CHAPTER 1

TMS320C6713 DSK

1.1 TMS320C6713 DSK Overview

Several tools are available for digital signal processing (DSP). These tools include the popular Code Composer Studio (CCS), which provides an integrated development environment, and the DSP starter kit (DSK) with the TMS320C6713 floating-point processor on board and complete support for input and output. Digital signal processors such as the TMS320C6x (C6x) family of processors are like fast special-purpose microprocessors with a specialized type of architecture and an instruction set appropriate for signal processing. Digital signal processors are used for a wide range of applications, from communications and controls to speech and image processing.DSP techniques have been very successful because of the development of low-cost software and hardware support. For example, modems and speech recognition can be less expensive using DSP techniques.DSP processors are concerned primarily with real-time signal processing. Real time processing requires the processing to keep pace with some external event, whereas non-real-time processing has no such timing constraint. The external event to keep pace with is usually the analog input. Whereas analog-based systems with discrete electronic components such as resistors can be more sensitive to temperature changes, DSP-based systems are less affected by environmental conditions. DSP processors enjoy the advantages of microprocessors. They are easy to use and economical.

1.2 DSK Support Tools

Following tools are used:

1. TIs DSP starter kit (DSK)

- Code Composer Studio (CCS)
- A board
- A universal synchronous bus (USB) cable that connects the DSK board to a PC
- A 5V power supply for the DSK board

2. An IBM-compatible PC

3. An oscilloscope, signal generator, and speakers

1.3 DSK BOARD

The DSK package is powerful board, with an approximate size of 5x8 in., includes the C6713 floating-point digital signal processor and a 32-bit stereo codec TLV320AIC23 (AIC23) for input and output.

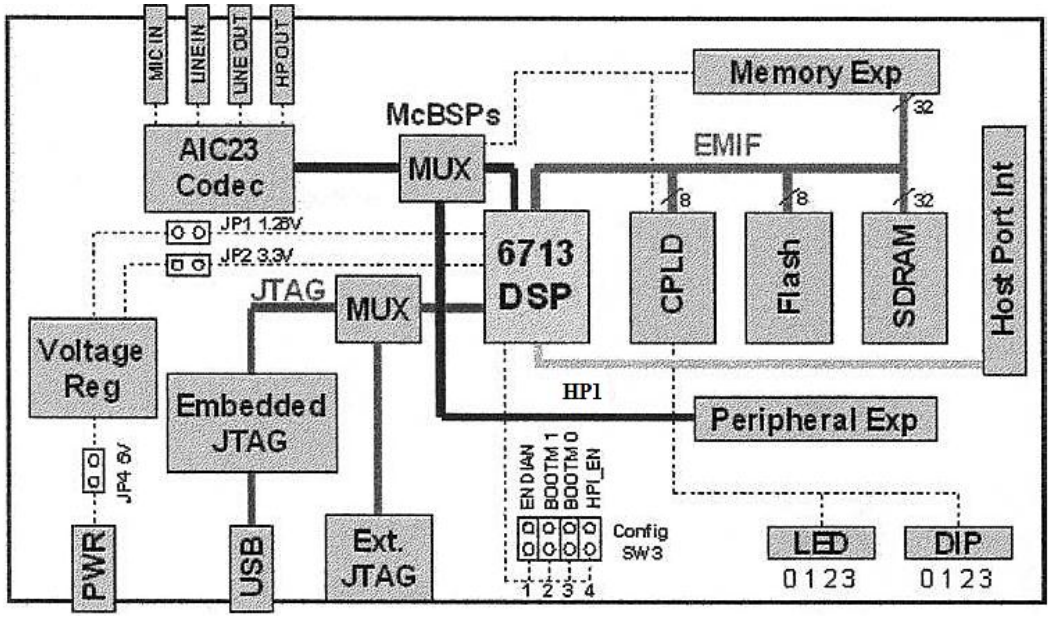


Figure 1.1: TMS320C6713-based DSK Board

The TMS320C6713 (C6713) is based on the VLIW architecture, which is very well suited for numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. For example, with a clock rate of 225MHz, the C6713 is capable of fetching eight 32-bit instructions every 1/(225 MHz) or 4.44 ns.

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.

1.4 FEATURES OF THE TMS320C6713 DSKS

- 264 kB of internal memory
- Eight functional or execution units composed of six arithmetic-logic units (ALUs)
- Two multiplier units
- A 32-bit address bus to address 4 GB
- Two sets of 32-bit general-purpose registers.
- On board Codec AIC23 provides ADC and DAC.
- 2-Multichannel Buffered Serial Ports (McBSP

CHAPTER 2

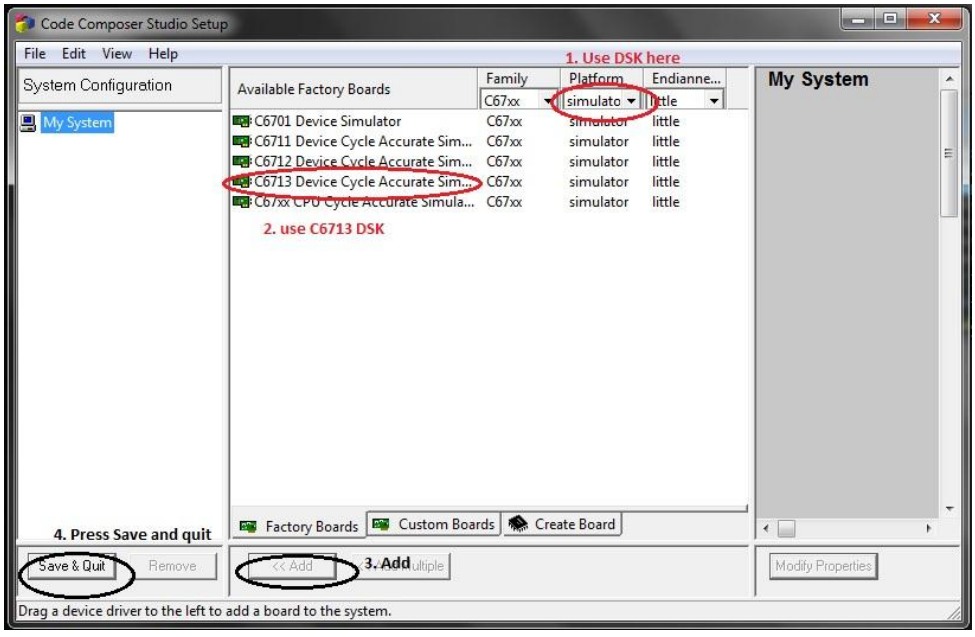
CODE COMPOSER STUDIO

2.1 INTRODUCTION

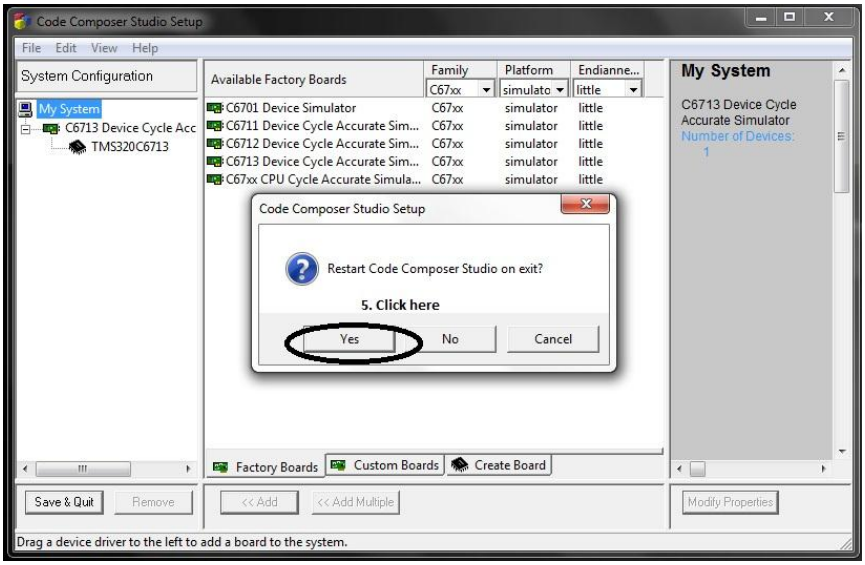
Code Composer Studio speeds and enhances the development process for programmers who create and test real-time, embedded signal processing applications. Code Composer Studio extends the capabilities of the Code Composer Integrated Development Environment (IDE) to include full awareness of the DSP target by the host and real-time analysis tools. It provides tools for configuring, building, debugging, tracing, and analyzing programs. Code Composer Studio, which is includes the TMS320C6000 code generation tools along with the APIs and plug-ins for both DSP/BIOS and RTDX. Within Code Composer Studio, you create an application by adding files to a project.

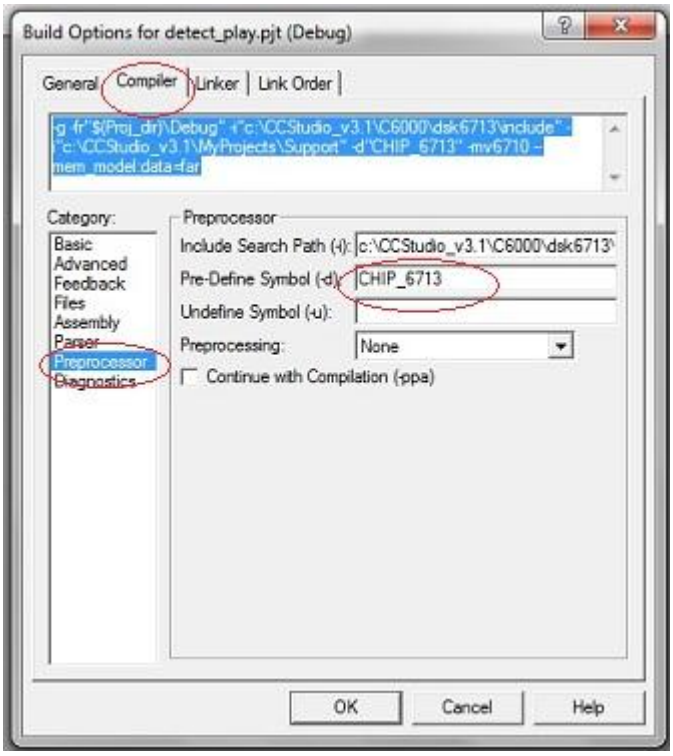
2.2 CREATING NEW PROJECT IN C. C. STUDIO

Following steps should be followed while creating a new project:

1. Create folder C:\CCStudio_v3.1\MyProjects\detect_play.
2. Place all source files including detect_play.c (main source program.)
3. Open **Setup** Code Composer Studio.
4. 

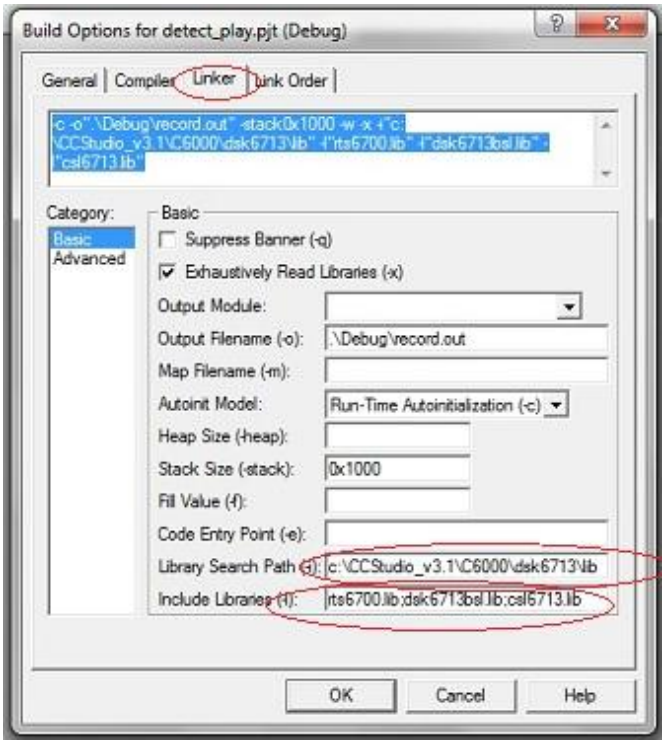
5.



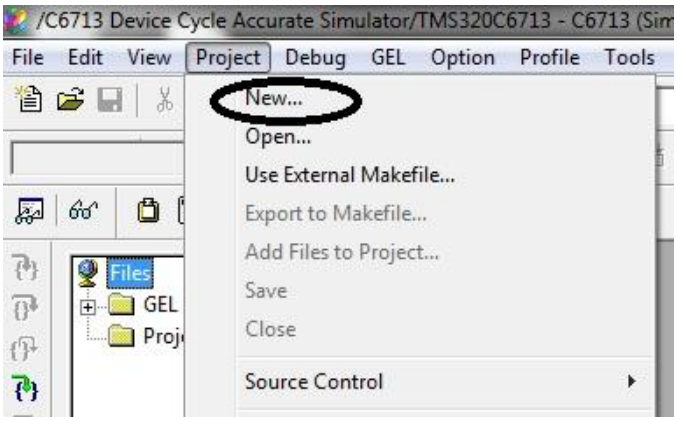


6. Go to *Project→Build* options. Select Compiler tab and go to Basic in Category listing. Change the Target Version to C671x(-mv6710) now select Advanced in the Category listing and change the Memory Models *to --mem_model:data=far* and in the last go to Preprocessor in the Category listing and change the Pre-Define Symbol and Include Search Path to *CHIP_6713* and *C:\C6713\C6000\dsk6713\include* respectively.

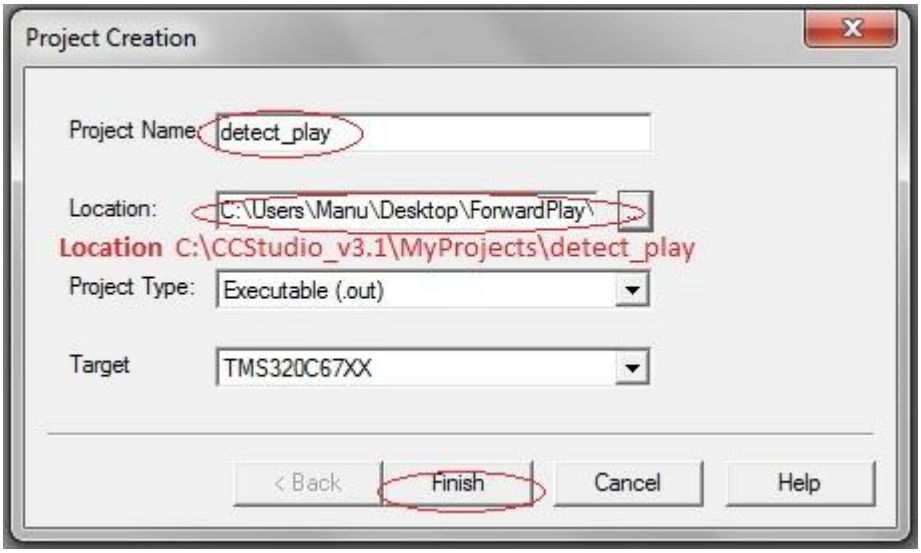
7. Go to *Project→Build* options click *Linker* Tab *Library Search Path=* C:\CCStudio_v3.1\C6000\dsk6713\lib;C:\CCStudio_v3.1\C6000\cgtools\lib; *Include Libraries* rts6700.lib, dsk6713bsl.lib, csl6713.lib: run-time, board, and chip support library files, respectively.



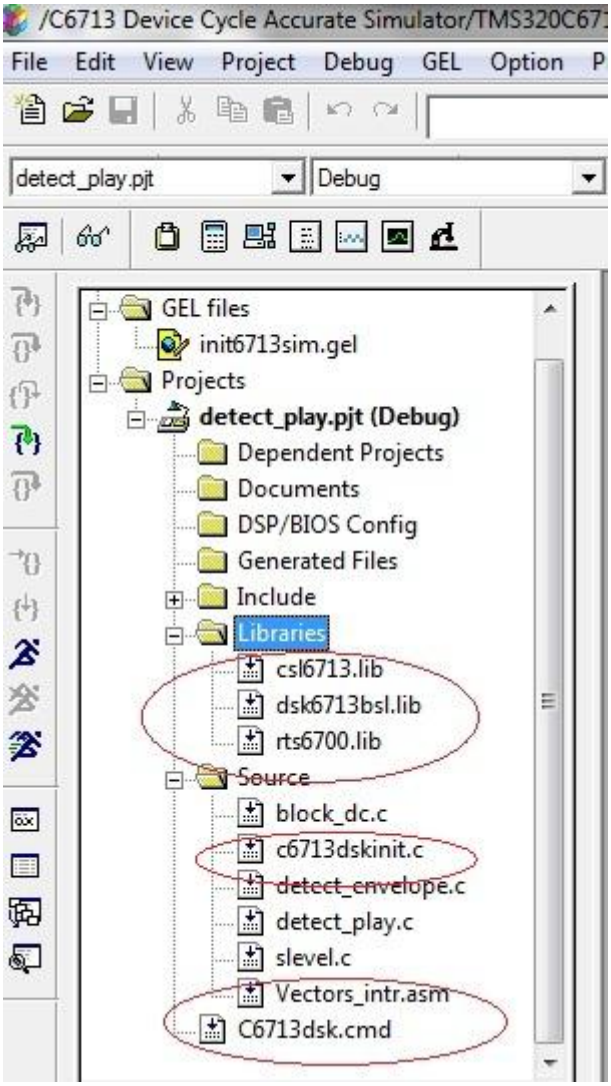
8. Create new project by going to *Project -> New…* OR *Open…* in latter case skip step 9, 10.



9.

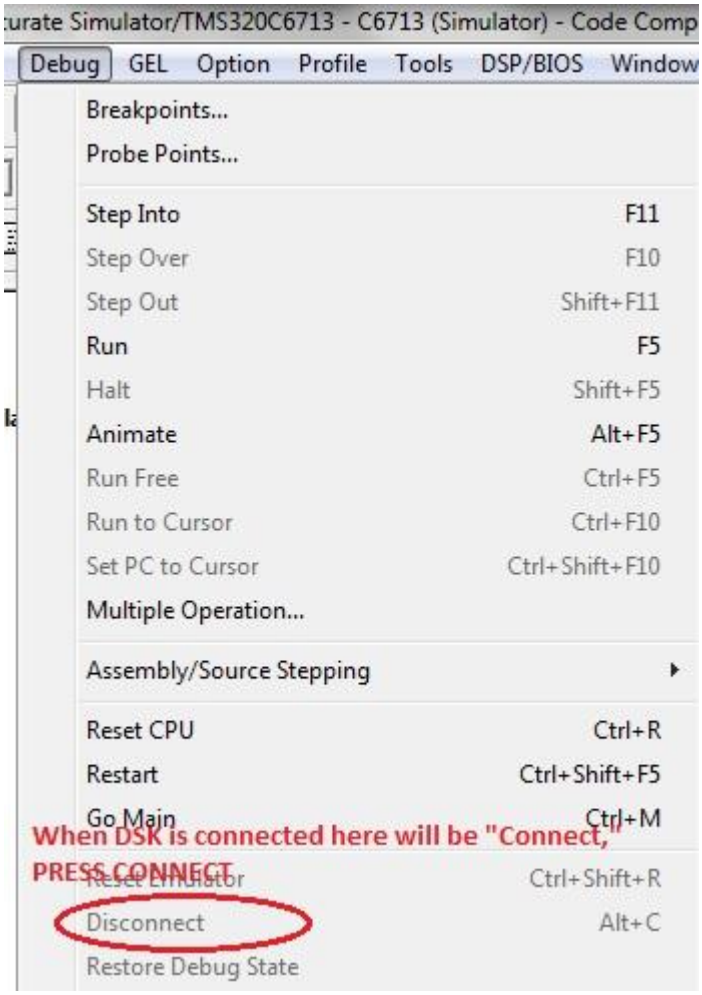


10. Add all source files and few necessary support files in support folder to project right click **detect_play.pjt (Debug)**

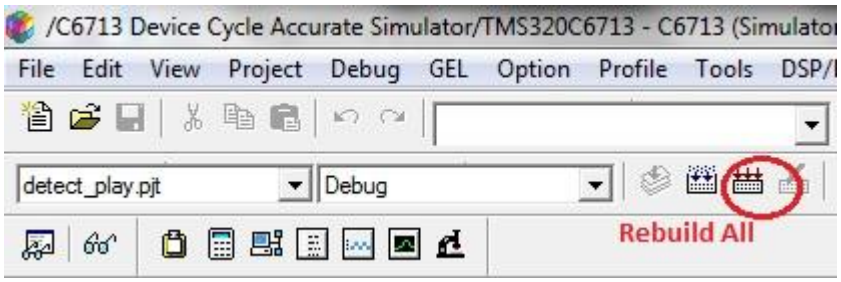


Click on *add files to project.* Do not add header files to project they will be automatically added later. Depending upon program Vectors_poll.asm file and some other files could also be required to add.

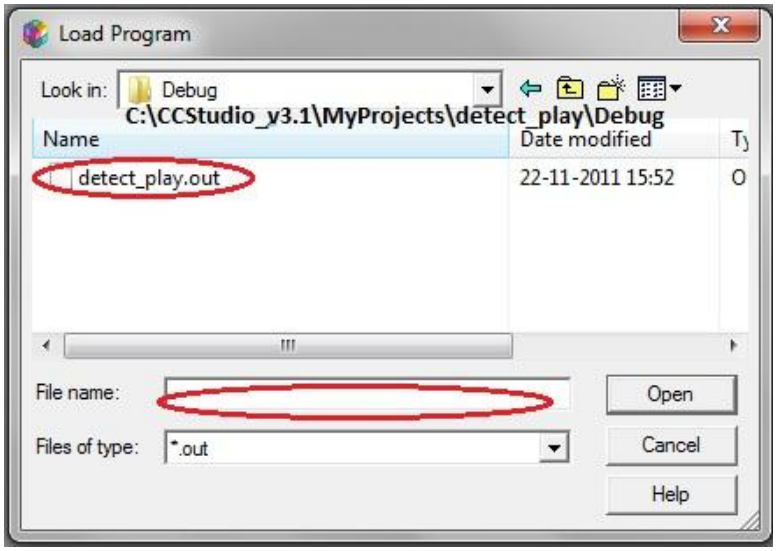
11.



12. Rebuild All

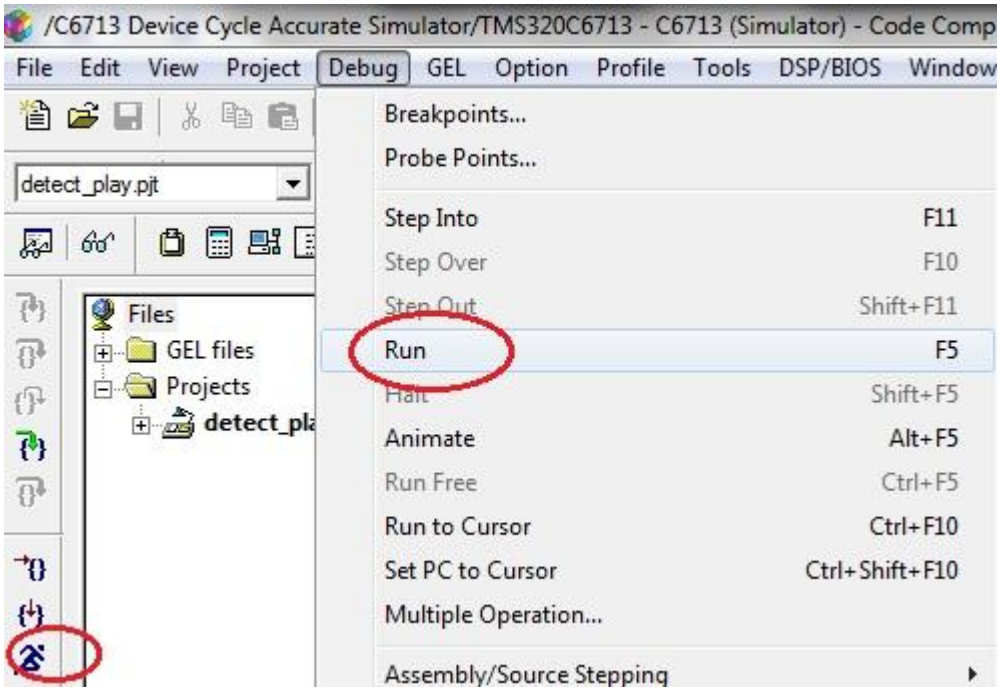


13. Go to



File -> Load Program.

14. Run Program



2.3 CCS DEVELOPMENT CYCLE

Code Composer Studio extends the basic code generation tools with a set of debugging and real-time analysis capabilities. Code Composer Studio supports all phases of the development cycle shown here:

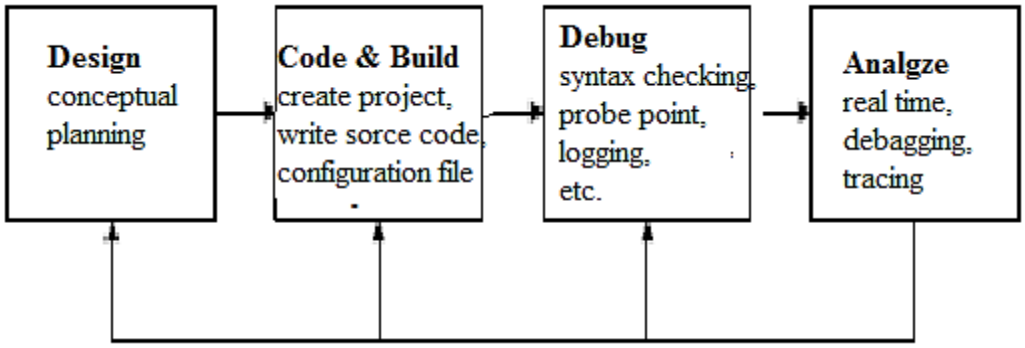


Figure 2.1: CCS Development cycle

CCS software tool is used to generate TMS320C6x executable files. CCS includes the assembler, linker, compiler, and simulator and debugger utilities. Figure 2.1 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 equipment 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.

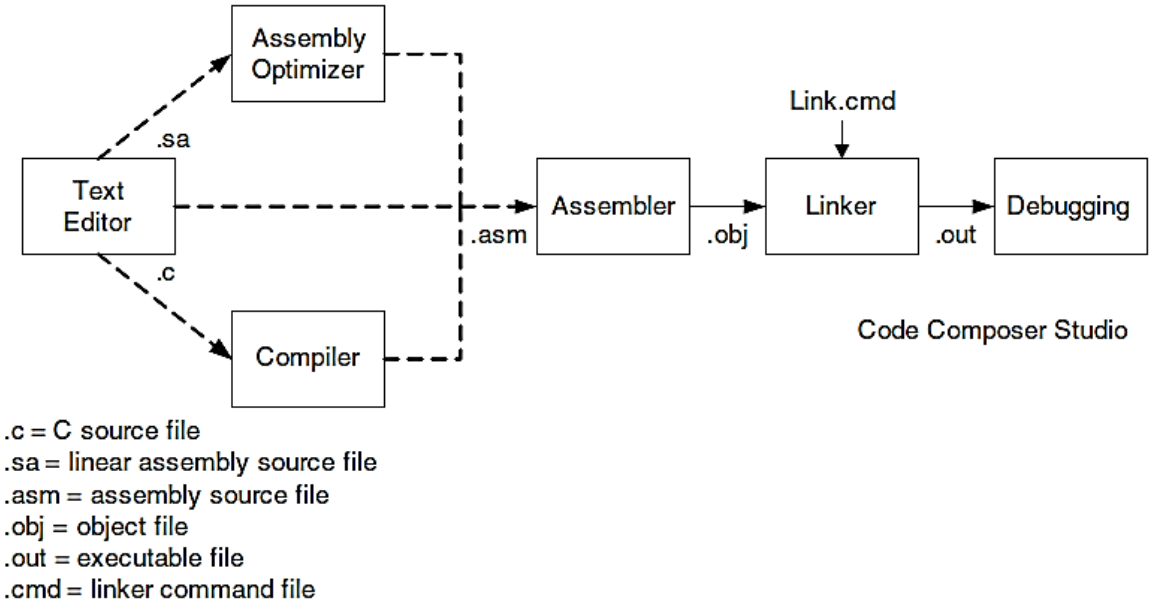


Figure 2.2: C6x CCS Software Tool

All the necessary files and steps required to build a project on code composer studio are explained in detail in [4].

2.4 Support Files

The following support files located in the folder support (except the library files) are used for most of the examples and projects discussed in this book:

1. *C6713dskinit.c:* contains functions to initialize the DSK, the codec, the serial ports, and for I/O. It is not included with CCS.

2. *C6713dskinit.h:* header files with function prototypes. Features such as those used to select the mic input in lieu of line input (by default), input gain, and so on are obtained from this header file (modified from a similar file included with CCS).

3. *C6713dsk.cmd:* sample linker command file. This generic file can be changed when using external memory in lieu of internal memory.

4. *Vectors_intr.asm:* a modified version of a vector file included with CCS to handle interrupts. Twelve interrupts, INT4 through INT15, are available, and INT11 is selected within this vector file. They are used for interrupt-driven programs.

5. Vectors_poll.asm: vector file for programs using polling.

6. *rts6700.lib,dsk6713bsl.lib,csl6713.lib:* run-time, board, and chip support library files, respectively. These files are included with CCS and are located in *C6000\cgtools\lib, C6000\dsk6713\lib*, and *c6000\bios\lib,* respectively.

CHAPTER 3

VOICE ACTIVITY DETECTION

Voice activity detection (VAD), also known as speech activity detection or speech detection, is a technique used in speech processing in which the presence or absence of human speech is detected. The main uses of VAD are in speech coding and speech recognition. It can facilitate speech processing, and can also be used to deactivate some processes during non-speech section of an audio session.

3.1 APPLICATIONS

VAD is an integral part of different speech communication systems such as audio conferencing, echo cancellation, speech recognition, speech encoding, and hands-free telephony. Its different applications are given below:

- In the field of multimedia applications, VAD allows simultaneous voice and data applications.
- Similarly, in Universal Mobile Telecommunications Systems (UMTS), it controls and reduces the average bit rate and enhances overall coding quality of speech.
- In cellular radio systems (for instance GSM and CDMA systems) based on Discontinuous Transmission (DTX) mode, VAD is essential for enhancing system capacity by reducing co-channel interference and power consumption in portable digital devices.
- It can avoid unnecessary coding/transmission of silence packets in Voice over Internet Protocol (VOIP) applications, saving on computation and on network bandwidth.

For a wide range of applications such as digital mobile radio, Digital Simultaneous Voice and Data (DSVD) or speech storage, it is desirable to provide a discontinuous transmission of speech-coding parameters. Advantages can include lower average power consumption in mobile handsets, higher average bit rate for simultaneous services like data transmission, or a higher capacity on storage chips. However, the improvement depends mainly on the percentage of pauses during speech and the reliability of the VAD used to detect these intervals. On the one hand, it is advantageous to have a low percentage of speech activity. On the other hand clipping, that is the loss of milliseconds of active speech, should be

minimized to preserve quality. This is the crucial problem for a VAD algorithm under heavy noise conditions.

3.2 PERFORMANCE EVALUATION

To evaluate a VAD, its output using test recordings is compared with those of an "ideal" VAD - created by hand-annotating the presence/absence of voice in the recordings. The performance of a VAD is commonly evaluated on the basis of the following four parameters:

- FEC (Front End Clipping): clipping introduced in passing from noise to speech activity;
- MSC (Mid Speech Clipping): clipping due to speech misclassified as noise;
- OVER: noise interpreted as speech due to the VAD flag remaining active in passing from speech activity to noise;
- NDS (Noise Detected as Speech): noise interpreted as speech within a silence period.

Although the method described above provides useful objective information concerning the performance of a VAD, it is only an approximate measure of the subjective effect. For example, the effects of speech signal clipping can at times be hidden by the presence of background noise, depending on the model chosen for the comfort noise synthesis, so some of the clipping measured with objective tests is in reality not audible. It is therefore important to carry out subjective tests on VADs, the main aim of which is to ensure that the clipping perceived is acceptable. This kind of test requires a certain number of listeners to judge recordings containing the processing results of the VADs being tested. The listeners have to give marks on the following features:

- Quality;
- Comprehension difficulty;
- Audibility of clipping.

These marks, obtained by listening to several speech sequences, are then used to calculate average results for each of the features listed above, thus providing a global estimate of the behavior of the VAD being tested. To conclude, whereas objective methods are very useful in an initial stage to evaluate the quality of a VAD, subjective methods are more significant. As, however, they are more expensive (since they require the participation of a certain number of people for a few days), they are generally only used when a proposal is about to be standardized.

3.3 VOICE ACTIVITY DETECTION IN NOISY ENVIRONMENTS

An important problem in many areas of speech processing is the determination of presence of speech periods in a given signal. This task can be identified as a statistical hypothesis problem and its purpose is the determination to which category or class a given signal belongs. The decision is made based on an observation vector, frequently called feature vector, which serves as the input to a decision rule that assigns a sample vector to one of the given classes. The classification task is often not as trivial as it appears since the increasing level of background noise degrades the classifier effectiveness, thus leading to numerous

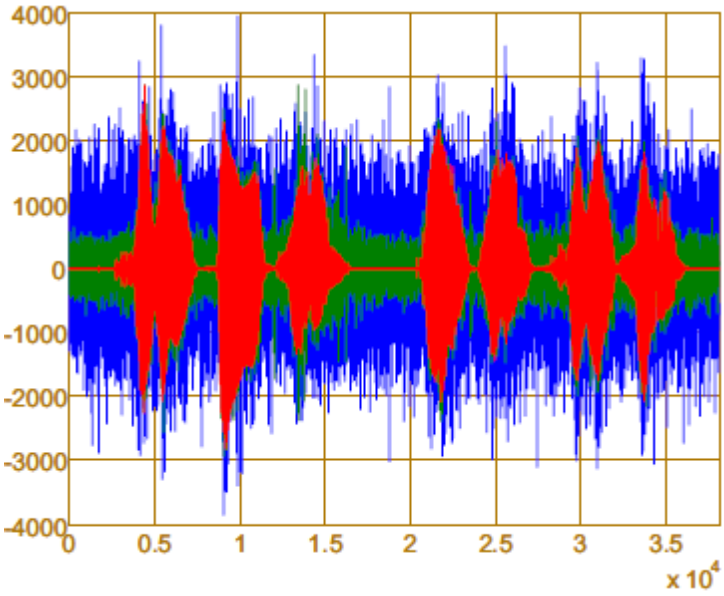


Fig. 3.1 Red Pure Signal, Green +5dB SNR, Blue -5dB SNR.

detection errors. Fig. 3.1 illustrates the challenge of detecting speech presence in a noisy signal when the level of background noise increases and the noise completely masks the speech signal. The selection of an adequate feature vector for signal detection and a robust decision rule is a challenging problem that affects the performance of VADs working under noise conditions. Most algorithms are effective in numerous applications but often cause detection errors mainly due to the loss of discriminating power of the decision rule at low SNR levels (ITU, 1996; ETSI, 1999). For example, a simple energy level detector can work satisfactorily in high signal-to-noise ratio (SNR) conditions, but would fail significantly when the SNR drops. VAD results more critical in non-stationary noise environments since it is needed to update the constantly varying noise statistics affecting a misclassification error strongly to the system performance.

3.4 Voice Detection & Reverse Play-Back

This project detects a voice signal from a microphone, and then plays it back in the reverse direction. Figure 3.2 shows the block diagram that implements this project. Two circular buffers are used: an input buffer to hold 80,000 samples (10 seconds of data) continuously being updated and an output buffer to play back the input voice signal in the reverse direction. The signal level is monitored, and its envelope is tracked to determine whether or not a voice signal is present.

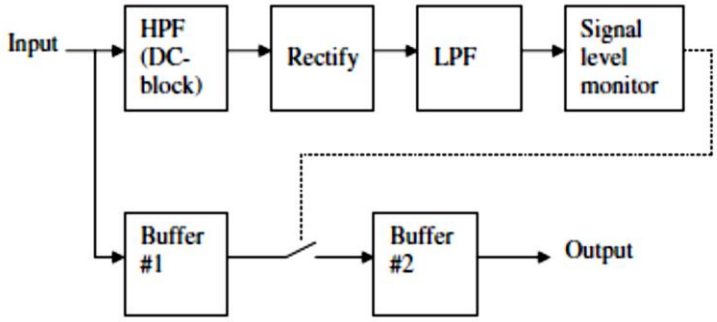


Fig. 3.2 Block diagram for the detection of a voice signal from a line-in and playback of that signal in the reverse direction [2]

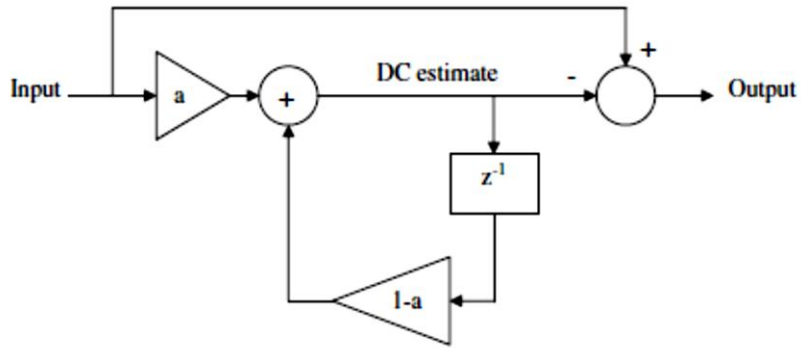


Fig. 3.3 DC blocking first-order IIR high-pass filter for voice signal detection and reverse playback [2]

When a voice signal appears and subsequently dies out, the signal-level monitor sends a command to start the playback of the accumulated voice signal, specifying the duration of the signal in samples. The stored data are transferred from the input buffer to the output buffer for playback. Playback stops when one reaches the end of the entire signal detected. The signal-level monitoring scheme includes rectification and filtering (using a simple first-order IIR filter).An indicator specifies when the signal reaches an upper threshold. When the signal drops below a low threshold, the time difference between the start and end is

calculated. If this time difference is less than a specified duration, the program continues into a no-signal state (if noise only). Otherwise, if it is more than a specified duration, a signal-detected mode is activated. Figure 3.3 shows the DC blocking filter as a first-order IIR high pass filter. The Coefficient a is much smaller than 1 (for a long time constant). The estimate of the DC filter is stored as a 32-bit integer. The low pass filter for the envelope detection is also implemented as a first-order IIR filter, similar to the DC blocking filter except that the output is returned directly rather than being subtracted from the input. The filter coefficient a is larger for this filter to achieve a short time constant.

Figure 3.4(a) shows voice recorded at rate of 8000 samples per second. Subsequent figures shows processing of this voice signal.

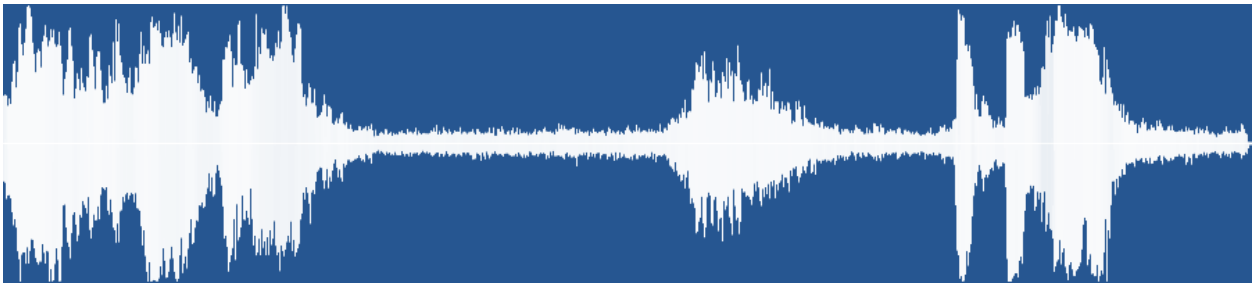


Fig 3.4(a) Actual sound with noise

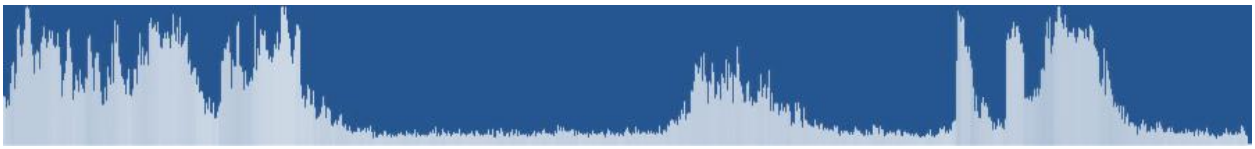


Fig 3.4(b) Rectified sound

Fig 3.4(c) Filtered sound

Fig 3.4(d) Sound with threshold level

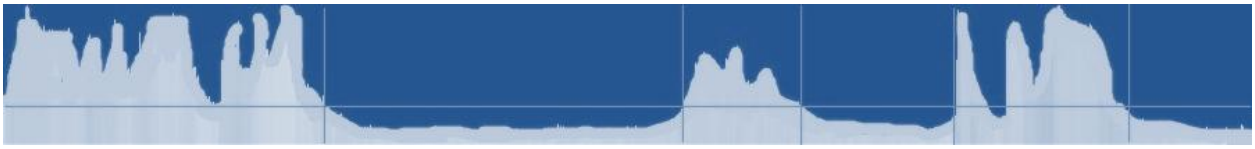


Fig 3.4(e)

Chopped voice portion (3000 samples)

Noise (short duration) (500 samples)

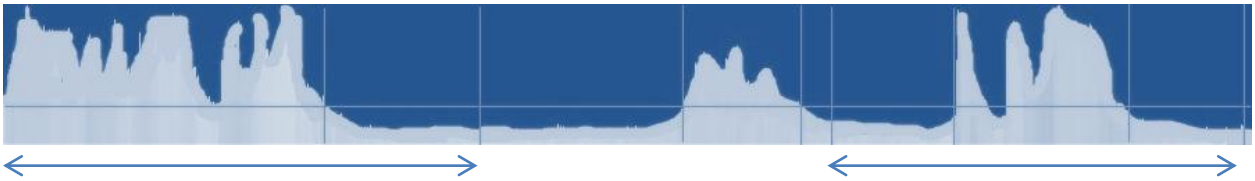


Fig 3.4(f) Voice without clipping

Voice without clipping

Figure 3.4(b) shows rectified voice signal, after passing it through LPF envelop of rectified signal is received as shown in figure 3.4(c). A voice signal is stronger than threshold level (figure 3.4(f)) and of long duration unlike noise signal which is shown in figure 3.4(e). Before and after voice signal crosses the threshold, a portion of voice is chopped. This chopped signal must also be recorded, which is about .375 seconds (3000 samples). Therefore chopped signal is also recorded as shown in figure 3.4(f).

CHAPTER 4

PROGRAMMING

4.1 DETECT_PLAY.C

```
#include "dsk6713_aic23.h"                                  //codec support
#include "slevel.h"
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
#define buffer_length 80000
#pragma DATA_SECTION(buffer,".EXT_RAM")
#pragma DATA_SECTION(playback_buffer,".EXT_RAM")
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;                  //set sampling rate
Uint16 inputsource=DSK6713_AIC23_INPUT_LINE;        // select input
short buffer[buffer_length];
short playback_buffer[buffer_length];
int buffer_pos = 0;             /* input buffer position */
int playback_pos = 0;           /* playback buffer position */
int gain = 1;                   /* output gain */
int duration = 0;               /* signal duration (playback on when duration > 0)*/
short buffer_data(short);       /* function declarations */
void start_playback(int *);
short playback(int *);
interrupt void c_int11()        /* interrupt service routine */
{       short sample_data;
        int temp;
        sample_data = input_sample();               /* input data */
        sample_data = buffer_data(sample_data);     /* buffer input */
        temp = signal_level(sample_data);           /* analyze the signal level */
 if (temp > 0 && duration == 0) {                   /* if signal detected and playback off */
```

```
            duration = temp;
            start_playback(&duration);          /* start playback */
        }
        if (duration > 0)                       /* if playback is on */
            sample_data = playback(&duration);  /* play stored data backwards */
        else
            sample_data = 0;                    /* output zero signal */
        output_sample(sample_data);             /* output data */
        return;
}
void main()
{  comm_intr();                                 /* init DSK, codec, McBSP */
   while(1);                                            /* infinite loop */
}                       /* store the input sample in a circular buffer */
short buffer_data(short sample)
{       buffer[buffer_pos] = sample;            /* store sample */
        buffer_pos++;                           /* increment buffer position */
        if (buffer_pos > buffer_length)
            buffer_pos = 0;                     /* buffer wrap-around */
        return sample;
}                       /* set up data structures for playback */
void start_playback(int *duration)
{       int i;
        if (*duration > buffer_length)
            *duration = buffer_length;      /* adjust duration to <= buffer length */
        playback_pos = buffer_pos;          /* copy buffer pointer */
        for (i=0;i<buffer_length;i++)       /* copy buffer */
            playback_buffer[i] = buffer[i];
}                       /* play back stored samples in reverse order */
```

```
short playback(int *duration)
{
        short output;
        output = playback_buffer[playback_pos]; /* outputting samples in reverse */
        output = gain * output;                 /* add gain to output */
        playback_pos--;  /* reducing the count to access the next sample */
        (*duration)--;   /* decrement duration (playback stops when duration == 0)*/
        if (playback_pos < 0)
                playback_pos += buffer_length;      /* buffer wrap-around */
        return output;  }
```

4.2 BLOCK_DC.C

```
//Block the DC in input signal.
//sample = current sample of input signal (16-bit:S15-bit)
//returns sample - DC (16-bit)
#define dc_coeff 10                         //coefficient for the DC blocking filter
int dc = 0;                                 //current DC estimate (32-bit: SS30-bit)
short block_dc(short sample)
{  short word1,word2;
  if (dc < 0) {
                word1 = -((-dc) >> 15);             //retain the sign when DC < 0
                word2 = -((-dc) & 0x00007fff);
                }
  else {word1 = dc >> 15;                           //word1=high-order 15-bit word of dc
        word2 = dc & 0x00007fff;                    //word2=low-order 15-bit word of dc
        }
    dc = word1 * (32768 - dc_coeff) +
    ((word2 * (32768 - dc_coeff)) >> 15) +
     sample * dc_coeff;                             //dc=dc*(1-coeff) + sample*coeff
     return sample - (dc >> 15);                        //return sample - dc
```

```
}
```

4.3 SLEVEL.C

```
#include "block_dc.h"
#include "detect_envelope.h"
#define threshold_low    200                /* signal loss threshold */
#define threshold_high   400                /* signal detection threshold */
#define threshold_start 500                 /* delay before turning the signal on */
#define threshold_stop   3000               /* delay before turning the signal off */
#define extra_samples    600                /* extra duration so signal is not chopped */
short signal_on = 0;                        /* "signal present" flag */
short signal_found = 0;                     /* "approved signal present" flag */
int duration1 = 0;                          /* approved signal duration */
int duration_lost = 0;                      /* signal loss duration */
int signal_level(short sample)
{  int signal;
   int ret = 0;
 signal = detect_envelope(block_dc(sample)); /* approx. signal envelope */
 if (signal_on)
        {                                   /* an approved signal is in progress */
        duration1++;
        if (signal < threshold_low)
                {                           /* if the signal is low */
                duration_lost++;            /* accumulate signal loss duration */
                if (duration_lost > threshold_stop)
                        {          /* signal lost: output duration */
                        ret = duration1 + extra_samples;       /* return signal duration */
                        signal_on = 0;                         /* indicate the signal is lost */
                        signal_found = 0;
```

```
                    duration1 = 0;
                    }
                }
        else    {duration_lost = 0;             /* reset signal loss duration */
                }
                }
  else if (signal_found)
        {                       /* a large enough signal was recently detected  */
        if (signal < threshold_low)
                {                       /* signal lost: reset duration */
                signal_found = 0;
                duration1 = 0;
                }
    else        {duration1++;
                if (duration1 > threshold_start) { /* signal is approved (not noise) */
                signal_on = 1;
                duration_lost = 0;
                }
        }
  }
  else if (signal > threshold_high)
        {                       /* a large enough signal is observed */
        signal_found = 1;       /* start signal tracking */
        duration1 = 1;
        }
  return ret;
}
```

4.4 DETECT_ENVELOPE.C

```
//Envelope detection routine
/* Approximate the envelope of the signal by rectifying and filtering
 *  sample = input sample (16-bit)
 *  returns signal envelope (16-bit) */
#define env_coeff 4000                    /* / 32768  envelope filter parameter */
int envelope = 0;                         /*current sample of signal envelope (32-bit) */
short detect_envelope(short sample)
{
  short word1,word2;
  if (sample < 0)
          sample = -sample;                                   /* rectify the signal */
  word1 = envelope >> 15;                                     /* high-order word */
  word2 = envelope & 0x00007fff;                              /* low-order word */
  envelope = (word1 * (32768 - env_coeff)) +
                        ((word2 * (32768 - env_coeff)) >> 15) +
                        sample * env_coeff;
/* envelope =envelope*(1-coeff) + sample*coeff */
  return envelope >> 15;
}
```

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 Using C and the TMS320C6713 DSK*", John Wiley & Sons Inc., New Jersey, 2005.

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

[4] Kamran Khan "*DSP Signal Generation Implementation On C6713 DSK",* Communication Laboratory, University of Kassel,2009

[5] Texas Instruments "*TMS320C6713 Code Composer Studio Tutorial", Literature Number:SPRU301C,2000*

[6] Lawrence Rabiner "Fundamentals of Speech Recognition," Prentice Hall International, 1993.