# DSP Signal Generator Implementation On C6713 DSK

Communications Laboratory – University of Kassel

Project Work
by
Kamran Khan (28247463)

Supervisor: Dipl.–Ing. Thomas Edlich

# Declaration

With this I declare that the present Project Report was made by myself. Prohibited means were not used and only the aids specified in the Project Report were applied. All parts which are taken over word-to-word or analogous from literature and other publications are quoted and identified.

Kassel, December 9, 2009.

*Kamran Khan*

# Table of Contents

**Acknowledgements**

**1 Introduction…...……………………………………………………………………………1**

1.1 Hardware and Software Tool……………………………………………………………...3

1.2 Signal Generator…..…………………………………………………………………………6
 1.2.1 Function Generator………………………………………………………………….6
 1.2.2 Arbitrary Waveform Generator……………………………………………………...7
 1.2.3 Applied Fields for Signal Generator…………………………………………………8

1.3 Project Motivation……………………………………………………………………………9

**2 Concept…………………………………………………………………………………10**

2.1 Waveform Generation…………………………………………………………………….10

2.2 Pseudo Random Noise Sequence Generation……………………………………………..11

2.3 Digital Modulation Schemes………………………………………………………………11

**3 Implementation………………………………………………………..…………12**

3.1 Waveform Generation………………………………………………………………………12
 3.1.1 Sine Wave Generation……………………………………………………………...12
 3.1.2 Square Wave Generation…………………………………………………………...15
 3.1.3 Triangular Wave Generation…………………………………………………………16
 3.1.4 Sawtooth Wave Generation………………………………………………………...19
 3.1.5 Multi-tone Signal Generation……………………………………………………....21

3.2 Pseudo Random Noise Sequence Generation……………………………………………..22

3.3 Digital Modulation Schemes………………………………………………………………26
 3.3.1 Pulse Amplitude Modulation…………………………………………………………26
  3.3.1.1 4- Level Pulse Amplitude Modulation………………………………………...27
  3.3.1.2 8- Level Pulse Amplitude Modulation………………………………………...29
 3.2.2 Phase Shift Keying……………………………………………………………………31
  3.3.2.1 Binary Phase Shift Keying……………………………………………………..31
  3.3.2.2 Quadrature Phase Shift Keying………………………………………………...34

**4 Conclusion and Future Work………………………………………………………….37**

**Appendix………………………………………………………………………………...38**

Signal Generator Menu……………………………………………………………………….38

**References………………………………………………………………………………40**

# Acknowledgments

All that I have achieved during my time as a student at the Communications Laboratory would not have been possible without the support of Prof. Dr. sc. techn. Dirk Dahlhaus, Head of Communications Laboratory, who has always shown resolute faith in me. Respectful mention must also be made of Dipl.-Ing. Herbert Lindenborn for his invaluable guidance whenever I sought an audience with him.

If not for the thoughtful insight and sharp acumen of my project advisor Dipl.-Ing. Thomas Edlich, this project would perhaps not have seen the light of the day; I thank him for bearing with my idiosyncrasies and leading me back to the right path whenever I wandered off-course during my project, but most of all I thank him for taking time out of his busy schedule for me and enlightening me with his invaluable suggestions. Working with him has been a great learning experience. His motivation to work and professionalism has inspired me a lot.

I am grateful to my friends specially Ibrahim and Zemene for their constructive criticism and suggestions. Last but not least, I would like to thank my family for always being there for me through the ups and downs of life.

# 1 Introduction

Mostly sensors generate analog signals in response to various phenomena. Signal processing can be carried out either in analog or digital domain. To do processing of analog signals in digital domain, first digital signal is obtained by sampling and followed by quantization (digitization). The digitization can be obtained by analog to digital converter (ADC).

The role of digital signal processor (DSP) is the manipulation of digital signals so as to extract desired information. In order to interface DSP with analog world, digital to analog converters (DAC) are used. Figure below shows basic components of a DSP system [1].

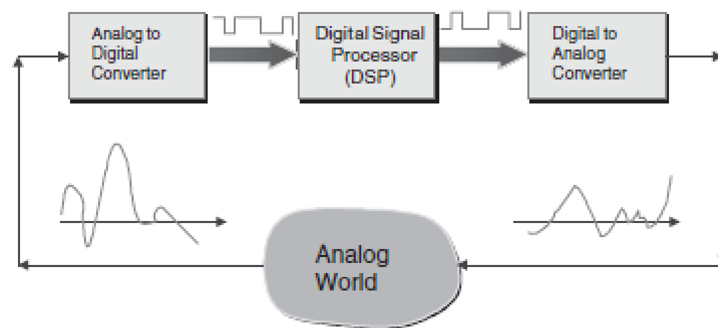


**Figure 1.1: Main components of a DSP system [1]**

ADC captures and inputs the signal. The resulting digital representation of the input signal is processed by DSP such as C6x and then output through DAC. Within in the basic DSP system, anti aliasing filter at input to remove erroneous signals and output filter to smooth the processed data is also used [2].

There are various reasons to process the analog signals in the digital domain:

- The same DSP hardware can be used for various applications by just changing the code.
- Digital circuits are more stable and tolerant than analog circuits.
- Many filters and adaptive systems are realizable only by the digital manipulation of signals.

Digital signal processing can be carried out on various platforms such as customized very large scale integrated (VLSI) circuits and DSP. A comparative review of both the platforms is as follows:

- DSPs are programmable allowing fair amount of application flexibility which not the case with hardwired digital circuits.
- DSPs are cost effective due to mass production and can be used for various applications whereas VLSI chip is normally built for a signal application.
- Often quite high sampling rates can be obtained by customized chips where in DSP sampling rates are limited due to architecture design and peripheral constraints [1].

Large market shares of DSPs belong to cost-effective real time embedded systems such as cell phones and modems. Real time requires keeping processing pace with some external event [2] or in other words completing the processing within the available time between samples which of course depends upon application. Real time processing depends upon two aspects a) sampling rate b) system latencies (delays) [1].

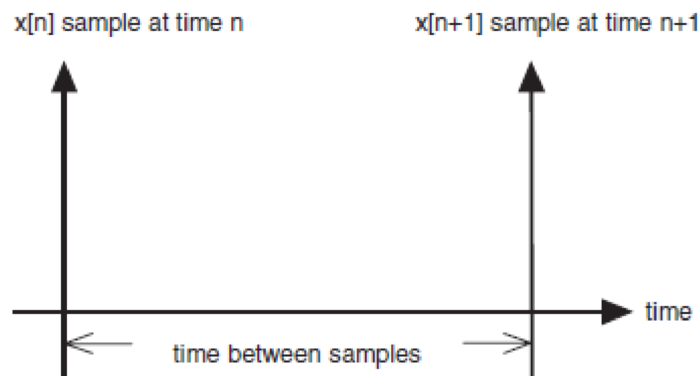


**Figure 1.2: Maximum number of instructions to meet real time =**

**Time between samples/ Instruction cycle time [1]**

## 1.1 Hardware and Software Tools

Many Companies produce DSPs such as Motorola, NEC, SGS-Thompson, Conexant, Lucent Technologies and Texas Instruments (TI). In this project, DSP TMS320C6713 manufactured and designed by TI is used. DSPs such as TMS320C6x are special purpose microprocessors by TI with specialized architecture which is well suited for numerical intensive calculations. DSPs offer wide range of applications from image processing to communications such as cellular phones, printers, digital cameras MP3 players and so on [2][8].

The TMS320C6713 DSP is very powerful by itself, but for development of programs, a supporting architecture is required to store programs and data and to bring signals on and off the board. In order to use DSP a circuit board is provided that contains appropriate components. Together, Code Composer Studio (CCS), DSP chip, and supporting hardware make up the DSP starter kit (DSK) [8]. Figure below shows the C6713 DSK board.

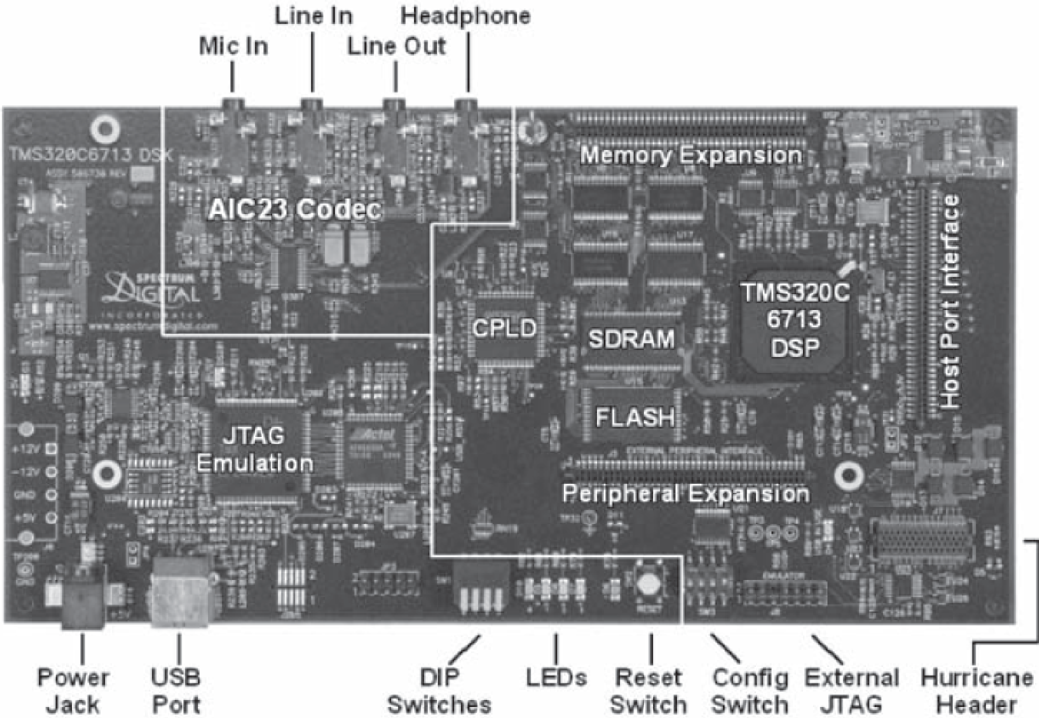


**Figure 1.3: C6713 DSK Board [1]**

The 6713 DSK board includes following hardware:

- C6713 DSP operating at 225 MHz
- 4 Kbytes memory for L1D data cache
- 4 Kbytes memory for L1P program cache
- 256 Kbytes memory for L2 memory
- 8 Mbytes of onboard SDRAM (Synchronous Dynamic RAM)
- 512 Kbytes of flash memory
- 16-bit stereo codec AIC23 with sampling frequency of 8 KHz to 96 KHz

CCS software tool is used to generate TMS320C6x executable files. CCS includes the assembler, linker, compiler, and simulator and debugger utilities. Figure 1.4 shows the intermediate steps involved for going from a source file to an executable file. In the absence of target board the simulator can be used to verify the code functionality, however in the absence of simulator, Interrupt Service Routine (ISR) cannot to be used to read signals samples from a signal source. To be able to process signal in real time DSK or an Evaluation Module (EVM) is required for code development. Other testing equipments include function generator, oscilloscope, microphone and cable with audio jacks. A DSK board can be connected through PC host through parallel or USB port. Two standard audio jacks are used for signal interfacing with DSK board [1] [3].

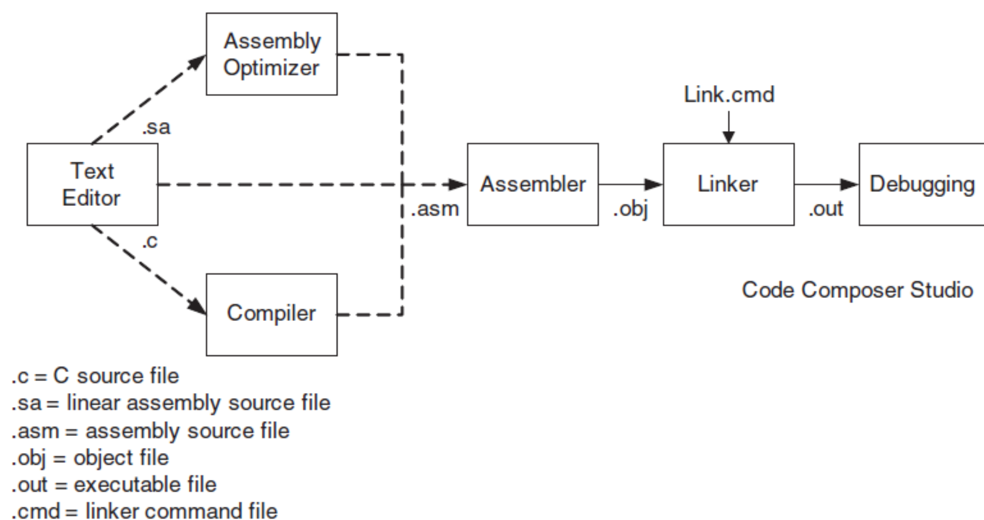


**Figure 1.4: C6x CCS Software Tool [1]**

All the necessary files and steps required to build a project on code composer studio are explained in detail in [4]. Assuming CCS is installed in C drive, below is a summarized listing of steps and files required to make a project:

- Open CCS.
- To create new project go to Project→ New. Remember to select target TMS320C67XX.
- Go to Debug and select Connect. Now the target board is connected to CCS.
- Add source file (.c) to the project.
- From the support folder (included in the CD) add the files *c6713dskinit.c*, *C6713dsk.cmd* and *vectors_poll.asm* to the project. It is important to note that if program is interrupt based then *vectors_poll.asm* should be replaced by *vectors_intr.asm.*
- Add run-time support library file *rts6700.lib*, board support library file *dsk6713bsl.lib* and chip support library file *csl6713.lib* to the project. These files can be found in C:\C6713\C6000 under subdirectories cgtools\lib, dsk6713\lib and csl\lib respectively.
- Go to *Project→Build* options. Select *Compiler* tab and go to *Basic* in *Category* listing. Change the *Target Version* to *C671x(-mv6710)* now select *Advanced* in the *Category* listing and change the *Memory Models* to *--mem_model:data=far* and in the last go to *Preprocessor* in the *Category* listing and change the *Pre-Define Symbol* and *Include Search Path* to *CHIP_6713* and *C:\C6713\C6000\dsk6713\include* respectively.
- Go to Project → Rebuild All.
- File → Load Program.
- Debug → Run

## 1.2 Signal Generator

The signal generator is referred to a wide range of devices used to test electronic equipments by generating a signal whose frequency, wave shape, and amplitude are independently adjustable over a wide range of settings [9].

Signal generator come in many different forms. The most prevalent signal generator types include function generator and arbitrary waveform generator. The signal generator types vary in their feature and functionality (and at varying level of expense) and are suitable for many different applications; in general, no single device is suitable for all possible applications [10].

### 1.2.1 Function Generator

Function generators create built-in waveforms, such as sine, square or triangle waves, at adjustable frequencies and amplitude. Some function generators can also generate white or pink noise.

Function generators can be either analog or digital based. Analog based function generators use electrical circuit which is combination of capacitor, resistor, inductors and other hardware to create simple functions and often they are used when a static sine or square wave at a specified frequency is required in a certain application.

Digital based function generators use combination of DSP, DAC and memory buffer to dynamically create signals. Many signal generators create signals by dividing an internal timebase by an integer factor. This is called the divide-by-N method. Divide-by-N clocking, however, gives a limited set of frequencies. Figure 1.5 shows a basic architecture of digital based function generators.

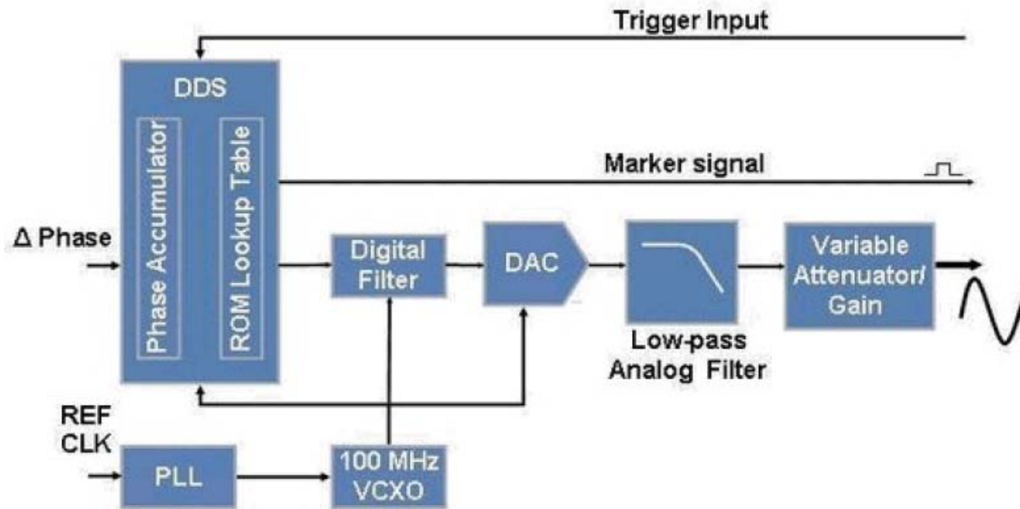


**Figure 1.5: Digital based Function Generator [10]**

Typically one complete cycle of the waveform which is to be generated is stored in the memory lookup table. The phase accumulator keeps track of the current phase of the output function. To output a very low frequency, the phase difference between the samples should be small. For example, a low frequency sine may have a phase difference of 1 degree such that sample 0 of the waveform would be the amplitude of the sine wave at 0 degrees; sample 1 of the waveform would be the amplitude of the sine wave at 1 degree, and so on. All 360degrees of the sinusoid, or exactly one cycle, would be output after 360 samples. A higher frequency sine wave may have a phase difference of 10 degrees or more. For phase difference between samples equal to 10 degree, one cycle of a sine wave would be generated in 36 samples. With a fixed sampling frequency, the low frequency sine wave would be 10 times lower in frequency than the high frequency sine wave [10].

### 1.2.2 Arbitrary Waveform Generators (AWG)

Arbitrary Waveform Generators (AWG), are sophisticated signal generators which allow the user to generate arbitrary waveforms, within certain limits of frequency range, accuracy, and output level.

Unlike function generators, which are limited to a simple set of waveforms; AWGs are much more flexible. AWGs are generally more expensive than function generators, and are often more highly limited in available bandwidth; as a result, they are generally limited to higher end design and test applications [11].

**Figure 1.6: Arbitrary Waveform Generator [12]**

AWGs can also operate as conventional function generators. Mostly they include standard waveforms such as sine, square, ramp, triangle, and noise. Some units include additional built in waveforms such as sinx/x, cardiac waveform while other may display a graph of the waveform on their screen. Some AWGs allow multiple channels to be operated with precisely controlled phase offsets or ratio-related frequencies. This allows the generation of polyphase sine waves and IQ constellations. Complex channel modulations are also possible [12].

### 1.2.3 Applied Fields for Signal Generator

The generation of signals is an important development in the troubleshooting and development of electronic design. The signal generator is used to provide known test conditions for the performance evaluation of electronic system design and for replacing signals that are missing in system during repairing work. While signal generators are widely used to test and maintain a wide range of RF equipment, such as radio receivers and transmitters, they are also extremely useful for testing digital, clock driven systems, especially high speed serial storage and serial communication ICs, circuit boards and devices. In general signal generators are used in

designing, testing, troubleshooting, and repairing electronic devices specially related to communication. Few of the examples in which signal generator can be used are:

- Testing setup for ICs and circuit boards for digital or differential output plus frequency sweep
- Testing the IF and RF sections of receivers and mobile communication bands
- Testing amplifiers for gain and for the 1 dB compression point
- Using as a programmable clock generator
- Calibrate receivers
- Low frequency filter testing (down to 1 Hz)
- Local oscillator source
- Radio frequency (RF) Exciter

Audio signal generators produce signals in a range from a few Hz up to several kHz. Signals can be injected into audio amplifiers to see how they behave at various audio frequencies. Amplification and frequency response can be measured and distortion of the signal can be observed. RF generators can provide frequencies from about 100 kHz up to several hundred MHz. With radio frequency generators it is usually possible to modulate the RF with an audio signal to simulate a radio station. Amplitude and frequency modulation are available. Using an RF generator the various tuned circuits in a radio can be adjusted for peak performance [13] [14].

## 1.3 Project Motivation

Due to various uses of signal generator in different fields of application, there is always a need of creating a customized signal generator which could suffice the needs in different scenarios. The basic task of this project is to create C6713 DSK implementation of specialized signal generator which could mimic the functionality of various signal generators.

# 2 Concept

The task at hand is to program a signal generator using C6713 DSK using its audio interfaces which could have various functionalities ranging from generating different waveforms to various digital modulation schemes. As compared to market available signal generators, implementation of signal generator on C6713 DSK gives the user an added advantage that the functionalities of signal generator are not limited to predefined waveforms/options but different new features can be added in already present signal generator as desired by modifying the source code.

Different features of the concept signal generator can be grouped as:

- Waveform Generation
- Pseudo Random Noise Sequence Generation
- Digital Modulation Schemes

A user can select any of the above stated categories by using the slider named “Mode” which is implemented in CCS using General Extension Language (GEL). GEL is an interpretive language similar to C which allows changing variable such as gain, sliding through different values while processor is still running [2].

Now different options available to a user in each category are discussed below:

## 2.1 Waveform Generation

A user can select different waveforms such as sine wave, square wave, triangular wave, ramp (saw tooth) or multi tone signal using the “Mode” slider. Gain and frequency of sine wave, square wave, triangular wave and sawtooth wave can be altered using “Gain” and “Frequency” slider.

In multi tone signal there are 4 sliders available to a user to change different attributes of multi tone signal. It is important to note that throughout the project where ever multi tone signal is referred, it is assumed to be consisting of only two tones.

A “Gain” slider defines the overall gain of multi tone signal. The gain selected by “Gain” slider is distributed among two frequencies. These two frequencies can be selected independent of each

other using slider “f1” and “f2”. Slider “Ratio_gain_f1_VS_f2” decides the ratio by which the overall gain is distributed among the two frequencies (f1 and f2) selected by slider “f1” and “f2”.

## 2.2 Pseudo Random Noise Sequence Generation

In this category a user can generate pseudo random noise sequence or can investigate the effect of noise on an external input signal. In both of these cases the “Gain” slider can be used to select appropriate gain of the noise as desired.

## 2.3 Digital Modulation Schemes

Different modulation schemes can be selected by the user using “mode” slider such as Pulse Amplitude Modulation (PAM) or Phase Shift Keying (PSK). The user has also the freedom to choose the modulation schemes, either to work either using Input signal from some external source or using predefined input signal for demonstration. Different variants of PAM and PSK modulations implemented in the concept signal generator are 4-level PAM, 8-level PAM, BPSK (Binary PSK) and QPSK (Quadrature PSK).

# 3 Implementation

In this section implementation of signal generator on C6713 DSK is discussed. For the ease of reader different functionalities of signal generator are discussed as independent modules where later all these modules are incorporated in a single program. To access the source code of this program, please refer to the file "sine_varFreq.c" which is present in the folder "sine_varFreq", included in the attached CD-ROM. It is assumed that the reader has a basic knowledge of C/C++ and has read [4].

## 3.1 Waveform Generation

The C6713 DSK can be used to generate various types of waveforms which have wide range of applications in communication systems. In this section generation of periodic waveforms namely sine wave, square wave, triangular wave, sawtooth wave and multi tone signal generation is discussed.

### 3.1.1 Sine Wave Generation

One of the main things to take in account in generating any waveform is writing the sample values of waveform on the codec at every sampling instant. In the DAC done at the codec these sample values of the waveform are first oversampled (to smoothen the waveform) and are then fed to a reconstruction filter (to cut frequencies out of band of interest).

There are two techniques for sending the data to codec namely polling technique and interrupt technique. Polling technique uses a continuous procedure of testing when the data is ready. Although it is simpler technique than interrupt technique, it is less efficient since the input and output data needs to be continuously tested to determine when they are ready to be received or transmitted. Figure 3.1 shows sine wave generated using interrupts technique, source code is shown in figure 3.2.

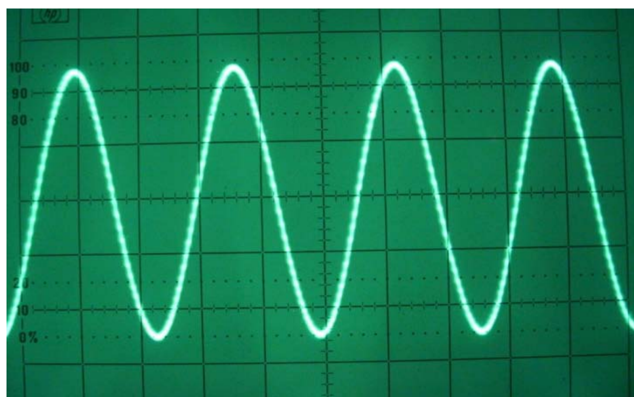

**Figure 3.1: Sine Wave obtained with the Scope**

Line 3 in the code sets the sampling frequency to 96 KHz. Different sampling frequencies supported by AIC23 codec are 8, 16, 24, 32, 44.1, 48, and 96 kHz. Line 7 initializes a array *sine_table* of length 128 which will be later used as a buffer to store different sine values. Within the *main* function, line 19-21 generates the data for sine wave. When the program execution reaches the line 23, program execution halts and it starts listening for the interrupt which occur at every sampling period $T_s$. At each Interrupt, program execution goes to the interrupt service routine defined in the lines 9-14. Within the service routine, the value in the buffer *sine_table* indexed by the variable *loop* is written on to the codec. Then the index *loop* is incremented by an amount equal to *frequency*. After reaching the line 13, program execution goes back to line 23 and then again starts listening for next interrupt and this process goes on. Variables *gain* and *frequency* defined in line 5 and 6 respectively set the gain and frequency of the generated sine wave.

```
1  include "dsk6713_aic23.h"                    //support file for codec, DSK
2  #define sineTableLen 128
3  Uint32 fs=DSK6713_AIC23_FREQ_96KHZ;         //set sampling rate
4  short loop = 0;                             //table index
5  short gain =1;
6  short frequency = 1;
7  short sine_table[sineTableLen]={0};             // table initialization
8  interrupt void c_int11()                    //interrupt service routine
9  {
10 output_sample(sine_table[loop]*gain);           //output sine values
11 loop += frequency;                        //increment frequency index
12 loop = loop%sineTableLen;         //reinitialize table if exceeds limit
13 return;                                   //return from interrupt
14 }                                         // end of interrpt routine
15 void main()
   {
16 short i=0;
17 float pi=3.14159;
18 for(i = 0; i < sineTableLen; i++)
19 {
20 sine_table[i] = 300*sin(2.0*pi*i/sineTableLen);
21 }
22 comm_intr();                              //init DSK, codec, McBSP
23 while(1);                                 //infinite loop
   }
```

**Figure 3.2: Source code for sine wave generation**

These variables (gain and frequency) are controlled by the user during the program execution on DSP with the help of graphical user interface named "slider" as shown in the figure 3.3.

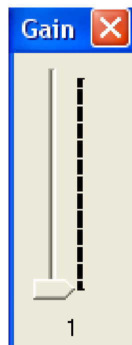


**Figure 3.3: Slider representing Amplitude of sine wave**

In the project we have set the Gain slider to take any discrete value from 1 to 100 determined by the user, correspondingly the *gain* variable can also take same range of values. In line 10 this *gain* variable is multiplied by different sine values to create a output sine wave of the same gain factor. Similarly we have set Frequency slider to take any discrete value from 1 to 20 determined

by the user, correspondingly the *frequency* variable can also take same range of values. Subsequently for 20 different values taken by the frequency slider, we can have 20 different frequencies ranging from 750 Hz (approx) to 15 kHz (approx) with the step size of 750 Hz (approx).

### 3.1.2 Square Wave Generation

In this section square wave generation is discussed. In the line 7, figure 3.5, data buffer for containing square wave values is declared. In the main function, first half of the buffer *square_table* is filled up with value 300 while the second half of the buffer is filled up with value -300.

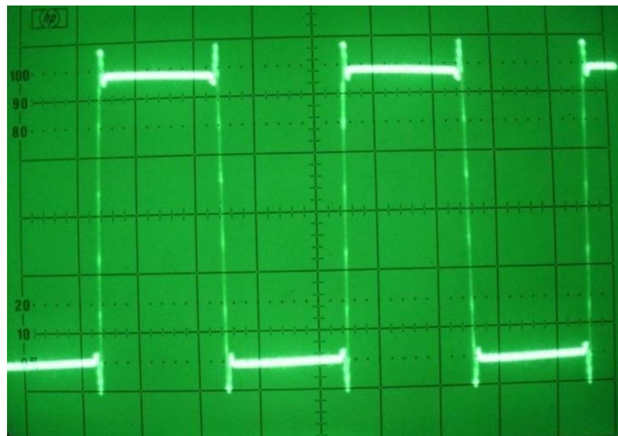


**Figure 3.4: Square Wave obtained with the Scope**

The maximum output signal which can be obtained with AIC23 codec present in 6713 DSK is $6V_{p\text{-}p}$. Correspondingly the maximum value which can be fed to the codec to produce distortion free output signal ranges from $-2^{15}$ to $2^{15}-1$ or between -32,768 and 32,767. In the lines 20 and 24 square wave is initialized with 300 and -300 such that when square wave values are multiplied with the *gain* and sent to the output (line 10), for the maximum value of gain i.e. 100, the product of *gain* and square wave values lies with the allowed input range of the codec.

```
1   include "dsk6713_aic23.h"                  //support file for codec, DSK
2   #define squareTableLen 800
3   Uint32 fs=DSK6713_AIC23_FREQ_96KHZ;        //set sampling rate
4   short loop = 0;                            //table index
5   short gain =1;
6   short frequency = 1;
7   short square_table[squareTableLen]={0};        // table initialization
8   interrupt void c_int11()                   //interrupt service routine
9   {
10  output_sample(square_table[loop]*gain);        //output square values
11  loop += frequency;                       //increment frequency index
12  loop = loop%squareTableLen;      //reinitialize table if exceeds limit
13  return;                                  //return from interrupt
14  }                                        // end of interrpt routine

15  void main()
16  {
17  short i=0;
18  for(i=0; i < squareTableLen/2; i++)
19     {
20       square_table[i] = 300;  // lookup table first half of waveform
21     }
22  for(i=squareTableLen/2; i <  squareTableLen; i++)
23     {
24      square_table[i] = -300;  // lookup table 2nd half of waveform
25     }
26  comm_intr();                             //init DSK, codec, McBSP
27  while(1);                                //infinite loop
```

**Figure 3.5: Source code for square wave generation**

### 3.1.3 Triangular Wave Generation

Now we will discuss generation of triangular waves.

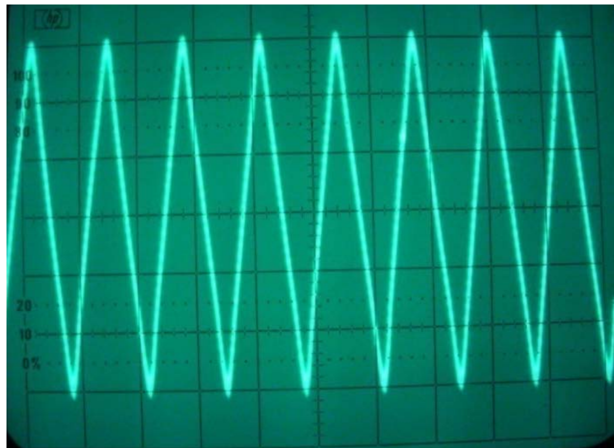

**Figure 3.6: Triangular Wave obtained with the Scope**

In the main function, buffer of values of triangular wave is defined in such a way that first entry in the buffer is given the value -300 (-0x12C) and as the index of the buffer increases the values in the buffer linearly decrease with the *step* defined in the line 18. When the index of the buffer reach in the middle of the buffer, then the values in the buffer again starts to increase linearly with the *step* size till it reaches the end of the buffer.

```
1  include "dsk6713_aic23.h"              //support file for codec, DSK
2  #define triangleTableLen 128
3  Uint32 fs=DSK6713_AIC23_FREQ_96KHZ;       //set sampling rate
4  short loop = 0;                              //table index
5  short gain =1;
6  short frequency = 1;
7  short triangle_table[triangleTableLen]={0};       // table initialization

8  interrupt void c_int11()                  //interrupt service routine
9  {
10 output_sample(triangle_table[loop]*gain);         //output triangle values
11 loop += frequency;                                //increment frequency
   index
12 loop = loop%triangleTableLen;              //reinitialize if exceeds limit
13 return;                                    //return from interrupt
14 }                                     // end of interrpt routine

15 void main()
16 {
17 short i=0;
18 short step = 2 * 0x12C/ (triangleTableLen);
19 for(i=0; i < triangleTableLen/2; i++)
20    {
21      triangle_table[i] = -0x12C + i * step;  // First half of waveform
22    }
23 for(i=triangleTableLen/2; i <  triangleTableLen; i++)
24    {
25     triangle_table[i] = triangle_table[i-1] - step; // second half of
   waveform
26    }
27 comm_intr();                               //init DSK, codec, McBSP
```

**Figure 3.7: Source code for triangular wave generation**

Another feature which can be easily incorporated with our current signal generator model is to control (ON/OFF) the generation of output (waveform/Noise/Digital Modulation schemes) with the help of DIP switch. As an example, figure 3.8 shows the sample code for controlling the generation of triangular wave with the help of DIP switch.

In the *main* function, line 34 initializes the DSK, the AIC23 onboard DSK codec and two Multi channel Buffered Serial Ports (McBSP) on the C6713 processor. The function *comm_intr* is present in communication and initialization support file *c6713dskinit.c*.Within *c6713dskinit.c*, the function DSK6713_init initializes the BSL file, which must be called before the two subsequent BSL functions DSK6713_LED_init and DSK6713_DIP_init (line 35 and 36), are invoked that initialize the four LEDs and the four DIP switches. In the line 11, the *if* statement becomes true when dip switch #0 is pressed and consequently LED #0 turns on (line 13) and waveform is generated (line 14).

```
1  include "dsk6713_aic23.h"              //support file for codec, DSK
2  #define triangleTableLen 128
3  Uint32 fs=DSK6713_AIC23_FREQ_96KHZ;       //set sampling rate
4  short loop = 0;                           //table index
5  short gain =1;
6  short frequency = 1;
7  short triangle_table[triangleTableLen]={0}; // table initialization
8
9  interrupt void c_int11()               //interrupt service routine
10 {
11 if(DSK6713_DIP_get(0)==0) //=0 if switch #0 pressed
12  {
13    DSK6713_LED_on(0); //turn LED #0 ON
14    output_sample(triangle_table[loop]*gain);//output triangle values
15    loop += frequency;                    //increment frequency index
16    loop = loop%triangleTableLen;       //reinitialize if exceeds limit
17  }
18 else DSK6713_LED_off(0); //LED #0 off
19 return;                            //return from interrupt
20 }
21
22 void main()
23 {
24 short i=0;
25 short step = 2 * 0x12C/ (triangleTableLen);
26 for(i=0; i < triangleTableLen/2; i++)
27    {
28      triangle_table[i] = -0x12C + i * step; // First half of waveform
29    }
30 for(i=triangleTableLen/2; i <  triangleTableLen; i++)
31    {
32      triangle_table[i] = triangle_table[i-1] - step; // second half
33    }
34 comm_intr();                          //init DSK, codec, McBSP
35 DSK6713_LED_init();                         //init LED from BSL
36 DSK6713_DIP_init();                     //init DIP from BSL
37 while(1);                                 //infinite loop
38 } //end of main
```

**Figure 3.8: Source code for controlling triangular wave through DIP switch**

### 3.1.4 Sawtooth Wave (Ramp) Generation

Now we will discuss the generation of sawtooth wave. Sawtooth wave is also sometimes referred as ramp.

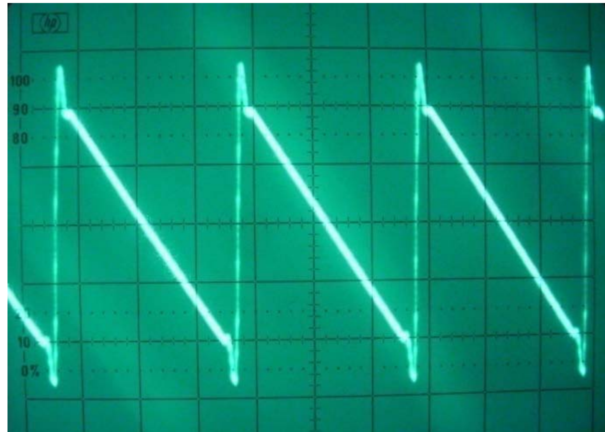

**Figure 3.9: Sawtooth Wave obtained with the Scope**

Ramp is created (Figure 3.10) in approximately the same way as the triangular wave with only the difference that after buffer index has reached the middle of buffer (line 22) unlike, triangular wave, values in the buffer *ramp_table* continue to decrease linearly with the *step* defined in line 19 until it reaches the end of the buffer. Here, it is important to note that at the output we get –ve sloped ramp due to the 2's- complement format of the AIC23 codec [2].

Another functionality which we can easily include in our current signal generator is to output ramp with positive and negative slope on left and right channel at the same time. Figure 3.11 shows the corresponding program listing. This program is same as in figure 3.10 except some changes in the ISR, so only ISR is shown in the figure 3.11. In line 6, a structure *AIC23_data* is declared which could hold data of two channels. This program uses the same data buffer *ramp_table* to create both +ve sloped and –ve sloped ramp. For the –ve sloped ramp at the output (left channel), the *loop* index starts from the start of *ramp_table* while for +ve sloped ramp at the output (right channel), the *loop* index starts from the end of *ramp_table* and then gradually decrease with the step defined by the *frequency*.

```
1  include "dsk6713_aic23.h"              //support file for codec, DSK
2  #define rampTableLen 128
3  Uint32 fs=DSK6713_AIC23_FREQ_96KHZ;       //set sampling rate
4  short loop = 0;                                 //table index
5  short gain =1;
6  short frequency = 1;
7  short ramp_table[rampTableLen]={0};  // table initialization
8  interrupt void c_int11()                  //interrupt service routine
9  {
10 output_sample(ramp_table[loop]*gain);        //output ramp values
11 loop += frequency;                     //increment frequency index
12 loop = loop%rampTableLen;       //reinitialize if exceeds beyond limit
13 return;
14 }                                    // end of interrpt routine

15 void main()
16 {
17 short i=0;
18 short step2 = 2*0x12C/ (rampTableLen);
19 for(i=0; i < rampTableLen; i++)        // Fill the lookup table
20 {
21 ramp_table[i] = -0x12C + i * step2;
22 }
23 comm_intr();                                //init DSK, codec, McBSP
24 while(1);                               //infinite loop
   }
```

**Figure 3.10: Source code for sawtooth wave generation**

```
1  short loop1 = 0;                                //table index
2  short loop2 = rampTableLen-1;                          //table index
3  short gain =1;
4  short frequency = 1;
5  short ramp_table[rampTableLen]={0};     // table initialization
6  union {Uint32 combo; short channel[2];} AIC23_data;
7
8  interrupt void c_int11()                  //interrupt service routine
9  {
10 AIC23_data.channel[LEFT]=ramp_table[loop1]*gain;      //output ramp values
11 loop1 += frequency;                  //increment frequency index
12 loop1 = loop1%rampTableLen;       //reinitialize if exceeds beyond limit
13
14 AIC23_data.channel[RIGHT]=ramp_table[loop2]*gain;     //output ramp values
15 loop2 -= frequency;                  //increment frequency index
16 if (loop2<0)
17 loop2= rampTableLen-1;
18
19 output_sample(AIC23_data.combo); //output to both channels
20 return;
21 }                                    // end of interrpt routine
```

**Figure 3.11: Code for generating +ve and –ve slopped ramp on right and left channel**

### 3.1.5 Multi-tone Signal Generation

Multi-tone signal generation uses the same method to generate the buffer for storing the values of the sine wave as depicted in the figure 3.2 with only the difference in the ISR. Figure 3.13 show the interrupt service routine for generating Multi-tone signal.

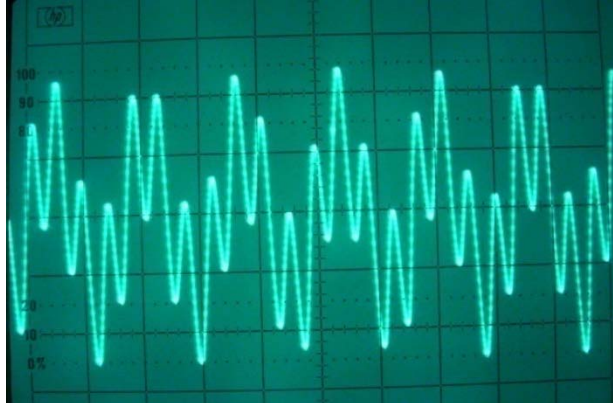

**Figure 3.12: Multi-tone wave obtained with the Scope**

In the line 12 two sinusoidal signals are generated and added together to form a multi-tone signal. The frequencies of these signals are determined by the variable *f1* and *f2* which are controlled through the slider. Variable *gain* defines the overall gain of the multi-tone signal which is distributed among the two sinusoids by the proportion determined by the variable *ratio*. The *ratio* is also user adjustable through the slider created with the help of GEL.

```
1   include "dsk6713_aic23.h"                 //support file for codec, DSK
2   #define sineTableLen 128
3   Uint32 fs=DSK6713_AIC23_FREQ_96KHZ;       //set sampling rate
4   short gain = 1;
5   short loop1 = 0;
6   short loop2 = 0;
7   short f1=1;                // frequency of multi tone signal f1
8   short f2=1;                // frequency of multi tone signal f2
9   short ratio=0;             // ratio of gain between f1 and f2
10  short sine_table[sineTableLen]={0};            // table initialization
11  interrupt void c_int11() {                 //interrupt service routine
12  output_sample(( sine_table[loop1]*ratio + sine_table[loop2]*(10-
    ratio))*(gain/10));
13  loop1 += f1;                   //increment frequency index
14  loop1 = loop1%sineTableLen;    //reinitialize if exceeds limit
15  loop2 += f2;                   //increment frequency index
16  loop2 = loop2%sineTableLen;    //reinitialize if exceeds limit return;
```

**Figure 3.13: Code for generating +ve and –ve slopped ramp on right and left channel**

## 3.2 Pseudorandom Noise Sequence Generation

Random noise is important in digital signal processing. For example, it limits how small of a signal an instrument can measure, the distance a radio system can communicate. A common task in signal processing is to generate signals that resemble various types of random noise. This is required to test the algorithms which should work in the presence of noise [5].

In this section, pseudorandom noise sequence generation is implemented using maximal length sequence (MLS) technique.MLS are also called m-sequence. MLS generator can be made from shift registers with proper feedback. If the shift generator has $m$ bits the length of generated sequence is $N=2^m-1$. MLS has several properties that make them good approximation to ideal binary random sequences when $N$ is large.

**Frequency of Occurrence of 1's and 0's**

The number of 1's in one period of MLS is $2^{m-1}$ and the number of 0's is $2^{m-1}-1$. Thus period contains one more 1 than 0. For large N, 1's and 0's appear with equal likelihood.

**Frequency of Runs of 1's and 0's**

A run of k 1's is defined to be a string staring with zero, followed by k 1's and ending with a zero. In one period of MLS, there is one run of m 1's. There is no run of m-1 1's.

**Correlation property**

For MLS, $N=2^m-1$ the periodic autocorrelation function is

$$R(n) = \begin{cases} \frac{-1}{N} & \text{for } n \text{ not a nultiple of } N \\ 1 & \text{for } n \text{ is a mutiple of } N \end{cases}$$

For ideal binary random sequence the autocorrelation function is 1 for n = 0 and 0 otherwise [6], [7].

Figure 3.14 shows the source code for generating pseudo random noise sequence. Initially 16 bits so called a seed is assigned to a register. The header file *noise_gen.h* (included in the CD) has initial shift register bits definition. Any 16 bit sequence other than all zero sequence can be used as a seed. XOR operation of some bits in the shift register say bit b0, b1, b11and b13 is taken and then outcome is placed in a feedback variable i.e. fb (line 14 and 15) as shown in figure 3.15. The register with initial seed is then shifted to the left by 1 bit (line 16). The value in the feedback variable is then assigned to b0 of the register (line 17). Depending upon the whether the register's bit b0 is 0 or 1; a scaled value is assigned to *prnseq* (line 10 to 13). This value corresponds to noise level amplitude [2].

```
1   #include "DSK6713_AIC23.h"
2   Uint32 fs=DSK6713_AIC23_FREQ_96KHZ;
3   #include "noise_gen.h" //header file for noise sequence
4   short fb;
5   shift_reg sreg; //shift register structure
6   short gain =1;
7   interrupt void c_int11()
8   {
9   short prnseq; //for pseudo-random sequence
10  if(sreg.bt.b0) //Ouput sequence based on bit b0
11  prnseq = -300; //scaled negative noise level
12  else
13  prnseq = 300; //scaled positive noise level
14  fb =(sreg.bt.b0)^(sreg.bt.b1); //XOR bits 0,1
15  fb ^=(sreg.bt.b11)^(sreg.bt.b13);//with bits 11,13 ->fb
16  sreg.regval<<=1; //shift register 1 bit to left
17  sreg.bt.b0 = fb;
18  output_sample(prnseq*gain); //output scaled sequence
19  return;
20  }
21  void main()
22  {
23  sreg.regval = 0xFFFF;
24  fb = 1; //initial feedback value
25  comm_intr(); //init DSK, codec, McBSP
26  while (1); //infinite loop
27  }
```

**Figure 3.14: Pseudorandom Noise Sequence Generation [2]**

In the line 18 *prnseq* is multiplied by the *gain* which is controlled by the user through the slider and governs the noise level amplitude at the output. The noise can be viewed in time domain on oscilloscope or heard on microphone. The spectrum of the generated noise sequence is band limited by the bandwidth of the anti-aliasing filter present in the codec AIC23.

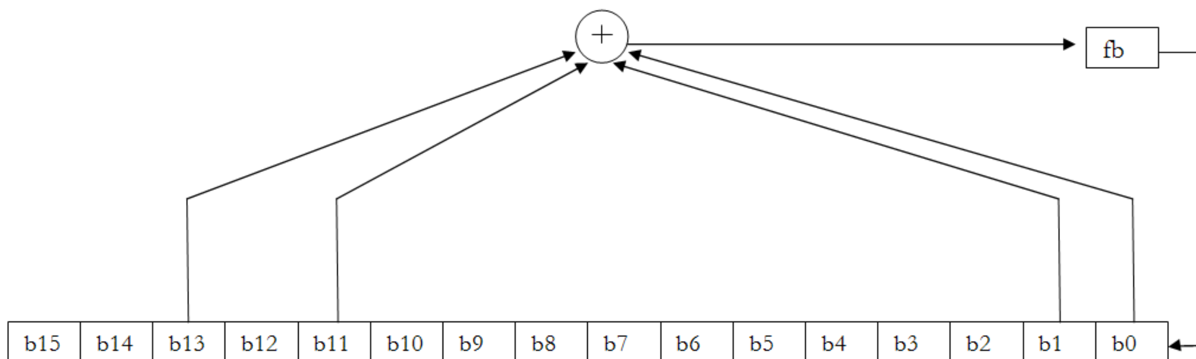


**Figure 3.15: Shift register architecture with feedback for noise sequence generation**

If we want to analyze the effect of superimposed noise on some external input signal, it can accomplished by the altering the ISR (line 13 and14) as shown in figure 3.16.

```
1.  interrupt void c_int11() //interrupt service routine
2.  {
3.  short prnseq; //for pseudo-random sequence
4.  sample_data = input_sample(); //input sample
5.  if(sreg.bt.b0) //sequence{1,-1}based on bit b0
6.  prnseq = -300; //scaled negative noise level
7.  else
8.  prnseq = 300; //scaled positive noise level
9.  fb =(sreg.bt.b0)^(sreg.bt.b1); //XOR bits 0,1
10. fb ^=(sreg.bt.b11)^(sreg.bt.b13);//with bits 11,13 ->fb
11. sreg.regval<<=1; //shift register 1 bit to left
12. sreg.bt.b0 = fb; //close feedback path
13. sample_data=sample_data+prnseq*gain;
14. output_sample(sample_data);    //output sample
15. return; //return from interrupt
16. }
```

**Figure 3.16: Pseudorandom Noise Sequence Generation**

Figure 3.17 shows pseudorandom noise as displayed on the oscilloscope and figure 3.18 shows the effect of superimposed noise on the external input sinusoidal signal at two different levels of noise *gain*.

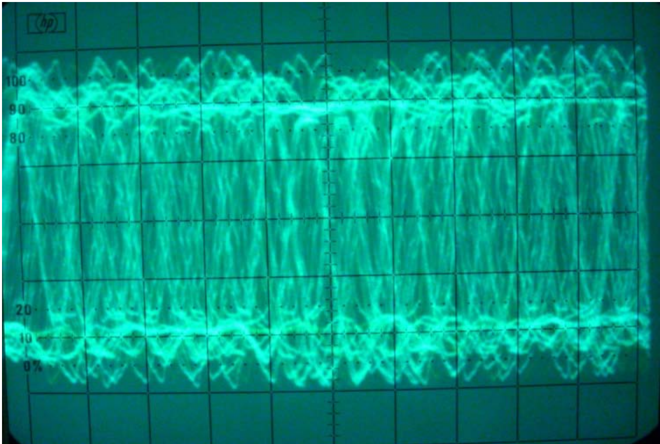


**Figure 3.17: Pseudorandom Noise Sequence obtained with the Scope**

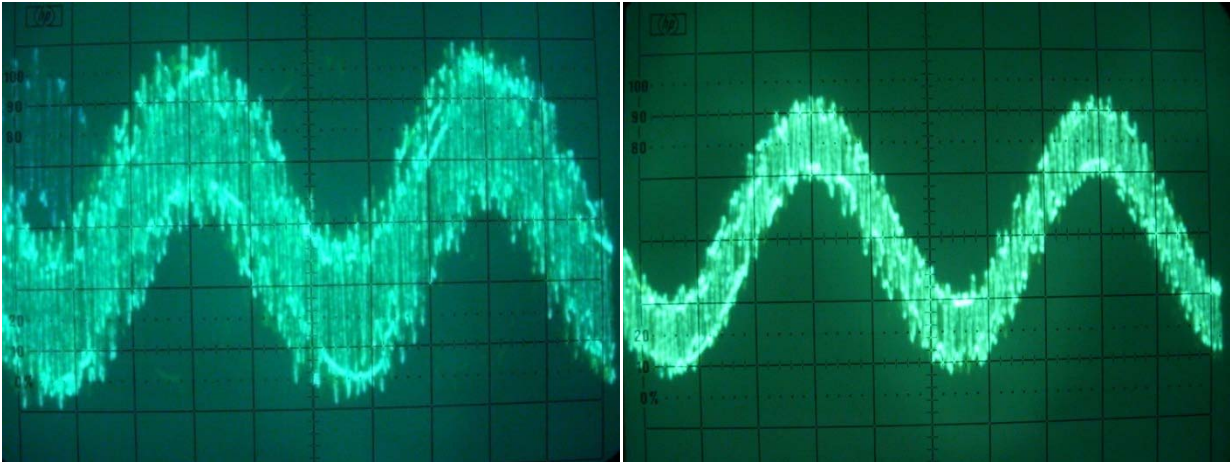

**Figure 3.18: Pseudorandom Noise superimposed on sinusoidal signal at two different noise levels**

# 3.3 Digital modulation Schemes

In this section digital modulation schemes namely PAM and PSK are discussed. Please refer to [2] for reference for this section.

## 3.3.1 Pulse Amplitude Modulation

PAM uses the amplitude of the pulse to convey the information while other parameters such frequency remains fixed. The incoming bit stream is grouped into J-bit words such that $2^J$ levels are uniquely assigned to them. With increasing J, the number of possible levels also increases. For example with J=2, there are 4 levels and with J=3, there are eight levels possible. These levels when mapped on the constellation diagram are equidistance from each other and centered across the zero. For J=3, eight constellation points representing levels are shown in the figure below:

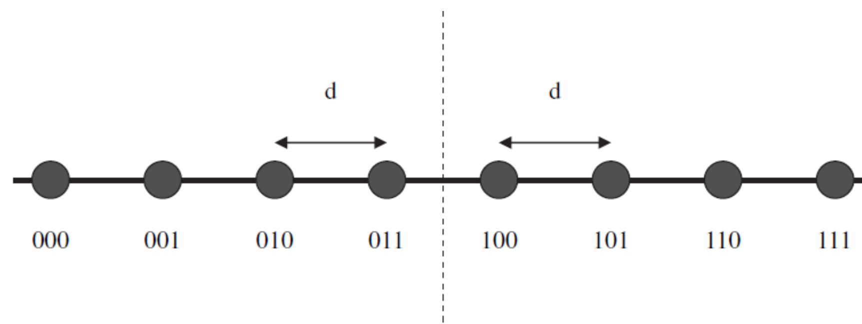


**Figure 3.19: 8-level PAM constellation diagram [2]**

These levels are then mapped into train of pulses such that amplitude of these pulses represent one-to-one mapping of information symbols to the respective levels of pulse. At the receiver the information sequence is retrieved back by mapping the pulse amplitude to the information symbol. Figure 3.20 shows the block diagram of the PAM system [2].

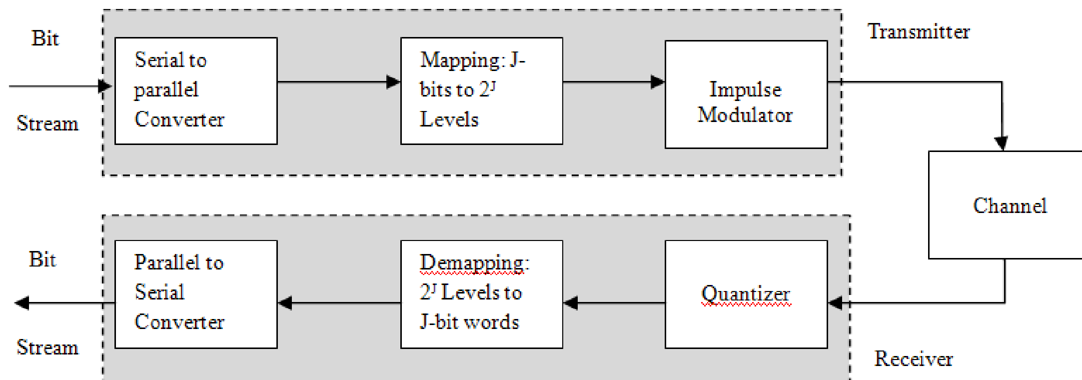


**Figure 3.20: A simple PAM system block diagram**

### 3.3.1.1 4-Level Pulse Amplitude Modulation

In the *main* function (figure 3.22) line 31 to 36 are used to generate 4-level PAM lookup table for mapping as shown in figure 3.21.

| Symbol Block | Level (in hex) |
|---|---|
| 00 | 0x7FFF |
| 01 | 0x2AAA |
| 10 | -0x2AAB |
| 11 | -0x8000 |

**Figure 3.21: 4-level PAM lookup table for mapping**

At the input we have 16 bit long input sample (line 12). For 4-level PAM, the input sample is decomposed into segments 2 bits long. So now each input sample is composed of 8 segments. Parsing the input sample is achieved through the use of masking and shifting. The first symbol block is obtained with masking of two least significant bits by anding the input sample with 0x0003 (line 16). Now this symbol block is mapped on one of the 4 uniformly spaced levels between –x8000 and 0x7FFF (line 17) using the lookup table created in the *main* function. The selected level is then transmitted as a square wave. The period of the square wave is achieved by outputting the same level many times to ensure a smooth-looking square wave at the output of the DSK (line 21).

The second symbol block is obtained through shifting the original input sample by two bits to the right (line 24) and masking the 2 least significant bits (LSBs). These steps are repeated until the end of input sample length and produce eight symbol blocks. As each input sample is decomposed into 8 symbol block and corresponding level of each symbol block is output 12 times so a new input sample is taken after every 96 samples (line 12) [2].

```
1   include "dsk6713_aic23.h"                 //support file for codec, DSK
2   Uint32 fs=DSK6713_AIC23_FREQ_96KHZ;       //set sampling rate
3   #include <math.h>        //for performing modulation operation
4   int i_4PAM_M, j_4PAM_M;
5   int masked_value,initial, output, assign;
6   int data_4PAM[4];
7   short k=0;

8   interrupt void c_int11()                  //interrupt service routine
9   {
10  if (i_4PAM_M==96)   //new input is taken every 96 samples
11  {
12  sample_data = input_sample();    //inputs data
13  i_4PAM_M=0;
14  j_4PAM_M=0;
15  }
16  masked_value = sample_data & 0x0003; //masks sample as 2-bit segments
17  output = data_4PAM[masked_value];    //gets corresponding level
18  output_sample(output);   //outputs corresponding voltage level 12 times
19  j_4PAM_M++;               //repeated output counter
20
21  if (j_4PAM_M==12)         //checks if repetition is over
22  {
23  j_4PAM_M=0;
24  sample_data = sample_data >> 2;   //shifts input to mask next segment
25  }
26  i_4PAM_M++;
27  return;
28  }                               // end of interrpt routine

29  void main()
30  {
31  initial=0x7FFF;
32  for(k=0; k<4; k++)        //forms look-up table for 4-level PAM
33  {
34        assign = initial-(k*0x5555);
35        data_4PAM[k] = assign;
36  }
37  i_4PAM_M=0;
38
39  comm_intr();                           //init DSK, codec, McBSP
40  while(1);                              //infinite loop
```

**Figure 3.22: 4-level PAM [2]**

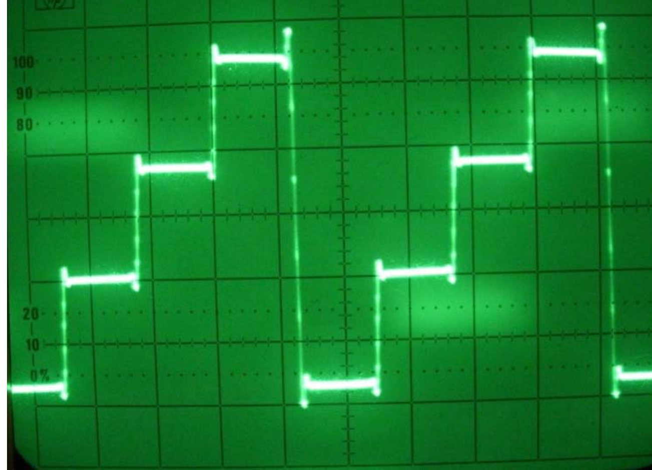


**Figure 3.23: 4-level PAM obtained with the Scope**

### 3.3.1.2 8-Level Pulse Amplitude Modulation

The 8-level PAM implementation is similar to 4-level PAM which is discussed earlier with difference in masking, shifting and lookup tables. For the 8-level PAM, if masking is done in the identical manner as that of 4- level PAM, when the input 16 bit long sample is decomposed into segments 3 bit long, 1 bit is left ungrouped. Two of the very simple approaches to solve this issue is to either we discard the LSB in each input sample so that remaining 15 bits become multiple of 3 or we store 3 consecutive input samples each 16 bit long in the memory buffer so that now total number of bits (48) become multiple of 3 and then perform the rest of operations. Just for ease of implementation we will stick to the first approach and discard LSB of every input sample assuming that it would not impart drastic effects on the modulated waveform. Figure 3.24 shows 8-level PAM lookup table for mapping.

| Symbol Block | Level (in hex) |
|---|---|
| 000 | 0x7FFF |
| 001 | 0x5B6D |
| 010 | 0x36DB |
| 011 | 0x1249 |
| 100 | -0x1249 |
| 101 | -0x36DB |
| 110 | -0x5B6D |
| 111 | -0x7FFF |

**Figure 3.24: 4-level PAM lookup table for mapping**

Figure 3.25 shows source code for generating 8-level PAM.

```
1  int i_8PAM_M, j_8PAM_M, masked_value, initial, output, assign, data_8PAM[8];
2  short k=0;
3  interrupt void c_int11()                  //interrupt service routine
4  {
5  if (i_8PAM_M==60)   //new input is taken every 60 samples
6  {sample_data = input_sample();    //inputs data
7  sample_data = sample_data >> 1;  //least significant bit discarded
8  i_8PAM_M=0;
9  j_8PAM_M=0;}
10 masked_value = sample_data & 0x0007; //masks sample as 3-bit segments
11 output = data_8PAM[masked_value];    //gets corresponding level
12 output_sample(output);   //outputs corresponding voltage level 12 times
13 j_8PAM_M++;                 //repeated output counter
14 if (j_8PAM_M==12)          //checks if repetition is over
15 {j_8PAM_M=0;
16 sample_data = sample_data >> 3;}  //shifts input to mask next segment
17 i_8PAM_M++;
18 return;
19 }                                // end of interrpt routine
20 void main()
21 {initial=0x7FFF;
22 for(k=0; k<8; k++)          //forms look-up table for 8-level PAM
   {  assign = initial-(k*0x2492); data_8PAM[k] = assign; }
   i_8PAM_M=0;
23 comm_intr();                              //init DSK, codec, McBSP
24 while(1);                                 //infinite loop
```

**Figure 3.25: 8-level PAM [2]**

Figure 3.26 shows the output of DSK for 8-PAM obtained with the oscilloscope.

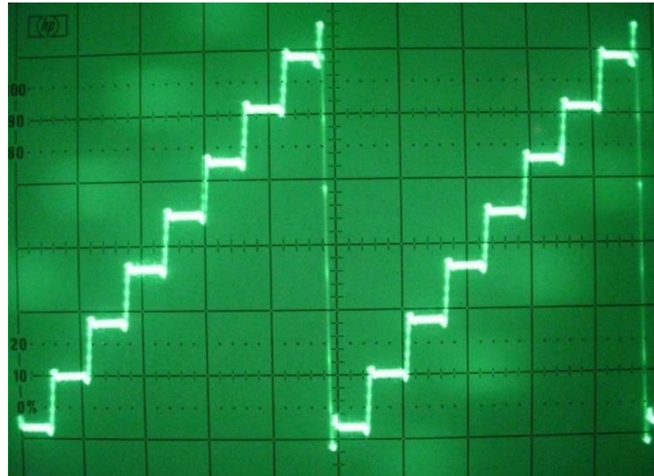

**Figure 3.26: 8-level PAM obtained with the Scope**

### 3.3.2 Phase Shift Keying

Phase shift keying (PSK) is a modulation schemes in which phase of the transmitted signal is varied in accordance with the input bit stream while other signal parameters such as amplitude and frequency remain fixed. There are several ways in which PSK can be implemented, the simplest one being binary PSK (BPSK) which uses only two signal phases (0° and 180°) to convey the information. Different other variants of PSK are 4-level PSK, 8-level PSK and so on. With increasing level of PSK, data is transmitted at faster rate per phase change. In 4-level PSK commonly known as Quadrature PSK (QPSK), two bits are grouped together to form a symbol. This symbol is a particular waveform sent across the channel with phase of 0°, +90°, -90°, or 180° depending upon bit combination. At the receiver the signal is demodulated and with the help of phase angle of the recovered symbol, it is check for the possible pair of bits which was sent [2].

#### 3.3.2.1 Binary Phase Shift Keying (BPSK)

The source code for BPSK is shown in figure 3.28. Line 12 obtains 16 bit long input sample of the external input. Now the input sample is segmented into sixteen 1-bit components by masking the input sample with 0x0001 (line 16).In order to obtain the next segment to be processed, the

previous input data is shifted once towards right (line 24). After segment extraction, sinusoidal wave with $0^o$ and $180^o$ phase are assigned for bits 0 and 1, respectively (line 17). As input sample is represented by 16 bits so every sampled data contains 16 segments and each symbol assigned to the segment is transmitted by a sinusoid generated by 4 points so the next input sample is obtained every 64 output samples of BPSK (line 10).

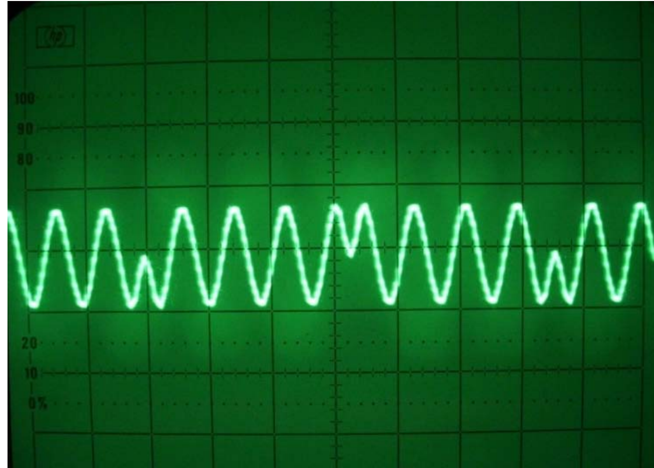


**Figure 3.27: BPSK obtained with the Scope**

```
1   include "dsk6713_aic23.h"                  //support file for codec, DSK
2   Uint32 fs=DSK6713_AIC23_FREQ_96KHZ;        //set sampling rate
3   #include <math.h>        //for performing modulation operation
4   int i_BPSK_M, j_BPSK_M;
5   int masked_value, output;
6   //Data table for BPSK MOD
7   int data_BPSK[2][4]={0, 1000, 0, -1000,
                         0, -1000, 0, 1000};
8   interrupt void c_int11()                   //interrupt service routine
9   {
10  if (i_BPSK_M==64)          //determines when to get new input
11  {
12        sample_data = input_sample();     //inputs data
13        i_BPSK_M=0;
14        j_BPSK_M=0;
15  }
16  masked_value = sample_data & 0x0001;       //masks input sample as 1-bit segments
17  output = data_BPSK[masked_value][j_BPSK_M]; //gets corresponding level from table
18  output_sample(output*10);                   //outputs corresponding sinusoid
19  j_BPSK_M++;                                 //repeated output counter
20
21  if (j_BPSK_M==4)                          //checks if 1-bit segment was output
22  {
23        j_BPSK_M=0;
24        sample_data = sample_data >> 1;  //shifts input so as to mask another part
25  }
26  i_BPSK_M++;
27  return;
28  }                               // end of interrpt routine

29  void main()
30  {
31  i_BPSK_M=64;
32  j_BPSK_M=0;
33  comm_intr();                              //init DSK, codec, McBSP
34  while(1);                                 //infinite loop
```

**Figure 3.28: BPSK source code [2]**

### 3.3.2.2 Quadrature Phase Shift Keying (QPSK)

The QPSK implementation is similar to BPSK which is discussed earlier with difference in masking, shifting and lookup for sinusoids. The source code for QPSK is shown in figure 3.29. The 16 bit long sample is fractioned into eight 2-bit components. This is done by masking the input with 0x0003 (line 18) .In orders to obtain the next segment to be processed, the previous input data is shifted twice (line 26). Following the extraction of segments, sinusoids with corresponding phases are assigned. For 2-bit components 00, 01, 11 and 10 sinusoids with phases 0, 90,180 and 270 are allotted.

```
1  include "dsk6713_aic23.h"                  //support file for codec, DSK
2  Uint32 fs=DSK6713_AIC23_FREQ_96KHZ;        //set sampling rate
3  #include <math.h>        //for performing modulation operation
4  int i_QPSK_M, j_QPSK_M;
5  int masked_value, output;
6  int data_QPSK[4][4]={0, 1000, 0, -1000,           //phase=0 & symbol=00
7                      1000, 0, -1000, 0,            //phase=90 & symbol=01
8                      -1000, 0, 1000, 0,            //phase=270 & symbol=10
9                      0, -1000, 0, 1000};           //phase=180 & symbol=11
10 interrupt void c_int11()                   //interrupt service routine
11 {
12 if (i_QPSK_M==32)          //determines when to get new input
13 {
14       sample_data = input_sample();     //inputs data
15       i_QPSK_M=0;
16       j_QPSK_M=0;
17 }
18 masked_value = sample_data & 0x0003;       //masks input sample as 2-bit segments
19 output = data_QPSK[masked_value][j_QPSK_M]; //gets corresponding level from table
20 output_sample(output*10);                   //outputs corresponding sinusoid
21 j_QPSK_M++;                                 //repeated output counter
22
23 if (j_QPSK_M==4)                        //checks if 2-bit segment was output
24 {
25       j_QPSK_M=0;
26       sample_data = sample_data >> 2;  //shifts input so as to mask another part
27 }
28 i_QPSK_M++;
29 return;
30 }                               // end of interrpt routine

31 void main()
32 {
33 i_QPSK_M=32;
34 j_QPSK_M=0;
35 comm_intr();                            //init DSK, codec, McBSP
36 while(1);                               //infinite loop
```

**Figure 3.29: QPSK source code [2]**

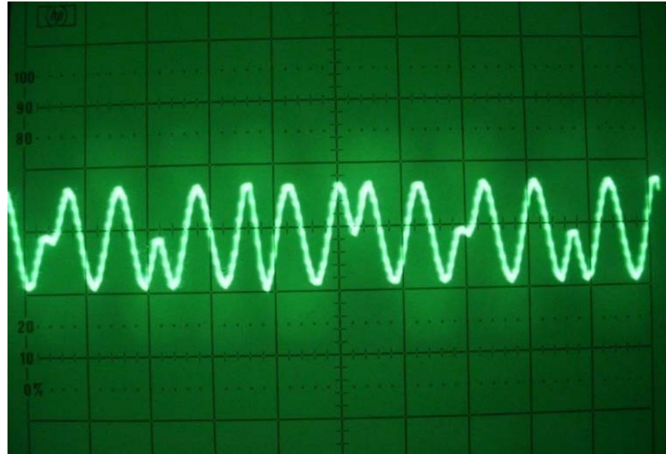

**Figure 3.30: QPSK obtained with the Scope**

All the programs discussed in the section “Implementation” are incorporated in a single source file “sine_varFreq.c” which is present in the folder “sine_varFreq”, included in the attached CD-ROM. Let’s now start the signal generator:

- Plug in the 5V power supply adapter on the DSK board to power it up.
- Plug in one end of the USB cable (included with the DSK package) into the DSK and other end in the PC’s USB port.
- Launch the DSK diagnostic utility to test the connection between PC and DSK.
- Launch the CCS from the icon on the desktop.
- If CCS version 3.1 is being used “No target connected” icon will appear on the bottom left of CCS screen. Go to “Debug” pull down menu and select “connect”. Now DSP will be connected to CCS.
- Go to the “Project” pull down menu and select “Open”. Now “Project Open” window will be displayed. Now select “sine_varFreq.pjt” from the folder “sine_varFreq”.
- If you are running the signal generator for the first, at this stage, CCS will ask for the paths of some files. Please refer to page 5 of this report to give CCS missing paths.

- Go to “File” pull down menu and select “Load GEL”. Now open “gainFreq.gel” from the folder “sine_varFreq”.
- Select Project → Rebuild All. This will compile and assembles all the C files and then links the resulting machine code with the library files to output the executable file that will be loaded on the DSP.
- Select Project → Load Program to load the executable file created in the previous step.
- After building and loading select Debug→ Run to run the  signal generator.
- To open the GUI of signal generator please select GEL → MENU. Under MENU, different sliders as discussed earlier will appear. Select the sliders which you want to use.

# 4 Conclusion and Future Work

Flexible signal generators are often used in measuring and testing of communication systems. Mostly they are used to generate baseband signals which either serve as a testing signal or they are processed in further stages such as filtering or modulation. These generated signals afterwards can also be fed to the input of the receiver to characterize certain transmission channel.

In this project implementation of signal generator using TI floating point C6713 DSK has been discussed. CCS, an integrated development environment is used to implement the DSP algorithms with higher level programming language i.e. C. CCS does the compilation and linking of DSP algorithms written in C and then transfers the machine code translation from host PC to C6713 DSK through JTAG emulation port (the USB port). GEL is used to create the GUI for signal generator.

The onboard 32-bit AIC23 stereo codec designed with sigma-delta technology performs all necessary task (ADC, DAC, sampling, low pass filtering) which are required to connect the DSK with the outside world. Variable sampling rates from 8 to 96 kHz can be set readily. So using Nyquist criterion, the signals that can be recovered from the sampled values are those that have frequency components less than half the maximum sampling frequency (96 kHz). Frequency components above the allowed frequency will be filtered out by the anti-aliasing filter. To generate the high frequency signals one needs to interface an external daughter card that can enhance the built-codec. One possible solution is to use 16-bit mono DAC8581 EVM along with 5-6K interface board to generate signals up to 1.5 MHz.

At the moment, a user can choose one of the 20 possible frequencies to generate a waveform from the signal generator. In future this limitation can be worked out and current signal generator can be upgraded to a more flexible signal generator.

# Appendix

## Signal Generator Menu

The signal generator is equipped with GUI objects called “sliders”. These sliders are used to vary different parameters of the generated output signal. In total, there are five sliders. A brief description of each slider is given below:

- **Mode:** This slider selects the type of the output signal (waveform, noise or digital modulation scheme).
- **Gain:** This slider varies the gain of the output signal.
- **Frequency:** This slider varies the frequency of the output signal.
- **f1:** This slider works only in multi-tone signal generation. It varies one of the two frequencies present in multi-tone signal.
- **f2:** This slider works only in multi-tone signal generation. It varies one of the two frequencies present in multi-tone signal.
- **Ratio_gain_f1_VS_f2:** This slider defines how the overall gain of the multi-tone signal, decided by “Gain” slider is distributed among the frequencies f1 and f2. When the slider is at position 0, only frequency f1 exists and when the slider is at potion 100, only frequency f2 is present in multi-tone signal. In between slider position 0 and 100, gain is distributed among frequencies f1 and f2 depending upon the position of slider.

The “Gain” slider has 100 steps so a user can scale the signal by 100 different gain levels. The “Frequency”, “f1” and “f2” sliders have 20 steps and user can vary the frequency of output signal ranging from 750 Hz (apporx) to 15 kHz (approx) with step size of 750 Hz (approx). In digital modulation schemes, the modulated signal frequency in BPSK and QPSK is 25 kHz. Below is the description of generated output signals corresponding to different positions of the mode slider. Green colour indicates the slider name while blue colour indicates the characteristics of the generated signal pertinent to a specific mode. Under each mode we have listed different sliders available to the user corresponding to that particular mode.

**Mode 1: Sine Wave**

Gain
Frequency

**Mode 2: Square Wave**

Gain
Frequency

**Mode 3: Triangular Wave**

Gain
Frequency

**Mode 4: Saw-tooth Wave**

Gain
Frequency

**Mode 5: Multi Tone Signal**

Gain
f1
f2
Ratio_gain_f1_VS_f2

**Mode 6: Pseudo Random Noise Sequence Generator**

Gain

**Mode 7: Pseudo Random Noise Sequence Generator + Input signal (external)**

Gain (gain of noise)

**Mode 8: 4-PAM Modulator**

Output signal corresponds to predefined input signal for demonstration

**Mode 9: 4-PAM Modulator**

Output signal corresponds to input signal by external source

**Mode 10: 8-PAM Modulator**

Output signal corresponds to predefined input signal for demonstration

**Mode 11: 8-PAM Modulator**

Output signal corresponds to input signal by external source

**Mode 12: BPSK Modulator**

Output signal corresponds to predefined input signal for demonstration

**Mode 13: BPSK Modulator**

Output signal corresponds to input signal by external source

**Mode 14: QPSK Modulator**

Output signal corresponds to predefined input signal for demonstration

**Mode 15: QPSK Modulator**

Output signal corresponds to input signal by external source

# References

[1] N. Kehtarnavaz, *"Real-Time Digital Signal Processing Based on the TMS320C6000"*, Elsevier Inc, Oxford, 2005.

[2] R. Chassaing, "*Digital Signal Processing and Applications with the C6713 and C6416 DSK*", John Wiley & Sons Inc., New Jersey, 2005.

[3] Texas Instruments, *"TMS320C6201/6701 Evaluation Module Reference Guide"*, Literature ID# SPRU 269F, 2002.

[4] Z.W. Mekonnen, *"Digital Signal Processing Applications using DSK C6713"*, Communications Laboratory, University of Kassel, 2009.

[5] S.W. Smith "*The Scientist and Engineer's Guide to Digital Signal Processing"*, California Technical Publishing, San Diego, 1999.

[6] J.B. Tsui, *"Fundamentals of Global Positioning System Receivers: A Software Approach"*, John Wiley & Sons, Inc., New Jersey, 2005.

[7] S.A. Tretter, *"Communication System Design using DSP Algorithms"*, Springer Science + Business Media, LLC, New York, 2008.

[8] B.P. Kumar, *"Digital Signal Processing laboratory",* CRC Press, Florida, 2005.

[9] Encarta encyclopedia website, URL: http:// encarta.msn.com/ dictionary_1861735066/ signal_generator.html (visited on 7.11.2009-1600 GMT)

[10] National Instruments website, URL: http://zone.ni.com/devzone/cda/tut/p/id/3348#toc0 (visited on 7.11.2009-1800 GMT)

[11] Wikipedia website, URL: http://en.wikipedia.org/wiki/Signal_generator (visited on 8.11.2009-1100 GMT)

[12] Wikipedia website, URL: http://en.wikipedia.org/wiki/Arbitrary_waveform_generator (visited on 8.11.2009-1125 GMT)

[13] URL: http://www.electronicsprojectdesign.com/SignalGenerator.html (visited on 8.11.2009 - 1500 GMT)

[14] URL: http://www.justsignalgenerator.com/ (visited on 8.11.2009 - 1545 GMT)