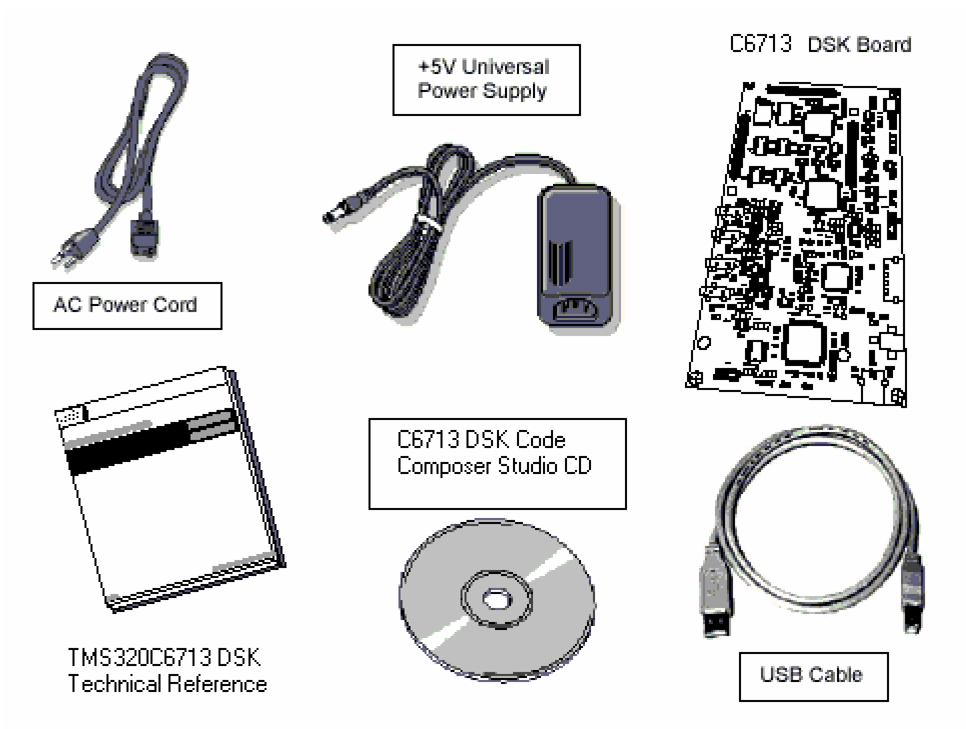

# TMS320C6713 DSK
# USER MANUAL

**Cranes Software International Limited**
**(TI-Solutions)**
**#5, Airport Road, Domlur Layout, Bangalore – 560 071.**
**Phone: 080 – 51254131/ 2/ 3/ 4, Fax: 080 - 25356299.**
**tisales@cranessoftware.com**

## CONTENTS

# TMS320C6713 DSK

## Package Contents




The C6713™ DSK builds on TI's industry-leading line of low cost, easy-to-use DSP Starter Kit (DSK) development boards. The high-performance board features the TMS320C6713 floating-point DSP. Capable of performing 1350 million floating-point operations per second (MFLOPS), the C6713 DSP makes the C6713 DSK the most powerful DSK development board.

The DSK is USB port interfaced platform that allows to efficiently develop and test applications for the C6713. The DSK consists of a C6713-based printed circuit board that will serve as a hardware reference design for TI's customers' products. With extensive host PC and target DSP software support, including bundled TI tools, the DSK provides ease-of-use and capabilities that are attractive to DSP engineers.

The following checklist details items that are shipped with the C6711 DSK kit.

- TMS320C6713 DSK    TMS320C6713 DSK development board
- Other hardware    External 5VDC power supply

IEEE 1284 compliant male-to-female cable

- CD-ROM Code Composer Studio DSK tools

The C6713 DSK has a TMS320C6713 DSP onboard that allows full-speed verification of code with Code Composer Studio. The C6713 DSK provides:

- A USB Interface
- SDRAM and ROM
- An analog interface circuit for Data conversion (AIC)
- An I/O port
- Embedded JTAG emulation support

Connectors on the C6713 DSK provide DSP external memory interface (EMIF) and peripheral signals that enable its functionality to be expanded with custom or third party daughter boards.

The DSK provides a C6713 hardware reference design that can assist you in the development of your own C6713-based products. In addition to providing a reference for interfacing the DSP to various types of memories and peripherals, the design also addresses power, clock, JTAG, and parallel peripheral interfaces.

The C6713 DSK includes a stereo codec. This analog interface circuit (AIC) has the following characteristics:

High-Performance Stereo *Codec*

- 90-dB SNR Multibit Sigma-Delta ADC (A-weighted at 48 kHz)
- 100-dB SNR Multibit Sigma-Delta DAC (A-weighted at 48 kHz)
- 1.42 V – 3.6 V Core Digital Supply: Compatible With TI C54x DSP Core Voltages
- 2.7 V – 3.6 V Buffer and Analog Supply: Compatible Both TI C54x DSP Buffer Voltages
- 8-kHz – 96-kHz Sampling-Frequency Support

Software Control Via TI McBSP-Compatible Multiprotocol Serial Port

- I 2 C-Compatible and SPI-Compatible Serial-Port Protocols
- Glueless Interface to TI McBSPs
- 

Audio-Data Input/Output Via TI McBSP-Compatible Programmable Audio Interface

- I 2 S-Compatible Interface Requiring Only One McBSP for both ADC and DAC
- Standard I 2 S, MSB, or LSB Justified-Data Transfers
- 16/20/24/32-Bit Word Lengths

**The C6713DSK has the following features:**

The 6713 DSK is a low-cost standalone development platform that enables customers to evaluate and develop applications for the TI C67XX DSP family. The DSK also serves as a hardware reference design for the TMS320C6713 DSP. Schematics, logic equations and application notes are available to ease hardware development and reduce time to market.

The DSK uses the 32-bit EMIF for the SDRAM (CE0) and daughtercard expansion interface (CE2 and CE3). The Flash is attached to CE1 of the EMIF in 8-bit mode.

An on-board AIC23 codec allows the DSP to transmit and receive analog signals. McBSP0 is used for the codec control interface and McBSP1 is used for data. Analog audio I/O is done through four 3.5mm audio jacks that correspond to 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. 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 the CPLD registers. The registers reside at the midpoint of CE1.

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

An included 5V external power supply is used to power the board. On-board voltage regulators provide the 1.26V DSP core voltage, 3.3V digital and 3.3V analog voltages. A voltage supervisor monitors the internally generated voltage, and will hold the board in reset until the supplies are within operating specifications and the reset button is released. If desired, JP1 and JP2 can be used as power test points for the core and I/O power supplies.

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.

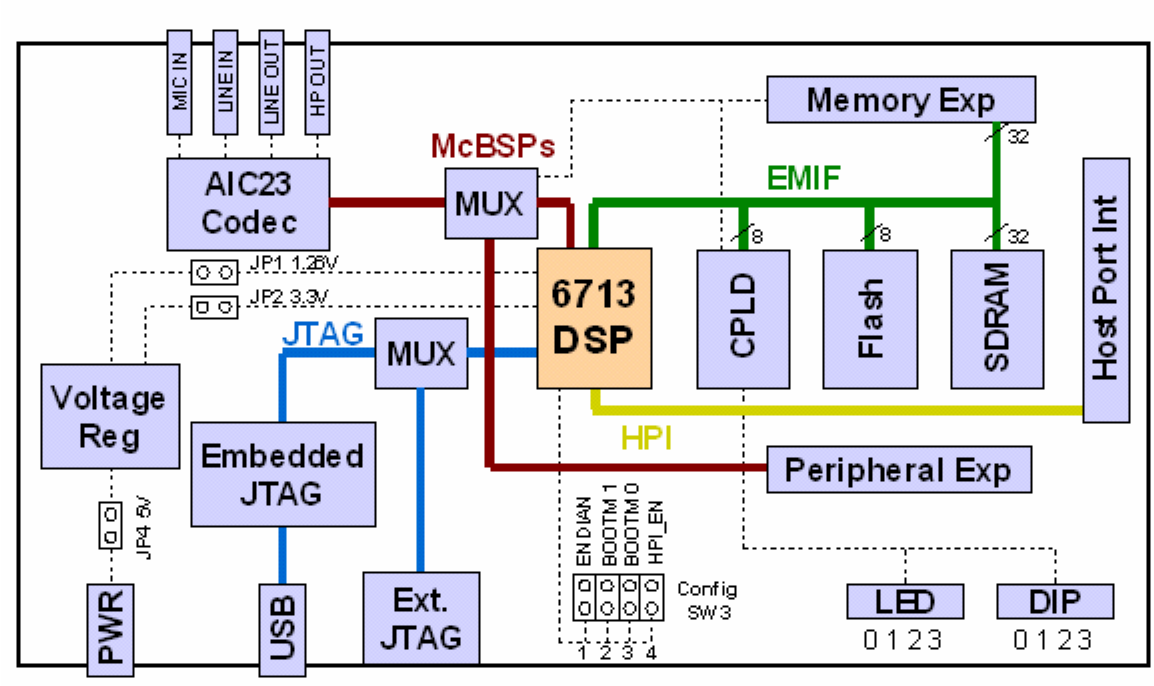
**TMS320C6713 DSP Features**

- Highest-Performance Floating-Point Digital Signal Processor (DSP):
  - Eight 32-Bit Instructions/Cycle
  - 32/64-Bit Data Word
  - 300-, **225-**, 200-MHz (GDP), and **225-**, 200-, 167-MHz (PYP) Clock Rates

  - 3.3-, 4.4-, 5-, 6-Instruction Cycle Times
  - 2400/1800, 1800/1350, 1600/1200, and 1336/1000 MIPS /MFLOPS
  - Rich Peripheral Set, Optimized for Audio
  - Highly Optimized C/C++ Compiler
  - Extended Temperature Devices Available
- Advanced Very Long Instruction Word (VLIW) TMS320C67x™ DSP Core
  - Eight Independent Functional Units:
    - Two ALUs (Fixed-Point)
    - Four ALUs (Floating- and Fixed-Point)
    - Two Multipliers (Floating- and Fixed-Point)
  - Load-Store Architecture With 32 32-Bit General-Purpose Registers
  - Instruction Packing Reduces Code Size
  - All Instructions Conditional
- Instruction Set Features
  - Native Instructions for IEEE 754
    - Single- and Double-Precision
  - Byte-Addressable (8-, 16-, 32-Bit Data)
  - 8-Bit Overflow Protection
  - Saturation; Bit-Field Extract, Set, Clear; Bit-Counting; Normalization
- L1/L2 Memory Architecture
  - 4K-Byte L1P Program Cache (Direct-Mapped)
  - 4K-Byte L1D Data Cache (2-Way)
  - 256K-Byte L2 Memory Total: 64K-Byte L2 Unified Cache/Mapped RAM, and 192K-Byte Additional L2 Mapped RAM
- Device Configuration
  - Boot Mode: HPI, 8-, 16-, 32-Bit ROM Boot
  - Endianness: Little Endian, Big Endian
- 32-Bit External Memory Interface (EMIF)
  - Glueless Interface to SRAM, EPROM, Flash, SBSRAM, and SDRAM
  - 512M-Byte Total Addressable External Memory Space
- Enhanced Direct-Memory-Access (EDMA) Controller (16 Independent Channels)
- 16-Bit Host-Port Interface (HPI)
- Two Multichannel Audio Serial Ports (McASPs)
  - Two Independent Clock Zones Each (1 TX and 1 RX)
  - Eight Serial Data Pins Per Port:
    Individually Assignable to any of the Clock Zones
  - Each Clock Zone Includes:
    - Programmable Clock Generator
    - Programmable Frame Sync Generator
    - TDM Streams From 2-32 Time Slots
    - Support for Slot Size:
      8, 12, 16, 20, 24, 28, 32 Bits
    - Data Formatter for Bit Manipulation
  - Wide Variety of I2S and Similar Bit Stream Formats

- Integrated Digital Audio Interface Transmitter (DIT) Supports:
  - S/PDIF, IEC60958-1, AES-3, CP-430 Formats
  - Up to 16 transmit pins
  - Enhanced Channel Status/User Data
- Extensive Error Checking and Recovery

- Two Inter-Integrated Circuit Bus ($I^2C$ Bus™) Multi-Master and Slave Interfaces
- Two Multichannel Buffered Serial Ports:
  - Serial-Peripheral-Interface (SPI)
  - High-Speed TDM Interface
  - AC97 Interface
- Two 32-Bit General-Purpose Timers
- Dedicated GPIO Module With 16 pins (External Interrupt Capable)
- Flexible Phase-Locked-Loop (PLL) Based Clock Generator Module
- IEEE-1149.1 (JTAG †) Boundary-Scan-Compatible
- Package Options:
  - 208-Pin PowerPAD™ Plastic (Low-Profile) Quad Flatpack (PYP)
  - 272-BGA Packages (GDP and ZDP)
- 0.13-µm/6-Level Copper Metal Process
  - CMOS Technology
- 3.3-V I/Os, 1.2 ‡ -V Internal (GDP & PYP)
- 3.3-V I/Os, 1.4-V Internal (GDP)(300 MHz only)




**TMS320C6713 DSK Overview Block Diagram**

# INSTALLATION

## SYSTEM REQUIREMENTS

| Minimum | Recommended |
|---|---|
| • 233MHz or Higher Pentium-Compatible CPU<br>• 600MB of free hard disk space<br>• 128MB of RAM<br>• SVGA (800 x 600 ) display<br>• Internet Explorer (4.0 or later) or<br>• Netscape Navigator (4.7 or later)<br>• Local CD-ROM drive | • 500MHz or Higher Pentium – Compatible CPU<br>• 128MB RAM<br>• 16bit Color |
| **Supported Operating Systems** | |
| • Windows® 98<br>• Windows NT® 4.0 Service Pack 4 or higher<br>• Windows® 2000 Service Pack 1<br>• Windows® Me<br>• Windows® XP | |

## DSK HARDWARE INSTALLATION

- Shut down and power off the PC
- Connect the supplied USB port cable to the board
- Connect the other end of the cable to the USB port of PC

  ***Note:*** *If you plan to install a Microphone, speaker, or Signal generator/CRO these must be plugged in properly before you connect power to the DSK*

- Plug the power cable into the board
- Plug the other end of the power cable into a power outlet
- The user LEDs should flash several times to indicate board is operational
- When you connect your DSK through USB for the first time on a Windows loaded PC the new hardware found wizard will come up. **So, Install the drivers** (The CCS CD contains the require drivers for C5416 DSK).
- Install the CCS software for C5416 DSK.

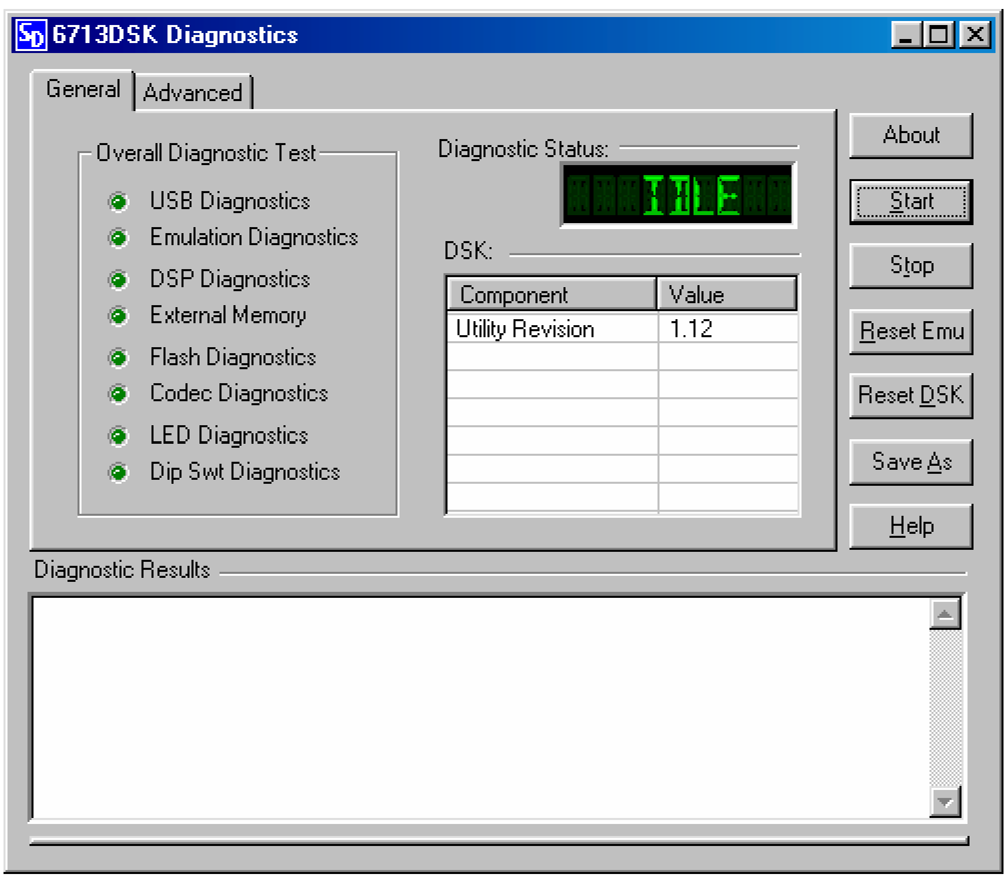
## Troubleshooting DSK Connectivity

If Code Composer Studio IDE fails to configure your port correctly, perform the following steps:

- Test the USB port by running DSK Port test from the start menu

*Use Start→Programs→Texas Instruments→Code Composer Studio→Code Composer Studio C6713 DSK Tools→C6713 DSK Diagnostic Utilities*

- *The below Screen will appear*
- *Select→ Start→Select 6713 DSK Diagnostic Utility Icon from Desktop*
- *The Screen Look like as below*
- *Select **Start** Option*
- *Utility Program will test the board*
- *After testing Diagnostic Status you will get **PASS***




*If the board still fails to detect*

*Go to CMOS setup→ Enable the USB Port Option*
*(The required Device drivers will load along with CCS Installation)*

## SOFTWARE INSTALLATION

*You must install the hardware before you install the software on your system*.

The requirements for the operating platform are;

- Insert the installation CD into the CD-ROM drive

  An install screen appears; if not, goes to the windows Explorer and run setup.exe

- Choose the option to install Code Composer Sutido

  If you already have C6000 CC Studio IDE installed on your PC, do not install DSK software. CC Studio IDE full tools supports the DSK platform

Respond to the dialog boxes as the installation program runs.

*The Installation program automatically configures CC Studio IDE for operation with your DSK and creates a CCStudio IDE DSK icon on your desktop. To install, follow these instructions:*

## INTRODUCTION TO CODE COMPOSER STUDIO

Code Composer is the DSP industry's first fully integrated development environment (IDE) with DSP-specific functionality. With a familiar environment liked MS-based C++TM, Code Composer lets you edit, build, debug, profile and manage projects from a single unified environment. Other unique features include graphical signal analysis, injection/extraction of data signals via file I/O, multi-processor debugging, automated testing and customization via a C-interpretive scripting language and much more.

## CODE COMPOSER FEATURES INCLUDE:

- IDE
- Debug IDE
- Advanced watch windows
- Integrated editor
- File I/O, Probe Points, and graphical algorithm scope probes
- Advanced graphical signal analysis
- Interactive profiling
- Automated testing and customization via scripting
- Visual project management system
- Compile in the background while editing and debugging
- Multi-processor debugging
- Help on the target DSP

**Note :**

*Documents for Reference:*

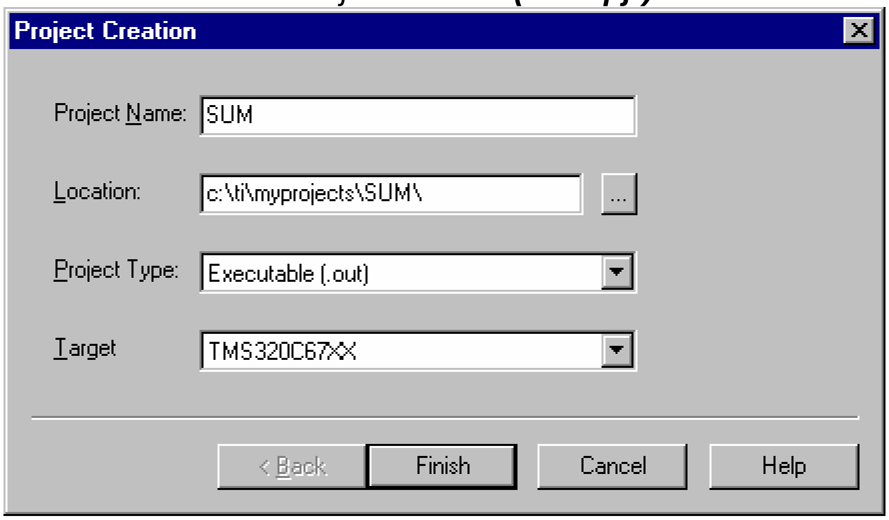
**spru509 → Code Composer Studio getting started guide.**
**spru189 → TMS320C6000 CPU & Instruction set guide**
**spru190 → TMS320C6000 Peripherals guide**
**slws106d → codec(TLV320AIC23) Data Manual.**
**spru402 → Programmer's Reference Guide.**
**sprs186j → TMS320C6713 DSP**

**Soft Copy of Documents are available at : c:\ti\docs\pdf.**

# Procedure to work on Code Composer Studio

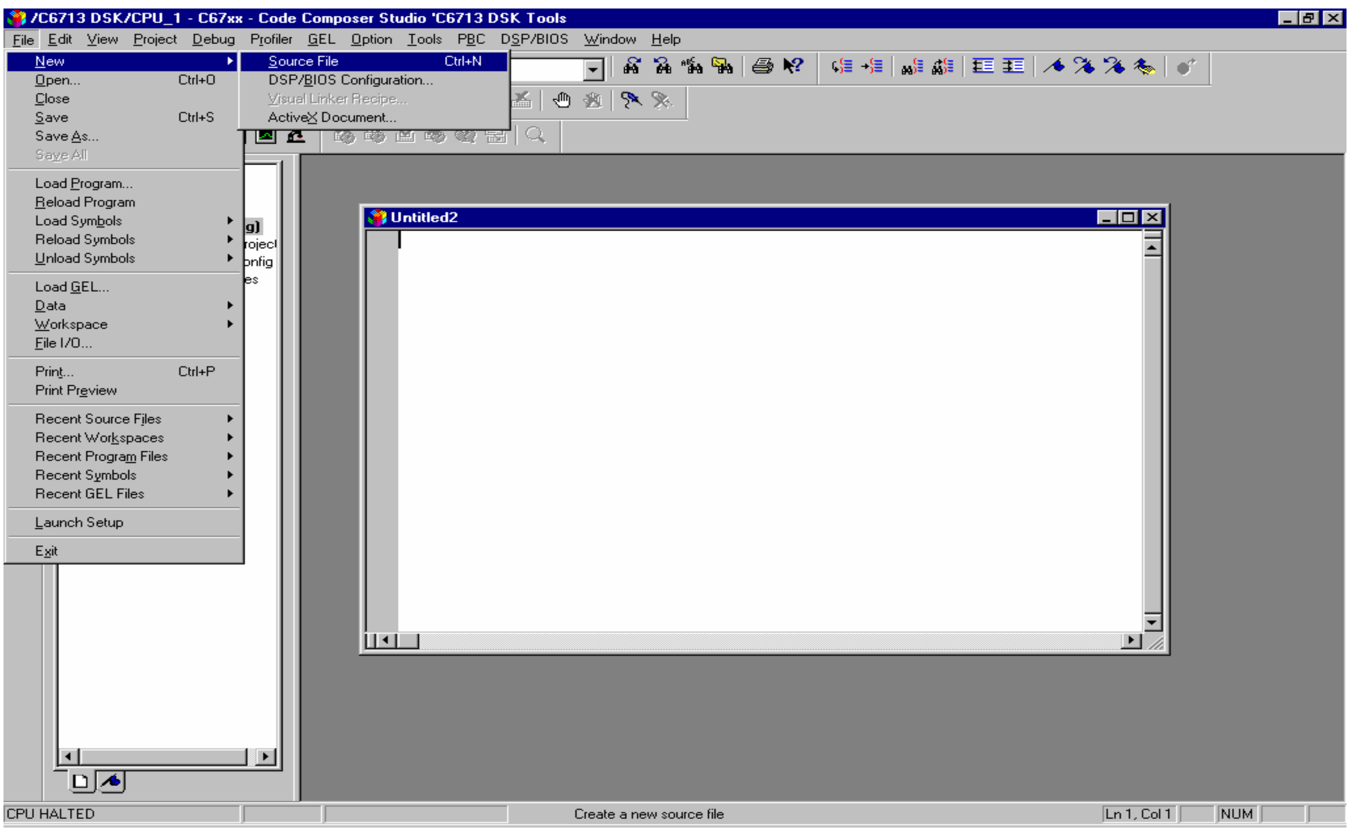
## 1. To create a New Project

*Project → New **(SUM.pjt)***




## 2. To Create a Source file

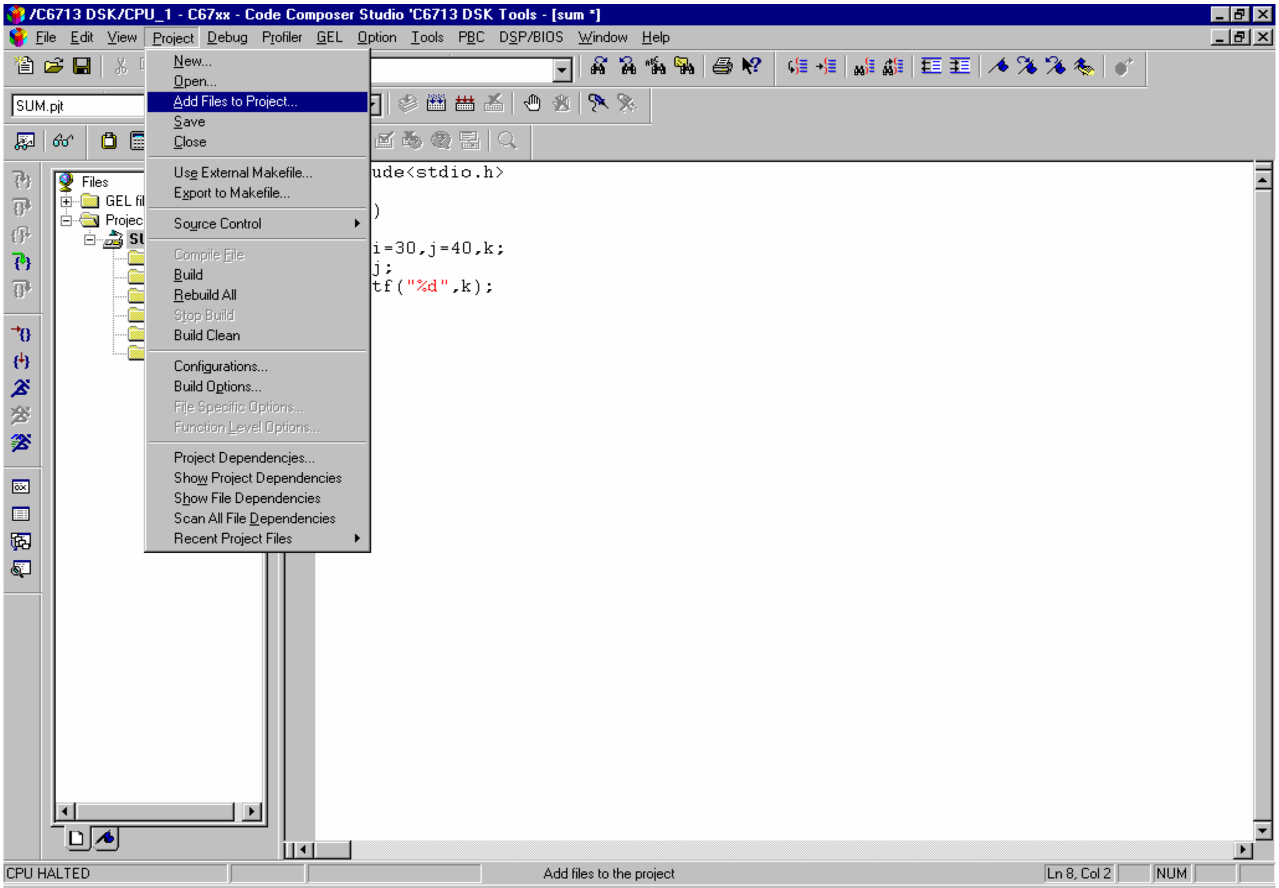
*File → New*

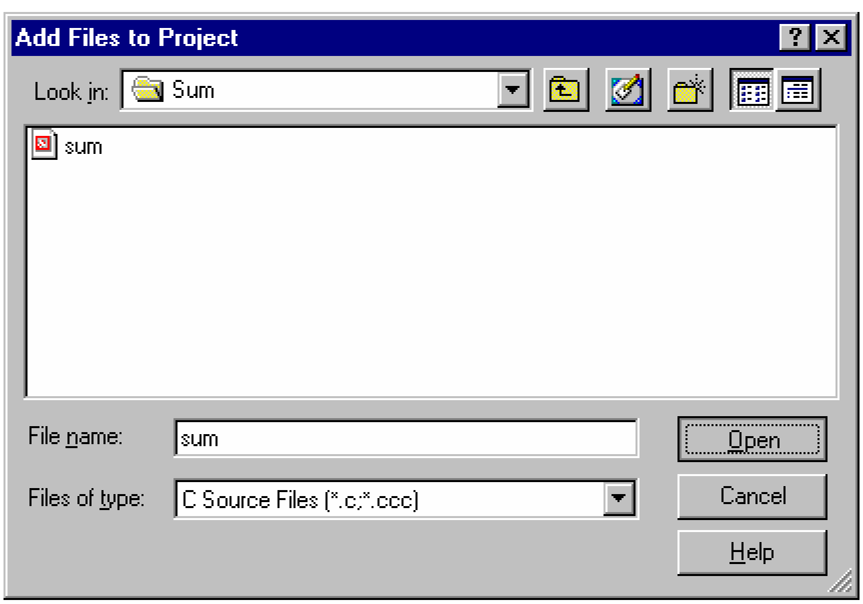



*Type the code (Save & give a name to file, Eg: **sum.c**).*

## 3. To Add Source files to Project

*Project → Add files to Project → sum.c*

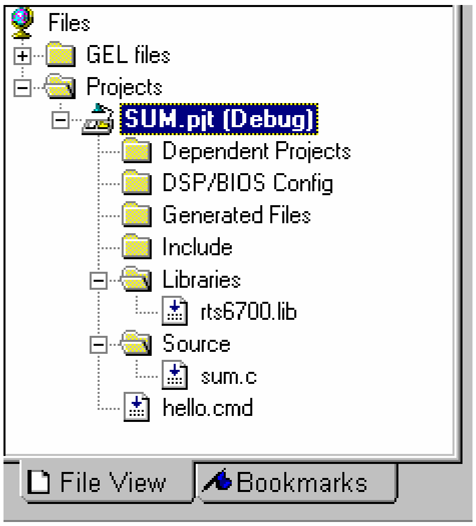


**4. To Add rts6700.lib file & hello.cmd:**

*Project → Add files to Project →rts6700.lib*
***Path: c:\CCStudio\c6000\cgtools\lib\rts6700.lib***
***Note: Select Object & Library in(*.o,*.l) in Type of files***

*Project → Add files to Project →hello.cmd*
***Path: c:\ti\tutorial\dsk6713\hello1\hello.cmd***
***Note: Select Linker Command file(*.cmd) in Type of files***




**5. To Compile:**

*Project → Compile File*

**6. To build or Link:**

*Project → build,*
*Which will create the final executable* ***(.out)*** *file.(Eg. sum.out).*

**7. Procedure to Load and Run program:**

***Load program to DSK:***

*File → Load program → sum. out*

**8. To execute project:**

*Debug → Run.*

## sum.c

```
#include<stdio.h>

main()
{
 int i=30,j=40,k;
 k=i+j;
 printf("%d",k);
}
```

**To Perform Single Step Debugging:**

1. Keep the cursor on the on to the line from where u want to start single step debugging.(eg: set a break point on to first line int i=0; of your project.)

   To set break point select  icon from tool bar menu.

2. Load the **Vectors. out** file onto the target.

3. Go to view and select Watch window.

4. Debug → Run.

5. Execution should halt at break point.

6. Now press F10. See the changes happening in the watch window.

7. Similarly go to view & select CPU registers to view the changes happening in CPU registers.
8. Repeat steps 2 to 6.

# DIFFERENCE EQUATION

An Nth order linear constant – coefficient difference equation can be represented as

$$\sum_{k=0}^{N} a_k\, y(n-k) = \sum_{r=0}^{M} b_r\, x(n-r)$$

If we assume that the system is causal a linear difference equation provides an explicit relationship between the input and output..this can be seen by rewriting above equation.

$$y(n) = \sum_{r=0}^{M} b_r/a_0\, x(n-r) \quad -- \quad \sum_{k=1}^{N} a_k/a_{0\,y}\, y(n-k)$$

**<u>'C' Program to Implement Difference Equation</u>**

```
#include <stdio.h>
#include<math.h>

#define FREQ 400

float y[3]={0,0,0};
float x[3]={0,0,0};
float z[128],m[128],n[128],p[128];

main()
{
 int i=0,j;
 float a[3]={ 0.072231,0.144462,0.072231};
 float b[3]={ 1.000000,-1.109229,0.398152};

 for(i=0;i<128;i++)
 {
  m[i]=sin(2*3.14*FREQ*i/24000);
 }

for(j=0;j<128;j++)
   {
    x[0]=m[j];
    y[0] = (a[0] *x[0]) +(a[1]* x[1] ) +(x[2]*a[2]) - (y[1]*b[1])-
                                                       (y[2]*b[2]);
    z[j]=y[0];
    y[2]=y[1];
    y[1]=y[0];
    x[2]=x[1];
    x[1] = x[0];
   }
```

```
}
```

## PROCEDURE:

- Open Code Composer Studio, make sure the DSP kit is turned on.

- Start a new project using 'Project-new ' pull down menu, save it in a separate directory(c:\ti\myprojects) with name **lconv.pjt.**

- Add the source files **DIFF EQ1.c** to the project using **'Project→add files to project'** pull down menu.

- Add the linker command file **hello.cmd** .
  *(Path: c:\ti\tutorial\dsk6713\hello1\hello.cmd)*

- Add the run time support library file **rts6700.lib**
  *(Path: c:\ti\c6000\cgtools\lib\rts6700.lib)*

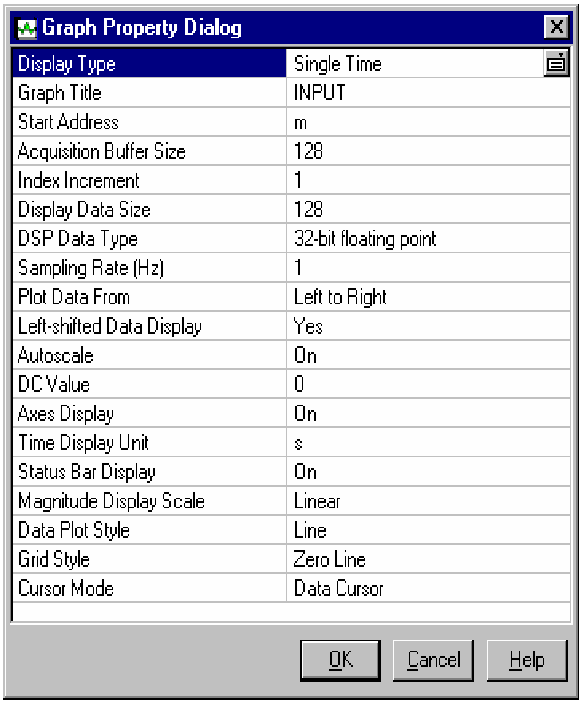
- Compile the program using the 'Project-compile' pull down menu or by clicking the shortcut icon on the left side of program window.

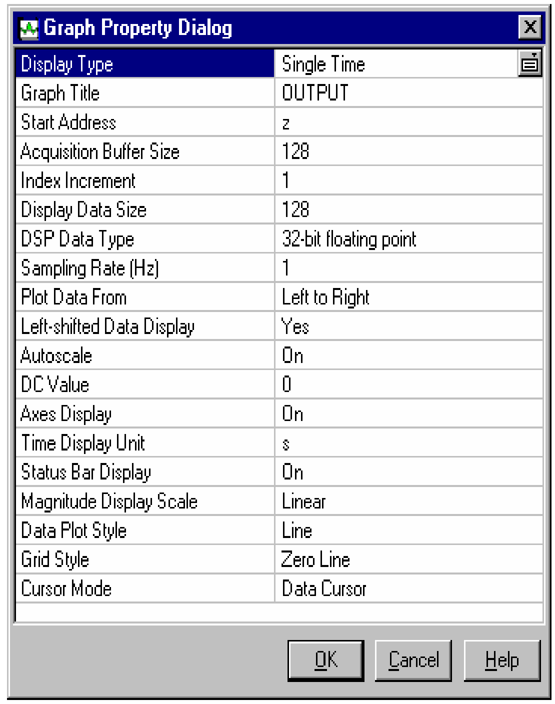
- Build the program using the 'Project-Build' pull down menu or by clicking the shortcut icon on the left side of program window.

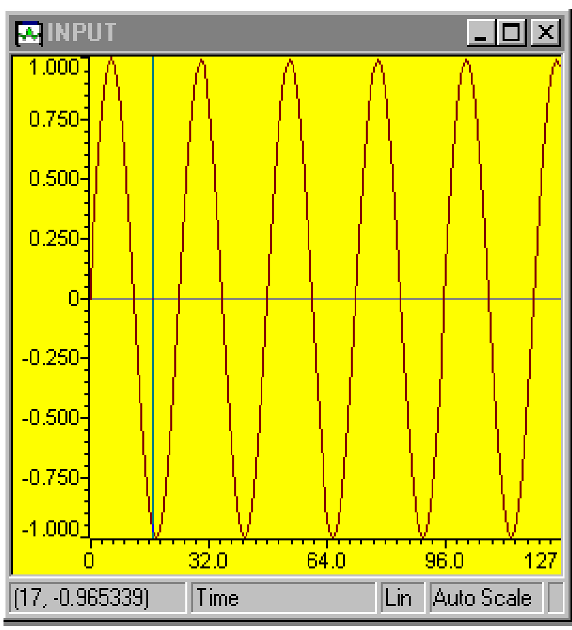
- Load the program(lconv.out) in program memory of DSP chip using the 'File-load program' pull down menu.

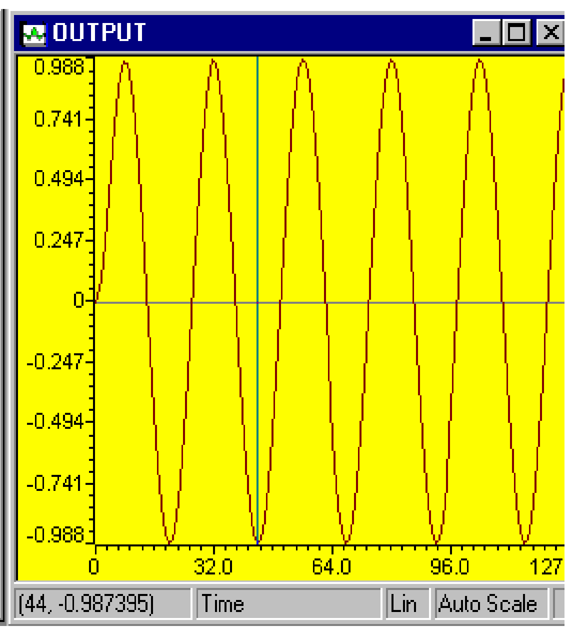
- To View output graphically
  Select view → graph → time and frequency.




***Note: To verify the Diffence Equation , Observe the output for high frequency and low frequency by changing variable "FREQ" in the program.***

# IMPULSE RESPONSE

## ‘C’ Program to Implement Impulse response:

```
#include <stdio.h>

#define Order  2
#define Len    10

float y[Len]={0,0,0},sum;

main()
{
 int j,k;

 float a[Order+1]={0.1311, 0.2622, 0.1311};
 float b[Order+1]={1, -0.7478, 0.2722};

for(j=0;j<Len;j++)
   {
                  sum=0;
                        for(k=1;k<=Order;k++)
                              {
                                    if((j-k)>=0)
                                          sum=sum+(b[k]*y[j-k]);
                              }
                        if(j<=Order)
                              {
                                    y[j]=a[j]-sum;
                              }
                              else
                              {
                                    y[j]=-sum;
                              }
                  printf("Respose[%d] = %f\n",j,y[j]);

   }
}
```

# LINEAR CONVOLUTION

## To Verify Linear Convolution:

Linear Convolution Involves the following operations.

1. Folding
2. Multiplication
3. Addition
4. Shifting

These operations can be represented by a Mathematical Expression as follows:

$$y[n] = \sum_{k=-\infty} x[k]h[n-k]$$

**x[ ]=** Input signal Samples
**h[ ]=** Impulse response co-efficient.
**y[ ]=** Convolution output.
**n =** No. of Input samples
**h =** No. of Impulse response co-efficient.

**<u>Algorithm to implement 'C' or Assembly program for Convolution:</u>**

**Eg: x[n] = {1, 2, 3, 4}**
**h[k] = {1, 2, 3, 4}**

**Where: n=4, k=4. ;Values of n & k should be a multiple of 4.**
**If n & k are not multiples of 4, pad with zero's to make multiples of 4**
**r= n+k-1 ; Size of output sequence.**
**= 4+4-1**
**= 7.**

| r= | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **n= 0** | x[0]h[0] | x[0]h[1] | x[0]h[2] | x[0]h[3] | | | |
| **1** | | x[1]h[0] | x[1]h[1] | x[1]h[2] | x[1]h[3] | | |
| **2** | | | x[2]h[0] | x[2]h[1] | x[2]h[2] | x[2]h[3] | |
| **3** | | | | x[3]h[0] | x[3]h[1] | x[3]h[2] | x[3]h[3] |

**Output: y[r] = { 1, 4, 10, 20, 25, 24, 16}.**

**NOTE: At the end of input sequences pad 'n' and 'k' no. of zero's**

## *'C' PROGRAM TO IMPLEMENT LINEAR CONVOLUTION*

```
/* prg to implement linear convolution */
#include<stdio.h>

#define LENGHT1 6       /*Lenght of i/p samples sequence*/
#define LENGHT2 4       /*Lenght of impulse response Co-efficients */

int x[2*LENGHT1-1]={1,2,3,4,5,6,0,0,0,0,0};     /*Input Signal Samples*/
int h[2*LENGHT1-1]={1,2,3,4,0,0,0,0,0,0,0};     /*Impulse Response Co-
efficients*/

int y[LENGHT1+LENGHT2-1];

main()
{
      int i=0,j;

      for(i=0;i<(LENGHT1+LENGHT2-1);i++)
      {
      y[i]=0;
      for(j=0;j<=i;j++)

            y[i]+=x[j]*h[i-j];

      }
      for(i=0;i<(LENGHT1+LENGHT2-1);i++)
      printf("%d\n",y[i]);

}
```

## PROCEDURE:

- Open Code Composer Studio, make sure the DSP kit is turned on.

- Start a new project using 'Project-new ' pull down menu, save it in a separate directory(c:\ti\myprojects) with name **lconv.pjt.**

- Add the source files **conv.c**
- to the project using 'Project→add files to project' pull down menu.

- Add the linker command file **hello.cmd** .
  (Path: c:\ti\tutorial\dsk6713\hello1\hello.cmd)

- Add the run time support library file **rts6700.lib**
  (Path: c:\ti\c6000\cgtools\lib\rts6700.lib)

- Compile the program using the 'Project-compile' pull down menu or by clicking the shortcut icon on the left side of program window.

- Build the program using the 'Project-Build' pull down menu or by clicking the shortcut icon on the left side of program window.

- Load the program(lconv.out) in program memory of DSP chip using the 'File-load program' pull down menu.

- To View output graphically
  Select view → graph → time and frequency.

***ASSEMBLY PROGRAM TO IMPLEMENT LINEAR CONVOLUTION***

## conv.asm:

```
                        .global _main
X           .half 1,2,3,4,0,0,0,0   ;input1, M=4
H           .half 1,2,3,4,0,0,0,0   ;input2, N=4
            .bss   Y,14,2                 ;OUTPUT, R=M+N-1
```

**;At the end of input sequences pad 'M' and 'N' no. of zero's**

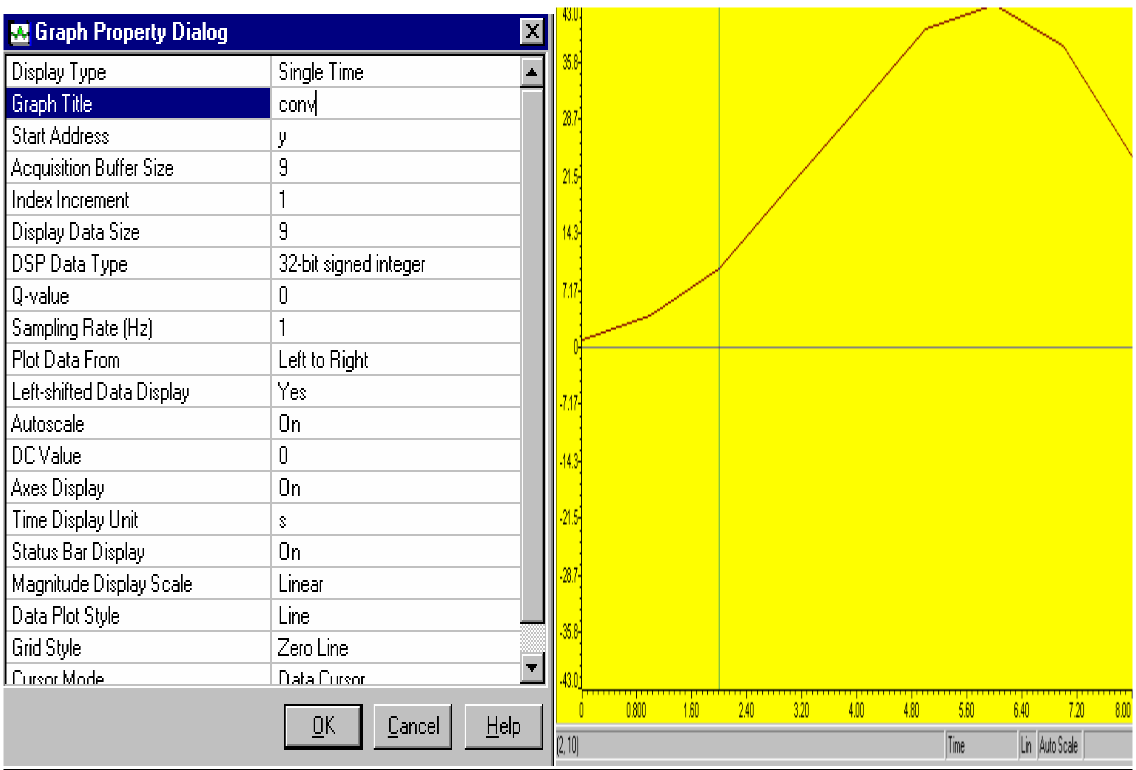
```
_main:
                  MVKL  .S1 X,A4
                  MVKH  .S1 X,A4          ;POINTER TO X
                  MVKL  .S2 H,B4
                  MVKH  .S2 H,B4          ;POINTER TO H
                  MVKL  .S1 Y,A5
                  MVKH  .S1 Y,A5          ;POINTER TO Y

                  MVK   .S2  7,B2         ;R=M+N-1
```

**;MOVE THE VALUE OF 'R'TO B2 FOR DIFFERENT LENGTH OF I/P SEQUENCES**

```
                  ZERO  .L1  A7

                  ZERO  .L1  A3           ;I=0
LL2:
                  ZERO  .L1  A2
                  ZERO .L1  A8            ;J=0, for(i=0;i<m+n-1;i++)
LL1:
                  LDH         .D1 *A4[A8],A6    ; for(j=0;j<=i;j++)
                  MV          .S2X A8,B5        ;    y[i]+=x[j]*h[i-j];
                  SUB         .L2  A3,B5,B7
                  LDH         .D2 *B4[B7],B6
                  NOP 4
                  MPY         .M1X A6,B6,A7
                  ADD         .L1 A8,1,A8
                  ADD         .L1  A2,A7,A2
                  CMPLT .L2X B5,A3,B0
      [B0]        B           .S2    LL1
                  NOP 5

                  STH   .D1  A2,*A5[A3]
```

```
ADD          .L1   A3,1,A3
             CMPLT .L1X A3,B2,A2
[A2]         B            .S1 LL2
             NOP 5

             B      B3
             NOP 5
```

**Configure the graphical window as shown below**



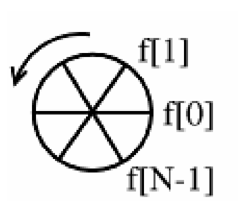
# Circular Convolution

## *Steps for Cyclic Convolution*

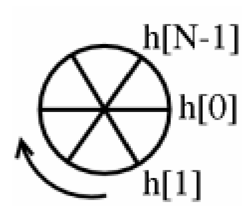
Steps for cyclic convolution are the same as the usual convolution, except all index calculations are done "mod N" = "on the wheel"

Steps for Cyclic Convolution

Step1: “Plot f[m] and h[−m]




Subfigure 1.1




Subfigure 1.2

Step 2: "Spin" h[−m] *n* times Anti Clock Wise (counter-clockwise) to get h[n-m]

(i.e. Simply rotate the sequence, *h*[*n*], clockwise by *n* steps)

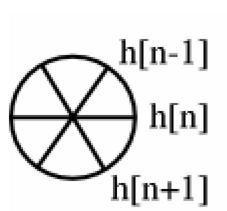
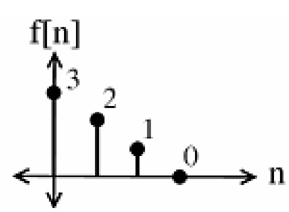



Figure 2: Step 2

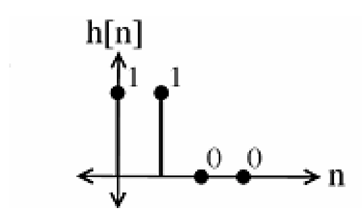
Step 3: Pointwise multiply the *f*[*m*] wheel and the *h*[n−m] wheel. sum=*y*[*n*]

Step 4: Repeat for all 0≤*n*≤*N*−1

Example 1: Convolve (n = 4)




Subfigure 3.1




Subfigure 3.2

Figure 3: Two discrete-time signals to be convolved.

- $h[-m]$ =

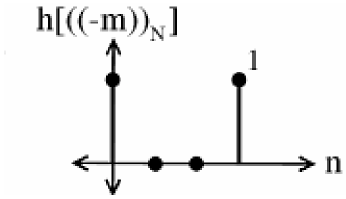



Figure 4

Multiply $f[m]$ and sum to yield: $y[0]$ =3

- $h[1-m]$

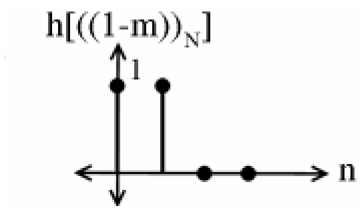



Figure 5

Multiply $f[m]$ and sum to yield: $y[1]$ =5

- $h[2-m]$

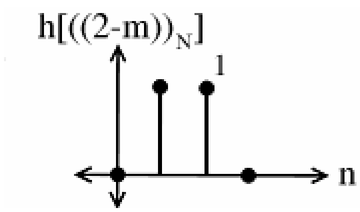



Figure 6

Multiply $f[m]$ and sum to yield: $y[2]$ =3

- $h[3-m]$

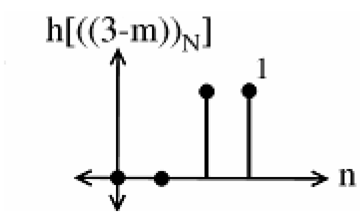



Figure 7

Multiply $f[m]$ and sum to yield: $y[3]$ =1

# Program to Implement Circular Convolution

```
#include<stdio.h>
   int m,n,x[30],h[30],y[30],i,j,temp[30],k,x2[30],a[30];
void main()
{
   printf("  enter the length of the first sequence\n");
   scanf("%d",&m);
   printf("  enter the length of the second sequence\n");
   scanf("%d",&n);
   printf("  enter the first sequence\n");
   for(i=0;i<m;i++)
   scanf("%d",&x[i]);
   printf("  enter the second sequence\n");
   for(j=0;j<n;j++)
   scanf("%d",&h[j]);
   if(m-n!=0)                 /*If length of both sequences are not equal*/
   {
      if(m>n)                 /* Pad the smaller sequence with zero*/
      {
      for(i=n;i<m;i++)
      h[i]=0;
      n=m;
      }
      for(i=m;i<n;i++)
      x[i]=0;
      m=n;
   }
   y[0]=0;
   a[0]=h[0];
   for(j=1;j<n;j++)                                  /*folding h(n) to h(-n)*/
   a[j]=h[n-j];
     /*Circular convolution*/
  for(i=0;i<n;i++)
   y[0]+=x[i]*a[i];
  for(k=1;k<n;k++)
  {
   y[k]=0;
   /*circular shift*/
   for(j=1;j<n;j++)
      x2[j]=a[j-1];
   x2[0]=a[n-1];
   for(i=0;i<n;i++)
    {
      a[i]=x2[i];
      y[k]+=x[i]*x2[i];
    }
 }
  /*displaying the result*/
  printf("  the circular convolution is\n");
  for(i=0;i<n;i++)
  printf("%d \t",y[i]);

  }
```

```
IN PUT:
          Eg:     x[4]={3, 2, 1,0}
                  h[4]={1, 1, 0,0}

OUT PUT           y[4]={3, 5, 3,0}
```

**PROCEDURE:**

- Open Code Composer Studio, make sure the DSP kit is turned on.

- Start a new project using 'Project-new ' pull down menu, save it in a separate directory(c:\ti\myprojects) with name **cir conv.pjt.**

- Add the source files **Circular Convolution.C**
- to the project using 'Project→add files to project' pull down menu.

- Add the linker command file **hello.cmd** .
  (Path: c:\ti\tutorial\dsk6713\hello1\hello.cmd)

- Add the run time support library file **rts6700.lib**
  (Path: c:\ti\c6000\cgtools\lib\rts6700.lib)

- Compile the program using the 'Project-compile' pull down menu or by clicking the shortcut icon on the left side of program window.

- Build the program using the 'Project-Build' pull down menu or by clicking the shortcut icon on the left side of program window.

- Load the program(lconv.out) in program memory of DSP chip using the 'File-load program' pull down menu.

# TMS320C6713 DSK CODEC(TLV320AIC23) Configuration Using Board Support Library

**1.0 Unit Objective:**
To configure the codec TLV320AIC23 for a talk through program using the board support library.

**2.0 Prerequisites**
TMS320C6713 DSP Starter Kit, PC with Code Composer Studio, CRO, Audio Source, Speakers and Signal Generator.

## 3.0 Discussion on Fundamentals:
Refer BSL API Module under, help → contents → TMS320C6713 DSK.

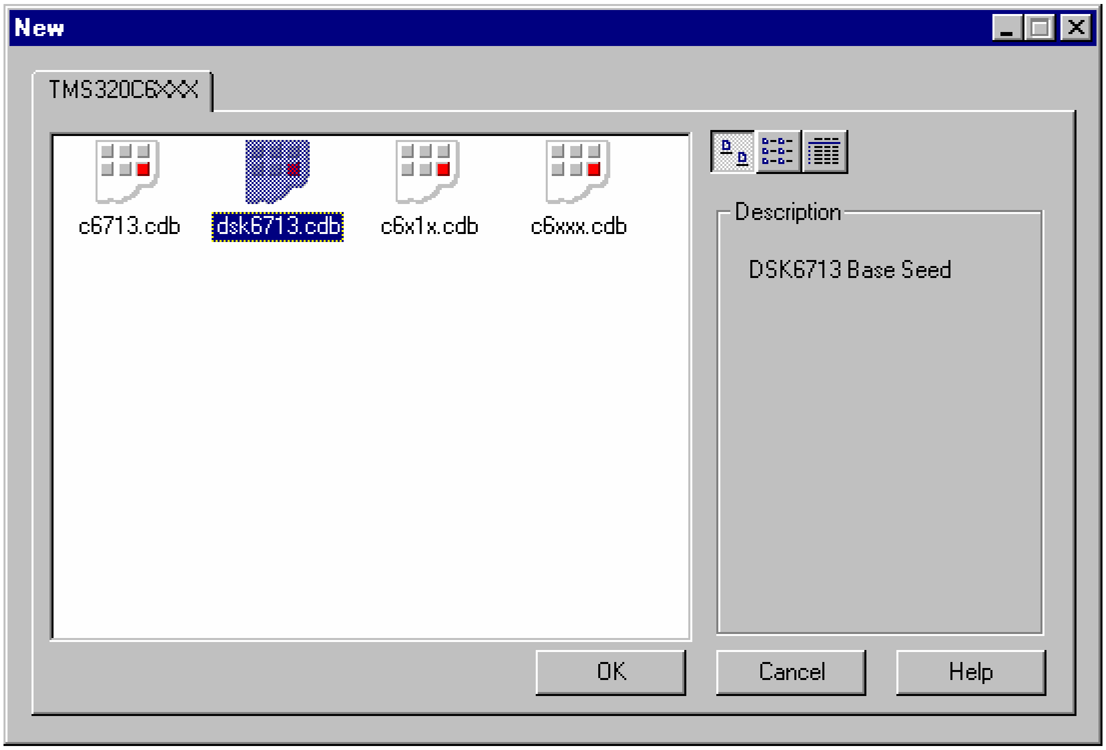
**4.0 Procedure**

- All the Real time implementations covered in the Implementations module follow code Configuration using board support library.
- The board Support Library (CSL) is a collection of functions, macros, and symbols used to configure and control on-chip peripherals.
- The goal is peripheral ease of use, shortened development time, portability, hardware abstraction, and some level of standardization and compatibility among TI devices.
- BSL is a fully scalable component of DSP/BIOS. It does not require the use of other DSP/BIOS components to operate.

**Source Code: codec.c**

**Procedure for Real time Programs :**

1. Connect CRO to the Socket Provided for **LINE OUT**.
2. Connect a Signal Generator to the **LINE IN** Socket.
3. Switch on the Signal Generator with a sine wave of frequency 500 Hz. and Vp-p=1.5v
4. Now Switch on the DSK and Bring Up Code Composer Studio on the PC.
5. Create a new project with name **codec.pjt.**
6. From the **File** Menu ➔ **new** ➔ **DSP/BIOS Configuration** ➔select "**dsk6713.cdb**" and save it as "**xyz.cdb**"




7. Add "**xyz.cdb**" to the current project.

8. Add the given "**codec.c**" file to the current project which has the main function and calls all the other necessary routines.

9. Add the library file "**dsk6713bsl.lib**" to the current project

   **Path →** "**C:\CCStudio\C6000\dsk6713\lib\dsk6713bsl.lib**"

10. Copy files **"dsk6713.h"** and **"dsk6713_aic23.h"** from **C:\CCStudio\C6000\dsk6713\include** and paste it in current project.

11. Build, Load and Run the program.

12. You can notice the input signal of 500 Hz. appearing on the CRO verifying the codec configuration.

13. You can also pass an audio input and hear the output signal through the speakers.

14. You can also vary the sampling frequency using the **DSK6713_AIC23_setFreq** Function in the "**codec.c**" file and repeat the above steps.

**5.0 Conclusion:**

The codec TLV320AIC23 successfully configured using the board support library and verified.

## codec.c

```
#include "xyzcfg.h"

#include "dsk6713.h"
#include "dsk6713_aic23.h"

/* Codec configuration settings */
DSK6713_AIC23_Config config = { \
   0x0017,  /* 0 DSK6713_AIC23_LEFTINVOL  Left line input channel volume */ \
   0x0017,  /* 1 DSK6713_AIC23_RIGHTINVOL Right line input channel volume */\
   0x00d8,  /* 2 DSK6713_AIC23_LEFTHPVOL  Left channel headphone volume */  \
   0x00d8,  /* 3 DSK6713_AIC23_RIGHTHPVOL Right channel headphone volume */ \
   0x0011,  /* 4 DSK6713_AIC23_ANAPATH    Analog audio path control */      \
   0x0000,  /* 5 DSK6713_AIC23_DIGPATH    Digital audio path control */     \
   0x0000,  /* 6 DSK6713_AIC23_POWERDOWN  Power down control */             \
   0x0043,  /* 7 DSK6713_AIC23_DIGIF      Digital audio interface format */ \
   0x0081,  /* 8 DSK6713_AIC23_SAMPLERATE Sample rate control */            \
   0x0001   /* 9 DSK6713_AIC23_DIGACT     Digital interface activation */   \
};

/* main() - Main code routine, initializes BSL and generates tone */

void main()
{
   DSK6713_AIC23_CodecHandle hCodec;
   int l_input, r_input,l_output, r_output;

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

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

    /*set codec sampling frequency*/
   DSK6713_AIC23_setFreq(hCodec, 3);

 while(1)
       {
          /* Read a sample to the left channel */
          while (!DSK6713_AIC23_read(hCodec, &l_input));

          /* Read a sample to the right channel */
          while (!DSK6713_AIC23_read(hCodec, &r_input));

          /* Send a sample to the left channel */
         while (!DSK6713_AIC23_write(hCodec, l_input));

         /* Send a sample to the right channel */
         while (!DSK6713_AIC23_write(hCodec, l_input));
       }

   /* Close the codec */
   DSK6713_AIC23_closeCodec(hCodec);
}
```

# Advance Discrete Time Filter Design(FIR)

## Finite Impulse Response Filter

**DESIGNING AN FIR FILTER :**

Following are the steps to design linear phase FIR filters Using Windowing Method.

I. Clearly specify the filter specifications.
Eg: Order = 30;
Sampling Rate = 8000 samples/sec
Cut off Freq. = 400 Hz.

II. Compute the cut-off frequency $W_c$
Eg: $W_c$ = 2*pie* $f_c$ / $F_s$
= 2*pie* 400/8000
= 0.1*pie

III. Compute the desired Impulse Response $h_d(n)$ using particular Window
Eg: b_rect1=fir1(order, $W_c$ , 'high',boxcar(31));

IV. Convolve input sequence with truncated Impulse Response x (n)*h (n)

**USING MATLAB TO DETERMINE FILTER COEFFICIENTS :**
***Using FIR1 Function on Matlab***

*B = FIR1(N,Wn) designs an N'th order lowpass FIR digital filter and returns the filter coefficients in length N+1 vector B.*

*The cut-off frequency Wn must be between 0 < Wn < 1.0, with 1.0 corresponding to half the sample rate. The filter B is real and has linear phase, i.e., even symmetric coefficients obeying B(k) = B(N+2-k), k = 1,2,...,N+1.*

*If Wn is a two-element vector, Wn = [W1 W2], FIR1 returns an order N bandpass filter with passband W1 < W < W2.*
*B = FIR1(N,Wn,'high') designs a highpass filter.*
*B = FIR1(N,Wn,'stop') is a bandstop filter if Wn = [W1 W2].*

*If Wn is a multi-element vector,*
*Wn = [W1 W2 W3 W4 W5 ... WN],*
*FIR1 returns an order N multiband filter with bands*
*0 < W < W1, W1 < W < W2, ..., WN < W < 1.*
*B = FIR1(N,Wn,'DC-1') makes the first band a passband.*
*B = FIR1(N,Wn,'DC-0') makes the first band a stopband.*

*For filters with a passband near Fs/2, e.g., highpass and bandstop filters, N must be even.*

*By default FIR1 uses a Hamming window. Other available windows, including Boxcar, Hanning, Bartlett, Blackman, Kaiser and Chebwin can be specified with an optional trailing argument. For example, B = FIR1(N,Wn,kaiser(N+1,4)) uses a Kaiser window with beta=4. B = FIR1(N,Wn,'high',chebwin(N+1,R)) uses a Chebyshev window.*

*By default, the filter is scaled so the center of the first pass band has magnitude exactly one after windowing. Use a trailing 'noscale' argument to prevent this scaling, e.g. B = FIR1(N,Wn,'noscale'), B = FIR1(N,Wn,'high','noscale'), B = FIR1(N,Wn,wind,'noscale').*

## ***Matlab Program to generate 'FIR Filter-Low Pass' Coefficients using FIR1***

```
% FIR Low  pass filters using rectangular, triangular and kaiser windows

% sampling rate - 8000

order = 30;
cf=[500/4000,1000/4000,1500/4000];     cf--> contains set of cut-off  frequencies[Wc ]

% cutoff frequency - 500
b_rect1=fir1(order,cf(1),boxcar(31));  Rectangular
b_tri1=fir1(order,cf(1),bartlett(31));    Triangular
b_kai1=fir1(order,cf(1),kaiser(31,8)); Kaisar [Where 8-->Beta Co-efficient]


% cutoff frequency - 1000
b_rect2=fir1(order,cf(2),boxcar(31));
b_tri2=fir1(order,cf(2),bartlett(31));
b_kai2=fir1(order,cf(2),kaiser(31,8));


% cutoff frequency - 1500
b_rect3=fir1(order,cf(3),boxcar(31));
b_tri3=fir1(order,cf(3),bartlett(31));
b_kai3=fir1(order,cf(3),kaiser(31,8));


fid=fopen('FIR_lowpass_rectangular.txt','wt');
fprintf(fid,'\t\t\t\t\t\t%s\n','Cutoff -400Hz');
fprintf(fid,'\nfloat b_rect1[31]={');
fprintf(fid,'%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,\n',b_rect1);
fseek(fid,-1,0);
fprintf(fid,'};');

fprintf(fid,'\n\n\n\n');
fprintf(fid,'\t\t\t\t\t\t%s\n','Cutoff -800Hz');
```

```
fprintf(fid,'\nfloat b_rect2[31]={');
fprintf(fid,'%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,\n',b_rect2);
fseek(fid,-1,0);
fprintf(fid,'};');

fprintf(fid,'\n\n\n\n');
fprintf(fid,'\t\t\t\t\t\t%s\n','Cutoff -1200Hz');
fprintf(fid,'\nfloat b_rect3[31]={');
fprintf(fid,'%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,\n',b_rect3);
fseek(fid,-1,0);
fprintf(fid,'};');

fclose(fid);
winopen('FIR_highpass_rectangular.txt');
```

## T.1 : Matlab generated Coefficients for FIR Low Pass Kaiser filter:

**Cutoff -500Hz**

float b_kai1[31]={-0.000019,-0.000170,-0.000609,-0.001451,-0.002593,-0.003511,-0.003150,0.000000,0.007551,0.020655,0.039383,0.062306,0.086494,0.108031,0.122944,0.128279,0.122944,0.108031,0.086494,0.062306,0.039383,0.020655,0.007551,0.000000,-0.003150,-0.003511,-0.002593,-0.001451,-0.000609,-0.000170,-0.000019};

**Cutoff -1000Hz**

float b_kai2[31]={-0.000035,-0.000234,-0.000454,0.000000,0.001933,0.004838,0.005671,-0.000000,-0.013596,-0.028462,-0.029370,0.000000,0.064504,0.148863,0.221349,0.249983,0.221349,0.148863,0.064504,0.000000,-0.029370,-0.028462,-0.013596,-0.000000,0.005671,0.004838,0.001933,0.000000,-0.000454,-0.000234, -0.000035};

**Cutoff -1500Hz**

float b_kai3[31]={-0.000046,-0.000166,0.000246,0.001414,0.001046,-0.003421,-0.007410,0.000000,0.017764,0.020126,-0.015895,-0.060710,-0.034909,0.105263,0.289209,0.374978,0.289209,0.105263,-0.034909,-0.060710,-0.015895,0.020126,0.017764,0.000000,-0.007410,-0.003421,0.001046,0.001414,0.000246,-0.000166, -0.000046};

## T.2 :Matlab generated Coefficients for FIR Low Pass Rectangular filter

**Cutoff -500Hz**

float b_rect1[31]={-0.008982,-0.017782,-0.025020,-0.029339,-0.029569,-0.024895,
-0.014970,0.000000,0.019247,0.041491,0.065053,0.088016,0.108421,0.124473,0.134729,
0.138255,0.134729,0.124473,0.108421,0.088016,0.065053,0.041491,0.019247,0.000000,
-0.014970,-0.024895,-0.029569,-0.029339,-0.025020,-0.017782,-0.008982};

**Cutoff -1000Hz**

float b_rect2[31]={-0.015752,-0.023869,-0.018176,0.000000,0.021481,0.033416,0.026254,-
0.000000,-0.033755,-0.055693,-0.047257,0.000000,0.078762,0.167080,0.236286,0.262448,
0.236286,0.167080,0.078762,0.000000,-0.047257,-0.055693,-0.033755,-0.000000,0.026254,
0.033416,0.021481,0.000000,-0.018176,-0.023869,-0.015752};

**Cutoff -1500Hz**

float b_rect2[31]={-0.020203,-0.016567,0.009656,0.027335,0.011411,-0.023194,-0.033672,
0.000000,0.043293,0.038657,-0.025105,-0.082004,-0.041842,0.115971,0.303048,0.386435,
0.303048,0.115971,-0.041842,-0.082004,-0.025105,0.038657,0.043293,0.000000,-0.033672,
-0.023194,0.011411,0.027335,0.009656,-0.016567,-0.020203};

## T.3 : Matlab generated Coefficients for FIR Low Pass Triangular filter

**Cutoff -500Hz**

float b_tri1[31]={0.000000,-0.001185,-0.003336,-0.005868,-0.007885,-0.008298,-0.005988,
0.000000,0.010265,0.024895,0.043368,0.064545,0.086737,0.107877,0.125747,0.138255,
0.125747,0.107877,0.086737,0.064545,0.043368,0.024895,0.010265,0.000000,-0.005988,
-0.008298,-0.007885,-0.005868,-0.003336,-0.001185,0.000000};

**Cutoff -1000Hz**

float b_tri2[31]={0.000000,-0.001591,-0.002423,0.000000,0.005728,0.011139,0.010502,
-0.000000,-0.018003,-0.033416,-0.031505,0.000000,0.063010,0.144802,0.220534,0.262448,
0.220534,0.144802,0.063010,0.000000,-0.031505,-0.033416,-0.018003,-0.000000,0.010502,
0.011139,0.005728,0.000000,-0.002423,-0.001591,0.000000};

**Cutoff -1500Hz**

float b_tri3[31]={0.000000,-0.001104,0.001287,0.005467,0.003043,-0.007731,-0.013469,
0.000000,0.023089,0.023194,-0.016737,-0.060136,-0.033474,0.100508,0.282844,0.386435,
0.282844,0.100508,-0.033474,-0.060136,-0.016737,0.023194,0.023089,0.000000,-0.013469,
-0.007731,0.003043,0.005467,0.001287,-0.001104,0.000000};

## *MATLAB Program to generate 'FIR Filter-High Pass' Coefficients using FIR1*

```
% FIR High pass filters using rectangular, triangular and kaiser windows

% sampling rate - 8000
order = 30;

cf=[400/4000,800/4000,1200/4000];                ;cf--> contains set of cut-off frequencies[Wc]

% cutoff frequency - 400
b_rect1=fir1(order,cf(1),'high',boxcar(31));
b_tri1=fir1(order,cf(1),'high',bartlett(31));
b_kai1=fir1(order,cf(1),'high',kaiser(31,8)); Where Kaiser(31,8)--> '8'defines the value of 'beta'.

% cutoff frequency - 800
b_rect2=fir1(order,cf(2),'high',boxcar(31));
b_tri2=fir1(order,cf(2),'high',bartlett(31));
b_kai2=fir1(order,cf(2),'high',kaiser(31,8));

% cutoff frequency - 1200
b_rect3=fir1(order,cf(3),'high',boxcar(31));
b_tri3=fir1(order,cf(3),'high',bartlett(31));
b_kai3=fir1(order,cf(3),'high',kaiser(31,8));

fid=fopen('FIR_highpass_rectangular.txt','wt');
fprintf(fid,'\t\t\t\t\t\t%s\n','Cutoff -400Hz');
fprintf(fid,'\nfloat b_rect1[31]={');
fprintf(fid,'%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,\n',b_rect1);
fseek(fid,-1,0);
fprintf(fid,'};');

fprintf(fid,'\n\n\n\n');
fprintf(fid,'\t\t\t\t\t\t%s\n','Cutoff -800Hz');
fprintf(fid,'\nfloat b_rect2[31]={');
fprintf(fid,'%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,\n',b_rect2);
fseek(fid,-1,0);
fprintf(fid,'};');

fprintf(fid,'\n\n\n\n');
fprintf(fid,'\t\t\t\t\t\t%s\n','Cutoff -1200Hz');
fprintf(fid,'\nfloat b_rect3[31]={');
fprintf(fid,'%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,\n',b_rect3);
fseek(fid,-1,0);
fprintf(fid,'};');

fclose(fid);
winopen('FIR_highpass_rectangular.txt');
```

## T.1 : MATLAB generated Coefficients for FIR High Pass Kaiser filter:

**Cutoff -400Hz**

```
float b_kai1[31]={0.000050,0.000223,0.000520,0.000831,0.000845,-0.000000,-0.002478,
-0.007437,-0.015556,-0.027071,-0.041538,-0.057742,-0.073805,-0.087505,-0.096739,
0.899998,-0.096739,-0.087505,-0.073805,-0.057742,-0.041538,-0.027071,-0.015556,
-0.007437,-0.002478,-0.000000,0.000845,0.000831,0.000520,0.000223,0.000050};
```

**Cutoff -800Hz**

```
float b_kai2[31]={0.000000,-0.000138,-0.000611,-0.001345,-0.001607,-0.000000,0.004714,
0.012033,0.018287,0.016731,0.000000,-0.035687,-0.086763,-0.141588,-0.184011,0.800005,
-0.184011,-0.141588,-0.086763,-0.035687,0.000000,0.016731,0.018287,0.012033,0.004714,
-0.000000,-0.001607,-0.001345,-0.000611,-0.000138,0.000000};
```

**Cutoff -1200Hz**

```
float b_kai3[31]={-0.000050,-0.000138,0.000198,0.001345,0.002212,-0.000000,-0.006489,
-0.012033,-0.005942,0.016731,0.041539,0.035687,-0.028191,-0.141589,-0.253270,0.700008,
-0.253270,-0.141589,-0.028191,0.035687,0.041539,0.016731,-0.005942,-0.012033,-0.006489,
-0.000000,0.002212,0.001345,0.000198,-0.000138,-0.000050};
```

## T.2 :MATLAB generated Coefficients for FIR High Pass Rectangular filter

**Cutoff -400Hz**

```
float b_rect1[31]={0.021665,0.022076,0.020224,0.015918,0.009129,-0.000000,-0.011158,
-0.023877,-0.037558,-0.051511,-0.064994,-0.077266,-0.087636,-0.095507,-.100422,0.918834,
-0.100422,-0.095507,-0.087636,-0.077266,-0.064994,-0.051511,-0.037558,-0.023877,
-0.011158,-0.000000,0.009129,0.015918,0.020224,0.022076,0.021665};
```

**Cutoff -800Hz**

```
float b_rect2[31]={0.000000,-0.013457,-0.023448,-0.025402,-0.017127,-0.000000,0.020933,
0.038103,0.043547,0.031399,0.000000,-0.047098,-0.101609,-0.152414,-0.188394,0.805541,
-0.188394,-0.152414,-0.101609,-0.047098,0.000000,0.031399,0.043547,0.038103,0.020933,
-0.000000,-0.017127,-0.025402,-0.023448,-0.013457,0.000000};
```

**Cutoff -1200Hz**

```
float b_rect3[31]={-0.020798,-0.013098,0.007416,0.024725,0.022944,-0.000000,-0.028043,
-0.037087,-0.013772,0.030562,0.062393,0.045842,-0.032134,-0.148349,-0.252386,0.686050,
-0.252386,-0.148349,-0.032134,0.045842,0.062393,0.030562,-0.013772,-0.037087,-0.028043,
-0.000000,0.022944,0.024725,0.007416,-0.013098,-0.020798};
```

## T.3 : MATLAB generated Coefficients for FIR High Pass Triangular filter

**Cutoff -400Hz**

```
float b_tri1[31]={0.000000,0.001445,0.002648,0.003127,0.002391,-0.000000,-0.004383,
-0.010943,-0.019672,-0.030353,-0.042554,-0.055647,-0.068853,-0.081290,-0.092048,
0.902380,-0.092048,-0.081290,-0.068853,-0.055647,-0.042554,-0.030353,-0.019672,
-0.010943,-0.004383,-0.000000,0.002391,0.003127,0.002648,0.001445,0.000000};
```

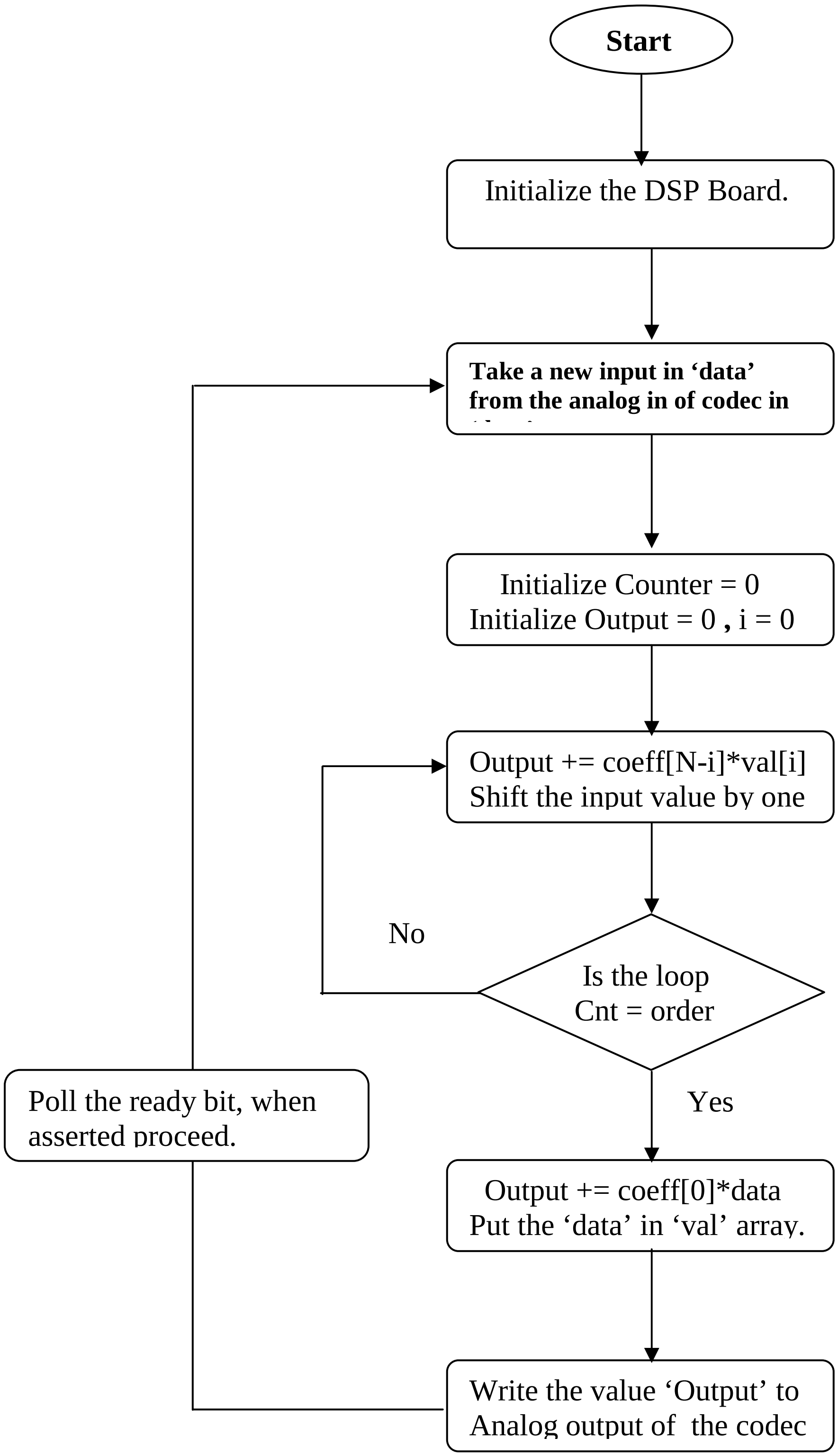
**Cutoff -800Hz**

```
float b_tri2[31]={0.000000,-0.000897,-0.003126,-0.005080,-0.004567,-0.000000,0.008373,
0.017782,0.023225,0.018839,0.000000,-0.034539,-0.081287,-0.132092,-0.175834,0.805541,
-0.175834,-0.132092,-0.081287,-0.034539,0.000000,0.018839,0.023225,0.017782,0.008373,
-0.000000,-0.004567,-0.005080,-0.003126,-0.000897,0.000000};
```

**Cutoff -1200Hz**

```
float b_tri3[31]={0.000000,-0.000901,0.001021,0.005105,0.006317,-0.000000,-0.011581,
-0.017868,-0.007583,0.018931,0.042944,0.034707,-0.026541,-0.132736,-0.243196,0.708287,
-0.243196,-0.132736,-0.026541,0.034707,0.042944,0.018931,-0.007583,-0.017868,-0.011581,
-0.000000,0.006317,0.005105,0.001021,-0.000901,0.000000};
```

## FLOW CHART TO IMPLEMENT FIR FILTER:



## C PROGRAM TO IMPLEMENT FIR FILTER:

fir.c

```
#include "filtercfg.h"

#include "dsk6713.h"
#include "dsk6713_aic23.h"

float filter_Coeff[] ={0.000000,-0.001591,-0.002423,0.000000,0.005728,
0.011139,0.010502,-0.000000,-0.018003,-0.033416,-0.031505,0.000000,
0.063010,0.144802,0.220534,0.262448,0.220534,0.144802,0.063010,0.000000,
-0.031505,-0.033416,-0.018003,-0.000000,0.010502,0.011139,0.005728,
0.000000,-0.002423,-0.001591,0.000000 };

static short in_buffer[100];

DSK6713_AIC23_Config config = {\
    0x0017, /* 0 DSK6713_AIC23_LEFTINVOL Leftline input channel volume */\
    0x0017, /* 1 DSK6713_AIC23_RIGHTINVOL Right line input channel volume*/\
    0x00d8, /* 2 DSK6713_AIC23_LEFTHPVOL  Left channel headphone volume */\
    0x00d8, /* 3 DSK6713_AIC23_RIGHTHPVOL Right channel headphone volume */\
    0x0011,  /* 4 DSK6713_AIC23_ANAPATH    Analog audio path control */\
    0x0000,  /* 5 DSK6713_AIC23_DIGPATH    Digital audio path control */\
    0x0000,  /* 6 DSK6713_AIC23_POWERDOWN  Power down control */\
    0x0043,  /* 7 DSK6713_AIC23_DIGIF    Digital audio interface format */\
    0x0081,  /* 8 DSK6713_AIC23_SAMPLERATE Sample rate control */\
    0x0001   /* 9 DSK6713_AIC23_DIGACT   Digital interface activation */   \
};

/*
 *  main() - Main code routine, initializes BSL and generates tone
 */

void main()
{
    DSK6713_AIC23_CodecHandle hCodec;

    Uint32 l_input, r_input,l_output, r_output;

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

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

    DSK6713_AIC23_setFreq(hCodec, 1);

       while(1)
        {   /* Read a sample to the left channel */
                  while (!DSK6713_AIC23_read(hCodec, &l_input));

                  /* Read a sample to the right channel */
                  while (!DSK6713_AIC23_read(hCodec, &r_input));

                        l_output=(Int16)FIR_FILTER(&filter_Coeff ,l_input);
```

```
                r_output=l_output;

                    /* Send a sample to the left channel */
            while (!DSK6713_AIC23_write(hCodec, l_output));

            /* Send a sample to the right channel */
            while (!DSK6713_AIC23_write(hCodec, r_output));
        }

    /* Close the codec */
    DSK6713_AIC23_closeCodec(hCodec);
}

signed int FIR_FILTER(float * h, signed int x)
{
int i=0;
signed long output=0;

in_buffer[0] = x;  /* new input at buffer[0]  */

for(i=29;i>0;i--)
in_buffer[i] = in_buffer[i-1]; /* shuffle the buffer   */

for(i=0;i<31;i++)
output = output + h[i] * in_buffer[i];

return(output);

}
```

## PROCEDURE :

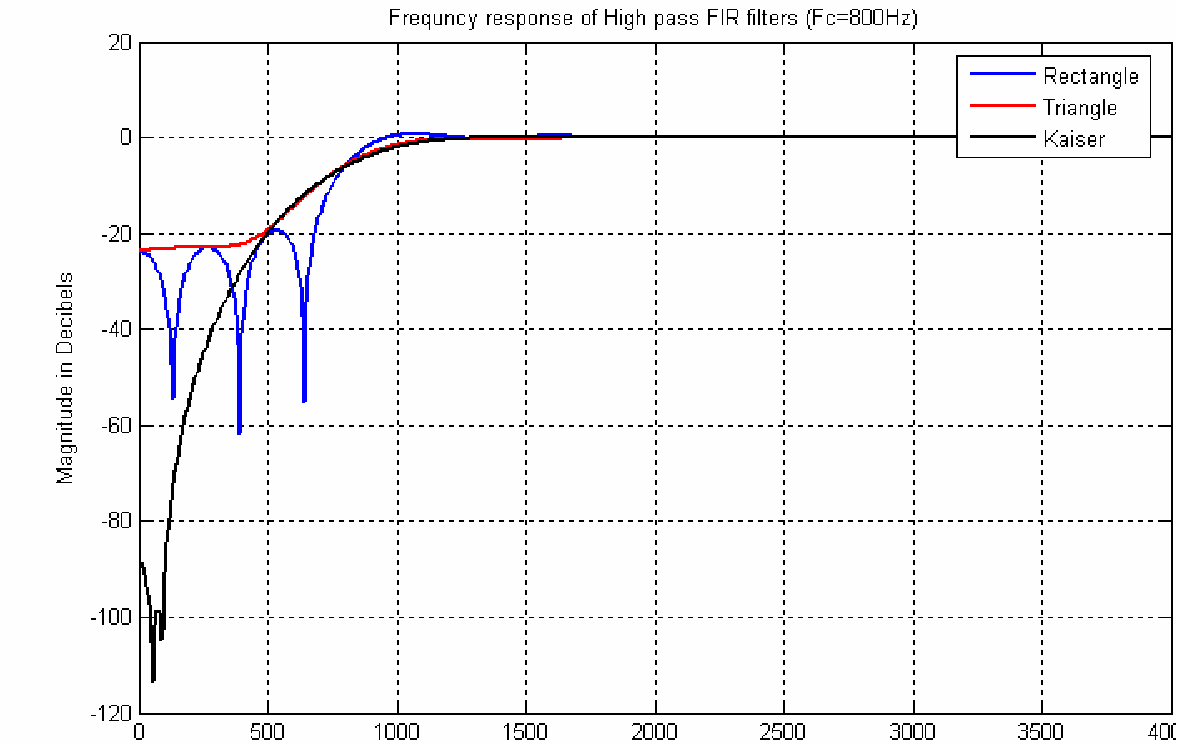
- Switch on the DSP board.
- Open the Code Composer Studio.
- Create a new project  
  Project → New (File Name. pjt , Eg: **FIR.pjt**)
- Initialize on board codec.

  ***Note:*** *"Kindly refer the Topic* ***Configuration of 6713 Codec*** *using BSL"*

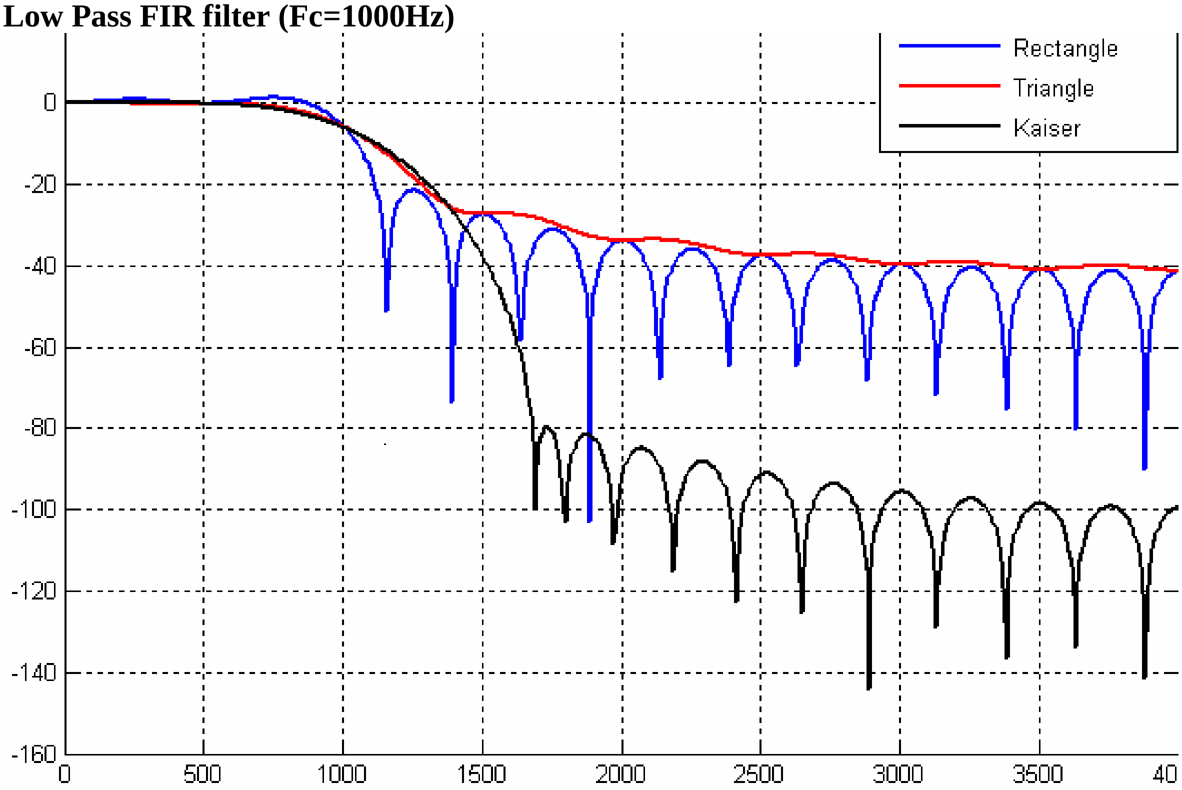
- Add the given above 'C' source file to the current project **(remove codec.c source file from the project if you have already added).**
- Connect the speaker jack to the input of the CRO.
- Build the program.
- Load the generated object file(*.out) on to Target board.
- Run the program
- Observe the waveform that appears on the CRO screen.
- Vary the frequency on function generator to see the response of filter.

# MATLAB GENERATED FREQUENCY RESPONSE

## High Pass FIR filter(Fc= 800Hz).




## Low Pass FIR filter (Fc=1000Hz)



# Advance Discrete Time Filter Design(IIR)

## IIR filter Designing Experiments

**GENERAL CONSIDERATIONS:**

In the design of frequency – selective filters, the desired filter characteristics are specified in the frequency domain in terms of the desired magnitude and phase response of the filter. In the filter design process, we determine the coefficients of a causal IIR filter that closely approximates the desired frequency response specifications.

**IMPLEMENTATION OF DISCRETE-TIME SYSTEMS:**

Discrete time Linear Time-Invariant (LTI) systems can be described completely by constant coefficient linear difference equations. Representing a system in terms of constant coefficient linear difference equation is it’s time domain characterization. In the design of a simple frequency–selective filter, we would take help of some basic implementation methods for realizations of LTI systems described by linear constant coefficient difference equation.

**UNIT OBJECTIVE:**

The aim of this laboratory exercise is to design and implement a Digital IIR Filter & observe its frequency response. In this experiment we design a simple IIR filter so as to stop or attenuate required band of frequencies components and pass the frequency components which are outside the required band.

**BACKGROUND CONCEPTS:**

An Infinite impulse response (IIR) filter possesses an output response to an impulse which is of an infinite duration. The impulse response is "infinite" since there is feedback in the filter, that is if you put in an impulse ,then its output must produced for infinite duration of time.

**PREREQUISITES:**

* Concept of Discrete time signal processing.
* Analog filter design concepts.
* TMS320C6713 Architecture and instruction set.

**EQUIPMENTS NEEDED:**

- Host (PC) with windows(95/98/Me/XP/NT/2000).
- TMS320C6713 DSP Starter Kit (DSK).
- Oscilloscope and Function generator.

**ALGORITHM TO IMPLEMENT:**

We need to realize the Butter worth band pass IIR filter by implementing the difference equation $y[n] = b_0x[n] + b_1x[n-1]+b_2x[n-2]-a_1y[n-1]-a_2y[n-2]$ where $b_0 – b_2$, $a_0$-$a_2$ are feed forward and feedback word coefficients respectively [Assume $2^{nd}$ order of filter].These coefficients are calculated using MATLAB.A direct **form I** implementation approach is taken.

- **Step 1 -** Initialize the McBSP, the DSP board and the on board codec.
  "Kindly refer the Topic **Configuration of 6713Codec** using BSL"

- **Step 2 -** Initialize the discrete time system , that is , specify the initial conditions. Generally zero initial conditions are assumed.

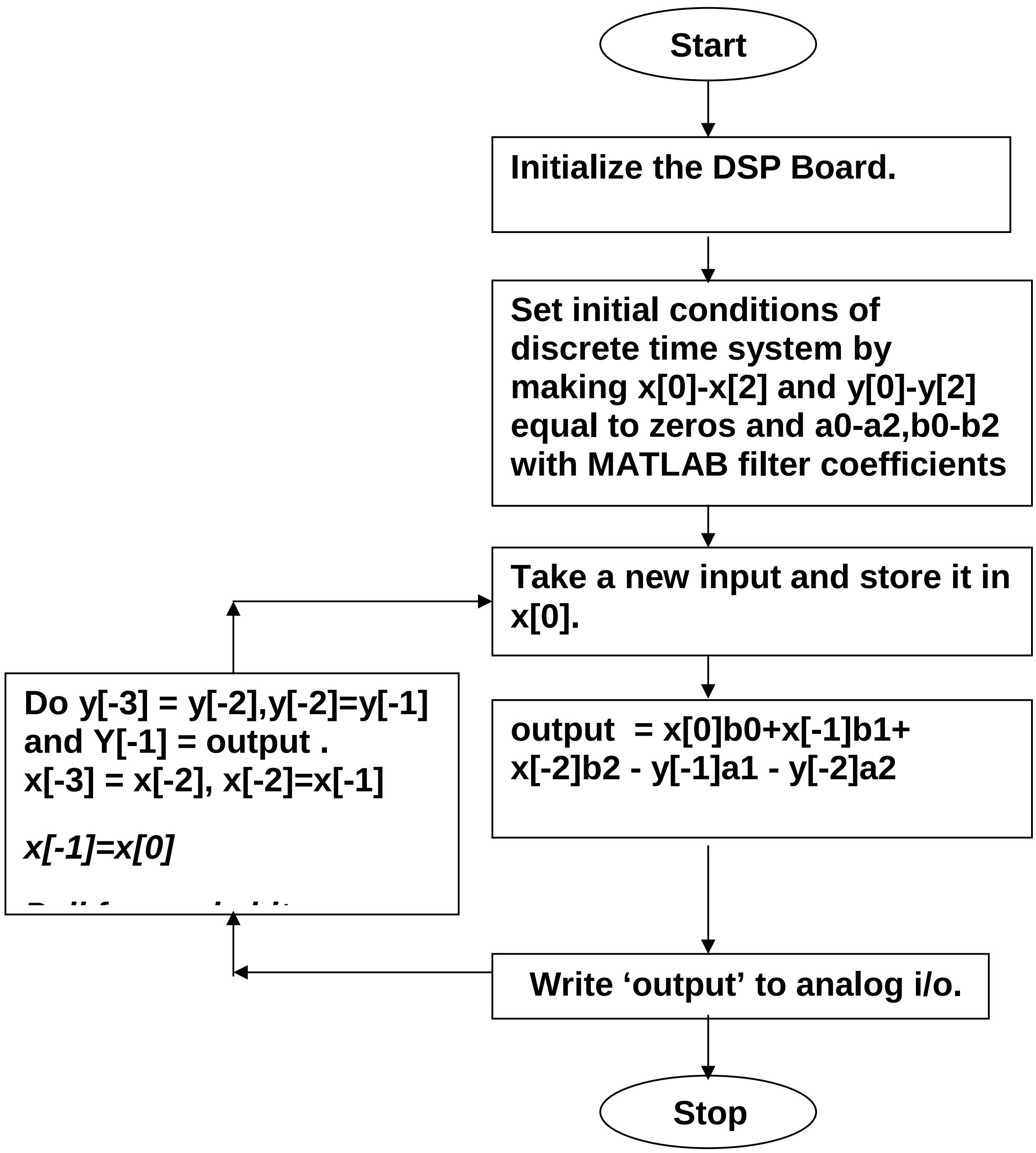
- **Step 3 -** Take sampled data from codec while input is fed to DSP kit from the signal generator. Since Codec is stereo , take average of input data read from left and right channel . Store sampled data at a memory location.

- **Step 4** - Perform filter operation using above said difference equation and store filter Output at a memory location .

- **Step 5 -** Output the value to codec (left channel and right channel) and view the output at Oscilloscope.

- **Step 6 -** Go to step 3.

**FLOWCHART FOR IIR IMPLEMENTATION:**




F.1 : Flowchart for implementing IIR filter.

## *MATLAB PROGRAM TO GENRATE FILTER CO-EFFICIENTS*

```
% IIR Low pass Butterworth and Chebyshev filters
% sampling rate - 24000

order = 2;
cf=[2500/12000,8000/12000,1600/12000];

% cutoff frequency - 2500
[num_bw1,den_bw1]=butter(order,cf(1));
[num_cb1,den_cb1]=cheby1(order,3,cf(1));

% cutoff frequency - 8000
[num_bw2,den_bw2]=butter(order,cf(2));
[num_cb2,den_cb2]=cheby1(order,3,cf(2));

fid=fopen('IIR_LP_BW.txt','wt');
fprintf(fid,'\t\t-----------Pass band range: 0-2500Hz----------\n');
fprintf(fid,'\t\t-----------Magnitude response: Monotonic-----\n\n\');
fprintf(fid,'\n float num_bw1[9]={');
fprintf(fid,'%f,%f,%f,%f,%f,\n%f,%f,%f,%f};\n',num_bw1);
fprintf(fid,'\nfloat den_bw1[9]={');
fprintf(fid,'%f,%f,%f,%f,%f,\n%f,%f,%f,%f};\n',den_bw1);

fprintf(fid,'\n\n\n\t\t-----------Pass band range: 0-8000Hz----------\n');
fprintf(fid,'\t\t-----------Magnitude response: Monotonic-----\n\n');
fprintf(fid,'\nfloat num_bw2[9]={');
fprintf(fid,'%f,%f,%f,%f,%f,\n%f,%f,%f,%f};\n',num_bw2);
fprintf(fid,'\nfloat den_bw2[9]={');
fprintf(fid,'%f,%f,%f,%f,%f,\n%f,%f,%f,%f};\n',den_bw2);

fclose(fid);
winopen('IIR_LP_BW.txt');

fid=fopen('IIR_LP_CHEB Type1.txt','wt');
fprintf(fid,'\t\t-----------Pass band range: 2500Hz----------\n');
fprintf(fid,'\t\t-----------Magnitude response: Rippled (3dB) -----\n\n\');
fprintf(fid,'\nfloat num_cb1[9]={');
fprintf(fid,'%f,%f,%f,%f,%f,\n%f,%f,%f,%f};\n',num_cb1);
fprintf(fid,'\nfloat den_cb1[9]={');
fprintf(fid,'%f,%f,%f,%f,%f,\n%f,%f,%f,%f};\n',den_cb1);
fprintf(fid,'\n\n\n\t\t-----------Pass band range: 8000Hz----------\n');
fprintf(fid,'\t\t-----------Magnitude response: Rippled (3dB)-----\n\n');
fprintf(fid,'\nfloat num_cb2[9]={');
fprintf(fid,'%f,%f,%f,%f,%f,\n%f,%f,%f,%f};\n',num_cb2);
fprintf(fid,'\nfloat den_cb2[9]={');
fprintf(fid,'%f,%f,%f,%f,%f,\n%f,%f,%f,%f};\n',den_cb2);
```

```
fclose(fid);
winopen('IIR_LP_CHEB Type1.txt');

%%%%%%%%%%%%%%%%%%%%
figure(1);
[h,w]=freqz(num_bw1,den_bw1);
w=(w/max(w))*12000;
plot(w,20*log10(abs(h)),'linewidth',2)
hold on
[h,w]=freqz(num_cb1,den_cb1);
w=(w/max(w))*12000;
plot(w,20*log10(abs(h)),'linewidth',2,'color','r')
grid on
legend('Butterworth','Chebyshev Type-1');
xlabel('Frequency in Hertz');
ylabel('Magnitude in Decibels');
title('Magnitude response of Low pass IIR filters (Fc=2500Hz)');


figure(2);
[h,w]=freqz(num_bw2,den_bw2);
w=(w/max(w))*12000;
plot(w,20*log10(abs(h)),'linewidth',2)
hold on
[h,w]=freqz(num_cb2,den_cb2);
w=(w/max(w))*12000;
plot(w,20*log10(abs(h)),'linewidth',2,'color','r')
grid on
legend('Butterworth','Chebyshev Type-1 (Ripple: 3dB)');
xlabel('Frequency in Hertz');
ylabel('Magnitude in Decibels');
title('Magnitude response in the passband');
axis([0 12000 -20 20]);
```

## IIR_CHEB_LP FILTER CO-EFFICIENTS:

| Co-Efficients | Fc=2500Hz | | Fc=800Hz | | Fc=8000Hz | |
|---|---|---|---|---|---|---|
| | Floating Point Values | Fixed Point Values(Q15) | Floating Point Values | Fixed Point Values(Q15) | Floating Point Values | Fixed Point Values(Q15) |
| **B0** | 0.044408 | 1455 | 0.005147 | 168 | 0.354544 | 11617 |
| **B1** | 0.088815 | 1455[B1/2] | 0.010295 | 168[B1/2] | 0.709088 | 11617[B1/2] |
| **B2** | 0.044408 | 1455 | 0.005147 | 168 | 0.354544 | 11617 |
| **A0** | 1.000000 | 32767 | 1.000000 | 32767 | 1.000000 | 32767 |
| **A1** | -1.412427 | -23140[A1/2] | -1.844881 | -30225[A1/2] | 0.530009 | 8683[A1/2] |
| **A2** | 0.663336 | 21735 | 0.873965 | 28637 | 0.473218 | 15506 |

**Note: We have Multiplied Floating Point Values with 32767($2^{15}$) to get Fixed Point Values.**

## IIR_BUTTERWORTH_LP FILTER CO-EFFICIENTS:

| Co-Efficients | Fc=2500Hz | | Fc=800Hz | | Fc=8000Hz | |
|---|---|---|---|---|---|---|
| | Floating Point Values | Fixed Point Values(Q15) | Floating Point Values | Fixed Point Values(Q15) | Floating Point Values | Fixed Point Values(Q15) |
| B0 | 0.072231 | 2366 | 0.009526 | 312 | 0.465153 | 15241 |
| B1 | 0.144462 | 2366[B1/2] | 0.019052 | 312[B1/2] | 0.930306 | 15241[B1/2] |
| B2 | 0.072231 | 2366 | 0.009526 | 312 | 0.465153 | 15241 |
| A0 | 1.000000 | 32767 | 1.000000 | 32767 | 1.000000 | 32767 |
| A1 | -1.109229 | -18179[A1/2] | -1.705552, | -27943[A1/2] | 0.620204 | 10161[A1/2] |
| A2 | 0.398152 | 13046 | 0.743655 | 24367 | 0.240408 | 7877 |

**Note: We have Multiplied Floating Point Values with 32767($2^{15}$) to get Fixed Point Values.**

## IIR_CHEB_HP FILTER CO-EFFICIENTS:

| Co-Efficients | Fc=2500Hz | | Fc=4000Hz | | Fc=7000Hz | |
|---|---|---|---|---|---|---|
| | Floating Point Values | Fixed Point Values(Q15) | Floating Point Values | Fixed Point Values(Q15) | Floating Point Values | Fixed Point Values(Q15) |
| B0 | 0.388513 | 12730 | 0.282850 | 9268 | 0.117279 | 3842 |
| B1 | -0.777027 | -12730[B1/2] | -0.565700 | -9268[B1/2] | -0.234557 | -3842[B1/2] |
| B2 | 0.388513 | 12730 | 0.282850 | 9268 | 0.117279 | 3842 |
| A0 | 1.000000 | 32767 | 1.000000 | 32767 | 1.000000 | 32767 |
| A1 | -1.118450 | -18324[A1/2] | -0.451410 | -7395[A1/2] | 0.754476 | 12360[A1/2] |
| A2 | 0.645091 | 21137 | 0.560534 | 18367 | 0.588691 | 19289 |

**Note: We have Multiplied Floating Point Values with 32767($2^{15}$) to get Fixed Point Values.**

## IIR_BUTTERWORTH_HP FILTER CO-EFFICIENTS:

| Co-Efficients | Fc=2500Hz | | Fc=4000Hz | | Fc=7000Hz | |
|---|---|---|---|---|---|---|
| | Floating Point Values | Fixed Point Values(Q15) | Floating Point Values | Fixed Point Values(Q15) | Floating Point Values | Fixed Point Values(Q15) |
| B0 | 0.626845 | 20539 | 0.465153 | 15241 | 0.220195 | 7215 |
| B1 | -1.253691 | -20539[B1/2] | -0.930306 | -15241[B1/2] | -0.440389 | -7215[B1/2] |
| B2 | 0.626845 | 20539 | 0.465153 | 15241 | 0.220195 | 7215 |
| A0 | 1.000000 | 32767 | 1.000000 | 32767 | 1.000000 | 32767 |
| A1 | -1.109229 | -18173[A1/2] | -0.620204 | -10161[A1/2] | 0.307566 | 5039[A1/2} |
| A2 | 0.398152 | 13046 | 0.240408 | 7877 | 0.188345 | 6171 |

**Note: We have Multiplied Floating Point Values with 32767($2^{15}$) to get Fixed Point Values.**

# Fast Fourier Transforms(FFT)

## The DFT Equation

$$X(k) = \frac{1}{N}\sum_{n=0}^{N-1} x(n) W_N^{nk}$$

$$where \quad W_N^{nk} = e^{-j\frac{2\pi nk}{N}} \ \text{[TWIDDLE FACTOR]}$$

---

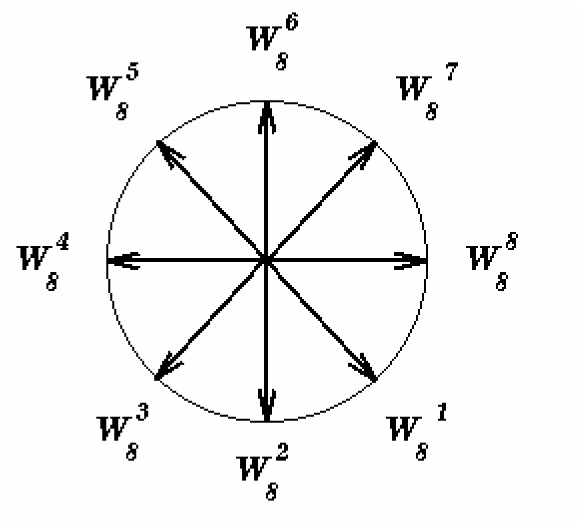
***Twiddle Factor***

In the Definition of the DFT, there is a factor called the *Twiddle Factor*

$$W_N^{nk} = e^{-j\frac{2\pi nk}{N}}$$

where N = number of samples.

If we take an 8 bit sample sequence we can represent the twiddle factor as a vector in the unit circle. e.g.




Note that

1. It is periodic. (i.e. it goes round and round the circle !!)
2. That the vectors are symmetric
3. The vectors are equally spaced around the circle.

***Why the FFT?***

If you look at the equation for the *Discrete Fourier Transform* you will see that it is quite complicated to work out as it involves many additions and multiplications involving complex numbers. Even a simple eight sample signal would require 49 complex multiplications and 56 complex additions to work out the DFT. At this level it is still manageable, however a realistic signal could have 1024 samples which requires over 20,000,000 complex multiplications and additions. As you can see the number of calculations required soon mounts up to unmanageable proportions.

The Fast Fourier Transform is a simply a method of laying out the computation, which is much faster for large values of N, where N is the number of samples in the sequence. It is an ingenious way of achieving rather than the DFT's clumsy P^2 timing.

The idea behind the FFT is the *divide and conquer* approach, to break up the original N point sample into two (N / 2) sequences. This is because a series of smaller problems is easier to solve than one large one. The DFT requires (N-1)^2 complex multiplications and N(N-1) complex additions as opposed to the FFT's approach of breaking it down into a series of 2 point samples which only require 1 multiplication and 2 additions and the recombination of the points which is minimal.

For example *Seismic Data* contains hundreds of thousands of samples and would take months to evaluate the DFT. Therefore we use the FFT.

FFT Algorithm

The FFT has a fairly easy algorithm to implement, and it is shown step by step in the list below. Thjis version of the FFT is the Decimation in Time Method

1. Pad input sequence, of N samples, with ZERO's until the number of samples is the nearest power of two.

   e.g. 500 samples are padded to 512 (2^9)

2. Bit reverse the input sequence.

   e.g. 3 = 011 goes to 110 = 6

3. Compute (N / 2) *two* sample DFT's from the shuffled inputs.

   *See "Shuffled Inputs"*

4. Compute (N / 4) *four* sample DFT's from the two sample DFT's.

   *See "Shuffled Inputs"*

5. Compute (N / 2) *eight* sample DFT's from the four sample DFT's.

   *See "Shuffled Inputs"*

6. Until the all the samples combine into one N-sample DFT

**Shuffled Inputs**

The process of decimating the signal in the time domain has caused the INPUT samples to be re-ordered. For an 8 point signal the original order of the samples is

*0, 1, 2, 3, 4, 5, 6, 7*

But after decimation the order is

*0, 4, 2, 6, 1, 5, 3, 7*

At first it may look as if there is no order to this new sequence, BUT if the numbers are represented as binary a patter soon becomes apparent.

| ORIGINAL INPUT | | RE-ORDERED INPUT | |
|---|---|---|---|
| Decimal | Binary | Binary | Decimal |
| 0 | 000 | 000 | 0 |
| 1 | 001 | 100 | 4 |
| 2 | 010 | 010 | 2 |
| 3 | 011 | 110 | 6 |
| 4 | 100 | 001 | 1 |
| 5 | 101 | 101 | 5 |
| 6 | 110 | 011 | 3 |
| 7 | 111 | 111 | 7 |

What has happened is that the bit patterns representing the sample number has been reversed. This new sequence is the order that the samples enter the FFT.

## ALGORITHM TO IMPLEMENT FFT:

- **Step 1 -** Select no. of points for FFT(Eg: 64)
- **Step 2 –** Generate a sine wave of frequency ‘f ‘ (eg: 10 Hz with a sampling rate = No. of Points of FFT(eg. 64)) using **math library function**.
- **Step 3 -** Take sampled data and apply FFT algorithm .
- **Step 4 –** Use Graph option to view the Input & Output.
- **Step 5 -** Repeat **Step-1 to 4** for different no. of points & frequencies.

**C PROGRAM TO IMPLEMENT FFT :**

## Main.c (fft 256.c):

```
#include <math.h>
#define PTS 64                          //# of points for FFT
#define PI 3.14159265358979

typedef struct {float real,imag;} COMPLEX;

void FFT(COMPLEX *Y, int n);            //FFT prototype
float iobuffer[PTS];                    //as input and output buffer
float x1[PTS];                          //intermediate buffer
short i;                                //general purpose index variable
short buffercount = 0;                  //number of new samples in
iobuffer
short flag = 0;                         //set to 1 by ISR when iobuffer
full
COMPLEX w[PTS];                         //twiddle constants stored in w
COMPLEX samples[PTS];                   //primary working buffer

main()
{
   for (i = 0 ; i<PTS ; i++)            // set up twiddle constants in w
   {
   w[i].real = cos(2*PI*i/(PTS*2.0)); //Re component of twiddle
constants
   w[i].imag =-sin(2*PI*i/(PTS*2.0)); //Im component of twiddle
constants
   }


   for (i = 0 ; i < PTS ; i++)     //swap buffers
    {
     iobuffer[i] = sin(2*PI*10*i/64.0);/*10- > freq,
                                       64 -> sampling freq*/
     samples[i].real=0.0;
     samples[i].imag=0.0;
    }
```

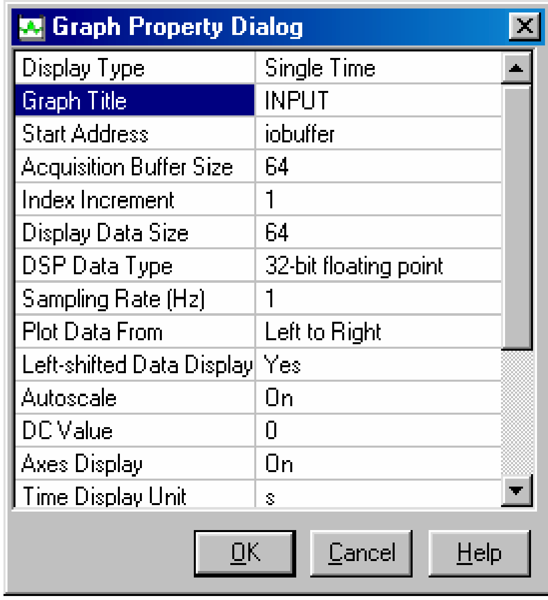
```
 for (i = 0 ; i < PTS ; i++)    //swap buffers
    {
     samples[i].real=iobuffer[i]; //buffer with new data
    }
   for (i = 0 ; i < PTS ; i++)
     samples[i].imag = 0.0;       //imag components = 0

FFT(samples,PTS);              //call function FFT.c

    for (i = 0 ; i < PTS ; i++)    //compute magnitude
    {
     x1[i] = sqrt(samples[i].real*samples[i].real
          + samples[i].imag*samples[i].imag);
    }

}                               //end of main
```

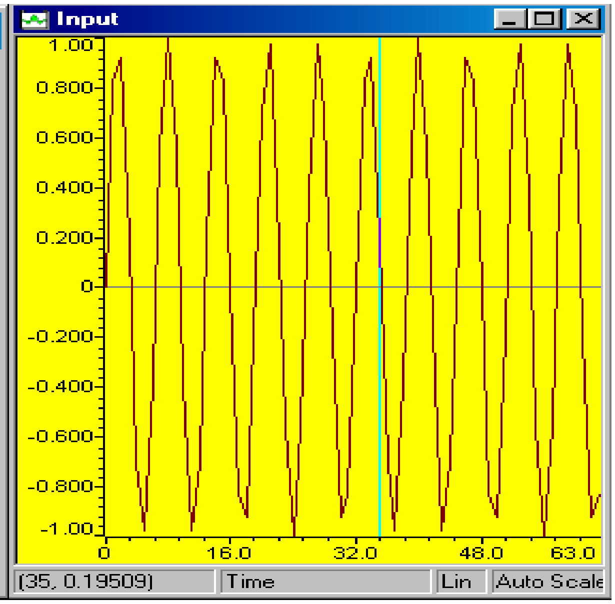
## fft.c:

```
#define PTS 64                  //# of points for FFT
typedef struct {float real,imag;} COMPLEX;
extern COMPLEX w[PTS];            //twiddle constants stored in w

void FFT(COMPLEX *Y, int N)   //input sample array, # of points
{
 COMPLEX temp1,temp2;         //temporary storage variables
 int i,j,k;                   //loop counter variables
 int upper_leg, lower_leg;    //index of upper/lower butterfly leg
 int leg_diff;                //difference between upper/lower leg
 int num_stages = 0;          //number of FFT stages (iterations)
 int index, step;             //index/step through twiddle
constant
 i = 1;                   //log(base2) of N points= # of stages

 do
  {
   num_stages +=1;
   i = i*2;
  }while (i!=N);
 leg_diff = N/2;          //difference between upper&lower legs
 step = (PTS*2)/N;       //step between values in twiddle.h
 for (i = 0;i < num_stages; i++)  //for N-point FFT
  {
   index = 0;
   for (j = 0; j < leg_diff; j++)
    {
```

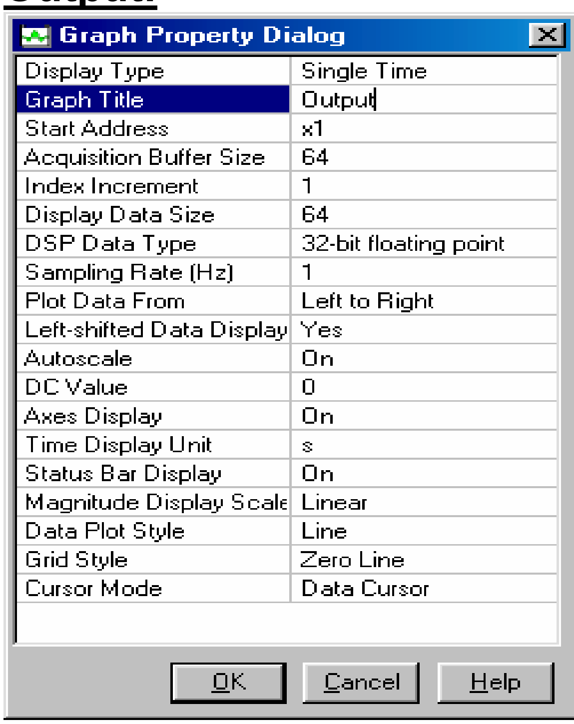
```
  for (upper_leg = j; upper_leg < N; upper_leg += (2*leg_diff))
   {
       lower_leg = upper_leg+leg_diff;
      temp1.real = (Y[upper_leg]).real + (Y[lower_leg]).real;
       temp1.imag = (Y[upper_leg]).imag + (Y[lower_leg]).imag;
       temp2.real = (Y[upper_leg]).real - (Y[lower_leg]).real;
       temp2.imag = (Y[upper_leg]).imag - (Y[lower_leg]).imag;
       (Y[lower_leg]).real = temp2.real*(w[index]).real
                        -temp2.imag*(w[index]).imag;
       (Y[lower_leg]).imag = temp2.real*(w[index]).imag
                        +temp2.imag*(w[index]).real;
       (Y[upper_leg]).real = temp1.real;
       (Y[upper_leg]).imag = temp1.imag;
   }
   index += step;
    }
     leg_diff = leg_diff/2;
     step *= 2;
      }
       j = 0;
       for (i = 1; i < (N-1); i++)     //bit reversal for
resequencing data
      {
     k = N/2;
     while (k <= j)
      {
       j = j - k;
       k = k/2;
      }
     j = j + k;


 if (i<j)
      {
       temp1.real = (Y[j]).real;
       temp1.imag = (Y[j]).imag;
       (Y[j]).real = (Y[i]).real;
       (Y[j]).imag = (Y[i]).imag;
       (Y[i]).real = temp1.real;
       (Y[i]).imag = temp1.imag;
      }
    }
   return;
}
```

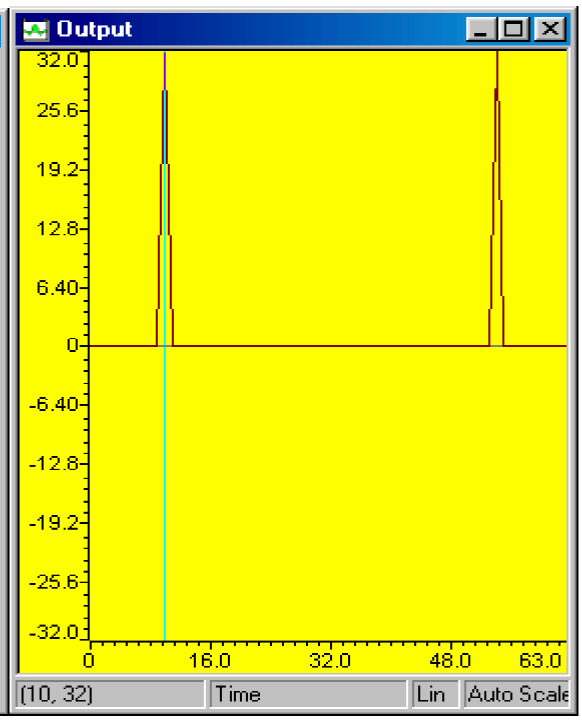
# Input:




# Output:



## HOW TO PROCEED

- Open Code Composer Studio, make sure the DSP kit is turned on.
- Start a new project using 'Project-new ' pull down menu, save it in a separate directory(c:\ti\myprojects) with name "**FFT.pjt".**
- Add the source files "**FFT256.c**" and **"FFT.C"** in the project using 'Project→add files to project' pull down menu.
- Add the linker command file "**hello.cmd"**
- Add the rts file **"rts6700.lib"**
- Compile the program using the 'Project-compile' pull down menu or by clicking the shortcut icon on the left side of program window.
- Load the program in program memory of DSP chip using the 'File-load program' pull down menu.
- Run the program and observe output using graph utility.