PC BASED REAL-TIME AUDIO SIGNAL PROCESSING

A Thesis

Submitted to the Department of Computer Science and Engineering

of

BRAC University

of

Kamru Faisal Ibne Jalal (ID:05310036)

Md.Shahriar Parvez (ID:02201075)

Md. Mamun vhuiyan (ID:01201073)

In Partial Fulfillment of the

Requirements for the Degree

of

Bachelor of Science in Electronics and Communication Engineering

September 2007

DECLARATION

We hereby declare that this thesis is based on the results found by ourselves. Materials of work found by other researcher are mentioned by reference. This thesis, neither in whole nor in part, has been previously submitted for any degree.

Signature of Supervisor

Signature of Author

ACKNOWLEDGMENTS

This work would not have been possible without the contribution of many people. We would specially like to thank DR. AKM Abdul Malek Azad for boosting this work, and his ideas for the audio processing in Real-Time Linux. We would also like to thank Mr. Rajob for his technical support with available laboratory facilities and Mr. Jiko for assist us in implementation process as well as his advice about RT-Linux operating system. We also like to thank Mr.Suffat Younus for giving us traveling support and his best try to bring the DAQ card from out side of Bangladesh.

ABSTRACT

In this contribution a new concept for the development of real-time audio processing on a general-purpose personal computer based on the Real-Time Linux Operating System (Knoppix3.3) is presented. To overcome important difficulties in Windows Operating System such as robustness and low latency, we applied RT-Linux (hard real-time) operating system as a platform.

This paper describes the Real Time Audio processing framework. First of all we discussed the objectives we pursue with its development and then we have given an overview of the system from both hardware and software point of view. Next we described the implementation details including hardware module, software module and host module (interfacing) and finally the applications.

TABLE OF CONTENTS

Page

LIST OF FIGURES:

LIST OF TABLES:

CHAPTER I

INTRODUCTION

Background and motivation

Music, just a little word but makes a great impression on human mind. It keeps everybody on the run. From the beginning of the human race, men loved music. For this passion for music, we always thought a way to develop music with the help of modern technology. As a student of Electronics and Communication the idea came into our mind that we can process audio signals in real-time which will be great addition in music industry. We choose to process audio in Real-Time to make music composition more ease for the music composers.

To fulfill our target, we have made a research on current audio signal processing tools and we found many obstacles. Most of the audio signal processing software are used for commercial purposes which are very costly and are based on closed-source solution i.e. windows operating system. So, general availability of audio processing tools is far beyond. There is few audio signal processing tools is developed based on open-source solution i.e. Linux. But the signal processing is based on software, which are efficient but costly. Thus, we choose to develop a low cost and efficient audio signal processing tool consists of both hardware and software in hard Real-Time on Linux platform.

Overview

This project is pc based Real-Time audio signal processing which has done in hard real time using Knoppix3.3 as a RT-Linux operating system and consists of both hardware and software. For hardware we have used Data

Acquisition Card (DAQ card- AX5411H) where a microphone is connected to its input line and a speaker is connected to the output line and the DAQ Board is linked to the Host through ISA bus. Via an Application Programmable Interface (API) the host has access to the DAQ Board and various software used to perform the overall process. Special DAQ device driver for real-time operatio is used to make converse between DAQ card and RT-Linux. Because standard Linux has built in device driver but RT-Linux has not. we implemented GUI (Graphical User Interface) consist of some controlling button to control the system, Vxscope shows the output signal and filtration program developed using C Language to filter the audio signal. To being alive all the processes in Real-Time the Thread Programming for real-time operation has been used. Since we work under RT-Linux we have not enough permission to directly access. So we need Thread Program to get access in the RT-Linux.

Objective

In the computer music research field it is not possible to evaluate a given sound synthesis or processing algorithm without listening to its output. If the algorithm has real time control parameters, proper experimentation can only be done with a real time implementation and a human interpreter involved. We have done the audio processing in RT Linux. We can also do it using the Windows Operating System in real time but there are some major problems that we have to face in Windows OS like robustness, latency, and accuracy. Wherein RT-Linux we can get more accuracy and low latency, which is the most important thing in audio processing system. Another opportunity is RT-Linux is a free operating system, which is machine independent. It is thus important to have a good application development framework, so that the loop time between the algorithm idea and its evaluation is reduced as much as possible. Converting a research application prototype into an application which can be used as a musical instrument in other scenarios should not be hard work if both the framework and the developer take enough care of

requirements such as efficiency, latency and robustness. This is what our framework aims to achieve.

Thesis Outline

In this paper we introduced a platform that aims real-time prototyping in the field of digital audio signal processing. We have described how we have successfully used it to implement an interactive sound processor which satisfied live performance requirement as well as preservation for long time through the following chapters accordingly.

In chapter II, the system overview which gives a brief idea of the processing system described with a block diagram.

Chapter III, implementation details consists of three parts operating system and Real-Time Kernel, hardware module and the software module described severally with necessary diagrams.

Chapter IV, described the applications of Real-Time Audio signal processing.

Chapter V, presents the conclusions of this project.

Lastly chapter VI, gives some suggestions for future development.

CHAPTER II

SYSTEM OVERVIEW

The purpose of this project is to digitization of analog audio signal in Real-Time Linux. To accomplished the target several steps to be done described by the following diagram:

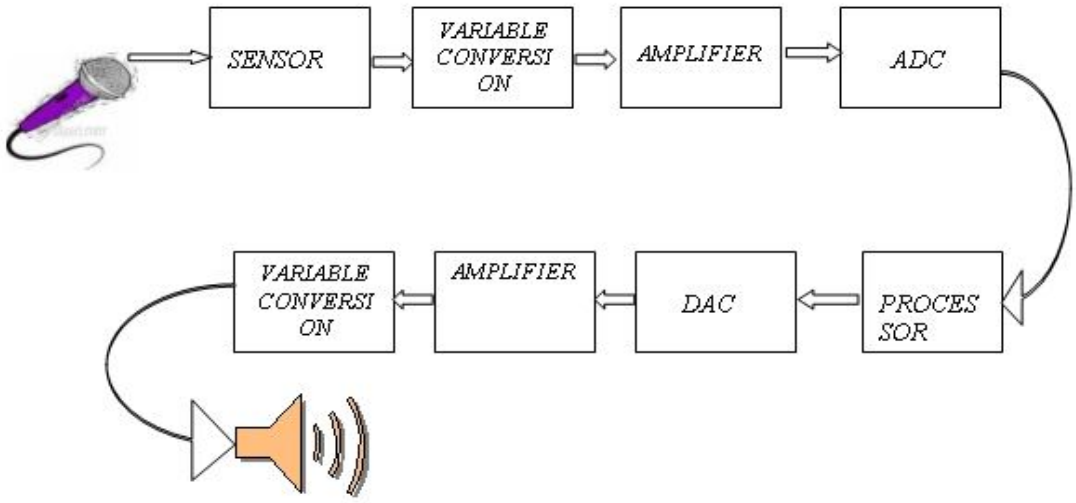

FIGURE 2.1: Block Diagram Of The Sysytem

According to the diagram input (analog audio signal) is recognized by SENSOR. The SENSOR senses the input signal (sound wave) and send it to VARIABLE CONVERSION. Actually the combination of SENSOR and VARIABLE CONVERSION is called TRANSDUCER which included the Microphone itself. The VARIABLE CONVERSION converts the sound wave to voltage form. Then the converted voltage is sent to the AMPLIFIER. The AMPLIFIER fixed the converted voltage level according to ADC requirement. The job of ADC is converted analog voltage level to Digital bit stream and send into PROCESSOR for filteration and preservation. Incase of online process (such as live performance) the PROCESSOR sends the digital data into DAC which converted into analog (voltage) form. Then VARIABLE

CONVERSION converted the voltage in sound form and AMPLIFIER amplifies the sound wave as required and send it to SPEAKER for listening. And all those functions perfom in Hard Real-time Linux where latency is totally undesirable.

CHAPTER III

IMPLEMENTATION DETAILS

Implementation Details describe a data acquisition system (DAQ) which is a combination of computer hardware and software that gathers, stores or processes data in order to control or monitor some sort of physical process in Hard Real-time Linux platform as an operating system. A typical data acquisition system comprises a computer system with DAQ hardware, wherein the DAQ hardware is typically plugged into one of the I/O slots of the computer system. The DAQ hardware is configured and controlled by DAQ software (the device driver for RT-Linux) executing on the computer system. Here we described those part separately as

(a) Operating System and Real-Time Kernel (3.1)

(b) Hardware module (3.2) and

(c) software Module (3.3)

3.1 OPERATING SYSTEM AND REAL-TIME KERNEL

An operating system is the interface between a user's program and the underlying computer hardware. It also manages the execution of user programs such that multiple programs can run simultaneously and access the same hardware. Everyone who uses a computer encounters an operating system, whether it is Windows, Mac, Linux, DOS, or Unix.

This chapter describes the architecture and then summarizes the requirements of a real-time operating system and RT-Linux system structure.

3.1.1 Architecture of an Operating System

An operating system, or more specifically the core or kernel of the operating system, is always resident in memory and provides the interfaces between user programs and the computer hardware. This is shown schematically in Figure 3.1. The name kernel follows from the analogy of a nut, where the kernel is the very heart of the nut and, in the computing domain, the kernel is the very heart of the operating system. Continuing the analogy, protective layers around the kernel that provide user authorization and interaction are called shells.

The physical memory in the computer is partitioned into user space and kernel space, with the kernel space reserved for the kernel code. The kernel of a multi-tasking operating system can manage multiple user programs running simultaneously in user space so that each program thinks it has complete use of all of the hardware resources of the computer and, other than for intentional messages sent between programs, each program thinks that it has it's own memory space and is the only program running.

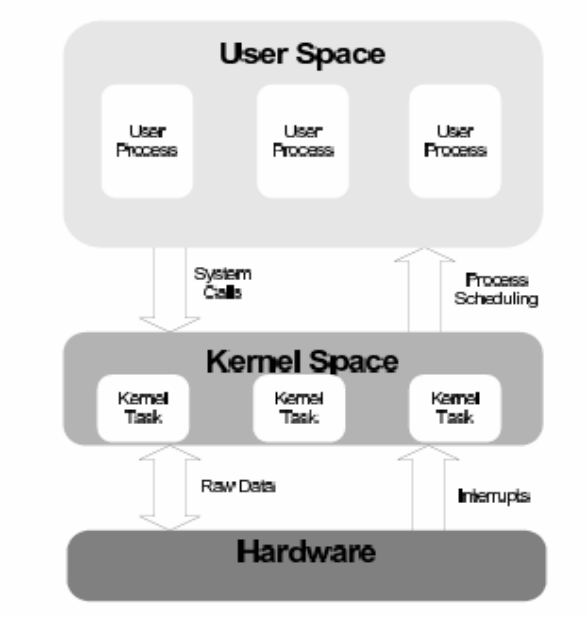

FIGURE 3.1: Operating System Architecture

Communication between user-space programs and the kernel code is achieved through system calls to the kernel code. These system calls typically are to access shared physical resources such as disk drives, serial/parallel ports, network interfaces, keyboards, mice, display screens, and audio and video devices. One unifying aspect of Linux systems is that all the physical resources appear to the user programs as files and are controlled with the same system calls such as open(), close(), read(), write().

All of the input/output activity is controlled by the kernel code so that the user-space programs do not have to be concerned with the details of sharing common physical resources. Device-specific drivers in the kernel manage those details. An operating system is thus tailored to run on specific computer hardware and it isolates user programs from the specifics of the hardware, allowing for portability of user-space application code. Linux is of such an operating system where modules can be loaded and unloaded into the kernel space by user-space commands.

The architecture of an operating system is thus a core or kernel that remains in memory at all times, a set of processes in user-space that support the kernel, plus various modules and utility programs that remain stored on disk until needed. The kernel manages simultaneous execution of multiple user programs and isolates user programs from the details of managing the specific hardware of the computer.

3.1.2 Required Functionality for Real-Time

Real-Time Operating Systems (RTOS) are those able to provide a required level of service in a bounded response time. They can deliver a response in a time less than a designated timing interval. The timing interval may be long in computing terms i.e. orders of seconds, or it may be short i.e. orders of microseconds. For example, a real-time process control system for a chemical or food plant may only sample a sensor and calculate a control command once a second. On the other hand, for smooth response, a stepper

motor must be serviced every few microseconds. A so-called hard real-time system is one that misses no timing deadlines, a soft real-time system can tolerate missing some timing deadlines.

3.1.3 Embedded vs Real-Time Operating Systems

Embedded programs are those that are a fixed and integral part of a device. For example, a hand-held computer, a telephone answering system, and the control computer for the engine of a car all have fixed programs that start up whenever power is turned on. These are called embedded applications. Depending on the required response time, the operating systems for embedded applications may or may not be considered real-time operating systems. Servicing human interactions does not in general require real-time performance, but controlling a machine tool, scientific experiment, or weapon system does.

3.1.4 Measures of Performance for Real-Time Operating Systems

The most vital characteristic of a real-time operating system is how responsive the operating system is in servicing internal and external events. These events include external hardware interrupts, internal software signals, and internal timer interrupts. One measure of responsiveness is latency, the time between the occurrence of an event and the execution of the first instruction in the interrupt code. A second measure is jitter, the variation in the period of nominally constant-period events. To be able to offer low latency and low jitter, the operating system must ensure that any kernel task will be preempted by the real-time task.

3.1.5 POSIX Extensions for Real-Time Applications

The POSIX real-time extensions provide an insight into the additional functionality that is required for a real-time operating system. These real time extensions add message queues (for communication between tasks), shared memory, counting semaphores (needed to synchronize accesses to shared

memory), priority-based execution scheduling, real-time signal extensions, and higher resolution timers. The timers can generate time intervals with at least one microsecond resolution.

3.1.6 RT-Linux System Structure:

The following diagram illustrates the structure of the real-time operating system. The diagram shows that Linux itself is treated just as another task to run, but with lowest priority. Linux in turn controls the running of its non-real-time processes, such as editors, browsers, consoles, viewers, utilities, etc.

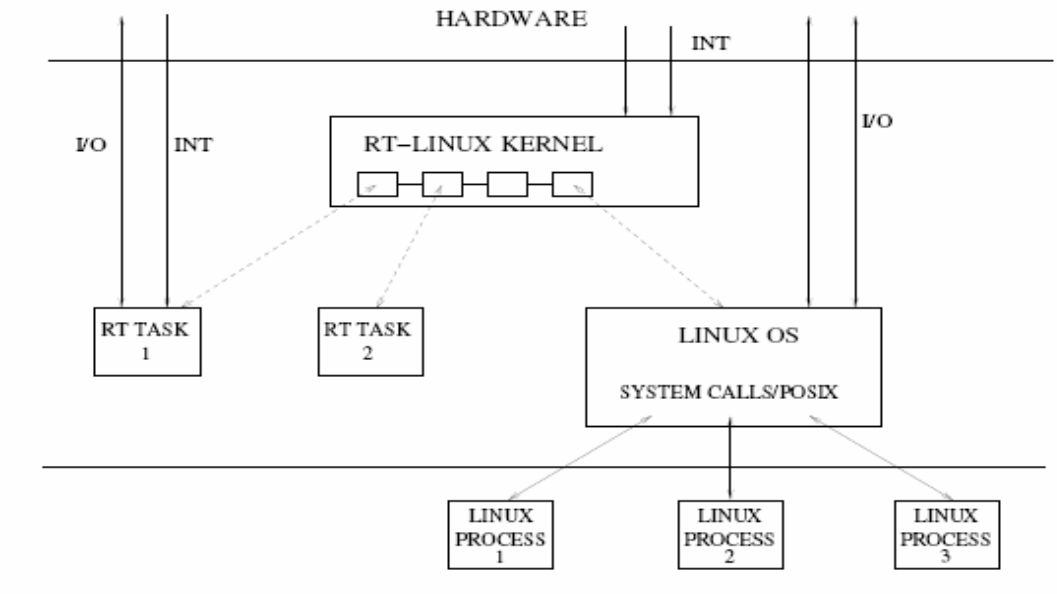

FIGURE 3.2: RT-Linux System Structure

The real-time requirements are met in RTLinux by the real-time kernel capturing all hardware interrupts. One consequence of Linux being relegated to lowest priority status is the possibility of Linux being completely \locked out" of operation. If the real-time processing engages all resources, such as the CPU, devices, and memory, then Linux will not get a chance to run. It will

appear to the user that the system has locked up completely, even though the real-time processing continues to execute.

Real-time tasks run at the kernel privilege level, giving them direct access to the computer resources, such as the CPU, memory, and hardware devices. Running at the kernel privilege level also gives the ability to change task priority, engage inter-process communication (IPC), run user de_ned IPC handlers, and execute user de_ned scheduling algorithms. With privilege however, come responsibility,... care must be taken when constructing real-time programs so that the program does not make undesirable changes to the system that would otherwise not be possible without this privilege. Not only do real-time tasks run at the privilege level of the kernel, but they all exist and are run within the same kernel address space. One consequence of this, aside from the security issue mentioned, is that switching between real-time tasks is made easier and quicker, again reducing latency.

3.2 HARDWARE MODULE

The Hardware Module is based on DAQ Board (Data Acquisition Board) where a microphone is connected to its input line and a speaker is connected to the output line. And the DAQ Board is linked to the Host through ISA bus. Via an Application Programmable Interface (API) the host has access to the DAQ Board. Functionality such as opening a driver and obtaining information on the hardware setup provided.

This chapter described the data acquisition system, summarized the DAQ system structure and the proposed DAQ card.

3.2.1 Data Acquisition System

Data acquisition systems through the DAQ Board are described broadly in the following:

DAQ systems are hybrid electronic devices (analog & digital) with the main role of interfacing the digital signal processing systems to the environment. The key functions of a DAQ system is consists of

i). Signal Conditioning
ii). Analog to Digital conversion (ADC)
iii). Digital to Analog conversion (DAC)
iv). Digital I/O

i) Signal Conditioning:

Non-electrical signals coming from the environment are transformed in electrical signals (current and/or voltage) by transducers. Signal conditioning is further necessary to adapt the output scale range of the transducers to the input signal characteristics of the A/D converters. Programmable Gain Amplifiers (PGA) is usually used to adjust the scale range of the input electrical signal.

ii) Analog to Digital conversion (ADC):

Analog to digital conversion of signals is one of the main goals of a data acquisition system. A/D conversion is the set of operations that establish an exact correspondence between an analog electrical value (current, voltage) and a finite-length binary code. ADC is a three-phase process, all of them being currently performed sequentially by a monolithic device - the analog-to-digital converter.

a) Sampling
b) Quantization
c) Binary coding

iii) Digital to Analog conversion (DAC)

Digital-to-analog conversion is the procedure reciprocal to ADC. With digital-to-analog conversion, each binary code of bits length at the input is related to an electrical value (current or voltage).

Similarly to ADC, the D/A operation requires a well-defined interval to perform the conversion (delay) which raises two issues: (a) what happens to the output electrical signal during a conversion period and (b) how can the resulting signal be shaped as close as possible to the desired contour of a natural, continuous signal.

iv) Digital I/O:

The digital I/O provided by the DAQ systems consists of serial interface and data buffering. The operation can be performed through one of the three independent 8-bidirectional data channels. Data transfer parameters are programmed into the on-board 8255 device for each separate channel.

Digital I/O applications include monitoring and control applications, video testing, chip verification, and pattern recognition. The most common digital I/O interface chip used is the 8255 programmable peripheral Interface (PPI). This PPI has three 8-bit digital ports (A, B, and C). When we configure a port that is part of an 8255 PPI, the 8255 PPI goes through a configuration phase, where all the ports within the same PPI chip get reset to logic low, regardless of the data direction. The data direction on other ports, however, is maintained. Each line in a port on an 8255 PPI has to be configured for the same direction; that is, all the lines in Port A have to be configured for either input or output. Port C on the 8255 can be configured as two 4-bit (nibble) ports, but this functionality is not accessible through the DAQ driver software. The registers on the 8255 must be accessed directly to implement this feature of the 8255 PPI.

3.2.2 General Architecture of Data Acquisition

As presented in the figure above, common data acquisition architectures feature the following components:

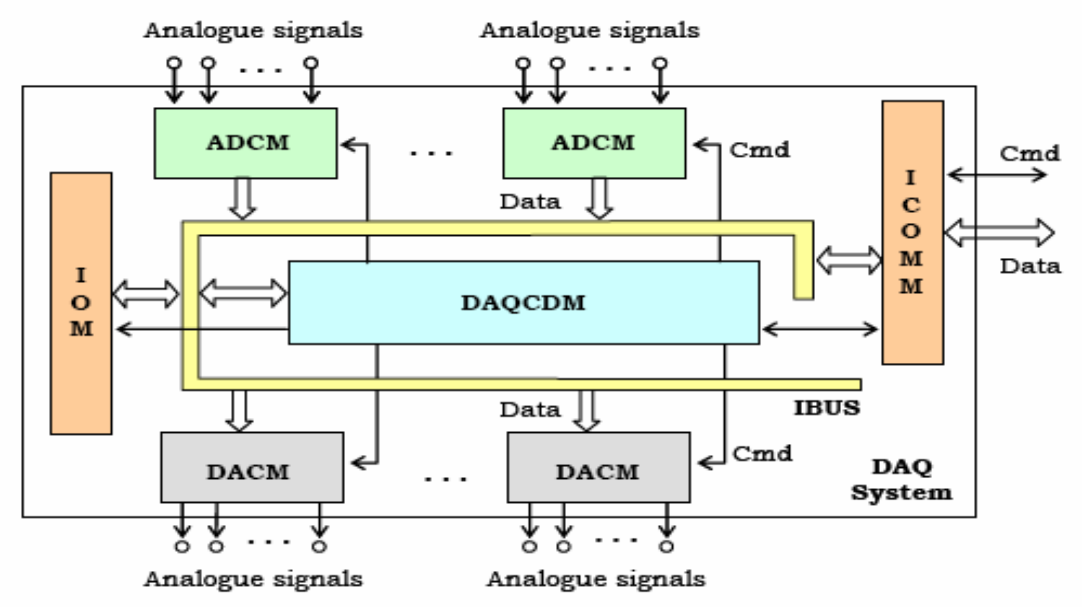

FIGURE 3.3: DAQ Architecture

a) One or more analog-to-digital signal conversion modules (ADCM) one or more digital-to-analog signal conversion modules (DACM) digital I/O module (IOM).

b) Data acquisition command module (DAQCDM)

c) Internal bus for data, address and command lines (IBUS)

d) Communication interface with the controlling system (ICOMM).

The A/D conversion module (ADCM) performs function (ii) and optionally (i) as described above, and it represents the key module of any DAQ system. Its main device is the A/D converter (ADC).

The D/A conversion module (DACM) performs function (iii) described above.The operational control function of the DAQ system is performed by the DAQ command Module (DAQCM).

3.2.3 AX5411H DAQ Board

In this project work we used AX5411H (DAQ Board) is a multifunction analog/digital input/output board. Analog input characteristics of the board are designed to allow user to sample data at high throughput. The combination of hardware auto-scanning multiplexer, a high-speed

sample/hold, and A/D converter allows input sampling speeds up to 60KHz. DMA transfer allows transferring of large amounts of data to memory at high speed. With programmable gains of 1, 2, 4, 8, and 16, and full scale ranges of 5V and 10V, user can define a particular range for each input corresponding to the signal level connected to that channel. Device driver program contained in appendices.

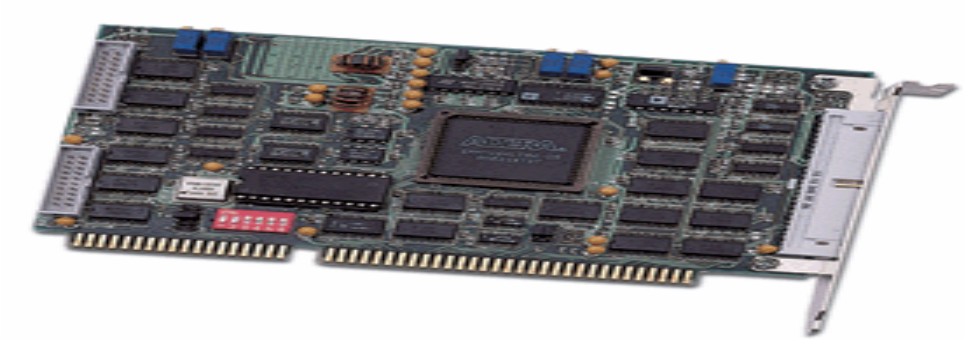

FIGURE 3.4: AX5411H DAQ Board

Table 3.1

The specifications are contained in the following:

Analog Input Subsystem	
Number of Inputs	16S.E.
Resolution	12-bit
Sampling Rate	60KHz max.
A/D Conversion Time	15µs max.
Channel Acquisition Time	5µs max.
System Accuracy	0.03% FSR
Input Range	±10V, ±5V, ±2.5V, ±1.25V, ±0.625V, ±0.3125V, all ranges are software selectable
Output Coding	Offset binary
Maximum Input Voltage without Damage	Power On: ±30V Power Off: ±45V
Input Impedance	Off Channel: 100MΩ, 10pF On Channel: >100MΩ, 50pF
Nonlinearity	±1 LSB
Differential Nonlinearity	±1 LSB
Analog Output Subsystem	
Bias Current	100nA
Number of Channel	2
Output Ranges	0 to 5V, 0 to 10V
Input Data Coding	Straight binary
Current Output, Voltage Range	+5mA max.
Nonlinearity	±1 LSBa
Digital I/O Subsystem	
Digital Input Lines	24
Digital Output lines	24
Improved Noise Margins	Hysteresis VT+ - VT- = 0.4 typ.
Input/Output Level	TTL/DTL compatible
Power Requirements	
+12Voc	40mA typ.
- 12Voc	10mA typ.
+ 5Voc	800mA typ.
Physical/Environmental	
Dimensions	167mm x 99mm
Weight	580g
Connector	One 50-pin mating connector Two 20-pin mating connectors
Relative Humidity	0 to 90%, non-condensing

But, due to time constrained we could not managed the AX5411H card. Thus, to develop our frame work we used comuter parallel port

instead of AX5411H for data communication from the external environment. In computers, ports are used mainly for two reasons: Device control and communication. We programmed PC's Parallel ports for data communication in real-time. Parallel ports are mainly meant for connecting the printer to the PC. But we can program this port for many more applications beyond that. In parallel port, all the 8 bits of a byte will be sent to the port at a time and a indication will be sent in another line. There will be some data lines, some control and some handshaking lines in parallel port.

3.3 SOFTWARE MODULE

Software, consisting of programs enables the computer to perform specific tasks, as opposed to its physical components (hardware) which can only do the tasks they are mechanically designed for. The term includes application software such as word processors which perform productive tasks for users and system software such as operating systems described earlier , which interface with hardware to run the necessary services for user-interfaces and applications.

In this paper we present a new technique that enables the designer in the field of digital audio processing to concentrate on the development of algorithms for processes the data in the hardware independent buffers that the host provided, negotiate sample rate, buffer size and amount of input and output channels in order to create a Real-Time prototype. With this approach a high level programming language (‘C’) has been used also for Real-Time prototype. The new technique furthermore comprises mechanism for a Real-Time messaging mechanism that especially in combination with a Graphical User Interface (GUI), will allow even the non-expert to optimize the parameters of the algorithm by simply clicking buttons.

In this chapter we described each part of the software module accordingly.

3.3.1 DAQ Driver

A device driver is a computer program allowing higher-level computer programs to interact with a computer hardware device.

A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware is connected. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

Linux device drivers are built into the OS kernel, and thus get built for the appropriate bit-width automatically. Provided that sufficient technical information about the hardware is available, the Linux kernel team will write the drivers free of charge. This absolves both hardware vendors and end users from having to worry about drivers. But in this case non-vendors have written numerous device (AX5411H DAQ Board) drivers, mainly for use with Real-Time operation which has been used for this project work.

3.3.2 Filtration

The most common processing approach in the time or space domain is enhancement of the input signal through a method called filtering. Filtering generally consists of some transformation of a number of surrounding samples around the current sample of the input or output signal.

Digital signal processing allows the inexpensive construction of a wide variety of filters. The signal is sampled and an analog to digital converter turns the signal into a stream of numbers. A computer program running on a CPU or a specialized DSP (or less often running on a hardware implementation of the algorithm) calculates an output number stream. This output is converted to a signal by passing it through a digital to analog converter. There are problems with noise introduced by the conversions, but these can be controlled and limited for many useful filters. Due to the

sampling involved, the input signal must be of limited frequency content or aliasing will occur.

The digital filter performs noiseless mathematical operations at each intermediate step in the transform. The primary source of noise in a digital filter is to be found in the initial analog-to-digital conversion (ADC) step, where in addition to any circuit noise introduced, the signal is subject to an unavoidable quantization error which is due to the finite resolution of the digital representation of the signal.

Noted also that frequency components exceeding half the sampling rate of the filter (cf. Nyquist sampling theorem) will be confounded (or aliased) by the filter. Thus an anti-aliasing filter is usually placed ahead of the ADC circuitry to prevent these high-frequency components from aliasing.

To overcome from all those obstacles in digital signal processing and considering the well featured of digital filter, we were supposed to fit filtration into the processing algorithm as a vital part which has developed in a high level language ‘C’. Appendices contains some part of the filtration program.

3.3.3 Graphical User Interface (GUI)

A graphical user interface (GUI) is a type of user interface which allows people to interact with a computer and computer-controlled devices which employ graphical icons, visual indicators or special graphical elements called "widgets", along with text, labels or text navigation to represent the information and actions available to a user. The actions are usually performed through direct manipulation of the graphical elements.

We also tried to make controlling the process flexible to the user using this advantages of GUI program have constructed a visual indicators which has shown below and a part of the GUI program is contained in the Appendices part.

Figure 3.5: Visual Indicators developed in GUI

3.3.4 Vxscope

We also introduced Vxscope to visualize the graphical view of out put signal in various form such as sine wave and square wave developed in 'C' language programming which looks like the following photo.

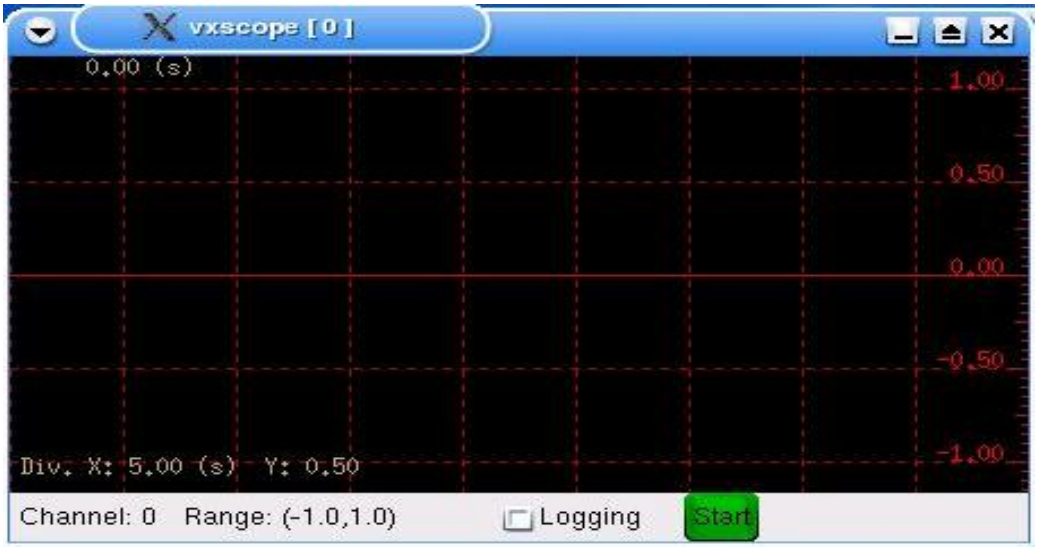

Figure 3.6: Vxscope

Vxscope means versatile XWindows scope. It displays a real-time signal on XWindow using shared memory. It polls the shared memory to get the valu and put it on screen. The value is updated by user process or program.The program is free software which can be redistributed and modified under the team of the GNU (General Public License).

Compile and Executon

Compile the source code and creat a module using the GCC compiler. To simplify things, it is better to creat a Makefile. Then typing 'make' compile the code. Make file can be created by typing in the following line and file named Makefile.

```
all: vxscope


cdsm_nrt.o:  cdsm.c
        gcc -c -O2 -o cdsm_nrt.o cdsm.c
cbuf.o: cbuf.c
        gcc -c -O2 -o cbuf.o cbuf.c
vxscope: cdsm_nrt.o cbuf.o
        gcc -O2 -c -o main.o main.c  `gtk-config --cflags`
        gcc -o vxscope main.o cdsm_nrt.o cbuf.o `gtk-config --libs`
clean:
        rm -f *.o
install:
        make
        cp -f vxscope.7.txt vxscope.7
        gzip vxscope.7
        mv -f vxscope.7.gz /usr/man/man7/
        cp -f vxscope /usr/local/bin/
```

3.3.5 Real-Time Thread Programming

A thread in computer science is short for a *thread of execution*. Threads are a way for a program to fork (or split) itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Threads and processes differ from

one operating system to another, but in general, the way that a thread is created and shares its resources is different from the way a process does.

Multiple threads can be executed in parallel on many computer systems. This *multithreading* generally occurs by time slicing (similar to time-division multiplexing), wherein a single processor switches between different threads, in which case the processing is not literally simultaneous, for the single processor is really doing only one thing at a time. This switching can happen so fast as to give the illusion of simultaneity to an end user. For instance, many PCs today only contain one processor core, but one can run multiple programs at once, such as typing in a document editor while listening to music in an audio playback program; though the user experiences these things as simultaneous, in truth, the processor quickly switches back and forth between these separate processes. On a multiprocessor or multi-core system, now coming into general use, threading can be achieved via multiprocessing, wherein different threads and processes can run literally simultaneously on different processors or cores.

Many modern operating systems directly support both time-sliced and multiprocessor threading with a process scheduler. The operating system kernel allows programmers to manipulate threads via the system call interface.

A real-time application is usually composed of several ``threads'' of execution. Threads are light-weight processes which share a common address space. In RTLinux, all threads share the Linux kernel address space. The advantage of using threads is that switching between threads is quite inexpensive when compared with context switch.

With the grate advantages of Real-Time Thread programming we have done our job in Real-Time successfully. The following example program clarify the real-time thread programming.

Example program:

The best way to understand the working of a thread is to trace a real-time program. For example, the program shown below will execute once every second, and during each iteration it will value from printer port.

The Program code (file - sample.c):

```
#include <linux/errno.h>
#include <time.h>
#include <rtl_sched.h>
#include <rtl_fifo.h>
#include <rtl.h>
#include <pthread.h>
#include <sys/io.h>
#include "mbuff.h"

#define LPT 0x379


#define PERIOD 10000000



void *sample_code(void *arg);

pthread_t sample_thread;

volatile int *y;


/* module initialisation */
int init_module(void)
{
        int module_status=0;

        /* initialise shared memory */
        y = (volatile int*) mbuff_alloc("lab1",1024);
                if (y == NULL) {
                rtl_printf("Shared Memory Creation Failed\n");
                return -1;
        }

        module_status =
pthread_create(&sample_thread,NULL,sample_code,0);
        if (module_status != 0) {
                rtl_printf("Thread initialisation failed: sample status
%d\n",module_status);
                return module_status;
```

```
	}

	return 0;
}


/* module destroy */
void cleanup_module(void)
{
	pthread_delete_np(sample_thread);
	mbuff_free("lab1",(void*)y);
}


/* sampling thread code */
void *sample_code(void *arg)
{
	struct sched_param p;
	hrtime_t now;

	now = gethrtime();

	pthread_setfp_np(pthread_self(),1);
	p.sched_priority = 1;
	pthread_setschedparam(pthread_self(),SCHED_FIFO,&p);
	pthread_make_periodic_np(pthread_self(),now,PERIOD);

	while(1) {




	// take sample from ax5411 card

	y=inb(LPT);


	//print the value that get from printer port

	rtl_printf("The value form Printer port= %d\n",y);
	pthread_wait_np();

	}

	return 0;
}
```

Compiling and Executing:

In order to execute the program, sample.c, (after booting rtlinux, of course) you must do the following:

1. First we have to write “su” to get permission to work under RT-LINUX.
2. Then we write “rtlinux start” in the console to get real time environment.
3. To creat object file we write “gcc –c –o sample.o sample.c ’rtl --config’”
4. After writing previous command the object file(sample.o) if there is no error in the file.
5. Then we should insert the object file in the real time kernel by write “insmod sample.o”
6. To see what happened inside the kernel we have to
7. write “dmesg”.
8. After write the previous command we just see the snap shot in the kernel.
9. To see continuous changing inside the kernel we have to push “Ctrl+Alt+F5”
10. To get back again from Real-time environment to soft Real-time environment we have to push “Ctrl+Alt+F4”
11. To remove object file from kernel we should write “rmmod smple ”

CHAPTER IV

CONCLUSION

In this paper, we introduced a platform that aims at Real-Time prototyping in the field of audio signal processing. The capabilities of general purpose PC, the DAQ card and specially the Real-Time Linux OS can be exploited to avoid time consuming development of new technology. In that cotext we proposed a technique that helps to simplify, most efficient and effective audio signal processing. We completely focused on Real-Time processing algorithm which does not consider any time delay. A powerful messaging mechanism rsupports run time user intraction for parameter optimization and verification, even controlled by a person who has no background in the field of digital signal processing. A very interesting feature of the applied technology is the possibility to use the same software and hardware component for offline processing where data have to preserved for long time and online processing for live performance. we have thus shown that a low cost software and hardware approach for high demand sound processing application is possible.

We have described a first version of development framework for hard real-time audio signal processing and how we have successfully implement an interactive sound processor which satisfied live performance requirements as well as long time preservation with a great accuracy and latency less.

REFERENCES

[1] S.K. Mitra, *Digital Signal Processing: A Computer-Based Approach*, New York, NY: McGraw-Hill, 1998.

[2] A.V. Oppenheim and R.W. Schafer, *Discrete-Time Signal Processing*, Upper Saddle River, NJ: Prentice-Hall, 1999.

[3] J.F. Kaiser, *Nonrecursive Digital Filter Design Using the Io-sinh Window Function,* Proc. 1974 IEEE Int. Symp. Circuit Theory, pp. 20-23, 1974.

[4] S.W.A. Bergen and A. Antoniou, *Design of Nonrecursive Digital Filters Using the Ultraspherical Window Function,* EURASIP Journal on Applied Signal Processing, vol. 2005, no. 12, pp. 1910-1922, 2005.

[5] L. R. Rabiner, J.H. McClellan, and T.W. Parks, FIR Digital Filter Design Techniques Using Weighted Chebyshev Approximation, Proc. IEEE, vol. 63, pp. 595-610, Apr. 1975.

[6] A.G. Deczky, *Synthesis of Recursive Digital Filters Using the Minimum p-Error Criterion*, IEEE Trans. Audio Electroacoust., vol. AU-20, pp. 257-263, Oct. 1972.

[7] Rabiner, Lawrence R., and Gold, Bernard, 1975: *Theory and Application of Digital Signal Processing* (Englewood Cliffs, New Jersey: Prentice-Hall, Inc.) ISBN 0139141014.

[8] A. Antoniou (1993). *Digital Filters: Analysis, Design, and Applications*. McGraw-Hill, New York, NY. ISBN 0070021171.

[9] S.K. Mitra (1998). *Digital Signal Processing: A Computer-Based Approach*. McGraw-Hill, New York, NY. ISBN 0072865466. .

[10] file:///C:/Documents%20and%20Settings/Thesis/Desktop/jn-24.7/THESIS/report/Data%20acquisition%20systems.htm

[11] software. (n.d.). *Dictionary.com Unabridged (v 1.1)*. Retrieved 2007-04-13, from Dictionary.com website: http://dictionary.reference.com/browse/software

[12] file:///C:/Documents%20and%20Settings/Thesis/Desktop/jn-24.7/THESIS/report/Developing%20a%20realtime%20Linux%20data%20acquisition%20application.htm

[13] http://tldp.org/HOWTO/RTLinux-HOWTO-5.html

[14] Motorola, Inc., "MSC8101 Reference Manual: *16-Bit Digital Signal Processor*", MSC8101RM/D, Rev. 1, June 2001.

[15] Texas Instruments, Inc., *"SM320C80 Digital Signal Processor Data Sheet"*, SGUS021, August 1996.

[16] Motorola, Inc., "DSP56000: *24-Bit Digital Signal Processor Family Manual*", DSP56KFAMUM/AD, 1995.

[17] E. C. Ifeachor, B. W. Jervis, "*Digital Signal Processing: A Practical Approach*", Addison-Wesley, 1993.

[18] J. A. Stankovic, "*Real-Time Computing", Invited paper*, BYTE, pp. (155-160), August 1992.

[19] J. A. Stankovic, "*Distributed Real-Time Computing: The Next Generation", Invited keynote paper, Special issue of "Journal of the Society of Instrumentation and Control Engineers of Japan*", Vol. 31, No. 7, pp. (726-736), 1992.

[20] K. Ghosh, B. Mukherjee, K. Schwan, "A Survey of Real-Time Operating Systems", Technical Report GIT-CC-93/18, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, February 1994.

[21] M. V. Micea, "*Signal Acquisition and Conditioning Systems: Laboratory Workshops*", ("Sisteme de achizitie numerica a datelor: Indrumator de laborator"), Comanda 270/2000, Centrul de multiplicare al Universitatii POLITEHNICA Timisoara, 2000.

[22] L. Toma, "*Digital Signal Acquisition and Processing Systems", ("Sisteme de achizitie si prelucrare numerica a semnalelor")*, Editura de Vest, Timisoara, 1996.

[23] Pentek, Inc., "*Digital Signal Processing and Data Acquisition*", Product Catalog, 1996.

[24] National Instruments Corporation, "*DAQ Designer 99 CD-ROM: The Interactive Configuration Advisor for PC-Based Data Acquisition*", 1999.

[25] V. Cretu, M. V. Micea, "*Data Acquisition System - from Design to Real-Life Applications Integration*", Proceedings of the International Conference on Engineering and Modern Electric Systems EMES'99, Computer Section, University of Oradea, May 26-29, 1999, pp. (154-162).

[26] J. G. Proakis, D. G. Manolakis, "*Digital Signal Processing: Principles, Algorithms and Applications*", 3-rd Edition, Prentice-Hall, 1996.

[27] A. V. Oppenheim, R. W. Schafer, "*Digital Signal Processing*", Prentice-Hall, 1996.

[28] A. V. Oppenheim, R. W. Schafer, "*Discrete-Time Signal Processing*", Prentice-Hall, 1989.

[29] P. A. Lynn, W. Fuerst, "*Introductory Digital Signal Processing with Computer Applications",* John Wiley & Sons, 1992.

[30] P. Denbigh, "*System Analysis and Signal Processing: With Emphasis on the Use of MATLAB*", Addison Wesley Longman, 1998.

[31] M. Shnier, "Dictionary *of PC Hardware and Data Communications Terms*", O'Reilly & Associates, Online, 1996. Available: http://www.ora.com/ reference/dictionary.

[32] aRts-project. http://www.arts-projects.org.

[33] Audacity. http://jackit.sourceforge.net.

APPENDICES

A.

Filter.c

```
/*****************************************************************/
/*                                                               */
/* filter.c - ELEC2042                                           */
/*                                                               */
/* The main code for the filter program is contained within      */
/* this file. Functions associated with the setting up and       */
/* running of the GUI (using GTK) are contained within the       */
/* source file gui.c. The source files are compiled and          */
/* linked together using the make utility and the "Makefile"     */
/*****************************************************************/



#include <stdio.h>
#include <unistd.h>
#include <gtk/gtk.h>
#include <gtk/gtkhscale.h>
#include <gtk/gtkvscale.h>
#include <math.h>
#include "mbuff.h"      // for shared memory
#include "filter.h"     // contains digital filter definition and
function declarations
#include "gui.h"        // contains the gui data structure
#include "cdsm.h"       // Common Data Shared Memory - for plotting



/* main program - first thing run */
int main( int argc, char *argv[] )
{

      /* initialisation of gui and */
      if (gui_init() == -1) {
            printf("Initialisation Problem - EXITING \n");
            return -1;
      }
```

```
	/* initialise the digital filter data structure */
	df_init();

	/* initialise shared data (CDSM) for plotting */
	CDSM_init();

	/* set up the gui in 'gui_main' */
	/* GTK will 'sit' within this function until an event occurs
such */
	/* as a mouse click, or a keyboard stroke on the GUI. Another
event */
	/* which will cause GTK to 'break-out' of this function is a
*/
	/* 'timeout' function which runs periodically. A timeout
function */
	/* is set up inside 'gui_main' - called 'filter_run' */
	gui_main();

	/* log data before exiting */
	store_log();

	  /* free up memory allocated to the digital filter structure
'df' */
	/* Will learn more about this in further labs */
	mbuff_free("filter",(void *)df);

	/* clean up shared data structure (CDSM) */
	CDSM_done();

	return 0;
}



/* initialise digital filter */
void df_init(void)
{
	int i;

	  // declare digital filter - allocate space for it
	df = (volatile dig_fil*) mbuff_alloc("filter",sizeof(dig_fil));

	df->n = 4; df->m = 4;	// maximum 4-th order
	for(i=0; i<MAXSIZE; i++) {   // initial signals to zero
		df->y[i] = 0;
		df->u[i] = 0;
	}
	df->time = -1;		// time index to -1 (because time
increments first thing)
	df->timestep = 1;	// time step (discrete time interval)
	df->mode = 0;		// in stop mode
	df->input = 0;		// manual input
	  frequency = 0.5;	  // set to 0.5Hz by default
}



/* Timeout function to run periodically */
/* This is the code that implements the digital filter difference
equation */
int filter_run(void)
```

```
{
      int i;
      float temp=0.0;


      // only update if in start mode
      if (df->mode) {

            //increment time
            df->time += 1;

                // update input (if impulse)
                if (df->input == 1) {
                        if (df->time == 0) df->u[df->time] = 1;
                        else df->u[df->time] = 0;
                }

                // update input (if step)
                if (df->input == 2) df->u[df->time] = 1;

                // update input (if sinusoid)
                if (df->input == 3) df->u[df->time] =
sin(2*3.1416*frequency*df->timestep*df->time);

            // update input (if manual) - why do we do this?
            if (df->input == 0) df->u[df->time] = df->u[df->time-1];

            // numerator
            for (i=0;i<=df->m;i++)
                  if ((df->time-i)>=0) temp += df->b[i]*df->u[df-
>time-i];

            // denominator
            for (i=1;i<=df->n;i++)
                  if ((df->time-i)>=0) temp -= df->a[i]*df->y[df-
>time-i];

            // store in buffer
            df->y[df->time] = temp;

            // print signals out to console
            printf("Time: %d \t Input: %4.3f \t Output: %4.3f \n",df-
>time,df->u[df->time],df->y[df->time]);

      }

      // set value in CDSM structure for plotting - scaled by 1000 as
CDSM stores integers
      CDSM_set(0,(1*temp));
      CDSM_set(1,(1*df->u[df->time]));

      // need to return TRUE otherwise timeout function will cease to
run
      return TRUE;
}



/* callback function for start button */
void start_function( GtkAdjustment *adj, int *arg)
{
      df->mode = 1;  // set to start mode
```

```
}


/* callback function for stop button */
void stop_function( GtkAdjustment *adj, int *arg)
{
      int i;

      /* save data */
      store_log();

      // set to stop mode and reset the signals
      for(i=0; i<MAXSIZE; i++) {
            df->y[i] = 0;
            df->u[i] = 0;
      }
      df->time = -1;  // reset time
      df->mode = 0;   // reset mode to stop
      df->input = 0;  // reset input to manual

        // reset timestep back to 1.0second and change timeout
function period
        df->timestep = 1.0;
      gtk_timeout_remove(flag);
      flag = gtk_timeout_add( (1*df->timestep),
(GtkFunction)filter_run, NULL );

      /* set input back to zero */
      gtk_adjustment_set_value( GTK_ADJUSTMENT(lab1gui.adjust[8]) , 0
);

}



/* callback function for impulse button */
void impulse_function( GtkAdjustment *adj, int *arg)
{
      df->input = 1;
      df->mode = 1;
}



/* callback function for step button */
void step_function( GtkAdjustment *adj, int *arg)
{
      df->input = 2;
      df->mode = 1;
}



/* callback function for sinusoid button */
void sinusoid_function( GtkAdjustment *adj, int *arg)
{
      df->mode = 1;
        df->input = 3;

        // run the sinusoidal signal faster (0.01 sec)
        df->timestep = 0.01;
```

```
      gtk_timeout_remove(flag);
      flag = gtk_timeout_add( (1*df->timestep),
(GtkFunction)filter_run, NULL );

}



/* callback function for changing input field */
void input_change( GtkAdjustment *adj, int *arg)
{
      // change input at t=time to value of widget
      df->input = 0;
      df->u[df->time] = adj->value;
}



/* callback function for changing frequency */
void freq_change( GtkAdjustment *adj, int *arg)
{
        frequency = adj->value;
}



/* callback function for changing 'b' parameter */
void b_parameter_change( GtkAdjustment *adj, int *arg)
{
      // change b coefficient to widget value
      df->b[arg[0]] = 2.0;
}



/* callback function for changing 'a' parameter */
void a_parameter_change( GtkAdjustment *adj, int *arg)
{
      // change a coefficient to widget value
      if (arg[0] != 0) df->a[arg[0]] = 1.0;
}



/* callback function for changing time step */
void time_step_change( GtkAdjustment *adj, int *arg)
{
      // change time step
      df->timestep = adj->value;
      gtk_timeout_remove(flag);
      flag = gtk_timeout_add( (1*df->timestep),
(GtkFunction)filter_run, NULL );
}



/* function to log data to a file `filterdata' */
int store_log(void)
{

      int i;
      FILE *fd_open;
```

```
      float u,y;

      fd_open = fopen("filterdata", "w+");
      if (fd_open == NULL) {
            printf("Error opening file \n");
            return -1;
      }

      for (i=0;i<df->time;i++) {
            u = df->u[i]; y = df->y[i];
            fprintf(fd_open, "%4.3f \t %4.3f \n",u,y);
      }

      fclose(fd_open);

      return 0;
}



/* end filter.c program */
```

Filter.h

```
#include <gtk/gtk.h>
#include <gtk/gtkhscale.h>
#include <gtk/gtkvscale.h>

/* define the maximum size of data arrays */
#define MAXSIZE 2000

/* type definitions of our digital filter */

typedef struct dig_filter
{
      int n,m;        // order of denominator/numerator
      float b[10];    // numerator coefficients
      float a[10];    // denominator coefficients
      int time;       // time index
      float timestep; // time step
      int mode;       // mode - start (1), stop (0)
      int input;      // 0 - manual, 1 - impulse, 2 - step, 3 -
sinusoid;

      float y[MAXSIZE];  // output sequence
      float u[MAXSIZE];  // input sequence

} dig_fil;



/* declare digital filter object - for shared memory */
volatile dig_fil *df;

/* declare flag for timeout function */
```

```
int flag;

/* declare sinusoidal frequency */
float frequency;



/* Function declarations used in filter.c */

/* initialise digital filter */
void df_init(void);

/* Timeout function to run periodically */
/* This is the code that implements the digital filter difference
equation */
int filter_run(void);

/* callback function for start button */
void start_function( GtkAdjustment *adj, int *arg);

/* callback function for stop button */
void stop_function( GtkAdjustment *adj, int *arg);

/* callback function for impulse button */
void impulse_function( GtkAdjustment *adj, int *arg);

/* callback function for step button */
void step_function( GtkAdjustment *adj, int *arg);

/* callback function for sinusoid button */
void sinusoid_function( GtkAdjustment *adj, int *arg);

/* callback function for input change field */
void input_change( GtkAdjustment *adj, int *arg);

/* callback function for changing frequency */
void freq_change( GtkAdjustment *adj, int *arg);

/* callback function for changing 'b' parameter */
void b_parameter_change( GtkAdjustment *adj, int *arg);

/* callback function for changing 'a' parameter */
void a_parameter_change( GtkAdjustment *adj, int *arg);

/* callback function for changing time step */
void time_step_change( GtkAdjustment *adj, int *arg);
```

Makefile

```
all: filter

filter: filter.o gui_f.o cdsm.o
      gcc -o filter filter.o gui_f.o cdsm.o `gtk-config --libs`

filter.o: filter.c
      gcc -c -o filter.o filter.c  `gtk-config --cflags`

gui_f.o: gui_f.c
      gcc -c -o gui_f.o gui_f.c `gtk-config --cflags`

cdsm.o: cdsm.c
```

```
	gcc -c -o cdsm.o cdsm.c

clean:
	rm filter
	rm *.o
```

Cdsm.c

```
File name: cdsm.c

	Data Sharing  interface

*/

#include "cdsm.h"
#ifdef __KERNEL__
#include <mbuff.h>
#include <linux/malloc.h>
#endif

inline void CDSM_init() {

	CDSM_data = (volatile long*)mbuff_alloc(CDSM_DATA_SI,
				CDSM_NUMBER_OF_CHANNEL*sizeof(long));

}

inline void CDSM_done() {
	mbuff_free(CDSM_DATA_SI,(void*)CDSM_data);

}

inline void CDSM_set(int chan, long data)
{
	*(CDSM_data+chan) = data;
}

inline long CDSM_get(int chan)
{
	return *(CDSM_data+chan);
}
```

Cdsm.h

```
CDSM - Common Data Sharing Mechanism

	written by: Linh Vu
	(C) 2002

*/

#ifndef __CDSM_H__
#define __CDSM_H__

// maximum of 16 channel, can change to any number
#define CDSM_NUMBER_OF_CHANNEL 128

// data in string id
#define CDSM_DATA_SI		"CDSM_DATA_SI"
```

```
// mechanism:
// common data sharing mechanism (CDSM)
//
// data will be shared among modules
// by read/write to an array of long type elements (also pointer)


volatile long *CDSM_data;

inline void CDSM_init();
inline void CDSM_done();
inline void CDSM_set(int chan, long data);
inline long CDSM_get(int chan);


#endif
```

B.

AX5411H.c

```
// ax5411.c
//      Implementation of various das 16 functions
//
#include "ax5411.h"


// init()
//      Initialise the AX5411 card
//
void init(void)
{
      // get permission to use I/O device (in non-RT)
      // only compile if used in non-real-time
#ifndef __RTL__
      ioperm(BASE, 16, 1);
#endif __RTL__

      /* reset control and status registers */
      outb(0, CONTROL);
      outb(0, STATUS);
}

// ax5411()
//    This function is used to either read a A/D value (should only
do
//    this when you received an interrupt) or write a D/A value to a
specified
//    channel (There are 2 write and 16 read channels we can use for
the card)
//    The format of the function is as follows:
//      inout: specify whether the operation is a read or a write
//           'a' = write
//           'm' = read
//      channel: specify which channel to read or write
//                 (use channel 0 for both read and write)
//      value: what value to write to the DAS 16 card (only
significant
```

```
//           if you are performing a write operation)
//    The function returns a value which is only signifcant if you
are
//    performing a read operation, or you try to write an invalid
value to
//    to DAS 16 card
//
int ax5411(char inout, int chan, int value)
{
        int ch;
      int ilo, ihi;
      int dataL, dataH;

      // User wants to do a write operation
      //
      if (inout == 'a') {
              // check make sure the value we are writing to the DAS
16
              // card is valid
              //
              if ( (value > 4095) || (value < 0)) { return (value); }

            // Split the value into a lower 4 bits, and higher 8 bits
            dataL = (value << 4) & 0x00F0;
            dataH = (value >> 4) & 0x00FF;

            // write our lower 4 bits to the D/A register
            outb( dataL, (BASE+4+chan) );

            // write our higher 8 bits to the D/A register
            outb( dataH, (BASE+5+chan) );
      }


      // User wants to perform a read operation
      //
      else if ( (inout == 'm') && (value == 0) ) {
              // mask out the higher 4 bits
              ch = chan & 0x000F;

            ch = ch + (ch << 4);

            // Select our channel by writing our value to the MUX
            //  ==> we start and finish on the same channel
            //
            outb( ch, (BASE+2) );

            // clear to A/D register 1st
            outb( 0, BASE);

            // wait until the A/D conversion is complete (shouldn't
            // be necessary
            //
            while (inb(BASE+8) & 0x80) {};

            // read our least significant 4 bits
            ilo = (inb(BASE) >> 4) & 0x000F;

            // read our most significant 8 bits
            ihi = (inb(BASE+1) << 4) & 0x0FF0;

            // combine our results and return the value
```

```
            value = (ihi | ilo);
      }
      return (value);
}
```

AX5411.h

```
// ax5411.h
//      header files and defintions of various functions
//

#include <sys/io.h>

#define   BASE          0x320 /* base address of ax5411 */
#define   STATUS        BASE+8 /* status for ax5411  */
#define   CONTROL       BASE+9 /* control for ax5411 */


////////////////////////////////////////////////////////////////
////////
// Here are the function defined in the file
//


// init(void)
//    function to initialise the AX5411 card
//      call within Linux task - NOT RT-Linux
void init(void);

// ax5411()
//      read or write some values to the das16 card
int ax5411(char inout, int channel, int value);
```

C.

GUI.c

```
#include <stdio.h>
#include <unistd.h>
#include <gtk/gtk.h>
#include <gtk/gtkhscale.h>
#include <gtk/gtkvscale.h>
#include "gui.h"
#include "ax5411.h"


// declare state of motor: 0 - off, 1 - on
int state = 0;

// declare a variable for the motor input
float input = 0;


/* main program - first thing run */
```

```
int main(int argc, char *argv[])
{

      /* initialise GTK */
      gtk_init (&argc, &argv);

      /* initialise ax5411 card */
      init();

      /* gui_main */
      gui_main();

      return 0;
}



/* callback for start/stop */
void start_function( GtkAdjustment *adj, int *arg)
{
      /* turn on */
      state = 1;

      /* send current input to D/A - channel 0 */
      ax5411('a',0,(int)(4095*input));

}

/* callback for start/stop */
void stop_function( GtkAdjustment *adj, int *arg)
{
      /* turn off */
      state = 0;

      /* send a zero to the motor to turn off */
      ax5411('a',0,0);
}


/* callback for changing input */
void input_change( GtkAdjustment *adj, int *arg)
{
      int dtoa;

      /* retrieve value from widget */
      input = adj->value;

      /* scale to 0-4095 */
      dtoa = (int)(input*4095);

      /* if ON then send to D/A - channel 0 */
      if (state) ax5411('a',0,dtoa);

}


/* This callback quits the program */
gint delete_event( GtkWidget *widget, GdkEvent  *event, gpointer
data )
{
    gtk_main_quit ();
    return(FALSE);
```

```
}


/* idle function to run when nothing else is happening */
int idle_run(void)
{
      if (state) printf("Input = %f\n",input);
      return TRUE;
}



/* main gui function - run from main */
void gui_main(void)
{

      /* Create a new window */
      window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
      gtk_widget_set_uposition (window, 0, 0);

      /* Set the window title */
      gtk_window_set_title (GTK_WINDOW (window), "MOTOR GUI");

      /* Set a handler for delete_event that immediately exits GTK.
*/
      gtk_signal_connect(GTK_OBJECT(window),"delete_event",GTK_SIGNAL
_FUNC (delete_event), NULL);

      /* Sets the border width of the window. */
      gtk_container_set_border_width (GTK_CONTAINER (window), 20);

      /* Create a 4x3 table */
      table = gtk_table_new (4, 3, TRUE);
      gtk_table_set_row_spacings (GTK_TABLE (table), 15);
      gtk_table_set_col_spacings (GTK_TABLE (table), 25);

      /* Put the table in the main window */
      gtk_container_add (GTK_CONTAINER (window), table);

      /* Put in label */
      label = gtk_label_new("Motor Input");
      gtk_label_set_justify(GTK_LABEL(label),GTK_JUSTIFY_LEFT);
      gtk_table_attach_defaults (GTK_TABLE(table), label, 0, 2, 0,
1);
      gtk_widget_show (label);

      /* Adjustment for input. The arguments are: */
      /* (start value, minimum, maximum, step increment, page
increment, page size) */
      adjust = gtk_adjustment_new( 0, 0, 1, 0.01, 0.1, 0);

      /* Put in widget for changing input */
      /* Connect it to adjustment */
      spin = gtk_spin_button_new( GTK_ADJUSTMENT(adjust), 0.01, 2);
      gtk_signal_connect(GTK_OBJECT (adjust), "value_changed",
GTK_SIGNAL_FUNC (input_change), NULL);
      gtk_table_attach_defaults (GTK_TABLE(table), spin, 0, 3, 1, 2);
      gtk_widget_show(spin);

      /* Separator */
      separator = gtk_hseparator_new ();
```

```
      gtk_table_attach_defaults (GTK_TABLE(table), separator, 0, 3,
2, 3);
      gtk_widget_show(separator);


      /* Put in buttons - start button */
      button = gtk_button_new_with_label ("Start");
      gtk_signal_connect(GTK_OBJECT (button), "clicked",
GTK_SIGNAL_FUNC (start_function), NULL);
      gtk_table_attach_defaults (GTK_TABLE(table), button, 0, 1, 3,
4);
      gtk_widget_show (button);

      /* Put in buttons - start button */
      button = gtk_button_new_with_label ("Stop");
      gtk_signal_connect(GTK_OBJECT (button), "clicked",
GTK_SIGNAL_FUNC (stop_function), NULL);
      gtk_table_attach_defaults (GTK_TABLE(table), button, 1, 2, 3,
4);
      gtk_widget_show (button);

      /* Put in buttons - Quit button */
      button = gtk_button_new_with_label ("Quit");
      gtk_signal_connect(GTK_OBJECT (button), "clicked",
GTK_SIGNAL_FUNC (delete_event), NULL);
      gtk_table_attach_defaults (GTK_TABLE(table), button, 2, 3, 3,
4);
      gtk_widget_show (button);

      /* add in idle function to run when nothing else is running -
IMPORTANT */
      flag = gtk_idle_add( (GtkFunction)idle_run, NULL );

      gtk_widget_show(table);
      gtk_widget_show(window);


      gtk_main();

}


/* end gui program */
```

GUI.h

```
// type definitions of our gui

typedef struct gtk_gui
{
      GtkWidget *window[3];
      GtkWidget *table[2];
      GtkWidget *label[4];
      GtkWidget *button;
      GtkWidget *radio[4];
      GSList *group[2];
      GtkObject *adjust[20];
      GtkWidget *separator;
      GtkWidget *spin[20];
```

```
      GtkWidget *textbox;
      GtkWidget *hbox;
      GtkWidget *vscrollbar;
} guiobj;

/* This callback quits the program */
gint delete_event( GtkWidget *widget, GdkEvent  *event, gpointer
data );

/* function to initialise gui */
int gui_init(void);

/* main gui function - run from main */
void gui_main(void);


/* declare gui object */
guiobj lab1gui;
```