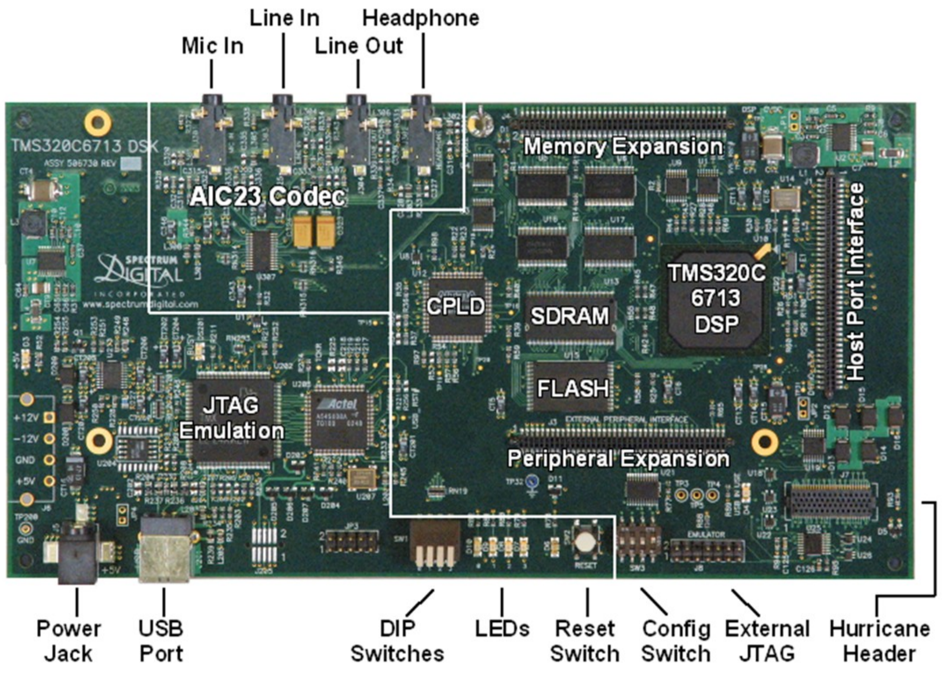
1. Familiarization of DSP Hardware and Software

TMS320C6713 DSP Starter Kit (DSK)

The C6713 DSK is a board that enables users to develop real-time DSP applications. The heart of the DSK is the Texas Instruments TMS320C6713 32-bit floating point Digital Signal Processor. DSP's differ from ordinary microprocessors in that they are specifically designed to rapidly perform the sum of products operation required in many discrete-time signal processing algorithms. They contain hardware parallel multipliers, and functions implemented by microcode in ordinary microprocessors are implemented by high-speed hardware in DSP's. Compared to fixed-point processors, floating-point processors are easier to program since issues like underflow, overflow, dynamic range etc can be ignored. The board is programmed using the TI Code Composer Studio (CCS) software, which connects to the board through a USB.



The DSK comes with a full complement of on-board devices that suit a wide variety of application environments. Key features include:

- A Texas Instruments TMS320C6713 DSP operating at 225 MHz.
- An AIC23 stereo codec
- 16 Mbytes of synchronous DRAM
- 512 Kbytes of non-volatile Flash memory (256 Kbytes usable in default configuration)
- 4 user accessible LEDs and DIP switches
- Software board configuration through registers implemented in CPLD
- Configurable boot options
- Standard expansion connectors for daughter card use
- JTAG emulation through on-board JTAG emulator with USB host interface or external emulator
- Single voltage power supply (+5V)

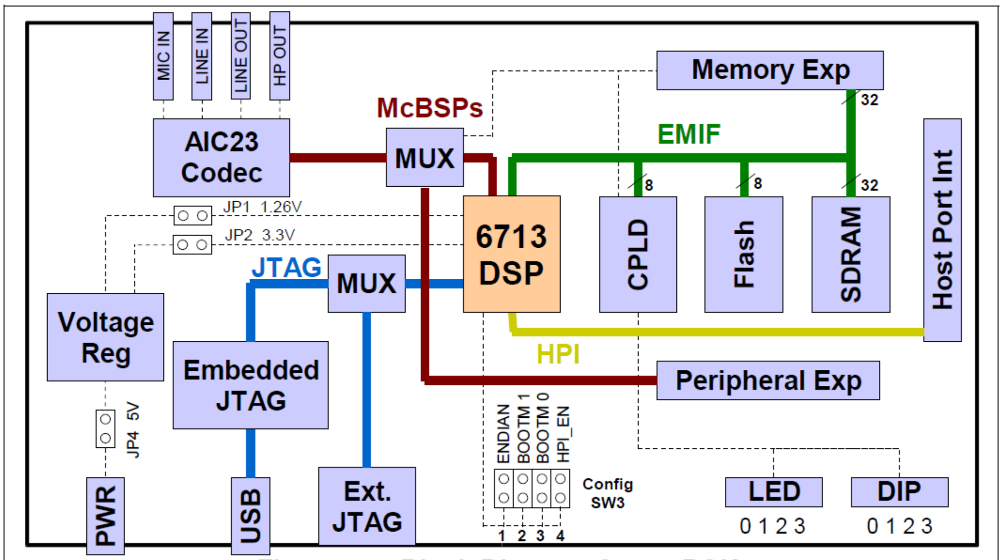


Figure 1-1, Block Diagram C6713 DSK

Functional Overview of the TMS320C6713 DSK

The DSP on the 6713 DSK interfaces to on-board peripherals through a 32-bit wide EMIF (External Memory InterFace). The SDRAM, Flash and CPLD are all connected to the bus. EMIF signals are also connected daughter card expansion connectors which are used for third party add-in boards.

The DSP interfaces to analog audio signals through an on-board AIC23 codec and four 3.5 mm audio jacks (microphone input, line input, line output, and headphone output). The codec can select the microphone or the line input as the active input. The analog output is driven to both the line out (fixed gain) and headphone (adjustable gain) connectors. Multichannel Buffered Serial Port 0 (McBSP0) is used to send commands to the codec control interface while McBSP1 is used for digital audio data. McBSP0 and McBSP1 can be re-routed to the expansion connectors in software.

A programmable logic device called a CPLD is used to implement glue logic that ties the board components together. The CPLD has a register-based user interface that lets the user configure the board by reading and writing to its registers.

The DSK includes 4 LEDs and a 4 position DIP switch as a simple way to provide the user with interactive feedback. Both are accessed by reading and writing to the CPLD registers.

A 5V external power supply is used to power the board. On-board switching voltage regulators provide the +1.26V DSP core voltage and +3.3V I/O supplies. The board is held in reset until these supplies are within operating specifications.

Code Composer communicates with the DSK through an embedded JTAG emulator with a USB host interface. The DSK can also be used with an external emulator through the external JTAG connector.

Memory Map

The C67xx family of DSPs has a large byte addressable address space. Program code and data can be placed anywhere in the unified address space. Addresses are always 32-bits wide. The memory map shows the address space of a generic 6713 processor on the left with specific details of how each region is used on the right. By default, the internal memory sits at the beginning of the address space. Portions of the internal memory can be reconfigured in software as L2 cache rather than fixed RAM. The EMIF has 4 separate addressable regions called chip enable spaces (CE0-CE3). The SDRAM occupies CE0 while the Flash and CPLD share CE1. CE2 and CE3 are generally reserved for daughtercards.

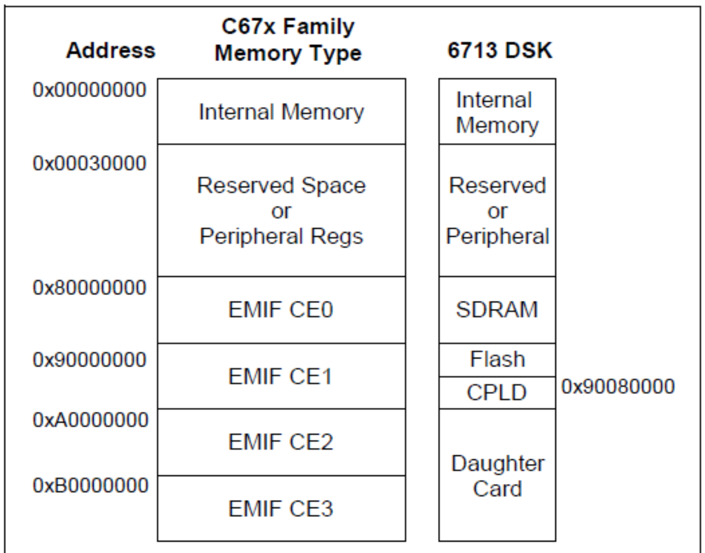


Figure 1-2, Memory Map, C6713 DSK

AIC23 Codec

The DSK uses a Texas Instruments AIC23 (part #TLV320AIC23) stereo codec for input and output of audio signals. The codec samples analog signals on the microphone or line inputs and converts them into digital data so it can be processed by the DSP. When the DSP is finished with the data it uses the codec to convert the samples back into analog signals on the line and headphone outputs so the user can hear the output.

The codec communicates using two serial channels, one to control the codec's internal configuration registers and one to send and receive digital audio samples. McBSP0 is used as the unidirectional control channel. It should be programmed to send a 16-bit control word to the AIC23 in SPI format. The top 7 bits of the control word should specify the register to be modified and the lower 9 should contain the register value. The control channel is only used when configuring the codec, it is generally idle when audio data is being transmitted, McBSP1 is used as the bi-directional data channel. All audio data flows through the data channel. Many data formats are supported based on the three variables of sample width, clock signal source and serial data format.

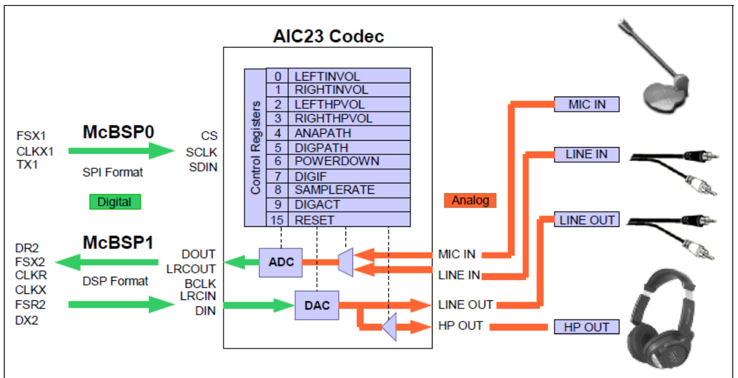


Figure 2-1, TMS320C6713 DSK CODEC INTERFACE

The codec has a 12MHz system clock. The 12MHz system clock corresponds to USB sample rate mode, named because many USB systems use a 12MHz clock and can use the same clock for both the codec and USB controller. The AIC23 can divide down the 12 MHz clock frequency to provide sampling rates of 8, 16, 24, 32, 44.1, 48 and 96 KHz.

Software

Texas Instruments’ Code Composer Studio (CCS) Integrated Development Environment (IDE) incorporates a C compiler, an assembler, and a linker. It is a development tool that allows users to create, edit and build programs, load them into the processor memory and monitor program execution. CCS communicates with the DSK via a USB connection. It supports real - time debugging has graphical capabilities. CCS is based on Eclipse, which is a Linux based open source software. CCSv7 and later does not require a paid license. The latest version is v11. But it does not support the debug probe on the 6713DSK. Older versions do not run on Windows 10. We will be using version 7.

A special Board Support Library (BSL) *dsk6713bsl32.lib* is supplied with the TMS320C6713 DSK. The BSL provides C-language functions for configuring and controlling all the on-board devices. The library includes modules for general board initialization, access to the AIC23 codec, reading the DIP switches, controlling the LED’s, and programming and erasing the Flash memory. TI also provides a Chip Support Library (CSL) *csl6713.lib* that contains C functions and macros for configuring and interfacing with all the ’C6713 on-chip peripherals. The pre-compiled board support and chip support libraries are provided to you in the *dsplab* folder. The C source code for BSL functions are also provided. The folder also contains the required header files for using BSL and CSL functions.

On power on, a power on self - test (POST) program, stored by default in the onboard flash memory, uses routines from the board support library (BSL) to test the DSK. It tests the internal, external, and flash memory, the two multichannel buffered serial ports (McBSP), DMA, the onboard codec, and the LEDs. If all tests are successful, all four LEDs blink three times and stop (with all LEDs on). During the testing of the codec, a 1kHz tone is generated for 1 second.

CCS Project

A very readable and useful user guide for CCS is available online at

https://software-dl.ti.com/ccs/esd/documents/users_guide/index.html

All work in CCS is based on projects, which are typically a collection of files and folders required for an application to be run on the DSK. Project folders are stored and organized in workspace folder. A workspace is the main working folder for CCS. When CCS is launched, it will prompt for the workspace folder location.

A Code Composer Studio project comprises all of the files (or links to all of the files) required in order to generate an executable file. A variety of options enabling files of different types to be added to or removed from a project are provided. In addition, a Code Composer Studio project contains information about exactly how files are to be used in order to generate an executable file. Compiler/linker options can be specified.

To create a new CCS project, follow the steps below:

Go to menu *Project → New CCS Project...* or *File → New → CCS Project*.

In the New CCS Project wizard:

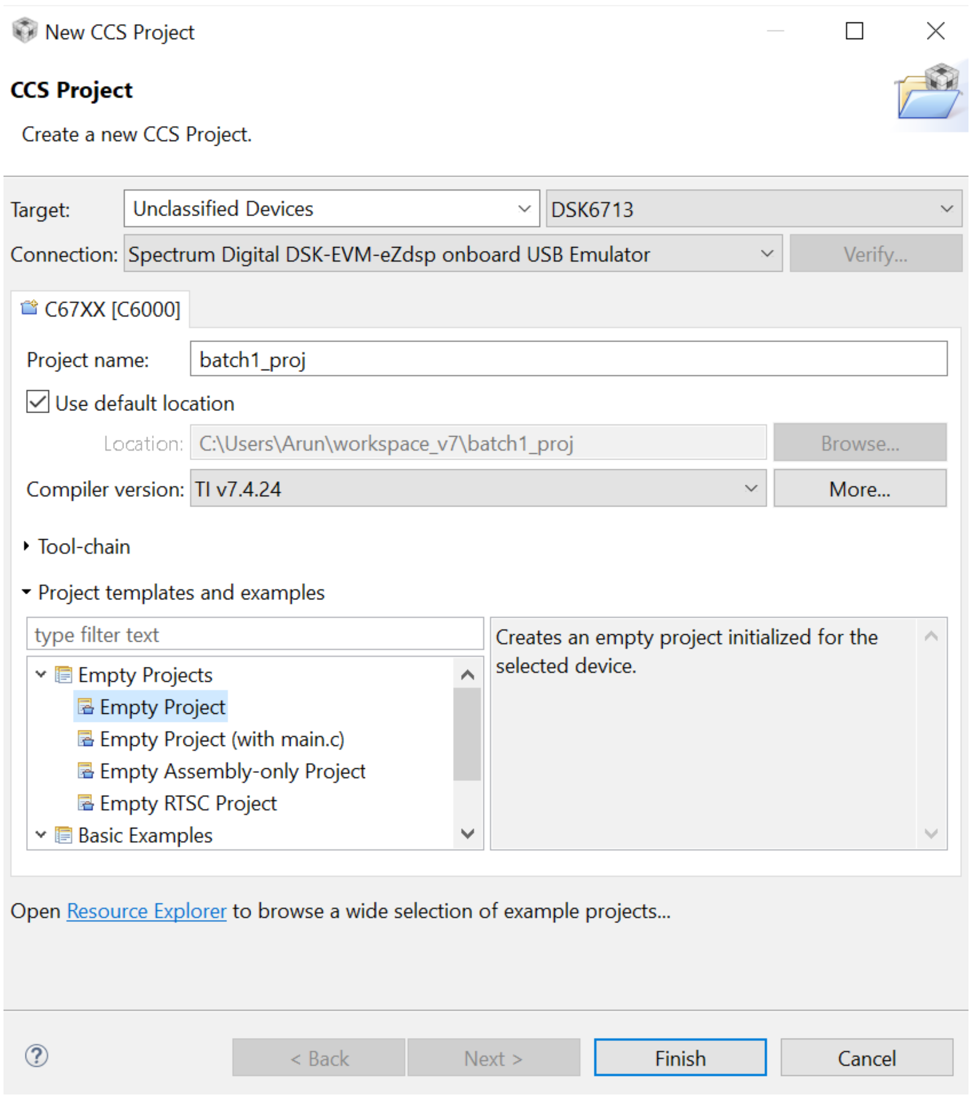
Type or select the *Target device*: Select *Unclassified Devices* and *DSK6713*

Connection: *Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator*

This automatically creates a Target Configuration File *DSK6713.ccxml*. The Target Configuration File is a plain text XML file, with a .ccxml extension, that contains all the necessary information for a debug session: the type of Debug Probe, the target board or device , and (optionally) a path to a GEL (General Extension Language) script, which is responsible for performing device and/or hardware initialization.

Project Name: Give your project a name, such as *batch1_proj*. The default location will be a folder with the name of your project within the *workspace_v7* folder

In project templates, select Empty project



Target: Unclassified Devices | DSK6713

Connection: Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator

C67XX [C6000]

Project name: batch1_proj

☑ Use default location

Location: C:\Users\Arun\workspace_v7\batch1_proj

Compiler version: TI v7.4.24

Creates an empty project initialized for the selected device.

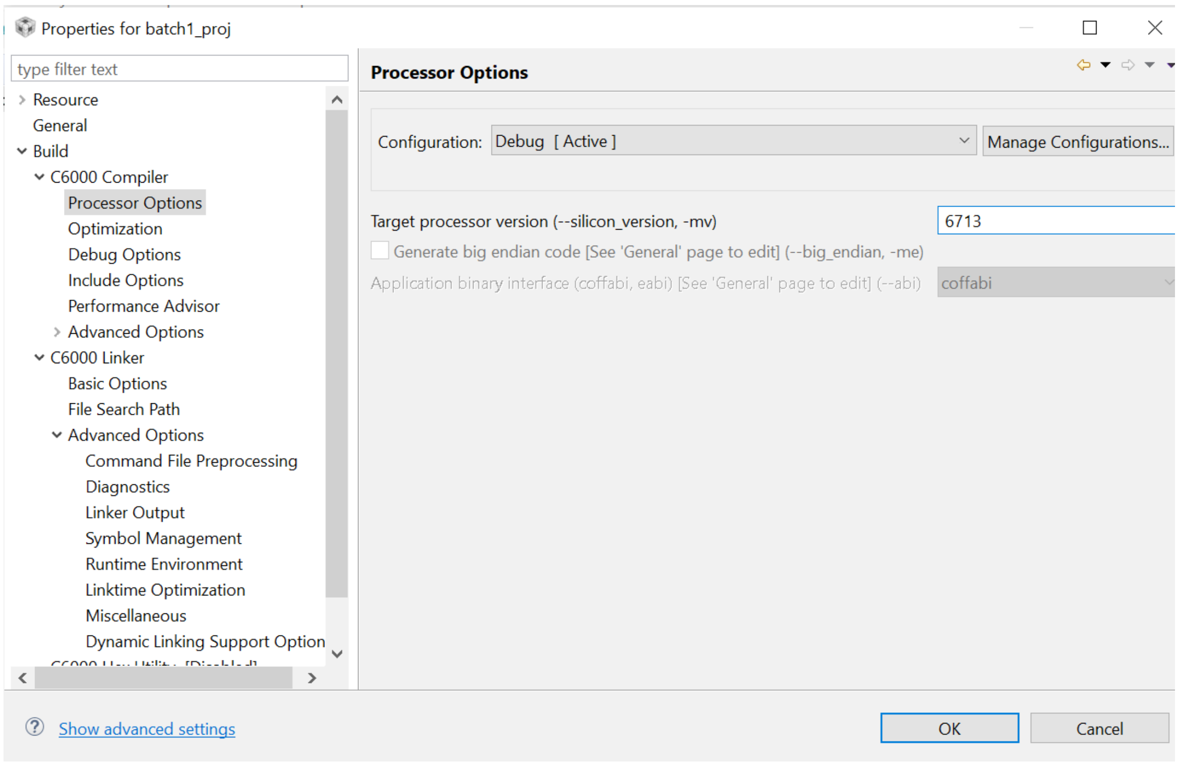
Open Resource Explorer to browse a wide selection of example projects...

Click Finish

Your project should now show in the Project Explorer window.

First, we need to set some properties for our project. Right-click on the project and go to Properties (Or from menu *Project > Properties*)

Under *Build>C6000 Compiler>Processor Options*, set *Target processor version* as 6713


Properties for batch1_proj

Processor Options

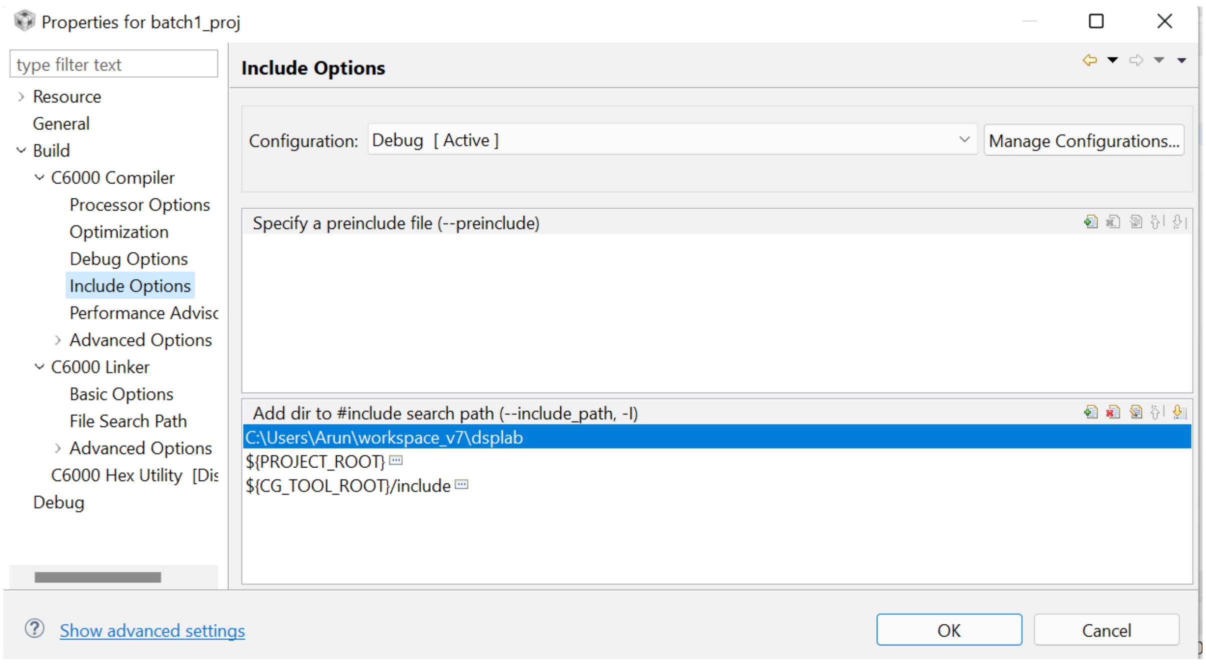
Configuration: Debug [Active]

Target processor version (--silicon_version, -mv): 6713

Generate big endian code [See 'General' page to edit] (--big_endian, -me)

Application binary interface (coffabi, eabi) [See 'General' page to edit] (--abi): coffabi

Under *Include Options*, go *to Add dir to #include search path*, click on the file icon with a green + mark and Browse to the folder *dsplab* provided to you and click OK. This folder contains the header files for the board support library and chip support library functions.


Properties for batch1_proj

Include Options

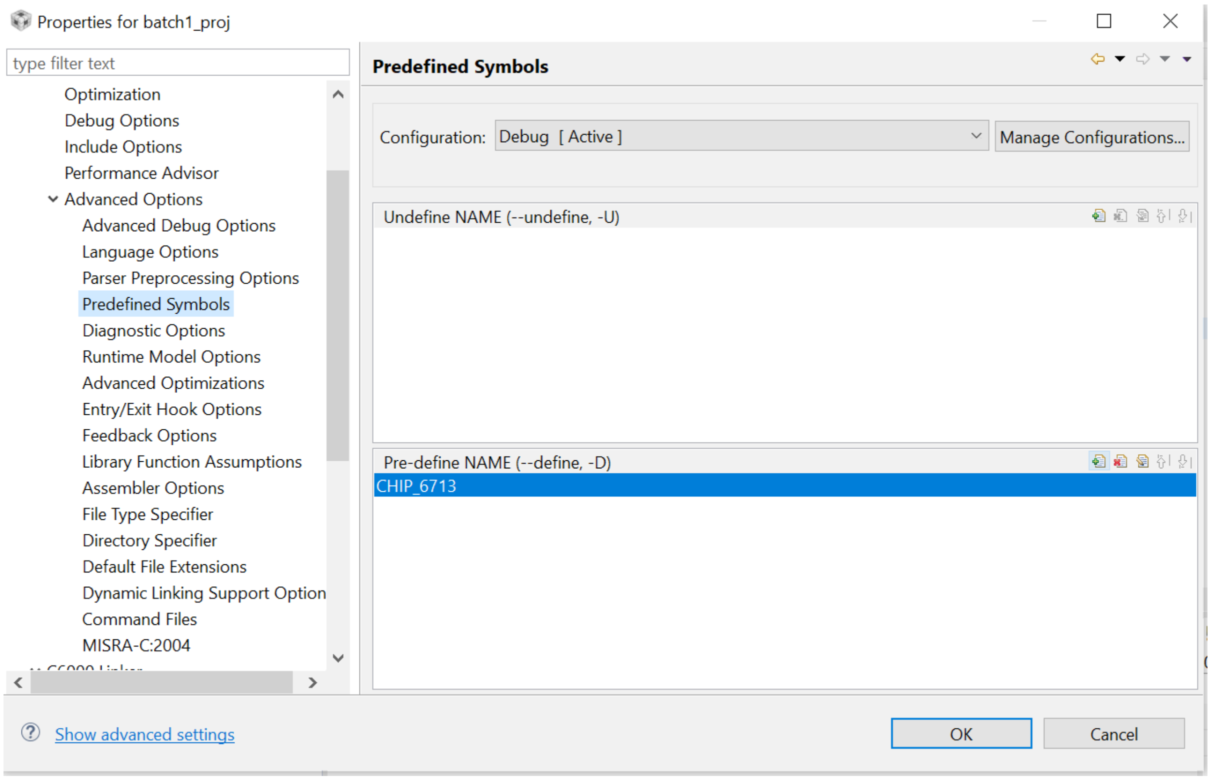
Configuration: Debug [Active]

Specify a preinclude file (--preinclude)

Add dir to #include search path (--include_path, -I)

C:\Users\Arun\workspace_v7\dsplab
${PROJECT_ROOT}
${CG_TOOL_ROOT}/include

Under *Advanced Options>Predefined Symbols*, in the *Pre-define NAME* window, click on the file icon with green + , and enter CHIP_6713. This symbol is used for conditional compilation. If you don't do this step, you will have to type the line #define CHIP_6713 in your source file.


Properties for batch1_proj

Predefined Symbols

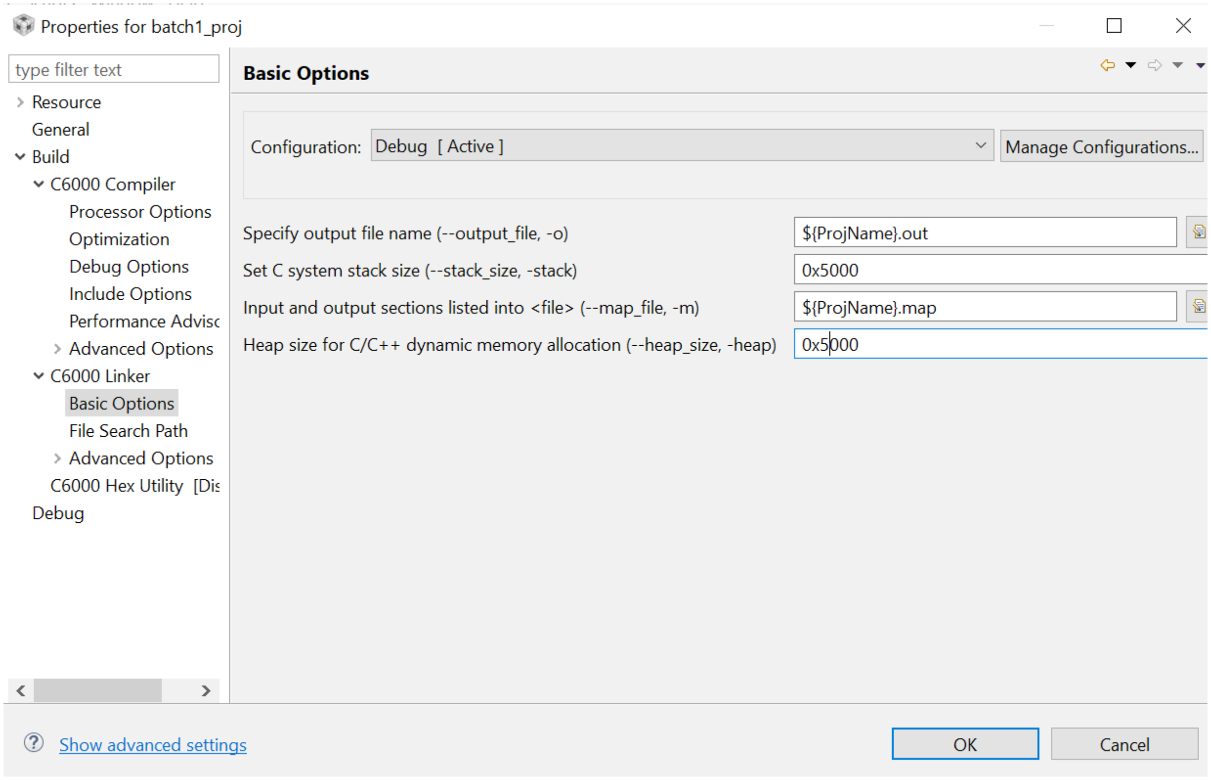
Configuration: Debug [Active]

Undefine NAME (--undefine, -U)

Pre-define NAME (--define, -D)

CHIP_6713

Under *C6000 Linker> Basic options*, enter a suitable value (eg: 0x5000) as the size for stack and heap


Properties for batch1_proj

Basic Options

Configuration: Debug [Active]

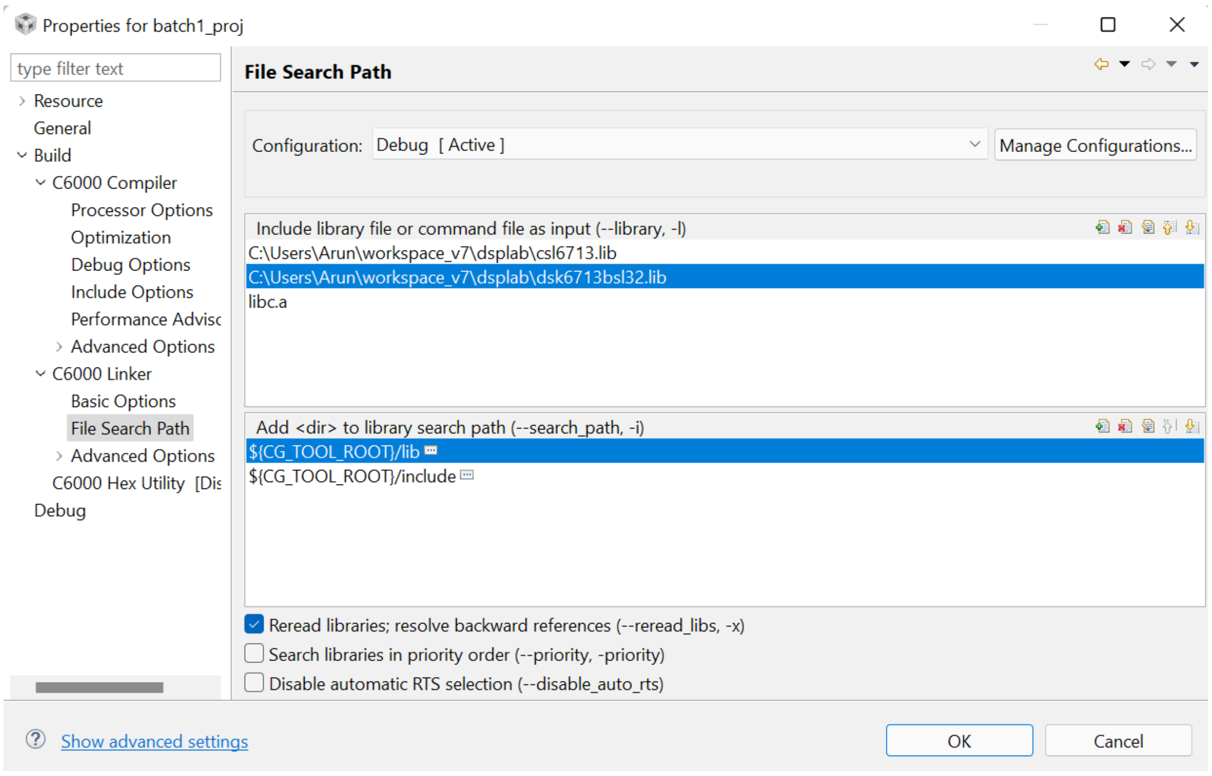
Specify output file name (--output_file, -o) ${ProjName}.out

Set C system stack size (--stack_size, -stack) 0x5000

Input and output sections listed into <file> (--map_file, -m) ${ProjName}.map

Heap size for C/C++ dynamic memory allocation (--heap_size, -heap) 0x5000

Under *C6000 Linker>File search path*, *Include library file or command file as input* window should already contain the file *libc.a* which is the standard C library. We need to add the chip support and board support libraries. Click on the file icon with green +, browse to *dsplab* folder, select the file *csl6713.lib*, click Open, then OK. Similarly add the file *dsk6713bsl32.lib*



Properties for batch1_proj

type filter text

- Resource
- General
- Build
  - C6000 Compiler
    - Processor Options
    - Optimization
    - Debug Options
    - Include Options
    - Performance Advisc
    - Advanced Options
  - C6000 Linker
    - Basic Options
    - File Search Path
    - Advanced Options
  - C6000 Hex Utility [Dis
- Debug

File Search Path

Configuration: Debug [Active]   Manage Configurations...

Include library file or command file as input (--library, -l)
```
C:\Users\Arun\workspace_v7\dsplab\csl6713.lib
C:\Users\Arun\workspace_v7\dsplab\dsk6713bsl32.lib
libc.a
```

Add <dir> to library search path (--search_path, -i)
```
${CG_TOOL_ROOT}/lib
${CG_TOOL_ROOT}/include
```

[✓] Reread libraries; resolve backward references (--reread_libs, -x)
[ ] Search libraries in priority order (--priority, -priority)
[ ] Disable automatic RTS selection (--disable_auto_rts)

Show advanced settings   OK   Cancel

a. LED Blink: In this first experiment, we will add source code that blinks LED #0 at a rate of about 2.5 times per second using the LED module of the DSK6713 Board Support Library. The example also reads the state of DIP switch #3 and lights LED #3 if the switch is depressed or turns it off if the switch is not depressed. The purpose of this experiment is to demonstrate basic Board Support Library usage as well as provide a project base for your own code. The BSL is divided into several modules, each of which has its own include file. The file *dsk6713.h* must be included in every program that uses the BSL. This example also includes *dsk6713_led.h* and *dsk6713_dip.h* because it uses the LED and DIP modules. To add a source file, right click on the project name and New>File and give file name as *led.c*. Click Finish. (You can also add from menu *File> New*). Type the following code in the *led.c* file:

```
/*  ======== led.c ========*/
#include <dsk6713.h>
#include <dsk6713_led.h>
#include <dsk6713_dip.h>
void main()
{
    /* Initialize the board support library, must be first BSL call
    */ DSK6713_init();

     /* Initialize the LED and DIP switch modules of the BSL */
    DSK6713_LED_init();
    DSK6713_DIP_init();

    while(1)
    {
          /* Toggle LED #0 */
          DSK6713_LED_toggle(0);

    /* Check DIP switch #3 and light LED #3 accordingly, */
    //0 = switch pressed
          if (DSK6713_DIP_get(3) == 0)
                /* Switch pressed, turn LED #3 on */
```

```
                        DSK6713_LED_on(3);
                else
                    /* Switch not pressed, turn LED #3 off */
                        DSK6713_LED_off(3);

                /* Spin in a software delay loop for about 200ms */
                DSK6713_waitusec(200000);
            }
        }
```

Save and Build the program (by clicking the hammer icon). Connect the DSK to the PC and start Debug (by clicking the bug icon). CCS will automatically perform a series of steps. It will switch to Debug perspective, connect to the debug probe, Load the project's executable file (.out) to the device memory, and will run to the function main(). Click Resume to continue execution. Verify that the program works as described. A number of debugging features are available in CCS, including setting breakpoints and watching variables, viewing memory, registers, and mixing C and assembly code, graphing results, and monitoring execution time. One can step through a program in different ways (step into, or over, or out). Since we will be using CCS for the rest of this lab course, you are strongly advised to read the CCS user manual online at https://software-dl.ti.com/ccs/esd/documents/users_guide/index.html and familiarize with its operation and features.

b. Ring counter: In this experiment, you will write a program to set up a ring counter using the four LEDs on the DSK. The counting speed should increase when dip switch 0 is pressed. You do not need to create a new project for this experiment. In the project explorer, right click on the file *led.c* and click Exclude from build. Now add a new source file to the project, name it *led_dip.c* and write your code there. This way, by swapping in and out files, you will be using the same project for all your programs.

c. Tone Generation: The code below will output a 1KHz tone by sending samples to the AIC23 codec onboard the DSK. The DAC converts the samples to an analog signal and outputs on the line-out and headphones out of the DSK. Modify your project by excluding the earlier file led_dip.c and adding the code below in a new file *tone.c*.

```
#include <math.h>
#include <dsk6713.h>
#include <dsk6713_aic23.h>
int main()
{
    float Fs = 8000.;
    float F0 = 1000.;
    float pi = 3.141592653589;
    float theta = 0.;
    float delta = 2. * pi * F0 / Fs; // increment for theta
    float sample;
    unsigned out_sample;
    /* Initialize the board support library, must be called first */
    DSK6713_init();

    DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
    DSK6713_AIC23_CodecHandle hCodec;
    /* Start the codec */
    hCodec = DSK6713_AIC23_openCodec(0, &config);

    /* Change the sampling rate to 16 kHz */
    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_8KHZ);

    for (;;)
    { /* Infinite loop */
```

```
        sample = 15000.0 * sin(theta); /* Scale for DAC */
        out_sample = (int)sample & 0x0000ffff; // Put in lower half
(R)

        /* Poll XRDY bit until true, then write to DXR */
        while (!DSK6713_AIC23_write(hCodec, out_sample))
            ;
        theta += delta;
        if (theta > 2 * pi)
            theta -= 2 * pi;

    }
}
```

In the code, right click on DSK6713_AIC23_DEFAULTCONFIG and click *Open Declaration*. This will open the header file which contains detailed information about how the codec is configured. (eg: how to decrease headphone volume?). The codec is started by calling the BSL function `DSK6713_AIC23_openCodec()`.

Each iteration of the infinite loop generates a sample and writes it to the codec. Note that the float value is converted to int and the upper 16 bits are set to 0 before outputting. This is because the BSL function that configures the codec sets McBSP1 to send and receive 32-bit words, with the left sample in the upper 16 bits and right sample in the lower 16 bits. The 16-bit samples are in signed 2s complement form. Since the upper 16 bits of out_sample are set to 0, the tone will be heard on the right channel only. If we want the output to be on the left channel, we can use the statement `out_sample = (int)sample << 16;` instead, which puts the 16-bit value in the top half, and sets the lower 16 bits to 0s. We can also pack two 16-bit samples in out_sample for output on both L and R channels.

The function `DSK6713_AIC23_write()` is used to write a pair of samples to the DAC. The function uses polling to write samples and returns 0 if codec is not ready and returns 1 if write is successful. The while loop continues till write is successful. Build and Debug the program. Connect your headphones to the headphone and listen to the tone(beware of volume). You should hear the tone on R channel only. Using bitwise operators in C, try to output the tone on both channels. Change frequency F0 to 500 Hz and listen. Next, change F0 to 7500 Hz and listen. Explain what you observe.

d. DIP controlled tone: Write a program that outputs a tone only if DIP switch #0 is pressed and held down. When pressed down, LED #0 should also light up.

e. Table lookup tone generation: An alternative approach to generate a sine wave is to store the samples for one period in an array and to lookup the array for each sample. Although lookup table approach is not very flexible to generate different frequencies, it has the advantage of less computational effort since the sine values are computed only once. Generate a 1KHz tone assuming a sampling frequency of 8KHz using the table lookup method.

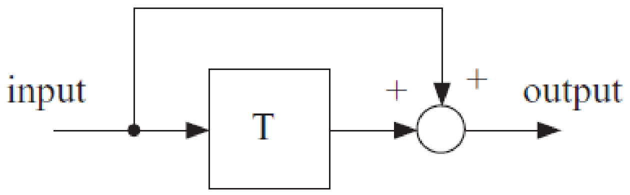
f. Audio Loopback: Code below reads pairs of left and right channel samples from the codec ADC and loops them back out to the codec DAC. The BSL function `DSK6713_AIC23_read()` is used to read a pair of samples from ADC and the function `DSK6713_AIC23_write()` is used to write a pair of samples to the DAC. Both functions use polling to read/write samples and returns 0 if codec is not ready and returns 1 if read/write is successful. The while loop continues till read/write is successful. Note that the function DSK6713_AIC23_read() uses a pointer variable. Build and Debug the program. Connect a 3.5mm aux cable from the headphone out of your PC (or phone). Connect a pair of headphones to the headphone out of the DSK. Verify that loopback is working.

```
// loopback.c
#include <dsk6713.h>
```

```c
#include <dsk6713_aic23.h>

void main(void)
{
    DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
    DSK6713_AIC23_CodecHandle hCodec;
    Uint32 sample_pair = 0;
    DSK6713_init(); /* In the BSL library */
    /* Start the codec */
    hCodec = DSK6713_AIC23_openCodec(0, &config);

    /* Change the sampling rate to 16 kHz */
    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_16KHZ);

    /* Read left and right channel samples from the ADC and loop */
    /* them back out to the DAC.                                  */
    for (;;)

        while (!DSK6713_AIC23_read(hCodec, &sample_pair))
            ;
        while (!DSK6713_AIC23_write(hCodec, sample_pair))
            ;
    }
}
```

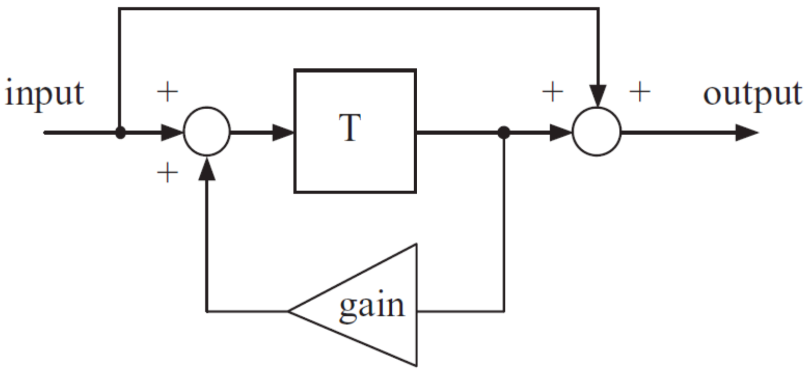
g. Quantization: This experiment studies quantization effects. The DSK's codec is a 16-bit ADC/DAC with each sample represented by a two's complement integer. The range of representable integers is: −32768 ≤ x ≤ 32767. For high-fidelity audio at least 16 bits are required to match the dynamic range of human hearing; for speech, 8 bits are sufficient. If the audio or speech samples are quantized to less than 8 bits, quantization noise will become audible. The 16-bit samples can be requantized to fewer bits by a right/left bit-shifting operation. For example, right shifting by 3 bits will knock out the last 3 bits, then left shifting by 3 bits will result in a 16-bit number whose last three bits are zero, that is, a 13-bit integer. Quantization can also be introduced by bitwise AND. For example
`sample_pair = sample_pair & 0xfffcfffc;`
will set the last two bits of L and R channels to zero. Insert this statement between the codec read and write operations in the loopback example and listen. Modify the statement so that the output samples are quantized to 12 bits, 10 bits, 8 bits.... You should hear an increase in the quantization noise as more number of bits are set to zero.

h. Delay: In the loopback experiment, we simply connected the input to the output. Typically, we will do some processing on the input sample and then output the processed sample. Some simple, yet striking, effects can be achieved simply by delaying the samples as they pass from input to output. Program delay.c, listed below, demonstrates this. A delay line is implemented using the array buffer to store samples as they are read from the codec. Once the array is full, the program overwrites the oldest stored input sample with the current, or newest, input sample. Just prior to overwriting the oldest stored input sample in buffer , that sample is retrieved, added to the current input sample, and output to the codec. Figure below shows a block diagram representation of the operation of program delay.c in which the block labeled T represents a delay of T seconds. Note that the sampling rate is set to 8 kHz, therefore, the delay of 8000 samples corresponds to a delay of 1 sec. Build and run the project, using line-in and headphones to verify its operation.

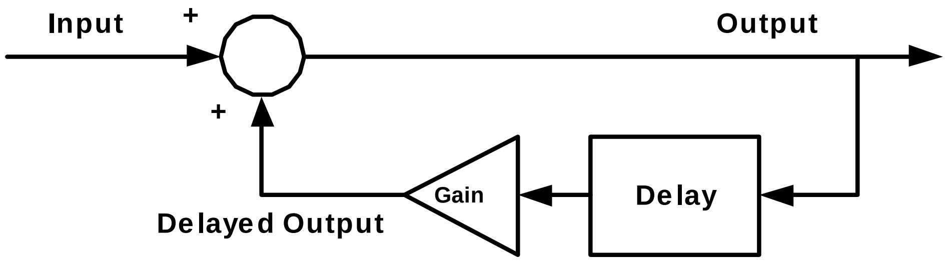


```
#include <dsk6713.h>
#include <dsk6713_aic23.h>
#define BUF_SIZE 8000
void main()
{

      Uint32 sample_pair; // both channels packed in 32-bits
      short i = 0, left, buffer[BUF_SIZE] = { 0 }, delayed, output;
      DSK6713_init();
      DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
      DSK6713_AIC23_CodecHandle hCodec;
      hCodec = DSK6713_AIC23_openCodec(0, &config);
      DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_8KHZ);
      while (1)
      {
            while (!DSK6713_AIC23_read(hCodec, &sample_pair))
                        ;
            //extract left sample and put in 16-bits
            left = (int)sample_pair >> 16; //
            delayed = buffer[i]; //read oldest sample
            output = left + delayed; //output sum of new and delayed
            buffer[i] = left; //replace oldest sample with input

            //increment i to point to the oldest sample
            if (++i >= BUF_SIZE)
                        i = 0;

            //put 16-bit sample in top-half
            sample_pair = (int)output << 16;

            while (!DSK6713_AIC23_write(hCodec, sample_pair))
                        ;

      }
}
```

i. Echo: Modify the delay program to create an echo. By feeding back a fraction of the output of the delay line to its input, a fading echo effect can be realized. A block diagram representation of the required program is given below. Note that the output is input + delayed as in the previous experiment, but the input to the delay line is now input + gain*delayed. Modify the previous program to realize an echo effect. Experiment with different values for gain (between 0.0 and 1.0) and also with different delays.



j. Reverberation: Whereas an echo is a single reflection of a soundwave off a distance surface, reverberation is the reflection of sound waves created by the superposition of such echoes caused by multiple reflections. A simplified model for reverb is given below. Realize the model in code an generate a reverberation effect. Set delay buffer length to 2500 and gain=0.5. Play some speech on your PC/phone and listen to the effect. Pause the play in between to observe the effect better. Change gain to 0.25 and 0.75 and note the effect of the change. For more realistic reverb effects, see [8]



2. Linear Convolution:

a. A program to compute the linear convolution of two finite duration sequences x[n] and h[n] is given below. It is assumed that x[n] extends from n=0 to L-1 and h[n] extends from n=0 to M-1. Length of y[n] is then L+M-1. The function **conv()** implements the convolution operation given by $y[n]=\sum_{k=0}^{M-1} h[k]x[n-k]$. The sequence x[n] is reflected and shifted. For each n, the range over which both sequences overlap needs to be determined. In the function, conditional operators are used to determine kmin and kmax.

```
#include <stdio.h>
void conv(float *x, float *h, float *y, short l, short m);
void main()
{
    float x[] = { 1, 2, 3, 4 };
    float h[] = { 1, 1, 1 };
    float y[6]; //length of y = L+M-1
    short n;
    conv(x, h, y, 4, 3);
    for (n = 0; n < 6; n++)
        printf("%.2f ", y[n]);
}

void conv(float *x, float *h, float *y, short l, short m)
{
    short k, kmin, kmax, n;
    for (n = 0; n < l + m - 1; n++)
    {
        y[n] = 0;
        kmin = (n > l - 1) ? n - l + 1 : 0;
        kmax = (n > m - 1) ? m - 1 : n;
        //printf("%d %d\n", kmin, kmax);
        for (k = kmin; k <= kmax; k++)
        {
            y[n] = y[n] + h[k] * x[n - k];

        }

    }

}
```

Build and Debug. During a debug session, we can visualize the signals using the Graph tool in CCS. Step Over the program line by line and halt when y has been computed. Click on *Tools>Graph>Single-Time*. Set the Graph Properties as follows:

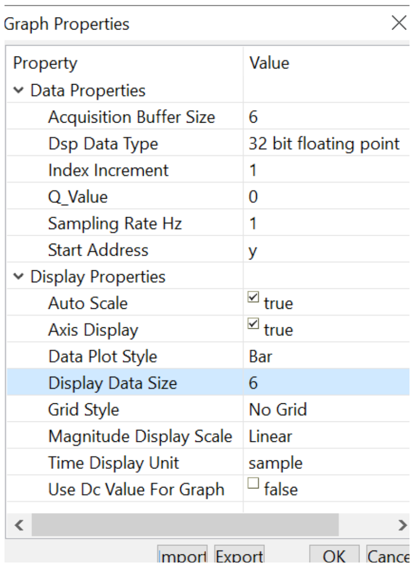
Acquisition buffer size: 6 (since y[n] has 6 elements)

DSP Data type: 32-bit floating point (since we are using float type variable)

Start Address: y (array name by itself is treated as the address of its first element)

Display Data Size: 6

Click Ok. The graph of y[n] should appear.


Graph Properties

Property	Value
Data Properties	
Acquisition Buffer Size	6
Dsp Data Type	32 bit floating point
Index Increment	1
Q_Value	0
Sampling Rate Hz	1
Start Address	y
Display Properties	
Auto Scale	☑ true
Axis Display	☑ true
Data Plot Style	Bar
Display Data Size	6
Grid Style	No Grid
Magnitude Display Scale	Linear
Time Display Unit	sample
Use Dc Value For Graph	☐ false

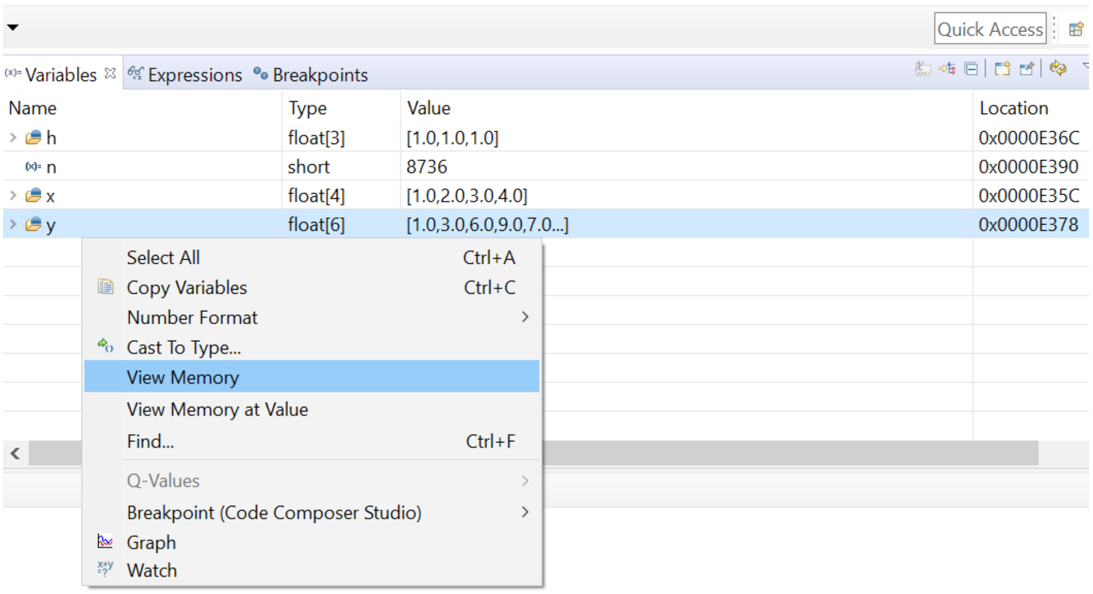
Instead of the above steps, you can go to the *Variables* view (from the cluster of tabbed views named Variables, Expressions, and Registers), right click on y and click Graph. To emphasize the discrete nature of y[n], click on *Show The Graph Properties* button in the graph window toolbar and change the *Data Plot Style* to Bar. Also, right click anywhere on the graph, on the context menu that opens, select *Display Properties*, click *Axes* and change *Y-axis Display format* to *Decimal*

b. Memory Save/ Load: In the previous experiment, the elements of the arrays x and h were hard-coded in the program itself. We might need to process input data saved in files (for example, when simulating real-time input with predigitized data captured at another time/place). CCS can read data from a file on the host computer and put the data in target processor memory. CCS can also write the processed data samples from the target processor to the host computer as an output file for analysis.

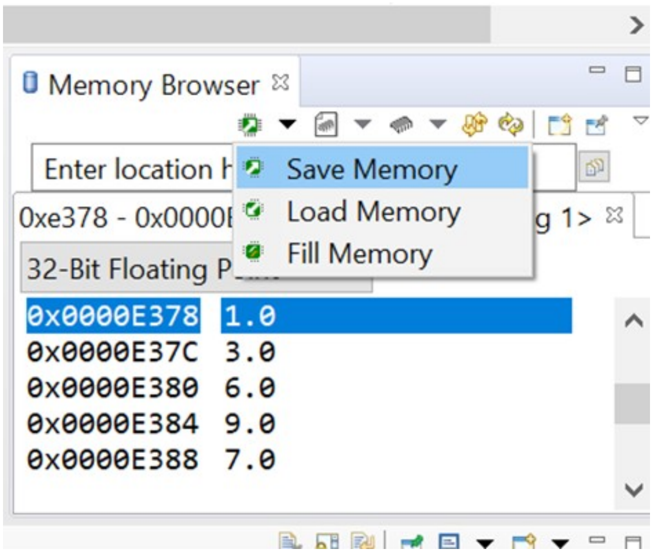
In this experiment, we will first save the output y[n] from the previous part in a file and then use that file as the input x[n] and perform convolution again. In effect, we are simulating the cascade of two identical systems with impulse response h[n].

First, start Debugging the program in part a once again and Step over line by line.

Once the conv() function returns, go to Variables view, right click on variable y and click *View Memory*.



Name	Type	Value	Location
h	float[3]	[1.0,1.0,1.0]	0x0000E36C
n	short	8736	0x0000E390
x	float[4]	[1.0,2.0,3.0,4.0]	0x0000E35C
y	float[6]	[1.0,3.0,6.0,9.0,7.0...]	0x0000E378



```
0x0000E378  1.0
0x0000E37C  3.0
0x0000E380  6.0
0x0000E384  9.0
0x0000E388  7.0
```

In the *Memory Browser* window that opens, you can see the memory addresses and data in y. In the memory browser toolbar, click on *Save memory*.

Browse to your workspace folder and give some file name. Click *Save*.

Leave *File type* as *TI data*.

Click *Next*, select *Format* as *32-bit floating point*.

Find out the starting address for y in the Memory Browser and type the correct start address in the *Start Address* field.

Specify the number of memory words as 6 (since y has 6 floats, and a float as well as the word-size on the 6713 is 32 bits, 6 words mean 6 floats).

Click Finish.

Your workspace folder should now have a file with .dat extension, containing the data in y.

Terminate the Debug session.

Next, we are going to use the saved file as the input x[n] for convolution. In the program, change the size of x array to 6, delete the initializer list, change size of y array to 8(=6+3-1).

Also make the necessary change in the length of x in call to the function conv(). Rebuild the program with the changes and start stepping over.

Once memory for x is allocated (when the line float x[6]; is executed), right click on x from the Variables view, and select *View Memory*.

From the memory browser toolbar, click *Load Memory*. Browse to the previously saved file, give file type, start address and length and click Finish.

The array x should now be filled with data from the previously saved file. Resume debugging and observe the new output.

c. C Library File IO: CCS also supports standard C library file I/O functions such as fopen( ), fclose( ), fread( ), fwrite( ), and so on. These functions not only provide the ability of operating on different file formats, but also allow users to directly use the functions on computers. Comparing with the memory load/save method introduced in the previous part, these file I/O functions are portable to other development environments. An example of C program that uses fopen( ), fclose( ), fread( ), and fwrite( ) functions is included below. Verify the working of the program in CCS.

```c
#include <stdio.h>
void writedatafile(void);
void conv(float *x, float *h, float *y, short l, short m);
void main()
{
    writedatafile();//put some data in a file
    FILE *fp;
    fp = fopen("../../myfilebin", "rb");//open file for binary read
    float x[5];
   //read 5 floats from file and store in x
   fread(x, sizeof(float), 5, fp);
    float h[3]={1, 1, 1};
    float y[7];
    conv(x, h, y, 5, 3);

}

void writedatafile()
{
    float a[] = { 1, 2, 3, 4, 5}; //some data
    FILE *fp;
    fp = fopen("../../myfilebin", "wb");//open for binary write

    //write 5 floats in array a to myfilebin
    fwrite(a, sizeof(float), 5, fp);
    fclose(fp);

}

void conv(float *x, float *h, float *y, short l, short m)
{
    short k, kmin, kmax, n;
    for (n = 0; n < l + m - 1; n++)
    {
        y[n] = 0;
        kmin = (n > l - 1) ? n - l + 1 : 0;
        kmax = (n > m - 1) ? m - 1 : n;
        // printf("%d %d\n", kmin, kmax);
        for (k = kmin; k <= kmax; k++)
        {
            y[n] = y[n] + h[k] * x[n - k];
        }

    }

}
```

3. FFT:

The DFT X[k] of a finite length sequence x[n] defined for n=0... N-1 can be obtained by sampling

its DTFT $X\left(e^{j\omega}\right)$ on the ω axis between $0 \le \omega < 2\pi$ at $\omega_k = \frac{2\pi k}{N}, k = 0 \ldots N-1$. Ie. $DFT\{x[n]\} = X[k] = X\left(e^{j\omega}\right) \lfloor \omega = \frac{2\pi k}{N}$. Using the commonly used notation $W_N = e^{\frac{-j2\pi}{N}}$,
$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, k = 0 \ldots N-1$$

Using Euler's relation $e^{-j\theta} = cos\theta - jsin\theta$, the real and imaginary parts of X[k] are:
$$ReX(k) = \sum_{k=0}^{N-1} \left(Rex(n)\cos(2\pi kn/N) + Imx(n)\sin(2\pi kn/N)\right)$$
$$ImX(k) = \sum_{k=0}^{N-1} \left(Imx(n)\cos(2\pi kn/N) - Rex(n)\sin(2\pi kn/N)\right)$$
The function dft() in the program below implements the above two equations to compute DFT. A structure COMPLEX array is used to store the real and imaginary parts of x[n] and X[k]. The computations are performed in-place with the input array over-written by the output array. The program computes the 64-point DFT on the 64 samples of a 1KHz signal sampled at 8000Hz.

```c
#include <math.h>
#define PI 3.1415926535897
#define M 64 //signal length
#define N 64 //DFT length
typedef struct
{
    float real;
    float imag;
} COMPLEX;
void dft(COMPLEX *);

void main()
{
    int n;
    float F = 1000.0, Fs = 8000.0;
    COMPLEX samples[N]={0.0};

    //Generate time-domain signal
    for (n = 0; n < M; n++) //M samples of x[n]
    {
        samples[n].real = cos(2 * PI * F * n / Fs);
        samples[n].imag = 0.0;
    }


    dft(samples); //call DFT function

}

void dft(COMPLEX *x)
{
    COMPLEX result[N];
    int k, n;
    for (k = 0; k < N; k++) // N point DFT
    {
        result[k].real = 0.0;
        result[k].imag = 0.0;
        for (n = 0; n < N; n++)
```

```
                {
                    result[k].real += x[n].real * cos(2 * PI * k * n / N) +
                                      x[n].imag * sin(2 * PI * k * n / N);
                    result[k].imag += x[n].imag * cos(2 * PI * k * n / N) -
                                      x[n].real * sin(2 * PI * k * n / N);
                }
            }
            for (k = 0; k < N; k++)
            {
                x[k] = result[k];
            }
        }
```

Procedure: Insert a breakpoint in the code on the line calling the dft() function (To add a breakpoint, double click on the line number OR right-click anywhere on the line and click Breakpoint(CCS) >Breakpoint. You should see a small blue mark on the line where the breakpoint is inserted).

Build the project and Debug.

The program will first halt at entry to main().Click Resume and program will halt at the breakpoint. The array `samples` now contain the time-domain signal. We can visualize the signal using the Graph tool in CCS. Click on Tools>Graph>Single-Time. Set the Graph Properties as follows:

Acquisition buffer size: 64 (since we have stored 64 samples of x[n]
DSP Data type: 32 bit floating point
Index increment: 2 (The nature of the structure array samples is such that it comprises 2N float values ordered so that the first value is the real part of x[0], the second is the imaginary part of x[0], the third is the real part of x[1], and so on. Since x[n] is purely real, we take alternate values only)
Start Address: samples
Data plot style: Bar
Display Data Size: 64
Click Ok. The graph of x[n] should appear.

Click *Step Over* and the program and will now halt after the function dft() returns. At this point, the array `samples` contain the 64 DFT coefficients X[k]. The graph now displays the real part of the X[k]. If needed, click on Reset the Graph button and then Refresh button in the graph window. You should see two distinct peaks at k= 8 and k=56.

The spike at k=8 with amplitude =32 corresponds to the frequency $F=\frac{k}{NT}=\frac{kF_s}{N}=\frac{8\,x\,8000}{64}=1\,KHz$. The spike at k=56 is a consequence of the fact that DFT of a real x[n] is conjugate symmetric; $X[k]=X^{*}[N-k]$. So $X[56]=X^{*}[64-56]=X^{*}[8]=X[8]$.

Other than these two real components, all other DFT coefficients are zero for this example.

To get a better understanding of the spectrum of x[n], we can compute a DFT with N > 64, say N=256. Change the line `#define N 64` to `#define N 256`. Rebuild and Debug. At the first breakpoint, open graph properties and change Acquisition Buffer Size and Display Data Size to 256 (other properties as before). Since we haven't changed M, we should see zeros after the first 64 samples (zero-padding). At the second break point, you should see the real part of the 256-point DFT of x[n].

In order to display the imaginary (rather than the real) parts of the sequence X[k], the Start Address must be set to the address of the second value of type float in the array samples. That address can be found by moving the cursor over an occurrence of the identifier samples in the source file. (The address can also be found from the variables window.) Its hexadecimal address will appear in a pop -

up box. Entering this value in the Start Address field of the Graph Property Dialog window in place of the identifier samples will result in the same Graphical Display. Adding four (the number of bytes used to store one 32 - bit floating point value) to the Start Address value will result in the imaginary parts of the sequence of complex values being displayed.

FFT Magnitude Graph: The magnitude and phase of the DFT can be computed from the real and imaginary parts with some effort, but more easily, the graph tool in CCS can be used to compute and display the FFT magnitude and phase of a time-domain data array. Restart the program and at the first breakpoint when the array samples contain the time-domain samples, click on Tools>Graph>FFT magnitude, set Acquisition Buffer Size to 256, DSP data type to 32-bit floating point, Index increment to 2 (since imaginary part of x[n] is 0), Sampling Rate to 8000, Signal Type to Real, Start Address to samples, Data Plot Style to Bar, FFT order to 8 (so that FFT Frame Size is 256). In the FFT magnitude graph that is displayed, there should be a peak at 1000 Hz. The graph shows frequencies upto Fs/2 only, since the information in the other half is redundant.

- **a.** Modify the program above to sample the signal $x(t)=\cos(2\pi 1000t)+0.75\cos(2\pi 500t)$ at 8 KHz and save 64 samples. Compute its DFT and display the real and imaginary parts. Using the FFT magnitude graph tool in CCS, display the magnitude of its 64-point DFT (Remember, the FFT Magnitude graph tool in CCS computes and displays the DFT magnitude of a time-domain input). Explain the location of the peaks. Also display its 128-point DFT magnitude and explain.
- **b.** Direct computation of a single DFT point using the dft() function requires N − 1 additions and N multiplications. Thus, direct computation of all N points requires N(N − 1) complex additions and N^2 complex multiplications. The computational complexity can be reduced to the order of $N \log_2 N$ by algorithms known as fast Fourier transforms (FFT's). One FFT algorithm is called the decimation-in-time algorithm. A C function fft() for computing a complex, radix-2, decimation-in-time FFT is included below (adapted from [4]). Replace the dft() function in part a with the fft() function given below and repeat the procedure in part a.

```c
void fft(COMPLEX *X)
/* X is an array of N = 2**M complex points. */
{
    COMPLEX temp1; /* temporary storage complex variable */
    COMPLEX W; /* e**(-j 2 pi/ N) */
    COMPLEX U; /* Twiddle factor W**k */
    int i, j, k; /* loop indexes */
    int id; /* Index of lower point in butterfly */
    int Nt, num_stages = 0; /* Number of stages */
    int N2 = N / 2;
    int L; /* FFT stage */
    int LE; /* Number of points in sub DFT at stage L, */
    /* and offset to next DFT in stage */
    int LE1; /* Number of butterflies in one DFT at*/
    /* stage L. Also is offset to lower */
    /* point in butterfly at stage L */
    float pi = 3.1415926535897;
    /*=============================================================*/
    /* Rearrange input array in bit-reversed order */
    /* */
    /* The index j is the bit reversed value of i. Since 0 -> 0 */
    /* and N-1 -> N-1 under bit-reversal, these two reversals are */
    /* skipped. */
    j = 0;
    for (i = 1; i < (N - 1); i++)
    {
        /*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
        /* Increment bit-reversed counter for j by adding 1 to msb and */
        /* propagating carries from left to right. */
        k = N2; /* k is 1 in msb, 0 elsewhere */
        /*-------------------------------------------------------------*/
        /* Propagate carry from left to right */
        while (k <= j) /* Propagate carry if bit is 1 */
        {
            j = j - k; /* Bit tested is 1, so clear it. */
            k = k / 2; /* Set up 1 for next bit to right. */
        }
        j = j + k; /* Change 1st 0 from left to 1 */
        /*-------------------------------------------------------------*/
        /*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
        /* Swap samples at locations i and j if not previously swapped.*/
        if (i < j) /* Test if previously swapped. */
        {
            temp1.real = (X[j]).real;
            temp1.imag = (X[j]).imag;
            (X[j]).real = (X[i]).real;
            (X[j]).imag = (X[i]).imag;
            (X[i]).real = temp1.real;
            (X[i]).imag = temp1.imag;
        }
        /*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
    }
    /*=============================================================*/
    /* Do M stages of butterflies */
    Nt = N;
    while (Nt >>= 1)
        ++num_stages;    //num_stages=log2(Nt)
    for (L = 1; L <= num_stages; L++)
    {
        LE = 1 << L; /* LE = 2**L = points in sub DFT */
        LE1 = LE / 2; /* Number of butterflies in sub-DFT */
        U.real = 1.0;
        U.imag = 0.0; /* U = 1 + j 0 */
        W.real = cos(pi / LE1);
        W.imag = -sin(pi / LE1); /* W = e**(-j 2 pi/LE) */
        /*----------------------------------------------------------------*/
        /* Do butterflies for L-th stage */
        for (j = 0; j < LE1; j++) /* Do the LE1 butterflies per sub DFT*/
        {

/*..............................................................*/
            /* Compute butterflies that use same W**k */
            for (i = j; i < N; i += LE)
            {
                id = i + LE1; /* Index of lower point in butterfly */
                temp1.real = (X[id]).real * U.real - (X[id]).imag * U.imag;
                temp1.imag = (X[id]).imag * U.real + (X[id]).real * U.imag;
                (X[id]).real = (X[i]).real - temp1.real;
                (X[id]).imag = (X[i]).imag - temp1.imag;
                (X[i]).real = (X[i]).real + temp1.real;
                (X[i]).imag = (X[i]).imag + temp1.imag;
            }

/*..............................................................*/
            /* Recursively compute W**k as W*W**(k-1) = W*U */
            temp1.real = U.real * W.real - U.imag * W.imag;
            U.imag = U.real * W.imag + U.imag * W.real;
            U.real = temp1.real;

/*..............................................................*/
        }
        /*----------------------------------------------------------------*/
    }
    return;
}
```

c. Real-time FFT: Rather than processing one sample at a time, the DFT and the FFT algorithms process blocks, or frames, of samples. Frame - based processing divides continuous sequences of input and output samples into frames of N samples. In this experiment, the DSK will collect blocks of N=1024 samples taken at 8KHz from the codec. N=1024 samples will be read from the codec and stored. The function fft() will compute the 1024-point DFT of the block. The results will be displayed on the PC by using Code Composer Studio's graphing capabilities. Another block of data is then read from the codec, its DFT computed and visualized, and the process is repeated. Your program should:

i. Initialize the DSK and codec as usual, setting the sampling rate to 8KHz

ii. Using a loop, read N sample_pairs (codec outputs L and R samples as a pair) from codec, extract one half of each pair (corresponding to L or R), convert to float and store in the real part of the structure array samples. These can be done with the following code fragment:

```
for (i = 0; i < N; i++) //read N samples from codec and store
 {
     while (!DSK6713_AIC23_read(hCodec, &sample_pair))
        ;
      samples[i].real = (int)sample_pair>>16;
      samples[i].imag = 0.0;

}
```

To reduce effects of signal truncation, multiply the frame by a Hamming window sequence. The Hamming window array should be stored beforehand:

```
    float hamming[N];
```

```
for(i=0; i<N; i++)
    hamming[i]=0.54-0.46*cos(2*pi*i/(N-1));
```

iii. Call fft() function to compute the N-point DFT of the frame
iv. Steps ii and iii should be enclosed in an infinite loop to repeat the frame read and DFT computation
v. Insert two breakpoints before and after fft() is called.
vi. Connect the headphone out of your PC to the line-in input of the DSK using a 3.5mm aux cable. Play a 1KHz tone from your PC (search youtube for 1KHz tone)
vii. Use Graph tool in CCS to display the time-domain signal at the first breakpoint and real part of the DFT at the second breakpoint. You might need to Reset the Graph and Refresh to auto-scale the graph at each breakpoint when the program is halted.
viii. Play tones of different frequencies from youtube (< 4KHz) and observe the signal and spectrum
ix. Play any audio signal from your PC and observe the signal and spectrum.

In the above code, the DSP spends nearly all the time waiting to receive samples from the codec. It processes the frame only when all the N samples have been acquired. A much more efficient approach is to let the DSP perform some other task in the background (such as computing the DFT of a previously acquired frame) and have the serial port interrupt the background tasks when a sample has been received. The interrupt service routine is called a foreground task. Such an approach requires two buffers(called ping pong buffers). While a new frame of input samples is being collected in one buffer using interrupts, a previously collected frame of input samples in the other buffer is processed. After the tasks are completed, the ping and pong buffers interchange their roles.

The TMS320C6713 has an enhanced direct memory access controller (EDMA) that can transfer data between any locations in the DSP's 32-bit address space independently of the CPU. Efficiency can also be improved by configuring the DMA controller to send/receive an entire block of data.