TECHNICAL UNIVERSITY BRNO

Faculty of Information Technology

Advanced sound card driver for Compaq iPAQ

Tomáš Kašpárek

Brno 2001–2002

I declare, that I worked on this project independently without the assistance of third party. My project leader was Petr Lampa and main consultant for technical issues was Tomas Heran. Publicly available information resources used in my project are listed in the bibliography.

Brno November 9, 2002 ....

I would like to thank Petr Lampa (FIT VUT Brno), Tomáš Heran and Bořivoj Tydlitát (Voice Technologies and Systems IBM), Jaroslav Kysela (ALSA Project), Russel M. King (ARM Linux port) and many others for their technical (and other) support and taken pains while helping me to solve problems related to this project.

Abstract

Goal of this diploma project is the improvement of quality of a specific speech recognizer on a specific hardware platform. Selected hardware platform is Compaq iPAQ (model H3650). It was selected because of its wide spread between users of handheld devices and its usable performance for speech recognition. Software selection is based on the recognizer which is special speech recognizer for embedded devices - Embeded ViaVoice (EVV) from IBM. Because of it's increasing role in the world of PDAs, the Linux operating system was selected. In this environment the Qt GUI system is used for real applications and there is a related diploma project for using this (improved) embedded speech recognizer in real applications on pocket computers. The goals of this project are as follows. At first place it is new driver for the iPAQ sound chip, that will facilitate full utilization of the abilities of that chip. Secondary goal is a speech recognition library to offer these services for programing language level access. As a side goal, it will be needed to solve problems with porting some software for iPAQ (ARM architecture) such as minimizing the size of the Linux kernel and modifying it for these special purposes. Some modifications of the Linux distribution used (Familiar) will be necessary too.

Keywords

Advanced Linux Sound Architecture (ALSA), ARM, Automatic Gain Control (AGC), boot loader, Compaq iPAQ H3650, cross-compiling, digital audio, digital mixer, embedded devices, Embeded ViaVoice (EVV), Familiar distribution, GCC 3.0.x, GNU/Linux, kernel branch 2.4.x and 2.5.x, large samples (word length of 18 or 20 bits), Linux kernel, modularity, Memory Technology Devices (MTD), open source, Open Sound System (OSS), personal digital assistant (PDA), Philips UDA1341TS, porting, Programmable Gain Amplifier (PGA), serial-line connection, Software Gain Control, sound chip driver, speech boundary detection, speech recognition, speech signal processing

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Spontaneous continuous communication of people to machines (and the same in reverse) is just a dream these days. This diploma project deals with a sound driver specialized for speech recognition. So why is speech recognition so difficult?

The main problem is, that human speech recognition is based on the data flow of the speech signal as well as on the context in which the communication occurs and on the context of the communication itself. This is the reason, why people can understand spoken language even if there is high level of noise. Understanding the context of something is then solved using broad knowledge of the surrounding world. And this is the source of difficulty and a topic for many years of research.

Second thing, that is needed to be comprehend is the variety of science disciplines, that anyone needs to know even for basic understanding of problems of speech recognition and spoken language communication in general. To name the basic disciplines: signal processing for such things like Fourier Transforms, acoustics for physics of sound and speech, pattern recognition for clustering and matching algorithms, artificial intelligence for knowledge representation, computer science for implementation and optimization or linguistics for lexical representation, syntax and semantics.

Fortunately, this project participates in just a small set of these problems using accomplished recognizer with the goal of improving it's performance and mainly ability to successfully recognize the speech. Another simplification is the orientation to specific hardware and software configuration. How to gain this goal, should be divided into three steps. First we should use a better sound subsystem, that extends the possibilities for developers of the recognizer, second we should make capital out of the abilities of the sound chip in the hardware we use and third we should implement some general algorithms into a special speech recognition support library.

Chapter 2 Here we explained the basis of the theory for speech recognition. It is included to summarize basic background knowledge for speech recognition. We use [2] and [3] as reference for this chapter. First we explain what is the motivation for teaching machines to communicate by spoken language. We will look on basic description of human organs for speaking, which should then be used in simulation of this process by machines. Next a minimum of the background theory is mentioned for the impatient and for those, who have not filled their desks with columns of specialized books on mathematics and signal processing. A more detailed section about the speech signal representation follows. The next important part of this chapter is dedicated to speech recognition. The environment interaction, such as noise reduction is mostly discussed here.

Chapter 3 This chapter is dedicated to short introduction to Embeded ViaVoice. All information is based on EVV manual from IBM. EVV is product of IBM, so you can read about some basic principles. Another in-depth information could be found in first chapter, where the general background technologies are introduced. We focus on basic EVV structure with respect to C language API. The API in available version includes four parts: EVV control, Vocabulary manager, Audio control and Services.

Chapter 4 Algorithm for improving speech recognizing devised for this project are discussed here. Automatic Gain Control (AGC) usage and purpose is showed first. Its positives and negatives for speech recognition are presented and requirements for our own solution are created. We use the good idea from the AGC and introduce a speech boundary detection algorithm with respect to real speech signal (not just silence/no silence). Result from these analyses is software gain control, but we notice, that it brings certain inefficiency because some processing is done twice. Without access to the information from speech recognizer, we must do our best to identify the real speech signal. Finally the possibility of using large samples (with word length of 18 or 20 bits) is discussed theoretically, because of lack of time for its implementation.

Chapter 5 Sound subsystems on Unix-like operating systems are based on the principles of special files. Special files make the interface between kernel mode sound card drivers and higher layers. Higher layer should be either end-user application or some additional abstract layer, which simplify the task of programing such sound subsystem. First we will introduce parts of a classic sound system and their function in speech recognition or in speech language systems in general. Then we focus on two most used sound subsystems on Linux. The first of them is Open Sound System (OSS), which is older and is based on direct access to special files. Its main goal is (relative) simplicity and maximal portability. It's widely used on both Unix-like and non-Unix systems, but in Linux it will be replaced by the other sound subsystem named ALSA. Advanced Linux Sound Architecture (ALSA) was created by Czech programmer and it's believed it is better then OSS. ALSA introduces another model, where applications use an user land library. This library then interacts with special files, but it also includes some extended functionality.

Chapter 6 The initial status of the hardware and software selected occupy the first part of this chapter. Problems with the older Familiar distribution and older boot loader are described and follow the process of upgrading to newer Familiar 0.5.1. Differences are discussed and the unsolved problems mentioned. Second step was the creation of development environment consisting of cross-compiler, cross-linker and other binary utilities and a basic set of development libraries and support files. Description of common steps in the process of developing the kernel takes place here. Briefly description of changes made and patches created takes place here too. Some of the patches are still pending, but most of them were applied into official Linux kernel.

Chapter 7 ALSA is available in several different fashions todays. For most users, prepared tarballs of separate ALSA packages are the best choice. These could be downloaded directly from ALSA homepage. Users need to configure them properly and build binaries themselves. These packages are well prepared for native configuration and build. Second possibility, how you could get ALSA working, is usage of development branch of Linux kernel. In version 2.5.5-pre1, ALSA was integrated into official kernel branch. In current kernel versions, it is (same as rest of the system) almost usable for most people. Relevant ALSA package (alsa-kernel) is regularly merged into Linux kernel. The last chance is use of CVS directly. None of necessary packages is prepared for cross-compilation, so we will need to do some modifications (mostly in configuration files).

Chapter 8 One of main parts of this project is creating kernel-level driver for Philips UDA1341TS sound chip used on Compaq iPAQ handheld. This chapter reviews whole process intending to be HOWTO-like documentation for other ALSA driver writers. It discuss creation of driver with basic functionality, first it's mixer part and then PCM part. Then extensions to this simple driver are shown, which enable full control over sound chip. At the end some remaining problems, which are not essential, are discussed. These will be not solved now and should be the goal of future improvements. Whole driver is now placed in ALSA CVS, where is accessible to whole community for testing. It will be merged into Linux kernel later when it will be clear that it works fine.

Chapter 9 Library implementation is to be divided into three parts. To make life of application developers easier, we provide set of functions to setup and open audio device for capturing audio signal. Available functions are described and small and simple example of its usage is included. Another set of functions is offered for audio data acquisition. Whole system is based on one big structure per audio handle that contain all necessary informations for all subsystems. Main part of this chapter is dedicated to software gain control (SGC) implementation and usage informations. Algorithms used are showed in depth. Histogram of maximal signal values in each processed buffer offers statistical point of view on the speech signal used.

Chapter 2

Spoken Language Systems

2.1 Motivation for Communicating by Speech

From forgotten ages to the present and probably to the future, speech communication was, is and will be the prevailing form of human communication. First two people, who would like to communicate using speech, must be on the same place to hear the other one well. As technology extend (with Bell's invention of phone), there is a way to speak to some person everywhere over the world. In this sense, future technological progress preserve the ability of speech communication.

Fifty years ago, when first computers originate, everyone was happy, that the machine can do some work in place of people. None was excited, that to compute few numbers, team of operators have to plug hundreds or thousands of wires to program the monster.

As the time went, managers want to earn money and scientists want to play, this was not enough. To sell the computer to everyone in the world or even to solve difficult problems in reasonable time, you would not like to spend hours with plugging the wires. So the computers began to behave *user friendly.* First programing with punch cards, then graphical interface and what next? As typing onto keyboard is tedious, slow and not natural for human, there should be some next step. To free hands, relax eyes we should involve ears into controlling the machine.

There are generally two categories of users who can benefit from adoption of speech as a control modality in parallel with others, such as the mouse or keyboard. For novice users, functions that are conceptually simple could be directly accessible by voice. For expert users, the GUI paradigm is sometimes perceived as an obstacle or nuisance and shortcuts are sought. Frequently these shortcuts allow the power user's hands to remain on the keyboard or mouse while mixing content creation with system commands. Here, the speech may serve as another input device (used in parallel) leaving your hands free for mouse and/or keyboard. For example with graphical editor you could use speech to select actual tool and mouse with keyboard to use this tool. Switching between individual tools is then possible without move of your hands. To see both positives and negatives, speech driven command and control is not as fast as keyboard shortcuts, so it should be used as supplement to other input devices and should not replace them.

In some situations you must rely on speech as an input or output medium. For example, with wearable computers, it may be impossible to incorporate a large keyboard. The same situation is with pocket computers or advanced mobile phones. There are some possibilities, like software keyboard, but taking a memo with these utilities takes too long. Other experiments, like recognizing of text written with the pen and touch-screen are either occupying too much place on small display or they are too complicated (or both of these) to be usable for all users.

To be rightful, usability for end-user is sometime satisfactory, but to obtain efficiency it require training and some unnatural improvements (i.e. special alphabet of *Graffiti* system used in Palm OS).

Certain manual tasks may also require full visual attention to the focus of the work. For example when taking memo, it's useful to watch the thing you are writing about and think about what you are writing, rather than concentrate on how to store your ideas.

Finally, spoken language interfaces offer obvious benefits for individuals challenged with a variety of physical disabilities, such as loss of sight or limitations in physical motion and motor skills.

2.2 Application of Spoken Language Communication

When we do not focus just on computers, we should find more areas of use for the ability of the machine to communicate with its neighborhood using speech.

A spoken language system has at least one of the following three subsystems: a speech recognition system that converts speech into words, a text-to-speech system that conveys spoken information, and a spoken language understanding system that maps words into actions and that plans system-initiated actions.

Using these subsystems, we should define some classes of usage. Using just speech recognition system from the trinity, we can build dictation. This is one of the best known application of speech recognition. Today real systems exist, but just as relatively simple applications or simple plugins for existing text processing programs. On this application, we should demonstrate one of the main problem within speech recognition. It is dependency on the operator. Because each set of human organs is unique, the sound signal produced by each human will differ from the same words said by anyone else. There is big amount of similarity in the signal, as we know that we are able to understand each other, but it's not simple to find this similarity with algorithms.

Some other tasks to perform with speech recognition is speech driven control of classical computer programs. The difference from the previous one is in the vocabulary, that the recognizer must accept. In dictation, each word must be transferred into its text form, but while controlling application, we should use only (relatively) small set of words or simple sentences to express our wishes. Language model is necessary for dictation whereas grammar is enough for speech driven control. On the other hand, for speech driven control, we would need second subsystem - that is spoken language understanding. In simple form, using few rules for the grammar, describing possible combinations of commands, it's possible to create it just now. But as the vocabulary grows, we are not able to define the grammar in suitable form anymore or it is too difficult.

The ultimate language understanding system should be able to be controlled with spoken commands from very large vocabulary or selectible without any constraints. The system must be able to reply using speech synthesis and be able to understand the context of the speech. For now, this is just a dream, but as the knowledge about all included science discipline grows, prototypes of such systems are better and better, and even now we can imagine the set of presumptions that must be performed to be able to construct and use such systems in daily traffic.

2.3 Human Speech Systems

Mainly from the beginning of the speech recognition and synthesis research scientists were interested in human (or animal) organs to get enough knowledge to simulate the speaking process.

Spoken language is used to communicate information from a speaker to a listener. Speech production and perception are both important components of the speech chain. Speech begins with a thought and intent to communicate in the brain, which activates muscular movements to produce speech sounds. A listener receives it in the auditory system, processing it for conversion to neurological signals the brain can understand. The speaker continuously monitors and controls the vocal organs by receiving his or her own speech as feedback.

Speech is produced by air-pressure waves emanating from the mouth and the nostrils of a speaker. Parts of human speech production apparatus are for example lips, upper and lower teeth, nasal cavity and tongue. All of these are used, but not all for one sound. For each sound, there is only some subset of these organs used and active. The others are either inactive or are used passively without any motion. All sounds could be partitioned into groups based on certain articulatory properties. These properties derive from the anatomy of a handful of important articulators and the places where they touch the boundaries of the human vocal tract. Additionally, a large number of muscles contribute to articulator positioning and motion.

The second part of whole system are organs for speech perception. There are two major components in the auditory perception system: the peripheral auditory organs (ears) and the auditory nervous system (brain). The ear processes an acoustic pressure signal by first transforming it into a mechanical vibration pattern on the basilar membrane, and then representing the pattern by a series of pulses to be transmitted by the auditory nerve. Perceptual information is extracted at various stages of the auditory nervous system.

The human ear has three sections: the outer ear, the middle ear and the inner ear. The outer ear consists of the external visible part and the external auditory canal that forms a tube along which sound travels. When air pressure variations reach this first part from the outside, it vibrates, and transmits the vibrations to bones adjacent to its opposite side. The vibration is at the same frequency as the incoming sound pressure wave. The middle ear is an air-filled space or cavity. The last part of the ear is the most complicated one, but for us, the only relevant information is, that it communicates directly with the auditory nerve, conducting a representation of sound to the brain.

2.4 Elements of Spoken Language

The scientific discipline, that study speech sounds and their production, classification, and transcription is named *phonetics*. Basic building block for speech are than called phonemes.

In speech science, the term phoneme is used to denote any of the minimal units of speech sound in a language that can serve to distinguish one word from another. We conventionally use the term phone to denote a phoneme's acoustic realization.

In most of the world's languages, the inventory of phonemes can be split into two classes:

- *consonants* - articulated with use of constrictions in the throat or obstructions in the mouth
- *vowels* - articulated without constrictions and obstructions

Vowels are characterized by two basic frequencies, major resonances of the oral and pharyngeal cavities, which are called F1 and F2, the first and the second formants respectively. They are determined by tongue placement and oral tract shape in vowels, and they determine the characteristic timbre or quality of the vowel. The relationship of F1 and F2 to one another can be used to describe the vowels. While the shape of the complete vocal tract determines the spectral outcome in a complex, nonlinear fashion, generally F1 corresponds to the back or pharyngeal portion of the cavity, while F2 is determined more by the size and shape of the oral portion, forward of the major tongue extrusion.

Consonants, as opposed to vowels, are characterized by significant constriction or obstruction in the pharyngeal and/or oral cavities. Some consonants are voiced, others are not. Many consonants occur in pairs, that is, they share the same configuration of articulators, and one member of the pair additionally has voicing which the other lacks.

Phonemes are small building blocks. To contribute to language meaning, they must be organized into longer cohesive spans, and the units so formed must be combined in characteristic patterns to be meaningful, such as syllables and words.

2.5 Speech Signal Representation

As mentioned in previous paragraphs, the last part of speech recognition process, we will study more detailed is creation of characteristic flags vector for each speech element. This is the goal of signal representation techniques. Now we should look on two representatives which are used for speech language systems. It looks, that cepstral analysis is one of the most useful, and we mention LPC at least to compare both methods. They are based on speech production models. We have other ones, inspired by speech perception models, like bilinear transform or perceptual linear prediction. The other class is not discussed here anymore.

Basic signal processing is common for both presented methods. We want to transform time spectrum representation of signal to another based on frequency. We would like to use Fourier transform, but there are some problems. First - for FT we need periodic signal. Speech signal is periodic, but only in small regions. Using large regions prohibit usage of FT. Second - exact definition of FT requires knowledge of the signal for infinite time. For both reasons, a new set of techniques called *short-time analysis*, are proposed. These techniques decompose the speech signal into a series of short segments, referred to as *analysis frames*, and analyze each one independently.

2.5.1 Speech Coding

To be able to work with audio signal in computer, we need to digitally encode it. Digital storage of audio signals, which can result in higher quality and smaller size then the analog counterpart is commonplace in many areas of interest. We should mention one of the best known example - *mp3* file format with MPEG compression. The goal is to have the best quality using the smallest possible space on storage medium. And this is not problem of just storage media. When transferring such data over network, small bit rate is very significant. On the other hand, using such good compression level means, we need quite powerful hardware to encode and decode it. In spoken language systems, we do not need to store the signal completely, often the first thing we do is reduction of information included in the signal to necessary minimum and then we are working with just a fraction of original data. But even with these conditions, we need to select some coding for acquired or produced data. We should focus of methods with low cost. Most popular are various modifications of scalar waveform coders.

We will not need to determine, which coding to use for acoustic speech signal as this is given (or at least limited) by hardware used, but same coding techniques should be used to store vectors with flags from FT and/or other transformations.

Linear Pulse Code Modulation

Analog-to-digital converters perform both sampling and quantization simultaneously. Quantization is the process, which encodes each sample with a fixed number of bits. Linear *Pulse Code Modulation* is based on the assumption, that the input discrete signal $x[n]$ is bounded and that we use *uniform quantization* with quantization step size Δ which is constant for all levels x_i:

$$\mid x[n] \mid \leq \; X_{max}$$

$$x_i - x_{i-1} = \Delta$$

The output $\hat{x}[n]$ should be obtained from the code and step through

$$\hat{x}[n] = c[n]\Delta$$

Being so simple, it's commonly implemented directly in hardware, and used in many simple formats for sound storage (*.wav, .au, .snd, .aif*). There is a theory about quality of audio coder. The main result of it is property called *signal-to-noise ratio* (*SNR*). This give us the ratio between the sound signal energy and noise energy (in dB). For pure PCM each bit contributes to $6dB$ of *SNR*. For communications systems $37dB$ of *SNR* is acceptable, and that needs 11 bits to store one sample. When selecting format parameters, we can scale either number of bits per sample or sampling frequency. For some simple recognizer, 8 bits should be enough, but most sound systems and sound formats use 16 bits as default value.

μ-law and A-law PCM

Human perception is affected by SNR, because adding noise to a signal is not as noticeable if the signal energy is large enough. Ideally, we want *SNR* to be constant for all quantization levels, which requires the step size to be proportional to the signal value. This can be done by using a logarithmic compander - nonlinear function that compands one part of x-axis - followed by a uniform quantizer.

$$y[n] = ln \mid x[n] \mid$$

$$\hat{y}[n] = y[n] + \epsilon[n]$$

This type of quantization is not practical, because an infinite number of quantization steps would be required. An approximation is the so-called μ-law compander which is approximately logarithmic for large values of $x[n]$ and approximately linear for small values of $x[n]$. A related compander called A-law is also used which has greater resolution than μ-law for small sample values, but a range equivalent to 12 bits. In practice, they both offer similar quality.

Adaptive PCM

When quantizing speech signals we confront a dilemma. On the one hand, we want the quantization step size to be large enough to accommodate the maximum peak-to-peak range of the signal and avoid clipping. On the other hand, we need to make the step size small to minimize the quantization noise. One possible solution is to adapt the step size to the level of the input signal.

We should let the step size $\Delta[n]$ be proportional to standard deviation of signal $\sigma[n]$.

$$\Delta[n] = \Delta_0 \sigma[n]$$

In practice it is advantageous to set limits on the range of values of $\Delta[n]$ where the ratio of $\Delta_{min}/\Delta_{max}$ should be 100 to obtain a relatively constant *SNR* over $40dB$. This scheme require to transmit the step size with the signal, but it's values evolve slowly in time and could be encoded effectively with low bit rate.

2.5.2 Linear Predictive Coding

Linear Predictive Coding is also known as LPC analysis or *auto-regressive* modeling. This method is widely used for its simplicity and speed. On the other hand, is should effectively estimate main parameters of speech signals. LPC predicts the current sample as a linear combination of its past p samples.

An all-pole filter with a sufficient number of poles is a good approximation of speech signals. When modeling speech creation as signal $X(z)$ originated from some basic signal $E(z)$ and modified by vocal tract acting like signal filter $H(z)$, we should write:

$$H(z) = \frac{X(z)}{E(z)} = \frac{1}{A(z)}$$

Inverse filter $A(z)$ is defined as

$$A(z) = 1 - \sum_{k=1}^{p} a_k z^{-k}$$

Using inverse z-transform we should take

$$x[n] = \sum_{k=1}^{p} a_k x[n-k] + e[n]$$

Let s define $x_m[n]$ as a segment of speech selected in the vicinity of sample m:

$$x_m[n] = x[m+n]$$

To estimate the predictor coefficients from a set of speech samples, we use the short-term analysis technique. given a signal $x_m[n]$, we estimate its corresponding LPC coefficients as those that minimize the total prediction error E_m.

$$E_m = \sum_n e_m^2[n]$$

Taking the derivative of this and equating to 0 we obtain:

$$< e_m, x_m^i >= \sum_n e_m[n] x_m[n-i] = 0$$

This last equation should be expressed as a set of p linear equations. Solution of the set of p linear equations results in the p LPC coefficients that minimize the prediction error. The solution of equations can be achieved with any standard matrix inversion package. Because of the special form of the matrix here, some efficient solutions are possible. As a result of LPC analysis we obtain vector of p flags. The higher p, the more details of the spectrum are preserved.

2.5.3 Cepstral Processing

Cepstrum is one homomorphic transformation that allows to separate the source from the filter. Complex cepstrum of signal $x[n]$ is defined as

$$\hat{x}[n] = \frac{1}{2\pi} \int_{-\pi}^{\pi} ln X(e^{j\omega})\ e^{j\omega n}\, \mathrm{d}\omega$$

where complex logarithm is used and real cepstrum of signal is defined as

$$c[n] = \frac{1}{2\pi} \int_{-\pi}^{\pi} ln \mid X(e^{j\omega}) \mid e^{j\omega n}\, \mathrm{d}\omega$$

It can be shown, that $c[n]$ is the even part of $\hat{x}[n]$:

$$c[n] = \frac{\hat{x}[n] + \hat{x}[-n]}{2}$$

For speech signal representation both real and complex spectrum could be used, but properties of real cepstrum are better, so it's much wider used. Problems with complex cepstrum are caused by computing complex logarithm - specially the phase. We will show two methods of computing cepstral coefficient, which are the representation of speech signal (they represent filter - vocal tract - and excitation - air flow at the vocal cords).

We can compute the cepstrum of speech segment by windowing the signal with a window of length N and using DFT as follows:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}$$

$$c[n] = \frac{1}{N} \sum_{k=0}^{N-1} ln \mid X[k] \mid e^{-j2\pi kn/N}$$

Second and much more used variant of computing cepstral coefficient uses results of previously mentioned LPC analysis. Updating LPC filter equation with gain, we obtain

$$G = E_m$$

$$H(e^{j\omega}) = \frac{G}{A(e^{j\omega})}$$

By some terrible mathematical modification (logarithm, derivation, and others) we should obtain following equations for computing cepstral coefficients from LPC coefficients recursively. While there are finite number of LPC coefficients, the number of cepstrum coefficients is infinite. Fortunately, finite number of cepstral coefficients (from 10 to 20) is sufficient to obtain reasonable results.

$$c[n] = \begin{cases} a_1 & \\ a_n + \sum_{k=1}^{n-1} \left(\frac{k}{n}\right) c[k] a_{n-k} & 0 < n \leq p \\ \sum_{k=n-p}^{n-1} \left(\frac{k}{n}\right) c[k] a_{n-k} & n > p \end{cases}$$

As with LPC we got vector of flags $< c_0 \dots c_p >$, that can be used for further recognition. Second presented technique of computing cepstral coefficients is more used in todays recognizer, because using definition equations and DFT is performance hungry.

2.6 Speech Recognition

At this moment we should have vector of flags, generated by some speech analysis. These vectors should be normalized and fixed length (though not necessary). Having some trained vocabulary, we would like to map some unknown sample into the vocabulary of trained (learned) samples (words or other sized utterances). We should now look on at least one of most used techniques to get trained vocabulary and to map unknown samples to this vocabulary.

2.6.1 Hidden Markov Models

About 10 years ago, two main techniques were used for speech recognition. First of them is Dynamic Time Warping (DTW) and the theory of dynamic programming. The second one was Hidden Markov Models (HMM). DTW is based on calculating overall distortion which is computed from accumulated distance between two vectors with flags - one which we would like to recognize and the other from the vocabulary. Then we should say, that vocabulary sample with smallest distance to tested sample is the right one.

Speech recognition based on DTW is simple to implement and fairly effective for small-vocabulary speech recognition. Dynamic programming can temporally align patterns to account for differences in speaking rates across speakers as well as across repetitions of the word by the same speaker. However, it does not have a systematic way to derive an averaged template for each pattern from a large amount of training samples. A multiplicity of reference training tokens is typically required to characterize the variation among different utterances. As such, the HMM is a much better alternative for spoken language processing.

Todays, it looks, that HMM is one of the most used methods for speech recognition. The HMM is powerful statistical method of characterizing samples of a discrete-time series.

Let $X = X_1, X_2, \dots, X_n$ be a sequence of random variables from a finite discrete alphabet $O = \{o_1, o_2, \dots, o_M\}$. The random variables X are said to form a first order Markov chain provided, that

$$P(X_i \mid X_1^{i-1}) = P(X_i \mid X_{i-1})$$

where $X_1^{i-1} = X_1, X_2, \ldots, X_{i-1}$. If we associate X_i to a state , the Markov chain can be represented by a finite state process with transitions between states specified by the probability function $P(X_i = s|X_{i-1} = s')$. Consider a Markov chain with N distinct states $1, \ldots, N$, with the state at time t denoted as s_t. Parameters of such model should be described as follows:

$$a_{ij} = P(s_t = j|s_{t-1} = i) \qquad 1 \leq i, j \leq N$$

$$\pi_i = P(s_1 = i) \qquad 1 \leq i \leq N$$

where a_{ij} is the transition probability from state i to state j and π_i is the initial probability that the Markov chain will start in state i. Markov chain described below is called observable Markov model - the output of the process is the set of all states in each time and each state correspond to a deterministically observable event.

A natural extension to the Markov chain introduces a non-deterministic process that generates output observation symbols in any given state. Thus, the observation is a probabilistic function of the state. This new model is known as a hidden Markov model.

A hidden Markov model is basically a Markov chain where the output observation is a random variable X generated according to a output probabilistic function associated with each state. There is no longer a one-to-one correspondence between the observation sequence and the state sequence, so you cannot unanimously determine the state sequence for a given observation sequence; i.e., the state sequence is not observable and therefore *hidden*.

Now, when we know, how HMM works, we need to solve to basic problems. We need to learn the HMM and we need to evaluate HMM. Evaluation should be defined as probability of the model, that it generate given sequence of observations. Learning means modifying parameters of HMM to maximize the probability for given model and observations. If we could solve the evaluation problem, we would have a way of evaluating how well a given HMM matches a given observation sequence. Therefore, we could use HMM to do pattern recognition. To be able to use this for speech recognition we must be able to decode the state sequence, as that is the requested information giving us recognized element.

For evaluation we should use *Forward algorithm* for decoding *Viterbi algorithm* and for learning *Baum-Welch algorithm*. They are quiet complicated, so you should use another literature to learn how they work (i.e. [2] or [3]).

And now, how to use HMM in speech recognition. We simply model elementary units with hidden Markov models, and by evaluating models, we get the probability for each of them of modeling used sample. Then we should select the most probable one, or more models. Then we construct larger objects from these elements using another rules (represented in many cases with grammar rules). Elements to be modeled with HMM should be words in simple cases with small vocabulary or smaller entities like phonemes or sub-phoneme units to be able to use larger vocabulary without complicated training. Complicated training is probably the only disadvantage of this method of speech recognition. This can be partially solved by using smaller units for models. Phonemes are fine, but such models achieve somewhat lower precision of recognizing by lose some coarticulation information from the speech. This can be solved with even smaller unit - sub-phonemes like alophonemes being the basic unit for hidden Markov models.

Chapter 3

Embeded ViaVoice

3.1 Introduction to EVV

IBM has two product lines for spoken language systems. IBM ViaVoice suite is designed for personal computers and workstations. It offers complete spectrum of speech services, from speech recognition to TTS. It allows continuous speech recognition. It is intended to be used on systems with sufficiency of computation power and fast and large storages. As the opposite, Embeded ViaVoice - EVV is dedicated to another group of system. Embedded in its name means at first less powerful. Compaq iPAQ, that is used in this project, could be marked as embedded device, but with its equipment and performance, it should be called embedded workstation or embedded mainframe. Compared to workstations or PCs, it is clear, that performance required by EVV must be highly reduced against ViaVoice system. With almost 140 MIPS and suitable fast Flash memory running EVV thread takes about 5 to 15 percent of CPU time. Second and in some cases more painful problem is data stream bandwidth. EVV directly works with one channel of audio stream with 16-bit samples at 11025Hz sample rate. That means in other words 22KB/s continuous data stream. Again for PCs, this is not problem, and it's not problem with iPAQ, but for some embedded devices, this could be the bottleneck. And third, about 600 to 800KB of memory is required for data of recognizer and audio data buffers. It is necessary to have on mind, that there must be enough performance, data bandwidth and memory for user's applications, because having just EVV thread is not very useful (except for simple demo or testing). All these limitations are achieved by allowing to recognize just limited number of phrases from strictly defined vocabulary. This is the main difference against ViaVoice, where continuous speech recognition could be achieved. Both ViaVoice and EVV products are distributed as libraries with some additional utilities (e.g. grammar compiler and data files).

3.2 EVV Modules

Whole EVV API could be divided into four modules at the moment. Modularization is used to separate logically independent parts of the API and thus allow easier development of EVV and make its interface more user friendly.

1. *EVV Control* - this part of the API is used to initialize, start and stop whole engine. It connects and disconnects audio stream from/to EVV services. It is responsible for installing and uninstalling of services. Furthermore, it adds and removes listeners. Listeners are callback functions used by EVV to notify application about errors, speech detection, successful recognition or rejection of current utterance.

2. *Vocabulary Manager* - all vocabularies are shared between instances of EVV engine and between multiple services. The vocabulary is a set of acceptable words and their sentences defined by grammar. It is stored in binary file, that is created by grammar compiler. To allow sharing of vocabularies to be done effectively, Vocabulary Manager (VM) is designed separately from the services. All data for EVV are stored in binary images produced by off-line compilers of grammars. Second image you need contain baseform oriented data for EVV. The application calls VM functions to register grammars only once. VM manages all images and allows to manipulate with them using handles. Handles are obtained from registration functions. Using these handles, you could then register and enable grammars to recognizer engine.

3. *Audio* - same as Vocabulary Manager, Audio handling was separated from recognition engine. EVV does not connect directly to some audio source (i.e. sound card), but instead leave application to provide audio data from arbitrary source. This could be used for multiplexing of both services and audio sources. Audio module does just encapsulation of internal buffers (audio queues) for both services and application. Internal buffers brings some additional overhead against zero-copy solution, but allow the application to be more independent. Each audio instance could be connected to one or more services.

4. *Services* - available version of EVV API supports multiple services to be used. There is some part of API, that is common to all services. Every service could be started, stopped or paused. Each service must be first installed and then started. This allows to install all requested services, but use just a subset of them at a moment. Some services could share parts of the engine. For example signal processing unit is used by all (or most of) services. By sharing, audio data buffers are processed only once, that saves system resources. The API offers functions to enable and disable routing of audio stream to particular service, so the application could have well-grained control over the engine.

3.3 Available Services

First available and mostly used service is recognition. The recognition service API is used for installing, uninstalling, enabling and disabling vocabularies. Application is not allowed to modify actual set of used vocabularies arbitrary. Set of vocabularies should be modified (i.e. expanded or shortened) only after successful or unsuccessful recognition. In practice, this mean in *decoderListener* callback or after (before) service stop (start). Another possibility to change vocabularies offers *pause* and *resume* pair, that is able to suspend service and resume it after requested changes. Apart of *decoderListener* callback, some other interfaces may be used by the application. *errorListener* is designed to inform the application about errors of whole engine. For single threaded applications, *idleListener* and *peekListener* may be used to perform another work when EVV has nothing to do (*idle*) or regularly while data processing (*peek*). For each utterance, whole work is divided into small peaces. These are queued and then performed one per step. At the end of each step, *peekListener* is called. For good interactivity, *peek* code should return immediately to end speech processing in short time after utterance. Recognition could be used in two modes. For some application, *push-button mode* is suitable. In this mode, EVV expect, that user mark start and end of utterance by pressing and releasing some (virtual) button. Having precise knowledge about start of utterance, successful recognition ratio of recognizer increases significantly. Second - *Always-speak mode* is used in those cases, where we do not have any "button". Usage of this mode results in worse successful recognition ratio.

Second service is called *Acoustic baseforms*. Current engine allows application to use speaker independent recognition with vocabulary with about thousand of active words. These words are prepared off-line (i.e. statically) in the form of binary image that is used by application. Sometimes it is convenient to be able to resize or modify used vocabulary on-the-fly. This service allows the user to pronounce some word, process this word and add it to set of vocabularies. User must provide the value of newly defined utterance, that means spelling of the utterance in most cases.

3.4 Silence Detector

Being important for this project, we should now focus on silence detector. At first glance, task of silence detector is simple. It just mark segments of audio signal as either speech or silence. This information is then used in following stages to save CPU cycles and bandwidth. Having just two output states, the detector could not exactly says if the signal is silence, something between or speech. It does not recognize general noise from speech signal well too. Resulting information is that actual signal is closer to either speech or silence. Any probability nor other informations are provided. Informations from silence detector are used internally by following stages and are available to application via *silenceDetectorListener* callback.

Chapter 4

Algorithms for SR Improvement

One of the most important part of this project was devising of algorithms which allow improving of accuracy of used recognizer. In this chapter, we will look at concept of introduced methods. Detailed description of their implementation, API and application could be found later in this thesis (in chapter 9). All improvements are designed mainly for iPAQ and UDA1341TS chip, but could be usable for other sound devices as well. Some useful information about abilities of UDA1341TS may be found in chapter 8 and appendix A.1.

4.1 Input Gain Control

As briefly described in section 8.5, the UDA1341TS chip contains *Automatic Gain Control (AGC)* logic. This piece of hardware can be used to automatically modify some mixer controls to obtain signal with predefined maximal amplitude. This may be used for recording to partially hide changes in speaker's volume. There are several controls to set requested behavior of AGC.

AGC timing constants - this is just one control, but there are two independent timings encoded inside. In UDA1341TS data sheet [11] you could see table with detailed specification. First timing constant is *Attack time*. It ranges from 11 to 21 ms and it says how long is the window used to determine, that signal amplitude is either lower or higher then expected value. In another words, this determine, how long will the logic wait, before it decides to change level of internal input gain control. The second timing parameter is *Decay time*. This one says how long it will take to modify internal input gain controls to obtain requested signal value. It could be set in range from 100 to 400 ms.

AGC output level - with this control, you can set requested level of basic input gain. It is proportional to initial setting of internal input gain control. It should be seen as prefixed control decreasing input gain in ranges from -9dB to -17.5dB.

Mixer sensitivity level - is intended for manipulating with final volume. You could set requested level of recorded signal and AGC will try to initiate its state to obtain it.

Again, please refer to UDA1341TS data sheet [11] for precise values, combinations and meaning of all referred controls. I am not sure, what should be AGC used for exactly. For speech recording (i.e. dictaphone) it can be useful but is it good for speech recognition? Idea of the algorithm used is clear, but as you can imagine without deeper knowledge, for speech recognition, this is not very useful. Let imagine this situation. You are walking with your iPAQ down the street and are trying to command it through speech. At some moment a tram or noisy big car drive near around you. With AGC enabled, level of input gain is lowered because of this newly recognized (or better recorded) noise (or in fact just another level of input signal). With the same microphone sensitivity, level of your recorded speech will decrease dramatically. At first glance, this doesn't matter at all. You will be speaking either with base tone of voice resulting in rejection from speech recognizer or you will try to solve this situation by increasing tone of your voice. In this second case, you will got the same result (rejected) from the recognizer probably, because distance of speech signal from noise would be probably not sufficient. As the second miss, you will decrease tone of your voice some measurable time after the problematic noise is over. Seeing from either first or second side, the recognizer will feel dizzy.

Well, we have some reasons why AGC is not the right choice. But something like this will be useful. Have another simple example. While watching the town, you were holding the iPAQ in front of your mouth in distance about 20 or 30 centimeters. Expect, that signal recording is optimal for this "configuration". Now you enter your office and place your iPAQ into its sleeve. This sleeve will be placed somewhere on your table, but rarely directly before you in some reasonable distance. Some corner of your table is more realistic forecast of its placement. Now speaking with the same intensity as before, recorded signal will have much smaller amplitude. You may notice, that in (relatively) calm office (e.g. after the dinner) you will seldom need to speak so aloud. But recognizer likes its level of signal to do its best, so some mechanism to modify input gain would be useful.

From both examples the conclusion should be, that we must care about the signal only if it is speech. Second idea should be, that we need to change input gain level much more slowly the AGC to have resulting signal smooth without great peaks from trams and trucks. Now let have a look at the problem of speech boundary detection. This is what we need - to find speech (or utterance) in input signal and handle input gain according to this signal.

4.2 Start and End of Speech Detection

In part about recognizer description we mentioned silence detector (section 3.4). This should be ideal instrument for speech boundary detection. Unfortunately, it could not be used at all. Really it should be, but it does not help us. Where is the problem? For speech driven command and control, that is main target on embedded platforms, we would like to use short and descriptive utterances. They should consist of one (in best case) up to three words to be still useful. As an example, let imagine speech controlled graphics program like Adobe Photoshop or GNU Gimp. We would like to select tools with speech control, because tool bar is to far away from workspace and keyboard is too outdated for us. Selecting new tool with *"I would like to use a pencil"* seems like bad idea to me. Just *"pencil"* is enough. This is just stupid example, but to be usable, speech command and control must be as quick as possible because even then it is still n-times slower then keyboard, mouse and other traditional input devices.

So let's dig deeper. Having about one second long utterances results in starting with processing of input signal when utterance is over. We are signaled about speech start just when signal level is back on silence level. When we use current level to compute input gain control update, whole process idea would be inverted. This is not the right way. We must choose. Either implement speech signal detection ourselves or got more information from recognizer. The second possibility looks easier. We could forgot all signal processing if we should got just value of peak of previous utterance. Then we could control input gain with just this one value reasonably and everything would be fine. But in the real world, we do not have information like this.

We rather implement our own speech/silence detector. Now we need to think a bit more about our needs. Silence detector from recognizer was discussed above and in previous chapter. Main problem with this solution was unpredictable delay of detected state. But there is another problem. We suppose, that two-state information in the form “is silence” or “is not silence” is enough. In *silenceDetectorListener* we could obtain one of two predefined types of transitions. Either *speech-to-silence* or *silence-to-speech.* This sounds good, but in practice, naming these events *start-silence* and *end-silence* would be more realistic. For information whether *non-silence* was (with some probability) speech, we need to bring next occurred *decodedListener* event to game. These two events would be enough to decide if actual signal should be used to compute input gain modification. But as is, the implementation of this model us unusable because of non-deterministic delay between signal occurrence and both events.

The question now stands on: Are we able to implement the optimal model? Of course we are, but we will need our own speech recognizer (but do not need silence detector). It does not take sense to solve it this way. For now we will implement some mechanism to decide at our best whether actual signal is important for input gain modifications of not. We take no care of speech recognition at all to be independent on used speech recognizer. Our goal is to create algorithm, which will select interesting signal (where interesting means speech signal) without interaction with recognizer. Actual implementation is described in chapter 9.

4.3 Large Samples Usage

The UDA3141TS chip offers one additional feature, which should be used to improve results of speech recognizer. There are three sample formats that are important for us. All referred formats are signed little endian integers and differ only in sample size. Each sample can be 16, 18 or 20 bits large. From both empirical and exact studies we know, that sample rate for speech recognition varies in range from 8000 to 11025Hz. Less is not enough to properly reconstruct speech signal and more is wasting of memory and bandwidth. Similar information is available for sample size. 8 bits is minimal requested size and 16 bits is maximal acceptable (for powerful system like workstations or PCs both parameters could be greater but improvement of recognizer ratio is not considerable). This means, that we would not like to provide 20-bit samples to recognizer, but instead we create 16-bit samples in all cases.

This additional processing may seem as unhelpful generating just more CPU load, but we show some sense in cost paid. In the first place, we get better resolution for our signal processing. But that is not very important. The second reason is more important. We get the ability of digital amplifying of speech signal (or better signal for speech recognizer). This should be important at most in situations where we got weak signal regardless of maximal input gain of microphone. In this situations we may provide just lower 16 bits of wider sample to increase amplitude of signal for recognizer. In fact, lower 16 bits are used always, but for normal signal we must shift the content right few times (until the mean significant bit moves on the position of MSB of 16-bit word). Performing less divisions, we strengthen the signal. This technique should improve results of recognizer significantly in marginal situations. Unfortunately it was not implemented, because of complexity of changes needed for ALSA system. It should be added later as part of following driver improvements.

Chapter 5

Linux Sound Subsystems

5.1 Sound Subsystem Basics

Being just another piece of hardware, sound cards needs some support from operating system to be useful for application programmers. Sound cards normally have several different devices or ports which are able to produce or record sound. There are differences between various cards, but most have the devices described below.

- The digitized voice device (also referred to as a codec, PCM, DSP or ADC/DAC device) is used for recording and playback of digitized sound. This is one of the most used part of each sound card, present in most music playing like playback of mp3s or recording sound via microphone. The main element of this part is analog-digital converter for recording sound and digital-analog converter for playing digital sound formats. On digital side of this device, sound is stored as set of samples created by ADC/DAC. Basic parameters for configuration of this device is sample rate, number of samples taken per second which could vary from as low as 8kHz up to 48kHz (or 96kHz for professional hardware) and depth of samples, number of bits used to store one sample. Another parameter is number of channels of sound. For some application one is enough - we have mono sound, mostly used is stereo sound - two channels and for realistic sound up to five channels may be used. As present in almost every card (except some special purpose or customer cards), each driver contains set of routines to manipulate with this device and to configure it. For speech recognition this is the most useful device (together with mixer).
- The mixer device is used to control various input and output volume levels. The mixer device also handles switching of the input sources from microphone, line-level input or CD input. Cheaper cards are able to use one source for input and one for output at any time. Better card support receiving or playing two or more streams at once. For the others, there is software support to do this. Being another very useful part of sound cards for speech recognizing and for any other usage, it will be second point of interest in our driver. Except from traditional mixer manipulation as known for all sound cards, UDA1341 has some extended support for gain control. Control code for this special hardware abilities will be the basic goal of our efforts. As output sound subsystem on iPAQ is poor without headphones, we should focus on sound recording.
- The synthesizer device is used mainly for generating music. It is also used to generate sound effects in games. One of the well known synthetizer devices is the Yamaha FM synthesizer chip which is available on most sound cards. Another type of synthesizer devices are

the so-called wave table synthesizers. These devices produce sound by playing back pre-recorded instrument samples. This method makes it possible to produce extremely realistic instrument timbres. Again this is not very useful for speech recognizing, but programmable wave table could be successfully used with speech synthesis.

- A MIDI interface is a used to communicate with devices, such as external synthesizers, that use the industry standard MIDI protocol. MIDI uses a serial interface running at 31.5 kbps which is similar to (but not compatible with) standard PC serial ports. The MIDI interface is designed to work with on-stage equipment like external synthesizers, keyboards, stage props, and lighting controllers. MIDI devices communicate by sending messages through a MIDI cable. For speech recognition this is not very useful device, even it should be used for speech synthesis with external speech synthetizer.

- Most sound cards also provide a joystick port. Some older cards (introduced before ATAPI interface for CD-ROM was devised) contain interface (IDE, SCSI, or proprietary) for a CD-ROM drive. These devices are not controlled by sound card driver and are not wide spread todays.

5.2 Open Sound System

The Open Sound System (OSS) is a device driver for sound cards and other sound devices. OSS was derived from the sound driver written for the Linux operating system kernel years ago. OSS is commercial product used for variety of operating systems. It's main feature is portability of applications written using OSS API. Communication and controlling of sound devices is based on usage of traditional Unix special files for each feature of sound card. As so, the API is very simple using just about five functions to manipulate devices. Advantage of this practice is simplicity of applications. They fully support traditional Unix style of thinking. For everything, there are just files. To record some sound you should use command

```
cat /dev/dsp > my_sound.raw
```

To play this sound you will just copy it back to special file */dev/dsp*. Well, in most cases this is poor theory and is not usable in practice. If nothing else matter, almost nobody would like to use raw data samples. Controlling of sound card devices (like mixer) is not trivial this way. But on the other hand, you don't need any special purpose library or services available just in one operating system. The principle of special files is well known and accepted over variable different platforms. Being so simple it could be emulated in cases when special files are not supported.

As you can presume, the functions used in OSS API are *open(), close(), read(), write()* and swiss-knife *ioctl()*. The remaining functionality of OSS API is implemented using large header files with definitions of constants for *ioctl()* call. This is nearly absolutely portable, but (without some exceptions) this API does not offer any advanced services. Using so much constants, our source code becomes a bit unreadable quickly.

For its missing advanced services and specially for future evolution of Linux sound oriented on following sound subsystem this will not be our choice and therefore we don't need to describe it in details.

To avoid mismatch, we need to mention, that there are two "versions" of OSS. On web page *http://www.opensound.com* you should find basic drivers. Commercial subject (*4Front Technologies*) is maintaining these drivers same as documentation and other support. These drivers are fully OSS compliant and should support all features of OSS. Sound drivers shipped with Linux kernel are the second “version”. The are maintained independently and they are called OSS drivers, because they implement OSS interface. For most of application, small subset of OSS functions is needed. Linux kernel drivers implement just */dev/snd* and */dev/mixer* interface in most cases. Rarely MIDI interface is supported too. Other features are either not supported at all or their implementation is limited.

5.3 Advanced Linux Sound Architecture

Originated from specialized sound driver, Advanced Linux Sound Architecture (ALSA) is similar to OSS in its evolution. From beginning it was designed to be modular, scalable and flexible. ALSA is compatible with OSS sound drivers using OSS emulation layer modules, but it’s application interface is even better. As the main difference, there is not so strong pressure for portability, so more programmer friendly interface is used. ALSA package should be divided into parts. Replacement of OSS drivers are ALSA kernel level drivers for sound cards. Middle level is made by ALSA user-level library that provide C language API to application programmer. Higher level forms basic utilities for manipulating with mixer, playing sound and manipulate some sound card’s specific features.

The lowest level will be at highest interest while writing sound card driver, so take a bit closer look on it.

5.3.1 Special Files

As OSS does, ALSA uses special files to interact between kernel space code (sound cards drivers) and userland code (library and user written application). Here we will discuss the difference in special file naming and usage. First short overview of OSS special files.

`/dev/sndstat`	human readable information about sound card
`/dev/mixerX`	setting of volume and source of sound
`/dev/dspX,dev/audioX`	digitized voice - record and playback
`/dev/sequencer,/dev/music`	synthetizer
`/dev/midi`	raw MIDI interface

Table 5.1: OSS special files

As mentioned in the previous section, OSS uses special files directly. Most frequently used are */dev/mixer* and */dev/snd*. On the other hand, most drivers does not implement */dev/sndstat* at all. *X* in file names is number. It’s used to distinguish between more devices on one board. Sound card should have two mixers for example and they will be named */dev/mixer0* and /dev/mixer1. Same situation is with */dev/dsp* and */dev/audio*. These two device types are very similar. They only differ in used default encoding after opening them (8-bit unsigned linear encoding versus μ-law with resolution of 12 or 16 bits). The */dev/audio* device is provided for compatibility with sound devices introduced in Sun’s workstations running SunOS.

For ALSA another naming scheme was devised. Special files used by ALSA should be divided into two major classes. First ale global devices and second are devices which should be present multiple times. Multiple times should be per a card or per a device. All special files are listed in following table.

`/dev/snd/seq`	synthetizer device (global)
`/dev/snd/timer`	global timer device (global)
`/dev/snd/controlC`*n*	control device for card *n*
`/dev/snd/mixerC`*n*`D`*m*	mixer *m* of card *n*
`/dev/snd/pcmC`*n*`D`*m*`p`	digitized sound device *m* of card *n* - playback
`/dev/snd/pcmC`*n*`D`*m*`c`	digitized sound device *m* of card *n* - recording (capture)
`/dev/snd/midiC`*n*`D`*m*	MIDI and synthetizer interface for raw access
`/dev/snd/hwC`*n*`D`*m*	hardware dependent device *m* of card *n*

Table 5.2: ALSA special files

As you can see, names are more complicated, then ones in OSS. The system is better, because you can precisely say, what is the device for and where it may be found. ALSA fully support *devfs* - device filesystem. For now, to offer compatible solution, device files are created under */proc/asound/dev* and symbolic link is made here as */dev/snd*. Difficult naming convention does not matter, because it's not used directly by application programmer. There is user level library to access ALSA API from applications. So the code, where these names are used is only inside ALSA library.

5.3.2 ALSA Driver Package

Whole ALSA project is well modularized and divided into (actually) four (relatively independent) packages. On the lowest level, in the kernel-land, operate device drivers for individual cards. They are implemented as kernel modules, and this package has strong dependencies on kernel source code. After significant changes in kernel, ALSA developers need some time to upgrade driver package to be conforming with new kernel. This should be one of the most obvious reason for integration of ALSA into Linux kernel.

Common kernel module for ALSA sound system creates *asound* entry in *proc* filesystem. In underlying files and directories we should find detailed information about actual hardware and software configuration.

Other common modules are dedicated to implement OSS compatibility. They observe special files for OSS and transform their requests to ALSA API calls which are then dispatched to ALSA library.

Main part of driver package makes device drivers for supported cards. Each driver should ideally consist of two modules. There should be one module for each chipset and one module for each sound card (or sound card set) using this chipset. This should avoid repetition of code in drivers for cards with same chipset and just few differences. For driver package there exists strict naming convention for file names, functions and data objects. Module for sound card *X* should be named *snd-X.[co]*. Header file should be named *X.h*. Exported (and in best case all) functions should follow the convention *snd_X_something()* and defined types *snd_X_something_t*. And at the end just list of sound kernel modules:

snd.o, snd-mixer.o, snd-pcm.o, snd-midi.o, snd-timer.o, snd-hwdep.o

5.3.3 ALSA Library Package

We can look on ALSA driver package as substitution of OSS drivers from Linux kernels. It will be possible to program user level applications with just kernel modules of ALSA. But this is not the goal of ALSA developers. They provide specialized library even for low level programing. Other layers of abstraction should be added with sound systems like *esound daemon* or *arts*. We should focus ourselves on two parts of ALSA library interface. In the center of our interest, there will be PCM (digital audio) interface of ALSA library. We should say something about configuration possibilities too.

We should look on configuration abilities of ALSA. Whole configuration is designed to define data structures required for ALSA library and core. You can modify (or overload) big set of runtime parameters. Syntax used is relatively intuitive but configuration files are huge, so it could take time to make sense of it. Simple example of ALSA configuration language possibilities follows. Expressions in square brackets are optional:

```
# Include a new configuration file
<filename_to_include>

# Simple assign
name [=] value [,|;]

# Compound assign
name.name1 [=] value [,|;]

# Array assign
name.0 [=] value0 [,|;]
name.1 [=] value1 [,|;]
```

Figure 5.1: ALSA configuration file syntax

This is just very simple example to gain basic overview. To learn more, read ALSA library documentation (for version 0.9.x) [9]. Basic configuration file is *$prefix/share/alsa/alsa.conf*, where *$prefix* is either */usr* or */usr/local* or any other directory selected when compiling ALSA package from sources. Main configuration file is used to set default values for basic data structures for all used interfaces (PCM, mixer, hwdep, timer, MIDI). It includes specialized configuration files for single sound card models.

Default values of all configuration files are fine for most users (basically all except some ALSA developers).

For PCM interface, there is no sense in listing all supported functions. Reference documentation generated directly from source codes is quite good (but it does not describe referred objects deeper). I would like to focus on basic different in programming style for ALSA using library API and for OSS using special files directly. This should be best viewed on another examples, which have no sense at all, but demonstrate the difference well. To have smaller code, all test for errors were removed. First code is for OSS sound system (Figure 5.2).

Now similar (but not equal) code for ALSA sound system using C library API (Figure 5.3). The main difference is, that instead of many symbolic constants, we have many functions. Question, if code for ALSA is more readable is topics for a discussion. Main gain should be the fact, that for OSS every operation is call to kernel and all code is placed in kernel level. For ALSA significant parts of code should be placed directly in library and for some functions no system calls are required. Userland code is also easier to debug.

```
audio_fd = open ("/dev/dsp", open_mode, 0);

int mask;
ioctl(audio_fd, SNDCTL_DSP_GETFMTS, &mask);

int format = AFMT_S16_LE;
ioctl(audio_fd, SNDCTL_DSP_SETFMT, &format);
```

Figure 5.2: OSS code style

```
err = snd_pcm_open(&handle, device, SND_PCM_STREAM_PLAYBACK, 0));

/* get current swparams */
err = snd_pcm_sw_params_current(handle, swparams);

/* start transfer when the buffer is full */
err = snd_pcm_sw_params_set_start_threshold(handle, swparams, buffer_size);

/* align all transfers to 1 samples */
err = snd_pcm_sw_params_set_xfer_align(handle, swparams, 1);

/* write the parameters to device */
err = snd_pcm_sw_params(handle, swparams);
```

Figure 5.3: ALSA code style

5.3.4 ALSA Utilities Package

Most of currently available programs are written for OSS sound API. There is OSS emulation in ALSA, but native applications are welcomed. They could serve as examples and first aid utilities when OSS support does not work. ALSA utilities package consist of five basic utilities for manipulation with sound systems and sounds at all. Summary of included utilities is in following table.

alsactl	an utility for sound card setting management
aplay/arecord	utilities for the playback and capture of .wav files
amixer	a command line mixer
alsamixer	another mixer with ncurses interface

Table 5.3: ALSA utilities

The most important one of all is command line mixer. We need them, after we have successful installed ALSA drivers and library. First step we need to do is unmute the card. This step is described in installation documentation in details [10]. It's recommended to try to use aplay/arecord pair to test full functionality of native ALSA interfaces and correctness of installation. Specialized utilities for some sound cards or chips could be found in ALSA tools package (includes setup tool for SoundBlaster 16, Envy24 (ise1712) cards, OPL2/3 FM instrument loader, assembler for emu10k1 DSP chip and possibly others).

5.3.5 ALSA OSS Package

ALSA and OSS introduce similar features. Both are sound systems with same basic abilities. Just small effort is necessary to implement the API of the other system. To be able to win in the battle field of open source software, ALSA needs to offer backward compatibility. Main functionality of OSS compatibility layer is covered in kernel modules. But there exist user land library too named *libaoss*, which acts as middle layer between kernel modules and ALSA library. To run programs with OSS sound system interface you may need both kernel modules and this library in some special cases (if you would like to apply some routing settings from ALSA configuration files). When you need to place this library on some nonstandard place, there is a script in the package which should act as simple wrapper to facilitate the use of the OSS compatibility library.

Chapter 6

Development Environment

After obtaining the iPAQ device, the first step we need to undertake is to setup some usable development environment. This should be divided into two main parts. First we need to get the iPAQ up and running and next we must have cross-compiling tools on the workstation that we use for the whole development.

6.1 Problems with iPAQ and Familiar 0.4

The Compaq originally supply it's PDA with preinstalled Microsoft Windows CE. When I got the device it was equipped with Linux (Familiar distribution). So I do not need to undergo the unhealthy process of replacing the product of Microsoft with Linux. But there was still need to upgrade the Linux system of the iPAQ. The old distribution used there and mainly the boot loader have some nuts I need to dispose of. The main problem was partition layout.

Most of PDAs have two kinds of memory. First traditional SDRAM memory, as we know it from workstations, but they have no fixed disk. Instead of the winchester, non-volatile memory is used. To advantages of this solutions we should count at least small dimensions and no mechanical components. This memory named *flash*, is then used as stable storage medium. Even when sized to around tens of megabytes, to keep similarity of I/O layers of operating system, the same structure is used. The *flash* device is divided into few independent parts called *partitions*. On the old Linux system, there were four partitions. First one was dedicated for boot loader, small program for booting the operating system after power up. The second partition was used for storing parameters of the boot loader. The third one was used for Linux kernel and the last one was for regular filesystem.

For the simplicity the layout of the partitions, their sizes and placing was fixed. With 16MB of flash, there was $2 * 256KB$ for boot loader and it's parameters, then come $512KB$ for the kernel and the rest ($15MB$) was free for the filesystem.

There are two problems with this. One is waste of expensive flash when using smaller boot loader (it should have around $160KB$) and just a few kilobytes of the parameter partition. On the other hand, having the kernel smaller than $512KB$ is not easy to achieve. Even when all possible parts of the kernel ale created as modules and stored on the filesystem, the core of the kernel should have around $600KB$ or more todays (with kernels 2.4.x or 2.5.x). We do not need big amount of code like parts for IDE or SCSI devices and others, but as penalty we must use special drivers for flash or some special input devices (e.g. touch screen).

Another minor problem we should take on mind is the fact, that the image of core kernel is not accessible as normal file via the filesystem. All partitions are visible (for both reading and writing) as special (block) devices, but this behavior differs from that one we know from workstations.

There are two solutions of the most painful problem. Either we could make the partitions variable-sized or simply change their sizes to meet new requirements, or we could move the kernel image on the regular filesystem and reduce actual number of partitions.

After studying the state of according parts of kernel sources (that time branch *2.4.x* in version *2.4.16-rmk1*), it was clear, that code of partition support was devised without any mean of changing the size of partitions. Address of the beginning same as the size of partitions was explicitly written in the source code, so any modification will be very difficult and time consuming. It will be not expandable to other modification of layout or other versions of hardware.

At the same time, the next release of Linux distribution used on iPAQ was released. One of new component is the boot loader, which was dramatically changed and now offer much new possibilities. One of which is booting the kernel image placed on the filesystem. Even it's not so simple as one may think, this second variant was selected for solving the problem with size of kernel core.

6.2 Upgrade to Familiar 0.5.1

This section should be divided into few independent parts. First we should say something about manipulation with partitions and possibilities of the bootloader at all. Next we should save old state of iPAQ. Then, the main and most dramatic step comes on the scene. After this is done, we need to reconfigure the partition layout and finally the new distribution could be placed on newly created partition. As the last thing to be at the same functionality as before this whole process, we will need to modify default configuration of Familiar distribution.

6.2.1 The CRL/OH ARM Bootloader Introduction

The bootloader is first software, that takes its role after iPAQ is powered on. There is possibility of selection between few bootloaders, but for iPAQ one of the most used is the ARM Bootloader from Cambridge Research Laboratory and Open Handhelds. Full but unfortunately not very up-to-date documentation can be found at [4]. As there is source code available it is the best way in the case of troubles. Most of PDAs have no keyboard at all. iPAQ is the exception, Compaq offers external keyboard, but it's not shipped in default configuration. Because of this, there must be any other possibility to control the bootloader. In this phase of system startup, the touchscreen or software keyboard it just a dream. Offered solution is console accessible via available ports. The iPAQ has both IrDA port and either serial or USB port via the cradle. At this time, the bootloader has support for IrDA and serial console. As you can presume, the second one is the winner of very hard selection. After connecting the serial cable into standard serial port of workstation, you need just some terminal emulator, which then communicate with the bootloader. There are plenty of such programs, name *kermit* or *minicom* just two of most known in the world of Unix. RedHat distribution (and all other major distributions) used for workstation contains the *minicom* package. Second package, we will need is *lrzsz* for implementation of *xmodem protocol* for file upload and download.

Default determination of minicom terminal emulator is to work with modems, testing of their configuration and remote logging on servers. For use with the bootloader we will need to change default configuration a bit. For security reasons, the special file for serial ports are accessible just for the owner, that is the root. Naturally we would like to use minicom under normal user, so we should either make minicom SUID root, or modify access rights for serial port.

As the least painful solution I choose to set the group of serial port's special file to *uucp* and add *read* and *write* rights for this group. Users, that should work with iPAQ this way were added into group *uucp*. Now minicom is usable for (selected) standard users. Next we need to create configuration for communication with bootloader. Requested settings are listed below:

serial device:	/dev/ttyS0
lockfile location:	/var/lock
speed:	115200 bps
number of data bits:	8
parity:	none
stopbits:	1
HW/ SW flow control:	none

Table 6.1: Minicom settings

The serial port device should differ (depends at least on used serial port). The location of locking files needs to be the same as for *pppd* - a PPP daemon discussed later. Now we have minicom prepared for accessing the bootloader.

When iPAQ is powered on, the splash screen of the bootloader is displayed. By pressing the *calendar* button, the serial console is activated. Now running minicom we should see the prompt of the console. Type command *help*, and all possible commands are displayed with some short hint. Now type command *show* to display content of variables, which may be used to adjust bootloader's behavior. Next tables summarize the most important commands and variables for our next work.

help	lists all supported commands with short description
load	load the image of given partition
boot	boot using default or specified type (nfsroot | flash | jffs2 [kernel_image])
show	show all or specified parameter's value
set	set specified parameter to given value
params	(save | reset) saves user defined parameters or reset them to default values
partition	(reset | show | define | save) used to manipulate with flash partitions

Table 6.2: Bootloader commands

In the table of variables, the values in brackets if any are the default ones.

baudrate	[115200] speed of serial communication.
boot_type	[flash] where to boot from (nfs | flash)
linuxargs	[noinitrd root=/dev/mtdblock1 init=/linuxrc console=ttySA0] parameters passed to the kernel

Table 6.3: Basic bootloader variables (version 2.14.5)

The most important variable for us will be the variable *linuxargs* to be able to set the Linux kernel appropriately. Usage of main commands and variables will be demonstrated on the process of upgrading the bootloader same as the rest of whole system.

6.2.2 Backup of Old System

Because this will be dancing on very thin ice, first think we need is to make some backdoor. When everything will be wrong we should be able to get on the start line and begin again.

Creation of backup on the workstation is quite easy. We just need to store the images of defined partitions and the original partition layout. Then whenever we want, we will be able to construct the content of the flash again. The layout of partitions is fixed as explained above. Original system has four partitions with these parameters:

name	**start**	**size**	**flags**
bootldr	0x000000	0x040000	0x02
params	0x040000	0x040000	0x02
kernel	0x080000	0x080000	0x00
root	0x100000	0xf00000	0x16

Table 6.4: Original partitions layout (Familiar 0.4)

Now how to store images on the workstation. We will use bootloader's command *save*. It has one argument - name of the partition to send using xmodem protocol. After typing the command, we need to instruct minicom to receive the image. Using keys *CTRL-A-R*, we say minicom we would like to receive file. Then we must select protocol, for us *xmodem* is the right choice. Finally enter name of file, where to store received data and that's all. You should see dialog window with informations about progress of the transfer. Using serial cable you may get about 20KB/s, so be patient.

```
boot> save params
About to xmodem send root
  flashword=50000000
  base=00040000
  nbytes=00040000
totalPackets=00000866

Upload Successful

Bytes Transferred=00040000
ackcnt=00000866
nakcnt=00000001
```

Figure 6.1: Partition saving

Using this approach, save partitions *params, kernel* and *root*. You should have files named *params.img, kernel.img, root.img* of the same size as their partitions now (real names of your files are not important of course).

6.2.3 Bootloader Upgrade

Having way back open, we can continue with bootloader upgrade without worry. Until now, it is possible to use documentation from the Familiar Distribution for installation of whole system

[5]. Installation of new bootloader is not difficult in number of steps or their complexity. The only one bugaboo is the fact, that when you admit the failure, you can end with totally unusable piece of iron and silicon named iPAQ. There is just one advice. Do not, under any circumstances, reset the iPAQ during this process. Ensure that the battery is fully charged, cross your fingers and let begin.

The approach is de facto just reverse of backup process. First we shall instruct the (yet old) bootloader to receive image of some partition, then say minicom to send the right file. For bootloader we use command *load* with the name of partition (now *bootldr*) as one parameter. Then instruct minicom to send file using *CTRL-A-S*. In file selection dialog you can use arrow to select the right file and *G* like *Go to* to change directory. As for receiving, progress information are shown during transfer.

We use bootloader version *2.17.18*, so the right file should be named *bootldr-2.17.18.bin*. You need to download it from *www.handhelds.org* with the rest of Familiar distribution [5]. After transfer is done, bootloader informs you about storing data info flash:

```
load bootldr
loading flash region bootldr
using xmodem
ready for xmodem download..
BSD sum value is: 00000000
programming flash...
unlocking boot sector of flash
Protect=00000000
erasing ...
Erasing sector 00000000
writing flash..
addr: 00000000 data: EA00008E
addr: 00010000 data: E1A0C00D
verifying ... done.
startAddress :00000000
limitAddress :00018980
Protecting sector 00000000
Protect=00010001
```

Figure 6.2: Bootloader partition loading

Your numbers can be quite different, but you need to see messages about erasing, verifying and writing the flash. If you don't see whole listing but just few last lines, you don't accept progress dialog quickly enough after transfer and some lines dismiss. But don't panic, this makes no troubles.

At this point, hit the reset button at the bottom right hand corner of the iPAQ. You should now see the new bootloader splash screen. Check the version number on last line and compare it with suggested version. To use some new features (booting from filesystem and no parameters partition) you need bootloader version *2.17.13* or later.

New version of bootloader has few additional variables to set up boot from JFFS2 filesystem properly. Meaning or default values of others should be quite different. The most important follows:

boot_type	[jffs2] where to boot from (nfs | flash | jffs2)
kernel_partition	[root] on which partition to search for kernel
kernel_filename	[boot/zImage] name of kernel image file within the filesystem

Table 6.5: Additional bootloader variables (version 2.17.18)

6.2.4 Partitions Reconfiguration

Layout of partitions has changed, so we need to configure it. Default partition layout is good enough for us, so the only one step we need to do is this command of bootloader:

```
boot> partition reset
argv[1]=reset
defining partition: bootldr
defining partition: root
```

Figure 6.3: Partition resetting

As you can see from the listing, just two partitions were created. *Bootldr* partition is the same as for previous version and the rest of flash is occupied by the other one. Whole *root* partition is used for JFFS2 filesystem. Kernel image and user modified parameters are stored there. Complex view of new layout should be obtained with *partition show*:

```
boot> partition show
argv[1]=show
npartitions=00000002
bootldr
  base: 00000000
  size: 00040000
   end: 00040000
 flags: 00000002
root
  base: 00040000
  size: 00FC0000
   end: 01000000
 flags: 00000018
```

Figure 6.4: New partitions layout

Using just two partitions we save some space and solve the problem with larger kernel images than the size of previously used kernel partition.

6.2.5 New Image Download and Modification

Using same approach as with bootloader image, we download image of new system. File is named *task-bootstrap.jffs2.* Again there should be some message about erasing, verifying and writing the flash on the console of bootloader. When done, the last step of upgrade is to reset iPAQ pushing reset button at the bottom right corner of the iPAQ. Splash screen of new bootloader will be shown. Type *boot* on your serial console or press calendar button on iPAQ to boot Familiar distribution version 0.5.1.

To use iPAQ for development, we need to modify its configuration slightly. Without any text editor, it's difficult to modify configuration files a bit, but we have *sed, echo* and *cut.* That is enough to configure whole system properly.

Because the serial line is the only one connection to the iPAQ, we will need some emergency scenario but another communication channel for comfortable work. For emergency the communication must be the easiest possible, so we will use serial port for Linux console and minicom for manipulation with iPAQ. For comfortable work we would like to establish full network connectivity to be able to use *ssh* for remote access and *NFS* for file sharing. These two modes are exclusive, so we need to be able to switch between them.

First iPAQ will boot with console on its touchscreen and with terminal directed to serial port (/dev/ttySA0). When everything is fine, we switch to fine work mode and free the serial port for PPP connection. We will start *ssh* and *NFS* daemons at the end.

For detailed description of Familiar root image modification and configuration changes see Appendix B. After all do not forget to backup whole new image using bootloader's *upload.*

6.3 Cross-compiler Tools

Using native compiler running on iPAQ is possible, but performance of iPAQ is nothing wonderful, so better alternative is cross-compiler running on workstation. With NFS we should leave ARM binaries on workstation and run it on iPAQ from shared filesystem.

We need to undertake three steps. Create cross-compiler, cross-linker and another utilities for manipulation with binary objects and last we will need basic libraries compiled for ARM architecture. Recommended compiler for ARM (mainly kernel) development is GCC version 2.95.X. Last versions of binutils (2.11.2) and GNU libc version 2.1.X make the toolchain complete. All these may be downloaded from Internet as single package that is ready to use. This is uncomplicated way, but as we would like to get deeper into ARM development, let's create our own binaries for the toolchain.

Creation (or better compilation) of toolchain is perfect example of chicken-egg problem. To compile standard C library we will need cross-compiler to work, but to create it, we will need some parts of that library. In creating linker, there is no sequence with the others. Clearly, there is a solution for this. Even two variants. We can use one of "eggs" from standard distribution, or we can follow instructions contained in GCC package or on the Internet to produce all part in one step.

For doing everything with own hands, we simply put all components into one directory and compilation process will be able to recognize the chicken-egg problem and solve it using more stages of cross-compiler creation. First temporary cross-compiler is created. It is used to compile necessary parts of libc, and then final version of cross-compiler tool is built using results of previous stages.

When I went into all these troubles, I decided to use proper C library from ARM port of Debian Linux distribution to make my task a bit easier. On the other hand, using newer version of GCC for workstation with success, I'd like to try it for iPAQ too. I choose GCC 3.0.2, because newer version has problems with cross-compiling itself. From my test emerged, that GCC 3.0.X for ARM is not completely stable and without bugs. Problems are concentrated around the part for C++ and it's library, so it's still useful for kernel development using C language. Later GCC version 3.0.4 was released, repairing most of previous bugs concerning with ARM architecture specific parts. This version looks good for both C and C++ language, but as GCC 3.0.2 looks sufficient for kernel development, I don't expect moving to this new version. Finally I give you at least used configuration for GCC:

```
Configured with: --enable-threads=posix --with-cpu=strongarm1100
                 --target=arm-linux
Thread model: posix
gcc version 3.0.2
```

Figure 6.5: GCC 3.0.4 configuration

Used ARM distribution is ARM port of Debian Linux version 2.2 Release 3. Apart from development package for GNU libc, other libraries like *libncurses, libm, libproc, libz, libcrypto* or *libwrap* are used.

6.4 Selection of Linux Kernel

In the time of this project, two main branches of the Linux kernel are on the scene. Currently stable version 2.4.x and newly created development version 2.5.x. As they differ in significant parts, it will not be possible (or reasonably achievable) to produce code for both of them.

First it looks that 2.4.x branch will be better choice, because for 2.5.x I need to study modified parts and probably modify the code to obtain working Linux kernel. In the time of deciding between these two alternatives, information about involving ALSA sound system info development series of Linux kernel was officially published. Because I would like to offer results of my work to Linux community, I will prefer progressive versions of Linux kernel and ALSA to avoid work with porting my work when done.

For workstation Linus' kernel and for iPAQ branch from Russel King are used. Currently that means these kernel versions:

- Intel x86 kernel - 2.5.7-pre1
- Arm kernel port - 2.5.6-rmk1

Both kernels will be upgraded only if there will be serious bugs repaired or necessary features added. Russel's port come out from Linus' branch and is continuously merged with it. Main information and download sources for both kernels are web pages *www.kernel.org* and *www.arm.linux.org.uk* respectively.

Until now, whole 2.5.x branch of Linux kernel go through the phase of implementing new features and modifying significant parts of code. Changes in block I/O layer and other major modification has deep impact on kernel functionality. Although all new improvements bring better performance and/or greater abilities to the kernel, there are lot of new bugs in modified or new code. Fortunately all changes was discussed well enough, to allow quick revision of new code and successful bugs removal. In version 2.5.4 of Linus' tree (thus in Russel's tree too) most of bugs are caught and removed, so it looks like current kernel versions are stable enough to be used at least for kernel development.

In kernel version *2.5.5-pre1* ALSA sound subsystem was integrated into the kernel being now recommended sound platform for future development of Linux kernel. For short time it is intended to be used by testers and developers only, until the process of it's integration into the kernel will complete. Because both ALSA and OSS are fully modularized, it is possible to have both sets of modules compiled and switch between them using one kernel without rebooting. After integration, it is expected, that ALSA will be used by much more installations. Thus there is an effort to improve web pages of the project as well as additional documentation. Even with new kernel it is possible to use old standalone packages of ALSA (stable 0.5 version same as development version 0.9). ALSA sources integrated into kernel were derived from version 0.9.

6.5 Kernel Modifications

Using pure kernel versions discussed above, I was not able to boot or even compile them. That's why I need to create few patches with modifications to be able to use them properly. For kernel of workstation (x86), there were minor problems with new block I/O layer and yet unmodified drivers for some end devices. I created a patch for ZIP driver (*ppa*) and send it to responsible person (author of changes in block layer - J. Axboe). For ARM kernel I need to modify driver for MTD device (*mtdblock* for read only access) to be able to boot the kernel. Patch was sent to Russel King and it was included in one of next releases of *-rmk* branch. All patches are included into project sources. Brief description of patches content follows.

spin_locks - In 2.4.x kernels, there was one big lock for exclusive access to data structures of block I/O layer for I/O request serving. The lock was named *io_request_lock* and was common for all block devices over the system. This solution was not easily extendible and scalable. In 2.5.x branch, there are two major modification to solve these problems. Common lock was removed and replaced with one lock per request queue, which is exclusive for each block device. With this change, other minor trouble is repaired. Before, the lock for request queue was locked too early and in many cases is was held unneeded wasting system resources. This makes multiprogramming less effective. In some cases the lock was unlocked in lower level routines and has no effect at all, so locking it was completely unneeded at all. For *ppa* driver (older parallel port IOmega ZIP drive) some calls of *spin_[un]lock_irq(io_request_lock)* call was removed and in others, lock from the request queue is used instead of global one.

Second major difference is in collecting buffers for scatter/gather mode of data transfers. The main thought is to offer buffers per disk sector (512B), but gain one page that is data unit for page cache. In computing layout of buffers, field named *address* was used in 2.4.x kernels. For full support of high memory, this was changed in 2.5.x to couple *page* and its *offset*. *Address* field is supposed to be removed in the early future. While stepping through kernel call graph, I do not find any asserts for badly prepared buffers. This caused NULL pointer dereference and because of this I added simple test of buffer address before reading data into it.

kdev_t - For unambiguous identification of devices, pair of minor and major numbers is used in Linux kernel. In 2.4.x tree, these two numbers were coded into one integer using its lower and higher half independently. As number of different devices grows, there is a need to make scope of both numbers wider. In the future, this may be solved using few different techniques, but for now, at least some abstraction is needed. This will offer to change implementation independently on the rest of code. New type *kdev_t* was introduced to accomplish this goal, which is now defined as unsigned short, but in the future structure with items for minor and major numbers is expected. To be able to manipulate with device numbers, set of macros or functions is introduced to convert the pair to integer, compare two devices and so. Another problem with this is existence of macros for access minor and major numbers from device number in standard include files (*sys/sysmacros.h*). These definition of macros *major()* and *minor()* are not the right one and should not be used in the kernel. Correct versions are named *MAJOR()* and *MINOR()* and are placed in *linux/kdev_t.h.* Now the definition is the same as in system includes, and it's intended to be copy for source code, that has access just to kernel includes, but it should be better to use the definition placed here. Some problems with *kdev_t* was needed to solve with ALSA sources too, but that is discussed in another chapter.

Unfortunately, patches for *kdev_t* modification I prepared were not used. Critical parts of code were replaced before I was able to send my patches to dedicated persons. For now, these patches repairing *kdev_t* problems in *Reiser* file system and other parts are useless and are not involved in project sources anymore.

The only patch, that was applied is *kdev_t* and spinlock modification for MTD block read-only access. The patch was supplied to maintainer of MTD CVS and applied there. It should be merged into Linux kernel some day in the future.

MTD partitions - In kernels from *2.5.x-rmk* branch, the support for dynamic partition layout detection via partition table parsing is broken. In Familiar distributions, kernels are patched to solve this. These patches add the parser's code into the kernel. Because the bootloader itself needs to parse the partition table before any other action could be taken, the kernel should not contain this code anymore. This is the reason, why the patch from Familiar distribution was not accepted and applied. Recent versions of boot loaders contain support for passing information about partitions layout to kernel (via kernel parameter), but unfortunately the code which will handle this options and sets the partition information correctly is not completely finished in current *-rmk* kernel. As the reason, new kernels are not able to boot on iPAQ. To solve this problem I created a patch which add and option to the kernel configuration to allow to switch between two most used statically defined partitions layouts. Appropriate changes were made in other parts of kernel sources. The patch was accepted by Russel King and is part of ARM Linux kernel since *2.5.4-rmk1.*

broken modularity - In latest Russel's kernel, there was some work on cleaning architecture dependent code. As a sider-effect, some symbols were no more exported from the kernel and so *SA1100 RTC* and *APM* features can not be build as modules anymore. Patch solving this was accepted and applied in *2.5.4-rmk1* and *2.5.5-rmk2.* For RTC, requested symbol was just exported from it's new location, but for APM the changes are deeper, because of file name collision. Original file *pm.c* was renamed to *pm-sa1100.c* and appropriate Makefile was updated to compute module versions from this new file. Patch with these changes was pending when updates to build system of the kernel allow me to undo renaming of *pm.c* and update just missing symbol.

others - for ARM kernel there was an error in Makefile for *fastfpe* - smaller of two floating point emulators used in ARM port. As there are no floating point instruction, code need to be compiled with *-msoft-float* option for GCC. For C source files, this was fine using default rules in *Rules.make*. Main parts of floating point emulator are written in assembler, and default rule for assembler to object file translation was not the right one. First, special rule was added to Makefile in *fastfpe* directory, but after discussion with Russel King, this was changed to definition of *USE_STANDARD_AS_RULE* symbol, which should be used to change the behavior.

In macros definition for *assabet* board, which are used for iPAQ too, there was some mistype in *ASSABET_BCR_frob()* macro. Newly it used two parameters, nevertheless it's only one empty command for actual configuration and iPAQ hardware. Definition in used ARM kernel has just one parameter, so the compiler was not satisfied. I just add the second parameter to the definition of this macro. Later is was removed from the patch, when Russel King notified me, that he has repaired this already.

Chapter 7

Preparing ALSA for iPAQ

Currently ALSA can be found in development branch of Linux kernel. But for our sound driver development we will use ALSA sources directly from CVS tree. Four packages from ALSA will be in the center of our interest. We would like to integrate new driver into *alsa-driver* package first to enable other developers to comment our work. After integration into Linux kernel, most files in *alsa-driver* are just links to appropriate ones in *alsa-kernel*. The strategy of creating new driver is: place it in *alsa-driver* first and as late as it relatively work move all source code to *alsa-kernel*, leave just link in *alsa-driver*. Except the driver itself, we will need ALSA library (*alsa-lib*) and some basic utils (like amixer, aplay and arecord) to work with ALSA from *alsa-utils*.

7.1 Modifications of ALSA-utils

All ALSA packages were developed in native environment, so there is no support for cross-compiling now. First we need to add this to be able to get binaries for iPAQ.

For *alsa-utils*, I will do any modifications, because all utilities we will need consist of just one source file. It's simpler to show requested command line to cross-compile needed file:

```
arm-linux-gcc -DHAVE_CONFIG_H -I. -I../include -O2 -Wall -pipe -g \
-o alsactl  alsactl.c  -lasound -lm -ldl

arm-linux-gcc -DHAVE_CONFIG_H -I. -I../include -O2 -Wall -pipe -g \
-o alsamixer alsamixer.c -lncurses -lasound -lm -ldl

arm-linux-gcc -DHAVE_CONFIG_H -I. -I../include -O2 -Wall -pipe -g \
-o amixer amixer.c -lasound -lm -ldl

arm-linux-gcc -DHAVE_CONFIG_H -I. -I../include -O2 -Wall -pipe -g \
-o aplay aplay.c -lasound -lm -ldl

rm -f arecord && ln -s aplay arecord
```

Figure 7.1: ALSA utilities cross-compilation

We will need these libraries installed to compile and use these utilities properly: *libdl, libm,* *libncurses* and ALSA library *libasound*. Being pure user-land code, we do not need to modify it for ARM architecture at all.

7.2 Modifications of ALSA-lib

More seriously we will worry about *alsa-lib* package. It is much more difficult, so we will need to solve this using support of *autoconf* for cross-compiling.

When you would like to cross-compile the *alsa-lib* package, you will need to configure it with additional parameters. You should run *configure* script as follows:

```
CC=arm-linux-gcc ./configure --target=arm-linux \
--with-soundbase=/usr/local/arm/src/linux/include
```

Figure 7.2: Cross-compiling parameters for ALSA-lib

In this example host where the library is built is guessed (should be given with *–host=platform* and target for which is the library build is Linux on ARM architecture. You need to specify the location of kernel sources configured for target architecture. This is used to find ALSA include files and so it should be set to prefix where */sound* directory is to be found. You should omit setting *CC* variable and cross-compiler will be guessed too. You should also use option *prefix* to specify ALSA include directory if $prefix/include/sound exists.

So the simplest version would be one of these two lines:

```
./configure --target=arm-linux --with-soundbase=/usr/local/arm/include
./configure --target=arm-linux --prefix=/usr/local/arm
```

Figure 7.3: Another possible parameters for ALSA-lib

As you will likely specify the prefix, the last possibility would be the best if everything match. On the listing of *configure* script, it's good to check correctness of detected values. The most important for cross-compiling is cross-compiler used, appropriate processor type and used kernel sources. For platform names in the form *cpu-vendor-os* (or aliases for these) you should look in *config.guess* script.

To get this behavior, We need to modify basic configuration script a bit. First, for comfortable work, we would like to detect cross-compiler automatically. To enable basic cross-compiling support for *autoconf*, which is used for whole ALSA system, we add macro *AC_CANONICAL_HOST* at the beginning of *configure.in* script. Then comes part for cross-compiler guessing. We will look for program named *$target-gcc*, which is one of the most used names for cross-compiler binary. Then we will try some other variants created with parts of canonical system name of the target. If unsuccessful, user must set compiler manually using *$CC* variable. Last, we must replace *$host_cpu* variable in architecture dependent settings with *$target* to allow cross-compiling build. I created patch with all these modifications that was later integrated into ALSA CVS.

7.3 Modifications of ALSA-driver and ALSA-kernel

Because whole driver will be placed in *alsa-driver* package, no modifications of *alsa-kernel* are needed now. We should use this package just to provide help and templates for us. I prepared necessary patches to integrate new driver into kernel, like *Config.in* and *Config.help* files, but they were not used in first stage of driver creation. After driver integration into the kernel, the patch was applied to ALSA CVS and later to Linux kernel.

For *alsa-driver*, we need to modify present configuration scripts and Makefiles to include new directories and then create these directories with its content. First we add the definition of *CONFIG_SND_ARM* variable which should control inclusion of directories with ARM specific sources. For *configure.in* script the way of *alsa-lib* is not followed, because we need to parse kernel configuration. This is best done using C preprocessor and compiler. But with cross-compiler it is not possible to run created binaries on host system. Thus we will leave configuration to be performed like if we are building native drivers. The only difference is new option *–with-cross*, that should be set to cross-compiler name prefix (e.g. arm-linux-). This value is then used with basic compiler name to get name for cross-compiler. The name of host compiler is modified to not contain path to obtain reasonable name of cross-compiler. The same solution is used for cross-preprocessor (mostly *gcc -E* is used for this), cross-linker (arm-linux-ld) and assembler (arm-linux-as).

Although we do not use support of *autoconf*, this solution is so simple, that it should work with most system configurations. Presented solution expects, that native and cross tools are built from the same source (i.e. both are one version of GCC) the same and they are placed in common locations. For *alsa-kernel*, this process is obsoleted by general kernel configuration and after inclusion into kernel, this problem will not pain anymore.

Last of general files which needs to be modified is *Modules.dep*. This covers kernel module dependency in ALSA subsystem. The same file is included in *alsa-kernel* package and both files are then used in *alsa-driver* (that one from *alsa-kernel* is linked into *alsa-driver*). For the beginning the only useful dependency is *snd-pcm*. That is one of basic modules, enabling whole PCM support same as all common routines through its dependency on other parts of sound core. We add both modules for our driver into *Modules.dep*. It is used for dependency building.

Actually, all created patches are included in ALSA CVS tree and accessible to all other developers and users of ALSA sound system. While preparing ALSA system for iPAQ, other minor bugs were found and repaired.

Chapter 8

Driver Implementation

All sound card driver source files will be placed somewhere in *alsa-driver* package until the driver will be stable enough to include it to Linux kernel. Then sources will move to *alsa-kernel* package.

For driver development we use information from data sheet for UDA1341TS from Philips [11]. Other sources of information are OSS driver for UDA1341 and ALSA source files. Questions and specific problems could be discussed at ALSA developers' mailing list *alsa-devel@list.sourceforge.org*. Mainly in first phase, sources for Intel 8x0 integrated audio cards and ESS1938 card are used as this hardware is used in my workstation.

Driver for UDA1341TS is divided into two modules. Module *snd-uda1341.o* contains mixer part of the driver, because this is common to all sound cards, that uses this sound codec. This module should finally contain all parts of code, which are specific directly to UDA1341 codec and are not dependent on underlying hardware. Second module (*snd-sa11xx-uda1341*) is used for ARM and mainly iPAQ H3600 specific parts. Control of DMA transfers and card initialization is placed here. Source files for *snd-uda1341.o* module are placed in *i2c/l3/* directory and source files for *snd-sa11xx-uda1341* module are in *arm/* directory. Include files could be found in *include/* directory. The driver and some of its files were renamed during development. Older name was *h3600-uda1341*. These names could be found in some initial versions in ALSA CVS. Later, the name was changed following common schema *DMA_bridge-codec_chip*. Some other minor changes was made through whole source tree, these will be summarized in next part.

8.1 Configuration and Build System Update

First, we need to update configuration and Makefile files to add new card and as new architecture to ALSA. Some basic updates were mentioned in previous chapter, these were used mainly to allow common cross-compiling. Now we add some other changes to include newly created driver.

In configure script, the only one thing we add is detection for CPU type and model. When *Intel StrongARM SA1100* is found variable *CONFIG_SND_ARM* is set to *'y'*. This variable is then used in *Makefile* to enable whole directory *arm/* and in *i2c/Makefile* to enable *i2c/l3/* directory.

To add our new card to list of supported cards (i.e. list of cards for configure script) we need to modify one more file. We add these lines to *utils/Modules.dep*.

This file has the same meaning as equally named bigger brother in *alsa-kernel*. This one is used for cards (modules), that are not yet placed in Linux kernel, or are somewhat specific to *alsa-driver*. First item is used just to get ALSA to know about *snd-uda1341* module. The other

```
%dir linux/sound/i2c/l3
snd-uda1341 snd-pcm

%dir linux/sound/arm
|snd-sa11xx-uda1341 snd-uda1341 snd-pcm
```

Figure 8.1: Modules.dep modification

describes dependency for main module (see the '|' character on the beginning of line). Main module uses the mixer part and common routines from ALSA core (its PCM part).

Last but not least, Makefiles for our card. Both files (*arm/Makefile, i2c/l3/Makefile*) have similar structure, they only differ in names of objects used.

```
TOPDIR = ..
include $(TOPDIR)/toplevel.config
include $(TOPDIR)/Makefile.conf
TOPDIR = $(MAINSRCDIR)

O_TARGET    := arm.o

list-multi  := snd-sa11xx-uda1341.o

snd-sa11xx-uda1341-objs := sa11xx-uda1341.o

# Toplevel Module Dependency
obj-$(CONFIG_SND_H3600_UDA1341) += snd-sa11xx-uda1341.o

include $(TOPDIR)/Rules.make

snd-sa11xx-uda1341.o: $(snd-sa11xx-uda1341-objs)
$(LD) $(LD_RFLAG) -r -o $@ $(snd-sa11xx-uda1341-objs)
```

Figure 8.2: Makefile for sa11xx-uda1341 module

Inclusion of common files is identical in almost all Makefiles in ALSA. These contain configuration and common targets. *O_TARGET* object is used when compiling into kernel image and not to modules (see Documentation/kbuild/makefiles.txt in kernel source tree for details). *snd-sa11xx-uda1341-objs* defines list of objects that will result in kernel module (though we have only one object file). System will create the module in dependency on definition and value of CONFIG_SND_H3600_UDA1341. It should have value *'n'* to disable our feature, *'m'* to build it as module or *'y'* to include it into kernel image. After including common rules, last thing we need to do is definition of process of creation of our module. Default rules are enough for us, so no peculiarities are found here.

Makefile for *snd-uda1341.o* module is modified, just changing all *sa11xx-uda1341* to *uda1341*.

Once created, there is small probability we will need to modify these file anymore. When no other files will be added or present one renamed. For now, it is assumed, that whole driver will be compiled into modules (at least until it will be stable and usable to other people then developers), so configuration and compilation into kernel image is not tested (but should work as long as all standard makefile variables are regularly defined).

8.2 Basic Mixer Support

Whole mixer is placed in *i2c/l3/uda1341.c* with include file *include/uda1341.h*. In this part, we will look on basic mixer, which will enable just *Master volume* control to be able to test PCM part. Other controls will be discussed in related part.

We need three entry points for the mixer. We must be able to init the mixer and shutdown it on sound card startup and shutdown respectively. Then we must have some interface to control the behavior of mixer part.

Being *L3* device, few common operations need to be implemented. These are send to upper common *L3* layer via *l3_add_driver()* call. This function takes pointer to *struct l3_driver* as its parameter. This structure describes *L3* interface of our driver. Following operations are defined:

uda1341_attach	attach L3 client to system - initialization of client
uda1341_detach	detach L3 client from system - de-initialization of client
uda1341_open	L3 device opened - initialization of device
uda1341_close	L3 device closed - shutdown device
uda1341_command	control L3 device

Table 8.1: Common L3 operations

The Open-Close pair is used to set/clear activity flag, so the rest of driver knows whether the device is on or off. In Open function we should set default values to all UDA1341TS registers. Similarly Attach/Detach pair allocate/free and set proper (default) values to register images in driver. Command function in our case just calls another functions to update requested registers both in driver and on chip.

Sound driver oriented interface consists of two exported functions - *snd_uda1341_mixer_new()* and *snd_uda1341_mixer_del()*. These are used in main driver module to initialize and shutdown the mixer part. Control of mixer in main module is done via L3 interface - the *uda1341_command()* function. *mixer_new()* function allocates memory for driver structure, attach the *L3* client and enable controls. It also enable the */proc* interface and passes pointer to mixer part driver structure to main driver module. Proc interface contains files */proc/asound/cardX/uda1341* and */proc/asound/cardX/uda1341-regs*, that show status of all registers in human readable form (or as binary numbers respectively). *mixer_del()* frees memory, stops */proc* interface and detach the *L3* client.

Each control is defined by its name, register used and position of information in that register. There are three methods to work with control. We have *get()* and *put()* to read and write values from/to control and *info()* to initialize control. *get()* function uses driver image of registers to obtain value of requested register. *put()* must set driver image same as on-chip register to correct value. There is macro for easy definition of new controls. They are all placed into array, that is processed in *mixer_new()* function.

Rest of the driver is made by routines to write values to registers on chip. High layer - function *snd_uda1341_cfg_write()* offers translation between service numbers (which is used by main module) and register numbers. Middle layer - *snd_uda1341_update_bits* (used by *put()* in controls) modify requested bits in copy of selected register in driver and if new value differs, low level function is called. Writing value to register in direct or extended addressing mode is work of *snd_uda1341_codec_write()*.

In this first stage, we define just one control - *Master Playback Volume* to control DAC gain for playback. This allow us to implement PCM part of the driver and then we come back and finish mixer part. Then we will be able to check behavior of additional mixer control.

8.3 PCM Device

PCM device implementation can be found in *arm/sa11xx-uda1341.c*. Origin of main module is in *snd_sa11xx_uda1341_init()* and *snd_sa11xx_uda1341_exit()* functions. The latter free memory and mixer and it is pretty simple. But the former is the heart of whole module. Here we first test, if the system is some version of iPAQ (i.e. H3100, H3600, H3700, H3800). If we have the right hardware for our driver, we can create sound card device and allocate memory for structure with internal driver informations. Then we activate mixer part calling *snd_uda1341_mixer_new()*. This sets registers on default values and prepares rest of mixer part. Then we activate PCM device, power management support and after setting card names (long, short and driver name) we register the driver in *ALSA* sound system.

Function *snd_sa11xx_uda1341_pcm()* must set up both playback and capture stream. We use *snd_pcm_ops_t* structure to define set of possible operations for both streams. We call another functions to initialize audio (*snd_sa11xx_uda1341_audio_init()*) and DMA (*audio_dma_request()*). We use two DMA channels - one for playback stream and second for capture stream.

snd_sa11xx_uda1341_audio_init() opens L3 device and powers the UDA1341 chip on. We call reset function and then set clock divisors for default sample rate (44100Hz). We use *QMUTE* feature of the chip to mute it while performing initialization. Following table summarizes possible sample rates with used clock divisor and frequency.

divisor/clock	12.288MHz	11.2896 MHz	4.096 MHz	5.6245 MHz
512 f_s	24000Hz	22050Hz	8000Hz	10985Hz
384 f_s	32000Hz	29400Hz	10666Hz	14647Hz
256 f_s	48000Hz	44100Hz	16000Hz	21970Hz

Table 8.2: Supported sample rates

Center of main module consist of a set of ALSA callback functions. These are called for card specific work in proper operations. Inside this operations rest of module code - the DMA framework is used. Now, let see which operations are used and what functions are they bound to. Summary can be found in table 8.3. Being important, we describe each function in details, mainly as help for other ALSA developers. These set of operations is used twice, once for each sound stream. Where not mentioned otherwise, both functions from the pairs (both for playback and capture) are very similar. We discuss *capture* operations, because they are a bit more difficult (for our driver - not generally).

- *snd_sa11xx_uda1341_capture_open(snd_pcm_substream_t * substream)*

  This function is called, when some userland application opens PCM device. In *ALSA* this is not done by user application. Instead *ALSA* library does this job and offers C language API. We have three goals in this function. First we must initialize driver related structure for newly created stream. This means cleaning of period counters and setting internal stream pointer. Second we need to reset the chip and DMA by calling *audio_reset()* function. Last we need to check HW and SW parameters of newly created stream for ALSA. We do some tests and send descriptive structure to ALSA defining constraints of our hardware and driver. These constraints include possible formats (for UDA1341 on iPAQ only unsigned 16-bit little endian is usable now), available sampling rates (see table 8.3), number of channels (for us only stereo is interesting - ALSA should then offer transformations to/from mono) and period description. For periods, we must specify minimal and maximal number of periods same as minimal and maximal number of bytes in one period. When processing sound, we got one big buffer, that is virtually divided into several smaller called periods. One period is processed in one DMA transfer. Size of one period is limited from bottom to value about 64 bytes to have some time to do all other work and from top by hardware. On StrongARM SA1100, maximal size of one DMA transfer is limited by *MAX_DMA_SIZE* symbolic constant (<*asm/arch-sa1100/dma.h*>) with actual value 8191B (0x1fff). Minimal number of periods is two, because we must have one period for actual transfer and another to fill up in the time of transfer of the first one. Using all these rules we got these equations:

$$buffer\_bytes\_max = period\_bytes\_min * periods\_max$$

$$buffer\_bytes\_max = period\_bytes\_max * periods\_min$$

  UDA1341TS offers some non-standard sampling rates (table 8.3). We must say to ALSA, which rates are available. This is done by defining array of possible rates and setting *SNDRV_PCM_RATE_KNOT* flag in *rates* field. This ensures, that ALSA will go through our table and use all accessible rates.

open:	snd_sa11xx_uda1341_capture_open
close:	snd_sa11xx_uda1341_capture_close
ioctl:	snd_sa11xx_uda1341_capture_ioctl
hw_params:	snd_sa11xx_uda1341_hw_params
hw_free:	snd_sa11xx_uda1341_hw_free
prepare:	snd_sa11xx_uda1341_capture_prepare
trigger:	snd_sa11xx_uda1341_capture_trigger
pointer:	snd_sa11xx_uda1341_capture_pointer

Table 8.3: Implemented ALSA PCM operations

- *snd_sa11xx_uda1341_capture_close(snd_pcm_substream_t * substream)*

  Here, we must do just cleaning. This means clearing *stream* field for capture/playback sound stream. We should clear some other fields in driver structures too, but it's pointless as the *stream* filed acts for us busy state indicator.

- *snd_sa11xx_uda1341_capture_ioctl(snd_pcm_substream_t * substream,*
  *unsigned int cmd, void *arg)*

  We have no special *IOCTLs* to handle by the driver itself, so we just pass control down to generic ALSA function (*snd_pcm_lib_ioctl()*).

- *snd_sa11xx_uda1341_hw_params(snd_pcm_substream_t * substream,*
  *snd_pcm_hw_params_t * hw_params)*

  This function is responsible for allocating memory for ALSA hardware and software parameters of playback or capture stream. We do not add any special items to these structures, so standard ALSA core function is everything we need to call.

- *snd_sa11xx_uda1341_hw_free(snd_pcm_substream_t * substream)*

  Same as in previous case, freeing allocated memory is done using standard core function.

- *snd_sa11xx_uda1341_capture_prepare(snd_pcm_substream_t * substream)*

  Prepare function is called immediately before start of data transfer (playback or capture). Here we know some other information, which are now stored in *substream* parameter and *runtime* element of *substream* structure. In *runtime* structure, we should find information about requested number of channels and sampling rate. We set the chip according to these requests. This is the only one step, we need to undertake here, because number of channels and data format is fixed for UDA1341 (at least for now).

- *snd_card_sa11xx_uda1341_capture_trigger(snd_pcm_substream_t * substream, int cmd)*

  Being one of the most important, its task is to start DMA transfer. *cmd* parameter says, which case should we handle - start or stop of DMA transfer. Now *substream* parameter is fully initialized and we should use information about DMA buffer and periods to set up DMA controller properly. We have informations about period and buffer sizes, number of periods and base of DMA buffer. We need to look out to units of these properties. Main problem should be with sizes, which are not handled in bytes (for both buffer and period) but are in frames instead. Size of one frame can differ in dependence of data format and number of channel. This makes our live easier, having just one combination. Our frame size is always 4 bytes (stereo with 16 bits per channel). This function must do its work quickly, so we just fire needed DMA transfer (or transfers as mentioned later) or stop active transfer(s) and return back to caller.

- *snd_card_sa11xx_uda1341_capture_pointer(snd_pcm_substream_t * substream)*

  Pointer function is used to tell ALSA where we are in DMA transfer. It expect number of frames from the start of DMA buffer, that are sent/received yet. We use one of DMA services for this to keep this function really simple. The *audio_get_dma_pos()* function is able to get address of actually transferred byte and with the knowledge of DMA buffer base we can simply compute requested offset in frames.

8.4 DMA Transfer Support

Another set of operations is used to control DMA transfers in this driver. Wrapper functions and SA1100 operations used there can be found in table 8.4. DMA controller for SA1100 is quite simple, so we are able to start and stop actual transfer only. No pausing and other features, that are usually found on greater computers, are not available here.

For DMA transfer, we use one channel for each direction. On iPAQ there is *DMA_Ser4SSPWr* channel for writing to sound chip and *DMA_Ser4SSPRd* for reading from it. These channels are configured in *snd_sa11xx_uda1341_audio_init()* and *snd_sa11xx_uda1341_pcm()* functions.

audio_dma_request()	→	sa1100_request_dma()
audio_dma_free()	→	sa1100_free_dma()
audio_get_dma_pos()	→	sa1100_get_dma_pos()
audio_stop_dma()	→	sa1100_stop_dma()
audio_process_dma()	→	sa1100_start_dma()
audio_dma_callback()	→	-

Table 8.4: Implemented DMA operations

When *audio_process_dma()* function is called to start DMA, we try to transfer as much periods as possible. SA1100 DMA controller is able to receive two periods at once. After both buffers are occupied, we will leave this function and try later again. We must manage counter of sent periods in buffer. This counter is incremented after each processed period with modulo of periods in DMA buffer. When we reach half of the buffer, ALSA fill the first half with next portion of samples (or reads the samples on capture).

When DMA transfer of one period is done, registered callback is called. This callback is invoked in interrupt context. We need to acknowledge proceeding of DMA transfer to ALSA. This is done by calling *snd_pcm_period_elapsed()* function. At the end, we calls *audio_process_dma()* one more time to process pending periods. We do not care of the end. ALSA takes care and call our trigger when there is time to stop. This should mean, that one more period will be transferred, but never mind.

One special trick is used for capture stream transfer. UDA1341TS chip does not have its own clock. It gets its synchronization pulses from DMA stream. With playback, there is a stream from clock source (DMA controller - or better SA1100), but with capture, this is not possible. This is solved with one auxiliary data stream. This stream is started whenever someone wants capture and do not play something in same time. In this case, we run special mode playback stream sending buffers of zeros to UDA1341 chip to satisfy its needs for clock pulses.

This special mode is started automatically from *trigger()* function. We have to solve situations, when one stream is started when the other is running (e.g. we are running capture and starting playback) or some stream is stopped while the other still runs (stopping playback while recording).

8.5 Extensions to Sound Card Driver

Things we discussed until now are present in OSS driver for UDA1341TS. But our ALSA driver offers much more. Our goal is full control over the chip, same as all features of ALSA. There are 19 controls on the chip, which can be used to set its exact behavior. All possible controls are: Soft Mute, Playback Volume, Bass Boost, Treble, Input and Output Gain switch, Mixer gain for channel 1 and for channel 2, Microphone sensitivity level, AGC (Automatic Gain Control) switch, AGC output level, AGC time constant, DAC Power, ADC Power, Peak detection position, Deemphasis, Mixer mode, Filter Mode and Input Amplifier Gain for channel 2. Just the listing is quite long. The OSS driver right now supports just Playback Volume, Bass, Treble, Mixer gain channels and AGC switch. Let study some control mode deeply. For speech recognition, capture stream is in the center of our interest, so some controls (for playback or other advanced features) are not involved.

- *Playback Volume*

  This control is used to adjust volume for playback stream. It ranges from value 0 (which means maximal gain) to value 61 (which means -60dB attenuation). Used step is 1dB. Complete tables for this (and all other) control could be found in [11].

- *Filter mode*

  Filter mode can be selected from three possible values. *Flat mode* does no sound processing. *Min mode* and *Max mode* add some sound processing filters with influence on Bass boost and Treble.

- *Bass Boost and Treble*

  For *Flat mode* these controls have no effect. For *Max mode*, gain up to 24dB (for Bass) or 6dB (for Treble) may be reached. In *Min mode* maximal values are 18dB and 6dB respectively.

- *Mixer mode*

  Digital mixer in UDA1341TS could work in one of four modes. First mode - *double differential* is not widely used. Next second and third mode enables channel one (or channel two) only. On this place, one detail should be explicitly said. Channel in documentation to this driver is not meant as Left and Right channel of stereo sound. Channel one is first input (named "line in") and channel two is connected to microphone. In iPAQ channel number one is not used nor connected and appropriate controls have no effect. Last mode of the mixer is *digital mixer mode.* Here two coefficients (*Mixer gain channel 1 and 2*) are used to set the ratio between two available channels.

- *Mixer Gain channel 1 and 2*

  For both channel one and two (line-in and microphone) coefficients for *digital mixer mode* are in range from 0 to 31, that mean 0 to -45dB attenuation with step of 1.5dB.

- *Input and Output gain switch*

  Apart the mixer mode, independent switches are available to obtain 6dB gain on input and output (i.e. on ADC and DAC). These should be switched on (the 6dB gain is active) or off (and no gain is added).

- *ADC feature*

  The UDA1341 chip is equipped with Automatic Gain Control feature. This feature could be switched on and off. If AGC is not used, output level is computed as combination of AGC output level and Gain Input Amplifier. If enabled, AGC is able to control level of input gain. We should set its behavior with Microphone sensitivity control and AGC output level control for desired input gain. AGC Time constant is used to set attack and decay time for AGC. These are in range of 11 to 21ms for attack time and 100 to 400 ms for decay time.

Chapter 9

Library Implementation

9.1 Audio Recording Functions

Comparing to OSS, ALSA brings more possibilities to application developer, but there is some price. Apart of increased computation power consumed by ALSA core driver and ALSA library, API of ALSA for application developer becomes more difficult and larger. To make life of application programmer easier wrapper for common audio tasks is included in the library with much simpler API. It is intended to be used in situations, where lot of lines of code for audio handling makes main program difficult to read and/or debug.

Whole audio API is build about *audio_handler* structure, that contain all informations needed to initialize and use audio services. In main program you just need to update items of this structure that do not fit your needs. API is as simple as it should be. There are three functions to handle the audio state:

- *audio_handle_t* ***audio_new***(*void*)

  Creates new audio handle structure and fills it with default values.

```
handle->format = SND_PCM_FORMAT_S16
handle->device = "default:0,0"
handle->rate = 11025
handle->channels = 1
handle->buffer_time = 500000
handle->period_time = 200000
handle->sleep_time = 0
handle->mic_gain = 60

handle->upper_limit = 70
handle->lower_limit = 40

handle->debug = 0
```

Figure 9.1: Audio handle initialization

Just significant fields are shown here. Rest of them is just cleared to avoid unacceptable usage. For field descriptions see structure definition. It returns newly created audio handle or *NULL* if some error occurs while allocating memory.

- *int* **init_audio**(*audio_handle_t* *handle)

  Opens audio device and prepares it for recording. All parameters for ALSA core are set from audio handle structure. They should have correct values before calling this function. Both *pcm* and *mixer* devices are prepared and proper fields in audio handle structure are initialized (*handle→pcm, handle→mixer*)

  CAUTION: All values except *handle.micGainId* could be left on their default values. The only one field, that must be initialized properly is *handle.micGainId.* You can get microphone gain ID using *amixer* utility from *alsa-utils* package.

- *int* **done_audio**(*audio_handle_t* **handle)

  Closes all audio devices (both *pcm* and *mixer*) and frees audio handle structure. It sets the audio handle to *NULL* to disable its future usage.

Complete example of API usage could be found in *erec.cpp.* Just simple outline to make basic idea is in figure 9.2

After audio initialization, you have two possibilities. Either you could use predefined audio data gather function that uses signal processing internally or you could use signal processing function stand alone with your own data buffers.

- *int* **audio_read**(*audio_handle_t* *handle, *unsigned char* *buffer, *int* size)

  This function reads samples from capture audio device into given buffer to fill it completely. For now the routine expects 16-bit samples so the size parameter should be number of samples times two. This function returns number of bytes read (greater or equal to zero) or error code from recovery routines (negative value). For reading data from audio device it uses zero-copy technology (*snd_pcm_mmap_readi()*) to speed up audio processing.

  It should always read whole buffer and may block inside if there are not enough data available in sound card buffers.

- *void* **audio_wait**(*audio_handle_t* *handle)

  To lower CPU utilization, this function is introduced. After audio data are read, the audio gather thread (if any) should block for a while until significant amount of new data will be available in sound card buffers. This ensure, that samples will be transferred in greater chunks with greater efficiency. This function sleeps *handle.sleep_time* microseconds. For default this is set to zero and if this feature is used it should be set to time needed to fill half of buffer used in *audio_read()* function (see above example). ALSA audio initialized with *audio_init()* uses buffered blocking I/O, but this function makes the CPU utilization significantly lower providing more effective thread scheduling.

9.2 Audio Handler Contents

Now it's right time to show whole audio handler structure. Description is included as C language comments (Figure 9.3). Fields not commented are described later (or sooner) in proper sections (applies mainly for Software Gain Control - SGC). For details of implementation see source files of the library.

```
audio_handle_t  *audio_handle=NULL;

io_handle = audio_new();
audio_handle->mic_gainId = ID_MIC_GAIN;

if ((c = getopt_long(...) < 0)
    break;
switch (c) {
    case 'D':
        audio_handle->device = strdup(optarg);
        break;
    case 'r':
        err = val = atoi(optarg);
        val = val < 4000 ? 4000 : val;
        val = val > 196000 ? 196000 : val;
        if (err != val) printf("Requested rate modified to %d\n", val);
        audio_handle->rate = val;
        break;
    case 'd':
        audio_handle->debug++;
        break;
...
audio_handle->sleep_time = \
    int(1.0/(double)(audio_handle->rate) // one sample duration (in sec)
    * BUFFER_SAMPLES     // number of samples in buffer
    * sleep_part/100.0   // wait just part of audio buffer time
    * 1E6);              // we need this in microseconds

init_audio(audio_handle);
...
done_audio(&audio_handle);
```

Figure 9.2: Library usage example

9.3 Software Gain Control

Motivation and requirements for software gain control were discussed in chapter 4. Now we focus on SGC API and implementation. SGC API consists of only one function that process current buffer and if necessary modify input gain control. Because it should be useful to application, internally used function for microphone gain changing is added to exported API too.

- *void* **process_period**(*audio_handle_t* *handle, *short* *ptr, *int* frames)

  Process actual buffer and modify microphone input gain if needed. It is used internally by *read_audio()*. If used standalone it should be called immediately after buffer is filled. Every delay means longer latency in microphone gain setting. It does not modify data in actual buffer. The buffer is expected to contain 16-bit samples for now and size of used

```
typedef struct audio_handle audio_handle_t;
struct audio_handle {
    snd_pcm_format_t   format;       // sample format
    char               *device;      // capture device
    int                rate;         // stream rate
    int                channels;     // count of channels
    int                buffer_time;  // buffer length in us (for ALSA)
    int                period_time;  // period time in us (for ALSA)
    snd_pcm_sframes_t  buffer_size;  // used internally by ALSA
    snd_pcm_sframes_t  period_size;  // used internaly by ALSA

    int                sleep_time;   // sleep time in us for audio thread
    int                mic_gain;     // volume for Gain Input Amplifier
    unsigned int       mic_gainId;   // which control handle for Gain control

    snd_output_t       *log;    // output device for ALSA errors (stdout)
    snd_pcm_t          *pcm;    // handle to PCM device (ALSA)
    snd_hctl_t         *mixer;  // handle to mixer device (ALSA)
    snd_hctl_elem_t* elem_mic_gain; // handle to input gain control

    unsigned int histo[101];    // histogram 0 .. 100 %

    int long_run;       // SGC internal
    float long_param;   // SGC parameter
    int is_speech;      // SGC internal
    int after_end;      // SGC parameter
    float speech_max;   // SGC internal
    int upper_limit;    // SGC parameter
    int lower_limit;    // SGC paramete

    int debug; // debug flag - set to >0 to see debug messages
};
```

Figure 9.3: Audio handle elements

buffer is given in number of samples in last parameter (*frames*).

- *int* **set_mic_gain**(*audio_handle_t* *handle, *int* delta)

  This function may be used for additional microphone gain modifications. Control handle and actual state is determined from audio handle. Control value ranges from 0 meaning no gain to 99 meaning maximal available gain.

Now we know how to use the API, so let see how it works internally. First of all maximal value is selected from all samples in current buffer. This value is referred as *max* later. With buffers small enough (256 ... 512 samples) it should represent some kind of envelope of given signal. Then we express the maximum as percent of maximal available value (*SHRT_MAX* for 16-bit samples). This is used for histogram update, where counter for this percent is incremented. From now, when not explicitly said something else, sequence of maximal values will be treated as new signal with particular maximums as samples. Current sample then means maximal value of current buffer and so on.

Except from real signal (of maximal values) we use another one referred as *long_run* signal. Samples of this new signal are computed with equation:

$$long\_run = (long\_param * max) + (1 - long\_param) * long\_run$$

It is signal that tend to basic signal but with some delay. This new one is used for most computations because it filter peaks with very short duration that could not be speech. Default value for *long_param* is 0.1 but it is modified in dependency of utterance presence.

There is a flags for current state. Its name is *is_speech* and it is set whenever speech begin occur and cleared at speech end.

Computing speech boundary is done independently for start and end of utterance. Start of utterance occurs when current signal (*max*) is two times greater then current *long_run* signal and flag *s_speech* is not set. When start of utterance occurs, we just set this flag and start to search maximal signal value within utterance. End of utterance occurs, when actual signal is smaller then actual *long_run* signal. Note, that in utterance speech signal is increased rapidly, so if the utterance takes some time (it is speech), *long_run* signal starts to increase too. This ensures right behavior - reacting on longer duration signal increase.

In speech end event, more important things take place. We must clear the *is_speech* flag and modify input gain control if necessary. Modification is done as follows. We use maximal value from utterance (that we start searching in speech start event). We must ensure this value is both not too small (speech signal is too weak) and not to big (peaks of signal during utterance are lost). Two parameters of audio handle are used for this purpose. *lower_limit* and *upper_limit* qualify both boundaries. Default values are 40% and 70% respectively. If maximal value of current utterance gets over or under these boundaries microphone input gain is changed by the difference (but at most by *GAIN_STEP*). Limited modification is used to avoid to fast modifications of microphone signal characteristics that could confuse recognizer. Currently *GAIN_STEP* means 5% of maximal speech signal.

Above described system works quite well, but have some weakness. If there is long(er) utterance with high maximal value and then some other with much smaller one follows, *long_run* signal does not have enough time to decrease below maximum of the second utterance and it is not detected. This problem is solved by adding another parameter - *after_end*. It is set to value greater then zero in speech end event and decreased in each call to *process_period()*. It is decreased until greater then zero each call and one more time per function call if current signal is greater then *long_run* (possible start of new speech). In each function call, *long_param* is computed using this rule:

- if current signal is smaller the *long_run* and *after_end* is positive

$$long\_param = 2 * after\_end * LONG\_PARAM\_DEFAULT$$

- else

$$long\_param = LONG\_PARAM\_DEFAULT$$

Where LONG_PARAM_DEFAULT is default value for *long_param* currently set to 0.1 (10% of new value is used for *long_run*).

When debugging is on, some statistical information are printed at the end of each call to *process_period*. It contains one line with scope of maximum of current buffer in percent (displaying just range from 0 to 50% with step of 0.5%). At the end of line current maximum and *long_run* values as numbers and percents are added. With other debug messages (e.g. speech begin and speech end) this gives good idea about actual state of whole system.

9.4 Histogram of Signal Levels

For statistical purposes, histogram of maximal values for each data buffer is saved. It is side effect of software gain control (SGC), so it could not be used standalone. After initialization histogram is empty and it is updated with each call to *process_period()* function.

Used histogram does not count occurrence of all possible levels of input signal. Instead we express maximal value of current buffer in percent of main available value (e.g. *SHRT_MAX* for 16-bit samples). There are 101 different levels of signal being counted (from 0% to 100% with step of 1%). Histogram API consist of two functions:

- *void* **dump_histo**(*audio_handle_t* *handle, *FILE* *out)

  Used to print actual histogram to file *out*. *out* could be *stdout* for screen output. Histogram is internal part of audio handle *handle*. Let see example of histogram dump:

  ```
  --- sum: 355 -------------------------------

    0 ##################################################  28.17% 100
    2 ##############################################      26.20%  93
    4 ###########################################         24.23%  86
    5 ######################################              21.41%  76
  ```

  First number is sum of all measurements (*sum* printed in delimiter). Then for each non-zero counter one line is reserved. First item says what counts this counter for (e.g. % of signal maximal value). Then scope for quick orientation is printed. At the end of each line, number of samples in this counter at the rate of all samples (*sum*) in percent and number of samples in this counter absolutely are displayed. Counters with no samples (zero value) are not printed at all to make whole histogram simpler.

- *void* **clear_histo**(*audio_handle_t* *h)

  This function clears whole histogram. It sets counters for all saved values to zero.

Chapter 10

Summary

Though there are number of scientific groups all over the word that are interested in spoken language systems, current state of our knowledge in this topic is far from our ideas and dreams. Most efforts are made at speech modeling systems. It is really scientific and long-term work.

This project does not have such noble goals. Having just limited background and time, simpler targets were marked out. For our Compaq iPAQ we update Linux kernel to unstable developer branch (2.5.x) to be usable for this project and prepare whole developer environment.

Then we choose newly integrated ALSA sound system as target platform for our improvements. This selection could be now revised as good with respect to future of Linux kernel and its abilities. On the other hand this system is new, not fully completed and poorly documented for both application and mainly driver developers. It needs much time to study internals of ALSA to get basic knowledge of sound driver implementation model. Writing the driver itself does not take so much time, but catching and removing of bugs was extremely time consuming. Absence of usable documentation and even comments in source code is main negative of this system. The same problem was solved again in example preparation, but for application programmers at least some documentation is available. Mailing lists and support from ALSA founders Jaroslav Kysela and Takashi Twai was the most useful source of information.

Second key component in this project - Embeded ViaVoice is much better documented. Having direct support from members of EVV development team Tomáš Heran and Bořivoj Tydlitát means it was easy to get in deep enough.

Two main problems were meant to be solved in this project. Unfortunately, although there are some results, it was not possible to reach expected improvements in its full scale. With software gain control, there was not problem how and when to modify proper controls. It was shown, that for good efficiency knowledge about speech occurrence in acoustic signal is essential. This information could be given just by used recognizer and used version of EVV does not offer such feature. Whole problem was discussed with EVV developers and designers and API updates of future versions were devised.

For wide samples, analysis and propositions were prepared, but for absence of time it was not implemented. Offered implementation expected huge modifications in ALSA system. This is impossible without detailed knowledge of whole ALSA sound system and that knowledge is hard to be achieved without any documentation. The implementation is scheduled for future improvements of complete spoken language system.

Bibliography

[1] Matthew, M., Stones, R.: Linux začínáme programovat, Praha, Computer Press, 2000

[2] Psutka, J.: Komunikace s počítačem mluvenou řečí, Praha, Academia, 1995

[3] Huang, X., Acero, A., Hon, H.: Spoken Language Processing: A Guide to Theory, Algorithm, and System Development, Prentice Hall, 2001

[4] Hicks, J.: Cambridge Research Laboratory/Open Handhelds ARM Bootloader, htp://www.handhelds.org/Compaq/bootldr.html (July 26, 2000)

[5] Guy, A.: Familiar v0.5.1 Installation Instructions, htp://familiar.handhelds.org/familiar/releases/v0.5.1/H3600/install.html (Jan 24, 2002)

[6] WWW of Open Sound System, http://www.opensound.com

[7] Tranter, J.: Open Sound System Programmer's Guide, version 1.11, 4Front Technologies, Jan 5, 2000

[8] Bartels, S.: ALSA 0.5.0 Developer documentation, revision Nov 21, 1999 http://www.math.tu-berlin.de/˜sbartels/alsa

[9] Kysela, J. and others: ALSA project - the C library reference, Jan 3, 2002 (for library version 0.9) http://www.alsa-project.org/alsa-doc/alsa-lib

[10] Sessink, V.: Alsa-sound-mini-HOWTO, revision v2.0-pre1, 12 November 1999 http://www.alsa-project.org/˜valentyn

[11] UDA1341TS Economy audio CODEC for MiniDisc home stereo and portable applications, Philips Semiconductors, June 29, 2001 http://www-us9.semiconductors.com/acrobat/datasheets/UDA1341TS_3.pdf

[12] ViaVoice for multiplatform - product documentation , internal document of IBM

Appendix A

Description of used hardware

A.1 Compaq iPAQ

Compaq iPAQ is inspiration for all producers of PDA today, mainly because of its elegant design. Main part of the device is occupied by LCD display and there are just five general buttons and one special 5-way joystick. Even when originally shipped only with preinstalled Microsoft Windows CE, it's not so difficult to overwrite the flash with prepared image of Linux system. Because all manipulations with the flash memory are extremely dangerous, and there is possibility to damage the OS Loader, Compaq offers special services for customers with so broken devices. When you got over initial troubles, you got fully-featured Linux machine with usable performance. Project of porting Linux to ARM architecture (even not only and specially to embedded devices) is very successful and it takes just days between the version for Intel x86 branch is out and that one for ARM architecture is ready to use.

Model	Compaq iPAQ PocketPC H3650
Processor	206-MHz Intel StrongARM SA-1110 32-bit RISC Processor
Memory	32-MB SDRAM, 16-MB Flash ROM Memory
Display	Color (4096 colors (12 bit) touch-sensitive reflective TFT LCD
Power Supply	950 mAh Lithium Polymer, rechargeable in docking cradle or with AC Adapter
Operating System	Familiar Linux distribution ver. 4.0
TFT Color Display	Number of Colors 4096
	Resolution (WxH) 240 x 320
	Dot Pitch 0.24 mm
	Viewable Image (WxH) 57.6 x 76.82 cm (96 mm diagonal)
	Display Type Color (4096 colors (12 bit) reflective TFT LCD
System Unit	Dimensions (HxWxD) 12.99 x 8.33 x 1.57 cm
	Weight 178.6 g (including battery)
Battery Life	32 MB up to 12 hours, 950 mAh Lithium Polymer rechargeable battery

Table A.1: iPAQ specification

A.2 Intel StrongARM SA1110

Intel Strong ARM was devised to meet the requirements of embedded devices and it's applications. It has rich palette of power management routines, and good performance even for multimedia applications. There is only one penalty. Because of its complexity and power demands, there is no floating point unit. All floating point operations must be done via software library, that provides emulation for basic routines. Whole ARM architecture is based on RISC principles. Model named SA-1110 is used in the iPAQ PDA. It's little enhanced variant of the basic model of this product line named SA-1100. The SA1100 and SA1110 processors implement the ARM v4 architecture standard. Architectural enhancements beyond the ARM v4 are implemented through use of coprocessor. Control register of coprocessor provide access to MMU, cache, and write and read buffer.

The SA-1110 MMU provide separate 32-entry translation look-aside buffers (TLBs) for the instruction and data streams. The SA-1110 contains 16 KB of instruction cache and 8 KB of data cache. In addition to this, a minicache is provided to prevent periodic large data transfers from thrashing the main data cache.

The SA-1110 also provides a write buffer and a read buffer. The read buffer allows critical data to be prefetched under software control, preventing pipeline stalls from occurring during external memory reads. The write buffer provides additional system efficiency by buffering between the CPU clock frequency and the actual bus speed when data is being written by the CPU to external memory.

Power management provides three modes of operation: normal, idle, and sleep. In normal mode, the CPU and peripherals are fully powered, but receive active clocks only when in use. In idle mode, clocks to the CPU are stopped, but the clocks to the peripheral functions are active. In sleep mode, once DRAM is placed in self-refresh, all functions are disabled except for the real-time clock. Wake-up from sleep occurs upon a preprogrammed interrupt.

High performance	235 MIPS @ 206 MHz (Dhrystone 2.1)
Low power (normal mode)	< 400 mW @1.75 V/206 MHz
Power-management features	Normal (full-on) mode Idle (power-down) mode Sleep (power-down) mode
Big and little endian op. modes	
3.3-V I/O interface	
Memory bus	Interfaces to ROM, synchronous mask ROM (SMROM), Flash, SRAM, SRAM-like variable latency I/O, DRAM, and synchronous DRAM (SDRAM)
32-way set-associative caches	16 KB instruction cache 8 KB write-back data cache
32-entry MMU	Maps 4 KB, 8 KB, or 1 MB
Write buffer	8-entry, between 1 and 16 bytes each
Read buffer	4-entry, 1, 4, or 8 words

Table A.2: Intel StrongARM specification

A.3 Philips UDA 1341TS

The UDA1341TS contains single-chip stereo Analog-to-Digital Converter (ADC) and Digital-to-Analog Converter (DAC) with signal processing features employing bitstream conversion techniques. Its fully integrated analog front end, including Programmable Gain Amplifier (PGA) and a digital Automatic Gain Control (AGC). Digital Sound Processing (DSP) features with the virtue of its low power and low voltage characteristics make it a very good choice for embedded and portable devices which are devised to use some kind of speech interaction with the user. DSP features include de-emphasis, volume bass boost, treble and soft mute. The chip is controlled via *L3* interface.

The chip accommodates slave mode only, this means that in all applications the system devices must provide the system clock. It supports multiple data formats as shown in table A.3.

General	Low power consumption
	3.0 V power supply
	256fs , 384fs or 512fs system clock frequencies (f_{sys})
	Small package size (SSOP28)
	Fully integrated analog front end including digital AGC
	ADC plus integrated high-pass filter to cancel DC offset
	Overload detector for easy record level control
	Separate power control for ADC and DAC
	Functions controllable via L3-interface
Data interface	I^{2}S-bus, MSB-justified and LSB-justified format compatible
	Three combinational data formats with MSB data output
	and LSB 16, 18 or 20 bits data input
	1f$_s$ input and output format data rate
DAC digital sound processing	Digital dB-linear volume control
	Digital tone control, bass boost and treble
	Soft mute

Table A.3: UDA1341TS specification

Appendix B

Familiar 0.5.1 Modification

Default runlevel for Familiar distribution is 2. We will use it for emergency communication mode. We define new run level 3 for network communication over serial line.

First we need to change default parameters of the bootloader. Last versions of CRL/OH bootloader does not need params partition anymore. There is regular file named */boot/params* on the filesystem:

```
set linuxargs=noinitrd root=/dev/mtdblock1 init=/linuxrc
lcdlight 1
```

Content of this file are commands for bootloader as they will be put to serial console command line. The main reason is to set kernel argument *console* to kernel's default value to free serial port. Be careful of error while editing this file, because it can make you troubles. Be sure, you are not using default *params* file from Familiar website. It's prepared for configuration with params partition and it could completely destroy your root image.

Next we set switching of runlevels. As the last command run in runlevel 2 initialization there will be call to

```
#!/bin/sh
/etc/init.d/lvl3 &
```

File with this should be named *S99lvl3*, placed in */etc/rc2.d* directory and have rights set to *766*. Running command *lvl3* in the background is very important, it allows the init process to run login prompt and to log in the system when there is some problem. Command */etc/init.d/lvl3* is also executable and it waits for 30 second and then switch the runlevel to 3. Printing PID of lvl3 is very useful, because in the case of troubles we will log in and kill it to stay in runlevel 2.

```
#!/bin/sh
echo "" > /dev/console
echo "Waiting 30s before switching to lvl 3!" > /dev/console
echo "lvl3 pid: $$" > /dev/console

for i in 5 10 15 20 25 30
do
  sleep 5
  echo -n . >/dev/console
done
```

```
init 3
```

Other scripts that run in runlevel 2 should be turned off by renaming them from *SXXname* to *KXXname*. Next step will be preparing run level 3 for first run when we need to update or add some files from workstation. In runlevel 3 we will need network configuration, PPP daemon and at least SSH daemon. Mounting some volumes from workstation and offering others via NFS will be helpful. Here is listing of */etc/rc3.d*:

```
S05networking -> /etc/init.d/networking
S10ppp -> /etc/init.d/ppp
S15mount -> /etc/init.d/mount
S20copy
S50ssh -> /etc/init.d/ssh
S60nfslock -> /etc/init.d/nfslock
S65nfs -> /etc/init.d/nfs
```

Scripts from */etc/init.d* that differ from distribution of Debian ARM port follows. Script *S20copy* is used to add missing files by copying them from mounted volume from workstation. We need *portmap*, *nfs* daemons, *ssh* daemon with configuration files, *bash* and necessary libraries for these programs to name just some of the basic. First listing is *ppp*:

```
#!/bin/sh
modprobe ppp0
/usr/sbin/pppd
```

The second one is *mount* for mounting volume from the workstation:

```
#!/bin/sh
modprobe nfs
mount /mnt/sirion
echo "NFS sirion:/usr/local/arm -> /mnt/sirion"
```

Next important thing is modification of PPP options file. New content of */etc/ppp/options* is:

```
updetach
nocrtscts
lock
lcp-echo-interval 5
lcp-echo-failure 3
/dev/ttySA0
115200
local
defaultroute
passive
persist
```

To wake this system up, changes in */etc/inittab* are primary. We need to add one line for newly defined runlevel (3) and restrict serial port usage of console to runlevel 2:

```
...
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l6:6:wait:/etc/init.d/rc 6
...
T0:2:respawn:/sbin/getty -L ttySA0 115200 vt100
```

Minor but also important modifications include changes in */etc/fstab, /etc/hosts, /etc/modules.conf*, creating directories to mount NFS volumes in and symbolic links for shell or */usr/local* directory.

When we complete everything, reboot iPAQ few times to add missing lines somewhere and add some more useful software and iPAQ will be as good developer's machine as any workstation. (Just a bit slower :)