SPEAKER RECOGNITION USING TMS320C6713DSK

Project submitted in partial fulfillment of requirements

For the Degree of

BACHELOR OF ENGINEERING

BY

SNEHA HEGDE

AMRUTA PENDHARKAR

PRATHAMESH PEWEKAR

ANIRUDDHA SATOSKAR

Under the guidance of

Internal Guide

Prof. K. T. TALELE

Department of Electronics Engineering

Sardar Patel Institute of Technology

University of Mumbai

2008-2009

BHARTIYA VIDYA BHAVAN'S

SARDAR PATEL INSTITUTE OF TECHNOLOGY

MUNSHI NAGAR, ANDHERI (W),

MUMBAI - 400058.

2008-09

CERTIFICATE OF APPROVAL

This is to certify that the following students

SNEHA HEGDE

AMRUTA PENDHARKAR

PRATHAMESH PEWEKAR

ANIRUDDHA SATOSKAR

have successfully completed and submitted the project entitled

"SPEAKER RECOGNITION USING TMS320C6713 DSK"

towards the fulfillment of Bachelor of Engineering course in Electronics of the Mumbai University

_____ _____

Internal Examiner **External Examiner**

_____ _____ _____

Internal Guide **Head of Department** **Principal**

ACKNOWLEDGEMENTS

We extend our heartfelt gratitude to our project guide Professor K. T. Talele, for his overwhelming support during the entire phase of our project. His apt guidance was instrumental in us achieving our goal. He has also left no stone unturned to ensure us with lab facilities throughout the year.

We would like to thank our college Sardar Patel Institute of Technology for providing us with all the resources that we required to go through the various phases of the project. Lastly, we are thankful to the entire staff of the Electronics Department for helping us make our project successful.

ABSTRACT

The technology of the automatic speech recognition is in full grow, a multitude of algorithms have been developed to improve the performance and robustness of ASR (Automatic Speech Recognition) systems. Automatic Speech recognition systems are increasingly widespread and used in very different acoustic conditions, and by very different speakers. The use of Mel frequency cepstral coefficients (MFCCs) for music information retrieval is one of the standard methods used in ASR systems. This paper describes a method to generate and process the Speech signal in digital domain using Texas Instruments' TMS320C6713 DSK.

Our aim is to develop software to recognize the speech samples from different users so as to restrict access to a predefined set of users. For this purpose, we form a database of different speech samples. The MFCCs for a particular Speech signal is unique for every individual. Therefore every such signal will generate different MFCCs. These are then compared with the previously stored MFCCs of signals to check if any match is found. For real time processing of Speech signal, fast processors like Digital Signal Processors are required.

INDEX

LIST OF FIGURES

1. INTRODUCTION

1.1 Development in the Digital Signal Processors Domain

The world of science and engineering is filled with signals: images from remote space probes, voltages generated by the heart and brain, radar and sonar echoes, seismic vibrations, and countless other applications. Digital Signal Processing is the science of using computers to understand these types of data. This includes a wide variety of goals: filtering, speech recognition, image enhancement, data compression, neural networks, and much more. DSP is one of the most powerful technologies that will shape science and engineering in the twenty-first century. [1]

Prior to the advent of stand-alone DSP chips, most DSP applications were implemented using bit slice processors. In 1978, Intel released the 2920 as an "analog signal processor". It had an on-chip ADC/DAC with an internal signal processor, but it did not have a hardware multiplier and was not successful in the market. In 1979, AMI released the S2811. It was designed as a microprocessor peripheral, and it had to be initialized by the host. In 1980 the first stand-alone, complete DSPs – the NEC µPD7720 and AT&T DSP1 – were introduced.

In 1983, Texas instruments launched its first DSP. [2] It was based on the Harvard architecture, and so had separate instruction and data memory. It already had a special instruction set, with instructions like load-and-accumulate or multiply-and-accumulate. It could work on 16-bit numbers and needed 390ns for a multiply-add operation. About five years later, the second generation of DSPs began to spread. They had 3 memories for storing two operands simultaneously and included hardware to accelerate tight loops, they also had an addressing unit capable of loop-addressing.

The main improvement in the third generation was the appearance of application-specific units and instructions in the data path, or sometimes as coprocessors. These units allowed direct hardware acceleration of very specific but complex mathematical problems, like the Fourier-transform or matrix operations. The fourth generation is best characterized by the changes in the instruction set and the instruction encoding/decoding.

1.2 Modern DSPs

Modern signal processors yield better performance. This is due in part to both technological and architectural advancements like lower design rules, fast-access two-level cache, (E) DMA circuit and a wider bus system. Most DSPs use fixed-point arithmetic, because in real world signal processing the additional range provided by floating point is not needed, and there is a large speed benefit and cost benefit due to reduced hardware complexity. Floating point DSPs may be invaluable in applications where a wide dynamic range is required. Product developers might also use floating point DSPs to reduce the cost and complexity of software development in exchange for more expensive hardware, since it is generally easier to implement algorithms in floating point. Generally, DSPs are dedicated integrated circuits; however DSP functionality can also be realized using Field Programmable Gate Array chips. Embedded general-purpose RISC processors are becoming increasingly DSP like in functionality.

A Texas Instruments C6000 series DSP clocks at 1.2 GHz and implements separate instruction and data caches as well as an 8 MiB 2nd level cache, and its I/O speed is rapid thanks to its 64 EDMA channels. The top models are capable of as many as 8000 MIPS (million instructions per second), use VLIW (very long instruction word) encoding, perform eight operations per clock-cycle and are compatible with a broad range of external peripherals and various buses (PCI/serial/etc). The other major players in the market that manufacture high end DSPs are Freescale, Analog Devices, and NXP Semiconductors.

2. LITERATURE SURVEY

The Literature survey included the study of various DSP processors available for real time applications in the market. While Motorola has been advancing the processing power of the PowerPC, Texas Instruments has been introducing new members of its C6000 family that offer more speed and flexibility to an already impressive DSP portfolio.[3] Hence TMS320, the widely used product from Texas Instruments-the leading manufacturer of DSP's is an optimum choice for real-time applications.

Furthermore, when choosing a processor, a fundamental question to answer is whether the application can be best addressed using a fixed-point or a floating-point processor. The Texas Instruments C6000 series of DSPs are available in both fixed- and floating-point varieties. For instance, in the C6201 and C6203, all eight functional units are fixed-point. In the C6701, six of the eight units are floating point. Because of their lower cost and power, fixed-point processors are best suited for high volume, heavily embedded applications. For fixed-point processors, the additional code complexity required for scaling may be offset by the lower cost of the silicon. Floating-point processors are best for applications that require extensive floating- point arithmetic, or in custom applications where the code is likely to change and the user can exploit the faster development effort. [4]

The software aspect of this project revolves around the premier code development tool the Code Composer Studio, a complete code development environment that runs on Windows workstations. It provides a highly flexible application development environment which suits the varying needs of real-time applications. With digital signal processing fast expanding its reach, subject matter related to this field is available in abundance. While working on this project we have studied matter from various sources such as books, IEEE papers, online articles and reference manuals. The knowledge gained from this activity has been of great help to us in understanding the basic concepts related to our project and has only ignited further interest in this topic.

3. HARDWARE COMPOSITION OF THE KIT

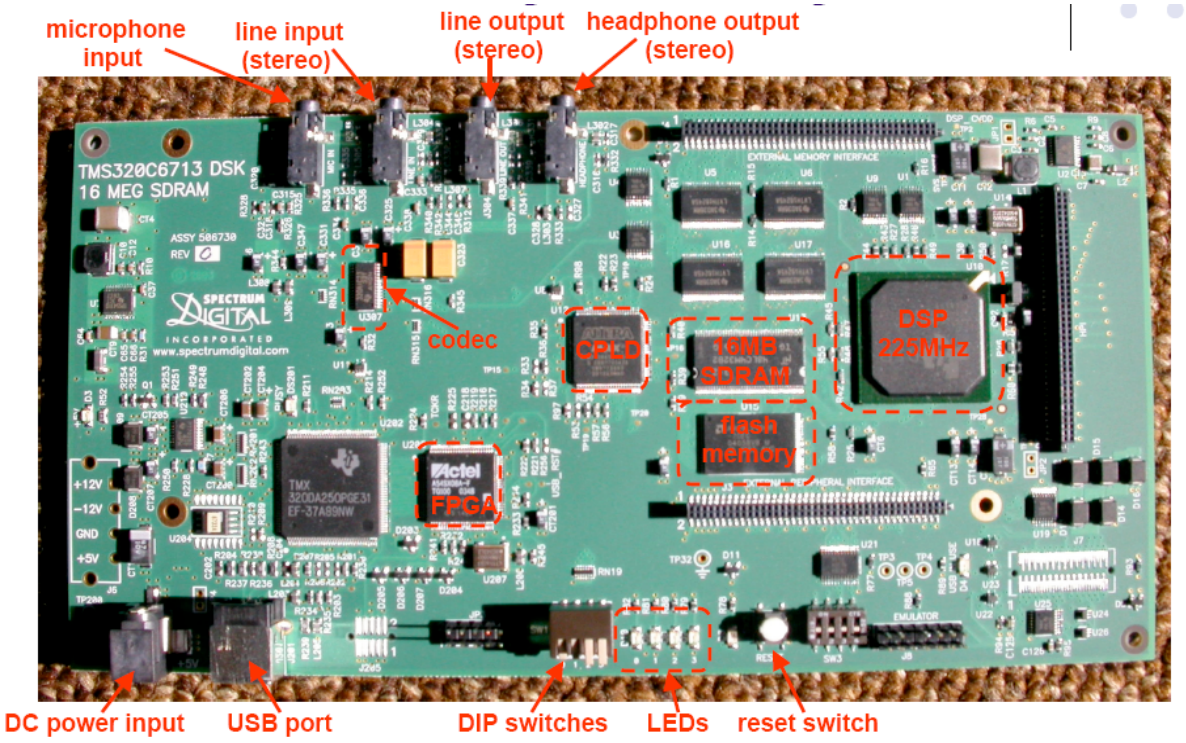
Before moving on to the actual application of the DSP processor it is necessary to understand the DSP processor's block diagram and the function of each component. Therefore let us familiarize ourselves with the TMS320C6713.

3.1 About DSK C6713

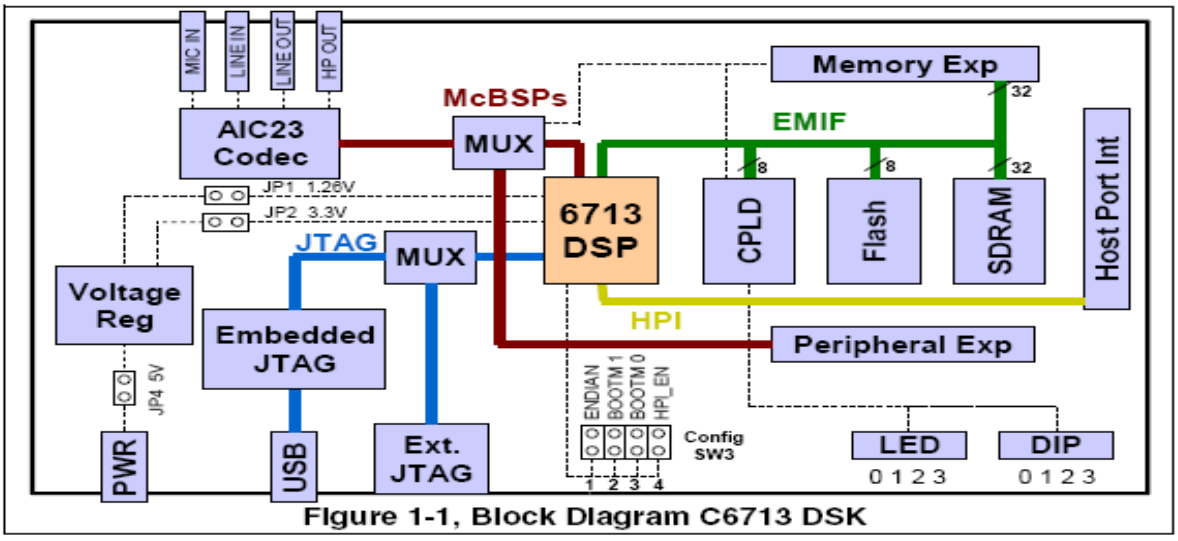
The C6713 DSK builds on TI's industry –leading line of cost easy to use DSP Starter Kit (DSK) development boards. The performance board features the TMS320C6713 floating point DSP. Capable of performing 1350 million floating –point operations per second (MFLOPS), the C6713 DSP makes the most powerful DSK development board. The DSK also serves as a hardware reference design for the TMS320C6713 DSP. Schematics, logic equations and application notes are available to ease hardware development and reduce time to market. [5]

The DSK starter kit includes the following hardware items:

TMS320C6713 DSK	TMS320C6713 DSK development board
Other hardware	External 5V DC power supply, IEEE 1284 compliant male-to-female cable
CD-ROM	Code Composer Studio DSK tools



3.1 SYSTEM LAYOUT OF TMS320C6713



3.2 BLOCK DIAGRAM OF C6713 DSK

3.2 Features of DSK C6713

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

- **A Texas Instruments TMS320C6713 DSP**

  The kit has Highest Performance Floating Signal Processor (DSP) which executes Eight 32-bit Instructions/cycle operating at 225 MHz. It has rich peripheral set which is optimized for Audio. It supports programming languages like C/C++.

- **An AIC23 stereo codec**

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

- **16 Mbytes of synchronous DRAM**

- **512 Kbytes of non-volatile Flash memory (256 Kbytes usable in default configuration)**

- **4 user accessible LEDs and DIP switches**

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

- **Software board configuration through registers implemented in CPLD**

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

- **Configurable boot options**

- **Standard expansion connectors for daughter card use**

- **JTAG emulation through on-board JTAG emulator with USB host interface or external emulator.**

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

- **Single voltage power supply (+5V)**

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

3.3 CPU (DSP core) description

The TMS320C6713B floating-point digital signal processor is based on the C67x CPU. The CPU fetches advanced very-long instruction words (VLIW) (256 bits wide) to supply up to eight 32-bit instructions to the eight functional units during every clock cycle. The VLIW architecture features controls by which all eight units do not have to be supplied with instructions if they are not ready to execute. The first bit of every 32-bit instruction determines if the next instruction belongs to the same execute packet as the previous instruction, or whether it should be executed in the following clock as a part of the next execute packet. Fetch packets are always 256 bits wide; however, the execute packets can vary in size.

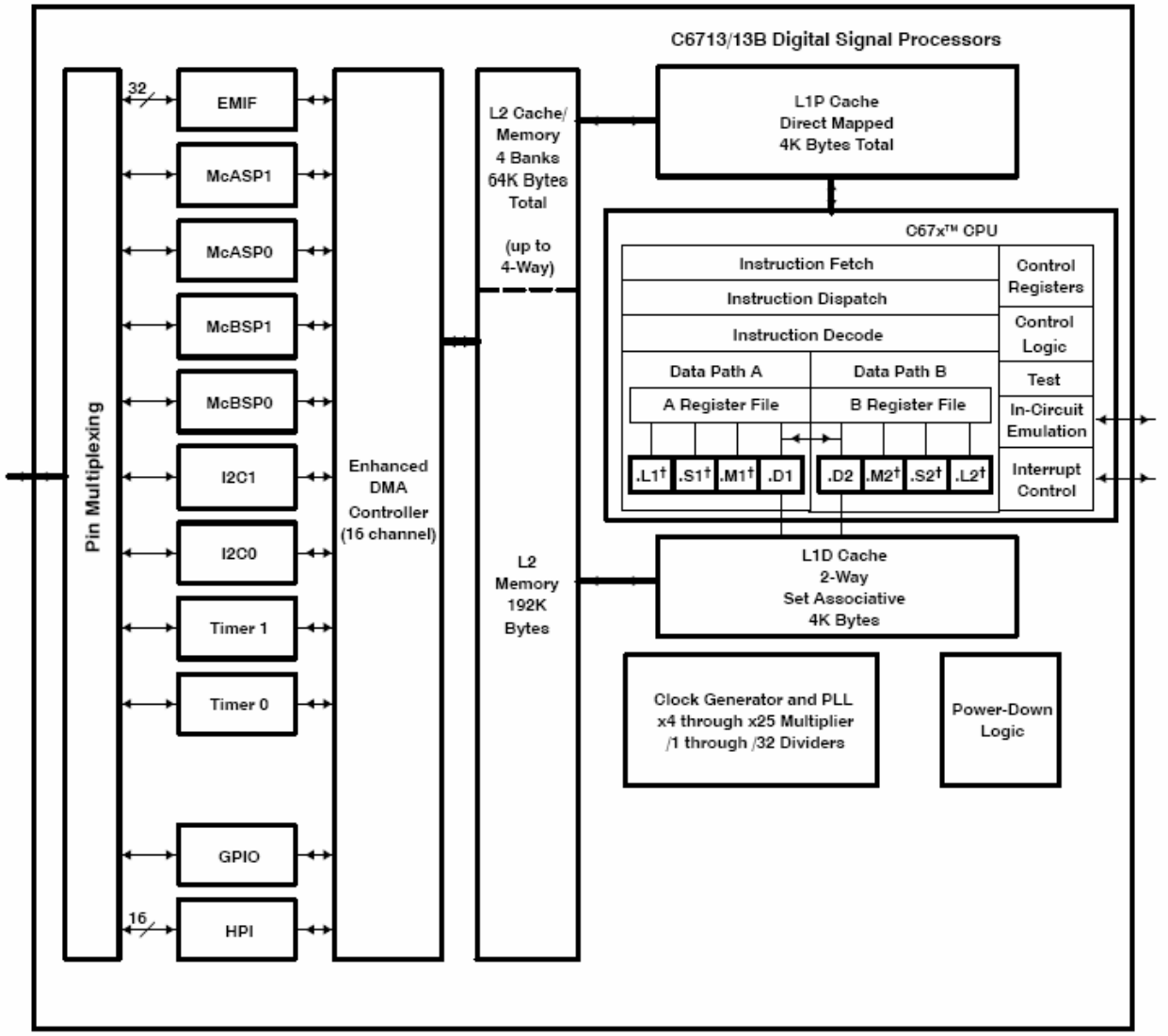
The variable-length execute packets are a key memory-saving feature, distinguishing the C67x CPU from other VLIW architectures. The CPU features two sets of functional units. Each set contains four units and a register file. One set contains functional units L1, .S1, .M1, and .D1; the other set contains units .D2, .M2, .S2, and .L2. The two register files each contain 16 32-bit registers for a total of 32 general-purpose registers. The two sets of functional units, along with two register files, compose sides A and B of the CPU (see the functional block and CPU diagram and Figure 1). The four functional units on each side of the CPU can freely share the 16 registers belonging to that side. Additionally, each side features a single data bus connected to all the registers on the other side, by which the two sets of functional units can access data from the register files on the opposite side. While register access by functional units on the same side of the CPU as the register file can service all the units in a single clock cycle, register access using the register file across the CPU supports one read and one write per cycle. [6]

The C67x CPU executes all C62x instructions. In addition to C62x fixed-point instructions, the six out of eight functional units (.L1, .S1, .M1, .M2, .S2, and .L2) also execute floating-point instructions. The remaining two functional units (.D1 and .D2) also execute the new LDDW instruction which loads 64 bits per CPU side for a total of 128 bits per cycle.

Another key feature of the C67x CPU is the load/store architecture, where all instructions operate on registers (as opposed to data in memory). Two sets of data-addressing units (.D1 and .D2) are responsible for all data transfers between the register files and the memory. The data address driven by the .D units allows data addresses generated from one register file to be used to load or store data to or from the other register file. The C67x CPU supports a variety of indirect addressing modes using either linear- or circular-addressing modes with 5- or 15-bit offsets. All instructions are conditional, and most can access any one of the 32 registers. Some registers, however, are singled out to support specific addressing or to hold the condition for conditional instructions (if the condition is not automatically “true”). The two .M functional units are dedicated for multiplies. The two .S and .L functional units perform a general set of arithmetic, logical, and branch functions with results available every clock cycle. The processing flow begins when a 256-bit-wide instruction fetch packet is fetched from a program memory. [6]

The 32-bit instructions destined for the individual functional units are “linked” together by “1” bits in the least significant bit (LSB) position of the instructions. The instructions that are “chained” together for simultaneous execution (up to eight in total) compose an execute packet. A “0” in the LSB of an instruction breaks the chain, effectively placing the instructions that follow it in the next execute packet. If an execute packet crosses the fetch-packet boundary (256 bits wide), the assembler places it in the next fetch packet, while the remainder of the current fetch packet is padded with NOP instructions. The number of execute packets within a fetch packet can vary from one to eight. Execute packets are dispatched to their respective functional units at the rate of one per clock cycle and the next 256-bit fetch packet is not fetched until all the execute packets from the current fetch packet have been dispatched. After decoding, the instructions simultaneously drive all active functional units for a maximum execution rate of eight instructions every clock cycle. While most results are stored in 32-bit registers, they can be subsequently moved to memory as bytes or half-words as well. All load and store instructions are byte-, half-word, or word-addressable. [6]

functional block and CPU (DSP core) diagram



† In addition to fixed-point instructions, these functional units execute floating-point instructions.

EMIF interfaces to:
–SDRAM
–SBSRAM
–SRAM,
–ROM/Flash, and
–I/O devices

McBSPs interface to:
–SPI Control Port
–High-Speed TDM Codecs
–AC97 Codecs
–Serial EEPROM

McASPs interface to:
–I2S Multichannel ADC, DAC, Codec, DIR
–DIT: Multiple Outputs

3.3 CPU CORE ARCHITECTURE OF C6713 DSK

Now that we have understood the working of the DSP processor let us move ahead to the actual applications from the following chapter onwards.

4. SOFTWARE USED TO ACCESS THE KIT

In order to communicate with the DSK, we use a software ambiance called the Code Composer Studio.

4.1 OVERVIEW OF CODE COMPOSER 3.1

Code Composer Studio (CCS) allows us to write a program in C language that can be used to initialize the DSK. Through CCS, we can initialize various ports and registers of the DSK. Code Composer provides a rich debugging environment that allows stepping through the code, set breakpoints, and examining the registers as the code is getting executed.

The Code Composer Studio (CCS) application provides an integrated environment with the capabilities like Integrated development environment with an editor, debugger, project manager, and profiler, C/C++ compiler, assembly optimizer and linker, Simulator, Real-time operating system (DSP/BIOS™), Real-Time Data Exchange (RTDX™) between the Host and the Target, and Real-time analysis and data visualization.

CCStudio integrated development environment includes host tools and target software that slashes development time and optimizes the performance for all real-time embedded DSP applications. Some of the Code Composer Studio's host side tools include TMS320 DSPs and OMAP Code, Drag and Drop CCStudio setup utility, Component manager support for multiple versions of DSP/BIOS and code generation tools within the IDE, Source Code Debugger common interface for both simulator and emulator targets, Connect/Disconnect; robust, resilient host to target connection, Application Code Tuning Dashboard, RTDX ™ data transfer for real time data exchange between host and target, Data Converter Plug-in to auto configure support for Texas Instruments Mixed Signal products, Quick Start tutorials and Help.

Code Composer Studio's target software includes DSP/BIOS ™ Kernel for the TMS320 DSPs, TMS320 DSP Algorithm Standard to enable software reuse, Chip Support Libraries to simplify device configuration, and DSP Libraries for optimum DSP functionality.

4.2 Installation of Code Composer Studio

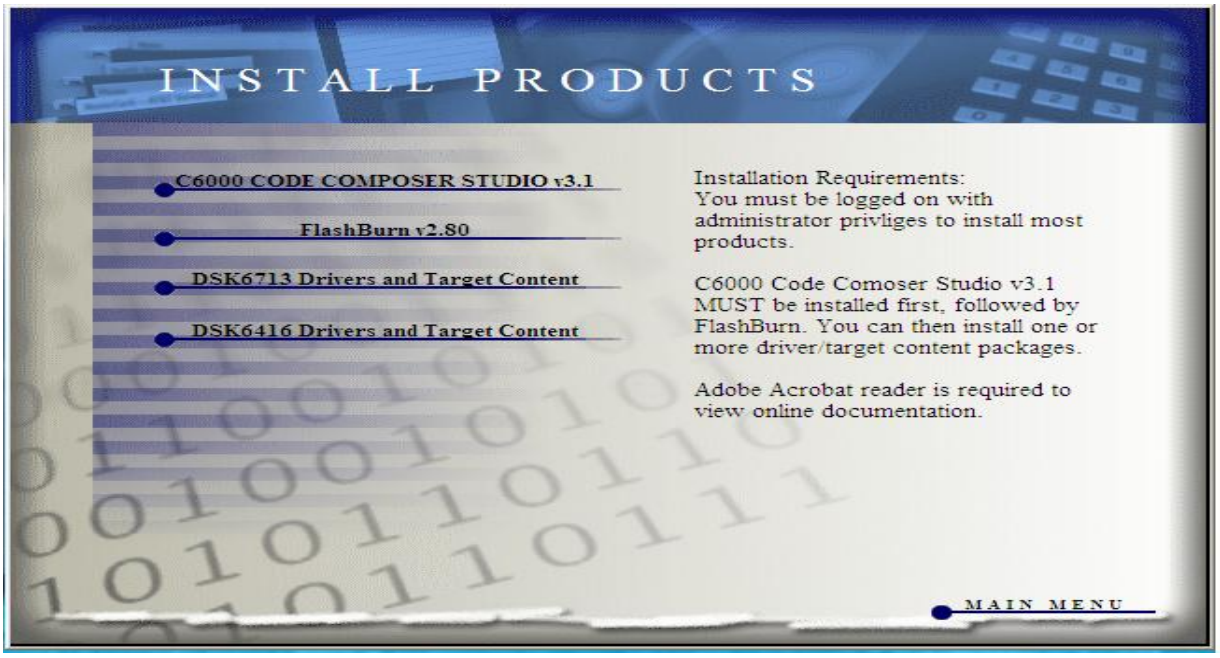
- The Code Composer Studio installation CD is included in the kit.

- The CD auto-runs to open a main menu dialog box.

4.1 MAIN MENU DIALOG BOX

- Click on ‘ Install Products’ . It will open another window with 4 options. Click on C600 CODE COMPOSER STUDIO v3.1.

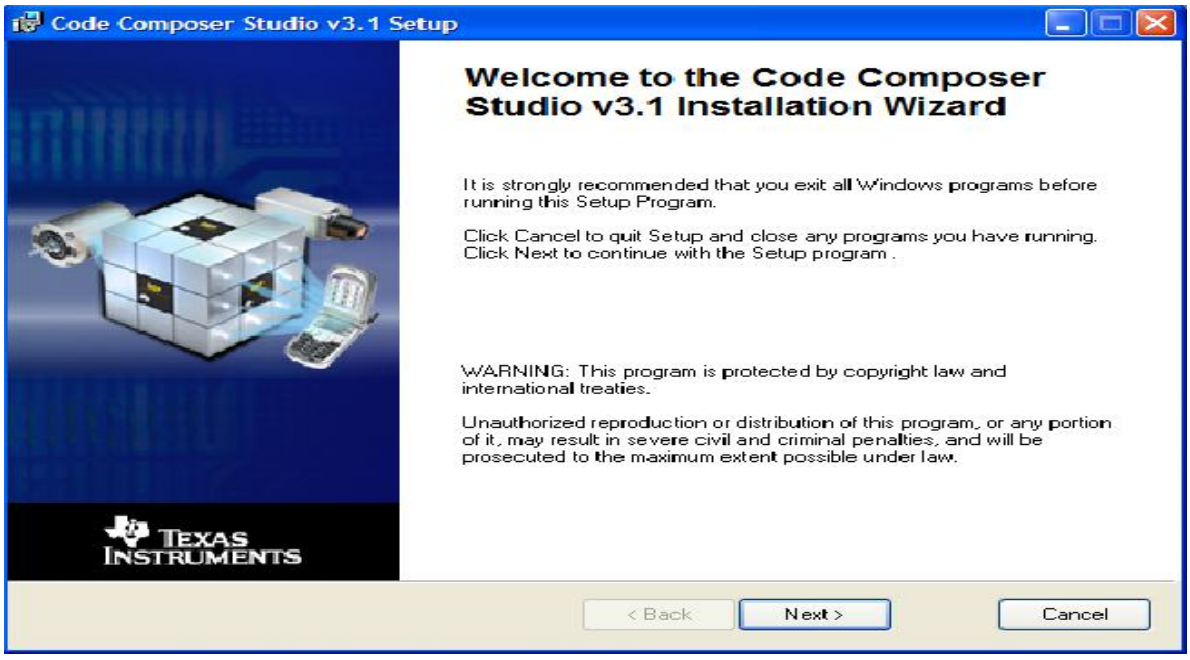
4.2 INSTALLATON SCREEN



- Click on C600 CODE COMPOSER STUDIO v3.1

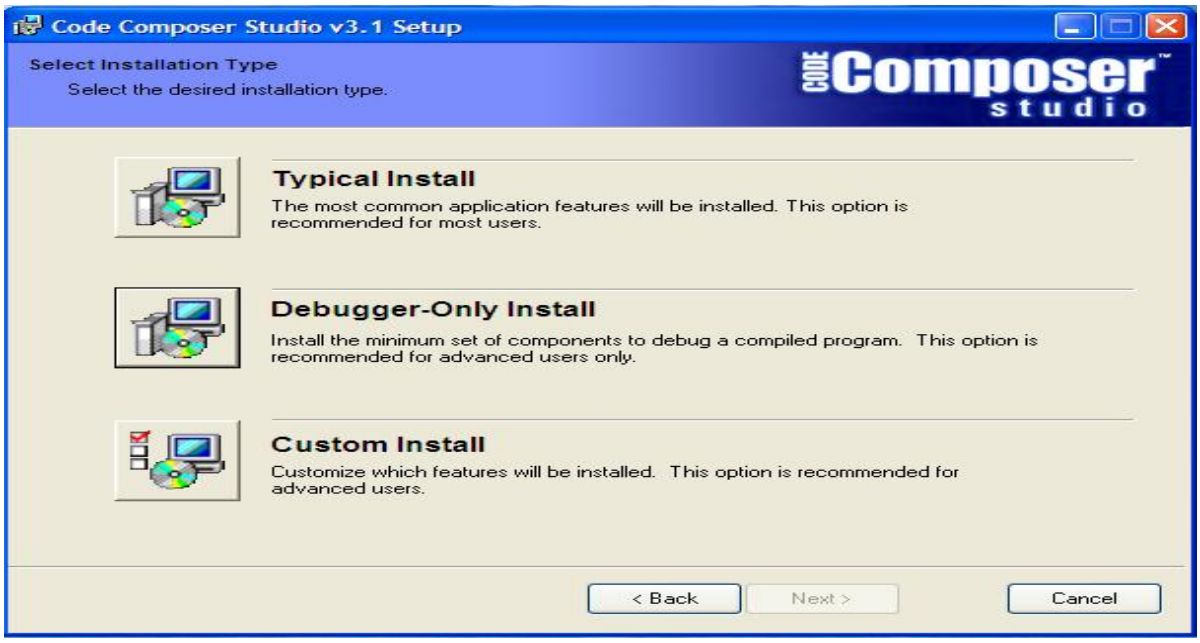
- Click on next.

4.3 WELCOME SCREEN



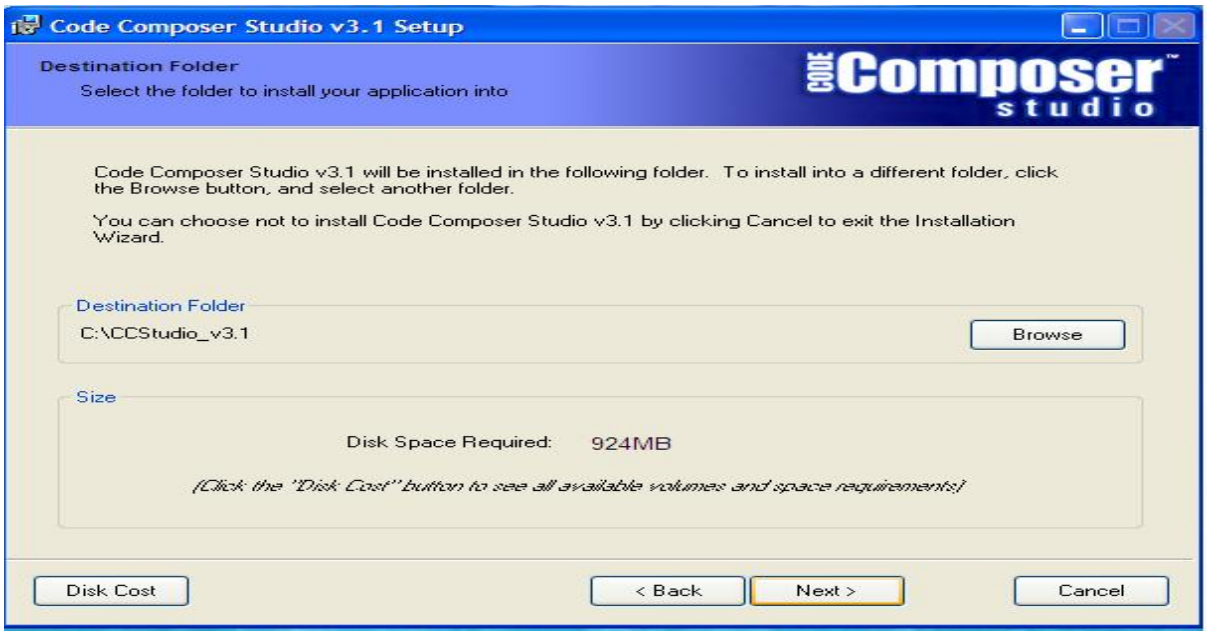
- The setup will ask for the type of install. A ‘typical’ install is recommended.

4.4 CUSTOMIZE INSTALLATION



- The rest of the steps are interactive and depend on user's choice.

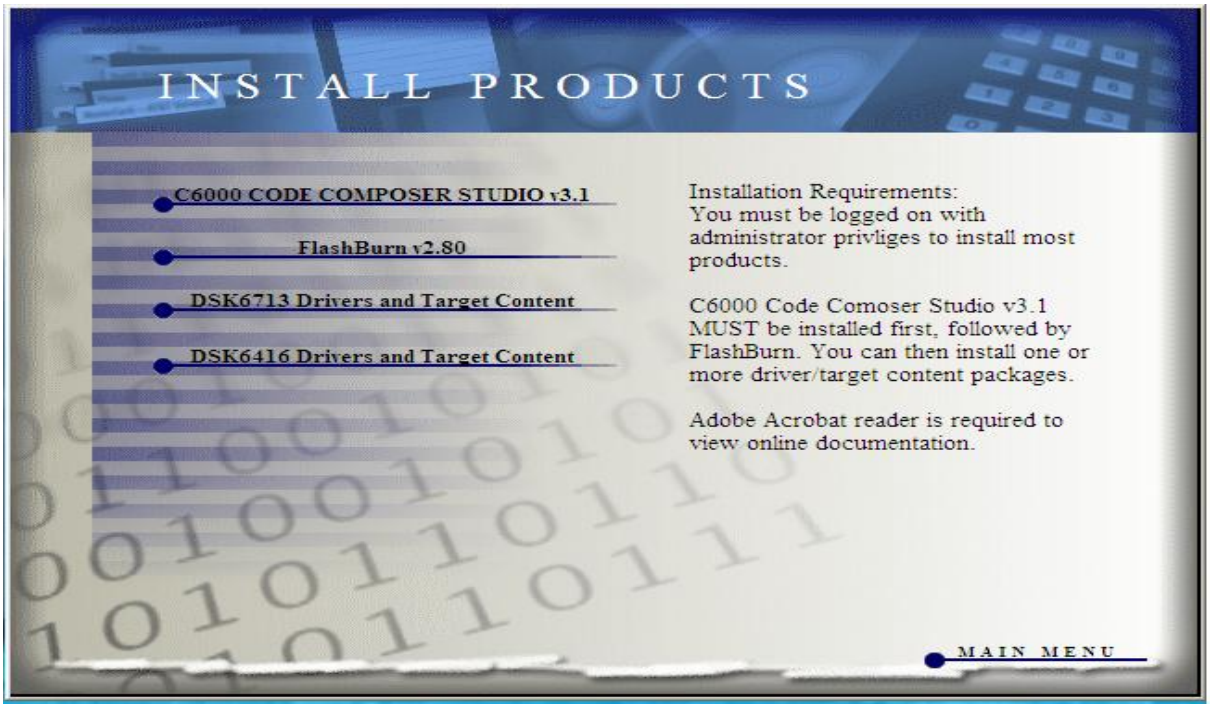
4.5 INSTALLATION LOCATION



4.6 INSTALLATION IN PROGRESS

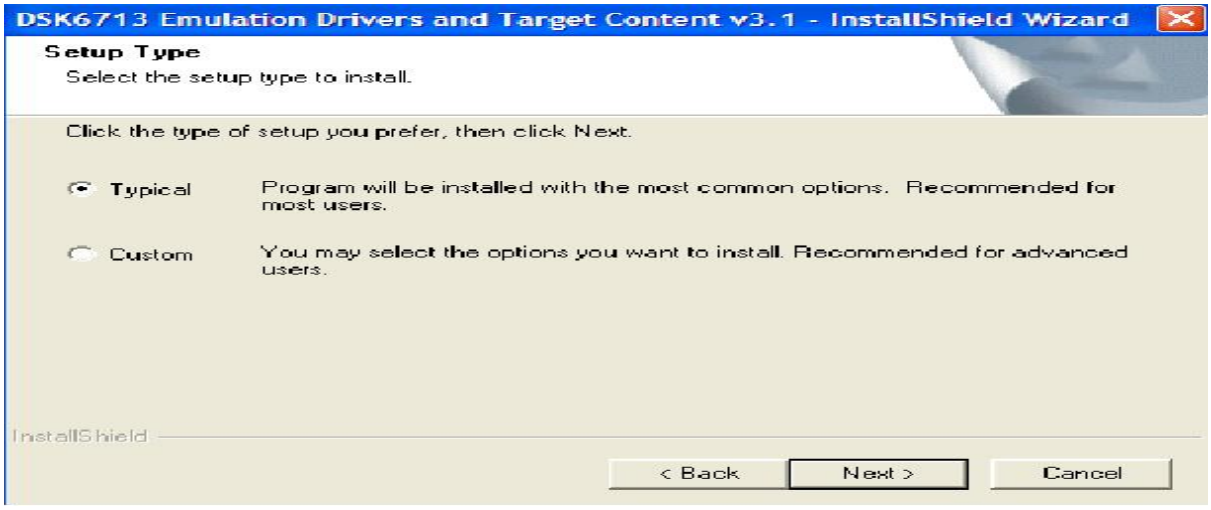
- After the CCS installation is complete, you are directed back to the main menu. Once you are there, click on the on ‘DSK 6713 Drivers and Target Content’.

4.7 DSK 6713 DRIVERS AND TARGET CONTENT



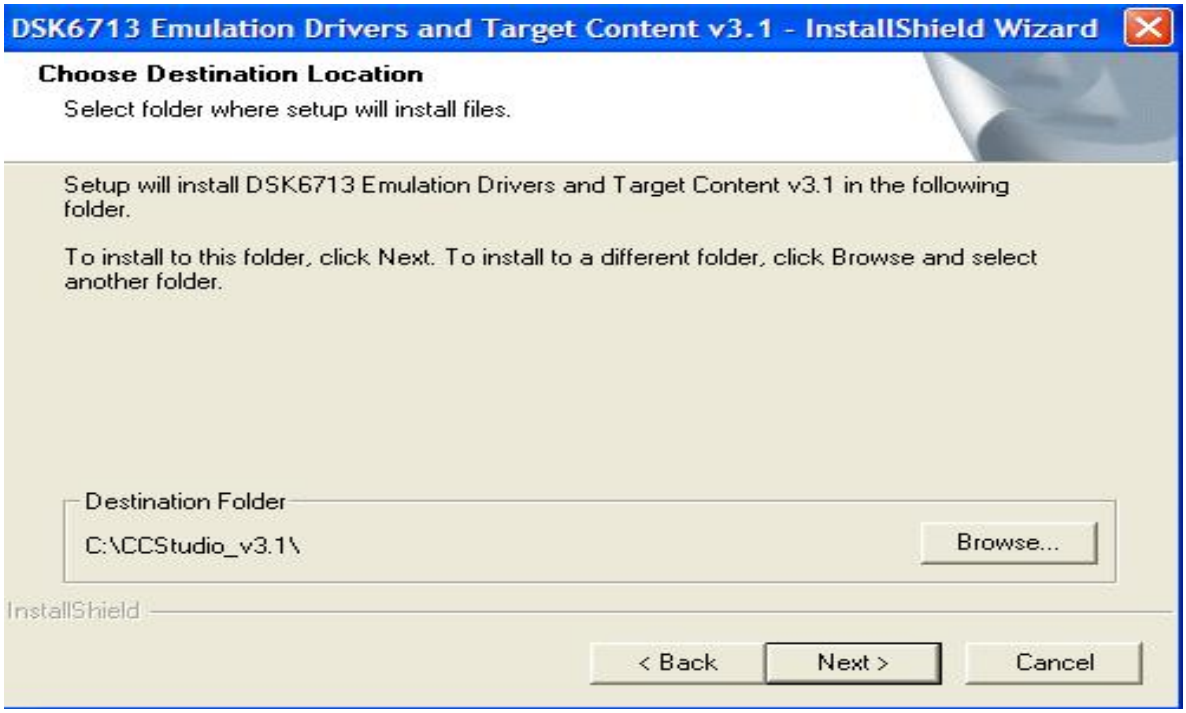
- Again you are prompted to enter type of installation. A ‘typical’ install is recommended.

4.8 INSTALLATION WIZARD



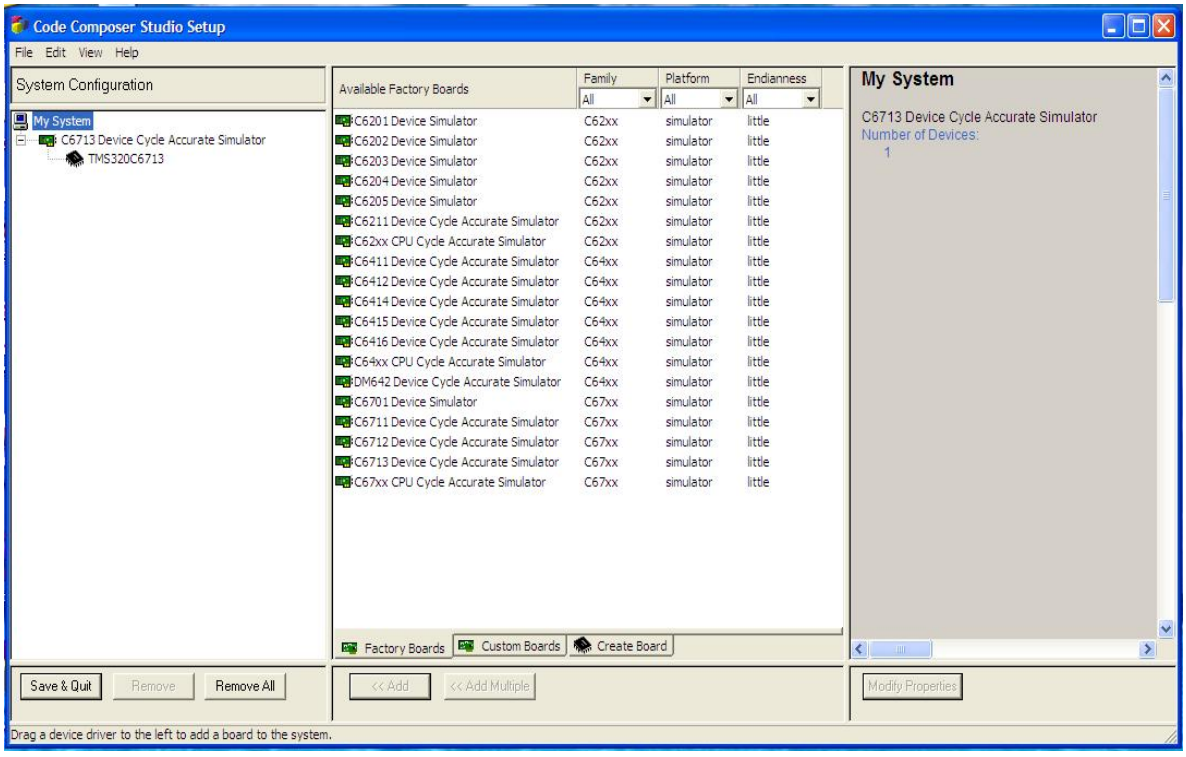
- Select directory and finish the installation.

4.9 DESTINATION FOLDER



- In order to interface the target device (DSK) with the computer using CCS, we have to open Code Composer Studio setup and select DSK 6713. Then we can proceed further.

4.9 TARGET DEVICE CONNECTION

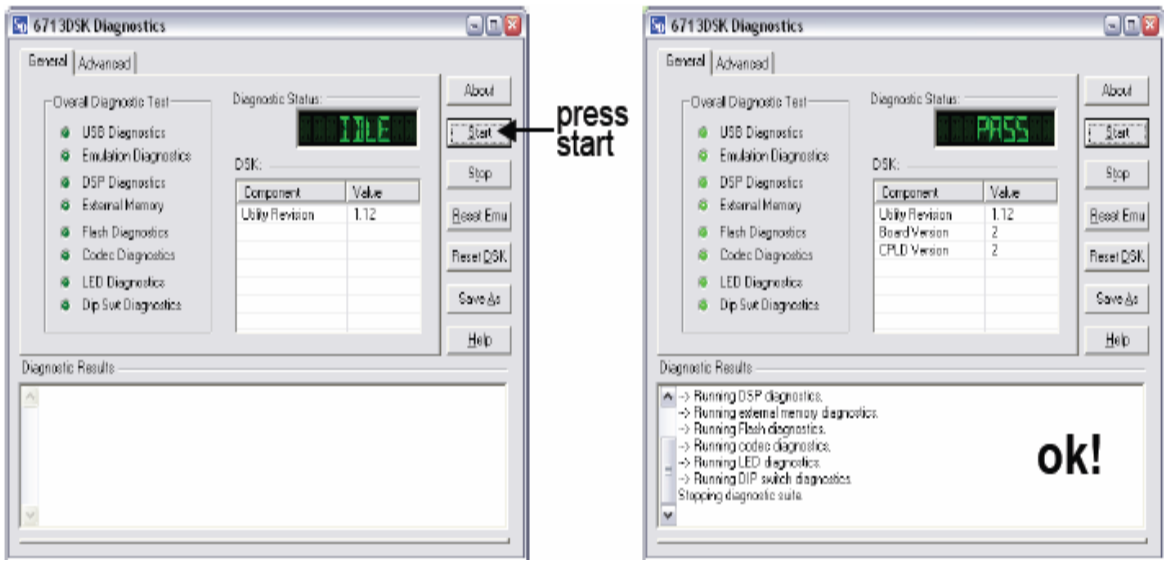


Note that before you install the DSK software; make sure the PC has a USB port and an operating system (Windows 98SE/2000/XP) that supports USB. For Windows 2000 and XP you must install Code Composer Studio using Administrator privileges. To run CCS on these systems requires write permission on the registry.

4.3 Testing Your Connection

If you want to test your DSK and USB connection you can launch the C6713 DSK Diagnostic Utility from the icon on your desktop.

From the diagnostic utility, press the start button to run the diagnostics. In approximately 20 seconds all the on-screen test indicators should turn green. [7]

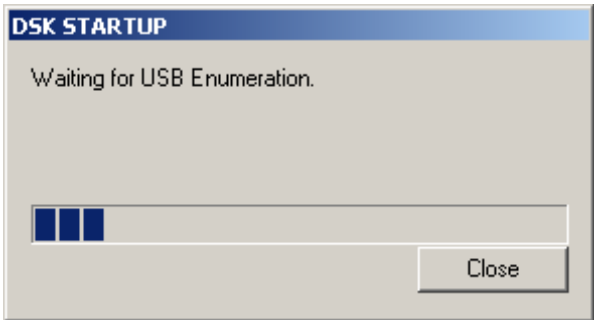


4.10 TESTING CONNECTION OF THE DSK

4.4 **Starting Code Composer**

To start Code Composer Studio, double click the C6713DSK CCS icon on your desktop.

The following window will appear when launching CCS or the Diagnostic Utility indicating the enumeration status**.** [8]



5. SPEAKER RECOGNITION

Speech is one of the natural forms of communication. Recent development has made it possible to use this in the security system. In speaker identification, the task is to use a speech sample to select the identity of the person that produced the speech from among a population of speakers. In speaker verification, the task is to use a speech sample to test whether a person who claims to have produced the speech has in fact done so. [9] This technique makes it possible to use the speakers' voice to verify their identity and control access to services such as voice dialing, banking by telephone, telephone shopping, database access services, information services, voice mail, security control for confidential information areas, and remote access to computers.

5.1 Traditional Algorithms Used for Speech Recognition

Acoustic modeling and language modeling are important parts of modern statistically-based speech recognition algorithms. Hidden Markov models (HMMs) are widely used in many systems. Language modeling has many other applications such as smart keyboard and document classification. Modern general-purpose speech recognition systems are generally based on Hidden Markov Models. These are statistical models which output a sequence of symbols or quantities. One possible reason why HMMs are used in speech recognition is that a speech signal could be viewed as a piecewise stationary signal or a short-time stationary signal. That is, one could assume in a short-time in the range of 10 milliseconds, speech could be approximated as a stationary process. Speech could thus be thought of as a Markov model for many stochastic processes. Dynamic time warping is an algorithm for measuring similarity between two sequences which may vary in time or speed.

5.2 Principles of Speaker Recognition

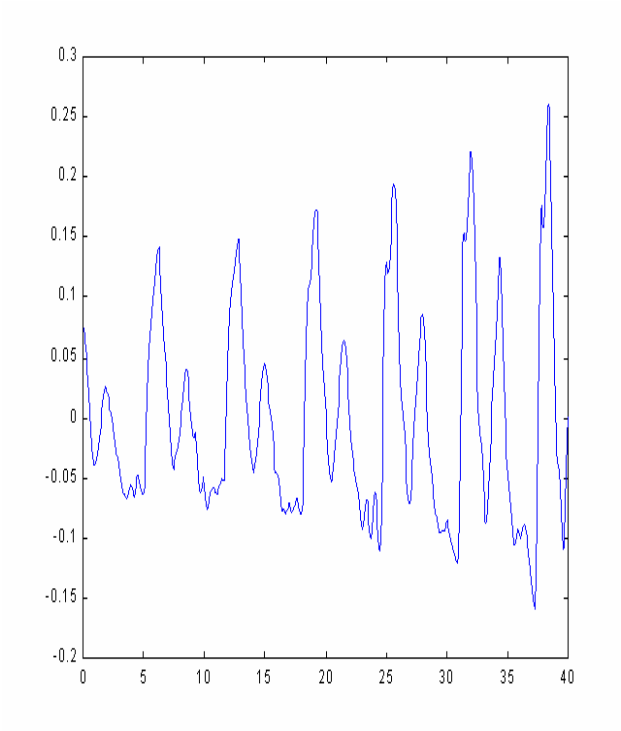
Speaker recognition methods can be divided into text-independent and text-dependent methods. In a text-independent system, speaker models capture characteristics of somebody's speech which show up irrespective of what one is saying. In a text-dependent system, on the other hand, the recognition of the speaker's identity is based on history her speaking one or more specific phrases, like passwords, card numbers, PIN codes, etc. Every technology of speaker recognition, identification and verification, whether text-independent and text dependent, each has its own advantages and disadvantages and may require different treatments and techniques. The choice of which technology to use is application-specific. At the highest level, all speaker recognition systems contain two main modules feature extraction and feature matching. [10]

Speech recognition systems can be characterized by many parameters. An isolated-word speech recognition system requires that the speaker pause briefly between words, whereas a continuous speech recognition system does not. Spontaneous, or extemporaneously generated, speech contains disfluencies, and is much more difficult to recognize than speech read from script. Some systems require speaker enrollment---a user must provide samples of his or her speech before using them, whereas other systems are said to be speaker-independent, in that no enrollment is necessary. Some of the other parameters depend on the specific task. Recognition is generally more difficult when vocabularies are large or have many similar-sounding words. When speech is produced in a sequence of words, language models or artificial grammars are used to restrict the combination of words.

6. SPEECH FEATURE EXTRACTION PROCESS

6.1 INTRODUCTION

The purpose of this module is to convert the speech waveform to some type of parametric representation (at a considerably lower information rate) for further analysis and processing. This is often referred as the *signal-processing front end*. The speech signal is a slowly timed varying signal (it is called *quasi-stationary*). An example of speech signal is shown below.



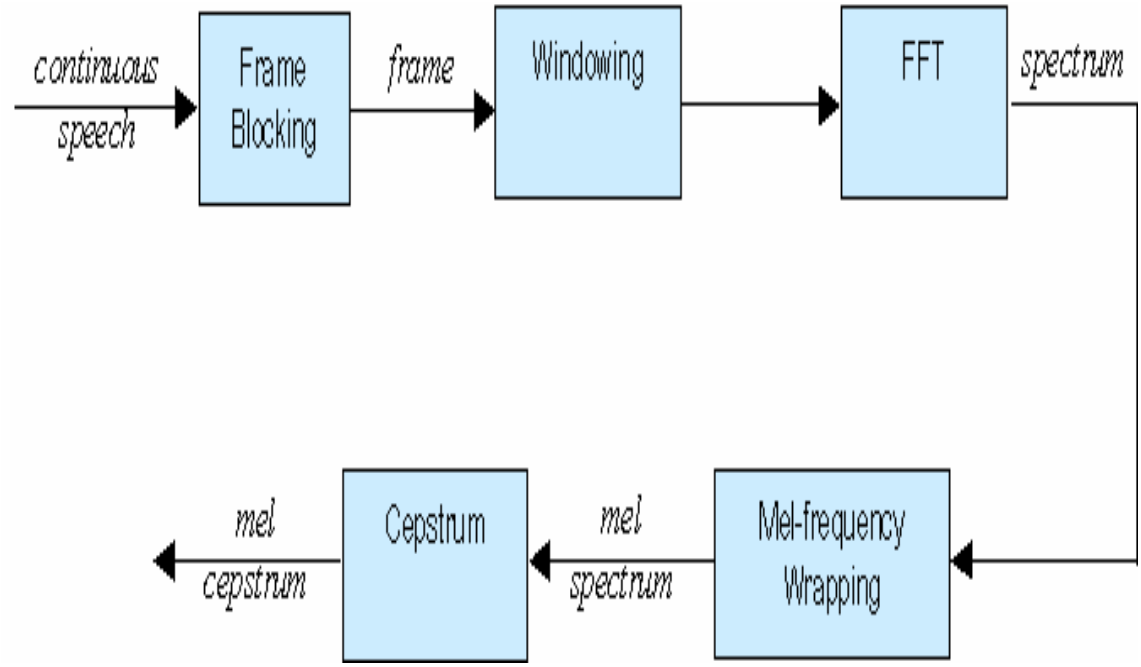
6.1 Example Of Speech Signal

When examined over a sufficiently short period of time (between 5 and 100 msec), its characteristics are fairly stationary. However, over long periods of time (on the order of 1/5 seconds or more) the signal characteristic change to reflect the different speech sounds being spoken. Therefore, *short-time spectral analysis* is the most common way to characterize the speech signal. [11]

6.2 THE "MFCC" PROCESSOR

MFCC's are based on the known variation of the human ear's critical bandwidths with frequency, filters spaced linearly at low frequencies and logarithmically at high frequencies have been used to capture the phonetically important characteristics of speech. This is expressed in the *mel-frequency* scale, which is a linear frequency spacing below 1000 Hz and a logarithmic spacing above 1000 Hz.

A block diagram of the structure of an MFCC processor is given in figure below.



6.2 Block Diagram of the MFCC Processor

The speech input is typically recorded at a sampling rate above 10000 Hz. This sampling frequency was chosen to minimize the effects of *aliasing* in the analog-to-digital conversion. These sampled signals can capture all frequencies up to 5 kHz, which cover most energy of sounds that are generated by humans. [12]

As been discussed previously, the main purpose of the MFCC processor is to mimic the behavior of the human ears. In addition, rather than the speech waveforms themselves, MFFC's are shown to be less susceptible to mentioned variations.

6.2.1 Framing

The sound signal is sampled and the sampled data is stored in an array. The size of the buffer varies depending upon the time for which the input sound signal is taken. Hence the length of the real-time sound signal is variable. So, before performing FFT on this data, it is necessary to split the data into uniform frames on which the FFT could be performed.

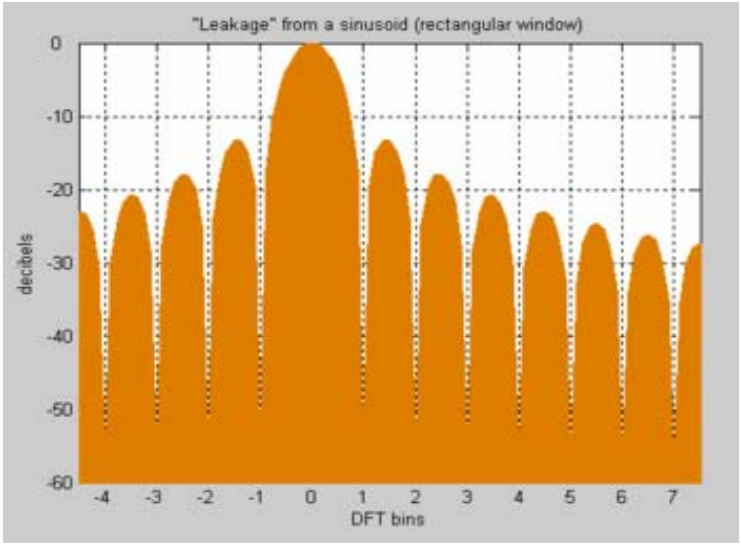
In this step the continuous speech signal is blocked into frames of N samples, with adjacent frames being separated by M ($M < N$). The first frame consists of the first N samples. This process continues until all the speech is accounted for within one or more frames. Typical values for N and M are $N = 256$ (which is equivalent to ~ 30 msec windowing and facilitate the fast radix-2 FFT) and $M = 100$.

6.2.2 Windowing

The next step in the processing is to window each individual frame so as to minimize the signal discontinuities at the beginning and end of each frame. The concept here is to minimize the spectral distortion by using the window to taper the signal to zero at the beginning and end of each frame. Once the data is framed, it is necessary to pass it through a window so as to reduce all spectral leakage.

6.2.2.1 Spectral leakage

The frequency spectrum of a 1000 Hz sine (or cosine) wave consists of a single sharp line. However, sine waves of other frequencies do not in general have such "clean" spectra.



6.3 Leakage in the Sinusoid

This spreading out of spectral energy across several frequency "channels" is called *spectral leakage*. Spectral leakage affects any frequency component of a signal which does not exactly coincide with a frequency channel. Since the frequency components of an arbitrary signal are unlikely to satisfy this requirement, spectral leakage is more likely to occur than not with real-life sampled signals.

6.2.2.1 Cause of spectral leakage

Spectral leakage occurs when a frequency component of a signal does not slot exactly into one of the frequency channels in the spectrum computed using the discrete Fourier transform. These "frequency channels", the frequencies represented by lines in the spectrum, are exact integer multiples (harmonics) of the fundamental frequency $1/Nh$. A sine wave with a frequency coinciding with one of these frequency channels has the property that you can fit an exact integer number of sine wave cycles into the complete sample length of the sampled signal. (The number of cycles is just the harmonic number).

If there is a mismatch, the sudden jump or discontinuity created by the pattern mismatch gives rise to the spurious components in the spectrum of the signal, causing a particular frequency component of the signal to appear not as a single sharp line but as a spread of frequencies, roughly centered around where the frequency component should be located, somewhere between the two nearest frequency channels either side.

The "real life" signals are not simple sine waves. Likewise, a speech waveform contains many components of different frequencies, and it is extremely unlikely that there will be a smooth match at the beginning and end of the sampled signal. Spectral leakage is therefore almost certainly going to affect the spectrum of any signal of practical interest. [13]

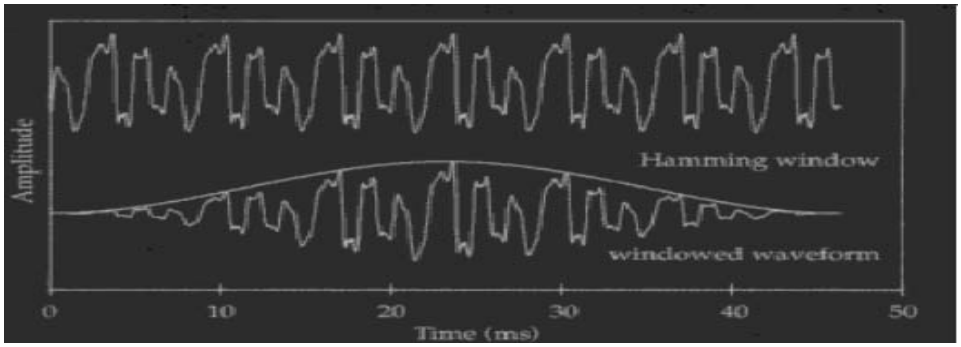
6.2.2.3 Reducing spectral leakage

The only way to avoid such leakage entirely would be to arrange that all the frequency components of the signal being examined coincide exactly with frequency channels in the computed spectrum. This, however, is impractical for an arbitrary signal containing many (usually unknown) frequency components.

While spectral leakage cannot in general be eliminated completely, its effects can be reduced. This is done by applying a window function to the sampled signal. The sampled values of the signal are multiplied by a function which tapers toward zero at either end, so that the sampled signal, rather than starting and stopping abruptly, "fades" in and out like some music CD tracks. This reduces the effect of the discontinuities where the mismatched sections of the signal join up and hence also the amount of leakage. [13]

6.2.2.4 Choice of window

Windowing addresses this spectral leakage problem by modifying the amplitudes of the waveform segment so that samples nearer the edges are low in amplitude and samples in the middle of the segment are at full amplitude. Computer programs often offer several window types for us to choose from. The two most common ones are "Hamming" and "Rectangular".



6.4 Hamming Window

The Hamming window reduces the amplitudes of the samples near the edges of the waveform chunk as illustrated in figure 1; whereas the rectangular window does not change the waveform samples at all. The Hamming window should be in the conjunction with FFT analysis, and rectangular windowing with all other types of analysis, including autocorrelation pitch tracking, RMS amplitude, and LPC analysis.

If we define the window as $w(n), 0 \le n \le N-1$, where *N* is the number of samples in each frame, then the result of windowing is

The signal $y_l(n) = x_l(n)w(n), \quad 0 \le n \le N-1$

Typically the *Hamming* window is used, which has the form:

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right), \quad 0 \le n \le N-1$$

6.2.3 Fast Fourier Transform

The next processing step is the Fast Fourier Transform, which converts each frame of *N* samples from the time domain into the frequency domain. The Fast Fourier transform (FFT) is a discrete Fourier transform algorithm which reduces the number of computations needed for Npoints from 2N2 to 2NlgN, where lg is the base-2 logarithm. FFTs were first discussed by Cooley and Tukey (1965), although Gauss had actually described the critical factorization step as early as 1805 (Bergland 1969, Strang 1993). The FFT is a fast algorithm to implement the Discrete Fourier Transform (DFT) which is defined on the set of *N* samples {*xn*}, as follow:

$$X_n = \sum_{k=0}^{N-1} x_k e^{-2\pi j k n / N}, \qquad n = 0,1,2,\ldots,N-1$$

Note that we use *j* here to denote the imaginary unit, i.e. $j = \sqrt{-1}$. In general *Xn'*s are complex numbers. The resulting sequence {*Xn*} is interpreted as follows: the zero frequency corresponds to *n* = 0, positive frequencies $0 < f < F_s / 2$correspond to values$1 \le n \le N/2 - 1$, while negative frequencies $-F_s / 2 < f < 0$correspond to $N/2 + 1 \le n \le N - 1$. Here, *Fs* denotes the sampling frequency.

The result after this step is often referred to as *spectrum* or *periodogram*. A discrete Fourier transform can be computed using an FFT by means of the Danielson-Lanczos lemma if the number of points N is a power of two. If the number of points N is not a power of two, a transform can be performed on sets of points corresponding to the prime factors of N which is slightly degraded in speed. Base-4 and base-8 fast Fourier transforms use optimized code, and can be 20-30% faster than base-2 fast Fourier transforms.

Fast Fourier transform algorithms generally fall into two classes: decimation in time, and decimation in frequency. The Cooley-Tukey FFT algorithm first rearranges the input elements in bit-reversed order, then builds the output transform (decimation in time). The basic idea is to break up a transform of length N into two transforms of length N/2 using the identity sometimes called the Danielson-Lanczos lemma.

$$\sum_{n=0}^{N-1} a_n e^{-2\pi i n k/N} = \sum_{n=0}^{N/2-1} a_{2n} e^{-2\pi i (2n) k/N} + \sum_{n=0}^{N/2-1} a_{2n+1} e^{-2\pi i (2n+1) k/N}$$

$$= \sum_{n=0}^{N/2-1} a_n^{\text{even}} e^{-2\pi i n k/(N/2)} + e^{-2\pi i k/N} \sum_{n=0}^{N/2-1} a_n^{\text{odd}} e^{-2\pi i n k/(N/2)},$$

The easiest way to visualize this procedure is perhaps via the Fourier matrix.

6.2.4 Power Spectrum of the Signal

Speech is a real signal, but its FFT has both real and imaginary components. The power of the frequency domain is calculated by summing the square of the real and imaginary components of the signal to yield a real signal. The second half of the samples in the frame are ignored since they are symmetric to the first half (the speech signal being real.

For a given signal, the power spectrum gives a plot of the portion of a signal's power (energy per unit time) falling within given frequency bins. [14] "Power Spectra" answers the question "which frequencies contain the signal´s power?" The answer is in the form of a distribution of power values as a function of frequency, where "power" is considered to be the average of the signal. In the frequency domain, this is the square of FFT´s magnitude.

Power spectra can be computed for the entire signal at once (a "periodogram") or periodograms of segments of the time signal can be averaged together to form the "power spectral density".[15]

6.2.5 Mel-Frequency wrapping

Triangular filters are designed using the Mel frequency scale with a bank of filters to approximate the human ear. The power signal is then applied to this bank of filters to determine the frequency content across each filter. Twenty filters are chosen, uniformly spaced in the Mel-frequency scale between 0 and 4 kHz. The Mel-frequency spectrum is computed by multiplying the signal spectrum with a set of triangular filters designed using the Mel scale. For a given frequency *f*, the Mel of the frequency is given by

$$B(f) = [1125 \ln (1+f\,700)] \text{ mels}$$

If *m* is the Mel, then the corresponding frequency is

$$B\text{-}1(m) = [700 \exp (m1125) - 700] \text{ Hz}$$

The frequency edge of each filter is computed by substituting the corresponding Mel. Once the edge frequencies and the center frequencies of the filter are found, boundary points are computed to determine the transfer function of the filter. [16]

6.2.6 Conversion to Decibels

After calculating the Mel Frequency Coefficients, we scale them using the logarithmic scale. A logarithmic scale is a scale of measurement that uses the logarithm of a physical quantity instead of the quantity itself.

A reason for using the decibel is that different sound signals together produce very large range of sound pressures. Because the power in a sound wave is proportional to the square of the pressure, the ratio of the maximum power to the minimum power is in (short scale) trillions. We need to plot the power spectrum of these signals where we need to deal with such a range, so we choose this conversion to decibels.

After finding out the power spectrum, the log Mel spectrum has to be converted back to time. The result is called the Mel frequency cepstrum coefficients (MFCCs). The cepstral representation of the speech spectrum provides a good representation of the local spectral properties of the signal for the given frame analysis. Because the Mel spectrum coefficients are real numbers (and so are their logarithms), they may be converted to the time domain using the Discrete Cosine Transform (DCT).

6.2.7 Discrete Cosine Transform

The final stage in extracting MFCC feature vectors is to apply a discrete cosine transform (DCT). The DCT serves two purposes. First, the DCT performs the final part of a cepstral transformation which separates the slowly varying spectral envelope (or vocal tract) information from the faster varying speech excitation. Lower order coefficients represent the slowly varying vocal tract while higher order coefficients contain excitation information. For speech recognition, vocal tract information is more useful for classification than excitation information. Therefore, to create the final MFCC vector, the output vector from the DCT is truncated to retain only the lower order coefficients. The second purpose of DCT is to decorrelate the elements of the feature vector making it suitable for diagonal covariance matrix statistical classifiers.

6.2.8 The Mel-frequency Cepstral Coefficients (MFCC)

In this final step, the log Mel spectrum is converted back to time. The result is called the Mel frequency cepstrum coefficients (MFCC). The cepstral representation of the speech spectrum provides a good representation of the local spectral properties of the signal for the given frame analysis. The Mel-frequency cepstrum (MFC) is one of the non-linear speech analysis methods in automatic speech recognition. Mel-frequency cepstral coefficients (MFCCs) are coefficients that collectively make up an MFC. They are derived from a type of cepstral representation of the audio clip (a nonlinear "spectrum-of-a-spectrum"). [17]

Because the Mel spectrum coefficients (and so their logarithm) are real numbers, we can convert them to the time domain using the Discrete Cosine Transform (DCT). Therefore if we denote those Mel power spectrum coefficients that are the result of the last step are $\tilde{S}_k, k = 1,2,\ldots,K$, we can calculate the MFCC's, $\tilde{c}_n$, as

$$\tilde{c}_n = \sum_{k=1}^{K} (\log \tilde{S}_k) \cos\left[n\left(k - \frac{1}{2}\right)\frac{\pi}{K}\right], \qquad n = 1,2,\ldots,K$$

Note that the first component is excluded, $\tilde{c}_0$, from the DCT since it represents the mean value of the input signal which carried little speaker specific information.

The number of Mel cepstrum coefficients, K, is typically chosen as 20. The first component is excluded from the DCT since it represents the mean value of the input signal which carries little speaker specific information. By applying the procedure described above, for each speech frame of about 30ms, a set of Mel-frequency cepstrum coefficients is computed. This set of coefficients is called an *acoustic vecto*r. These acoustic vectors can be used to represent and recognize the voice characteristic of the speaker. Therefore each input utterance is transformed into a sequence of acoustic vectors.

7. IMPLEMENTATION OF THE PROJECT

The aim of this project is to determine the identity of the speaker from the speech sample of the speaker and the trained vectors. Trained vectors are derived from the speech sample of the speaker at a different time.

First the input analog speech signal is digitized at 8 KHz sampling frequency using the on board ADC (Analog to Digital Converter). The Speech sample is stored in a one-dimensional array. The speech signal is split into frames. Each frame consists of 128 Samples of Speech signal. Speech sample in one frame is considered to be stationary.

After Framing, to prevent the spectral leakage we apply windowing. Here Hamming window with 128 coefficients is used.

Third step is to convert the Time domain speech Signal into Frequency Domain using Discrete Fourier Transform. Here Fast Fourier Transform is used. The resultant transformation will result in a signal being complex in nature. Speech is a real signal but its Fourier Transform will be a complex one (Signal having both real and imaginary).

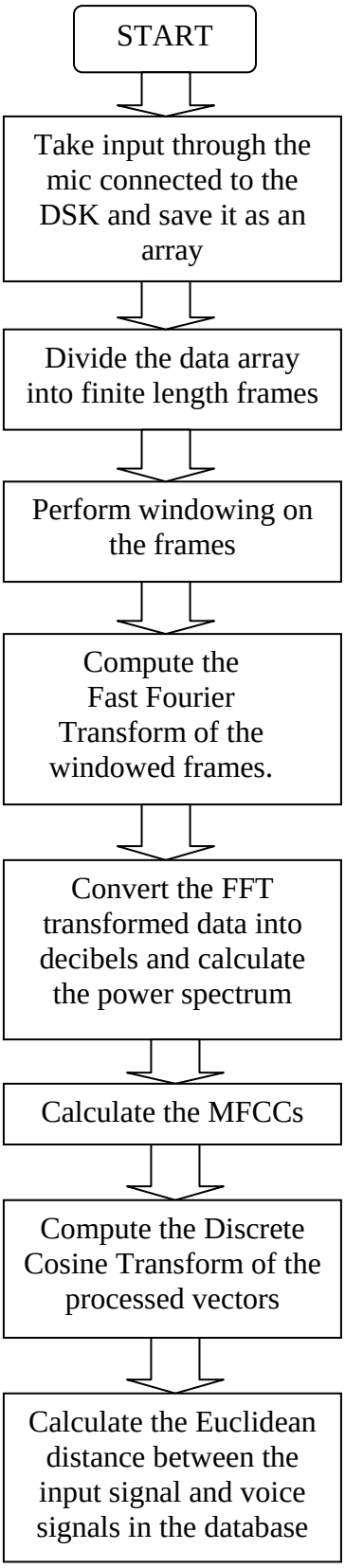
The power of the signal in Frequency domain is calculated by summing the square of Real and Imaginary part of the signal in Frequency Domain. The power signal will be a real one.

Triangular filters are designed using Mel Frequency Scale. This bank of filters will approximate our ears. The power signal is then applied to this bank of filters to determine the frequency content across each filter. In our implementation we choose total number of filters to be 20. These 20 filters are uniformly spaced in Mel Frequency scale between 0-4KhZ.

After computing the Mel-Frequency Spectrum, log of Mel-Frequency Spectrum is computed. Discrete Cosine Transform of the resulting signal will result in the computation of the Mel-Frequency Cepstral Co-efficient.

Euclidean distance between the trained vectors and the Mel-Frequency Cepstral Coefficients are computed for each trained vectors. The trained vector that produces the smallest distance will be identified as the speaker.

7.1 Flowchart of the Program



8. SOFTWARE USED IN THE PROJECT

8.1 Code for Training

```
#include "DSK6713_loopcfg.h"
#include "dsk6713.h"
#include "dsk6713_aic23.h"
#include "stdio.h"
#include "c6713dskinitmic.h"
#include<stdio.h>
#include<math.h>
#include<stdlib.h>

int rand_int(void);
#define N 65536                  //large buffer size
#define PI 3.14159
#define column_length 128          // Frame Length of the one speech signal
#define row_length 100             // Total number of Frames in the given speech signal
#define Number_Of_Filters 20       // Total Number of Filters
Uint32 xL;
long i,k,j,g,c,z;
int program_control=0;
```

```
//Generic Structure to represent real and imaginary part of a signal
struct complex
{
float real;
float imag;
};
//Structure to store the input speech sample
struct buffer
{
struct complex data[row_length][column_length];
};
//Structure to store the Mel-Frequency Co-efficients
struct mfcc
{
float data[row_length][Number_Of_Filters];
};
short buffer1[N];
#pragma DATA_SECTION(buffer1,".EXTRAM")
//real_buffer is used to store the input speech.
struct buffer real_buffer;
#pragma DATA_SECTION(real_buffer,".EXTRAM")
/Codec data handle structure
DSK6713_AIC23_CodecHandle hCodec;
```

```
double MFCC_Y[row_length][Number_Of_Filters];

float hamming_window[256] =
{
8.000000e-002,8.013963e-002,8.055844e-002,8.125618e-002,8.223242e-002,8.348657e-002,8.501786e-002,

8.682537e-002,8.890801e-002,9.126449e-002,9.389341e-002,9.679315e-002,9.996197e-002,1.033979e-001,

1.070989e-001,1.110628e-001,1.152870e-001,1.197691e-001,1.245063e-001,1.294957e-001,1.347344e-001,

1.402191e-001,1.459465e-001,1.519131e-001,1.581153e-001,1.645494e-001,1.712114e-001,1.780973e-001,

1.852029e-001,1.925239e-001,2.000559e-001,2.077942e-001,2.157342e-001,2.238711e-001,2.321999e-001,

2.407156e-001,2.494129e-001,2.582867e-001,2.673315e-001,2.765418e-001,2.859121e-001,2.954366e-001,

3.051097e-001,3.149253e-001,3.248775e-001,3.349604e-001,3.451677e-001,3.554933e-001,3.659309e-001,

3.764742e-001,3.871168e-001,3.978522e-001,4.086739e-001,4.195753e-001,4.305498e-001,4.415908e-001,

4.526915e-001,4.638452e-001,4.750452e-001,4.862846e-001,4.975566e-001,5.088543e-001,5.201710e-001,

5.314997e-001,5.428336e-001,5.541657e-001,5.654893e-001,5.767974e-001,5.880831e-001,5.993396e-001,

6.105602e-001,6.217378e-001,6.328659e-001,6.439376e-001,6.549462e-001,6.658850e-001,6.767474e-001,

6.875267e-001,6.982165e-001,7.088102e-001,7.193015e-001,7.296839e-001,7.399512e-001,7.500970e-001,


7.601153e-001,7.700000e-001,7.797450e-001,7.893445e-001,7.987927e-001,8.080837e-001,8.172119e-001,
```

8.261719e-001,8.349581e-001,8.435653e-001,8.519882e-001,8.602216e-001,8.682607e-001,8.761004e-001,

8.837362e-001,8.911632e-001,8.983771e-001,9.053733e-001,9.121478e-001,9.186964e-001,9.250150e-001,

9.310999e-001,9.369473e-001,9.425538e-001,9.479159e-001,9.530303e-001,9.578940e-001,9.625040e-001,

9.668575e-001,9.709518e-001,9.747846e-001,9.783533e-001,9.816560e-001,9.846905e-001,9.874550e-001,

9.899479e-001,9.921676e-001,9.941128e-001,9.957824e-001,9.971752e-001,9.982905e-001,9.991275e-001,

9.996858e-001,9.999651e-001,9.999651e-001,9.996858e-001,9.991275e-001,9.982905e-001,9.971752e-001,

9.957824e-001,9.941128e-001,9.921676e-001,9.899479e-001,9.874550e-001,9.846905e-001,9.816560e-001,

9.783533e-001,9.747846e-001, 9.709518e-001,9.668575e-001,9.625040e-001,9.578940e-001,9.530303e-001,

9.479159e-001,9.425538e-001,9.369473e-001,9.310999e-001,9.250150e-001,9.186964e-001,9.121478e-001,

9.053733e-001,8.983771e-001,8.911632e-001,8.837362e-001,8.761004e-001,8.682607e-001,8.602216e-001,

8.519882e-001,8.435653e-001,8.349581e-001,8.261719e-001,8.172119e-001,8.080837e-001,7.987927e-001,

7.893445e-001,7.797450e-001,7.700000e-001,7.601153e-001,7.500970e-001,7.399512e-001,7.296839e-001,

7.193015e-001,7.088102e-001,6.982165e-001,6.875267e-001,6.767474e-001,6.658850e-001,6.549462e-001,

6.439376e-001,6.328659e-001,6.217378e-001,6.105602e-001,5.993396e-001,5.880831e-001,5.767974e-001,

5.654893e-001,5.541657e-001,5.428336e-001,5.314997e-001,5.201710e-001,5.088543e-001,4.975566e-001,

4.862846e-001,4.750452e-001,4.638452e-001,4.526915e-001,4.415908e-001,4.305498e-001,4.195753e-001,

4.086739e-001,3.978522e-001,3.871168e-001,3.764742e-001,3.659309e-001,3.554933e-001,3.451677e-001,

3.349604e-001,3.248775e-001,3.149253e-001,3.051097e-001,2.954366e-001,2.859121e-001,2.765418e-001,

2.673315e-001,2.582867e-001,2.494129e-001,2.407156e-001,2.321999e-001,2.238711e-001,2.157342e-001,

2.077942e-001,2.000559e-001,1.925239e-001,1.852029e-001,1.780973e-001,1.712114e-001,1.645494e-001,

1.581153e-001,1.519131e-001,1.459465e-001,1.402191e-001,1.347344e-001,1.294957e-001,1.245063e-001,

1.197691e-001,1.152870e-001,1.110628e-001,1.070989e-001,1.033979e-001,9.996197e-002,9.679315e-002,

9.389341e-002,9.126449e-002,8.890801e-002,8.682537e-002,8.501786e-002,8.348657e-002,8.223242e-002,

8.125618e-002,8.055844e-002,8.013963e-002,8.000000e-002,

};

float H[Number_Of_Filters+2] =

{

0.0,2.349535731,4.945514224,7.813784877,10.98290838,

14.48444125,18.35324982,22.62785770,27.35082918,

32.56919306,38.33491098,44.70539514,51.74407917,

59.52105066,68.11374874,77.60773478,88.09754483,

99.68763091,112.4934010,126.6423682,142.2754203,0.0
};

/* Variable to store the vector of the speech signal */

```
float mfcc_vector[20];

//coeff is used to store the Mel-Frequency Spectrum
.
#pragma DATA_SECTION(coeff,".EXTRAM")

struct mfcc coeff;

//mfcc_ct is used to store the Mel-Frequency Cepstral Co-efficients.

#pragma DATA_SECTION(mfcc_ct,".EXTRAM")

struct mfcc mfcc_ct;

FILE *fptr;

float x[column_length],y[column_length];

/**********************FUNCTION DECLARATIONS**********************/

void function(struct buffer *);

void log_energy(struct mfcc *);

void mfcc_coeff(struct mfcc *, struct mfcc *);

void mfcc_vect(struct mfcc *, float *);

void mfcc(struct buffer *, struct mfcc *);

/******************** START OF MAIN ***********************/

void main()

{

// Initialize the board support library, must be called first

DSK6713_init();

// Start the codec

hCodec = DSK6713_AIC23_openCodec(1, &config);
```

```
//Set sampling frequency via the number before KHZ in the define.
//Choose from 8, 16, 24, 32, 44.1, 48, or 96 Khz.
DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_8KHZ);
DSK6713_DIP_init();
DSK6713_LED_init();

/************START OF INITIALIZING THE VARIABLES TO ZERO*********/


for ( i=0; i < row_length ; i++ )  /* Total Number of Frames */
        {
          for ( j = 0; j < column_length ; j++) /* Total Number of Samples in a Frame */
        {
          real_buffer.data[i][j].real = 0.0; /* Initializing real part to be zero */
          real_buffer.data[i][j].imag = 0.0; /* Initializing imaginary part to be zero*/
        }
        }
for ( i=0; i<row_length; i++)  /* Total Number of Frames */
                    {
                   for ( j=0; j<Number_Of_Filters; j++)  /* Total Number of Filters */
                    {
                    coeff.data[i][j] = 0.0; /* Initializing the co-effecient array */
                     mfcc_ct.data[i][j] = 0.0; /* Initializing the array for storing MFCC */
                    }
```

```
for(i=0;i<N;i++)
buffer1[i]=0;
for(i=0;i<column_length;i++)  /*- Initialization -*/
        {
         x[i]=0;
         y[i]=0;
        }

/************END OF INITIALIZING THE VARIABLES TO ZERO*************/

function(&real_buffer); //Function for obtaining input from user, framing and windowing
for(g=0;g<column_length;g++)
        {for(i=0;i<column_length;i++)
                {
                 x[i]= real_buffer.data[g][i].real;
                 y[i]=0;
                }
         z=column_length;
         fft(z,x,y);                    //FFT Function call for each frame
         for(i=0;i<column_length;i++)
                {
                 real_buffer.data[g][i].real=x[i];
                 real_buffer.data[g][i].imag=y[i];
```

```
        }
power_spectrum(&real_buffer); //Call power spectrum function

mfcc(&real_buffer,&coeff); //Mel Freq Spectrum of power spectrum

log_energy(&coeff); //Converting to Decibel Scale

mfcc_coeff(&mfcc_ct,&coeff); //Compute DCT

mfcc_vect(&mfcc_ct,mfcc_vector); //Compute mel Vector


/* Store the Vector in a Flat File */
fptr = fopen("train_vect.dat","w");
        fprintf(fptr, "{");
        for ( i =0; i < Number_Of_Filters ; i++)
                fprintf(fptr, "%f, ",mfcc_vector[i]);
        fprintf(fptr,"}");
        fclose(fptr);

printf("Thank you\n");
exit(0);
/*************************END OF MAIN*******************************/
```

```
/**********************START OF IO FUNCTION **********************/

void function(struct buffer *real_buffer)
{
printf("Press DIP switch3 and speak into mic\n");
while(1)
        {
         if(DSK6713_DIP_get(3) == 0)         //if SW#3 is pressed
        {
        DSK6713_LED_on(3);                   //turn on LED#3
        for (i = 0; i<N; i++)
        {
        if(DSK6713_DIP_get(3) == 0)
        while (!DSK6713_AIC23_read(hCodec, &xL));
        {
          buffer1[i] =xL;                    //input data
         if(DSK6713_DIP_get(3) == 1)
         break;
         }
         }

 k=i;
        DSK6713_LED_off(3);                  //LED#3 off when buffer full
```

```
 break;
            }
           }
c=0;
j=0;
for(i=0;i<k;i++)
           {
           real_buffer->data[c][j].real = ((float)buffer1[i])*hamming_window[j];
            j++;
            if(j>column_length-1)
                      {
                       j=0;
                      c++;
                      }
            if(c>row_length-1)
                      {
break;
                       }
                      }
return;
}
/********************* START OF FFT FUNCTION ************************/
fft( int n, float x[N],float y[N])
{
```

```
int i,j,t;
int n1,n2,l;
float a,c,s,e;
float xt,yt;
int q=n/2;
n2=n;
for(t=0; t<q; t++)
        {
         n1=n2;
         n2=n2/2;
         e=6.283185307179586/n1;
         for(j=0;j<n2;j++)
                 {
                  a = j*e;
                  c = cos(a);
                  s = -sin(a);

 for(i=j;i<n;i+=n1)
                          {
                           l=i+n2;
                           xt=x[i]-x[l];
                           x[i]=x[i]+x[l];
                           yt=y[i]-y[l];
```

```
                    y[i]=y[i]+y[l];
                     x[l] = xt*c - yt*s;
                     y[l] = xt*s + yt*c;
                    }
                   }
        }
bitreversal(n, x, y);
return;
}
/***************** START OF BIT REVERSAL FUNCTION ******************/
bitreversal(int n, float x[N], float y[N])
{
int i,j,p,n1;
j=0;
n1=z-1;
for(i=0;i<n1;i++)
        {
                float temp;
                if(i>=j) goto end;
                        temp=x[j];
                x[j]=x[i];
                x[i]=temp;
                temp=y[j];
```

```
                y[j]=y[i];
                y[i]=temp;
                end: p=n/2;
                kk:  if(p>j) goto  kt;
              j=j-p;

 p=p/2;
            goto kk;
          kt: j=j+p;
           }
 return(0);
}
/*************** FUNCTION TO COMPUTE POWER SPECTRUM *************/

power_spectrum(struct buffer *real_buffer)
{
for (i=0; i<row_length; i++)  /* For all the Frames */
        {
for ( j=0; j < column_length; j++)  /* For all the samples in one Frame */
                {
real_buffer->data[i][j].real =
(
(real_buffer->data[i][j].real)*(real_buffer->data[i][j].real)) +
((real_buffer->data[i][j].imag)*(real_buffer->data[i][j].imag)
);
                 }/* Compute Power (real)^2 + (imaginary)^2 */
```

```
        }
return;
}
/************ FUNCTION TO CONVERT TO DECIBEL SCALE*************/
void log_energy(struct mfcc *coeff)
{
for ( i=0; i<row_length; i++)
        {
for ( j=0; j<Number_Of_Filters; j++ )
                {
coeff->data[i][j] =  20*log(coeff->data[i][j]+1);
                }
        }
return;
}
/************** FUNCTION TO COMPUTE MEL COEFFICIENTS*************/

void mfcc(struct buffer *real_buffer, struct mfcc *coeff)

{
int F[22] =
{
0,100,200,300,400,500,600,700,800,900,1000,1149,1320,1516,1741,2000,2297,2639,303
1,3482,4000,4000
};
```

```
 int Fi_up,Fi_down;
 float delt_f = 8000.0 //column_length;
 float MFCC_k = 0;
 int delt_F_up, delt_F_down, v,w1,w2;
for(v=0;v<100;v++)  //For each frame
          {
           w1=0;
           MFCC_k=0;
          for(i = 1; i < 21; i ++)
{
coeff->data[v][i-1]=0;
           delt_F_up = F[i] - MFCC_k;
           Fi_up = (int)(((double) delt_F_up)/delt_f);
           delt_F_down = F[i+1] - (MFCC_k + (Fi_up+1)*delt_f);

Fi_down = (int)(((double) delt_F_down)/delt_f);
                    for(k = 0; k < Fi_up; k++)
                    {
coeff->data[v][i-1] += (MFCC_k - F[i-1]) * real_buffer->data[v][w1].real / (F[i] - F[i-1]);
MFCC_k += delt_f;
 w1++;
                    }
                     w2=w1;
```

```
   for(k = 0; k < Fi_down; k++)

{

coeff->data[v][i-1] += (F[i+1] - MFCC_k) * real_buffer->data[v][w2].real / (F[i+1] -
F[i]);

MFCC_k += delt_f;

 w2++;

                    }

 MFCC_k -= delt_f*(Fi_down-1);

                    }

}

return;

/********* FUNCTION TO COMPUTE DISCRETE COSINE TRANSFORM*******/


void mfcc_coeff(struct mfcc *mfcc_ct, struct mfcc *coeff)

{

for ( i=0; i<row_length; i++)  /* For all the frames (100 Frames) */

          {

for (j=0; j<Number_Of_Filters; j++ )  /* For all the filters */

                    {

 mfcc_ct->data[i][j] = 0.0;

 for ( k=0; k<Number_Of_Filters; k++)

                              {

mfcc_ct->data[i][j] = mfcc_ct->data[i][j] + coeff->data[i][k]*cos((double)((PI*j*(k-
1/2))/Number_Of_Filters));
```

```
 }
                    }
          }
 return;
}
/****FUNCTION TO COMPUTE DISTANCE AND CONVERSION TO VECTOR***/

void mfcc_vect(struct mfcc *mfcc_ct,float *mfcc_vector)
{
for ( i=0; i< Number_Of_Filters; i++ )
          {
mfcc_vector[i] = 0;
for (j=0; j< row_length; j++)

{
mfcc_vector[i] = mfcc_vector[i] + ((mfcc_ct->data[j][i]));
 }
          }
return;
}

/********************** END OF PROGRAM************************/
```

8.2 Code for Recognition of a trained user

While recognizing a trained speaker, the code needs to be appropriately modified. Instead of writing the generated vector to a file, we compare the generated vector with the already available vectors to find a match. The User Number is asked for, and the input voice sample is compared with the voice samples for that user.

```
for(i=0;i<Number_Of_Speakers;i++)
        {
if(training_vector[i][20]==code)
        {
        range=i;
        break;
                }
        }

if(i==Number_Of_Speakers)
        {
printf("Invalid Password\nAccess Denied");
exit(0);
        }

//Identifying the Speaker
for ( i=range; i<range+5; i++ )          // For the 5 samples of the identified user
        {
```

```
distance = 0.0;
  for ( j=1; j<Number_Of_Coefficients; j++ )
  {
distance = distance + abs(mfcc_vector[j]-training_vector[i][j]);
}
// Identify the speaker sample with least distance
            if ( distance < ref_distance )
                        {
                         speaker = i;
                         ref_distance = distance;
                        }
           }
/* Print the identified Speaker */
if(ref_distance<30000)                    //Threshold for Euclidean Distance
           {
                  if(speaker>=0&&speaker<5)
                     {
                      printf("Aniruddha Identified\n");
                     }
            if(speaker>=5&&speaker<10)
                        {
                         printf("Amruta Identified\n");
                        }
```

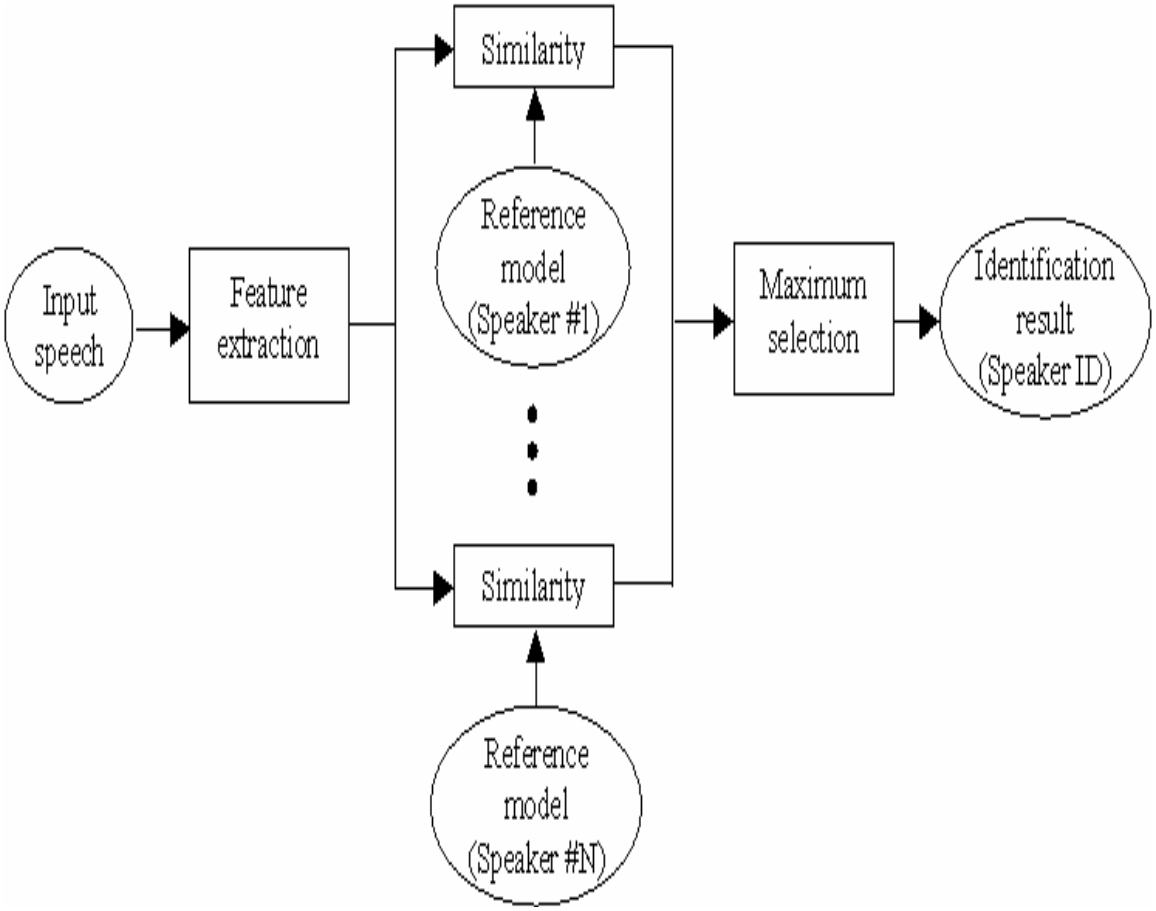
```
        if(speaker>=10&&speaker<15)
                {
                 printf("Kavita Identified\n");
                }
        if(speaker>=15&&speaker<20)
                {
                printf("Prathamesh Identified\n");
                }
        if(speaker>=20&&speaker<25)
                {
                 printf("Sneha Identified\n");
                }
/* Print the identified Sample */
        printf("Access Granted\n");
        }
else
printf("Invalid Password \nAccess Denied\n");
```

9. RESULT ANALYSIS

Part 1: Speaker Identification

Initially our project was speaker identification as the text password was not used. When a user gave an input voice sample, feature extraction process was performed on this input sample i.e the Mel frequency coefficients were computed. These were then compared with the reference models (i.e database of trained coefficients) for each speaker (speaker 1 to speaker N) . The selection was performed depending on the minimum difference between input sample and reference model. Thus speaker identification was performed. This process is summarized in figure below.

9.1 Speaker Recognition Model



For the speaker identification process we made a database of 5 users. Each user trained 5 times. Here there were three possibilities: Correct speaker is recognized, Incorrect Speaker is recognized or Access is denied because difference in the input sample and stored coefficients is very high. Each user was made to speak 10 times. The results are as shown below

User Number	Correctly Recognized	Incorrectly Recognized	Access Denied------ Prompt to try again
USER1	8	2	0
USER2	9	1	0
USER3	9	0	1
USER4	7	2	1
USER5	10	0	0
TOTAL=5	**TOTAL=43**	**TOTAL=5**	**TOTAL=2**

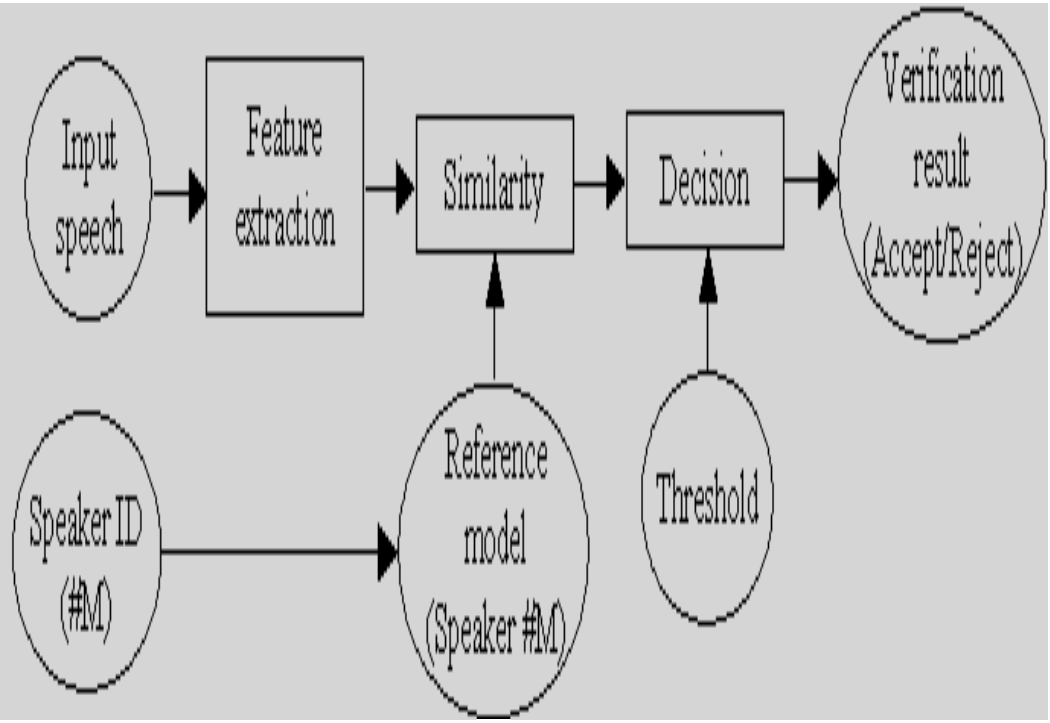
Considering the access denied cases as also correct results because it might be due to noise in the surroundings of the user, we can compute efficiency as:

$$\text{EFFICIENCY} = \frac{(\text{Total Samples} - \text{Incorrectly Recognized})}{(\text{Total Samples})}$$

Therefore efficiency achieved is 90%.

Part 2: Speaker Verification

The second part of our project was speaker verification as the text password has been used. Therefore initially the user is prompted for a text password i.e. User number. Then the user gives an input voice sample which is subjected to the feature extraction process i.e. the Mel frequency coefficients are computed. These are then compared with the reference model M selected using text password for only that speaker. The verification is then performed by comparing difference between the computed vectors for input sample and database vectors to an empirical threshold. Thus speaker verification is performed. This process is summarized in figure below.



9.2 Speaker Verification Model

For the speaker verification process also we made a database of 5 users. Each user trained 5 times. Here there were two cases: Correct speaker tries to access or incorrect speaker tries to access. In both cases there are two possibilities: Access is granted or

access is not granted. Each user was made to speak 10 times in his user ID and randomly 10 incorrect samples were taken for each user ID. The results are as shown below

User Number	Correct User		Incorrect User	
	Access granted	Access not granted	Access granted	Access not granted
USER1	10	0	0	10
USER2	9	1	1	9
USER3	8	2	1	9
USER4	10	0	0	10
USER5	10	0	0	10
TOTAL=5	**TOTAL=47**	**TOTAL=3**	**TOTAL=2**	**TOTAL=48**

Considering the access granted to incorrect user as incorrect results, we can compute efficiency as:

$$\text{EFFICIENCY} = \frac{(\text{Total Samples} - \text{access granted to incorrect user})}{(\text{Total Samples})}$$

Therefore efficiency achieved is 96%.

Thus efficiency achieved is higher for the speaker verification part as compared to speaker identification part for this project.

10. APPLICATIONS

The main application of speaker recognition is in security systems to identify a person. Thus access will granted only to the person who has permission granted by the administrator. The person's speech samples must first be included in the database by the administrator. Speaker recognition for access control can be extended to wide variety of applications ranging from voice dialing, banking by telephone, telephone shopping, database access services, Information services, voice mail, security control for confidential information areas, and remote access to computers. Recognition of speech can also be extended to speech to text converters. A widely used application of the recognition of spoken words is the voice tags application found in most new mobiles.

11. CONCLUSION

We have thereby, effectively implemented speaker identification and speaker verification using TMS320C6713 DSK. The results show that a high efficiency can be achieved for both purposes using this algorithm based on Mel Frequency Cepstral Coefficients. This speaker recognition module performs accurately as both a speaker-dependent and text -dependent system.

The results acquired by our system confirm that the use of Fourier Transform with MFCC parameterization is a very promising method in the Automatic Speaker Recognition field. For real time processing of Speech signal, fast processors like Digital Signal Processors are required. Therefore the TMS320C Digital Signal Processors provide an excellent platform for the development of speaker recognition modules due to involvement of complex Fourier analysis in their algorithm.

The cepstral representation of the speech spectrum provides a good representation of the local spectral properties of the signal for the given frame analysis. Mel scale is also less vulnerable to the changes of speaker's vocal cord in course of time. The present study is still ongoing, which may include following further works. HMM may be used to improve the efficiency and precision of the segmentation to deal with crosstalk, laughter and uncharacteristic speech sounds.

Even though much care is taken it is difficult to obtain an efficient speaker recognition system since this task has been challenged by the highly variant input speech signals. The principle source of this variance is the speaker himself. Speech signals in training and testing sessions can be greatly different due to many facts such as people voice change with time, health conditions (e.g. the speaker has a cold), speaking rates, etc. There are also other factors, beyond speaker variability, that present a challenge to speaker recognition technology. Because of all these difficulties this technology is still an active area of research.

REFERENCES

1. Steven W. Smith, Ph.D., ”The Scientist and Engineer's Guide to Digital Signal Processing”, http://www.dspguide.com/whatdsp.htm
2. Thomas Farley and Ken Schmidt , Privateline Telecomunication Expertise, http://www.privateline.com/PCS/history9.htm
3. Processor Comparison: TI C6000 DSP and Motorola G4 PowerPC , Pentek Inc. , www.pentek.com/deliver/TechDoc.cfm/ProcComp.pdf?Filename=ProcComp.pdf
4. TI Training Brochure 2005 (Rev. C) , SPRT294C , “ TI Education Events “ , 2005 , focus.ti.com.cn/cn/lit/an/sprt294c/sprt294c.pdf
5. TMS320C6713DSK, Technical Reference, 2003, Printed in 2003
6. TMS320C6713B Floating-Point Digital Signal Processor (Rev. B), SPRT294B, Revised June 2006 , focus.ti.com/lit/ds/symlink/tms320c6713b.pdf
7. Chptr10
8. DSP Starter Kit (DSK) for the TMS320C6713, Quick Start Installation Guide, 506206-4001B , www.spectrumdigital.com
9. Lawrence Rabiner and Biing-Hwang Juang, “Fundamental of Speech Recognition”, Prentice-Hall, Englewood Cliffs, N.J., 1993.
10. Zhong-Xuan, Yuan & Bo-Ling, Xu & Chong-Zhi, Yu. (1999). “Binary Quantization of Feature Vectors for Robust Text-Independent Speaker Identification” in *IEEE Transactions on Speech and Audio Processing, Vol. 7, No. 1*, January 1999. IEEE, New York, NY, U.S.A.
11. Speech recognition using DSP ,www.vgyan.com/seminar/download/
12. //mfcc processor Jr., J. D., Hansen, J., and Proakis, J. Discrete-Time Processing of Speech Signals, second ed. IEEE Press, New York, 2000
13. FFT Spectrum Analyser applet: guidance notes, http://www.dsptutor.freeuk.com/analyser/guidance.html#leakage .
14. Weisstein, Eric W. "Power Spectrum.", MathWorld—A W olfram Web Resource. http://mathworld.wolfram.com/PowerSpectrum.html

15. Press, William H. [et. al.], "Power Spectrum Estimation Using the FFT", sec. 13.4, Numerical Recipes in C, 2nd ed., Cambridge University Press, 1992.
16. Md. Rashidul Hasan, Mustafa Jamil, Md. Golam Rabbani Md. Saifur Rahman , 3rd International Conference on Electrical & Computer Engineering ICECE 2004, 28-30December2004,Dhaka,Bangladesh www.buet.ac.bd/eee/icece2004/P140.pdf
17. F. Soong, E. Rosenberg, B. Juang, and L. Rabiner, "A Vector Quantization Approach to Speaker Recognition", AT&T Technical Journal, vol. 66, March/April 1987, pp. 14-26