LABORATORY MANUAL

DEPARTMENT OF
ELECTRONICS&COMMUNICATION ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY-WARANGAL

Digital Signal Processing Lab

Introduction to TMS320C6748 DSP Processor

Digital Signal processing lab : List of experiments:

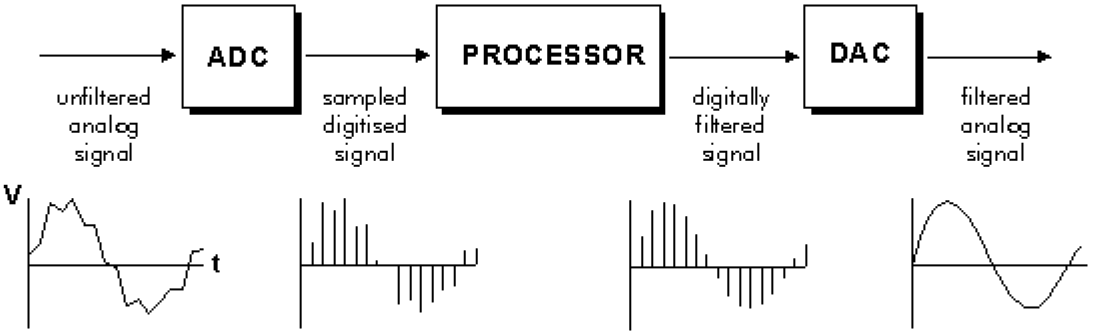
Introduction

Digital Signal Processors (DSP) are specific processors used for signal processing. Digital signal processing is used in many areas such as sound, video, computer vision, speech analysis and synthesis, etc. Each of these areas need digital signal processing and can therefore use these specific processors. DSPs are found in cellular phones, disk drives, radios, printers, MP3 players, HDTV, digital cameras and so on. DSPs take real world signal like voice, audio, video, temperature, pressure, position, etc. that have been digitized using an analog-to-digital converter and then manipulate them using mathematical operations. DSPs are usually optimized to process the signal very quickly and many instructions are built such that they require a minimum amount of CPU clock cycles.

Typical optimization of the processors include hardware multiply accumulate (MAC) capability, hardware circular and bit-reversed capabilities, for efficient implementation of data buffers and fast Fourier transforms (FFT), and Harvard architecture .One specificity of DSPs are their MAC operation (Multiply accumulate). Once a signal has been digitized, it is available to the processor as a serie of numerical values. Digital signal processing often involves the convolution operation, which correspond to a serie of sums and products. On a general CPU, each of these instruction takes more than one clock cycle. However, DSP’s MAC operation performs a product and a sum in one clock cycle, allowing thus faster signal processing. Another specificity of DSPs is their ability to access more than one memory address in one cycle. This allows the processor to read an instruction and the corresponding data simultaneously. This is an improvement over general processors. DSPs are typically very similar to microcontrollers. They usually provide single chip computer solutions integrating on-board volatile and non-volatile memory as well as a range of peripheral interfaces. They moreover have a small footprint, making them ideal for embedded applications. In addition, they tend to have low power consumption requirements, which makes them suitable for portable applications, for example in cell phones, or in a car.

DSP is composed of a processor and analog interfaces, as showed in the Figure1.1. Instrument C6748 floating point processor and a TLV320AIC3106 (AIC3106) codec. The codec allows to convert the analog signal to a digital signal (ADC: Analog to Digital Converter) and to convert a digital signal back to an analog one (DAC: Digital to

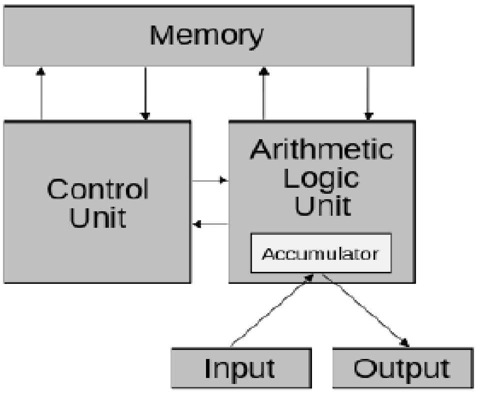
Analog Converter). The codec communicates with the DSP via a multichannel serial port (McASP) and I2C interfaces. The latter are available to a user on the board, thus allowing the user to connect a different codec rather than the default one.



General DSP system

Processors architecture

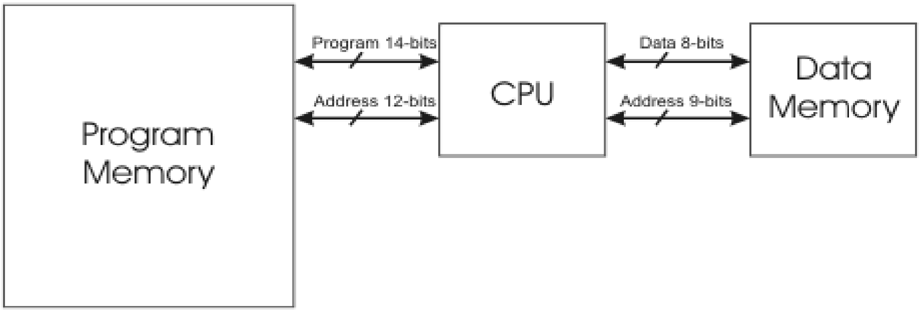
There are two types of architectures used in processors. Von Neuman architecture, and Harvard architecture. In the first one, programs and data are stored in the same memory zone, as showed in Figure 1.2a. In that kind of architecture, an instruction contains the operating code (opcode) and the addresses of the operands.



Von Neumann architecture

In the second type of architecture, there are two memories: one for the program, and one for the data. Both memories are separated and are accessed via separated buses, as shown in Figure 1.2b. That organization allows to read an instruction and the data simultaneously, which improves the overall performance of the processor.

General microprocessors usually make use of the Von Neuman architecture. However, DSPs tend to use the Harvard architecture which is more suited for real-time applications. Some DSPs use a modified Harvard architecture, in which there is one external bus for the data, and one external bus for the addresses, like the Von Neuman architecture. However, the DSP contains two internal distinct data buses and two distinct addresses buses.



Harvard architecture

transfer between the external and internal buses are performed using time multiplexing. This is the case in some Texas Instrument processors (C6000 serie), in particular, this is the case of the C6748 processor used in this course.

TMS320 DSP Family Overview

The TMS320 DSP family consists of fixed-point, floating-point, and multiprocessor digital signal processors (DSPs). TMS320DSPs have an architecture designed specifically for real-time signal processing.

TMS320C6000 DSP Family Overview

With a performance of up to 6000 million instructions per second (MIPS) and an efficient C compiler, the TMS320C6000 DSPs give system architects unlimited possibilities to differentiate their products. High performance, ease of use, and affordable pricing make the C6000 generation the ideal solution for multichannel, multifunction applications, such as:

- Pooled modems
- Wireless local loop base stations
- Remote access servers (RAS)
- Digital subscriber loop (DSL) systems
- Cable modems
- Multichannel telephony systems

The C6000 generation is also an ideal solution for exciting new applications; for example:

- Personalized home security with face and hand/fingerprint recognition
- Advanced cruise control with global positioning systems (GPS) navigation and accident avoidance

- Remote medical diagnostics
- Beam-forming base stations
- Virtual reality 3-D graphics
- Speech recognition
- Audio
- Radar
- Atmospheric modeling
- Finite element analysis
- Imaging (examples: fingerprint recognition, ultrasound, and MRI)

Hardware installation:

Most of the examples presented in this book involve the development and testing of short programs intended to demonstrate fundamental DSP concepts in a laboratory setting. To perform the experiments described in the book, a number of hardware and software tools are required

- **A TMS320C6748 CLDK kit package includes the following:** LCDK development board, A universal serial bus (USB) cable that connects the C6748 board to a host PC.,A 5 V universal power supply for the C6748 board. (Power brick and power cord), Micro SD card with SD-adapter, Quick start guide
- A host PC This is used to run the Code Composer Studio IDE. The C6748 LCDK board is connected to a USB port on the host PC.
- Code Composer Studio software, version 6 The LCDK kit includes a DVD containing Code Composer Studio software version 6 . Alternatively, the Code Composer Studio IDE may be downloaded from the Texas Instruments wiki at http://processors.wiki.ti.com/index. Php/Download_ CCS. Code Composer Studio software provides an IDE, bringing together Compiler, assembler, linker, and debugger

TMS320C678 LCD Kit Board

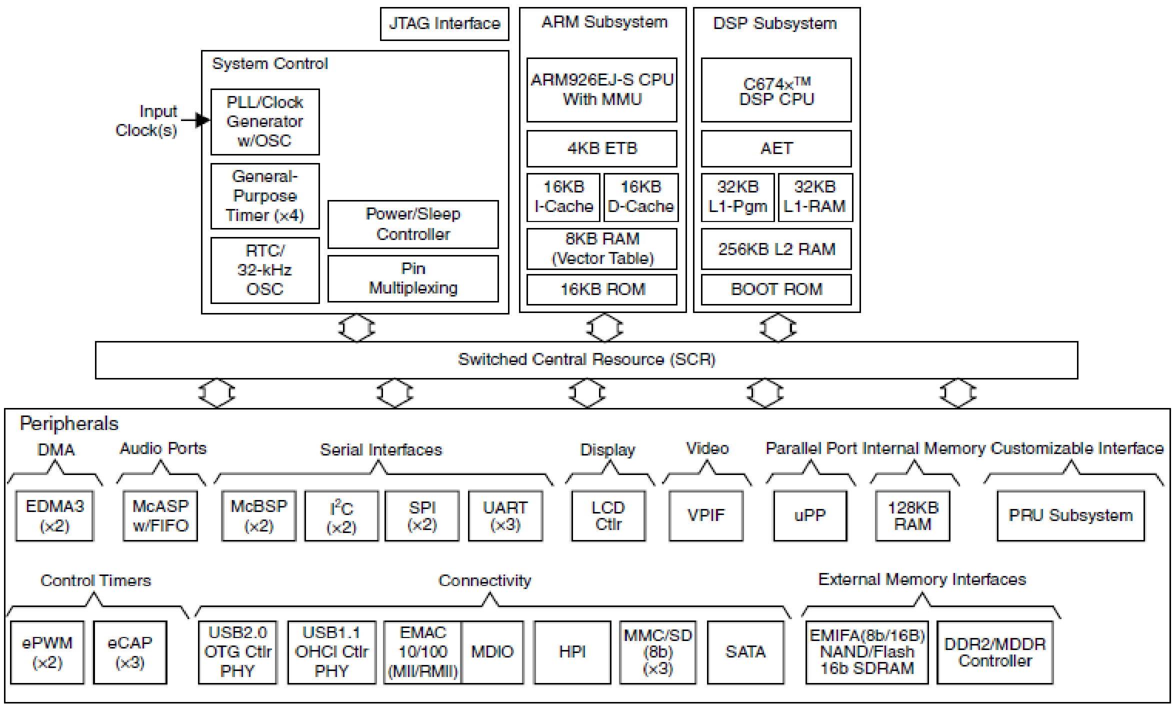
The LCDK board is a powerful, yet inexpensive, development system with the necessary hardware and software support tools for real-time signal processing. From the point of view of the example programs in this book, it is a complete DSP system. The board, which measures approximately 5_7 inches, includes a 375 MHz C6748 processor and a 16-bit stereo codec TLV320AIC3106 (AIC3106) for analog input and output.

Numerous other interfaces are provided by the LCDK but are not used by the example programs in this book. The onboard codec AIC3106 uses sigma–delta technology that provides analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC) functions. It uses a 24.576MHz system clock and its sampling rate can be selected from a range of alternative settings from 8 to 48 kHz. The LCDK boards each include 128MB of synchronous dynamic RAM (mDDR SDRAM) and 8MB of NOR flash memory. Two 3.5mm jack socket connectors on the boards provide analog input and output: LINE IN for line level input and LINE OUTfor line level output. The status of eight user DIP switches on the board can be read from within a program running on the processor and provide a simple means of user interaction. The states of two LEDs on the board can be controlled from within a program running on the processor.

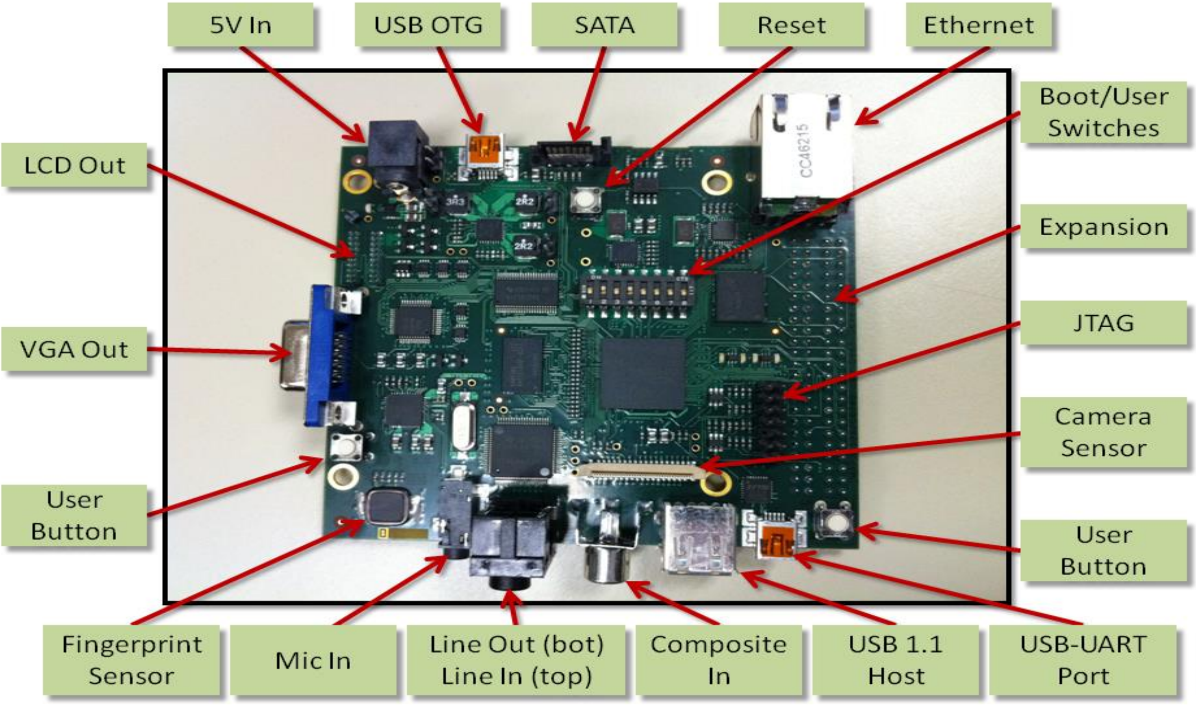
Top and bottom View of TM320C6748 DSP Development Kit(LCDK)

C6748 Processor

The DSP core in the OMAP-L138 device (L138) is a C6748 DSP based on Texas Instruments very long instruction word (VLIW) architecture and is very well suited to numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. With a clock rate of 375 MHz, the C6748 is capable of fetching eight 32-bit instructions every 1/(375MHz) or 2.67 ns. As part of the C674x family, it incorporates both floating-point and fixed-point architectures in one core. Features of the C6748 include 326 kB of internal memory (32 kB of L1P program RAM/cache, 32 kB of L1D data RAM/cache, and 256 kB of L2 RAM/cache), eight functional or execution units composed of six ALUs and two multiplier units, an external memory interface addressing 256MB of 16-bit mDDR SDRAM, and 64 32- bit general-purpose registers. In addition, the OMAP-L138 features 128 kBof on-chip RAM shared by its C6748 and ARM9 processor cores .



Functional block diagram of OMAP-L138 processor



Features:

- **Processor:**
  - TI TMS320C6748 DSP or OMAP-L138 Application Processor
  - 456-MHz C674x Fixed/Floating Point DSP
  - 456-MHz ARM926EJ RISC CPU (OMAP-L138 only)
  - On-Chip RTC
- **Memory**
  - 128 MByte DDR2 SDRAM running at 150MHz
  - 128 MByte 16-bit wide NAND FLASH
  - 1 Micro SD/MMC Slot
- **Interfaces**
  - One mini-USB Serial Port (on-board serial to USB)
  - One Fast Ethernet Port (10/100 Mbps) with status LEDs
  - One USB Host port (USB 1.1)
  - One mini-USB OTG port (USB 2.0)
  - One SATA Port (3Gbps)
  - One VGA Port (15 pin D-SUB)
  - One LCD Port (Beagleboard XM connectors)
  - One Composite Video Input (RCA Jack)
  - One Leopard Imaging Camera Sensor Input (36-pin ZIP connector)
  - Three AUDIO Ports (1 LINE IN & 1 LINE OUT & 1 MIC IN)
  - 14-pin JTAG header (No onboard emulator; external emulator is required)

Introduction to code composer studio

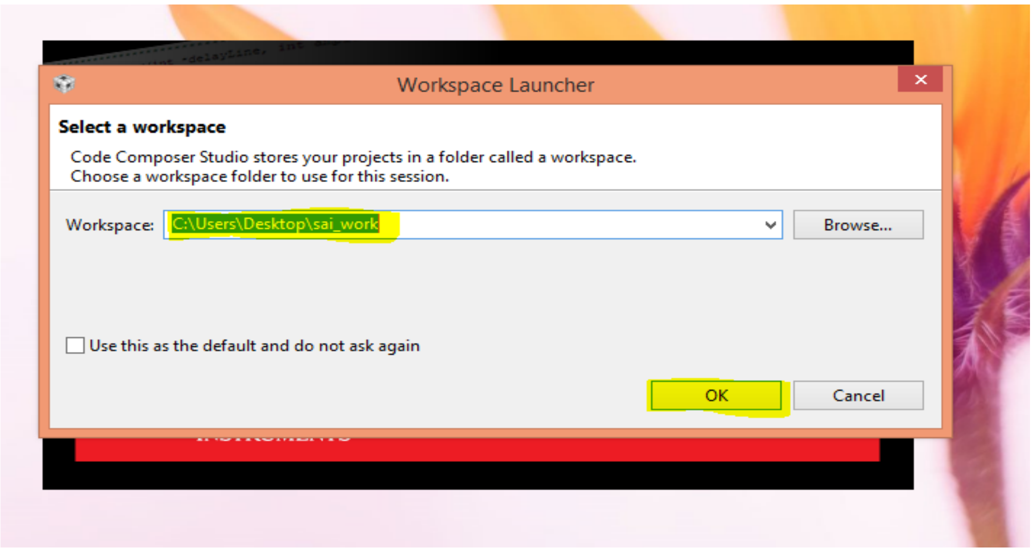
Code Composer is the DSP industry's first fully integrated development environment (IDE) with SDP-specific functionality. With a familiar environment likes MS-based C++ TM, Code Composer lets you edit, build, debug, profile and manage projects from a single unified environment. Other unique features include graphical single analysis, injection/execution of data signals via a C-interpretive scripting language and much more.

It is based on the Eclipse framework and therefore requires a Java Runtime Environment (JRE). This chapter details the setup of CCS in order to be able to program the EVM board OMAP-L138 and to run programs on the DSP. Some specific files are needed to communicate with the DSP and to program it. These files are stored in the folder C:\DSP in your workstation. This folder contains configuration files and libraries useful to program the EVM board.

- ## Procedure to work on Code Composer Studio:

1. To create **New project:**

To start Code Composer Studio(ignore this if CCS is already running), double click the Code Composer Studio 6.0.1 icon on your desktop. And Give the workspace location as shown in figure.



Workspace Launcher

Select a workspace

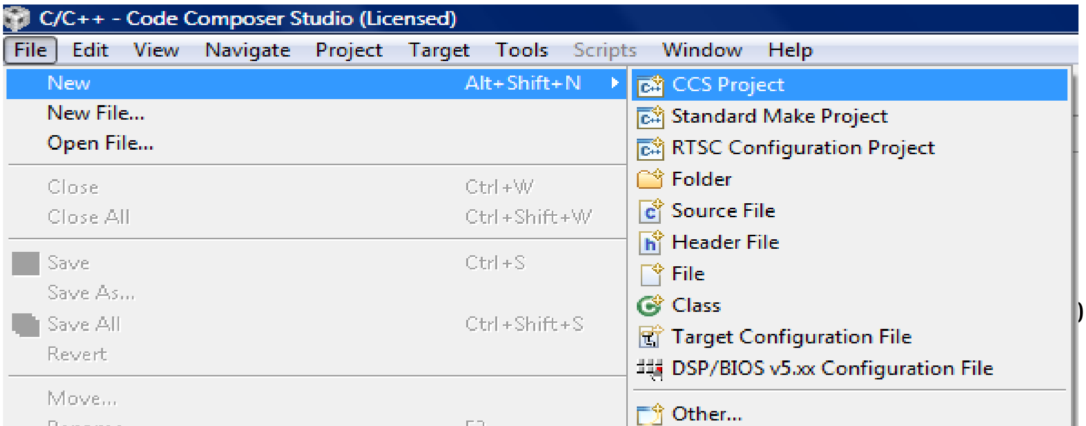
Code Composer Studio stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\Users\Desktop\sai_work

□ Use this as the default and do not ask again

Workspace Launcher

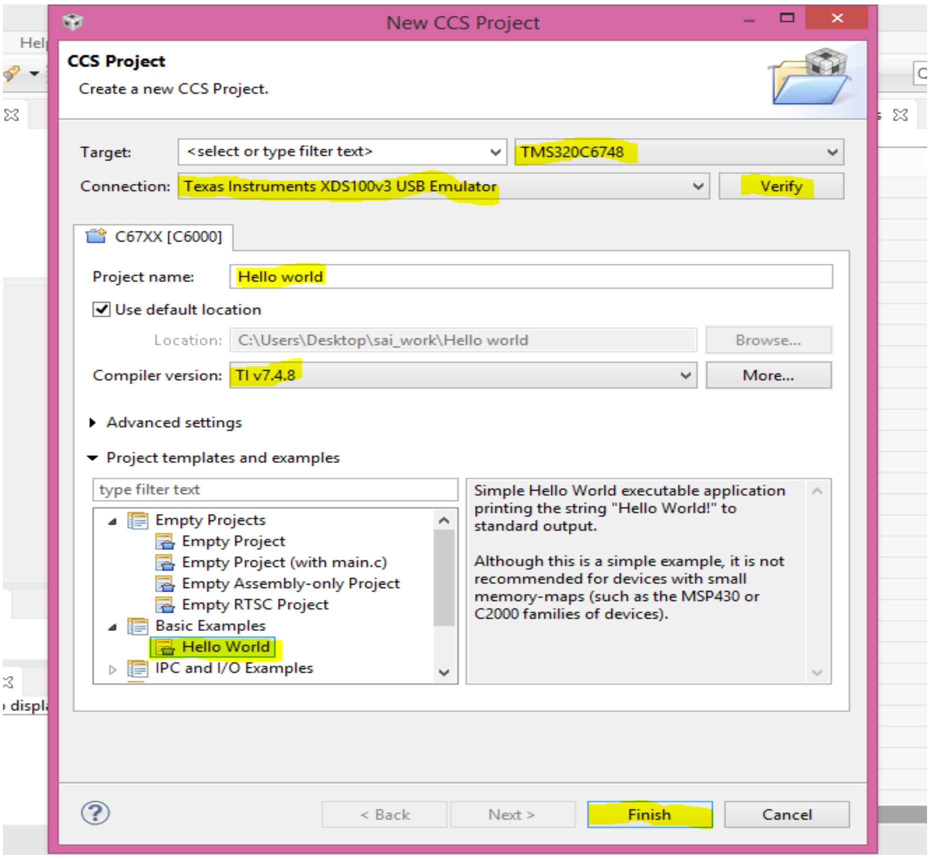
To create a CCS project select **File→New→CCS Project→**



C/C++ - Code Composer Studio (Licensed)

This will bring up a window where you can enter the name of the project. The location selected is the default location for project files. Since the example uses the TMS320C6748 processor the project type selected is C6000. The project configurations are Debug and Release. Select the finish button. On the Project Settings Screen, select the items that pertain to the type of project to be created. Since the project will be executed select Output Type: Executable.

The processor type is TMS320C67xx so the Device Variant is Generic C67xx Device. This project will use Little Endian. The code generation tools are the latest version (in this case TI v7.0.3). The runtime support library is selected to match the device variant and the endianness selected. The library can be selected automatically. Press Next. Since this project will have C programs only, select the Hello world Project. Select Finish and the project will show up in the C/C++ Projects view.



New CCS Project

CCS Project

Create a new CCS Project.

Target: <select or type filter text> TMS320C6748

Connection: Texas Instruments XDS100v3 USB Emulator

C67XX [C6000]

Project name: Hello world

☑ Use default location

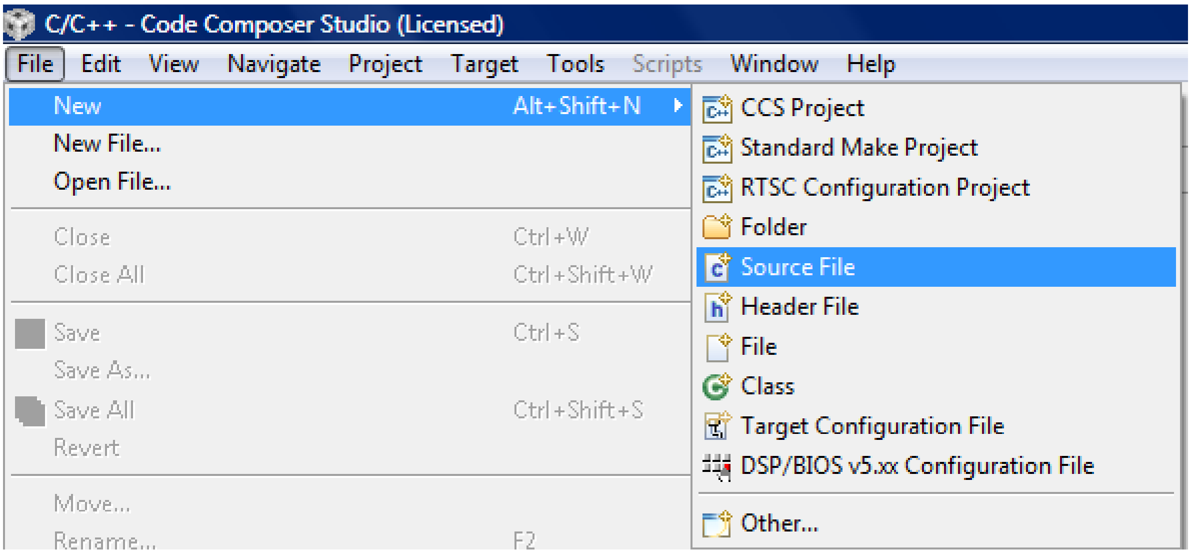
Location: C:\Users\Desktop\sai_work\Hello world

Compiler version: TI v7.4.8

Simple Hello World executable application printing the string "Hello World!" to standard output.

Although this is a simple example, it is not recommended for devices with small memory-maps (such as the MSP430 or C2000 families of devices).

Create the main.c file that contains the C to execute. Select **File→New→Source File**. The New Source File dialog opens and you can enter the source file name. Add a .c(.asm for assembly program) extension for c program file. The source folder should be the folder of your project. Select **Finish** and a new file is opened.

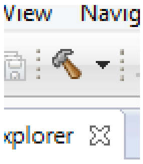


Enter the c code shown below in main.c.

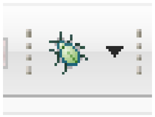
```
#include <stdio.h>
int main(void) {
        printf("Hello World");
        return 0;
```

- ## Build and Debug the project

To start a debug session first build the project by using build icon and select Target→Debug Active Project or press the debug icon in the menu bar.

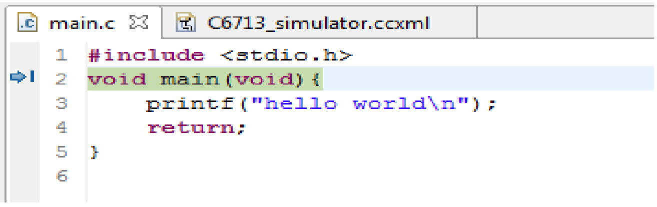


Build icon



Debug icon

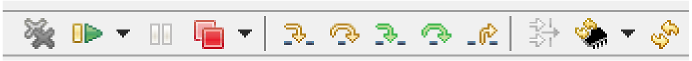
When the debug session starts the code _les are compiled and linked into an executable file and the file is loaded onto the target. The initialization code is executed and the program is halted at the beginning of the main function.



```
#include <stdio.h>
void main(void){
    printf("hello world\n");
    return;
}
```

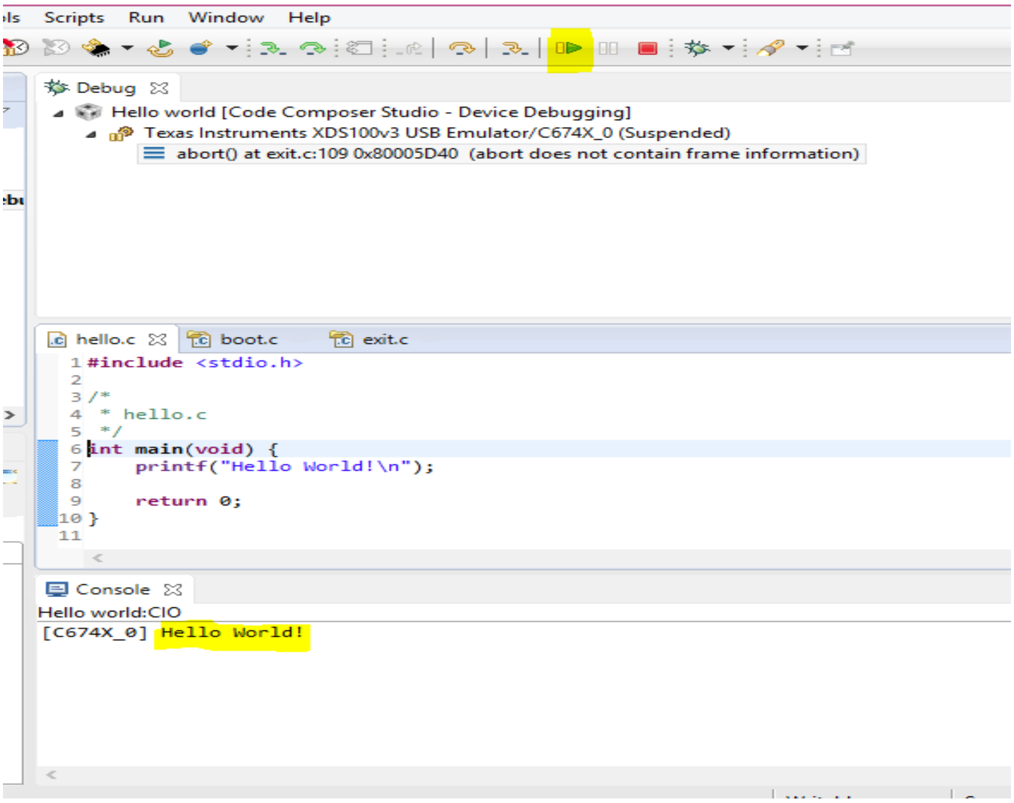
Debugging session

Icons on the debug window can be used to run the program and other debugging tools.



Debugging tools

Press the **green arrow** to run the program. The main function will be executed and the program will leave the main function. In the program the printf function prints to the console which if not shown by default can be viewed by selecting **View→Console**. In the console is shown the "hello world" that was printed.



```
Hello world [Code Composer Studio - Device Debugging]
  Texas Instruments XDS100v3 USB Emulator/C674X_0 (Suspended)
    abort() at exit.c:109 0x80005D40  (abort does not contain frame information)
```

```
#include <stdio.h>

/*
 * hello.c
 */
int main(void) {
    printf("Hello World!\n");

    return 0;
}
```

```
Hello world:CIO
[C674X_0] Hello World!
```

Console output of the program

Experiment-1
Basic Arithmetic operations

Aim:
To perform basic arithmetic operations on DSP processor using CCS

Components and Equipment:

TMS320c6748 Processor kit, PC with CCS

32 BIT addition:

A 32-bit register can store 2^{32} different values. The range of integer values that can be stored in 32 bits depends on the integer representation used. With the two most common representations, the range is 0 through 4,294,967,295 (2^{32} - 1) for representation as an (unsigned) binary number, and -2,147,483,648 (-2^{31}) through 2,147,483,647 (2^{31} - 1) for representation as Two's complement.

C program for Addition:

```
#include <stdio.h>
#include <math.h>
int main(void) {
        long a=0x12345678;
        long b=0x87654321;
        long c=a+b;
return 0;}
```

Assembly code generated by compiler:

```
80000ba0:   07FFF052        ADDK.S2      -32,SP
80000ba4:   022B3C28        MVK.S1       0x5678,A4
80000ba8:   02091A69        MVKH.S1      0x12340000,A4
80000bac:   0280A358 ||     MVK.L1       0,A5
80000bb0:   023C23C4        STDW.D2T1    A5:A4,*+SP[1]
80000bb4:   022190AA        MVK.S2       0x4321,B4
80000bb8:   0243B2EB        MVKH.S2      0x87650000,B4
80000bbc:   0280A35A ||     MVK.L2       0,B5
80000bc0:   023C43C6        STDW.D2T2    B5:B4,*+SP[2]
80000bc4:   033C23E6        LDDW.D2T2    *+SP[1],B7:B6
80000bc8:   0210C53A        ADDU.L2      B6,B5:B4,B5:B4
80000bcc:   0294E07A        ADD.L2       B7,B5,B5
80000bd0:   023C63C6        STDW.D2T2    B5:B4,*+SP[3]
```

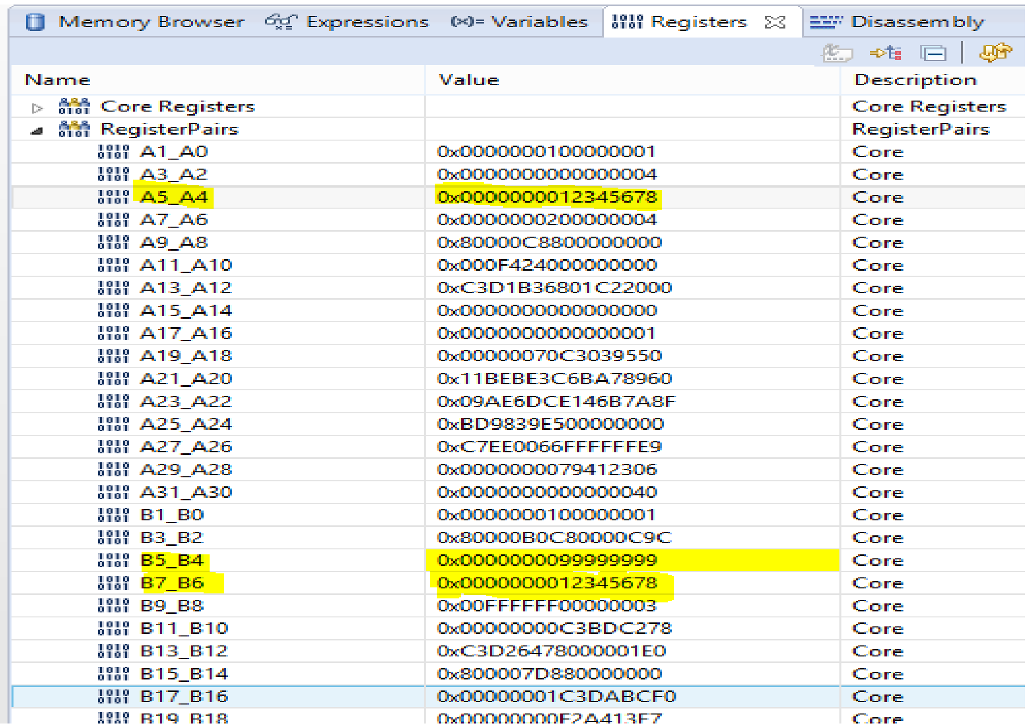
Execution Procedure:

1.Register A4 =1234 5678 h   2.Register pair A5:A4 = 0000 0000 1234 5678 h

3.Register B4 =8765 4321 h   4. Register pair B5:B4 = 0000 0000 8765 4321 h

5.Addition Operation: Register B5:B4 = 0000 0000 9999 9999

Go to view & select CPU registers to view the changes happening in CPU registers.



Name	Value	Description
Core Registers		Core Registers
RegisterPairs		RegisterPairs
A1_A0	0x0000000100000001	Core
A3_A2	0x0000000000000004	Core
A5_A4	0x0000000012345678	Core
A7_A6	0x0000000200000004	Core
A9_A8	0x80000C8800000000	Core
A11_A10	0x000F424000000000	Core
A13_A12	0xC3D1B36801C22000	Core
A15_A14	0x0000000000000000	Core
A17_A16	0x0000000000000001	Core
A19_A18	0x00000070C3039550	Core
A21_A20	0x11BEBE3C6BA78960	Core
A23_A22	0x09AE6DCE146B7A8F	Core
A25_A24	0xBD9839E500000000	Core
A27_A26	0xC7EE0066FFFFFFE9	Core
A29_A28	0x0000000079412306	Core
A31_A30	0x0000000000000040	Core
B1_B0	0x0000000100000001	Core
B3_B2	0x80000B0C80000C9C	Core
B5_B4	0x0000000099999999	Core
B7_B6	0x0000000012345678	Core
B9_B8	0x00FFFFFF00000003	Core
B11_B10	0x00000000C3BDC278	Core
B13_B12	0xC3D26478000001E0	Core
B15_B14	0x800007D880000000	Core
B17_B16	0x00000001C3DABCF0	Core
B19_B18	0x00000000F2A413F7	Core

Registers View

32 BIT multiplication

C program for Multiplication:

```
#include <stdio.h>
#include <math.h>
int main(void) {
            long a=0x12345678;//Hex
            long b=0x87654321;//Hex
            long c=a*b;//Multiplication
      return 0;}
```

Assembly code generated by compiler:

```
80000ba0:   07FFF052          ADDK.S2       -32,SP
80000ba4:   022B3C28          MVK.S1        0x5678,A4
80000ba8:   02091A69          MVKH.S1       0x12340000,A4
80000bac:   0280A358 ||       MVK.L1        0,A5
80000bb0:   023C23C4          STDW.D2T1     A5:A4,*+SP[1]
80000bb4:   022190AA          MVK.S2        0x4321,B4
80000bb8:   0243B2EB          MVKH.S2       0x87650000,B4
80000bbc:   0280A35A ||       MVK.L2        0,B5
80000bc0:   023C43C6          STDW.D2T2     B5:B4,*+SP[2]
80000bc4:   043C43E6          LDDW.D2T2     *+SP[2],B9:B8
80000bc8:   033C23E6          LDDW.D2T2     *+SP[1],B7:B6
80000bcc:   00006000          NOP           4
80000bd0:   0820EB02          MPY32SU.M2    B7,B8,B17:B16
80000bd4:   02190632          MPY32U.M2     B8,B6,B5:B4
80000bd8:   03192B02          MPY32SU.M2    B9,B6,B7:B6
80000bdc:   00004000          NOP           3
80000be0:   0340C07A          ADD.L2        B6,B16,B6
80000be4:   0298A07A          ADD.L2        B5,B6,B5
80000be8:   023C63C6          STDW.D2T2     B5:B4,*+SP[3]
```

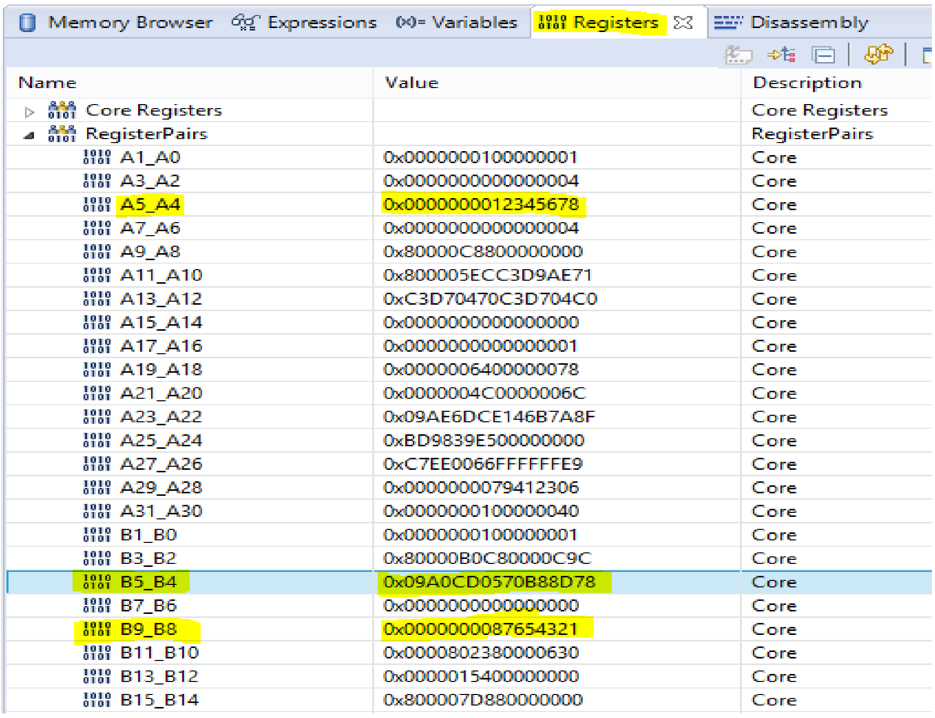
Execution Procedure:

1.Register A4 = 1234 5678 h   2.Register pair A5:A4 = 0000 0000 1234 5678 h

3.Register B4 = 8765 4321 h   4. Register pair B5:B4 = 0000 0000 8765 4321 h

5.Multiplication Operation:

Register B5:B4 = 09A0 CD05 70B8 8D78



Name	Value	Description
Core Registers		Core Registers
RegisterPairs		RegisterPairs
A1_A0	0x0000000100000001	Core
A3_A2	0x0000000000000004	Core
A5_A4	0x0000000012345678	Core
A7_A6	0x0000000000000004	Core
A9_A8	0x80000C8800000000	Core
A11_A10	0x800005ECC3D9AE71	Core
A13_A12	0xC3D70470C3D704C0	Core
A15_A14	0x0000000000000000	Core
A17_A16	0x0000000000000001	Core
A19_A18	0x0000006400000078	Core
A21_A20	0x0000004C0000006C	Core
A23_A22	0x09AE6DCE146B7A8F	Core
A25_A24	0xBD9839E500000000	Core
A27_A26	0xC7EE0066FFFFFFE9	Core
A29_A28	0x0000000079412306	Core
A31_A30	0x0000000100000040	Core
B1_B0	0x0000000100000001	Core
B3_B2	0x80000B0C80000C9C	Core
B5_B4	0x09A0CD0570B88D78	Core
B7_B6	0x0000000000000000	Core
B9_B8	0x0000000087654321	Core
B11_B10	0x0000802380000630	Core
B13_B12	0x0000015400000000	Core
B15_B14	0x800007D880000000	Core

Registers View

32 BIT floating point addition:

C program for floating point addition:

```
#include <stdio.h>
#include <math.h>
int main(void) {
        float a=123.321;
        float b=332.125;
        float c=a+b;
        return 0; }
```

Assembly code generated by compiler:

```
80000be0:   07BE09C2        SUB.D2      SP,0x10,SP
80000be4:   01D22D28        MVK.S1      0xffffa45a,A3
80000be8:   01A17B68        MVKH.S1     0x42f60000,A3
80000bec:   01BC22F4        STW.D2T1    A3,*+SP[1]
80000bf0:   0208002A        MVK.S2      0x1000,B4
80000bf4:   0221D36A        MVKH.S2     0x43a60000,B4
80000bf8:   DC45            STW.D2T2    B4,*B15[2]
80000bfa:   DC5D            LDW.D2T2    *B15[2],B5
80000bfc:   E8100000        .fphead     p, l, W, BU, nobr, nosat, 1000000
80000c00:   020CB21A        ADDSP.L2X   B5,A3,B4
80000c04:   00004000        NOP         3
80000c08:   023C62F6        STW.D2T2    B4,*+SP[3]
```

Execution Procedure:

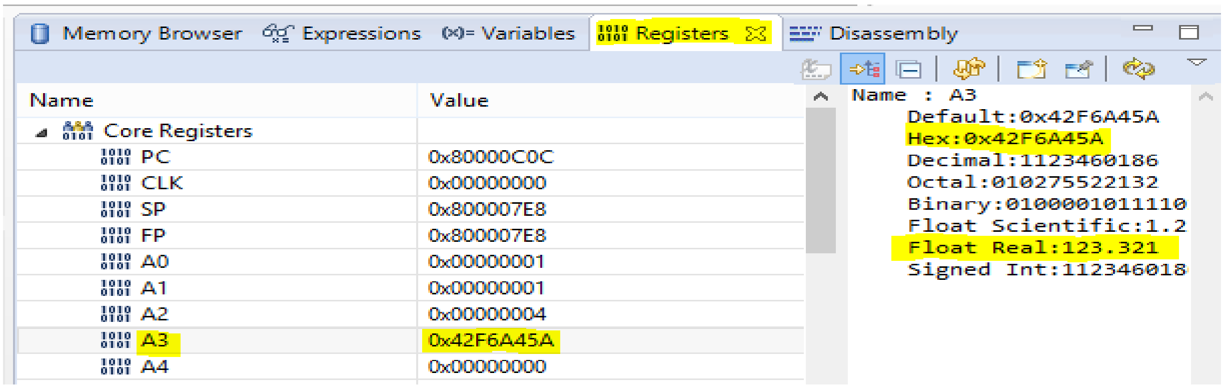
1.C Compiler is automatically convert the floating point value to hexadecimal value.

2.Register A3 =42F6 A45A h 2.Register pair A5:A4 = 0000 0000 42F6 A45A h

3.Register B4 =43A6 1000 h 4. Register pair B5:B4 = 0000 0000 43A6 1000 h

5.Addition Operation:

Register B5:B4 =0000 0000 43E3 B916



Name	Value
Core Registers	
PC	0x80000C0C
CLK	0x00000000
SP	0x800007E8
FP	0x800007E8
A0	0x00000001
A1	0x00000001
A2	0x00000004
A3	0x42F6A45A
A4	0x00000000

```
Name : A3
    Default:0x42F6A45A
    Hex:0x42F6A45A
    Decimal:1123460186
    Octal:010275522132
    Binary:0100001011110
    Float Scientific:1.2
    Float Real:123.321
    Signed Int:112346018
```

Register View

32 BIT floating point multiplication

C program for floating point Multiplication:

```
#include <stdio.h>
#include <math.h>
int main(void) {
        float a=123.321;
        float b=332.125;
        float c=a*b;
        return 0; }
```

Assembly code generated by compiler:

```
80000be0:   07BE09C2        SUB.D2      SP,0x10,SP
80000be4:   01D22D28        MVK.S1      0xffffa45a,A3
80000be8:   01A17B68        MVKH.S1     0x42f60000,A3
```

```
80000bec:   01BC22F4         STW.D2T1      A3,*+SP[1]
80000bf0:   0208002A         MVK.S2        0x1000,B4
80000bf4:   0221D36A         MVKH.S2       0x43a60000,B4
80000bf8:   DC45             STW.D2T2      B4,*B15[2]
80000bfa:   DC5D             LDW.D2T2      *B15[2],B5
80000bfc:   E8100000         .fphead       p, l, W, BU, nobr, nosat, 1000000
80000c00:   020CBE02         MPYSP.M2X     B5,A3,B4
80000c04:   00004000         NOP           3
80000c08:   023C62F6         STW.D2T2      B4,*+SP[3]
```

Execution Procedure:

1.C Compiler is automatically convert the floating point value to hexadecimal value.

2.Register A3 = 42F6 A45A h   2.Register pair A5:A4 = 0000 0000 42F6 A45A h

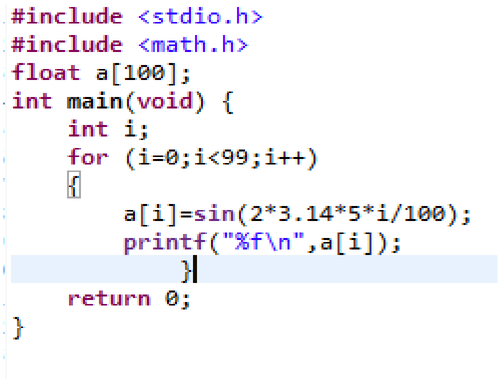
3.Register B4 = 43A6 1000 h   4. Register pair B5:B4 = 0000 0000 43A6 1000 h

5.Addition Operation:

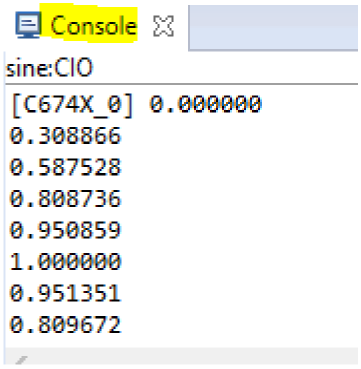
Register B5:B4 = 0000 0000 471F FDFD h

Generation of sine waveform:

C Program sine wave:



```
#include <stdio.h>
#include <math.h>
float a[100];
int main(void) {
    int i;
    for (i=0;i<99;i++)
    {
        a[i]=sin(2*3.14*5*i/100);
        printf("%f\n",a[i]);
            }
    return 0;
}
```



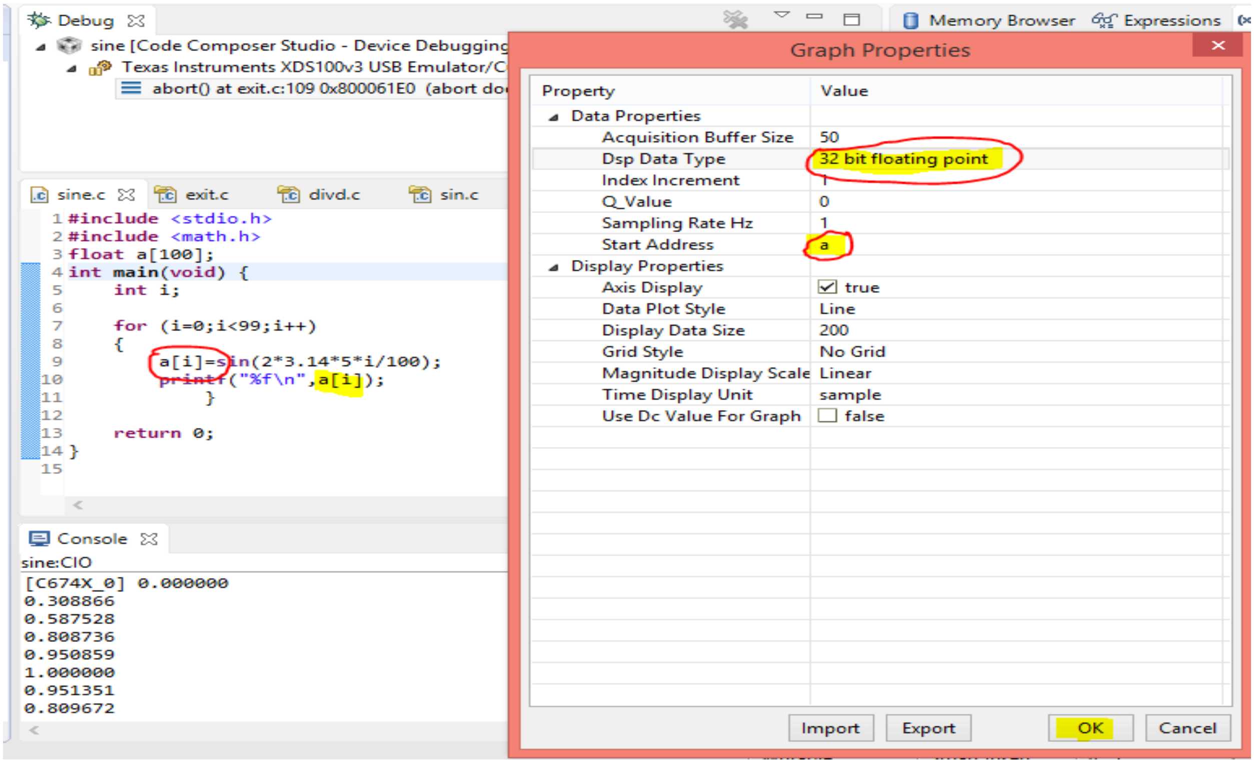
```
sine:CIO
[C674X_0] 0.000000
0.308866
0.587528
0.808736
0.950859
1.000000
0.951351
0.809672
```

Output values in Console

Graph Plotting:

Goto CSS tool bar **Tools -→ Graph -→ Single Time**

(In any graph the start address is the important parameter to be checked)



```
sine [Code Composer Studio - Device Debugging
  Texas Instruments XDS100v3 USB Emulator/C
    abort() at exit.c:109 0x800061E0  (abort do
```

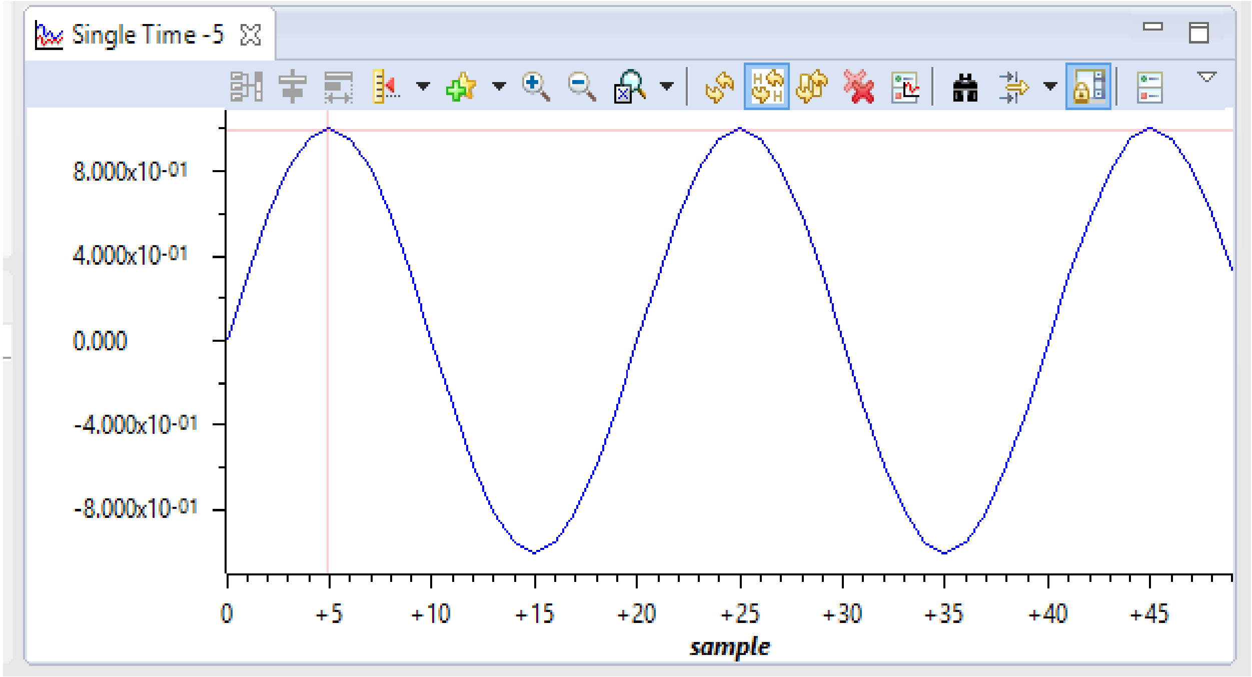
```c
#include <stdio.h>
#include <math.h>
float a[100];
int main(void) {
    int i;

    for (i=0;i<99;i++)
    {
        a[i]=sin(2*3.14*5*i/100);
        printf("%f\n",a[i]);
            }

    return 0;
}
```

```
sine:CIO
[C674X_0] 0.000000
0.308866
0.587528
0.808736
0.950859
1.000000
0.951351
0.809672
```

Property	Value
Data Properties	
Acquisition Buffer Size	50
Dsp Data Type	32 bit floating point
Index Increment	1
Q_Value	0
Sampling Rate Hz	1
Start Address	a
Display Properties	
Axis Display	☑ true
Data Plot Style	Line
Display Data Size	200
Grid Style	No Grid
Magnitude Display Scale	Linear
Time Display Unit	sample
Use Dc Value For Graph	□ false



Output wave form

Experiment-2

Aim:
To perform Linear and circular convolution on DSP processor using CCS

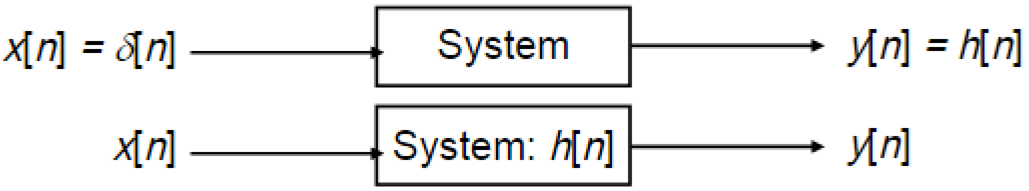
Components and Equipment:

TMS320c6748 Processor kit, PC with CCS

Theory:

Linear Convolution:

Convolution is an operator that takes an input signal and return an output signal, based on knowledge about the system's unit impulse response h[n].Linear convolution is the basic operation to calculate the output for any linear time invariant system given its input and its impulse response.



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:

$$y(n) = x(n) * h(n)$$

$$= \sum_{k=-\infty}^{\infty} x(k) h(n-k)$$

where

x[]=Input signal Samples

h[]=Impulse response coefficient.

y[]=Convolution output

n=No. of Input samples

h=No. of impulse response coefficient

C program for linear convolution:

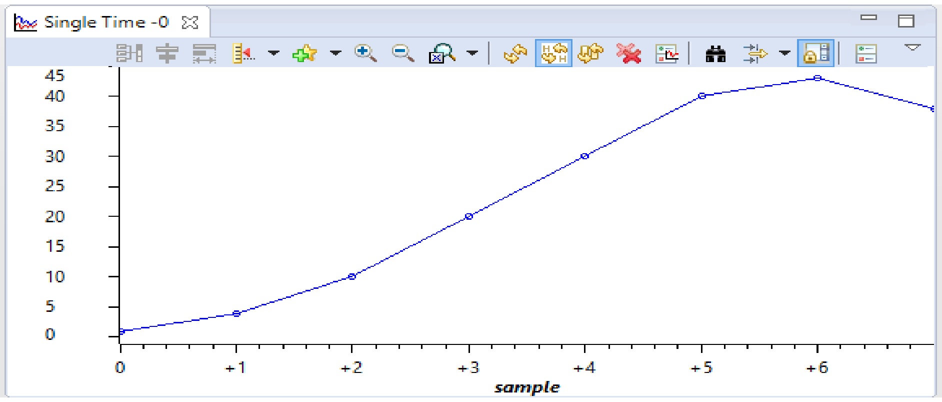
For convolution of two sequences we took two sequences are x[9]=1,2,3,4,5,6,0,0,0 and h[9]= 1,2,3,4,0,0,0,0,0. And result will stored in y[9].

```
#include<stdio.h>
#include <math.h>
int y[9];
int x[9]={1,2,3,4,5,6,0,0,0};
int h[9]={1,2,3,4,0,0,0,0,0};
int main(void)
{         int i,j;
          for(i=0;i<9;i++)
          {
                    y[i]=0;
                    for(j=0;j<=i;j++)
                    {
                    y[i]+=x[j]*h[i-j];
                    }
                    printf("%d\n",y[i]);
          }
          return 0;
```

the Out put of convolution of two sequences.

Out put:

```
[C674X_0] 1
4
10
20
30
40
43
38
24
```



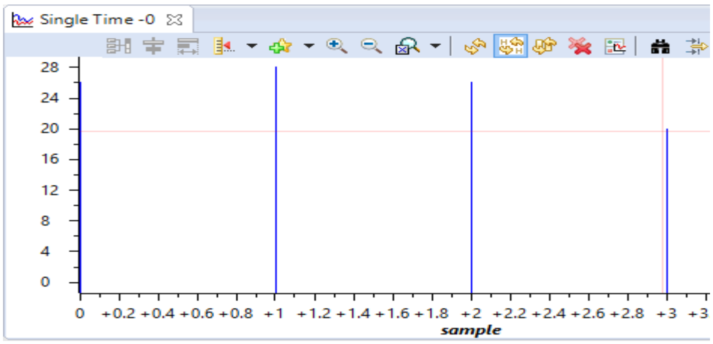
C program for Circular convolution:

For Circular convolution of two sequences we took two sequences are x[4]=1,2,3,4 and h[4]=1,2,3,4. And result will stored in y[4].

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

```
#include<math.h>
int m=4;
int n=4;
int x[4]={1,2,3,4};
int h[4]={1,2,3,4};
int y[4];
int i,j,temp[4],k,x2[4],a[4];
void main()
{
    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]);
}
```

the Out put of circular convolution of two sequences.



- **Out put**

```
[C674X_0] the circular convolution is
26      28      26      20
```

Experiment-3

Discrete Fourier Transform

Aim:

To calculate DFT of discrete time sequence on DSP processor using CCS

Components and Equipment:

TMS320c6748 Processor kit, PC with CCS

Theory:

Fourier analysis describes the transformations between time and frequency domain representations of signals. Four different forms of Fourier transformations (the Fourier transform (FT), the Fourier Series (FS), the discrete-time Fourier transform (DTFT) and the discrete Fourier transform (DFT)) are applicable to different classes of signals according to whether they are discrete or continuous and whether they are periodic and aperiodic.

The DFT is the form of Fourier Transform analysis applicable to signals that are discrete and periodic in both domains (time and frequency). Thus, it transforms a discrete, periodic time domain sequence into a discrete, periodic frequency domain representation. A periodic signal may be characterized entirely by just one cycle. If that signal is discrete, then one cycle comprises a finite number of samples. The Fast Fourier transform is a computationally efficient algorithm for computing the DFT. It requires fewer multiplications than a more straightforward programming implementation of the DFT and its relative advantage in this respect increases with the length of the sample sequences involved. The FFT makes use of the periodic nature of twiddle factors and of symmetries in the structure of the DFT expression. Applicable to spectrum analysis and to filtering, the FFT is one of the most commonly used operations in digital signal processing.

Any periodic sequence x(n) with period N, can be represented as summation of exponentials or

$$x(n) = \sum_{k=0}^{N-1} c_k e^{j2\pi kn/N}$$

This representation is named fourier series where the series coefficients C_k are given by

$$c_k = \frac{1}{N}\sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}$$

C_k which us the amplitude of the frequency spectrum are repeated every N terms, or $C_k = C_{k+N}$

Thus the spectrum of a periodic signal with period N is also periodic with period N.

Discrete Fourier Transform:

The Discrete Fourier Transform for a finite segment of a digital signal containing N samples x[0]; : : : ; x[N - 1] is defined as follows.

$$\boxed{\textbf{DFT :}\qquad X(k) = \sum_{n=0}^{N-1} x(n)\, e^{-j2\pi kn/N}\,, \qquad k = 0, 1, \ldots N-1}$$

here 'k' indicates the index of the frequency.

- **Inverse DFT**

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k) e^{j2\pi kn/N}$$

$$= \frac{1}{N}\sum_{k=0}^{N-1} X(k) W_N^{-kn}, \text{ for } n = 0,1,\ldots, N-1$$

where Twiddle factors = W_N^{kn}

- **Relationship between frequency Bin 'k' and its associated frequency in Hz**

$$f = \frac{kf_s}{N}\ (\text{Hz})$$

- **Amplitude Spectrum:**

$$A_k = \frac{1}{N}|X(k)| = \frac{1}{N}\sqrt{(\text{Real}[X(k)])^2 + (\text{Imag}[X(k)])^2},$$
$$k = 0, 1, 2, \ldots, N-1.$$

- **Phase Spectrum:**

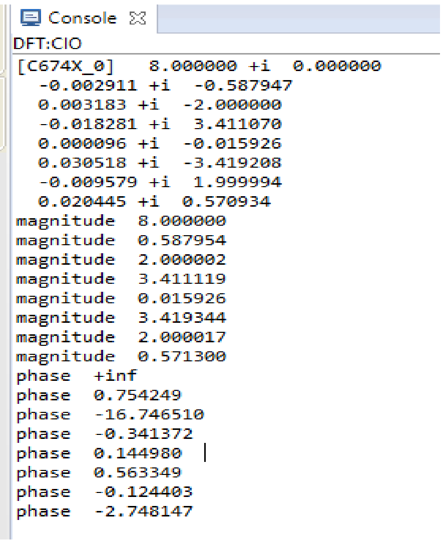
$$\varphi_0 = \tan^{-1}\left(\frac{\text{Imag}[X(0)]}{\text{Real}([X(0)]}\right)$$

C program for 8 point DFT:

```
#include <stdio.h>
#include <math.h>
float x[8]={1,1,2,0,1,2,0,1};
float XR[8];
float XI[8];
float amplitude[8];
float phase[8];
int N=8;
int main(void) {
        int i,j;
        //DFT
        for(j=0;j<N;j++)
        {
                XR[j]=0;
                XI[j]=0;
                for(i=0;i<N;i++)
                {
                        XR[j]=XR[j]+(x[i]*cos(2*3.14*i*j/N));
                        XI[j]=XI[j]-(x[i]*sin(2*3.14*i*j/N));
                }
                printf("  %f +i  %f \n",XR[j],XI[j]);
        }
        for(i=0;i<N;i++)
        {
                amplitude[i]=sqrt((XR[i]*XR[i])+(XI[i]*XI[i]));
                printf("amplitude  %f  \n",amplitude[i]);
        }
        for(i=0;i<N;i++)
        {
                phase[i]=1/tan(XI[i]/XR[i]);
                printf("phase  %f  \n",phase[i]);
        }return 0;}
```

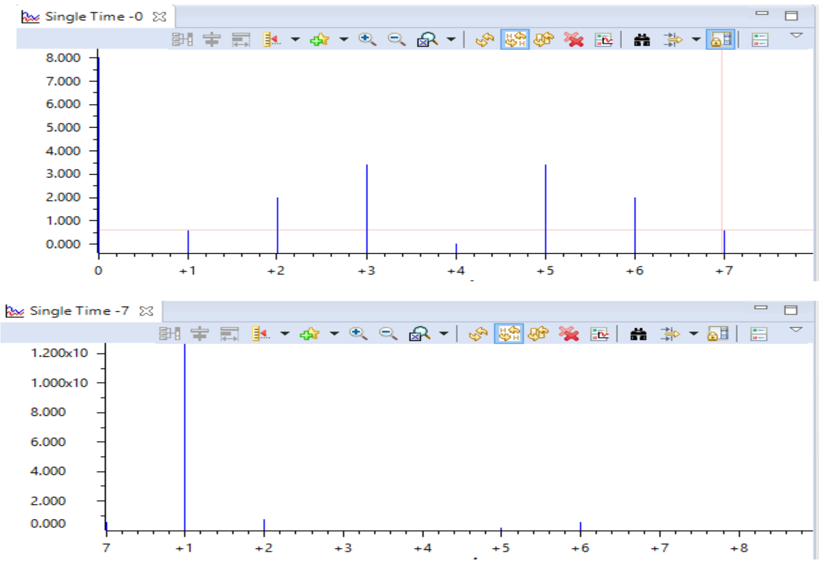
- **Output:**

    8point DFT in console, amplitude and phase plots.



```
[C674X_0]    8.000000 +i   0.000000
   -0.002911 +i   -0.587947
   0.003183 +i   -2.000000
   -0.018281 +i   3.411070
   0.000096 +i   -0.015926
   0.030518 +i   -3.419208
   -0.009579 +i   1.999994
   0.020445 +i   0.570934
magnitude  8.000000
magnitude  0.587954
magnitude  2.000002
magnitude  3.411119
magnitude  0.015926
magnitude  3.419344
magnitude  2.000017
magnitude  0.571300
phase  +inf
phase  0.754249
phase  -16.746510
phase  -0.341372
phase  0.144980
phase  0.563349
phase  -0.124403
phase  -2.748147
```

Console Output



Amplitude and Phase spectrum

- **Output:**

    Then plot the amplitude and phase plots.

Experiment-4
Fast Fourier Transform (FFT)

Aim:

Perform FFT of discrete time sequence on DSP processor TMS320C6748

Components and Equipment:

TMS320c6748 Processor kit, PC with CCS

Theory:

Forward FFT

The N-point complex DFT of a discrete-time signal x[n] can be written

$$X_N[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}$$

where the constants W are referred to as twiddle constants or twiddle factors, where

$$W_N = e^{-j(2\pi/N)}$$

Computing all N values of XN[k] involves the evaluation of N2 terms of the form x[n]W^{kn} $_N$, each of which requires a complex multiplication. For larger N, the computational requirements of the DFT can be very great (N2 complex multiplications).

The FFT algorithm takes advantage of the periodicity

$$W^{k+N} = W^k$$

and symmetry

$$W^{k+N/2} = -W^k$$

of W_N (where N is even).

The twiddle factors $W^{kn}{}_N$ for N = 8 plotted as vectors in the complex plane. Due to the periodicity of Wkn N , the 64 different combinations of n and k used in evaluation of Equation result in only 8 distinct values. The FFT makes use of this small number of precomputed and stored values rather than computing each one as it is required. Another in which the FFT saves computational effort is by decomposing N-point DFTs into combinations of N=2-point DFTs. The decompositions are carred out until 2-point DFTs have been reached, which do not involve any multiplication. This allows to speed up the process a lot. The FFT is much faster then the classical DFT as its complexity is O(Nlog(N)), versus O(N2) for the classical DFT.

Inverse FFT

The inverse discrete Fourier transform (IDFT) converts a discrete frequency domain sequence XN[k] into a corresponding discrete-time domain sequence x[n]. It is defined as

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X_N[k] W_N^{-kn}$$

Comparing this with the forward DFT Equation 6.3, we can see that the forward FFT algorithm described previously can be used to compute the inverse DFT if the following changes are made:

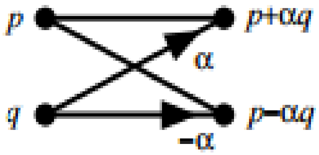
1. Scale the result by 1=N.

2. Replace twiddle factors W^{kn}

N by their complex conjugates $W\text{-}^{kn}{}_N$.

Butterflies and Bit-Reversal.

The FFT algorithm decomposes the DFT into log2 N stages, each of which consists of N/2 butterfly computations. Each butterfly takes two complex numbers p and q and computes from them two other numbers, p + αq and p − αq, where α is a complex number. Below is a diagram of a butterfly operation.

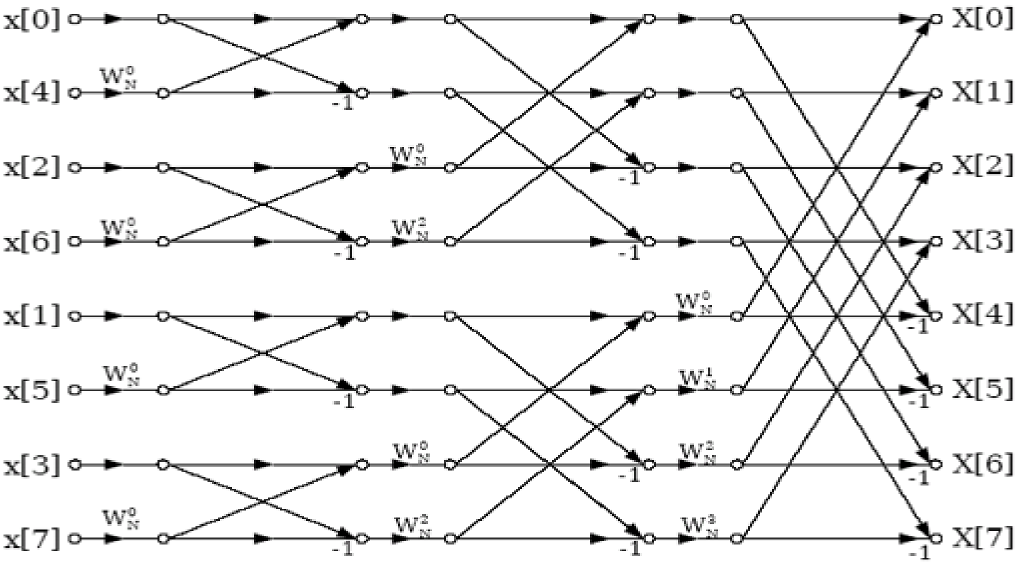


2 point Butterfly structure

8 Point FFT decimation in time:

Algorithm:

Taking as an example the reordered length N sequence xðnÞ in the case of DIT, the index of each sample may be represented by a log2(N) bit binary number (recall that in the foregoing examples, N is an integer power of 2). If the binary representations of the N index values 0 to (N_1) have the order of their bits reversed, for example, 001 becomes 100, 110 becomes 011, and so on, then the resulting N values represent the indices of the input sequence xðnÞ in the order that they appear at the input to the FFT structure. In the case of N = 8 (Figure 4.7), the input values x(n) are ordered x(0); x(4); x(2); x(6); x(1); x(5); x(3); x(7). The binary representations of these indices are 000, 100, 010, 110, 001, 101, 011, 111. These are the bit-reversed versions of the sequence 000, 001, 010, 011, 100, 101, 110, 111. The bit-reversed interpretation of the reordering holds for all N (that are integer powers of 2).



Block diagram representation of 8-point FFT using decimation-in-time with radix-2.

C program for FFT decimation in time:

- For finding 8 FFT take sequence x[8]={1,1,2,0,1,2,0,1}
- Twiddle factors are WR[4]={1.0,0.707,0.0,-0.707}; WI[4]={0.0,-0.707,-1.0,-0.707};

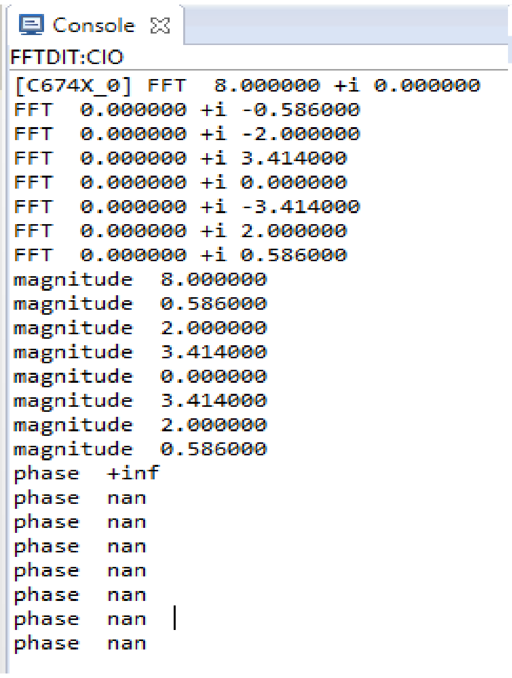
  WR for real and WI for Imaginary.

  Compute the 8 point FFT, Amplitude and phase.

```
#include<stdio.h>
#include<math.h>
double x[8]={1,1,2,0,1,2,0,1};//given 8 point sequence
double XR[8]={0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
double XI[8]={0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
double Xr[8]={0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
double Xi[8]={0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
double WR[4]={1.0,0.707,0.0,-0.707};//twiddle Factors
double WI[4]={0.0,-0.707,-1.0,-0.707};
double amplitude[8],phase[8];
int main(void)
{ int i,j;
        XR[0]=x[0];// Bit reversol ordder
        XR[1]=x[4];
        XR[2]=x[2];
        XR[3]=x[6];
        XR[4]=x[1];
        XR[5]=x[5];
        XR[6]=x[3];
        XR[7]=x[7];
```

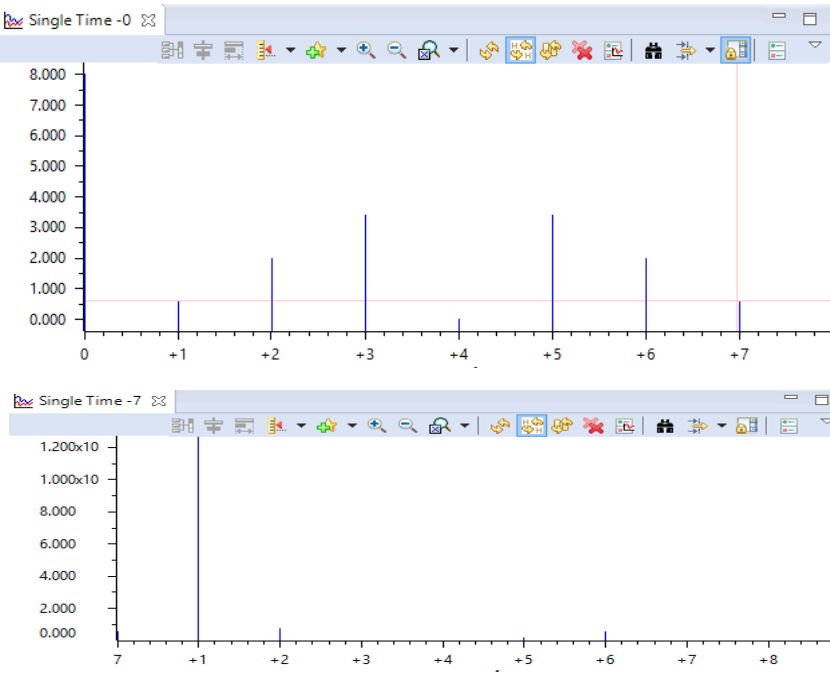
```
    //first stage
        i=0;
    for(j=0;j<=3;j++)
    {
        double sum1r=XR[i]+XR[i+1];
        double sum1i=XI[i]+XI[i+1];
        double sum2r=(XR[i]-(WR[0]*XR[i+1])+(WI[0]*XI[i+1]));
        double sum2i=(XI[i]-(WR[0]*XI[i+1])-(WI[0]*XR[i+1]));
        XR[i]=sum1r;
        XI[i]=sum1i;
        XR[i+1]=sum2r;
        XI[i+1]=sum2i;
        i=i+2;}//second stage
    i=0;
    int x;
    for(x=0;x<2;x++)
    {       int w=0;
        for(j=0;j<2;j++)
        {
            double sum1r=(XR[i]+(WR[w]*XR[i+2])-(WI[w]*XI[i+2]));
            double sum1i=(XI[i]+(WR[w]*XI[i+2])+(WI[w]*XR[i+2]));
            double sum2r=(XR[i]-(WR[w]*XR[i+2])+(WI[w]*XI[i+2]));
            double sum2i=(XI[i]-(WR[w]*XI[i+2])-(WI[w]*XR[i+2]));
            XR[i]=sum1r;
            XI[i]=sum1i;
            XR[i+2]=sum2r;
            XI[i+2]=sum2i;
            i=i+1;
            w=w+2;
        }
        i=i+2;
    }//third stage
            i=0;
            int w=0;
            for(j=0;j<4;j++)
            {
                double sum1r=(XR[i]+(WR[w]*XR[i+4])-(WI[w]*XI[i+4]));
                double sum1i=(XI[i]+(WR[w]*XI[i+4])+(WI[w]*XR[i+4]));
                double sum2r=(XR[i]-(WR[w]*XR[i+4])+(WI[w]*XI[i+4]));
                double sum2i=(XI[i]-(WR[w]*XI[i+4])-(WI[w]*XR[i+4]));
                XR[i]=sum1r;
                XI[i]=sum1i;
                XR[i+4]=sum2r;
                XI[i+4]=sum2i;
                i=i+1;
                w=w+1;
            }
    for(i=0;i<8;i++)
    {           printf("FFT  %f +i %f \n",XR[i],XI[i]);             }
    for(i=0;i<8;i++)  {
        amplitude[i]=sqrt((XR[i]*XR[i])+(XI[i]*XI[i]));
        printf("amplitude  %f  \n",amplitude[i]);     }
    for(i=0;i<8;i++)
    {
        phase[i]=1/tan(XI[i]/XR[i]);
        printf("phase  %f  \n",phase[i]);
    }
return 0;
}
```

- Run and Debug the program.



```
FFTDIT:CIO
[C674X_0] FFT  8.000000 +i 0.000000
FFT  0.000000 +i -0.586000
FFT  0.000000 +i -2.000000
FFT  0.000000 +i 3.414000
FFT  0.000000 +i 0.000000
FFT  0.000000 +i -3.414000
FFT  0.000000 +i 2.000000
FFT  0.000000 +i 0.586000
magnitude  8.000000
magnitude  0.586000
magnitude  2.000000
magnitude  3.414000
magnitude  0.000000
magnitude  3.414000
magnitude  2.000000
magnitude  0.586000
phase  +inf
phase  nan
phase  nan
phase  nan
phase  nan
phase  nan
phase  nan
phase  nan
```

Console Output

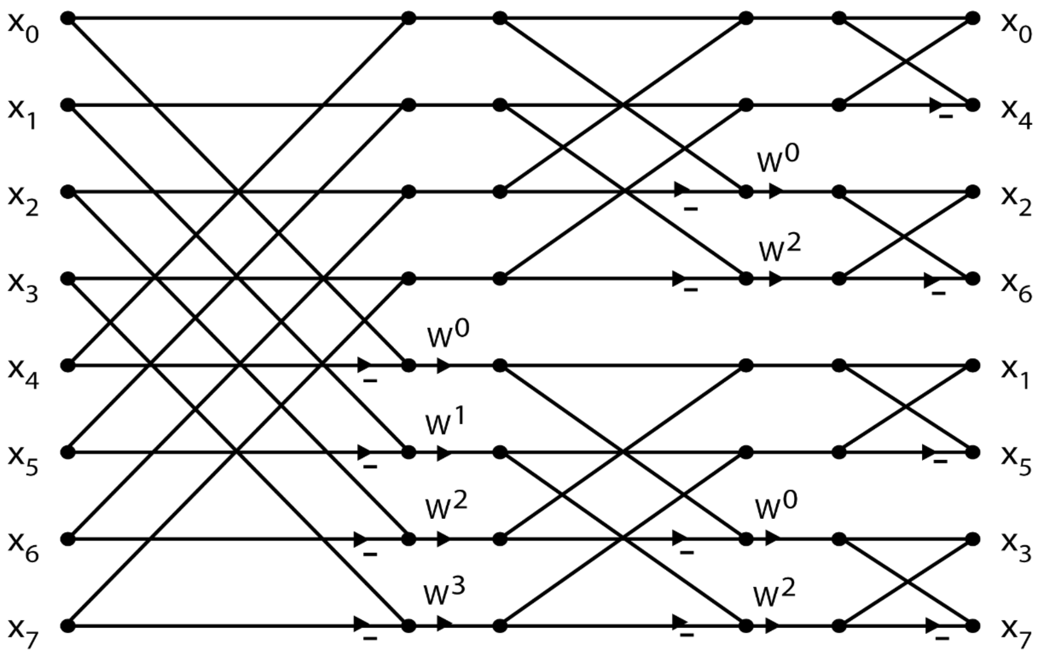


Amplitude and Phase spectrum

8 Point FFT decimation in frequency:

Algorithm:

The 8-point DFT has thus been decomposed into a structure (Figure 4.9) that comprises only a small number of multiplications (by factors other than_1). The FFT is not an approximation of the DFT. It yields the same result as the DFT with fewer computations required. This reduction becomes more and more important, and advantageous, with higher order FFTs. The decimation-in-frequency process may be considered as taking the N-point input sequence x(n) in sequence and reordering the output sequence X_N (k) in pairs, corresponding to the outputs of the final stage of N/2 2-point DFT blocks.



Block diagram representation of 8-point FFT using decimation-in-frequency with radix-2.

C program for FFT decimation in frequency:

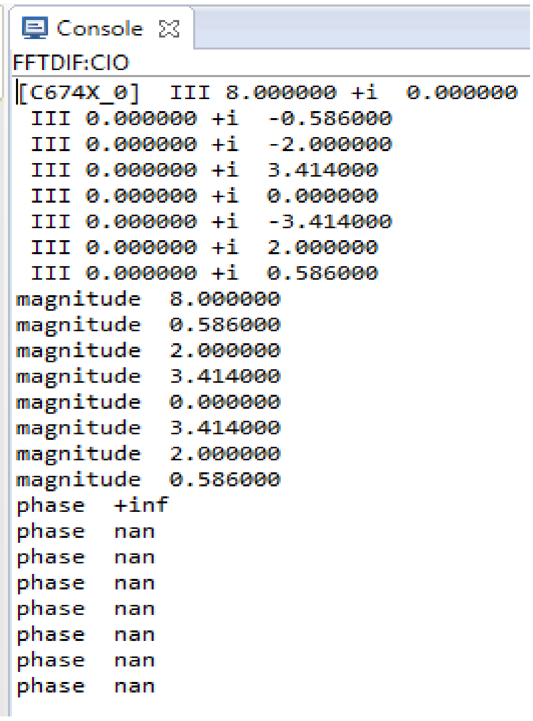
- For finding 8 FFT take sequence x[8]={1,1,2,0,1,2,0,1}
- Twiddle factors are WR[4]={1.0,0.707,0.0,-0.707}; WI[4]={0.0,-0.707,-1.0,-0.707};

  WR for real and WI for Imaginary.

  Compute the 8 point FFT , Amplitude and phase.

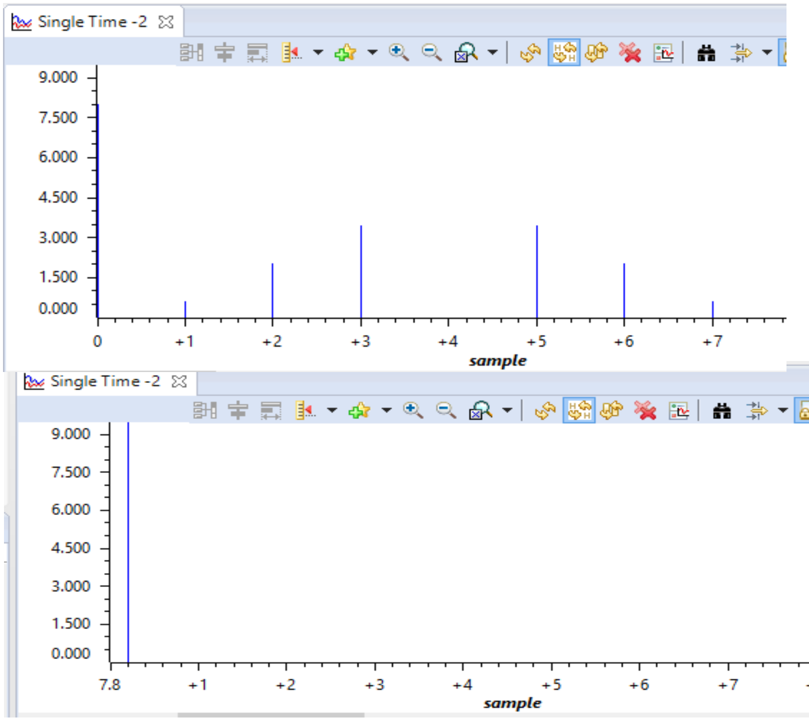
```
#include<stdio.h>
#include<math.h>
double XR[8]={1,1,2,0,1,2,0,1};
double XI[8]={0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
double Xr[8]={0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
double Xi[8]={0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
double WR[4]={1.0,0.707,0.0,-0.707};
double WI[4]={0.0,-0.707,-1.0,-0.707};
double amplitude[8],phase[8];
int main(void)
{ int i,j;
            //first stage
                        i=0;
                        int w=0;
                        for(j=0;j<4;j++)
                        {
                                    double sum1r=XR[i]+XR[i+4];
                                    double sum1i=XI[i]+XI[i+4];
                                    double sum2r=((XR[i]-XR[i+4])*WR[w])-((XI[i]-XI[i+4])*WI[w]);
                                    double sum2i=((XI[i]-XI[i+4])*WR[w])+((XR[i]-XR[i+4])*WI[w]);
                                    XR[i]=sum1r;
                                    XI[i]=sum1i;
                                    XR[i+4]=sum2r;
            XI[i+4]=sum2i;
            i=i+1;
            w=w+1;
        }
        //second stage
        i=0;
        int x;
        for(x=0;x<2;x++)
        {
            int w=0;
            for(j=0;j<2;j++)
            {
            double sum1r=XR[i]+XR[i+2];
            double sum1i=XI[i]+XI[i+2];
            double sum2r=((XR[i]-XR[i+2])*WR[w])-((XI[i]-XI[i+2])*WI[w]);
            double sum2i=((XI[i]-XI[i+2])*WR[w])+((XR[i]-XR[i+2])*WI[w]);
                XR[i]=sum1r;
                XI[i]=sum1i;
                XR[i+2]=sum2r;
                XI[i+2]=sum2i;
                i=i+1;
                w=w+2;
            }
            i=i+2;
        }
//third stage
    i=0;
    w=0;
    for(j=0;j<=3;j++)
    {
        double sum1r=XR[i]+XR[i+1];
        double sum1i=XI[i]+XI[i+1];
        double sum2r=((XR[i]-XR[i+1])*WR[w])-((XI[i]-XI[i+1])*WI[w]);
        double sum2i=((XI[i]-XI[i+1])*WR[w])+((XR[i]-XR[i+1])*WI[w]);
        XR[i]=sum1r;
        XI[i]=sum1i;
        XR[i+1]=sum2r;
        XI[i+1]=sum2i;
        i=i+2;
    }
    Xr[0]=XR[0];
    Xr[1]=XR[4];
    Xr[2]=XR[2];
    Xr[3]=XR[6];
    Xr[4]=XR[1];
    Xr[5]=XR[5];
    Xr[6]=XR[3];
    Xr[7]=XR[7];
            Xi[0]=XI[0];
            Xi[1]=XI[4];
            Xi[2]=XI[2];
            Xi[3]=XI[6];
            Xi[4]=XI[1];
            Xi[5]=XI[5];
            Xi[6]=XI[3];
            Xi[7]=XI[7];
    printf(" III %f +i  %f  \n",Xr[0],Xi[0]);
    printf(" III %f +i  %f  \n",Xr[1],Xi[1]);
    printf(" III %f +i  %f  \n",Xr[2],Xi[2]);
    printf(" III %f +i  %f  \n",Xr[3],Xi[3]);
    printf(" III %f +i  %f  \n",Xr[4],Xi[4]);
printf(" III %f +i  %f  \n",Xr[5],Xi[5]);
printf(" III %f +i  %f  \n",Xr[6],Xi[6]);
printf(" III %f +i  %f  \n",Xr[7],Xi[7]);
for(i=0;i<8;i++)
        {
        amplitude[i]=sqrt((Xr[i]*Xr[i])+(Xi[i]*Xi[i]));
        printf("amplitude  %f  \n",amplitude[i]);
        }
for(i=0;i<8;i++)
        {
        phase[i]=1/tan(Xi[i]/Xr[i]);
        printf("phase  %f  \n",phase[i]);
        }
return 0;
}
```

- Output:
- Run and Debug the program.



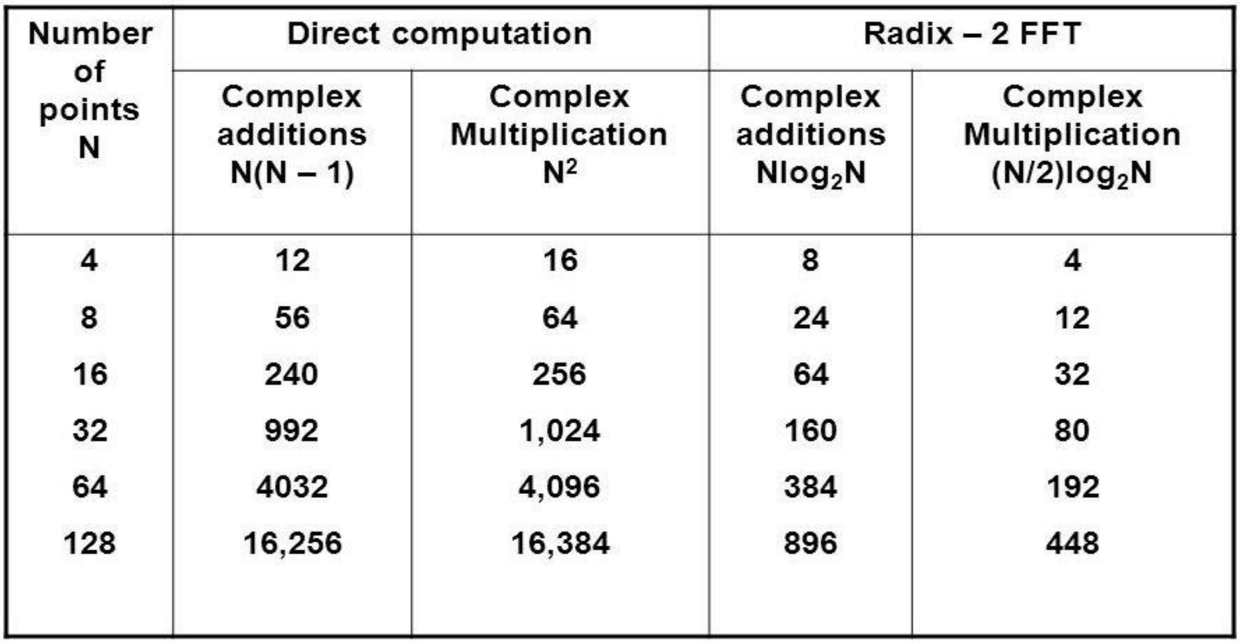
```
[C674X_0]  III 8.000000 +i  0.000000
 III 0.000000 +i  -0.586000
 III 0.000000 +i  -2.000000
 III 0.000000 +i  3.414000
 III 0.000000 +i  0.000000
 III 0.000000 +i  -3.414000
 III 0.000000 +i  2.000000
 III 0.000000 +i  0.586000
magnitude  8.000000
magnitude  0.586000
magnitude  2.000000
magnitude  3.414000
magnitude  0.000000
magnitude  3.414000
magnitude  2.000000
magnitude  0.586000
phase  +inf
phase  nan
phase  nan
phase  nan
phase  nan
phase  nan
phase  nan
phase  nan
```

Console Output



Amplitude and Phase spectrum

Comparison of number of computations in direct DFT and FFT



Number of points N	Direct computation		Radix – 2 FFT	
	Complex additions N(N – 1)	Complex Multiplication N^2	Complex additions $N\log_2 N$	Complex Multiplication $(N/2)\log_2 N$
4	12	16	8	4
8	56	64	24	12
16	240	256	64	32
32	992	1,024	160	80
64	4032	4,096	384	192
128	16,256	16,384	896	448

The FFT is much faster than the classical DFT as its complexity is O(Nlog(N)), versus O(N2) for the classical DFT.

Experiment-5

Aim:

To design FIR and IIR filter on DSP processor using CCS

Components and Equipment:

TMS320c6748 Processor kit, PC with CCS

Theory: Filtering is one of the most common DSP operations. Filtering can be used for noise suppression, signal enhancement, removal or attenuation of a specific frequency, or to perform special operations such as differentiation, integration, etc. Filters can be thought of, designed and implemented in either the time domain or the frequency domain. There are two main types of filtering techniques. The first kind of filters are the Finite Impulse Response (FIR) filters. Their non-recursive version directly implements the convolution operation with a finite impulse response. The second kind of filters are the Infinite Impulse Response Filters (IIR). These filters are always recursive as they use feedback information to compute their output. This chapter is a small introduction to filtering. This is not an exhaustive course on filter design. It gives some information on how to implement digital filters in an efficient way on a DSP.

FIR filters are filters whose impulse response is of finite duration. These filters are very stable, in contrast to IIR filters. Most FIR filters are non-recursive and are carried-out by convolution. Thus, the output y of a linear time invariant (LTI) system is determined by convolving its input signal x with its impulse response h. For a discrete time

filter, the output is therefore a weighted sum of the current sample and a finite number of previous input samples. Equation describes this operation. The impulse response of a Nth-order FIR filter has a length of N + 1 samples.

Finite Impulse Response Filters:

I. Clearly specify the filter specifications.

Ex Order, Sampling Rate, Cut off Frequency.

II. Compute the cut-off frequency Wc or center frequency for band pass filters.

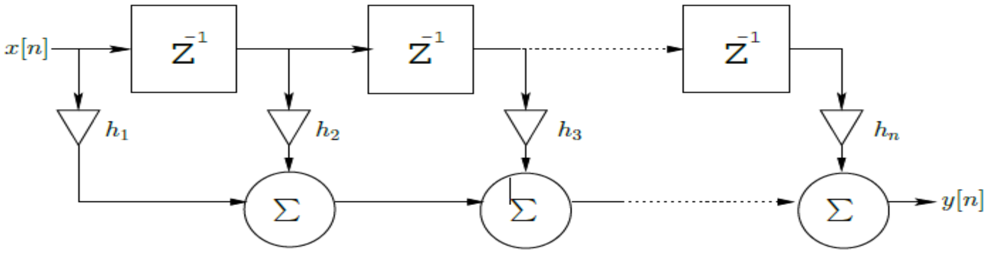
Ex Wc=2*PI*fc/Fs

III. Compute the desired Impulse Response h(n) using particular Window.

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

$$y[n] = \sum_{i=0}^{N} h[i]x[n-i] = h[0]x[n] + h[1]x[n-1] + \cdots + h[N]x[n-N]$$

the block diagram of a non-recursive FIR filter. The diagram is a graphical representation of above Equation.



Block diagram of a non-recursive FIR filter.

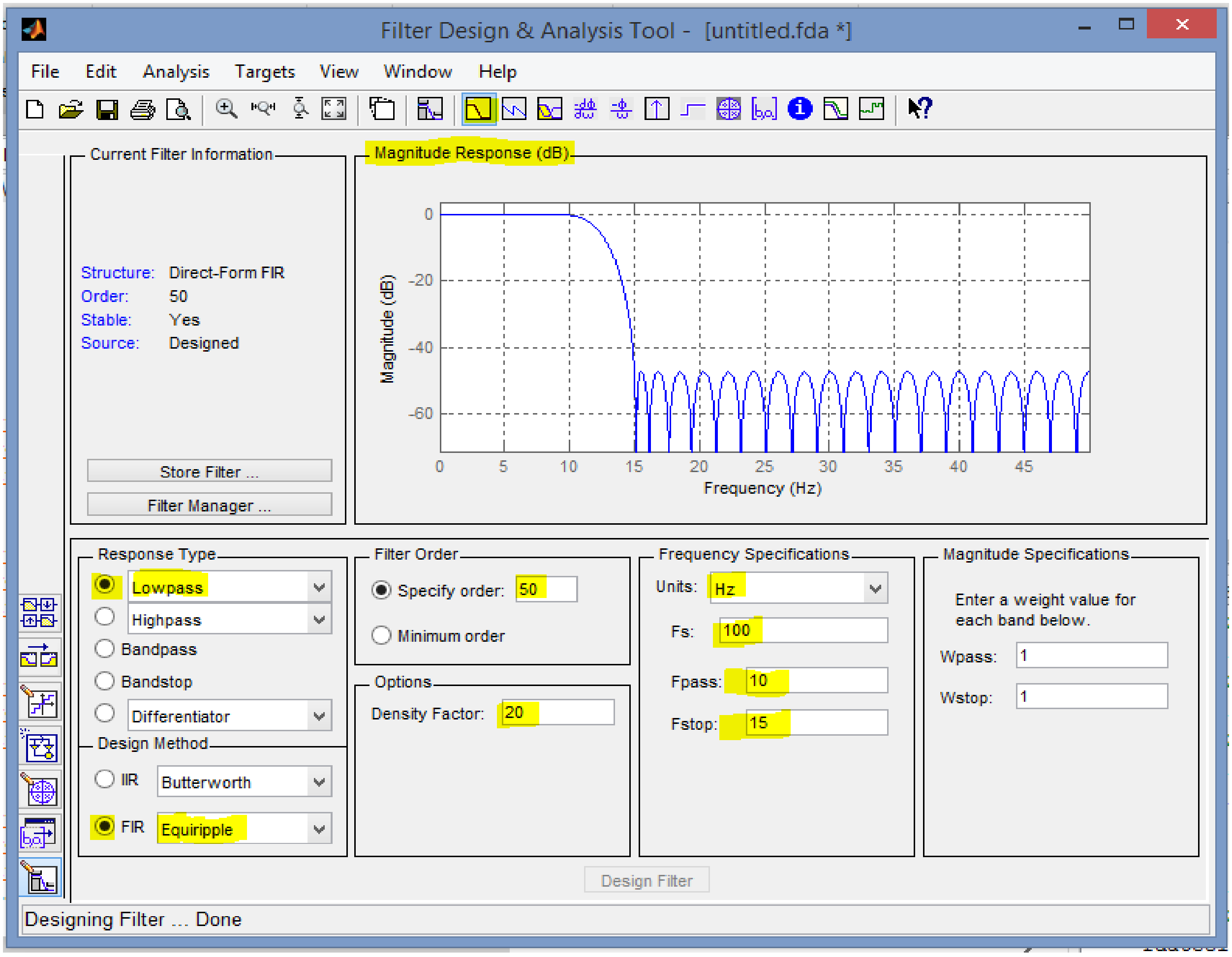
FIR Low pass filter:

Using MATLAB to determine filter coefficients:

The MATLAB filter design and analysis tool fdatool can be used to calculate FIR filter coefficients and to export them to the MATLAB workspace.

MATLAB→ type **fdatool** in command window→ Refer the fig. for the filter properties

To generate C header file: Targets→ Genarate C header file→ give location to store C header file. Then include this file in CCstudio. Similarly We can design all types of filters.



Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

Response Type
● Lowpass
○ Highpass
○ Bandpass
○ Bandstop
○ Differentiator

Design Method
○ IIR Butterworth
● FIR Equiripple

Filter Order
● Specify order: 50
○ Minimum order

Options
Density Factor: 20

Frequency Specifications
Units: Hz
Fs: 100
Fpass: 10
Fstop: 15

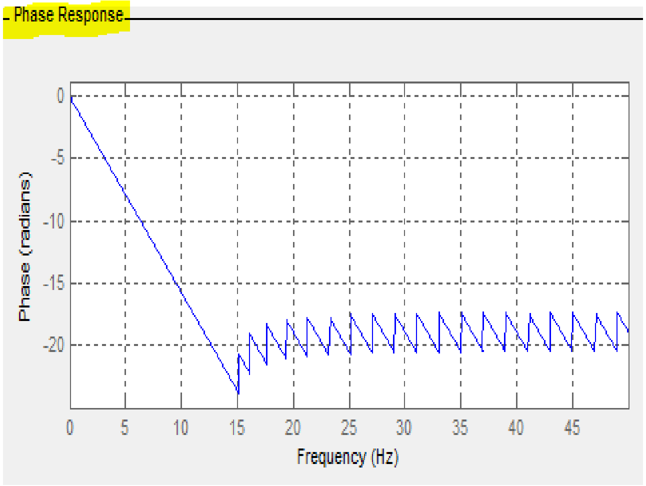
Magnitude Specifications
Enter a weight value for each band below.
Wpass: 1
Wstop: 1

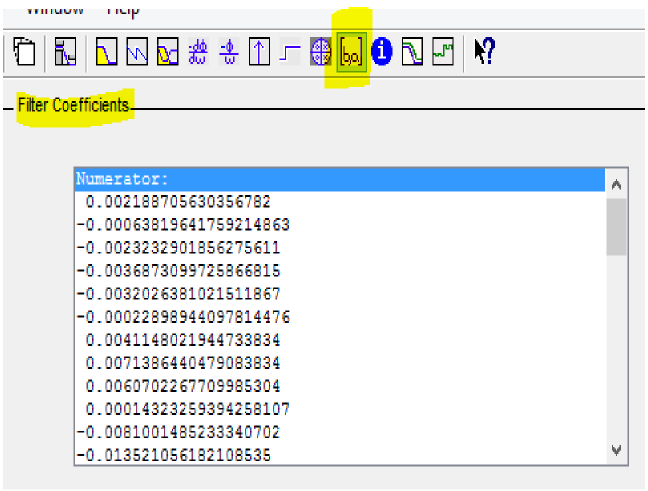
Designing Filter ... Done

the filter properties

To design low pass filter we need to know cut off frequency, sampling frequency and order of filter.

For example design low pass filter which has cut off frequency 10Hz, sampling frequency has 100Hz and order is specified to 50.





```
Numerator:
 0.002188705630356782
-0.00063819641759214863
-0.0023232901856275611
-0.0036873099725866815
-0.0032026381021511867
-0.00022898944097814476
 0.0041148021944733834
 0.0071386440479083834
 0.0060702267709985304
 0.00014323259394258107
-0.0081001485233340702
-0.013521056182108535
```

Phase Response **Filter Coefficients**

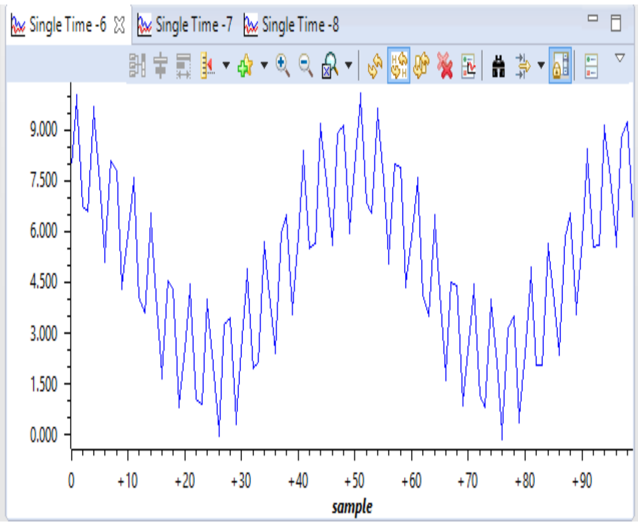
C program to implement FIR Low pass filter:

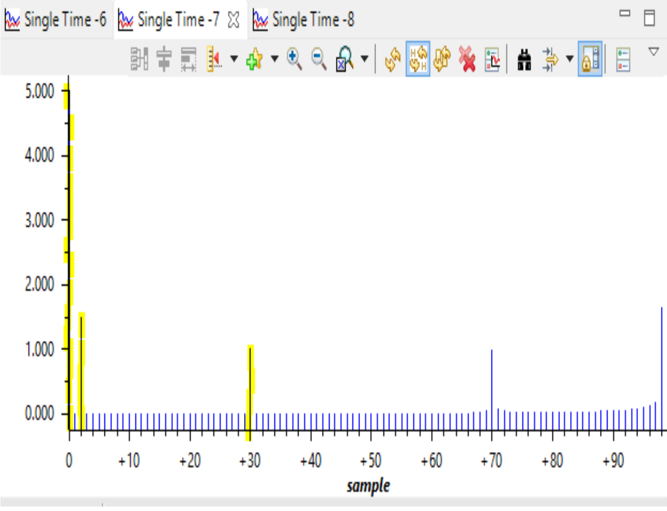
Program implements an FIR low pass filter which pass the frequencies below 10Hz and reject the frequencies above 10Hz.

```
#include <stdio.h>
#include <math.h>
float x[100];
float XR[100];
float XI[100];
float b[51];
float y[100];
int N=100;
float amplitude[100];
float amplitude1[100];
float phase[100];
float h[51]={-0.03012856352907,0.07518556094106, 0.0347107415489, 0.020773388533,
0.01458846949463,        0.009901508624105,        0.004815070324426,-0.0009912477392302,-
0.007236113317258, -0.01328621421835, -0.01853229090741, -0.02212338859297,         -
0.02350058075885, -0.02208352290888, -0.01748432902234,-0.009758034457342,
0.001093534289564,  0.01455515002771,  0.02991549107912,  0.04627745357896,
0.06270133716713,  0.07800956621943,  0.09121853292119,   0.1013742672991,
0.10783870071,   0.1100289455531,    0.10783870071,   0.1013742672991,
0.09121853292119,  0.07800956621943,  0.06270133716713,  0.04627745357896,
0.02991549107912,  0.01455515002771, 0.001093534289564,-0.009758034457342,         -
0.01748432902234, -0.02208352290888, -0.02350058075885, -0.02212338859297,         -
0.01853229090741, -0.01328621421835,-0.007236113317258,-0.0009912477392302,
0.004815070324426, 0.009901508624105,  0.01458846949463,    0.020773388533,
0.0347107415489,  0.07518556094106, -0.03012856352907};
int main(void) {
int i,j,k,n;
for (i=0;i<100;i++)
{           x[i]=0;
            x[i]=5+(2*sin(2*3.14*30*i/100))+(3*cos(2*3.14*2*i/100));
}//DFT
for(j=0;j<N;j++)
{
            XR[j]=0;
            XI[j]=0;
            for(i=0;i<N;i++)
            {
                        XR[j]=XR[j]+(x[i]*cos(2*3.14*i*j/N));
                        XI[j]=XI[j]-(x[i]*sin(2*3.14*i*j/N));
            }
}
for(i=0;i<N;i++)
{
            amplitude[i]=sqrt((XR[i]*XR[i])+(XI[i]*XI[i]));
}//filter
for(n=0;n<100;n++)
{
            y[n]=0;
            b[0]=x[n];
            for(i=51;i>0;i--)
            {
                        b[i]=b[i-1];
            }
            for(i=0;i<51;i++)
            {
                        y[n]=y[n]+(h[i]*b[i]);
```

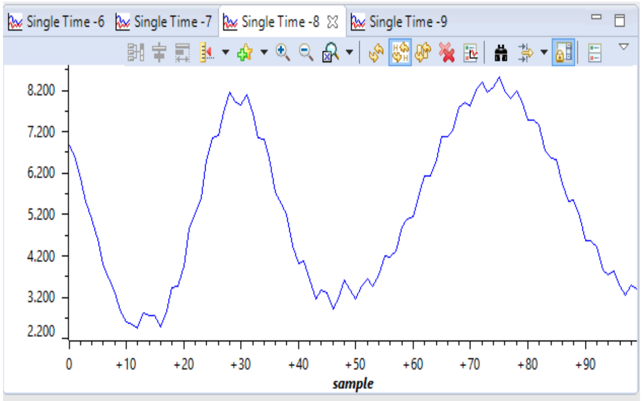
```
        }
        }//DFT
for(j=0;j<N;j++)
{
        XR[j]=0;
        XI[j]=0;
        for(i=0;i<N;i++)
        {
                XR[j]=XR[j]+(y[i]*cos(2*3.14*i*j/N));
                XI[j]=XI[j]-(y[i]*sin(2*3.14*i*j/N));
        }
}
for(i=0;i<N;i++)
{
        amplitude1[i]=sqrt((XR[i]*XR[i])+(XI[i]*XI[i]));
}
        return 0;}
```

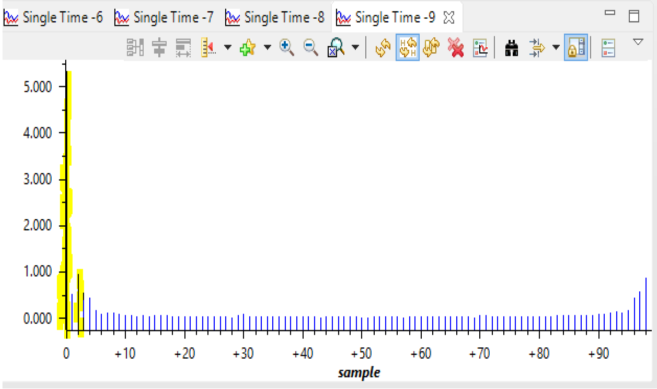
- **Output:**





Input Signal **Frequency Spectrum**





Filtered Output **Filtered Frequency Spectrum**

Infinite Impulse Response Filters:

A general input–output equation of the form

$$y(n) = \sum_{k=0}^{M} b_k x(n-k) - \sum_{l=1}^{N} a_l y(n-l)$$

or, equivalently,

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + \cdots + b_Mx(n-M)$$
$$-a_1y(n-1) - a_2y(n-2) - \cdots - a_Ny(n-N)$$

IIR Low pass filter:

Traditionally, IIR filter design is based on the concept of transforming a continuous time, or analog, design into the discrete-time domain. Butterworth, Chebyshev, Bessel, and elliptic classes of analog filter are widely used. In this example, a second-order, type II Chebyshev low-pass filter with 2 dB of pass band ripple and a cutoff frequency of 3 Hz is used.

$$H(s) = \frac{232.29}{S^2+15.15s+292.43}$$

Starting with the filter transfer function, we can make use of the Laplace transform pair

$$L\{Ae^{-\alpha t}\sin(\omega t)\} = \frac{A\omega}{s^2+2\alpha s+(\alpha^2+\omega^2)}$$

The z-transform pair

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0.021450\ z^{-1}}{1-1.8323\ z^{-1}+0.8594\ z^{-2}}$$

From H(z), the following difference equation may be derived:

$$\boldsymbol{y(n) = 0.021450\ x(n-1) + 1.8323y(n-1) - 0.8594y(n-2)}$$

we can see that a_1 = 1.8323, a_2 = -0.8594 and b_0 = 0.0000 and b_1 =0.021450

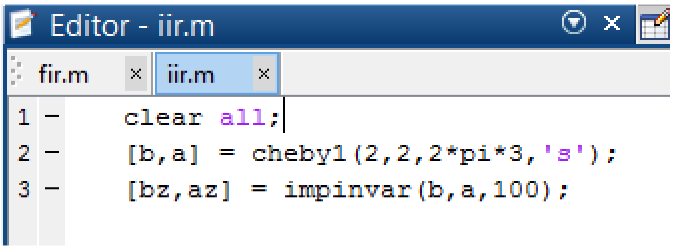
Using MATLAB to determine filter coefficients:

Using MATLAB, the coefficients of this s-transfer function can be generated by typing **>> [b,a] = cheby1(2,2,2*pi*10,’s’);** at the command line.

Our task is to transform this design into the discrete-time domain. One method of achieving this is the impulse invariance method. In order to apply the impulse invariant method using MATLAB, type

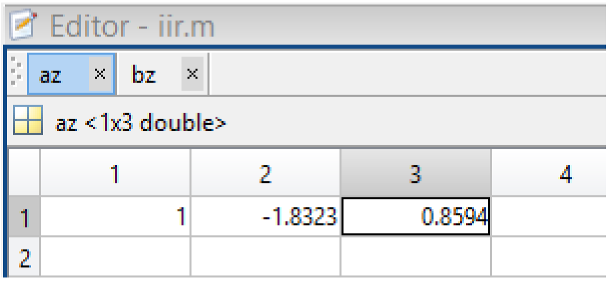
>> [bz,az] = impinvar(b,a,100);

This discrete-time filter has the property that its discrete-time impulse response h(n) is equal to samples of the continuous-time impulse response h(t).The values of bz and az are copy and include in CCS project.



```
clear all;
[b,a] = cheby1(2,2,2*pi*3,'s');
[bz,az] = impinvar(b,a,100);
```

Figure 5.10: Matlab code for determine filter coefficients



az <1x3 double>

	1	2	3	4
1	1	-1.8323	0.8594	
2				

Filter coefficients

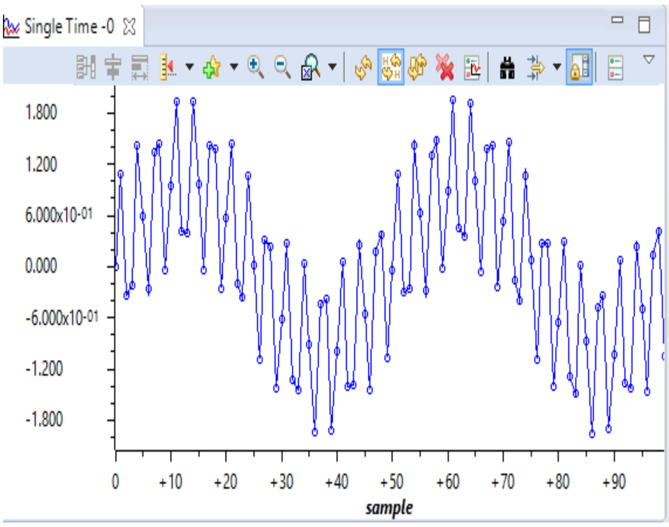
C program to implement IIR low pass filter

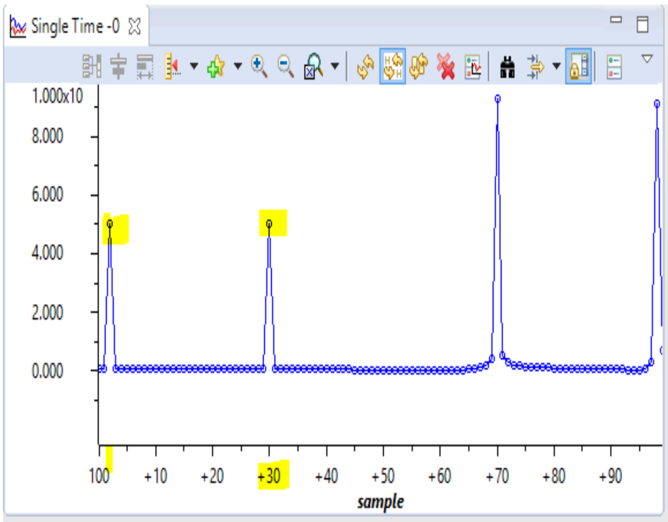
Program implements a generic IIR filter using cascaded direct form II second-order sections and coefficient values stored in a same file.

```
#include <stdio.h>
#include <math.h>
#define NUM_SECTIONS 1
float b[NUM_SECTIONS][3]={ {0.0, 0.0214501370969255, 0.0} };//filter coefficients
float a[NUM_SECTIONS][3]={ {1.0, -1.83233418096295, 0.859404279941073} };//filter coefficients
float x[100];
int N=100;
float XR[100];
float XI[100];
float m1[100];
float m2[100];
float y[100];
float w[NUM_SECTIONS][2]={0};// no of sections
int main(void) {
        int i,j;
        for (i=0;i<N;i++)
                {       x[i]=0;
                        x[i]=sin(2*3.14*2*i/N)+sin(2*3.14*30*i/N);
                }
```

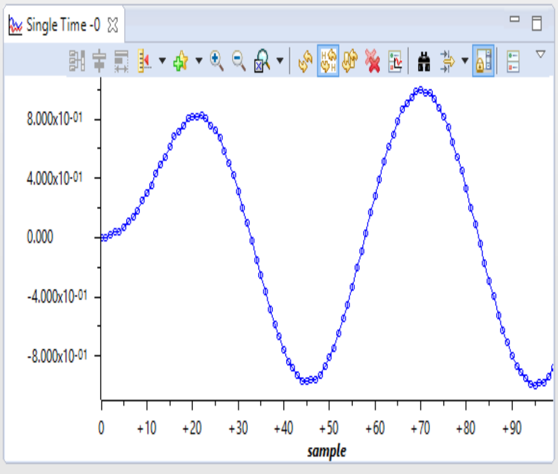
```
    //DFT of input signal
    for(j=0;j<N;j++)
    {
        XR[j]=0;
        XI[j]=0;
        for(i=0;i<N;i++)
        {
            XR[j]=XR[j]+(x[i]*cos(2*3.14*i*j/N));
            XI[j]=XI[j]-(x[i]*sin(2*3.14*i*j/N));
        }
        }
    for(i=0;i<N;i++)
    {
        m1[i]=sqrt((XR[i]*XR[i])+(XI[i]*XI[i])/N);
    }
    //iirType equation here.
    int section;
    float input;
    float wn,yn;
    for (i=0;i<N;i++)
    {
        input =x[i];
        for(section=0;section<NUM_SECTIONS;section++)
        {
            wn=input-a[section][1]*w[section][0]-a[section][2]*w[section][1];
            yn=b[section][0]*wn+b[section][1]*w[section][0]+b[section][2]*w[section][1];
            w[section][1]=w[section][0];
            w[section][0]=wn;
            input=yn;
        }
        y[i]=yn;
    }
    //DFT of Filtered signal
    for(j=0;j<N;j++)
        {
            XR[j]=0;
            XI[j]=0;
            for(i=0;i<N;i++)
            {
                XR[j]=XR[j]+(y[i]*cos(2*3.14*i*j/N));
                XI[j]=XI[j]-(y[i]*sin(2*3.14*i*j/N));
            }
            }
    for(i=0;i<N;i++)
    {
        m2[i]=sqrt((XR[i]*XR[i])+(XI[i]*XI[i])/N);
    }
    return 0;
}
```

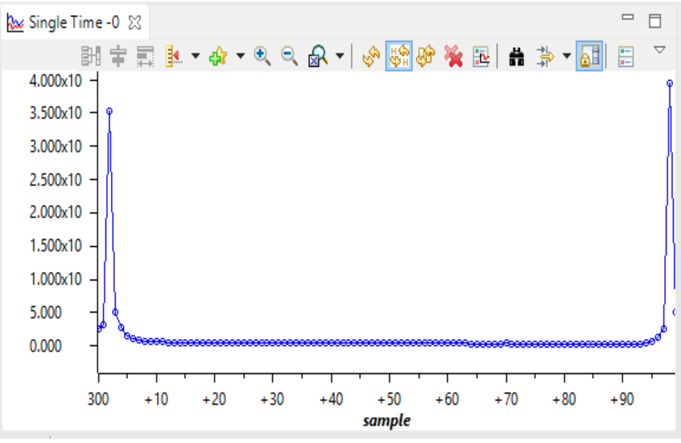
- **Output**





Input Signal **Frequency Spectrum**





Filtered Output **Filtered Frequency Spectrum**