **View → Graph → Time/Frequency...** – for plotting arrays in time or frequency. Specify the variable name as the starting address
- .GEL functions – GEL (General Extension Language) is a C-like language that lets you change variables in real-time. Various functions, such as sliders, dialogs and hot menus are available. This can also be used to provide a user interface to the DSK. For more information and an example see the comments in the section on the **IIR_adjustable** project.

Other methods for debugging are described in [1] and the CCS documentation.

5 Introduction to the accompanying projects

The main programs are listed in order of occurrence in the last section of this guide. The following sections give a short introduction to the various projects.

It should be emphasized that the projects that use the onboard codec do not run successfully on all DSK boards. The reason for this is unknown, but believed to be due to a bug on older boards. The problem is entirely related to the codec. The simplest way of avoiding these problems is to use the audio daughter card instead.

Notice that when you adapt a given project to your own application, you should never need to perform any changes in any of the common support files listed in 4. The only changes required are in the main .c source file which typically bears the same name as the project.

The main .c file follows the same pattern in all projects. It consists of two functions:

- A `main()` function which sets up interrupts and communication to the AD/DA, and starts an infinite loop
- The interrupt service routine (ISR in TI jargon) `c\_int11()` which reads the input from the ADC and writes the output to the DAC.

Some of the projects make use of Matlab scripts for saving filters in appropriate formats. They can be found in the **Matlab** directory provided with this guide.

5.1 The loop_PCM3003 project

This project simply copies data from the input to the output of the audio daughter card PCM3003. The function

```
c_int11()
```

is the interrupt service routine. The functions

```
input_sample()
```

and

```
output_sample(unsigned int)
```

reads from the input and writes to the output respectively. The left and right inputs are 15-bit integer numbers defined as short variables (the last bit is not available to the user but used for control purposes). The input and output samples are stored in the variables

```
leftIn
leftOut
rightIn
rightOut
```

which are combined into two unsigned integers (one for the input, one for the output) in the data structures

```
inData
outData
```

Check the source code in **loop_PCM3003.c** to see how the program works. In adapting the code so as to suit your application, you should typically only need to apply changes in this source file, which in its basic form only consists of the `main()` function and the interrupt service routine `c_int11()`.

The sampling frequency of the PCM3003 AD/DA can be set by adjusting the variable `Fs`. The actual sampling frequency that you obtain differs somewhat from the one you specify. Consult Appendix F in [1] for a sampling rate conversion table.

5.2 The FIR_PCM3003 project

This project implements a simple FIR filter routine using circular buffers and an assembler routine for the filter loop. It works on a sample-by-sample basis. For optimized performance, use the block-based **FIR2_PCM3003** project which is capable of running longer filters. The reason for including the **FIR_PCM3003** project in this guide is that it provides sample-by-sample filtering which may be needed in certain applications. In all other applications, use the **FIR2_PCM3003** instead.

The **FIR_PCM3003** project builds on the **loop_PCM3003** project but includes a FIR processing stage. Each channel has its own assembly language FIR routine, since these routines keep the delayed samples in an internal buffer which is not seen from the main .c file. The FIR filter is specified in an include file which can be constructed from Matlab using the function `saveDSPfilterheader(h,filename)` where `h` is a vector of filter coefficients, and `filename` is the name of the filter file as a string.

5.3 The FIR2_PCM3003 project

This project implements a FIR filter routine using an assembly language block-based filter routine from **DSPlib** (cf. Section 4.2). With this program, filters of length up to 600 taps can be used at a sampling rate of 44.1kHz. The filtering routine is block-based which means that a block of `NUM_Y/2` samples are generated at a time, using `NUM_H+NUM_Y/2-1` input samples (where `NUM_H` is the number of taps in the filter) in each processing stage.

The filter coefficients are read from an include file which can be generated from Matlab using `saveDSPfiltercoeffs(h,filename)`. Notice that this project assumes a different file format than the **FIR_PCM3003** project. The filter length is specified in the main .c file (`NUM_H`), as is the block length (`NUM_Y/2`).

In this project, compiler optimization level `-o2` is specified in the build options. The compiler optimization levels can sometimes be used to increase the execution speed, but a word of caution is needed; The author, in constructing an FFT-based FIR filtering routine, attempted to use level `-o3` optimization to boost performance. It resulted in the compiler omitting the entire filtering stage! So the reader is cautioned to use the compiler optimization options with care.

5.4 The IIR project

This project provides a simple IIR filter routine for the PCM3003 audio daughter card. The IIR filter is implemented as a Direct Form II structure, as a series of cascaded second order IIR filters. From Matlab, an IIR filter in the form given by **SPtool** can be transformed to cascaded second order stages and saved in a format suited for this project by using the Matlab function `saveDSPIIRfilter(filtstruct,filename)`. The name of the filter include file is specified in the main .c file.

The program follows the **loop_PCM3003** structure with the addition of a C-loop performing the IIR filtering.

The project is quite sensitive to input levels. If the output level includes hissing noise or is noise contaminated in other ways, this suggests that the input level is too high, yielding overflow on the output. Avoid this by changing the level of the signal fed to the input. (Alternatively, change the scaling in the program.)

5.5 The IIR_adjustable project

This is an example of the use of a GEL file to manipulate parameters used by the application in real-time. The example uses a simple one-stage Direct Form II IIR filter, with two symmetrically placed poles and zeros. The code is entirely similar to the **IIR** project. However, by using a so-called GEL-file (GEL = General Extension Language) we can change the pole and zero locations during run-time. In the project directory, there is a GEL file called `poleszerosPolar.gel` (there are other similar files as well). Open the file and see how it is used to change the location of the poles and zeros. A GEL file is run by first loading it (**File** → **Load GEL...**) and then running it from the GEL menu.

There are other examples of GEL files in this and the other projects. For instance, there is a GEL program that turns on and off filtering in the FIR projects. This is particularly useful as a debugging tool. Refer to the comments in the .c source files to see other GEL examples.

5.6 The loop_8k project

This project simply copies data from the input to the output of the onboard codec (AD535). The sampling frequency of the AD535 is fixed at $f_s = 8000$Hz. Notice that the codec is mono only.

The program also shows how to use the three onboard user-defined dip switches and the three LEDs. This is done by using the Board Support Library described above. In order to use the functions you must include three header files in your .c program:

```
#include "bsl.h"
#include "bsl_led.h"
#include "bsl_dip.h"
```

BSL is first initialized in the main function by calling

```
BSL_init()
```

Then, for instance, if you want to check the first dip switch simply call

```
DIP_get(DIP_1)
```

which returns 1 if the switch is up, or 0 if it is down. A LED (the first LED in this example) can be lit by calling

```
LED_on(LED_1)
```

and turned off by a similar call to

```
LED_off(LED_1)
```

The LED can be toggled on/off by using

```
LED_toggle(LED_1)
```

The BSL piece of the code can easily be copied from this project to another. The only file to make changes in is the main .c file. The support files do not require any changes.

5.7 The FIR2_8k, IIR_8k, and IIR_adjustable_8k projects

These projects are equivalent to their respective PCM3003 counterparts described above. The difference is only that these projects use the onboard 8kHz mono AD535 codec.

loop_PCM3003 .c source

```
// loop_PCM3003.c
// Basic program which copies data from the input of the PCM3003
// audio daughter card to the output.
//
//#include <stdio.h>
#define LEFT  1
#define RIGHT 0
short   leftIn   = 0;
short   rightIn  = 0;
short   leftOut  = 0;
short   rightOut = 0;

float Fs = 44100.0; // set sampling frequency (irrelevant if jumper 5 in 3-4)

volatile union{unsigned int uint; short channelIn[2];} inData;
volatile union{unsigned int uint; short channelOut[2];} outData;

interrupt void c_int11()      //interrupt service routine
{
leftOut  = 0;
rightOut = 0;
inData.uint = input_sample();
leftIn  = inData.channelIn[LEFT]; // LEFT = 1, RIGHT = 0
rightIn = inData.channelIn[RIGHT];

// Insert signal processing here!


leftOut  = leftIn;
rightOut = rightIn;
//printf("%d\n",leftOut);
outData.channelOut[LEFT]  = leftOut;
outData.channelOut[RIGHT] = rightOut;
output_sample(outData.uint);

return; //return from interrupt
}

void main()
{
comm_intr();             //init DSK, codec, McBSP
  while(1);                //infinite loop
}
```

FIR_PCM3003 .c source

```
// FIR_PCM3003.c   Not very fast FIR-filter routine.
// Use the FIR2_PCM3003 directory instead for a more efficient implementation.
//
// FIR-filters the input to the PCM3003 codec
// and outputs the filtered signal.
// The filter coefficients are stored in the file "filter.cof"
//
// The filter length can be 341 in stereo. Probably not much more.
// 381 does not work.
//
// The gain of the filter should be about 10 to produce
// reasonable output power. If there is no audible sound
// coming from the output, increase the gain of the filter.
//
//#include <stdio.h>
#include "filter.cof"
#define LEFT  1
#define RIGHT 0
short   leftIn   = 0;
short   rightIn  = 0;
int     leftOut  = 0;
int     rightOut = 0;

float Fs = 44100.0; // set sampling frequency (irrelevant if jumper 5 in 3-4)
volatile union{unsigned int uint; short channelIn[2];} inData;
volatile union{unsigned int uint; short channelOut[2];} outData;

interrupt void c_int11()      //interrupt service routine
{
```

```
leftOut  = 0;
rightOut = 0;
inData.uint = input_sample();
leftIn  = inData.channelIn[LEFT]; // LEFT = 1, RIGHT = 0
rightIn = inData.channelIn[RIGHT];

// Filter the input with filter <h> of length <N>
leftOut = fircircfuncleft(leftIn, h, N);
rightOut = fircircfuncright(rightIn, h, N);

//printf("%d\n",leftOut);
outData.channelOut[LEFT] = leftOut;
outData.channelOut[RIGHT] = rightOut;
output_sample(outData.uint);

return; //return from interrupt
}

void main()
{
comm_intr();            //init DSK, codec, McBSP
  while(1);               //infinite loop
}
```

FIR2_PCM3003 .c source

```
// FIR_PCM3003.c
// FIR-filters the input to the PCM3003 codec
// and outputs the filtered signal.
// The filter coefficients are stored in an include file
// which can be generated from Matlab with the function "saveDSPfilter.m".
//
// Notes:
// 1. The filter length is specified below by the constant NUM_H.
// 2. The filter length can be about 600 in stereo at fs=44100 Hz.
// 3. The filter file name is specified below in the variable g[NUM_H].
// 4. The FIR routine uses the fir_gen function from DSPLIB which must
//    be installed for the program to function.
// 5. The implementation is block-based. A block of NUM_Y data is filtered
//    every NUM_Y:th input interrupt.
// 6. A filter with gain 1 has a gain of 1 also when run in the DSP. This means
//    that a filter with gain higher than one may result in overflow.
// 7. The output level can be manipulated via the GEL-file
//    File->Load GEL...->filtergain.gel (then GEL->Filtergain)
// 8. The filter can be swithed on/off by loading the GEL-file
//    filterOnOff.gel
// 9. The sampling frequency is set by adjusting Fs below.
//

//#include <stdio.h>
#include <fir_gen.h>

#define LEFT  1
#define RIGHT 0
#define NUM_H 600 // Length of the FIR filter.
#define NUM_Y 32 // Block length

/* Align the arrays in memory */
#pragma DATA_ALIGN (h, 8);
#pragma DATA_ALIGN (x1, 8);
#pragma DATA_ALIGN (y1, 8);
#pragma DATA_ALIGN (x2, 8);
#pragma DATA_ALIGN (y2, 8);

/* Read the coefficients from file */
float g[NUM_H] =
{
//#include "dali_p.h"
//#include "reconst.h"
//#include "qbook.h"
//#include "one.h"
#include "ken.h"
//#include "coeffs.h"
//#include "dali_polk_900.h"
//#include "sv_rakt_152.h"
//#include "sv.h"
//#include "LP_64taps_fc1500.h"
};

short h[NUM_H];

short gain=1;

short x1[NUM_H + (NUM_Y/2) - 1];

short y1[NUM_Y/2];

short x2[NUM_H + (NUM_Y/2) - 1];

short y2[NUM_Y/2];
```

```
int     buffercount = 0;
short   inbuffer1[NUM_Y/2];
short   inbuffer2[NUM_Y/2];
short   outbuffer1[NUM_Y/2];
short   outbuffer2[NUM_Y/2];
short   leftIn   = 0;
short   rightIn  = 0;
int     leftOut  = 0;
int     rightOut = 0;
int     f = 1;
float Fs = 44100.0; // set sampling frequency (irrelevant if jumper 5 in 3-4)
volatile union{unsigned int uint; short channelIn[2];} inData;
volatile union{unsigned int uint; short channelOut[2];} outData;

interrupt void c_int11()      //interrupt service routine
{
    int temp = 0;
int i = 0;

inData.uint = input_sample();
inbuffer1[buffercount] = inData.channelIn[LEFT];
inbuffer2[buffercount] = inData.channelIn[RIGHT];
outData.channelOut[LEFT] = gain*(outbuffer1[buffercount]);
outData.channelOut[RIGHT] = gain*(outbuffer2[buffercount]);
output_sample(outData.uint);

buffercount++;
if (buffercount > NUM_Y/2-1)
{
buffercount = 0;
if(f)
{
//LED_on(1);
// Arrange the input arrays
     #pragma MUST_ITERATE(NUM_H-1);
     for (i=(NUM_Y/2);i<NUM_H+(NUM_Y/2)-1;i++)
     {
     temp = i -(NUM_Y/2);
     x1[temp] = x1[i];
     x2[temp] = x2[i];
     }
     #pragma MUST_ITERATE(NUM_Y/2);
      for (i=NUM_H-1;i<(NUM_Y/2)+NUM_H-1;i++)
     {
```

```
     temp  = i-NUM_H+1;
     x1[i] = inbuffer1[temp];
     x2[i] = inbuffer2[temp];

     }

     // filter the input data
         fir_gen(x1, h, y1, NUM_H, NUM_Y/2);
fir_gen(x2, h, y2, NUM_H, NUM_Y/2);

// copy filtered data to output buffer
#pragma MUST_ITERATE(NUM_Y/2);
for (i=0;i<NUM_Y/2;i++)
{
outbuffer1[i] = y1[i];
outbuffer2[i] = y2[i];
}
}
else
{
//LED_off(1);
// copy input data to output buffer

#pragma MUST_ITERATE(NUM_Y);
for (i=0;i<NUM_Y/2;i++)
{
outbuffer1[i] = inbuffer1[i];
outbuffer2[i] = inbuffer2[i];
}
}
} // end if(buffercount>NUM_Y/2-1)

return; //return from interrupt
}

void main()
{
int j=0;
comm_intr();            //init DSK, codec, McBSP
// Init filter
for (j=0;j<NUM_H;j++)
{
h[j] = (short)(32768*g[j]);
}
   while(1);               //infinite loop
}
```

IIR .c source

```
// IIR.c  IIR filter using cascaded Direct Form II
// Filters of arbitrary order can be created in Matlab using a script called
// saveDSPIIRFilter.m (included in this code library)
//
//
// Notes:
// 1. The filter coefficient file is specified in the second include line below.
//    Either specify a new file name, or save yours as the one specified.
// 2. The input signal power level should be rather low to avoid overflow.
//    If the output is contaminated with noise, adjust the volume
//    of the input signal.
// 3. The PCM3003 audio daughter card is used in this program. The
//    sampling frequency can be set by adjusting the value of Fs below.
// 4. The coefficients of the first stage (only) can be adjusted in real-time
//    using either of:
//      poleszeros.gel       = set pole/zero location in rectangular coordinates
//      poleszerosPolar.gel  = same but polar coordinates
//      filterparameters.gel = set a0,a1,a2,b0,b1 directly
//    These functions are enabled in the GEL-menu in CCS by performing:
//      File->Load GEL... (choose desired .gel file in the IIR_adjustable directory)
// 5. The pole/zero placement GEL files always assume symmetrical locations,
//    even when the specified location is on the real line, ie. it is not
//    possible to place only one pole or zero on the real line; you
//    automatically get two.
//
// Mathias Johansson 2003.

//#include <stdio.h>
#include <math.h>
#include "iir2.cof"         //Lowpass filter (fco=2205 Hz) coefficient file
#define LEFT  1
#define RIGHT 0
short   leftIn   = 0;
short   rightIn  = 0;
short   leftOut  = 0;
short   rightOut = 0;
int     poleflag = 0;
int     zeroflag = 0;
float   zero_angle = 0;
float   zero_mag   = 0;
float   pole_angle = 0;
float   pole_mag   = 0;

float Fs = 44100.0; // set sampling frequency (irrelevant if jumper 5 in 3-4)

volatile union{unsigned int uint; short channelIn[2];} inData;
volatile union{unsigned int uint; short channelOut[2];} outData;

short dly1[stages][2] = {0};    //delay samples per stage (left channel)
short dly2[stages][2] = {0};    //delay samples per stage (right channel)

interrupt void c_int11()  //ISR
{
 int i, input1, input2;
 int un1, yn1, un2, yn2;

 inData.uint = input_sample();
 input1 = inData.channelIn[LEFT];
 input2 = inData.channelIn[RIGHT];

 for (i = 0; i < stages; i++)  //repeat for each stage
  {

   // Left channel
   un1=input1-((a[i][0]*dly1[i][0])>>15) - ((a[i][1]*dly1[i][1])>>15);

   yn1=((b[i][0]*un1)>>15)+((b[i][1]*dly1[i][0])>>15)+((b[i][2]*dly1[i][1])>>15);

   dly1[i][1] = dly1[i][0];   //update delays
   dly1[i][0] = un1;             //update delays
   input1 = yn1;    //intermediate output->input to next stage

   // Right channel
   un2=input2-((a[i][0]*dly2[i][0])>>15) - ((a[i][1]*dly2[i][1])>>15);

   yn2=((b[i][0]*un2)>>15)+((b[i][1]*dly2[i][0])>>15)+((b[i][2]*dly2[i][1])>>15);

   dly2[i][1] = dly2[i][0];   //update delays
   dly2[i][0] = un2;             //update delays
   input2 = yn2;    //intermediate output->input to next stage

  }
   //printf("%d\n",yn);
    outData.channelOut[LEFT] = yn1;
outData.channelOut[RIGHT] = yn2;
output_sample(outData.uint);
```

```
   return;  //return from ISR
}

void main()
{
  float zero_r = 0.0;
  float zero_i = 0.0;
  float b1 = 0.0;
  float b2 = 0.0;
  float pole_r = 0.0;
  float pole_i = 0.0;
  float a1 = 0.0;
  float a2 = 0.0;

  comm_intr();                 //init DSK, codec, McBSP

  while(1)  //infinite loop
  {
   if (zeroflag) // set by the GEL file
   {
   zeroflag=0;
   zero_r = zero_mag*cos(zero_angle);
   zero_i = zero_mag*sin(zero_angle);
   b1 = zero_r*2;
b2 = zero_r*zero_r + zero_i*zero_i;
b[0][0] = (int)(32768);
b[0][1] = (int)(b1*32768);
b[0][2] = (int)(b2*32768);
   }
   if (poleflag) // set by the GEL file
   {
   poleflag=0;
   pole_r = pole_mag*cos(pole_angle);
   pole_i = pole_mag*sin(pole_angle);
   a1 = pole_r*2;
a2 = pole_r*pole_r + pole_i*pole_i;
a[0][0] = (int)(a1*32768);
a[0][1] = (int)(a2*32768);
   }

  }
}
```

IIR_adjustable .c source

```
// IIRadjustable.c  IIR filter using cascaded Direct Form II.
// Second order filter with symmetrical poles/zeros
// The coefficients can be adjusted in real-time using either of:
//   poleszeros.gel       = set pole/zero location in rectangular coordinates
//   poleszerosPolar.gel  = same but polar coordinates
//   filterparameters.gel = set a0,a1,a2,b0,b1 directly
// These functions are enabled in the GEL-menu in CCS by performing:
//   File->Load GEL... (choose desired .gel file in the IIR_adjustable directory)
//
// The filter is initialized as an all-pass filter.
//
// Notes:
// 1. The input signal power level should be rather low to avoid overflow.
//    If the output is contaminated with noise, adjust the volume
//    of the input signal.
// 2. The PCM3003 audio daughter card is used in this program. The
//    sampling frequency can be set by adjusting the value of Fs below.
// 3. The pole/zero placement GEL files always assume symmetrical locations,
//    even when the specified location is on the real line, ie. it is not
//    possible to place only one pole or zero on the real line; you
//    automatically get two.
//
// Mathias Johansson 2003.

//#include <stdio.h>
#include <math.h>
#include "allpass.cof"          //All-pass filter coefficient file
#define LEFT  1
#define RIGHT 0
short   leftIn   = 0;
short   rightIn  = 0;
short   leftOut  = 0;
short   rightOut = 0;
int     poleflag = 0;
int     zeroflag = 0;
float   zero_angle = 0;
float   zero_mag   = 0;
float   pole_angle = 0;
float   pole_mag   = 0;

float Fs = 44100.0; // set sampling frequency (irrelevant if jumper 5 in 3-4)

volatile union{unsigned int uint; short channelIn[2];} inData;
volatile union{unsigned int uint; short channelOut[2];} outData;
```

```
short dly1[stages][2] = {0};    //delay samples per stage (left channel)
short dly2[stages][2] = {0};    //delay samples per stage (right channel)

int f = 1;

interrupt void c_int11()  //ISR
{
 int i, input1, input2;
 int un1, yn1, un2, yn2;

 inData.uint = input_sample();
 input1 = inData.channelIn[LEFT];
 input2 = inData.channelIn[RIGHT];

 if (f) {
 for (i = 0; i < stages; i++)  //repeat for each stage
  {
   // Left channel
   un1=input1-((a[i][0]*dly1[i][0])>>15) - ((a[i][1]*dly1[i][1])>>15);

   yn1=((b[i][0]*un1)>>15)+((b[i][1]*dly1[i][0])>>15)+((b[i][2]*dly1[i][1])>>15);

   dly1[i][1] = dly1[i][0];   //update delays
   dly1[i][0] = un1;             //update delays
   input1 = yn1;    //intermediate output->input to next stage

   // Right channel
   un2=input2-((a[i][0]*dly2[i][0])>>15) - ((a[i][1]*dly2[i][1])>>15);

   yn2=((b[i][0]*un2)>>15)+((b[i][1]*dly2[i][0])>>15)+((b[i][2]*dly2[i][1])>>15);

   dly2[i][1] = dly2[i][0];   //update delays
   dly2[i][0] = un2;             //update delays
   input2 = yn2;    //intermediate output->input to next stage

  }
  } else
  {
   yn1 = input1;
   yn2 = input2;
  }
  //printf("%d\n",yn);
  outData.channelOut[LEFT] = yn1;
  outData.channelOut[RIGHT] = yn2;
  output_sample(outData.uint);
  return;  //return from ISR
}

void main()
{
  float zero_r = 0.0;
  float zero_i = 0.0;
  float b1 = 0.0;
  float b2 = 0.0;
  float pole_r = 0.0;
  float pole_i = 0.0;
  float a1 = 0.0;
  float a2 = 0.0;

  comm_intr();                 //init DSK, codec, McBSP

  while(1)  //infinite loop
  {
   if (zeroflag) // the flag is set from poleszerosPolar.gel
   {
   zeroflag=0;
   zero_r = zero_mag*cos(zero_angle);
   zero_i = zero_mag*sin(zero_angle);
   b1 = zero_r*2;
b2 = zero_r*zero_r + zero_i*zero_i;
b[0][0] = (int)(32768);
b[0][1] = (int)(b1*32768);
b[0][2] = (int)(b2*32768);
   }
   if (poleflag) // the flag is set from poleszerosPolar.gel
   {
   poleflag=0;
   pole_r = pole_mag*cos(pole_angle);
   pole_i = pole_mag*sin(pole_angle);
   a1 = pole_r*2;
a2 = pole_r*pole_r + pole_i*pole_i;
a[0][0] = (int)(a1*32768);
a[0][1] = (int)(a2*32768);
   }

  }
}
```

poleszerosPolar.GEL source

```
/* poleszerosPolar.gel */
// Lets user change poles/zeros of a 2nd order IIR filter in real-time

menuitem "Filter poles/zeros (polar coordinates)"

dialog ZeroLocation(zero_m "Magnitude", zero_a "Angle")
{
if (zero_m<0)
zero_mag = -zero_m;
else
zero_mag = zero_m;
zero_angle = zero_a;
zeroflag = 1;
}

dialog PoleLocation(pole_m "Magnitude", pole_a "Angle")
{
if (pole_m<0)
pole_mag = -pole_m;
else
pole_mag = pole_m;
pole_angle = pole_a;
poleflag = 1;
}




/*slider a0(0,20,1,1,a0val) { // From 0 to 20 by 1 = 0 to 2
bass_gain = ((float)bass_slider)/10;  }   // ->From 0.0 to 1.0 by 0.1
*/
```

loop_8k .c source

```
// loop_8k.c
// Basic program which copies data from the input of the
// onboard codec to the output.
// Also checks the user switches on the evaluation board
// and turns on/off the corresponding diodes.
// Notes:
// 1. There seems to be a bug in the program that causes the program to
//    output only silence while the switches still work. The bug occurs
//    when the board is first turned on and can be avoided by first
//    running some other program (such as IIR_8k). The bug probably
//    has something to do with the combined use of BSL and the codec.
// 2. The onboard codec is mono only.
// 3. The sampling frequency is 8000Hz.
// 4. BSL (Board Support Library) and CSL (Chip Support Library) is used
//    to provide easy access to the switches and the LEDs. BSL needs to
//    be installed on the computer. Also check the build options (both
//    compiler and linker options).
// 5. During compilation, you may obtain a warning for the flash macro. This
//    should not cause any problems.
//
// Mathias Johansson 2003.

//#include <stdio.h>
#include "bsl.h"
#include "bsl_led.h"
#include "bsl_dip.h"
#include "c6xdsk.h"

Uint32 dipval;

short input, output;

interrupt void c_int11()      //read interrupt service routine
{
    input = input_sample();

    // Insert signal processing here!

    output = input; // Just copy the input to the output

    //printf("%d\n",leftOut);

    output_sample(output);

    // Check the DIP switches and light corresponding LEDs
    dipval = DIP_get(DIP_1);
    if (dipval == 1)
    {
        LED_on(LED_1);
    } else
        LED_off(LED_1);

    dipval = DIP_get(DIP_2);
    if (dipval == 1)
    {
        LED_on(LED_2);
    } else
        LED_off(LED_2);

    dipval = DIP_get(DIP_3);
    if (dipval == 1)
    {
        LED_on(LED_3);
    } else
        LED_off(LED_3);

    return;         //return from interrupt
}

void main()
{
    //CSL_init(); // Should not be needed
    BSL_init();
    comm_intr();          //init DSK, codec, McBSP
    while(1);              //infinite loop
}
```

6 More information

This guide is deliberately kept short. For more information, the reader is referred to Chassaing's book [1] and the CCS documentation which covers in detail most aspects of programming the DSK.

References

[1] R. Chassaing, *DSP Applications Using C and the TMS320C6x DSK*, Wiley, 2002.