12-17-2004

Wearable Personal Data Information Capture System

Jiangpeng Shi

University of New Orleans

Follow this and additional works at: https://scholarworks.uno.edu/td

Recommended Citation
https://scholarworks.uno.edu/td/197

This Thesis is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. For more information, please contact scholarworks@uno.edu.
WEARABLE PERSONAL DATA INFORMATION CAPTURE SYSTEM

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Department of Computer Science

by

Jiangpeng Shi

B.S., University of International Business and Economy, 2000
B.S., WuHan Technology Institute of Chemistry Equipment and Machine, 1992

December 2004
Table of Contents

Chapter

1 Introduction.................................................................................................................. 1
   1.1 Wearable Personal Data Information Capture System ......................................... 3
   1.2 The Motivation of the Project............................................................................... 5
   1.3 What we do in this project.................................................................................... 5
   1.4 About Sharp Zaurus SL5600 ................................................................................ 5
2 Audio Capture.............................................................................................................. 8
   2.1 Capture Audio Data .............................................................................................. 8
      2.1.1 About Digit Audio ....................................................................................... 8
      2.1.2 Digit Audio Under Linux............................................................................. 9
   2.2 Pause and Resume Audio Capture Automatically.............................................. 13
   2.3 Save raw data as .wav ......................................................................................... 16
   2.4 Audio Compression ........................................................................................... 19
      2.4.1 Audio Compression for WPDICS............................................................... 19
      2.4.2 MP3 and LAME.......................................................................................... 19
      2.4.3 MP3 Encoder Programming ....................................................................... 21
      2.4.3.1 Initialize the encoder......................................................................................21
      2.4.3.2 Set internal configurations....................................................................... ......22
      2.4.3.3 Encode some data ..........................................................................................23
      2.4.3.4 Flush the buffers and free the internal data structures...................................24
   2.5 Conclusion .......................................................................................................... 26
3 Visual Capture ........................................................................................................... 27
   3.1 Visual Data Capture............................................................................................ 27
      3.1.1 About Visual Data Capture........................................................................ 27
      3.1.2 Image Capture under Linux ....................................................................... 30
   3.2 Image File Format.............................................................................................. 37
      3.2.1 PPM File Format....................................................................................... 38
      3.2.2 JPEG File Format...................................................................................... 40
   3.3 Conclusion ......................................................................................................... 45
4 System Integrity......................................................................................................... 47
   4.1 Cross Compiler ................................................................................................... 47
   4.2 Compile Our Program with Cross Compiler ...................................................... 48
5 Conclusion .................................................................................................................. 51
   5.1 Experience with the Prototype ........................................................................... 51
   5.2 Conclusion and Future Work .............................................................................. 52
Reference .......................................................................................................................... 55
Vita.................................................................................................................................... 56
List of Figures

Figure

1.1 The imagination scheme of a WPDICS ................................................................. 2
1.2 Head-wearing WPDICS ...................................................................................... 2
1.3 Sharp Zaurus SL5600 ....................................................................................... 6
1.4 Terminal of Zaurus SL 5600 ............................................................................. 7
2.1 Open and read sound device ............................................................................. 10
2.2 Audio waveform under 8 bit, 8KH and mono ................................................. 11
2.3 Audio waveform under 32bit, 44KH and stereo ............................................. 11
2.4 Sample code for setting sample ....................................................................... 12
2.5 Binary data of a piece of silent audio .............................................................. 14
2.6 Binary data of a piece of “un-silent” audio ..................................................... 15
2.7 Sample code for implementation of auto-pause ............................................. 16
2.8 Sample code for writing a wave file header .................................................. 18
2.9 Sample of modifying file header ..................................................................... 18
2.10 Audio File Size in Megabytes ...................................................................... 20
2.11 Initialize the encoder ..................................................................................... 22
2.12 Set encoder’s parameters ............................................................................. 22
2.13 Set internal configurations .......................................................................... 23
2.14 Sample code of function lame_encoder() .................................................... 24
2.15 Sample code for encoding ............................................................................ 25
3.1 Scheme of image capture ................................................................................ 28
3.2 Image sensor chip ......................................................................................... 29
3.3 Scheme of CCD with value of each photosite ............................................. 29
3.4 Sample code of fixed device driver ............................................................... 31
3.5 Sample code of opening video device ............................................................ 31
3.6 Getting information from device .................................................................. 32
3.7 160x120 dpi ................................................................................................. 33
3.8 640x480 dpi ................................................................................................. 33
3.9 Sample code of setting image capture window ........................................... 34
3.10 Test the capability of scaling ....................................................................... 34
3.11 Set the image’s properties ........................................................................... 35
3.12 Replacement value of depth and palette ...................................................... 35
3.13 Definition of palette .................................................................................... 36
3.14 Capture image by calling “read” ................................................................. 37
3.15 PPM files in hex .......................................................................................... 38
3.16 Save image as PPM file .............................................................................. 39
3.17 320x240, JPG format .................................................................................. 40
3.18 320x240, PPM format ............................................................................... 41
3.19 Comparisons of PPM and JPG ................................................................. 42
Abstract

This thesis is motivated by development of The Wearable Personal Data Information Capture System (WPDICS), which is used to support the "physical" data stream requirement on a person as they experience daily life, including visual and audio information. To fit the acquirement of “wearable”, the system should be very tiny in size and very low cost.

Our purpose is to conduct a feasibility analysis for WPDICS using commercially available components (Sharp Zaurus SL 5600 Linux PDA) and evaluate the acquisition of "physical" data streams grabbed by this component. In the thesis, we also develop some software test the data acquisition, especially audio and visual data.
Chapter 1

Introduction

1.1 Wearable Personal Data Information Capture System (WPDICS):

WPDICS is a system that can preserve our memories as we experience them and in the way we experience them.

WPDICS is a very low cost and small size data capture system to support the "physical" data stream acquisition on a person as they experience daily life. The physical data stream, at a minimum, includes visual and audio information. It has a built-in tiny digital camera, microphone, user interface, processor, several hundred megabytes of memory, USB FireWire interface, and a replaceable battery. It can be carried easily by the user. Figure 1-1 shows a possible deployment scheme of WPDICS. It consists of a pinhole camera mounted on a shoulder strap and a handheld mouse pad for data entry and labeling.
Figure 1-1: the imagination scheme of a WPDICS

Figure 1-2 shows another possible implementation of WPDICS. The pin-hole camera and microphone can be head-mounted unit, and connected to the power and storage device via a power and data cable. The user can wear the camera and microphone around head, and power and storage box around the waist. [1]
The WPDICS is not a "wearable computer" in the sense of providing computing resources to the user. It is a data acquisition system, which is only used for grabbing “physical” data, including audio and video data, and storing them to memory media.

1.2 The motivation of the project

In our life, a lot of things deserve to remember. For example, we may want to remember entire talk and every audience’s face when we present in an important conference. Or, a person on a wonderful trip doesn’t want to lose any images on his trip. But nobody can hold a camera and recorder all the way. When a man gets 100 years old, he has nothing but his memory to prove to himself that he has a lot of bright spot in his life. A businessman may hope to remember every word from opponent in a fierce negotiation. There are too many things we should remember when we experience our daily life. But there is no way that we can hold a digital camera or recorder with us anytime and anywhere. What we can do now is only remember it by our own brain. [2]

Unfortunately, our memories are not that reliable in our brains. First, when we get older, we have less and less memory capability. Some our memory is inevitably lost. Second, our real life is getting more complex. So it is even harder to remember things clearly. Missing and confusion are quite common. Last, even we can remember something, and we don’t lose this part of memory forever, but we can’t really “see” or “hear” what we remember in our memory. We can only think them in our brain. This is not going to be a clear replay, and not shareable.

Consider if we can record all the scenes which are interesting and valuable, and save them into storage media, but not our brain, then we can almost remember everything in
our life. Now a camera and recorder and do this job. But there is no way that we can carry a camera and recorder everyday. Even we can, we will have tones of video tape and cassettes in our home after couple of years.

This is the motivation of this thesis. All problems can be solved if we have a portable personal data information capture system. The operational concept for WPDICS is as follows. The user puts WPDICS on and starts recording data. At anytime, the user can stop the device and upload the data to a PC and recharge or replace the battery. The device listens for voice, especially the user's voice, and tries to capture complete conversations within privacy considerations. Data can be recorded in different quality rate depending on user's choice. The higher record rate gets the better quality but consumes more storage space. Data can also be recorded in different priority level. When the storage is short, the lower priority level data will be replaced by higher rate data. The data is stored in a very efficient file format which can save storage space. The software uses a record algorithm to pause and resume the recording automatically, which can save power and storage space. The software also has a user interface which should provide inputs to enter personal preference concerning "level of priority", "quality rate", or "reset", etc.

The design of WPDICS is not impossible. First, audio capture and video capture are not new multimedia technology. Second, the price of large capacity storage media is lower and lower: a 120GB hard driver now is 40 dollars and a 1GB compact flash is about 100 dollars. Third, the portable handheld device is very popular now. It is easy to find an available component to be our antitype.
1.3 What we do in this project:

In this project, we conduct a basic feasibility analysis for the WPDICS system using commercially available components. The components can be configured using a modular architecture and not constrained by size, weight, and power limitations. The component we used in this project is Sharp Zaurus SL 5600 PDA. Based on this component, we will focus on the efficient data capture and storage, and application integrity.

Since Sharp Zaurus uses Linux 2.4.18 as its operation system kernel, our first step is to simulate data capture under Linux OS. We choose Redhat 9.0 Linux as our development platform.

First, we conducted research of how to capture audio data. The basic of audio capture is the study of programming /dev/dsp. Under linux, the audio data captured by dsp is ‘raw’ data, which is binary data and the most common audio file is a .wav file, which contains raw audio data. It has good quality but very huge storage size. To save storage space, we should find a more space-save audio file format. So, after we can capture audio data, we also need to convert the data file into a specific storage format. After this part, then we do similar research of how to capture and store visual data.

The second step is an integrated design, where we will try to transplant the application to the PDA. In these step, the program running under PC must be re-compiled using cross-compiler for the PDA, with good compatibility.

1.4 About Sharp Zaurus SL 5600:

Sharp Zaurus SL 5600 is a commercially available component we used to conduct the basic feasibility analysis for the WPDICS. Its underlying Operating System is Linux,
even though at the top level it looks like any PDA OS. Figure 1-3 shows the overview picture of this PDA and Figure 1-4 shows the picture of its terminal.

The reason we choose Sharp Zaurus is because of flexibility and the versatility of its Linux OS. The Zaurus SL-5600 runs Linux embedded OS Qtopia Linux, providing a powerful and open operating environment—allowing many Linux developers to write applications for the SL-5600. It makes programming easier than any other PDA. With a few simple changes, you can have your Zaurus SL-5600 and desktop Linux looking exactly alike. This profits us to transplant application developed under desktop into PDA and makes the application integration easier.

Figure 1-3 Sharp Zaurus SL5600
Figure 1-4: Terminal of Zaurus SL 5600

1.4 outline of this thesis:

This thesis falls into 5 chapters. This is the first chapter and in this chapter we just give a brief introduction of the thesis.

In chapter 2, we implement an audio data capture and MP3 encoding algorithm under Linux.

In chapter 3, we solved the problems of image data capture. We will write program for image capture using webcam, and save the image data with the format which has best trade-off between quality and file size.

Chapter 4 is system integration. In this chapter, we tested audio and image capture program on PDA.

Chapter 5 is the last part and we talked about the test experience on prototype and draw a conclusion. In this chapter we also described the improvement space of this thesis.
2.1 Capture Audio data:

2.1.1 About Digital Audio:

First, let’s see the principle of audio capture. In a general way, to capture a piece of audio data by computer, we just need to convert analog sound signal to digital sound signal. We can call this process as Analog to Digital Conversion, and this is the most commonly used method to represent real voice via computer. DSP (*digital signal processor*), which is a codec device in sound card, is the kernel device used in audio digitalization. In sound card, DSP is a specialized processor chip optimized for digital signal analysis. It can be a dedicated DSP chip, or may implement the functions with a number of discrete devices. It always contains two devices: A/D (Analog to Digital Converter) and D/A (Digital to Analog Converter). Opening for read-only access allows you to use the A/D converter for sound input. Opening for write only will access the D/A converter for sound output. So, we can capture audio data by reading from DSP. [3]

After digitalization, sound is stored as a sequence of samples taken from the audio signal using constant time intervals. A *sample* represents the volume of the signal at the moment when it was measured. In an uncompressed digital audio each sample requires one or more bytes of storage. The number of bytes required depends on
number of channels and sample size. Another important parameter is sample rate, which means the number of samples the DSP can capture in a constant time intervals.

2.1.2 Audio Programming Under Linux

Under Linux, the capture of audio data is enabled by reading /dev/dsp. DSP is the digital sampling and digital recording device, and probably the most important for multimedia applications. It can produce sound by writing to the device accesses the D/A converter; and can also record sound by reading the device activates the A/D converter. Reading from DSP is the technique we use in this project to capture audio data.

After opening a DSP device, reading from the DSP device returns digital sound samples obtained from the A/D converter. Analog data is converted to digital samples by the A/D converter under control of the kernel sound driver and stored in a buffer internal to the kernel. When an application program invokes the read system call, the data is transferred to the calling program's data buffer. It is important to understand that the sampling rate is dependent on the kernel driver, and not the speed at which the application program reads audio data. [4]

The implementation of the open and read operation on DSP device is very simple. Figure 2-1 shows the sample C code of open and read operation of /dev/dsp.
Figure 2-1 Open and read sound device

There are three parameters associated with each sample which can affect the size and quality of audio data: sample size, sample rate, and the number of channels (mono or stereo). These parameters are set to default values each time the device is opened. The default sample size is 8-bit, using one channel, and sample rate is 8 kHz. If you don't like the defaults, you can change them through ioctl calls. These parameters can be modified after opening the sound device before any calls to read or write. This modification affects the size and quality of each sample, and also affects the size and quality of audio files which contain a lot of samples. In fact, the “raw data” audio files, like .wav files, consist of two parts: data part and file header. Data part contains a lot of sample data, which got from reading /dev/dsp. The file header contains some file information and sample information, like sample rate, channel number, sample size. The size of file header is fixed, for example, it is 44 bytes for .wav file. So the size of audio file (so far we only talk about raw data audio file) is decided by each sample and the total number of samples in this file. In each audio file, or just an audio sample, the
size is a very important aspect of it quality: the bigger size, the better quality. *Sample rate* is the number of samples *DSP* can get in a constant time intervals. A higher value of *sample rate* can result better audio quality but a larger file. Figure 2-2 and Figure 2-3 show the waveform model of digital audio data under different sample size, sample rate and channels.

![Figure 2-2: audio waveform under 8 bit, 8KH and mono](image1)

![Figure 2-3: audio waveform under 32bit, 44KH and stereo](image2)

The size of a raw audio file can be calculated by following formula:
Size(bytes) = sample size * sample rate * channels* time period

It is possible for user of WPDICS to change the record rate and get different audio quality by modifying these parameters. The system call “ioctl” for the DSP device is used to change the value of sample size, sample rate and channels:

**SNDCTL_DSP_SETFMT**: Sets the sample size, in bits.

**SNDCTL_DSP_CHANNELS**: Sets the number of channels-- 1 for mono, 2 for stereo.

**SNDCTL_DSP_SPEED**: Sets the sampling rate in samples per second.

Figure 2-4 shows the code for setting sample size, number of channels and sample rate before read from /dev/dsp.

```c

//set sample size
int sampleSize=8;
if(ioctl (id,SNDCTL_DSP_SETFMT,(char *)&sampleSize)==-1)
    /* fatal error */
    perror("SNDCTL_DSP_SETFMT");

//set the number of channels, 1 for mono, 2 for stereo/
int channels=1
if(ioctl (id,SNDCTL_DSP_CHANNELS,(char *)&channels)==-1)
    /* fatal error */
    perror("SNDCTL_DSP_CHANNELS");

//set sample rate/
int sampleRate=8000;
if(ioctl (id,SNDCTL_DSP_SPEED,(char *)&sampleRate)==-1)
    perror("SNDCTL_DSP_SPEED");

Figure 2-4 Sample code for setting samples
2.2 Pause and Resume Audio Capture Automatically

When users use WPDICS to capture audio data, they may face such a situation: there must be long silent periods. It is wasteful to capture the silent part and store them into an audio file because the power and storage capacity is limited. So it is necessary to design an audio algorithm which can stop the audio capture when the silence exceeds a time period and restart when the silence is broken. To solve this problem, let’s take a look how to generate an audio file.

As we talked before, reading from the DSP device returns digital sound samples obtained from the A/D converter, and those samples are written to a buffer. The programmer can decide the buffer’s size, like 1024 or 2048 Bytes. When the buffer is full, the data in this buffer will be written into an audio file, like, a .wav file, then the buffer fills with the next batch of sample data. This process is in a loop so the data in buffer which obtained from A/D are appended to the audio file one after the other, until user stops it. Then we can see that only the process “writing audio file from buffer” decides the size of file; and the DSP and A/D just hear from outside.

If we can stop writing when there is silence, and restart writing when the silence is broken (like putting the transmission in neutral gear), then the useless silence can be squeezed out from an audio data file. To do this, we should look at binary audio data file first.

Figure 2-5 shows the binary code of a piece of audio file recorded while in silence. Figure 2-6 shows the binary code of a piece of audio file which is recorded under normal conversation; To be simple, the sample size of this two audio data is set to 8bit, so as shown in the figure, each hex number represents the value of one sample.
As shown in figure 2-5, we can see that in continuous samples, their value difference is very small. This is because that silence is a kind of continually smooth “voice”. This feature can be used to solve that auto pause problem. We consider each buffer as a unit, and calculate the sum of value difference of all the neighbored samples in this buffer. Then we need a value as the dividing point of silence and un-silence. If the sum is less then this point, (it’s silent outside), we can mark this buffer as a “silent buffer”. When the silence last more than 2 or 3 seconds (decided by programmer and user), which means, in past 2 or 3 seconds, all the data buffers are marked as “silent buffer”, the writing process is paused. In unwritable situation, when the sum of value difference in a buffer is bigger than that dividing point, (the silence is broken), this buffer will be
marked back to “un-silent”, and the status of writing process is set back to “writable” and writing is restarted.

Figure 2-6 binary data of a piece of “un-silent” audio

There is no standard critical point of silence and un-silence in this project. We analyzed a lot of binary data sample, and we found the sum of sample values difference in an un-silent buffer is much larger than that is a silent one. The gap is so huge that we can pick a value in a very easy way. By experience, we choose 400 as the critical point value: if the sum of value difference is less than 400 in a buffer, this buffer is a “silent” buffer; or else it would be an “un-silent” buffer.

By this means, WPDICS is not totally stand-by when it is silent outside, but the silence is not recorded to the storage media. This can save a large mount of power and storage of capacity of WPDICS. Figure 2-7 shows the sample code for function of auto pause and restart.
Figure 2-7 sample code for implementation of auto-pause

2.3 Saving the raw data audio as a .wav

The audio data returned by reading audio device (DSP) is a group of raw data, which is not in any file format. Wave file is the simplest file format to save those data.

The Wave file format is Windows' native file format for storing digital audio data. It has become one of the most widely supported digital audio file formats on the PC due to the popularity of Windows and the file’s simple format.

Wave file consists of 3 parts: RIFF chunk, format chunk and data chunk. In each wave file, from 0 to 11 Byte is RIFF chunk. 0 to 3 bytes are ASCII characters “RIFF”, which indicates the file is a wave file. The next four bytes tell us the length of the entire file

```c
int gap;  // value difference of conjunctive sample /
int gapsum=0;  // sum of gaps in a buffer /
for (k=0; k < 1024 & & k < i; k++) {
    gap = abs((unsigned) testbuf[k+1] - (unsigned) testbuf[k]);
    gapsum = gapsum + gap;
}

if (gapsum >= 400) {
    silence = 0;
    writable = 1;
} else if (gapsum < 400 & & !silence) {
    time(& first);
    silence = 1;
    writable = 1;
} else if (gapsum < 400 & & silence) {
    time(& end);
    chunk = diff time(end, first);
    if (chunk > 3) {
        writable = 0;
    }
}
printf("now is recording......... \n");
```
minus the 8 bytes for the "RIFF" and length (4 bytes). The last 4 bytes in RIFF chunk is ASCII characters “WAVE”. The length of this chunk is 12 Bytes.

The next part is the format chunk, which contains file’s format information. This first 4 bytes in this chunk is ASCII characters “fmt “. In the next 4 bytes we find the value 0x00000010 which is the length of the format chunk: it is always constant at 0x10. The next 2 bytes indicate the file is mono or stereo: 0x0001 is mono and 0x0002 is stereo. Next 2 bytes indicate the number of channels: 0x0001 for 1 channel and 0x0002 for 2 channels. Next 4 bytes indicate the sample rate and next 4 for number of bytes per second. Next 2 bytes indicate BPS (Bytes Per Sample): 1=8 bit Mono, 2=8 bit Stereo or 16 bit Mono, 4=16 bit Stereo. Last 2 bytes in this chunk indicate the number of bits per sample. The total length of this chunk is 24 bytes.

The last part of a wave file is the data chunk, which is typically the largest part. The first 4 bytes of data chunk is ASCII characters “data”. Next 4 bytes is the total length of data followed. All the rest part contains raw audio data. In all wave files, the first 44 bytes contain file information and rest contains raw data. The first 44 bytes is the file header and the rest is the sound data. The following figure shows some sample code of writing information to a wave file header.
Figure 2-8 Sample code for writing a wave file header

This function finished the part of header, because this function is called before writing sound data into file, so, at this time, we still don’t have the exact file length. We can only have the file length after writing data to the file. Next function is used to finish the file length part of header each time after we write to the file. Figure 2-9 shows this function.

```c
void gethead(FILE *fid)
{
    char rif[4]="RIFF";
    char wav[4]="WAVE";
    char format[4]="fmt ";
    char dat[4]="data"
    int firstInt=0;
    int fchunk=16;
    int channels =131073;
    int samplerate=8000;
    int bps=32000;
    int bpsample = 1048577;
    fwrite(rif, 1, 4, fid);
    fwrite(&firstInt, 4, 1, fid);
    fwrite(wav, 1, 4, fid);
    fwrite(format,1, 4, fid);
    fwrite(&fchunk, 4, 1, fid);
    fwrite(&channels, 4, 1, fid);
    fwrite(&samplerate, 4, 1, fid);
    fwrite(&bps, 4, 1, fid);
    fwrite(&bpsample, 4, 1, fid);
    fwrite(&dat, 1, 4, fid);
    fwrite(&firstInt, 4, 1, fid);
}
```

Figure 2-9 Sample of modify file header

```c
void toWave(FILE *fid)
{
    int flength=f.tell(fid);
    int packlength=flength-8;
    int datalength=flength-44;
    fseek(fid, 4, SEEK_SET);
    fwrite(&packlength, 4, 1, fid);
    fseek(fid, 40, SEEK_SET);
    fwrite(&datalength, 4, 1, fid);
    fseek(fid, flength, SEEK_SET);
}
```
After this step, raw data generated from DSP in written into a wave file.

2.4 Audio Compression

2.4.1 Audio Compression for WPDDCS

Normally, audio data captured by A/D and DSP are stored as raw data, and the simplest raw data file format is, as we use in this project, is .wav file. Raw data file is a high quality and large size file format. WPDDICS is not a studio device, so high quality is not very important issue to us. To save more storage space, audio compression is necessary.

Audio compression is a form of data compression designed to reduce the size of audio data files. As with other specific forms of data compression algorithms, there exist two main categories: "lossless" and "lossy" to achieve the compression effect. Lossless audio compression is an audio compression algorithm in which no data is lost. By this algorithm, the audio quality is pretty high but the size is big too. The primary users of lossless compression are audio engineers and those consumers who have high-standard for audio quality. Lossy algorithm suffers audio quality when a file is compressed, but the ratio of compression is very high. This makes lossy-compressed files not welcome for audio engineering applications. But it is a right choice for WPDDICS, as a less than a megabyte can store about a more than minute's worth of audio at very good quality.

2.4.2 MP3 and LAME

MP3 (which stands for MPEG-1/2 Audio Layer 3) is an audio compression algorithm capable of greatly reducing the amount of data required to reproduce audio. [6] This algorithm is devised for removing parts of the data from the raw audio data and this
process is typically based on psycho-acoustic coding theory, which means human listener will not notice the data removal anyway.

In this project, we choose MP3 (actually only the encoding part), which is a popular open standard compression algorithm, as our compression codec. The reason to choose MP3 is that it has a high compression ratio, low quality loss, and it is supported by almost all the audio players. Figure 2-8 shows the comparison of audio file size of .wav file and MP3 file. [5]

<table>
<thead>
<tr>
<th></th>
<th>30 Minute</th>
<th>60 Minute</th>
<th>120 Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wave (11k, 16bit)</td>
<td>38MB</td>
<td>75MB</td>
<td>150MB</td>
</tr>
<tr>
<td>MP3 (16kbps)</td>
<td>4MB</td>
<td>8MB</td>
<td>16MB</td>
</tr>
</tbody>
</table>

Figure 2-10 Audio File Size in Megabytes

There are many different MP3 encoders available now, each producing files of differing quality; so far, the best encoder for us is LAME. The name LAME is a recursive acronym for LAME Aint an MP3 Encoder. There are two reasons why we choose LAME as our audio encoder: first, LAME is an open source MP3 audio compression project and it provides a friendly application programming interface. There are no available MP3 encoders for Zaurus PDA so far, so the only way to get the audio compression part done under WPDICS is: develop our own encoding program for PDA. In this point, an open source project and easy-going API is very important. The second reason is that LAME is not restricted by MP3’s patent. The key algorithms of MP3 are protected by Fraunhofer's patent, and anyone has to pay licensing fees if he wants to create an MP3 encoder. But LAME used patches against the ISO demonstration source, which is distributed separately and available without charge. In
the middle of May 2000, the LAME project reimplemented the last of the ISO source code so compiling the recent versions of LAME no longer requires the ISO source code. In this project, we choose the LAME 3.93 as our reference. We will develop a simple MP3 encoding program which can convert a .wav file into MP3 file. The program should be tested on Linux desktop, and on the Zaurus PDA (we will talk about this part in chapter 4).

2.4.3 MP3 Encoder Programming:

The details of how to convert a .wav file to MP3 file is very complicated and in this project, we will not focus on the theory of MP3 encoding algorithm. Lame 3.93 provides a library, which contains routines of MP3 encoding. Our task is to write a frontend encoder program which can use this library to do compression.

For all MP3 encoding, each of following steps should be done: [7]

1. Initialize the encoder and set default encoder parameters
2. Set internal configuration
3. Encode some data.
4. Flush the buffers and return a final few MP3 frames.
5. Free the internal data structures.

We will introduce the outline of encoder programming along with our program code, following these steps.

2.4.3.1: Initialize the encoder:

This step is done by calling routine init_lame(). In this step, first, we declare a globalflag structure *gf, which points to an encoder module. The function init_lame() opens this encoder module, and sets all the parameters to default. After this, we should
initialize the input file and output file. The following figure is the code of encoder initialization.

```c
/* initialize libmp3lame: open a *gf */
libmp3lame *gf;
libmp3lame_set_global_flags(gf);
if (NULL == (gf = libmp3lame_init()))
    fprintf(stderr, "fatal error during initialization\n");
exit(0);

/* initialize input file. This also sets samplerate and as much 
other data on the input file as available in the headers */

outf = init_files(gf, wav, mp3);
```

Figure 2-11 Initialize the encoder

Although all the parameters are set to default, we still can reset them as necessary. The following is the code how to set encoder’s parameters.

```c
libmp3lame_set_num_channels(gf, 2);
libmp3lame_set_in_samplerate(gf, 44100);
libmp3lame_set_brate(gf, 128);
libmp3lame_set_mode(gf, 1);
libmp3lame_set_quality(gf, 2); /* 2=high 5 = medium 7=low */
```

Figure 2-12 Set encoder’s parameters

### 2.4.3.2: Set internal configurations:

In last step, we set some parameters for encoding, like number of channels and bit rate. But those parameters are only set to the encoder module, which is not in operation yet. To take effect those parameters, we should do this step: set internal configurations. To
do this we only need to call routine `lame_init_params(gf)`. The following is the sample code.

```c
/* analyze the setting options and set internal configuration */
i = lame_init_params(gf);
```

Figure 2-13 Set internal configurations

It is very easy to finish this step just by one function call. But the actual procedure in this routine is very complicated. The following is the jobs done by this function: [7]

1. Set some CPU related flags
2. Check if we are mono->mono, stereo->mono or stereo->stereo
3. Compute bitrate and output samllerate:
4. Set some options which depend on output samplerate
5. Compute the actual compression ratio
6. Set mode based on compression ratio

All these steps are the

**2.4.3.3: Encode some data:**

After parameter and internal configuration setting, now we are ready to encode wav file to MP3 file. This step is done by calling another routine `lame_encoder()`. In this function, the input is wav data, and output is MP3 frames. This routine handles all buffering, resampling and filtering for you. The required MP3buffer_size can be computed from num_samples, samplerate and encoding rate. In this function, the return code is the number of bytes output in MP3buffer. This can be 0. If it is <0, an error occurred. The lame project provides the source code of encoding algorithm. In
this project, we just hack and simplify the source code. The main routine in this function is `lame_encode_buffer_int()`, which is defined in `lame.c`, part of library. The following figure is the code of this routine:

```c
int
lame_encoder(lame_global_flags *gf, FILE *outf, int nogap, char *inPath,
             char *outPath)
{
    unsigned char mp3buffer[LAME_MAXMP3BUFFER];
    int Buffer[2][1152];
    int iread, imp3;
    static const char *mode_names[2][4] = {
        (“stereo”, “j-stereo”， “dual-ch”, “single-ch”),
        (“stereo”, “force-ms”, “dual-ch”, “single-ch”)
    };
    int frames;
    /* encode */
    imp3 = lame_encode_buffer_int(gf, Buffer[0], Buffer[1], iread,
                                  mp3buffer, sizeof(mp3buffer));

    if (fwrite(mp3buffer, 1, imp3, outf) != imp3) {
        printf(stderr, “Error writing mp3 output \n”);
        return 1;
    }
}

while (iread);
/* may return one more mp3 frame */
imp3 = lame_encode_flush(gf, mp3buffer, sizeof(mp3buffer));

fwrite(mp3buffer, 1, imp3, outf);

return 0;
}
```

Figure 2-14 Sample code of function `lame_encoder()`

### 2.4.3.4: Flush the buffers and free the internal data structures

The routine `lame_encode_flush()` will flush internal PCM sample buffers, then MP3 buffers. The calling of this function may return a final few MP3 frames.

The last step is free the internal buffer. This is done by calling routine `lame_close()`.

This function can be referenced from library. This routine frees all malloc’d data in
internal data structure, and then free the internal data structure. The all encoding process end.

Now, we have successfully finished the whole MP3 encoding process. In fact, the actual encoding computation in these steps is very complicated, but it is not necessary in this project to show the details of that. Our goal is simply to implement the MP3 encoder. The following figure shows the sample code of frontend encoder.

```c
void getMP3(char *wav, char *mp3)
{
    int ret;
    int bit;
    lame_global_flag *gf;

    FILE *outf;
    input_format = sf_unknown;

    /* initialize l13mp3lame: open a *gf */
    if (!gf = lame_init())
    {
        printf(stderr, "fatal error during initialization\n");
        exit(0);
    }

    /* initialize input file. This also sets sample rate and as much other data on the input file as available in the headers */
    outf = init_files(gf, wav, mp3);

    /* Now that all the options are set, lame needs to analyze them and set some more internal options and check for problems */
    l = lame_init_params(gf);

    /* encode a single input file */
    ret = lame_encoder(gf, outf, 0, wav, mp3);
    fclose(outf); /* close the output file */
    close_infile(); /* close the input file */
    lame_close(gf);
}
```

Figure 2-15 Sample code for encoding
2.5 Conclusion

Now, we can capture audio data by a microphone, and convert the data into MP3. As a conclusion, we can outline the work in this chapter as following:

1. Capture audio data by reading /dev/dsp
2. Write grabbed audio into a file from internal buffer
3. Add automatically pause and restart recording algorithm
4. Add file header to the file to generate a wave file
5. Convert wave file to MP3 using Lame MP3 algorithm

The program runs well on Linux PC. We also test it on the Sharp Zaurus SL5600 PDA. We will talk about this in chapter 4.
Chapter 3

Visual Capture

3.1 Visual Data Capture

3.1.1 About Visual Data Capture

Visual information is even more important than audio data. More than 70% of the information people get is from their visual system. In our proposal, we try to implement a video capture application under Zaurus, which can simulate WPDICS’s video recording function. But, unfortunately, Sharp Zaurus SL 5600 doesn’t support video recording and playing. So in this project, we will try to develop an application which can grab images by periodically, which can simulate digital video. Further more, there are a number of file formats used to store a set of images, and we will find out the best one for WPDICS.

In this project, we are not going to learn how to digitize a picture in details. We just take an overview of how to grab and store a picture under Linux and implement a simple application of image capture.

Figure 3-1 shows the arrangement which is used to capture and store digital images produced by a digital camera. In our case, we are using a webcam as our capture device, so the image grabbed by it will be transferred directly to computer. Now let take a brief look of how to grab a picture by a digital camera or webcam.
An image sensor, which is a solid-state device within a digital camera, is the key device used to capture an image. A CCD (charge-coupled device), which is used in most digital camera and camcorder, is a popular image sensor. An image sensor is a silicon chip which consists of a two-dimensional grid of light-sensitive cells called photosites. Each of these photosites records the intensity or brightness of the light that falls on it. Each photosite reacts to the light that falls on it by accumulating a charge; the more light, the higher the charge. The level of charge is then read out and converted into a digital value using an AD converter. These values are used to produce a digital image.
Figure 3-2 shows a picture of an image sensor; Figure 3-3 shows the scheme of image sensor after reacted with light fall on it. The values express the intensity of light.

In fact, Image sensors record only the gray scale—a series tones ranging from pure white to pure black. Basically, they only capture a range of brightness. To capture color image, color filters are used on the sensor's photosites to separate out the colors of the light reflected from a scene. This is not very important to our project so we can skip the details.
3.1.2 Image Capture Under Linux

First, we test image capture under Linux desktop.

All the video programming under Linux is based on Video4Linux (V4L), which is the kernel driver for all video devices. Similar to audio devices under Linux, all the video device are in form of device files, which can be read and write directly. The capture of image is by means of reading device /dev/video, in the same way as opening and reading from a normal file. [8]

In this project, the webcam we are using is Intel CS330 USB Webcam. Unlike Windows, Linux doesn’t have built-in USB webcam driver program by itself; and manufacturer of webcam normally doesn’t provide driver for Linux OS either. So, to get thing started, we need to install device driver for webcam. We choose the SPCA50X USB Camera Linux Driver from www.sourceforge.net, which is a GNU/Linux kernel driver for USB cameras based on Sunplus spca50 and some spca5 chipsets. This is an open source package, which supports our Intel CS330 USB webcam. After compile and loading the driver, the USB camera is ready to use. [9]

Note: We found a problem of the driver when we try to compile it: the source code provided from http://sourceforge.net/projects/spca50x/ doesn’t compile under Red Hat 9.0, because in Red Hat 9.0, the API of mmap() is changed. There are 4 argument in mmap() function in RH9, whereas there are 3 in others. The simplest way to fix this problem is add a new argument to the source code of driver: spca50x.c. Figure 3-4 shows the sample code: the argument struct vm_area_struct *vma in the function static int spca50x_mmap(....) is the new argument.
Figure 3-4 sample code of fixed device driver

Now the driver is compiled and loaded, the webcam is ready to use. We will talk about how to capture image under Linux.

1. Open video device:

This step is relatively easy: use the `open` function to return a device descriptor. If the call to `open` was a success, then device descriptor will contain a valid handle to the device. If the call failed, then device descriptor will be set to -1. Device name will generally be `/dev/video`, but this may be different (e.g. `/dev/video0`) depending upon how VFL is configured on your system. In our project, the device name is `/dev/video0`.

Figure 3-5 shows the sample code of open from video device file.

```c
if((camera->dev<0))
{
    camera->dev = open(camera->devname, O_RDWR);
    // printf("Opening: %d\n", camera->dev);
    if (camera->dev < 0) {
        perror("/dev/video");
        printf("check if your webcam is plugged in correctly! \n");
        exit(1);
    }
}
```

Figure 3-5 sample code of opening video device
2. Get information of device:

This step is optional, but it is a good idea to check the capabilities of the device if your program may be running on different machines or with different devices. We can communicate to the VFL device via `ioctl` system calls. This step can provide us the capabilities of the device such things, for example, maximum and minimum capture dimensions, the actual capture size, the color depth, and so on. This information is very important to an image’s definition and storage size, and this will be discussed in a later section. The Figure 3-6 shows the sample code query the device for its capabilities:

```c
ioctl(webcam->dev, VIDIOCGBCAP, &webcam->vid_caps);
ioctl(webcam->dev, VIDIOCGRWIN, &webcam->vid_win);
ioctl(webcam->dev, VIDIOCGRPICT, &webcam->vid_pic);
```

Figure 3-6 Getting information from device

3. Set the width and height of the capture image:

The capture window size of an image is very important conception in our project. The image’s capture window size is related to picture’s definition: larger size, clearer picture. Figure 3-7 and Figure 3-8 shows two pictures, which are captured under different capture window sizes: 160x120 and 640x480. It is easy to see which one is clearer.
On the other hand, the capture window size decides image’s storage size, in some degree. The size of raw data captured by a webcam (with 8bit color depth) can be calculated by the following formulation:

$$\text{Size} = \text{width} \times \text{height} \times 3 \text{/ byte}$$

In our project, because the storage space of WPDICS is limited, so it is necessary to provide a function which can let user modify images’ capture window size, depend on user’s requirement. Similar as querying information from device, we can set device’s
configuration via system call \textit{ioctl} too. The sample code in Figure 3-9 will show how to set the width and height of the capture image.

\begin{verbatim}
......
int width;
int height;
printf("please input value of width! \n");
scanf("%d", &width);
camera->vid_win.width=width;
printf("input the height of image: \n");
scanf("%d", &height);
camera->vid_win.height=height;
if (ioctl (camera->dev, VIDEOSW, &camera->vid_win) == -1) {
    perror ("ioctl (VIDEOSW)"");
    ......
\end{verbatim}

Figure 3-9 Sample code of setting image capture window

Not every device supports image scaling, therefore it is important to test for this capability. So, it would be necessary to test for this capability before we set the scale. The code in Figure 3-10 shows the test for scaling capabilities

\begin{verbatim}
......
if ({camera->vid_caps.type & VID_TYPE_SCLAES) != 0) {
    // supports the ability to scale captured images
}
......
\end{verbatim}

Figure 3-10 test the capability of scaling

4. \textit{Set other properties of image}:

We can also change the images brightness, colour, contrast, color depth and palette.

This section is optional too if we are willing to work with the default value of the device. But it is also necessary if user want to change the property of the image to get
the best capture effect. The sample code in Figure 3-11 will set the image properties, for example, brightness, depth and palette for capture. Because there are many other fields in the *video_picture* structure, we first read the default values into the structure and then set the fields whose values we want to change.

```c
// get default image properties
if (ioctl (webcam->dev, VIDEOSPICT, webcam->vid_pic) != -1)
{
    // successfully retrieved the default image properties
    // set image properties
    webcam->vid_pic.brightness = brightness*256;
    webcam->vid_pic.contrast = contrast*256;
    webcam->vid_pic.whiteness = whiteness*256;
    webcam->vid_pic.depth = 8;
    webcam->vid_pic.palette = VIDEO_PALETTE_GRED;
    if (ioctl (webcam->dev, VIDEOSPICT, &webcam->vid_pic) == -1)
    { // failed to set the image properties
```

Figure 3-11 set the image’s properties

The value of depth should match the value of palette, or else the settings are not going to work. Figure 3-12 shows some common replacement values for depth and palette, and Figure 3-13 shows the definition of palette.

<table>
<thead>
<tr>
<th><em>vid_pic.depth</em></th>
<th><em>vid_pic.palette</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>15bit RGB</td>
<td>VIDEO_PALETTE_RGB555</td>
</tr>
<tr>
<td>16bit RGB</td>
<td>VIDEO_PALETTE_RGB565</td>
</tr>
<tr>
<td>24bit RGB</td>
<td>VIDEO_PALETTE_RGB24</td>
</tr>
<tr>
<td>32bit RGB</td>
<td>VIDEO_PALETTE_RGB32</td>
</tr>
</tbody>
</table>

Figure 3-12 replacement value of depth and palette
5. Capture image frames by using system call “read”:

So far, our capture device has been configured for the type of image it will capture and we have queried this information from the video device. Now we can talk about capturing images from the video device.

The basic principle of capture an image is: Image sensor in webcam produces the digital data of an image, and we can read these data to a buffer from device file /dev/video0. The simplest way to implement it is to perform a system call “read” from the device. This is a blocking call that will return the entire frame. The call will wait for a frame to be received from the capture device, and then the call will copy the frame buffer into a user provided buffer. Note, if we will use this method, it is the responsibility of the caller to allocate enough space for the buffer and to pass the buffer and its length to the read function. After reading, the buffer should be released. The
simplest way to do this is using function `free()`. The code in Figure 3-14 shows how to capture an image via system call “read”.

```c
.
camera->picbuff = NULL;
len = read (camera->dev, camera->picbuff, camera->vid_caps.maxwidth
    * camera->vid_caps.maxheight*3);
if (len <= 0)
    fprintf(stderr, "Error reading image...\n");
else {
    //save or compress image
}
free(camera->picbuff);
```

Figure 3-14 capture image by calling “read”

6. **Close the VFL device:**

At this point, the process of image capture is finished. Actually, the job is still not done here. We have to save the captured buffer into an image file. The format of image file is very important to WPDICS, so we will talk about this in next section.

After capture the image, the last step is close the V4L device. This is very simple:

```
close (camera->dev);
```

### 3.2 Image file format

Now, an image is successfully captured. The next problem is how to save the image data.

There are too many different image file formats in use. Here we only talk about two of them: PPM and JPEG file. The reason of choosing these two types is:

1. PPM file is a very simple image file format, which is easy to process and implement.
2. JPEG is a most popular and effective image compression algorithm.
In this section, we will learn their features and how to implement them.

### 3.2.1. PPM file format:

PPM, which stands for portable pixel map, is a simple color graphics format. It is always used as an intermediate format for storing color bitmap information generated by video device. It uses 24 bit per pixel, 8 for red, 8 for blue, and 8 for green.

A PPM file consists of two parts: file header and raw image data. The header consists of at least three parts: The first part is a magic PPM identifier, which can be "P3" or "P6". The next part consists of the width and height of the image as ASCII numbers. The last part of the header gives the maximum value of the color components for the pixels, this allows the format to describe more than single byte (0..255) colour values.

For example, a header of a PPM file can be:

```
P6 160 120 255
```

Another part of PPM file, is the image data, which generated by image sensor and A/D converter. Figure 3-15 shows a part of PPM file in its Hex. From this figure, we can see the first line is file header, and the rest is raw data.

```
000000Ch: 50 36 0D 0A 31 36 30 20 31 32 33 0D 0A 32 35 35 ; P6..160 120..255
000001Ch: CD 0A FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF Fd ; ..
000002Ch: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ;
000003Ch: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ;
000004Ch: D0 D4 AE CB BE 0B 09 06 04 02 Fd ; 创 恋 师 师
000005Ch: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ;
000006Ch: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ;
000007Ch: A6 24 A0 8A 0B 09 06 04 02 Fd ;
000008Ch: E1 A3 A9 A5 9C AE AD A1 A7 Ff A5 ;
000009Ch: S9 06 A0 8A 0B 09 06 04 02 Fd ;
00000aCh: A8 A3 9E A1 A5 99 9F A0 95 QA 97 93 98 95 96 A2 ; 年 许 晚 梅 樱
```

Figure 3-15 PPM files in hex

The format, like “header+data” structure, is very simple; and unlike most graphics file formats, a PPM file is plain text and can be processed with text processing tools. So to
programmer, it is very easy to save a captured image in PPM format. The code in Figure 3-16 will save image data from a buffer into a PPM file.

```
...........
if (camera->grayscale){
for (i = 0; i < camera->vid_win.width*camera->vid_win.height; i++){
    buff[0] = camera->pic[i];
    buff[1] = camera->pic[i];
    buff[2] = camera->pic[i];
    fwrite(buff, 1, 3, outfile);
}
}
else {
for (i = 0; i < camera->vid_win.width * camera->vid_win.height * 3; i++) {
    fputc(camera->pic[i], outfile);
}
}
...........
```

Figure 3-16 save image as PPM file

From code above, we can see the process of saving raw image data from buffer to a PPM file is pretty easy: just put the binary data from buffer to the file.

Although it is very easy to write and analyze programs to process PPM format, it has a huge shortcoming. It should be noted that this format is very inefficient. The reason is that this format is highly redundant, while containing a lot of information that the human eye can't even discern. This makes the size of PPM file very huge. It is apparent that PPM file format does not fit WPDICS’ limit storage space, even though it is easy to process.

To solve this problem, the best way is image file compression. There are a large number of image compression algorithms in use now. After some research, we choose JPEG as compression algorithm in our project.
3.2.2. JPEG file format:

JPEG is a standard method of lossy image compression based on the DCT transform. The file format which uses this compression algorithm is commonly also called JPEG or .JPG. The name stands for Joint photographic Experts Group. Actually, JPEG itself doesn’t specifies how to compress a graphic image, but how an image is transformed into a stream of bytes. A further standard, created by the Independent JPEG Group, called JFIF (JPEG File Interchange Format) specifies how to produce a file suitable for computer storage and transmission from a JPEG stream. So, in common usage, a "JPEG file" generally means a JFIF file. JFIF also provides program API to implement the JPEG, which is used in this project. This is the right format for those photo images which must be very small files, for example, for WPDICS.

Figure 3-17 320x240, JPG format

JPG compression is a lossy (loss of image quality) compression, which means that the output image is not exactly identical to the input image. But, on typical photographic
images, very good compression levels can be obtained with no visible change, and remarkably high compression levels are possible if the image quality is not very important. It often compressed by 90%, or to only 1/10 of the size of the original data. Figure 3-17 and Figure 3-18 shows two images captured under same window size, but respectively in PPM and JPG format.

![Figure 3-18 320x240 PPM format](image)

From above two pictures, we can find there are no major visible differences between them. But the size of this to picture is a big difference. The table in Figure 3-19 shows the different size of PPM and JPG file, which captured under same capture window size.

<table>
<thead>
<tr>
<th>Size</th>
<th>PPM</th>
<th>JPG</th>
</tr>
</thead>
<tbody>
<tr>
<td>160x120</td>
<td>57K</td>
<td>4K</td>
</tr>
<tr>
<td>240x180</td>
<td>226K</td>
<td>10K</td>
</tr>
<tr>
<td>320x240</td>
<td>226K</td>
<td>11K</td>
</tr>
<tr>
<td>640x480</td>
<td>901K</td>
<td>32K</td>
</tr>
</tbody>
</table>

![Figure 3-19 comparison of PPM and JPG](image)
JPG compression has such a high efficiency because it is intentionally designed to be lossy, designed to give very small files without the requirement for full recoverability. JPG modifies the image pixel data (color values) to be more convenient for its compression method. The redundancy detail which doesn’t effect visual quality and compress well can be ignored, without visible quality change. This allows amazing size reductions, but when we open the file and expand the data to access it again, it is no longer the same data as before. The lost of data breaks the integrity of original file, but, in our project, this is not a big problem.

3. Implementation of JPEG compression:

The detail of JPEG compression algorithm is very complicated. We don’t focus on the theory of this algorithm in this project. What we want to do is how to implement JPEG compression under Linux.

It is not too hard to implement JPEG compression under Linux. Red Hat 9 has a set of library routines for reading and writing JPEG image files and Independent JPEG Group provides a software package contains open source C program to implement JPEG image compression and decompression. The program for JPEG compression in this project is written under reference of those open source program.

The following are the rough outlines of a JPEG compression operation. [10]

1. Specify the destination file (eg, .jpg)
2. Compression object allocation and initialization
3. Set parameters for compression
4. Start to compress: jpeg_start_compress(...);
5. Compressing....

    while (scan lines remain to be written)

        jpeg_write_scanlines(...);

6. Finish compress:

7. Release the JPEG compression object

We will show how to achieve these steps along with our program code. The following is the code and description to implement the jpeg compression.

```c
/*
 * Before we proceed, we should make sure if the input data format is standard.
 * The standard input image format is a rectangular array of pixels, with each pixel having the same number of "component" values. Each pixel row is an array of JSAMPLEs, which typically are unsigned chars. In this project, the input data is raw image data captured by reading video device, which match this requirement perfectly. So we can just pass a pointer to our image buffer where we use to save raw image data.
 */

/*The structre jpeg_compress_struct contains the JPEG compression parameters and pointers to working space (which is allocated as needed by the JPEG library).*/

struct jpeg_compress_struct cinfo;
/* This struct represents a JPEG error handler. */
struct jpeg_error_mgr jpeg;

// pointer to JSAMPLE row(s)
JSAMPTR row_pointer[MAX_ROW];
// physical row width in image buffer
int row_stride;

/* Step 1: open a data destination file. In this project, it is a .jpg file. */

open an output jpeg file
outfile = fopen(filename, "w");
if (outfile == NULL)  
    return;

jpeg_stdio_dest(&cinfo, outfile); /* continued...... */
```

Figure 3-20 Step 1 of Jpeg compression
// Step 2: allocate and initialize JPEG compression object
// We have to set up the error handler first, in case the initialization
// step fails.
*/
cinfo.err = jpeg_std_error(&err);
// Now we can initialize the JPEG compression object. */
jpeg_create_compress(&cinfo);

// Step 3: now we set parameters for compression */

// First we supply a description of the input image.
// The input image size is actual capture window size
*/
cinfo.image_width = webcam->vid_win.width;
cinfo.image_height = webcam->vid_win.height;

// Now set the number of color components and colorspace
// or input image per pixel. In our program, we consider two
// cases: grayscale and RGB
*/
if (camera->gray) {
cinfo.input_components = 1;
cinfo.in_color_space = JCS_GRAYSCALE;
}
else {
cinfo.input_components = 3;
cinfo.in_color_space = JCS_RGB;
}

// Set smoothing factor. This value can be decided by user */
cinfo.smoothing_factor = webcam->saveStruct.smoothness;
if (camera->saveStruct.smoothness) {
cinfo.optimize_coding = TRUE;
}

// Now use the library's routine to set default compression parameters.
// Note: cinfo.in_color_space must be set before calling this, since the
// defaults depend on the source color space.
*/
jpeg_set_defaults(&cinfo);

// Now we can set any non-default parameters you wish to.
// In our project, we just illustrate the use of quality scaling. 
*/
jpeg_set_quality(&cinfo, camera->saveStruct.quality, TRUE);

// Step 4: Start compressor
// TRUE ensures that we will write a complete interchange-JPEG file.
*/
jpeg_start_compress(&cinfo, TRUE);

continued.....

Figure 3-21 step2 – step4 of Jpeg compression
/* Step 5: Compressing....
* Here we use the library's state variable cinfo.next_scanline as the
* loop counter, so that we don't have to keep track ourselves.
* To keep things simple, we pass one scanline per call; you can pass
* more if you wish, though.
*/

row_stride = cinfo.input_components?webcam->vid_win.width;
while (cinfo.next_scanline < cinfo.image_height) {

/* jpeg_write_scanlines expects an array of pointers to scanlines.
* To make code simple, here the array is only one element long, but
* user could pass more than one scanline at a time
*/
row_pointer[0] = &webcam->pic[cinfo.next_scanline * row_stride];
jpeg_write_scanlines(&cinfo, row_pointer, 1);
}

/* Step 6: Finish compression
* After finish_compress, we can close the output file.
*/
jpeg_finish_compress(&cinfo);
fclose(outfile);

/* Step7: release JPEG compression object
* This is an important step since it will release good deal of memory.
*/
jpeg_destroy_compress(&cinfo);

return;
/* Here we done! */

Figure 3-22 Step5 to step7 of Jpeg compression

3.3 Conclusion:

Now we can capture images under Linux with a webcam. As a conclusion, we can
roughly outline the image capture operation under Linux into three steps:

1. Select appropriate video device, and device driver for Linux.

2. Implement image capture operation under Linux

3. Choose appropriate image file format, and implement image compression.

Actually the program about image capture we talk about so far is a prototype, and its
improvement space is huge. For example, we can set a timer in this program, so the
webcam can capture picture in a specified interval, which can make WPDICS like a
real live “image recorder”. We also can add a time stamp to the file name of each captured image, which can record the “environment” more clearly.
Although now we can implement an application which record audio data and capture image, and store them with minimum storage space. But this is not the end. The main target of this project is to integrate this application into a commercially available portable component, which can simulate a Wearable Personal Data Information Capture System. We will talk about this in next section.
Now we can record audio data and compress them into MP3 files; and we also can capture image and save them as .jpeg file. To simulate WPDICS performance, we need to integrity those applications to our test component: Zaurus SL5600 Linux PDA.

4.1 Cross Compiler:

Normally, programs are developed and compiled on one computer, then distributed to other computers to be used. When the host system (the one the compiler is running on, a IBM laptop in our project) and the target system (the one the resulting programs will run on, Zaurus SL5600 PDA) are not compatible systems, the process called cross compilation is necessary.

In our project, the target system Zaurus SL5600 uses Intel Xscale 400MHz as its processor, which is a low power-consumption ARM embedded processor. This processor is not compatible with our host system, which using Intel Pentium processor. The executables compiled and produced on host system can’t run directly on the Zaurus. And Zaurus SL5600 doesn’t provide a native set of compilation tools. So, to integrate our application to Zaurus, we have to do cross compilation.

First, we need to choose and download a proper cross compiler. As we know, the processor of Zaurus is an ARM processor, so the cross compiler we choose is ARM-
Linux cross compiler. To set up ARM-Linux cross compiler in host system, we need followed resource. These resources are available in Zaurus’s developer site.

1. gcc-cross-sa1100-2.95.2-0.i386.rpm (gcc compiler for ARM architecture)
2. binutils-cross-arm-2.11.2-0.i386.rpm (binary utilities for ARM architecture)
3. glibc-arm-2.2.2-0.i386.rpm (GNU C libraries for ARM architecture)
4. linux-headers-arm-sa1100-2.4.6-3.i386.rpm (Linux header files for ARM architecture)

Next, we'll install the RPMs. To do this, we only need to use installation method:

```
rpm -Uvh filename.rpm
```

After the installation, we can use arm-linux-gcc compile and link our program instead of gcc. [11]

**4.2 Compile Our Program With Cross Compiler**

The programs in this project can be categorized into 4 parts: audio capture, MP3 encoding, and image capture, jpeg compression. All of them have to be compiled by cross compiler. The situation of their compilation are little different.

Compilation of audio capture and image capture parts are simple. The program of audio capture and image capture are written by very basic C language. To compile them, just use arm-linux-gcc instead of gcc. The program can be compiled and linked well by the cross compiler.

When we compile MP3 encoder program, there are something we should notice. In MP3 program, we need to call a library, libMP3lame. This library is provided by lame project, which is MP3 compress algorithm program package. This library is not a built-in library of Linux, so no matter which compiler we are going to use, we have to compile this library from its source code before we compile MP3 encoder program. Lame project provides a perfect Makefile to compile the library using gcc. Before we
compile our encoder by cross compiler, we should compile this library using cross
compiler. To compile library, we only need substitute gcc with arm-linux-gcc. We can
do this by following command:

```
export CC="/opt/Embedix/tools/bin/arm-linux-gcc"
```

After this, we can compiler the program by “make” command. Then we can compile
the library by provided Makefile. After library is compiled, we can simply compile and
link our encoder by using arm cross compiler directly.

Another part is jpeg compression program. The situation here is a little different to
compilation of MP3 compiler. In Linux, there is a built-in jpeg library. So, when we
use gcc to compile this program, we can call library directly in compile command and
don’t need to compile library first:

```
Gcc –ljpec –o cam cam.c
```

But this built-in library is not compiled by cross compiler and ARM cross compiler can
not identify it. We can’t call this library correctly when we using cross compiler. So to
compile jpeg compression program, we need to re-compile the library by cross
compiler first.

In this project, at first we find the source code for jpeg library, and compile them with
cross compiler. Independent JPEG Group provides a software package which contains
source code for the library. Then we just compile the library. This step is similar to
what we did with MP3 encoder. After library is compiled, we can compile and link our
program by using following command:

```
arm-linux-gcc –o cam3 cam3.c libjpeg.a
```
All the program in this project can be compiled and linked by using ARM cross compiler. After we have compiled the program, check the type of the output file with the file command. The followed two pictures show the result of checking. Figure 4-1 shows the result of executable produced by gcc and Figure 4-2 shows executable produced by ARM cross compiler.

```bash
[root@localhost audio]# file mp3encoder
mp3encoder: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linked (uses shared libs), not stripped
```

Figure 4-1 type of output produced by gcc

```bash
[root@localhost lame-3.92]# file mp3encoder
mp3encoder: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.0.0, dynamically linked (uses shared libs), not stripped
```

Figure 4-2 type of output produced by arm-linux-gcc

The executables produced by cross compiler are test on Zaurus SL5600.
Chapter 5

Conclusion

5.1 Experience with the prototype

All the data capture programs are tested under PC with Linux OS and Sharp Zaurus SL5600.

In audio part, the data capture program and MP3 encoding program successfully run under Linux PC. The audio capture application can record voice and generate a wave file for every 15 second. The wave files use actual time when the file is produced as it file name. After wave files are generated, they will be encoded into same name MP3 files immediately. The time used for MP3 encoding is very short, which definitely doesn’t affect next round recording. So, the results of running are a sequence of MP3 files with time stamp as their file name.

The results about MP3 file size are quite positive. We recorded a piece of voice under 8bit sample size, mono mode and 8K sample rate, and encoded it into MP3 file. The size of wave file is about 450K, and 40K in MP3. So, in this situation, we can have MP3 files with size less than 11MB for 1 hour’s recording by using MP3 encoding.

We also test image capture under Linux PC. The program runs very fast when the capture window size is not too big. It will take less than 0.5 second to take a picture and encode it into jpeg file, when the capture window size is set to 320x240; and it will take about 1.5 seconds when the capture window size is set to 640x480. In fact, 1.5
seconds is a little long for the WPDICS, but, the picture quality is good enough when the capture window size is set to 320x240. So we don’t have to set the capture size as 640x480.

In 320x240, each jpeg file size is only 11KB. So, if we take 1 picture per second, the total size of pictures for 1 minute is only 660K.

We also test the program in Sharp Zaurus SL 5600 PDA. The audio capture program runs very well in the PDA: it can record voice and produce wave audio file. And the automatically pause and restart part is successful too.

The problem is about MP3 encoding. It takes too much long to finish the MP3 encoding. We tried to encode an 850KB wave file. The encoding process finished successfully, and encoded MP3 file is 84K. But the process took about 6 hours. The reason is the big difference of computing capability between PDA’s processor and PC’s CPU. Sharp Zaurus SL 5600 uses Xscale embedded processor, which is designed for low power consumption. It’s computing capability is similar to Pentium I. This is not fast enough to do a complex computing, like MP3 encoding.

We can’t test image capture application in PDA, because we don’t have camera for this PDA.

5.2 conclusion and future work

As a conclusion, the design of WPDICS is not impossible, even though it still need some work.

First, the audio data capture is quite successful, and we assure the image capture part is successful too even though we can’t test it because the lack of device.
Second, the results about data file size are also very good. Based on test, we only need less than 11MB for 1 hour continuous audio recording, and 40MB for 1 hour snapshot, with speed 1 picture per second. So, if we run audio and image recording together, for 24 hours continuous recording, the total storage requirement is only 1.2GB. As we know, it is so easy to get a memory media like this for any portable handheld. And to hard driver for PC, this number is really low. Maybe we can record several years’ data and save it in a 100GB hard disk.

The problem we need to solve is how to low the time consumption of MP3 encoding. We pretty there is no way to finish this step only by computing using PDA’s processor. The best way to solve this problem is to use hardware to finish the MP3 encoding process. This is not impossible, because there are already several MP3 recorder pens available in the market, so the hardware technology of MP3 encoder is not a big problem.

Actually, besides the MP3 encoder hardware, we still have some work space on this project. First, we can implement a video capture instead of image capture. To do this, we need to choose a handheld platform which supports video play. Based on this, we can find an appropriate video file format for WPDICS. This is also possible: we test some video clips, and we can get about 30 minutes video clip with size less than 10M using .RM format.

Second, we can also improvement our MP3 encoding algorithm. In our project, after a piece of audio is captured, the data are written from a buffer into a wave file; and in MP3 encoding process, MP3 encoder reads data from wave file into a MP3 buffer first, then encodes it. In future, we can improve this step so that MP3 encoder can read audio
data directly from buffer where DSP store samples. This can decrease the time consumption of MP3 encoding.
Reference

[1] A. Helal, S. E. Moore, and B. Ramachandran,
"An Integrated Navigation System for Visually Impaired and Disabled",


[3] Jeff Tranter,


[10] The Independent JPEG Group's JPEG Open Source:
ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz

Vita

Jiangpeng Shi was born in HuBei, P. R. China and received his first B.S. degree in Chemistry Machine and Equipment from WuHan Chemistry Technology Institute, and second B.S. degree in International Financial and Trade from University of International Business and Economy. He was admitted to the graduate school of University of New Orleans in 2000 and worked under the guidance of Professor Golden G. Richard III. Jiangpeng Shi’s research interests include computer security, embedded system, and Linux operational system. His hand-on experiences include computer security and forensics, distributed multithreaded client and server programming, embedded system programming, Linux based handheld programming, and webpage designing using CGI, SQL and Javascript.