UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

IP-telefonie v Linuxu se zaměřením na vestavěné systémy

DIPLOMOVÁ PRÁCE

2009 **Bc. Pavel PAVLÍK**

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

IP-telefonie v Linuxu se zaměřením na vestavěné systémy

Bc. Pavel Pavlík

Diplomová práce
2009

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Katedra softwarových technologií
Akademický rok: **2008/2009**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Pavel PAVLÍK**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**

Název tématu:	**IP-telefonie v Linuxu se zaměřením na vestavěné systémy**

Z á s a d y   p r o   v y p r a c o v á n í :

Navrhněte a vytvořte aplikaci pro společnost RADOM, s. r. o., která bude umožňovat IP-telefonii v operačním systému Linux. Aplikace musí fungovat naprosto spolehlivě s důrazem především na dobrou kvalitu zvuku a co možná nejkratší dobu zpoždění (max. 0,5 s). Dále aplikace musí spolehlivě fungovat na vestavěných systémech (konkrétně EXM32 od MSC), na kterých bude reálně využívána k přenosu zvuku.
* V úvodní části práce bude představen zvukový server v Linuxu (ALSA). Instalace, jak funguje, jak ho využít pro vlastní aplikace, důležité funkce API, ukázky jednoduchý zvukových aplikací, ovládání hlasitosti atd.
* Další část se bude zabývat síťovou komunikací v operačním systému Linux (sokety) a mutlithreadingem. Jak fungují, důležité funkce, jednoduché ukázky použití atd. Také bude krátce popsána kompilace pro jiné platformy (tzv. cross-compiling).
* Dále se již přistoupí k samotnému návrhu, implementaci a testování aplikace. Návrh bude doplněn příslušnými diagramy UML. Budou popsány způsoby testování a řešení případných problémů. Musí být vyřešena synchronizace komunikujících stran a komprese komunikace (předpokládá se použití kodeku gsm). Konečná verze aplikace musí dosahovat nízkého zpoždění, dobré kvalitu zvuku a musí fungovat plně duplexně.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

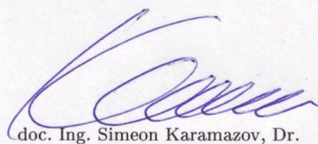
Seznam odborné literatury:

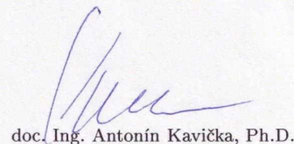
1. **MATTHEW, Neil, STONES, Richard. Linux Začínáme programovat. 2000. 912 s. ISBN 8072263072.**
2. **PRATA, Stephen. Mistrovství v C++. 2007. 1120 s. ISBN 978-80-251-1749-1.**
3. **ECKEL, Bruce. Myslíme v jazyku C++ – knihovna programátora. 2000. 556 s. ISBN 80-247-9009-2.**
4. **KYSELA, Jaroslav, et al. ALSA API – doc [online]. c2008 [cit. 2008-10-05]. Text v angličtině. Dostupný z WWW: <http://www.alsa-project.org/alsa-doc/alsa-lib/>.**

Vedoucí diplomové práce: **Mgr. Tomáš Hudec**
Katedra informačních technologií

Datum zadání diplomové práce: **31. října 2008**

Termín odevzdání diplomové práce: **22. května 2009**





doc. Ing. Simeon Karamazov, Dr.
děkan

L.S.

doc. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 4. listopadu 2008

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně Univerzity Pardubice.

V Pardubicích dne 17. 5. 2009

Pavel Pavlík

ANOTACE

Práce se zabývá IP-telefonii v OS Linux. Cílem práce je vytvořit aplikaci umožňující přenos hlasu v podobné kvalitě jako u běžného hovoru přes mobilní telefon. Výsledná aplikace je primárně určena pro vestavěný systém EXM32, kde musí bez problému fungovat.

KLÍČOVÁ SLOVA

IP-telefonie; VoIP; Linux; ALSA; sokety; UDP

TITLE

IP-telephony in Linux with a view to embedded systems

ANNOTATION

This dissertation is about IP-telephony in OS Linux. The target of the dissertation is to create such an application which makes possible voice transmission over IP network in the quality of cell phone calls. The final application is suitable mainly for embedded system EXM32, where it should work without any problems.

KEYWORDS

IP-telephony; VoIP, Linux; ALSA; sockets; UDP

OBSAH

1. ÚVOD ... 8

2. DIGITALIZACE ZVUKU ... 9

2.1 Proč digitalizovat ... 9
2.2 Jak se zvuk digitalizuje ... 11
2.2.1 Použití kodeku GSM v OS Linux ... 12
2.3 Přenos multimediálních dat IP sítí ... 13
2.3.1 Architektura TCP/IP ... 13
2.3.2 TCP a UDP přenos ... 14
2.3.3 Speciální protokoly pro multimediální přenos ... 15
2.3.4 Sokety a UDP přenos v OS Linux ... 16

3. ZVUKOVÉ SYSTÉMY V OPERAČNÍM SYSTÉMU LINUX ... 20

3.1 Obecný princip přehrávání a nahrávání ... 21
3.2 OSS (Open Sound System) ... 22
3.3 ALSA (Advanced Linux Sound Architecture) ... 22
3.3.1 O zvukovém systému ALSA ... 22
3.3.2 Architektura zvukového systému ALSA ... 23
3.3.3 Instalace zvukového systému ALSA ... 25
3.3.4 Ovládání směšovače ... 27

4. PROGRAMOVÁNÍ ZVUKOVÉ APLIKACE PRO SYSTÉM ALSA ... 29

4.1 Popis důležitých termínů a parametrů ... 29
4.1.1 ALSA zařízení a základní parametry ... 29
4.1.2 Hardwarové parametry a chyby xrun ... 30
4.1.3 Softwarové parametry ... 32
4.2 Základní struktura programu pro zvukový systém ALSA ... 33
4.2.1 Otevření zařízení ... 33
4.2.2 Nastavení HW parametrů ... 34
4.2.1 Nastavení SW parametrů ... 36
4.2.2 Zápis a čtení ALSA bufferu ... 37
4.2.3 Uzavření zařízení ... 39

5. KOMPILACE PRO VESTAVĚNÝ SYSTÉM A JEHO TESTOVÁNÍ ZHLEDISKA VOIP ... 40

5.1 Kompilace pro jinou platformu ... 40
5.2 Testování vestavěného zařízení pro VoIP ... 41
5.2.1 Testování dostatečného výkonu ... 41
5.2.2 Testování spolehlivosti zvukového systému ... 44

6. NÁVRH APLIKACE PRO VOIP ... 46

6.1 Obecný princip IP-telefonie ... 46
6.2 Architektura aplikace ... 48
6.2.1 Shrnutí požadavků ... 48
6.2.2 Reprezentace v paměti a vlákna ... 49

6.2.3 Návrh tříd a jejich UML diagram 50

7. IMPLEMENTACE NAVRŽENÉ APLIKACE 54

7.1 Implementace navržených tříd 54

7.1.1 Vlákna v OS Linux 54
7.1.2 Implementace třídy pro použití kodeku GSM 55
7.1.3 Implementace třídy pro sdílený buffer 56
7.1.4 Implementace tříd pro UDP komunikaci 56
7.1.5 Implementace třídy výjimek 57
7.1.6 Implementace třídy pro přehrávání a nahrávání 57
7.1.7 Implementace tříd pro logování 58
7.1.8 Implementace zapouzdřující třídy 59
7.1.9 Implementace vláken pro nahrávání a odesílání 60
7.1.10 Implementace vlákna pro příjem dat 61
7.1.11 Implementace vlákna pro přehrávání 63
7.1.12 Implementace vlákna pro kontrolu spojení 63
7.1.13 Shrnutí implementace navržených tříd 64

7.2 Implementace konzolové aplikace 65

7.3 Implementace GUI aplikace pro testovací a prezentační účely 66

7.4 Implementace kontrolního mechanismu 69

8. ZÁVĚR 73

Seznam obrázků

Obr. 1) Vzorkování signálu 11
Obr. 2) Kvantování 12
Obr. 3) Digitalizace analogového signálu ve formě sinusové křivky 12
Obr. 4) Architektura TCP/IP 14
Obr. 5) Schéma zvukového systému v OS Linux 20
Obr. 6) Princip přehrávání 21
Obr. 7) Architektura zvukového systému ALSA 24
Obr. 8) Architektura ALSA systému a tok požadavků 25
Obr. 9) Nastavení ALSA v kernelu 26
Obr. 10) Alsamixer 28
Obr. 11) Generování testovacích dat 42
Obr. 12) Přehrávání SH7760 43
Obr. 13) EXM32, SH7760 starterkit 43
Obr. 14) Informační tok procesů při IP-telefonii 46
Obr. 15) Rekonstrukce analogového signálu 47
Obr. 16) Primární využití VoIP aplikace 49
Obr. 17) Architektura, dvě vlákna 49
Obr. 18) Architektura, čtyři vlákna, dva sdílené buffery 50
Obr. 19) Základní návrhový diagram tříd 51
Obr. 20) Pokročilý návrhový diagram tříd 52
Obr. 21) Použití statické knihovny 53
Obr. 22) Duplikování vzorků 62
Obr. 23) Strategie přidávání a odebírání vzorků v bufferu 62
Obr. 24) Qt Creator 67
Obr. 25) Vyvinutá GUI aplikace 69
Obr. 26) Kontrolní mechanismus 69

1. Úvod

Digitalizace je současným trendem již po několik let. Přináší s sebou výhody, které je možné uplatnit jak v průmyslu, tak v domácnostech. Samotná IP-telefonie neboli přenos z digitalizovaného hlasu přes IP síť, je velice populární. IP-telefonii lze dnes bez problémů provozovat na běžných osobních počítačích pomocí aplikací, které jsou dostupné pro každý moderní operační systém.

Společnost RADOM, s.r.o. se rozhodla ve svých nových komunikačních systémech, založených na vestavěných zařízeních EXM32 s OS Linux, využít výhod digitálního přenosu zvuku oproti stávajícímu analogovému. Na internetu je možné získat celou řadu softwaru umožňujícího IP-telefonii v operačním systému Linux ([35], [36], [37], [38]), ale žádný nevyhovoval z důvodu funkčnosti nebo licenčních podmínek.

Proto bylo rozhodnuto vyvinout vlastní aplikaci umožňující IP-telefonii v operačním systému Linux, která bude zajišťovat následující charakteristiky:

- bezproblémový chod na daném vestavěném systému,
- vlastnosti aplikace přesně podle daných potřeb,
- možnost dalších budoucích úprav,
- možnost použití jak v konzoly, tak v grafickém prostředí.

Jednotlivé části práce popisují prostředky a technologie, které byly nezbytnou součástí vývoje této aplikace. Práce se především zabývá: digitalizací zvuku, zvukovými systémy v OS Linux, vývojem zvukových aplikací v OS Linux, přenosem multimediálních dat IP sítí, testování vestavěného zařízení ohledně přenosu zvuku a samotným návrhem i implementací výsledné aplikace.

2. Digitalizace zvuku

Pro vývoj softwaru, který se zabývá IP-telefonií, je nutné mít alespoň základní informace o digitalizaci. IP-telefonie je vlastně digitální přenos zvuku. V této kapitole bylo využito zdrojů [1], [2], [3].

2.1 Proč digitalizovat

Digitalizace analogového signálu nám přináší celou řadu výhod. Tyto výhody plynou ze samotné podstaty digitalizace. Při analogovém přenosu nás zajímá přesná hodnota nějaké veličiny (např. okamžitá hodnota napětí), zatímco při digitálním přenosu nás zajímá pouze to, jestli hodnota spadá do určitého intervalu nebo nikoli (např. je-li hodnota větší než 5 V či ne). Dále s digitálními daty se manipuluje daleko lépe než s analogovým signálem. Z těchto faktů plynou dle mého názoru nejpodstatnější výhody digitalizace:

- lze přenést hodnotu s ideální přesností, kvalita dat se při přenosu nemění (odolnost proti rušení, útlumu atd.),
- chybovost dat lze účinně minimalizovat a detekovat (samoopravné kódy, CRC, kontrola parity, atd.),
- přenos je efektivnější a lze dosahovat vyšších přenosových rychlostí (např. při analogovém vysílání TV je na jednom kanále pouze jeden program, při digitálním několik),
- je možné zajistit bezpečný přenos, a to jak z hlediska spolehlivosti (potvrzování), tak z hlediska zneužití (šifrování),
- digitální data lze snadno zpracovávat a modifikovat (aplikace různých digitálních filtrů, komprimace atd.),
- je možné naráz přenášet různé druhy provozu (zvuk, obraz, jiná data).

Těchto výhod lze samozřejmě dobře využít i při digitálním přenosu zvuku. Podle zdroje [2] je při běžném analogovém telefonním hovoru přenášeno ticho až 61,5 % celkového času (pauzy v řeči, poloviční využití obousměrné komunikace). Navíc lidskou řeč lze s mírnou ztrátou kvality osmkrát až dvanáctkrát zkomprimovat (např. pomocí

kodeku GSM). Tyto dvě věci vedou k několikanásobnému zvýšení efektivnosti přenosových cest v digitální telefonní síti.

Na digitálně přenášený zvuk lze navíc snadno aplikovat různé digitální filtry, které mohou sloužit k různým účelům např.:

- detekce ticha či tónů na určité frekvenci,
- filtrování tónů na určité frekvenci,
- eliminace vlastních ozvěn (tzv. echo canceller),
- upravování vlastností přenášeného zvuku.

Ovšem digitální přenos zvuku má i svoji nevýhodu, která při službě probíhající v reálném čase může být velice problematická. Jedná se o zpoždění, které vzniká v důsledku přenosu a zpracování digitálních dat. Při klasickém analogovém hovoru, který si můžeme nadneseně představit jako „dvě sluchátka spojená kabelem“, je zpoždění prakticky nulové. Zpožděním je myšlena doba, za jakou druhá strana uslyší to, co první řekne a naopak. Při VoIP (Voice over IP) přes Internet může být zpoždění značné. Dle mého názoru, který vychází z řady experimentů, je maximální rozumné zpoždění u VoIP 0,5 sekundy. To znamená, že řekne-li něco jedna strana, dozví se odpověď od druhé strany nejdříve za 1 sekundu. Při větším zpožděním než 0,5 sekundy si obě strany neustále skáčou do řeči a podobně. Aby k tomuto nedocházelo, je potřeba především zajistit:

- připojení do internetu s dostatečným datovým tokem, krátkou odezvou a minimálními výpadky,
- dostatečný výpočetní výkon zařízení, na kterých je VoIP provozována.

Internetové připojení s dostatečnými parametry lze v dnešní době pořídit v celku bez problému a výpočetní síla dnešních osobních počítačů několikanásobně převyšuje požadovaný výkon – i díky tomu stoupá obliba IP-telefonie. Ovšem nedostatečný výkon může být limitující pro řadu vestavěných zařízení.

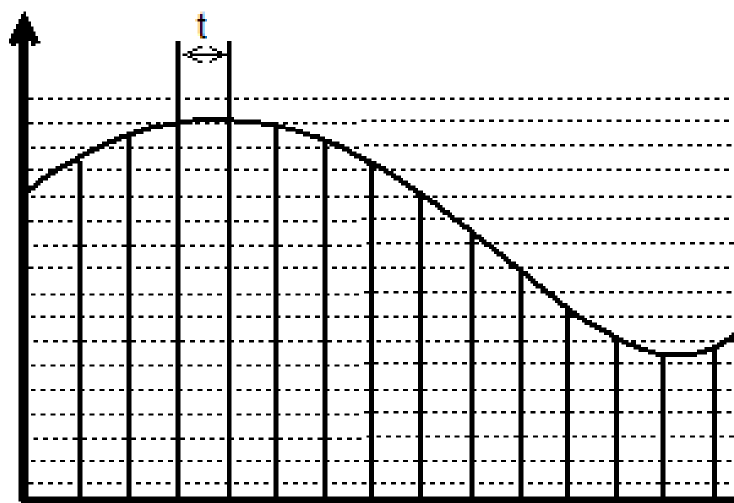
2.2 Jak se zvuk digitalizuje

Proces převodu analogového signálu do digitálních dat (pouze jedničky a nuly) je rozdělen do třech částí a probíhá následovně. Nejprve z analogového signálu odečítáme v intervalech aktuální hodnoty, tomu se říká *vzorkování*. Odečtené hodnoty vyjádříme jako digitální čísla z určitého rozsahu, tomu se říká *kvantování*. Nakonec se získaná data komprimují nebo jinak upravují, tomu se říká *kódování*.

S těmi to třemi úkony souvisejí tři otázky:

1. Jak často vzorkovat?
2. Jak velký rozsah hodnot při kvantování zvolit?
3. Jak zmenšit získaná data na co možná nejmenší velikost?

Na první otázku odpověděl již v roce 1928 H. Nyquist pomocí Fourierova rozvoje, *optimální je vzorkovat dvakrát za periodu.* Jedná se o tak zvaný Nyquistův teorém (Nyquistův-Shannonův teorém). Při přenosu zvuku se používá nejčastěji vzorkovací frekvence 8000 Hz. Tato hodnota vychází z toho, že pro digitalizaci se bere rozsah 0 až 4000 Hz ($2 \cdot 4000 = 8000$), což je dostatečné pro srozumitelnost hovoru. Lidské ucho má však schopnost vnímat rozsah daleko vyšší (cca 20 až 20000 Hz). Obrázek Obr. 1 zobrazuje vzorkování signálu (zdroj vlastní).



Obr. 1) Vzorkování signálu

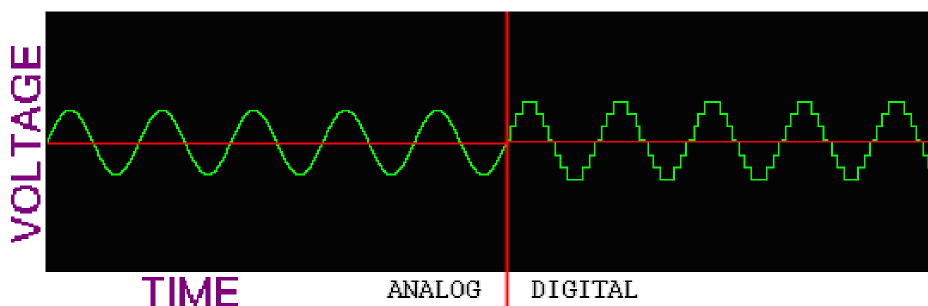
Při kvantování dochází k přiřazení k nejbližší diskrétní hodnotě a při tom vzniká tak zvaný kvantizační šum. Obrázek Obr. 2 zobrazuje kvantování (zdroj vlastní). Čím je

kvantizační šum menší, respektive rozsah hodnot větší, tím je zvuk přesněji zachycen. Při digitalizaci zvuku se používá nejméně 256 hodnot (1 bajt), což je sice pro srozumitelný hovor dostatečné, ale kvalita není nejlepší. Proto se nejčastěji používají pro kvantování 2 bajty, které už poskytují dostatečné množství hodnot (Experimentálně je ověřeno, že dostačuje 12 bitů.).

Obr. 2) Kvantování

Pro kódování při přenosu zvuku (hovoru) se dnes běžně používají kodeky, které při standardním nastavení (8000 Hz, 2 bajty na vzorek) např. generují datový tok 13 kbits/s (GSM Codec), 16 kbits/s (G728), atd., což je několikanásobné zmenšení oproti nekomprimovanému datovému toku, při zanedbatelné ztrátě kvality hovoru.

Pro lepší představu o digitalizaci analogového signálu přikládám obrázek Obr. 3. Předloha pro tento obrázek pochází ze zdroje [8]. Levá část obrázku zachycuje analogovou podobu, pro testovací účely často používané, dokonalé sinusové křivky. Pravá část zobrazuje její digitální podobu.



Obr. 3) Digitalizace analogového signálu ve formě sinusové křivky

2.2.1 Použití kodeku GSM v OS Linux

Ve většině distribucí Linuxu je kodek GSM k dispozici a jde bez problémů nainstalovat. V některých distribucích je rozdělen na dvě části. První část obsahuje sdílené knihovny a aplikace pro používání kodeku, druhá obsahuje statické knihovny a hlavičkové soubory určené pro vývoj. Pro ladění a vývoj aplikací, které používají kodek

GSM, jsou samozřejmě potřeba části obě. V některých distribucích Linuxu je ovšem dodán hlavičkový soubor „gsm.h“, se kterým není možné použít kodek GSM s programovacím jazykem C++, ale pouze s jazykem C. Jako přílohu A přikládám lehce modifikovaný hlavičkový soubor „gsm.h“ použitelný pro jazyk C++.

Samotné použití kodeku není složité, v podstatě se jedná o čtyři funkce:

- `gsm_create()` – inicializace kodeku,
- `gsm_encode()` – zakóduje 160 dvoubajtových vzorků do 33 bajtů,
- `gsm_decode()` – dekóduje 33 bajtů do 160 dvoubajtových vzorků, v případě neúspěchu funkce vrací -1, jinak 0,
- `gsm_destroy()` – uvolnění zdrojů, které kodek používá.

Zakódovaná data ovšem nelze žádným způsobem míchat, rozdělovat či spojovat. Dekódovat je lze úspěšně pouze tehdy, když jsou do funkce `gsm_decode()` dodána přesně tak, jak je zakódovala funkce `gsm_encode()`. V případě, že se dekódování relativně správných dat nepovede, a to několikrát za sebou, je nutné kodek opět inicializovat (experimentálně se to stávalo po dekódování zhruba dvou miliónů částí). Při kompilaci programu využívající kodeku GSM je nutné linkovat příslušnou statickou knihovnu („libgsm.a“). Některé další informace, včetně krátkého příkladu, lze najít v manuálových stránkách kodeku (pro Linux příkaz man 3 gsm).

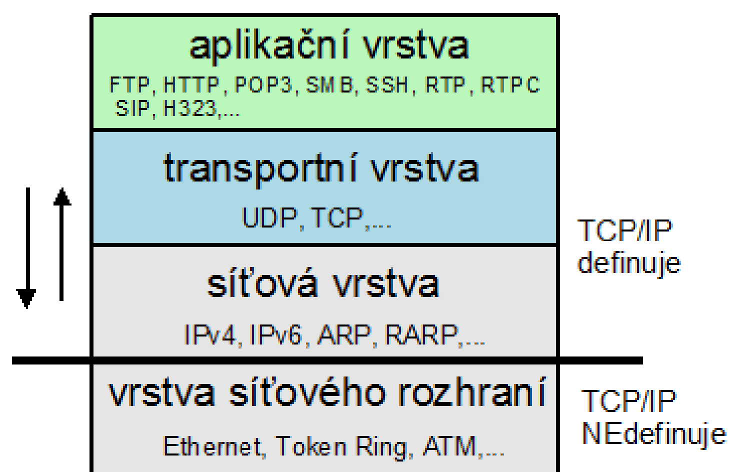
2.3 Přenos multimediálních dat IP sítí

Multimediální data (zvuk, video) se samozřejmě přenášejí stejným způsobem jako data jiná, proto je nutné znát minimálně základy dnes používané síťové architektury. V následujících podkapitolách byly použity zdroje [14] až [20].

2.3.1 Architektura TCP/IP

Dnešní sítě jsou založeny na rodině protokolů TCP/IP. Samotná architektura TCP/IP vznikala v osmdesátých letech minulého století na síťových seminářích, které vedli V. Cerf a R. Kahn. Od roku 1983 přešel ARPANET (předchůdce dnešního Internetu) na protokoly TCP/IP. Architektura TCP/IP je rozdělena do čtyř vrstev, každá vrstva poskytuje vyšší vrstvě nějaké služby. Tyto vrstvy lze procházet pouze směrem naho-

ru (příjem dat) nebo dolů (odesílání dat), jednotlivé vrstvy nelze žádným způsobem přeskakovat. TCP/IP definuje jednotný přenosový protokol (protokol IP), jednotné transportní protokoly (TCP, UDP) a jednotné základy aplikací (přenos souborů, email, atd.). Ovšem TCP/IP nedefinuje použité sítě a přenosové technologie. To znamená, že na nich nezáleží. Je jedno, o jakou se jedná konkrétně síť (GSM, kabelová TV, bezdrátové technologie, atd.), vše je přikryto jednotnou pokličkou protokolu IP. IP dokáže fungovat nad každou technologií, která fyzicky umí přenášet data. Architektura TCP/IP je zobrazena na Obr. 4 (zdroj vlastní).



Obr. 4) Architektura TCP/IP

IP protokol, který je součástí síťové vrstvy zajišťuje pouze nespojovaný nespolehlivý přenos. Není zde žádný mechanismus zajišťující kvalitu služby. Může docházet ke ztrátám přenášených paketů (rámec dat), a to bez varování i beze snahy o nápravu. Kapacita linky je plně využita pro přenos bez jakékoli režie. Filozofie TCP/IP protokolu je taková, že zajištění spolehlivosti komunikace je starostí koncových účastníků komunikace. K této myšlence samozřejmě existuje několik důvodu, vždy nepotřebujeme spolehlivý přenos, levnější a jednodušší stavba sítě, atd.

2.3.2 TCP a UDP přenos

Transportní vrstva poskytuje aplikacím dva základní protokoly, které jsou nadstavbou nad protokolem IP. Jsou to protokoly UDP (User Datagram Protocol) a TCP (Transmission Control Protocol). UDP komunikace je nespojovaná a nespolehlivá. TCP komunikace je stavová, spojovaná a spolehlivá (spojovaný přenos je emulován).

UDP protokol je jen lehkou nadstavbou nad protokolem IP, je pouze přidána identifikace služby (číslo portu) a kontrolní součet. UDP komunikace neobsahuje žádný kontrolní mechanismus. Odeslaná data mohou k adresátovi dojít v jiném pořadí či mu nemusí dojít vůbec a odesílatel se o tom vůbec nedozví.

TCP komunikace je omnoho složitější, rozlišuje několik stavů, odeslaná data musí adresát potvrdit, jinak jsou odeslána znova (metoda posuvného okénka), atd. Ovšem na oplátku za svojí složitost protokol TCP zaručuje, že data budou v pořádku doručena, a to dokonce ve správném pořadí.

Potvrzování a řízení TCP komunikace ovšem stojí nějakou režii, která by mohla být využita k samotnému přenosu. I když pro většinu přenosů v dnešních sítích je použit protokol TCP, existují aplikace, pro které je co možná největší rychlost komunikace důležitější než její spolehlivost. IP-telefonie a všechny multimediální přenosy probíhají-cí prakticky v reálním čase jsou toho příkladem. Není důležité, že všechna data přijdou ve správném pořadí, ale to, že přijdou včas. Data, která nepřišla včas, nejsou pro tyto aplikace vůbec zajímavá. Proto veškeré multimediální přenosy probíhající v reálném čase jsou založeny na protokolu UDP.

2.3.3 Speciální protokoly pro multimediální přenos

Především při komunikaci ve velkých sítích ovšem nastává spousta situací, které je třeba řešit, aby např. telefonní hovor probíhal korektně (rozdílné rychlosti připojení, umístění za firewallem, apod.). Speciálně pro přenos audia a videa v reálném čase vznikla celá řada méně či více složitých protokolů (H.323, RTP, SIP, atd.), které si kladou za cíl řešit všechny aspekty realtimové komunikace přes IP sítě.

Časté je spojení protokolů SIP, RTP a SDP. Protokol SIP se stará o relace a má na starost pět následujících činností:

- lokalizace účastníka,
- zjištění v jakém je účastník stavu (je-li schopen navázat spojení),
- zjištění možností účastníka (přenosová rychlost, některé detaily o jeho HW),
- navázání spojení,
- řízení probíhajícího spojení.

Při navázání spojení vstupuje do hry protokol SDP, který popisuje navázané spojení a předá ho protokolu pro přenos dat. Samotný přenos dat je uskutečňován protokolem RTP, který se stará o přehrání dat ve správný okamžik. Do přenášeného paketu je přidána RTP hlavička, která obsahuje několik informací, díky kterým je to možné zajistit (časové razítko, pořadové číslo, atd.). Více informací o těchto protokolech je možné zjistit ze zdrojů [16], [17], [18].

Použití těchto protokolů není ovšem jednoduché. Jak je vidět na obrázku Obr. 4 tyto protokoly se nacházejí v aplikační tedy poslední vrstvě architektury TCP/IP. Z toho vyplývá, že nejsou poskytnuty předchozí vrstvou. Pro využití těchto protokolů v nějaké aplikaci je nutné buď protokoly samostatně implementovat, což je velice zdlouhavé a náročné, nebo využít implementaci třetí strany. Implementací třetích stran, které jsou pro volné použití, je ovšem momentálně velice málo. Příkladem takové implementace je zdroj [15], tento projekt s názvem *ccRTP* implementuje stejnojmenný protokol.

2.3.4 Sokety a UDP přenos v OS Linux

Komunikační rozhraní pro komunikaci po síti v operačním systému Linux se nazývá sokety. Skrze sokety lze mezi dvěma stranami komunikovat jak lokálně, tak vzdáleně. Sokety využívají architektury klient/server. Většina dnešní síťové komunikace je založena právě na soketech. Sokety vznikly na univerzitě v Berkeley a všechny moderní operační systémy toto rozhraní implementují.

Soket v operačním systému Linux si lze představit jako speciální soubor, do kterého když zapíšeme nějaká data, jsou po síti odeslána na druhou stranu spojení. Naopak když ze soketu data čteme, jsou to data, která jsme po síti obdrželi. Komunikace pomocí soketů bývá často přirovnávána k telefonnímu hovoru např. do nějaké větší organizace. Telefonní číslo do této organizace se dá přirovnat k IP adrese, která jednoznačně určuje síťový uzel, kterému mají být data doručena. Číslo oddělení, kam je hovor přepojen z recepce organizace, se dá přirovnat k číslu portu, které jednoznačně identifikuje síťovou službu. Každý příchozí hovor do organizace je přesměrován na požadované oddělení, přesně tak fungují i sokety. Nově přijatý paket je přesměrován podle čísla portu příslušné službě.

Chceme-li odeslat nějaká data protokolem UDP v OS Linux, musíme použít několik systémových volání. Struktura těchto volání může být např. následující:

1. socket(AF_INET, SOCK_DGRAM, 0),
2. connect(sockfd, address, sizeof(address)),
3. send(sockfd, data, data_size),
4. close(sockfd).

Systémové volání socket() vytvoří soket s požadovanými parametry a vrátí jeho popisovač. Parametr AF_INET nastavuje, že bude použit protokol IPv4, parametr SOCK_DGRAM nastavuje, že se jedná o komunikaci protokolem UDP (pro TCP komunikaci SOCK_STREAM).

Systémové volání connect() nastaví IP adresu a číslo portu, na které budou data skrze tento soket defaultně zasílána. Parametr socfd označuje popisovač soketu, parametr address strukturu, ve které je uložena IP adresa a číslo portu. Posledním parametrem je velikost předchozí struktury. Adresa je datového typu sockaddr_in a má následující části:

- sin_family – zde se nastavuje opět, že se jedná o protokol IPv4, tedy hodnota AF_INET,
- sin_addr.s_addr – zde se nastavuje samotná IP adresa, může být využito funkce inet_addr(), tato funkce převádí textovou reprezentaci IP adresy složenou z číslic a teček do binární podoby, např. tedy hodnota inet_addr("192.168.1.10"),
- sin_port – zde se nastavuje číslo portu, mělo by být využito funkce htons(), která zařídí správné pořadí bajtů na každém systému, např. tedy hodnota htons(1015).

Systémové volání send() odešle požadovaná data skrze daný soket. Parametr sockfd je popisovač soketu, parametr data ukazatel na data, které se odesílají, a poslední parametr je jejich délka. Místo send() by bylo možné využit i jiné volání např. write(), protože soket je, jak už bylo řečeno, také soubor. Také je možné přímo specifikovat, kam zadaná data budou odeslána, takovou to možnost poskytuje sendto().

Posledním, ale neméně důležitým voláním je `close()`. Toto volání uzavře soket a uvolní systémové zdroje, které s ním byly svázány. V případě, že soket nebyl uzavřen, není možné otevřít další soket se stejnými parametry.

Podobné je to i při příjmu dat protokolem UDP v OS Linux, ale existují některé odlišnosti. Struktura systémových volání může být např. následující:

1. `socket(AF_INET, SOCK_DGRAM, 0)`,
2. `bind(sockfd, address, sizeof(address))`,
3. `recv(sockfd, buff, max_size)`,
4. `close(sockfd)`.

Význam systémových volání `socket()` a `close()` je naprosto shodný jako u odesílání dat.

Systémové volání `bind()` přiřadí soketu adresu datového typu `sockaddr_in`, která tentokrát znamená, na kterém portu a IP adrese budou data přijímána. Pro přiřazení IP adresy může být použita opět funkce `inet_addr()` nebo v případě, že chceme, aby bylo přijímáno ze všech dostupných síťových rozhraní, funkce `htonl()` s parametrem `INADDR_ANY`. Nastavení čísla portu je opět vhodné provést pomocí funkce `htons()`. Funkce `htons()` a `htonl()` opět zařídí správné pořadí bajtů na všech systémech.

Systémové volání `recv()` přijme data z daného soketu. Parametr `socfd` je popisovač soketu a `buff` by měl být ukazatel na alokované místo o velikosti `max_size`, do kterého budou uložena přijatá data. I pro příjem je možné použít jiné systémové volání např. `read()`, `recvfrom()`. Funkce `recvfrom()` má navíc další parametr datového typu `sockaddr`, do kterého je uloženo odkud byla data přijata. Systémová volání pro příjem dat jsou ovšem obvykle[1] blokující, dokud nejsou data přijata, což nemusí být vždy žádoucí. Proto je volání pro příjem dat často spojeno s dalším systémovým voláním `select()`, které zajistí timeout podle požadavků uživatele.

Všechna zmíněná systémová volání vrací při chybě -1. O jakou chybu se jedná lze zjistit pomocí globální chybové proměnné `errno`. Systémová volání pro odeslání či

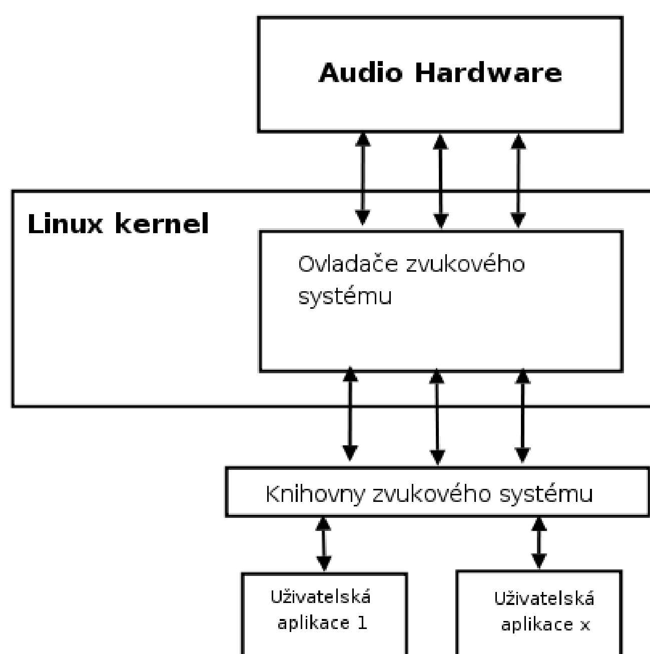
[1] Lze použít „flag“ MSG_DONTWAIT, který zařídí plně neblokující přístup.

příjem dat vrací počet odeslaných či přijatých bajtů. Více informací o komunikaci skrze sokety lze získat z příslušných manuálových stránek OS Linux (např. man 2 socket) nebo ze zdrojů [19], [20]. Jako přílohu B přikládám zdrojové kódy jednoduchých programů, které demonstrují UDP komunikaci v operačním systému Linux.

3. Zvukové systémy v operačním systému Linux

Zvukový systém v operačním systému Linux by se dal definovat jako vrstva, která je nad samotným zvukovým hardwarem a umožňuje nám ho využívat. Respektive poskytuje vyšší vrstvě, kterou tvoří uživatelské aplikace, programové prostředky k využití zvukového hardwaru. V této kapitole bylo použito zdrojů [4] až [13].

Zvukový systém je tvořen ovladači pro zvukové karty, které jsou součástí jádra operačního systému Linux. Dále pak základní sadou aplikací pro uživatele např. směšovač pro ovládání hlasitosti, jednoduchý přehrávač či jednoduchá aplikace pro záznam zvuku atd. Nakonec zvukové systémy obsahují sdílené a statické knihovny, které slouží k běhu a vývoji aplikací pro daný zvukový systém. Situace je znázorněna na obrázku Obr. 5 (zdroj vlastní).

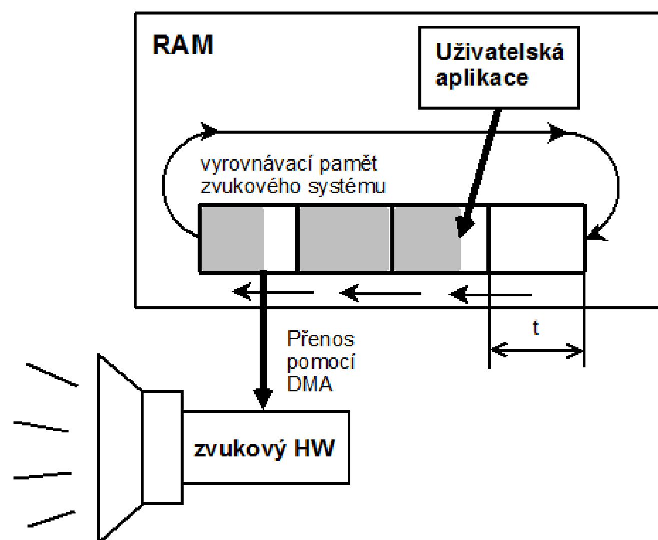


Obr. 5) Schéma zvukového systému v OS Linux

V současnosti je možné v operačním systému Linux používat zvukový systém OSS (Open Sound System) nebo ALSA (Advanced Sound System). Vzhledem k zadání se práce bude zabývat především zvukovým systémem ALSA.

3.1 Obecný princip přehrávání a nahrávání

Přehrávání a nahrávání zvuku na většině zvukových systémů funguje následujícím způsobem. Zvukové systémy mají svojí vyrovnávací paměť, která je rozdělena na několik stejných částí. Minimální počet těchto stejných částí jsou dvě, lépe však čtyři či osm. Všechny části jsou pomyslně spojeny do kruhové fronty. Při přehrávání jsou z části, která je právě na řadě přepravována data pomocí sběrnice DMA (bez účasti procesoru) ke zvukovému HW, kde jsou zpracovávána. Přehrání jedné části vyrovnávací paměti trvá nějakou konstantní dobu. Po uplynutí této doby dojde k přerušení, ve kterém je část první vyměněná za část následující, ze které se začne přehrávat. Přehranná část vyrovnávací paměti je vložena na konec fronty a jsou do ní od uživatelské aplikace vkládány další data pro přehrání. Obrázek Obr. 6 znázorňuje popsanou situaci (zdroj vlastní).



Obr. 6) Princip přehrávání

Nahrávání probíhá logicky opačným způsobem. Do části, která je na řadě jsou pomocí DMA od zvukového HW přenášena data. Při přerušení je nahraná část vyměněna za „prázdnou" část, která je na řadě. Nahraná část je umístěna na konec fronty, kde si z ní nahraná data odebírá uživatelská aplikace.

Ve skutečnosti nejsou samozřejmě prohazovány bloky dat (zbytečně velká náročnost), ale jsou pouze nastavovány ukazatele na jiné bloky.

3.2 OSS (Open Sound System)

Starší z obou dostupných zvukových systému je OSS, který v roce 1992 napsal Hannu Savolainen. OSS není dostupné pouze pro Linux, ale funguje také na dalších unixových systémech jako Solaris, FreeBSD atd. Po úspěchu projektu OSS autor založil firmu s názvem 4Front Technologies, která OSS nadále vylepšuje, přidává podporu nových zařízení a funkcí. Ovšem šíří ho jako proprietární software. Současná cena OSS pro všechny unixové systémy je cca padesát amerických dolarů.

Open Sound System byl jediným zvukovým systémem pro Linux do jádra verze 2.4. Od verze linuxového jádra 2.6 (rok 2003) je již OSS zavrhován a to hned z několika důvodů:

- v důsledku toho, že zdrojové kódy OSS již nebyly otevřené, vznikly různé verze OSS, a tím i problém s kompatibilitou (OSS/Free, OSS vX),
- podpora OSS/Free pro nový HW nebyla dostatečně pružná,
- základy systému OSS byly navrženy v době, kdy zvukový HW byl oproti současnosti značně omezen, v důsledku toho některé nyní požadované funkce (HW podpora MIDI, HW mixování, plně duplexní operace) nejsou nebo nebyly v programovém rozhraní podporovány.

Důvodů, proč je již nyní OSS zavrhován, by se našlo určitě více. Tyto důvody daly nepřímý podmět ke vzniku nového zvukového systému pro Linux s názvem ALSA.

3.3 ALSA (Advanced Linux Sound Architecture)

3.3.1 O zvukovém systému ALSA

Tento systém je vyvíjen od roku 1998. Za jeho vznikem stojí Čech Jaroslav Kysela, který doposud celý projekt řídí. ALSA odstraňuje hlavní nedostatky zastaralého zvukového systému OSS a od linuxového jádra 2.6 (rok 2003) je doporučovaným zvukovým systémem pro OS Linux. Aby nedošlo k nekompatibilitě softwaru, který je napsán pro starší OSS, a který se stále používá i na jiných unixových systémech, je ALSA z OSS zpětně kompatibilní, respektive obsahuje modul pro podporu OSS. To souvisí s řadou výhod, které hrají ve prospěch zvukového systému ALSA:

- podpora více procesorů,
- modulární architektura,
- plně duplexní operace,
- hardwarové mixování,
- hardwarová podpora MIDI,
- lepší a propracovanější API, která odstraňuje nízko úrovňové volání `ioctl()`, které je typické pro OSS,
- dobrá podpora pro všechna zvuková rozhraní, ke kterým výrobci zveřejnili dokumentaci,
- logické pojmenovávání zařízení, která např. umožňuje dynamické spojení několika zvukových proudů do jednoho či naopak,
- licence GPL.

Kladů zvukového systému ALSA by se určitě našlo více. Pro uživatele i vývojáře poskytuje zvukový systém ALSA oproti OSS více možností.

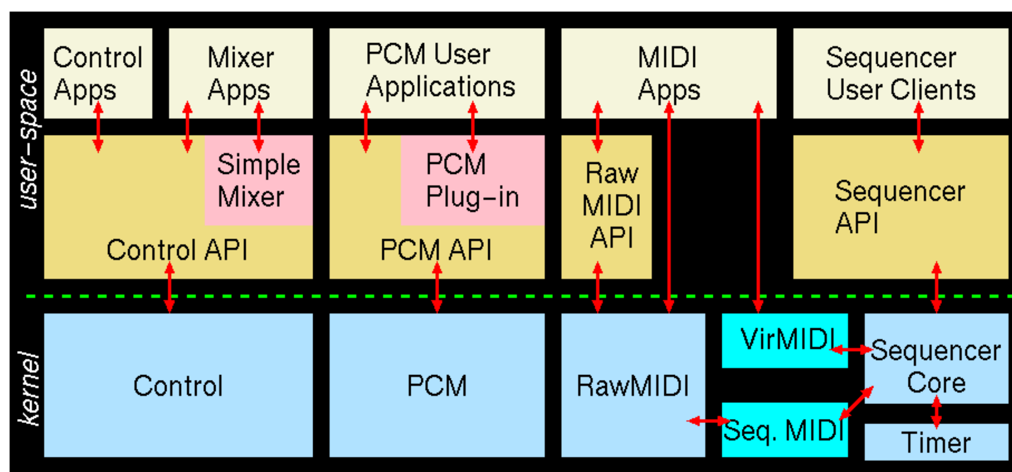
3.3.2 Architektura zvukového systému ALSA

Architektura zvukového systému ALSA poskytuje několik programových rozhraní (API).

- **Control Interface** – poskytuje správu či registraci zvukových zařízení a získání informací o těchto zařízeních
- **PCM Interface** – poskytuje funkce pro digitální přehrávání a nahrávání zvuku (pro vývojáře nativních ALSA aplikací je toto rozhraní většinou nejdůležitější)
- **Raw MIDI interface** – podpora pro MIDI (Musical Instrument Digital Interface), standart využívaný v hudebním průmyslu pro komunikaci zvukových zařízení v reálném čase, toto API poskytuje přímý přístup k MIDI událostem
- **Timer Interface** – poskytuje API pro přístup k časovači zvukového zařízení, které se používá k synchronizaci zvukových událostí

- **Sequencer interface** – API pro komplexnější programování MIDI aplikací než Raw MIDI interface
- **Mixer interface** – API, které se dá využít např. pro ovládání hlasitosti

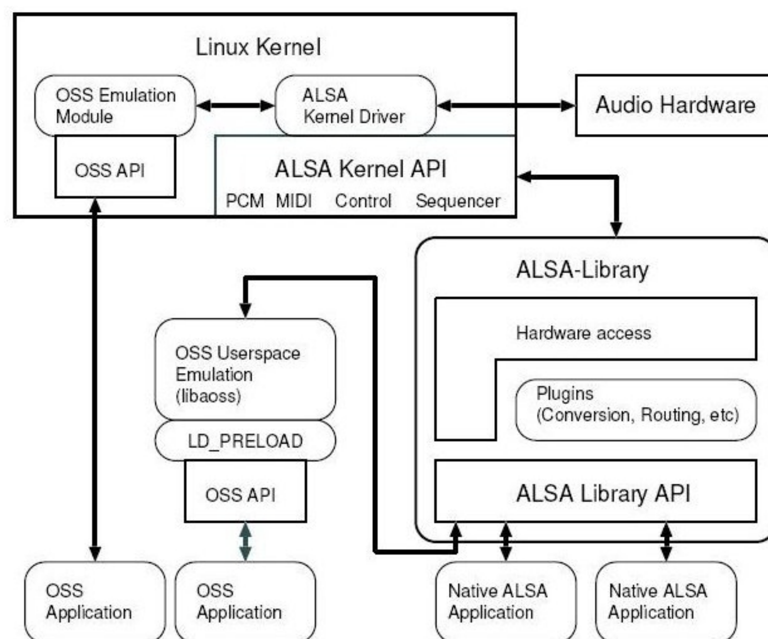
Dokumentace ke zmíněným rozhraním je zdroj [8]. Dokumentace je samozřejmě v anglickém jazyce a bohužel není příliš podrobná a rozhodně ani není úplná. Některé věci, dle mého názoru, se dají bohužel zjistit jenom pomocí studování zdrojových kódů hotových aplikací. To je samozřejmě velice zdlouhavé a krajně nepříjemné.



Obr. 7) Architektura zvukového systému ALSA

Na obrázku Obr. 7, který pochází ze zdroje [12], je znázorněna architektura zvukového systému ALSA. Zelenou čárkovanou čárou je oddělena oblast spadající do linuxového jádra od oblasti uživatelské. Prostřední část tvoří knihovny poskytující jednotlivé programové rozhraní.

Na obrázku Obr. 8, který pochází ze zdroje [13], je také znázorněna architektura ALSA a to včetně emulace OSS. Z obrázku lze také vypozorovat tok, jakým se ubírají požadavky od klientských aplikací až ke zvukovému HW.



Obr. 8) Architektura ALSA systému a tok požadavků

Z obrázku je patrné, že požadavek od nativní ALSA aplikace (aplikace napsaná pomocí ALSA API) putuje přes sdílené knihovny ALSA-Library do jádra OS Linux, kde skrze ovladače zvukového zařízení doputuje až k samotnému HW. Obdobná situace nastává při použití OSS aplikace, ovšem musí být použita emulace, a to buď na úrovni knihovny nebo kernelu.

Architektura zvukového systému ALSA je opravdu plně modulární. Veškeré funkcionality i ovladače jednotlivého zvukového HW jsou k dispozici ve formě modulů do linuxového jádra. Není tedy nutné automaticky zavádět všechny možné funkcionality a ovladače a tím zbytečně zvětšovat samotné jádro systému. Podpora nové funkcionality či nového zvukového HW lze jednoduše přidat zavedením příslušného modulu, a to zcela bez přerušení chodu zvukového systému (není potřeba žádný restart). To samé samozřejmě platí i v opačném případě.

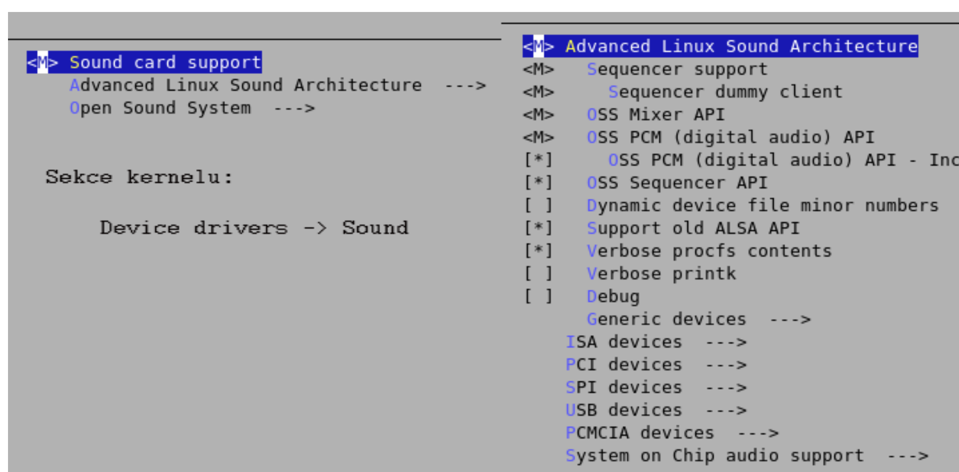
3.3.3 Instalace zvukového systému ALSA

Po instalaci jakékoli alespoň trochu uživatelsky přívětivé distribuce Linuxu (Mandriva, SuSe, KUbuntu, atd.) je zvukový systém ALSA plně nainstalován a připraven k použití. Jestli je ALSA připravena k použití lze zjistit např. ze souborového systému „/proc“, kde by měl být adresář „/proc/asound“, ze kterého lze získat spoustu informací o nainstalovaném zvukovém systému ALSA (podpora „/proc“ musí být zapnutá

v jádře OS). V případě, že se zvukový systém jeví jako nefunkční, ať už z jakéhokoli důvodu, je nutné provést instalaci ručně. V podstatě se jedná o několik jednoduchých kroků, které by se daly shrnout následovně:

- zapnutí podpory zvukového systému v kernelu, zapnutí podpory relevantního zvukového HW v kernelu,
- překlad kernelu, překlad modulů kernelu a nabootování do nového jádra,
- naistalování balíčku alsa-utils,
- spuštění skriptu alsaconf.

Typické nastavení jádra systému lze vypozorovat z Obr. 9 (zdroj vlastní). Všechny položky je lepší zapnout pouze ve formě modulu. Umožňuje to lepší manipulaci se zvukovým systémem a zbytečně to nezvětšuje samotné jádro OS. Ovladače zvukového HW se nacházejí v části „PCI devices” u běžného PC nebo v části „System on Chip support” u vestavěného zařízení. V případě nejistoty, jaký ovladač je potřeba, ničemu nevadí označit všechny. Jaká zvuková zařízení jsou podporována systémem ALSA lze zjistit ze zdroje [7]. Ovšem jedná-li se o vestavěné zařízení, o jehož ovladače se většinou stará jeho dodavatel, nejspíše ho v tomto soupise nenaleznete. Je nutné se obrátit na dodavatele vestavěného systému, jestli ovladač do zvukového systému ALSA pro své zařízení má k dispozici. Jak provést samotnou konfiguraci linuxového kernelu a jeho překlad např. zdroj [21].



Obr. 9) Nastavení ALSA v kernelu

Balíček alsa-utils obsahuje nezbytné systémové utility jako alsamixer (systémový směšovač), alsaplay (základní přehrávač), alsaconf (konfigurátor) atd. a je velice vhodné ho nainstalovat. Zvukový systém ALSA nabízí i jiné balíčky, které je možné z distribuce nainstalovat např. alsa-drivers. Tento balíček však není vhodné ve většině případů instalovat, ovladače jsou již zahrnuty v jádře OS.

Poslední krok instalace zvukového systému ALSA je spuštění skriptu alsaconf. Tento skript sám detekuje, jaké ovladače jsou pro váš zvukový HW potřeba a zjištěné informace uloží do konfiguračních souborů. Po spuštění zvukového systému jsou zavedeny do jádra pouze ovladače, které byly zvoleny. V případě, že tento skript hlásí, že nenalezl žádný zvukový HW, je to důsledek:

- nebyl v jádře vybrán příslušný ovladač,
- zařízení nelze detekovat.

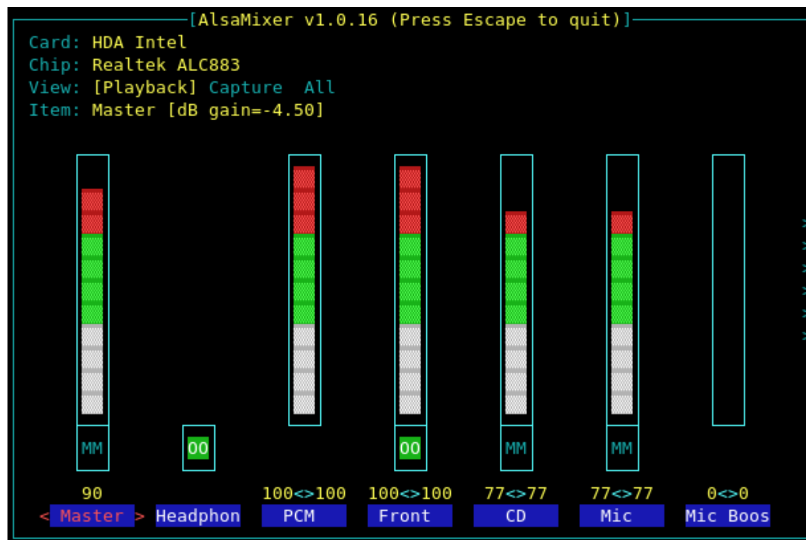
Pro nápravu této situace je potřeba přidat v kernelu podporu pro dané zařízení, respektive zkontrolovat, jestli je vůbec podporováno. V případě, že byl příslušný ovladač určitě vybrán, ale zvukový HW nebyl přesto detekován (stává se u vestavěných zařízení), je nutné pokusit se o zavedení ovladače ručně pomocí příkazu `modprobe`. Více informací o instalaci zvukového systému ALSA lze najít např. zde [22].

3.3.4 Ovládání směšovače

K nastavení hlasitosti, respektive směšovače, lze použít několik konzolových nástrojů systému ALSA. První z nich je příkaz `amixer`. Použití tohoto programu demonstruje následující ukázka.

```
amixer -q set 'Master' 80% unmute
```

Tímto příkazem se provede nastavení „Master“ na 80% a „Master“ nebude utlumený. Nastavování systémového mixéru tímto nástrojem není sice příliš komfortní, ale zase ho lze využít v jiné aplikaci či v nějakém skriptu. Další nástroj, který je již uživatelsky více přívětivý, je nástroj `alsamixer` (Obr. 10, zdroj vlastní). Ovládání směšovače tímto nástrojem je velice intuitivní. Detaily o zmíněných nástrojích lze najít v příslušných manuálových stránkách OS Linux (např. man alsamixer). Na ovládání mixéru lze samozřejmě použít i jiné nástroje, které nejsou součástí zvukového systému ALSA (např. KMix).



Obr. 10) Alsamixer

4. Programování zvukové aplikace pro systém ALSA

První podkapitola se věnuje popisu parametrů a termínů. Ostatní podkapitoly popisují určité části programů pro zvukový systém ALSA. Aby bylo možné si tyto části ucelit, přikládám jako přílohu C jednoduchý program, který po spuštění přehraje několik sekund náhodného zvuku. Program je vytvořen v programovacím jazyce C, pro ALSA aplikace je také možné využít jazyka C++. V této kapitole bylo čerpáno ze zdrojů [8] a [9]. Ve zdroji [8] lze najít mnoho ALSA API funkcí, které zde nejsou uvedeny.

4.1 Popis důležitých termínů a parametrů

Předtím, než je možné se pustit do programování zvukové aplikace v ALSA API, je nutné se seznámit s některými termíny a parametry, jejichž význam nemusí být zřejmý.

4.1.1 ALSA zařízení a základní parametry

Nejmenší skupina parametrů, která je potřeba nastavit nejdříve, by se dala označit jako skupina základní. Jedná se o:

- ***device*** – název zařízení, které je použito pro nahrávání či přehrávání, více v následujících odstavcích,
- ***stream*** – nastavuje, zda se bude nahrávat či přehrávat, (`SND_PCM_STREAM_CAPTURE, SND_PCM_STREAM_PLAYBACK`),
- ***mode*** – nastavuje, jestli budou volání ALSA API funkcí pro zápis či čtení bufferu zvukového systému blokující, většinou se uvádí 0, což má za následek právě blokující volání.

Výchozí zařízení zvukového systému ALSA se nazývá *default* a je k dispozici v každém systému ALSA. V případě, že nepotřebujeme provádět žádné úpravy zvukového streamu, lze toto zařízení bez problému použít pro nahrávání i přehrávání. Jak bylo zmíněno v kapitole 3.3.1 zvukový systém ALSA umožňuje logické pojmenování zařízení za účelem různých úprav datového streamu. Těmito úpravami je myšleno nejčastěji softwarové mixování více zvukových proudů do jednoho. Většina zvukového HW stále

nepodporuje hardwarové mixování a chceme-li ve stejný okamžik přehrávat více zvuků, je nutné využít mixování softwarové.

Na osobních počítačích je většinou SW mixování řešeno externími zvukovými servery, jako je aRts, Pulseaudio atd. Tyto zvukové servery nabízejí větší možnosti úpravy datového proudu, ale jejich použití je náročnější na systémové zdroje. Proto se nehodí právě pro vestavěná zařízení, kde je celkově výhodnější použít interního mixování zvukového systému ALSA. Pro IP-telefonii to znamená, že během hovoru je možné přehrát i další zvuky např. nějaké systémové varování. O SW mixování uvnitř ALSA se stará tzv. „plugin" s názvem DMIX. Dle mého názoru je nejvýhodnější předefinovat výchozí zařízení tak, aby při jeho použití automaticky docházelo k SW mixování skrze DMIX. Tohoto efektu lze docílit přidáním několika následujících řádků do konfiguračního souboru „/etc/asound.conf" nebo „~/.asoundrc".

```
pcm.!default {
        type plug
        slave.pcm "dmix"
    }
```

Více informací o pojmenovávání zvukových zařízení a vytváření zařízení vlastních za účelem různých konfigurací lze najít např. ve zdrojích [8], [23].

4.1.2 Hardwarové parametry a chyby xrun

Následuje výpis největší skupiny parametrů, které jsou v ALSA API označovány jako parametry hardwarové:

- ***sample*** – označuje jeden zvukový vzorek, důležitý je jeho formát, existuje jich celá řada, jedná se o datový typ `snd_pcm_format_t`, dle mého názoru nejčastěji používané jsou formáty `SND_PCM_FORMAT_U8` (bezznaménkový, jeden bajt) a `SND_PCM_FORMAT_S16_LE` (znaménkový, dva bajty, little-endian[2]),
- ***channels*** – nastavení počtu kanálů, to znamená např. (mono – 1 kanál, stereo – 2 kanály), samozřejmě počet kanálů musí být podporován HW, pro IP-telefonii dostačuje kanál jeden,

[2] Určuje pořadí bajtů. Na nejnižší adresu v paměti se ukládá nejméně významný bajt.

- ***rate*** – označuje vzorkovací frekvenci, v ALSA API je vždy v jednotkách Hz, pro IP-telefonii dostačuje hodnota 8000 Hz, což znamená 8000 vzorků za sekundu, při dvou kanálech a dvoubajtovém vzorku se tedy generuje datový tok 32 000 B/s,
- ***access*** – označuje, jestli se jedná o přístup prokládaný nebo neprokládaný, jestli je například zvoleno stereo a přístup prokládaný, data jsou ze nebo do vyrovnávací paměti zvukového systému transportována LPLPLP…, zatímco při neprokládaném LLL… PPP…. (L – označuje vzorek levého kanálu, P – pravého kanálu), častěji se volí způsob prokládaný, pro ALSA API je to hodnota `SND_PCM_ACCESS_RW_INTERLEAVED`,
- ***frame*** – označuje jednotku, která se skládá z jednotlivých vzorků, vždy pro každý kanál jeden vzorek, velikost framu je tedy logicky závislá na formátu vzorku a počtu zvolených kanálů, např. pro stereo a dvoubajtový formát vzorku, to je 4 bajty (2 bajty na vzorek krát 2 kanály),
- ***period*** – perioda, označuje jednu část vyrovnávací paměti zvukového systému (více v kap. 3.1), pro 8000 Hz to většinou bývá 170 framů, hodnota roste se vzorkovací frekvencí,
- ***period time*** – doba, za kterou se přehraje nebo nahraje jedna perioda v mikrosekundách, většinou to bývá okolo hodnoty 21333, po této době dojde k přerušení,
- ***buffer*** – označuje celou vyrovnávací paměť zvukového systému, je složená z period, většinou to bývá 8,
- ***buffer time*** – doba, za kterou se přehraje nebo nahraje celá vyrovnávací pamět v mikrosekundách, je to logicky „period time“ krát počet period.

Velikost periody a jejich počet jsou velice důležité parametry, které zásadně ovlivňují kvalitu přehrávání a nahrávání. Pro IP-telefonii ovlivňují také další velice důležitý parametr, a to zpoždění. Je-li např. buffer pro přehrávání téměř plný a zapíšeme-li do něj vzorek, který chceme přehrát, přehraje se až za „buffer time“. Nastaví-li se ovšem buffer příliš malý, tak abychom minimalizovali zpoždění, dochází často k chybám, které se souhrnně nazývají jako tzv. „xrun“. Jedná se o:

- ***overrun*** – neboli přetečení bufferu, k němu dochází při nahrávání, a to z důvodu, že uživatelská aplikace nestačila dostatečně rychle odebírat data z bufferu a zvukový systém je nemá kam ukládat,
- ***underrun*** – podtečení bufferu, k němu dochází při přehrávání, a to z důvodu toho, že uživatelská aplikace nestačila dostatečně rychle data do bufferu přidávat a zvukový systém neměl co přehrát.

Těmto chybám není možné zcela úplně předejít, občas k nim prostě dojde. Proto musí být náležitě ošetřeny. Základem je nastavit co nejmenší možnou dostačující velikost bufferu, která zaručí minimální nastávání podtečení či přetečení bufferu, ale stále zaručí malé zpoždění. Pro většinu systémů je možné použít buffer o velikosti čtyř period (každá perioda o velkosti 170 framů), což je při frekvenci vzorkování 8000 Hz dostatečně velký buffer i pro slabší vestavěná zařízení, přičemž tato velikost stále zaručuje velice slušné zpoždění okolo 100 ms.

4.1.3 Softwarové parametry

Poslední skupina parametrů je v ALSA API označena jako softwarové parametry. Ty na rozdíl od HW parametrů není nutné nastavovat, ale minimálně některé je nastavit vhodné. Dle mého názoru se jedná o tyto parametry:

- ***star_ threshold*** – nastavuje, jaké musí být minimální obsazení bufferu, aby se začalo přehrávat či nahrávat,
- ***avail_min*** – nastavuje, jaké musí být minimální obsazení bufferu, aby se přehrávalo, či kolik minimálně musí být v bufferu místa, aby se nahrávalo,
- ***silence_threshold, silence_size*** – nastavuje hranici, která když je překročena, dojde k naplnění části bufferu o velikosti silence_size tichem.

Správné nastavení těchto SW parametrů má pozitivní vliv na počet „xrun“ chyb.

4.2 Základní struktura programu pro zvukový systém ALSA

Poté, co byly v předchozí kapitole vysvětleny důležité parametry a další termíny, je možné se věnovat samotnému programování v ALSA API. Struktura nativního ALSA API programu by se dala rozdělit do následujících částí:

1. Otevření příslušného zařízení.
2. Nastavení hardwarových parametrů.
3. Případné nastavení softwarových parametrů.
4. Datové transfery mezi uživatelskou aplikací a bufferem ALSA zvukového systému (nahrávání nebo přehrávání).
5. Uzavření otevřeného ALSA zvukového zařízení.

Všechny ALSA API funkce jsou naštěstí velice logicky pojmenovány. Veškeré API funkce vždy začínají globální předponou snd a dále pak předponou modulu, do kterého patří. V případě programování běžné zvukové aplikace je to modul PCM (Pulse Code Modulation), a je to tedy předpona „snd_pcm“. Téměř všechny ALSA API funkce mají číselnou návratovou hodnotu. Je-li tato hodnota záporná, došlo v API funkci k nějaké chybě. Textový popis této chyby lze získat pomocí `snd_strerror()`. Tato API funkce přijímá jediný parametr, který je číselné označení chyby a vrací její textovou reprezentaci.

4.2.1 Otevření zařízení

Otevření zařízení se provede API funkcí `snd_pcm_open()`. Ta za své parametry požaduje ty parametry, které byly v předchozí kapitole (4.1.1) označeny jako základní. Jedná se tedy o název zařízení, které má být otevřeno, jestli se má jednat o nahrávání či přehrávání a zda má být zápis či čtení do ALSA bufferu blokující. Tato API funkce na oplátku poskytuje manipulátor (handle) datového typu `snd_pcm_t`, který je nutný pro téměř všechny ostatní API funkce. Volání pro otevření výchozího zařízení pro přehrávání v neblokujícím módu může vypadat následovně.

```
snd_pcm_open(&pcm_handle, default, SND_PCM_STREAM_PLAYBACK, 0);
```

4.2.2 Nastavení HW parametrů

Nastavení parametrů, které byly popsány v kapitole 4.1.2, je rozděleno do několika kroků:

1. Alokování struktury pro hardwarové parametry.
2. Naplnění struktury defaultními hodnotami.
3. Samotná konfigurace parametrů.
4. Nastavení nakonfigurovaných parametrů do již otevřeného zařízení.
5. Dealokace struktury pro hardwarové parametry.

O alokování struktury pro HW parametry se stará ALSA API funkce `snd_pcm_hw_params_malloc()`. Naplnění struktury defaultními parametry zařídí API funkce `snd_pcm_hw_params_any()`. K zavolání této funkce již potřebujeme na alokovanou strukturu pro HW parametry a otevřené zařízení. Po provedení konfigurace HW parametrů nastavíme parametry do otevřeného zařízení ALSA API funkcí `snd_pcm_hw_params()`. Zvláště u této funkce je nutné kontrolovat, zda proběhla v pořádku. Nakonec se alokovaná struktura, která již není potřeba, dealokuje pomocí volání `snd_pcm_hw_params_free()`. Sekvence těchto API funkcí typicky vypadá následovně.

```
snd_pcm_hw_params_t *hw_params;
snd_pcm_hw_params_malloc (&hw_params);
snd_pcm_hw_params_any (pcm_handle, hw_params);
/* SAMOTNA KONFIGURACE HW PARAMETRU*/
err = snd_pcm_hw_params (pcm_handle, hw_params);
snd_pcm_hw_params_free (hw_params);
if (err < 0){
	/*REAKCE NA CHYBU*/
}
```

Všechny ALSA API funkce sloužící na konfiguraci HW parametrů požadují jako vstup manipulátor (handle) otevřeného zařízení, samotnou strukturu HW parametrů a hodnotu, která má být nastavena. Hodnota, která se nastavuje, se většinou předává přes ukazatel. Díky tomu je na adresu této hodnoty API funkcí nastavena skutečná hodnota, která byla nastavena. Požadovaná a nastavená hodnota nemusí být shodné. Některé z funkcí navíc umožňují vložit vstupní parametr, do kterého je uloženo, jestli je uložená hodnota od API funkce o něco menší, přesně taková, nebo o něco větší než reálná. Zadávat tento parametr není ovšem nutné, stačí místo něho zadat hodnotu 0 (na některých

systémech je to dokonce nutnost, s jinou hodnotou než 0 nelze program přeložit). Pro konfiguraci většiny HW parametrů lze použít více než jednu API funkci. Nejpoužívanější jsou však ALSA API funkce následující:

- nastavení přístupu (access) prokládaného či neprokládaného se provede funkcí `snd_pcm_hw_params_set_access()`, která přijímá hodnoty `SND_PCM_ACCESS_RW_INTERLEAVED` (prokládaný přístup, běžnější), `SND_PCM_ACCESS_RW_NONINTERLEAVED` (neprokládaný přístup),

- nastavení vzorkovací frekvence (rate) se provede ALSA API funkcí `snd_pcm_hw_params_set_rate_near()`, funkce se pokusí nastavit vzorkovací frekvenci co nejblíže to je možné frekvenci požadované,

- nastavení formátu vzorku (sample formát) se provede funkcí `snd_pcm_hw_params_set_format()`, zde přichází v úvahu spoustu hodnot, doporučuji `SND_PCM_FORMAT_S16_LE`, celkový soupis lze získat ve zdroji [8],

- nastavení počtu zvukových kanálů (channels) se provede funkcí `snd_pcm_hw_params_set_channels()`,

- nastavení velikosti periody, respektive jedné části bufferu lze provést funkcí `snd_pcm_hw_params_set_period_size_near()`, velikost se nastavuje v jednotkách framů, nikoli bajtů,

- nastavení velikosti bufferu, respektive počtu period, ze kterých se skládá, umožňuje `snd_pcm_hw_params_set_buffer_size_near()`, opět se nastavuje v jednotkách framů, nikoli bajtů, typicky je to 8krát velikost periody.

Jak už napovídá název této skupiny parametrů (HW parametry), není možné nastavit jakékoli hodnoty. Záleží především na možnostech daného HW. HW parametry lze nastavovat pouze v určitém rozsahu. Tento rozsah je definován ve struktuře typu `snd_pcm_hardware`, která musí být součástí každého ovladače pro zvukový systém ALSA. Na většině systémů je tento rozsah naprosto dostačující a není potřeba ho nějak zkoumat či upravovat. Ovšem na některých vestavěných systémech s nižším výkonem

nemusí být zvláště pro IP-telefonii vhodný. Například je možné nastavit pouze hodnoty, které mají za následek velkou velikost bufferu, čímž dochází k velkému zpoždění případného hovoru. Pak je řešením pouze upravování tohoto rozsahu přímo v ovladači. Dle mého názoru, který vychází z praktické zkušenosti, toto experimentování nebude mít ve většině případů význam. Vývojáři nízko úrovňových ovladačů znají většinou velice dobře možnosti daného HW.

V přehledu API funkcí pro konfiguraci HW parametrů nebyly záměrně uvedeny funkce pro nastavování časové délky periody či celého bufferu. Většinou je jejich použití zbytečné. Délka periody je na většině systémů udržována na poměrně výhodné hodnotě (i pro VoIP) okolo 21 milisekund. Tomu je podřízena i určitá logika. S rostoucí vzorkovací frekvencí (roste datový tok) automaticky roste i velikost periody tak, aby její přehrání či nahrání trvalo pořád stejnou dobu a to již zmíněných 21 milisekund.

Pro závěrečné shrnutí uvedu, že bezproblémově lze měnit: přístup, formát vzorku, počet kanálů a vzorkovací frekvenci. Se zbylými parametry je situace, jak bylo nastíněno, složitější. Před uvolněním struktury pro nastavení parametrů funkcí `snd_pcm_hw_params_free()` je vhodné nastavené HW parametry zkontrolovat. Nastavené hodnoty parametrů se mohly v rámci uvedené logiky změnit (především velikost periody, velikost bufferu) po zavolání API funkce `snd_pcm_hw_params()`. Zjištění nastavení jednotlivých parametrů je možné pomocí API funkcí, které mají v názvu místo `set` logicky `get`. Např. pro zjištění délky periody, velikosti periody, délky bufferu, velikosti bufferu jsou to ALSA API následující:

```
snd_pcm_hw_params_get_period_time(),
snd_pcm_hw_params_get_period_size(),
snd_pcm_hw_params_get_buffer_time(),
snd_pcm_hw_params_get_buffer_size().
```

4.2.1 Nastavení SW parametrů

Nastavení SW parametrů se provádí podobným způsobem jako nastavení parametrů HW. Nejdříve se musí alokovat potřebná struktura. Následně se naplní hodnotami, které by se daly považovat za výchozí. Dále se provádí samotná konfigurace SW parametrů. A nakonec dojde k jejich uložení do otevřeného zařízení s následnou dealokací struktury SW parametrů. Postup je zřejmý na následujícím kódu.

```
snd_pcm_sw_params_t *sw_params;
snd_pcm_sw_params_malloc (&sw_params);
snd_pcm_sw_params_current (pcm_handle, sw_params);
/*SAMOTNE NASTAVENI SW PARAMETRU*/
snd_pcm_sw_params(pcm_handle, sw_params);
snd_pcm_sw_params_free (sw_params);
```

SW parametry nejsou sice tak těsně svázány s daným HW, ale jejich nastavení není také úplně snadné. Často může jejich špatnou konfigurací dojít k chybě. Proto bude u každé API funkce uvedena experimentálně ověřená hodnota. Pro nastavení základních SW parametrů, které byly představeny v kapitole 4.1.3, lze použít např. funkce:

- pro nastavení hodnoty „treshold" API funkci `snd_pcm_sw_params_set_start_threshold()`, jako vhodné nastavení se zdá být velikost bufferu mínus velikost jedné periody,
- pro nastavení hodnoty „avail_min" `snd_pcm_sw_params_set_avail_min()`, vhodné nastavení je dle mého názoru hodnota okolo půl periody,
- pro nastavení hodnoty „silence_treshold" `snd_pcm_sw_params_set_silence_threshold()`, tato hodnota by měla být o něco větší než hodnota „ avail_min",
- pro nastavení hodnoty „silence_size", která je svázána s hodnotou „silence_treshold" , `snd_pcm_sw_params_set_silence_size()`, vhodná se zdá být hodnota okolo půl periody.

Po zavolání těchto konfiguračních funkcí doporučuji kontrolovat, jestli nedošlo k chybě. Nastavené hodnoty lze také získat příslušnou „get" funkcí podobně jako u HW parametrů.

4.2.2 Zápis a čtení ALSA bufferu

Přehrávat a nahrávat, respektive provádět datové transfery s ALSA bufferem, lze několika způsoby. První, který je nejjednodušší a nejvíce používaný, spočívá v pouhém zavolání funkce, která nahraje či přehraje požadovaný počet framů. Další způsob spočívá v registrování tzv. „call back" funkce, která je zavolána při přerušení. Tato registrovaná funkce se musí postarat o datové transfery mezi aplikací a ALSA bufferem. Ukáz-

ku tohoto principu přikládám jako přílohu D. Oba tyto přístupy mají ovšem jednu nevýhodu. Dochází při nich ke zbytečnému kopírování bloků paměti. Lze uplatnit i přístup, kdy je do ALSA bufferu přistupováno přímo skrze namapovanou paměť pomocí mmap (systémové volání OS Linux). Tento přístup ovšem není zcela triviální a dle mých zkušeností nějaké větší zrychlení nepřinese. Tento přístup lze nastudovat z příkladů zdrojových kódů, které jsou uvedeny ve zdroji [8].

Dále bude předveden pouze přístup nejběžnější a bude předpokládáno použití prokládaného přístupu a blokovaného módu. Tento přístup spočívá v zavolání funkce `snd_pcm_writei()`, která do bufferu zapíše (přehraje) požadovaný počet framů. Nebo zavolání funkce `snd_pcm_readi()`, která z bufferu načte (nahraje) požadovaný počet framů. Obvyklý počet framů, který se nahrává či přehrává je právě jedna perioda. Při těchto transferech je nutné detekovat a ošetřovat chyby typu „xrun“ (vysvětleno v kapitole 4.1.2). Na tyto chyby je nutné reagovat voláním ALSA API funkce `snd_pcm_prepare()`. Po nastání chyby typu „xrun“ je zastaveno přehrávání i nahrávání do doby, než se `snd_pcm_prepare()` zavolá. Samotné chvilkové přerušení přehrávání či nahrávání při „xrun“ chybě ve většině případů způsobí nepříjemný efekt, který je slyšet ve formě prasknutí. Proto je nutné se těmto chybám vyhnout vhodným nastavením HW parametrů, SW parametrů a v neposlední řadě dobře zvolenou konstrukcí programu. Zvláště u aplikace zabývající se IP-telefonií je potřeba reagovat na situaci, kdy nejsou včas k dispozici data, která by mohla být přehrána. Následující kód ukazuje, jakým způsobem je např. možné do ALSA bufferu nahrávat data (přehrávat).

```
int returnCode = snd_pcm_writei(m_handle, buffer, frames);
//nastal underrun
if (returnCode == -EPIPE)
{
      if (snd_pcm_prepare(m_handle) != 0 ) //prepare selhalo, reakce
      //reakce, napr. zalogovani ze doslo k teto chybe
}
//jina chyba
else if (returnCode < 0)
{
      if (snd_pcm_prepare(m_handle) != 0 ) //prepare selhalo, reakce
      //reakce, napr. zalogovani ze doslo k teto chybe
}
//kratky zapis nebo ok
```

K jiným chybám než „xrun“ dle mých zkušeností nedochází, ale je potřeba je ošetřit. Dle dokumentace by mělo ve většině případů dostačovat zavolání `snd_pcm_prepare()`. Funkce pro transfer vrací zápornou hodnotu v případě nějaké

chyby a kladnou jako počet nahraných či přehranných framů. V blokujícím módu sice funkce pro transfer volající vlákno blokuje, dokud se nepodaří přenést požadovaný počet framů, ale občas se stává, že se to najednou nepovede. Proto je vhodné kontrolovat, jestli se počet přenesených framů shoduje s požadovaným, a v případě, že tomu tak není, transfer dokončit. Více lze nastudovat ze zdrojů [8] a [9].

4.2.3 Uzavření zařízení

Uzavření zařízení se provede API funkcí `snd_pcm_close()`. Po tomto volání jsou uvolněny veškeré systémové zdroje a aplikace přestane být vázána se zvukovým systémem. V případě, že `snd_pcm_close()` není řádně zavoláno, zařízení již nemusí jít opět otevřít.

Před samotným ukončením by měla být zavolána funkce `snd_pcm_drain()`, která zařídí dokončení transferů dat, které se nacházejí v ALSA bufferu. Nebo funkce `snd_pcm_drop()`, která data v bufferu zahodí.

5. Kompilace pro vestavěný systém a jeho testování zhlediska VoIP

5.1 Kompilace pro jinou platformu

Při vývoji SW pro vestavěná zařízení je prakticky nezbytné kompilovat vyvíjenou aplikaci na běžném PC. To je téměř vždy odlišné architektury než vestavěné zařízení. Vývoj aplikace na PC přinese velké zrychlení a daleko větší efektivitu práce, než kdyby se aplikace vyvíjela přímo na vestavěném zařízení. Osobní počítače jsou z pravidla mnohem výkonnější a samotný vývoj aplikace je mnohem komfortnější (použití IDE, možnost krokování, sledování proměnných atd.). V této podkapitole bylo použito zdroje [29].

Při tzv. „cross-kompilaci“, tedy kompilaci pro jinou platformu, vznikne binární kód, který je možný spustit na jiné architektuře procesoru, než na které byl vyvíjen. Tento proces je nutný, protože aplikaci funkční na běžném PC, které je většinou architektury x86 nebo x86_64, není možné spustit např. na nějakém mobilním telefonu, který je většinou architektury Arm[3]. Spuštění aplikace z PC na mobilním telefonu by selhalo jednoduše z důvodu toho, že procesor nerozumí binárnímu kódu určenému pro PC.

Při „cross-kompilaci“ se často mluví o termínech „Host platform“ a „Target platform“. První označuje platformu, na které se kompiluje. Druhý určuje cílovou platformu, pro kterou se kompiluje. Proto, aby bylo možné kompilovat SW pro jinou platformu, je nutné mít k dispozici několik tzv. „toolchains“ pro cílovou platformu (kompilátor, asembler, linker, knihovny pro daný jazyk). Tyto programové balíky obsahují vše, co je potřeba. V OS Linux se jedná především o programové balíky: binutils, gcc, glibc.

V některých linuxových distribucích[4] bývá k dispozici SW, který umožňuje jednotnou instalaci a správu všech potřebných nástrojů pro „cross-kompilaci“. Např. pro Gentoo Linux je to aplikace jménem „crossdev“. Pro architekturu mipsel je pomocí crossdev možné vytvořit kompletní toolchains následujícím příkazem.

```
USE="nptl" crossdev -b -v -t mipsel --b 2.17 --g 4.1.2 --l 2.5
```

[3] Neplatí pro technologie .Net a Java.

[4] Dobrá podpora pro různé architektury je v distribucích Gentoo a Debian.

Výsledkem pak může být např. kompilátor `mipsel-unknown-linux-gnu-gcc`. Při využití výsledného kompilátoru bývá často nutné použít různé volby, aby např. fungovaly správně operace s plovoucí desetinnou čárkou. Tyto nastavení lze najít v manuálu od příslušného „gcc“. Ovšem instalace všech potřebných nástrojů není vždy bezproblémová. Z osobní zkušenosti mohu potvrdit, že často dochází k chybám, které lze vyřešit jen s velkými obtížemi. Obvykle je mnohem rychlejší překopírovat příslušné soubory a adresáře ručně, jsou-li jinde k dispozici. Následuje výpis klíčových adresářů, které obsahují symbolické odkazy či soubory, které je nutné při ruční instalaci překopírovat (pro distribuci Gentoo Linux, u jiných linuxových distribucí se může lišit): „/usr“, „/usr/bin“, „/usr/i686-pc-linux-gnu“, „/usr/lib/binutils“, „/usr/lib/gcc“, „/usr/lib/gcc-lib“, „/usr/libexec/gcc“, „/usr/share/binutils-data“, „/usr/share/gcc-data“. Další informace o „cross-kompilaci“ lze najít např. ve zdroji [28].

5.2 Testování vestavěného zařízení pro VoIP

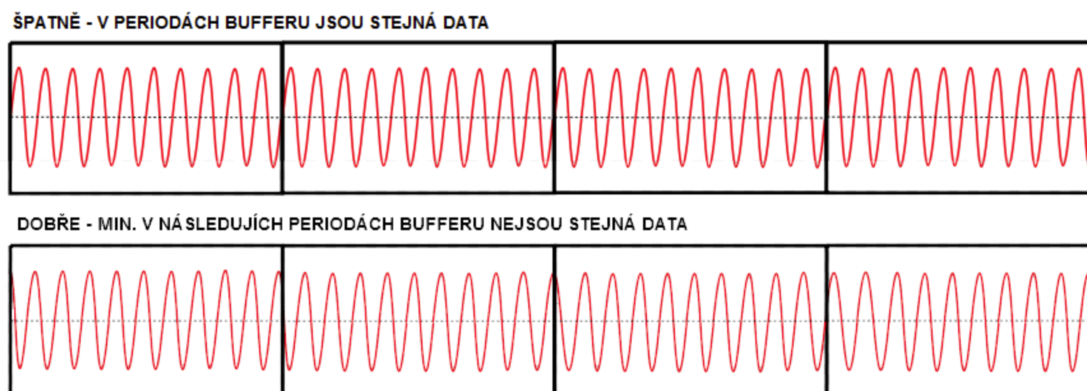
Následující dvě podkapitoly obsahují návod, jakým lze otestovat, zda je příslušné zařízení dostatečně výkonné na provozování IP-telefonie v OS Linux a jaká je spolehlivost zvukového systému ALSA. Uvedené testy a postupy byly prakticky ověřeny, když byly řešeny problémy z obou zmíněných kategorií. Včasné provedení následujících testů může mít za následek ušetření spousty času a starostí.

5.2.1 Testování dostatečného výkonu

Jestli je daný systém dostatečně výkonný se dobře testuje přehráváním nějakého předem definovaného periodicky se opakujícího signálu např. sinusoidy. V takovém to signálu jsou velice dobře slyšet jakékoli zvukové defekty. Vhodné je také na osciloskopu kontrolovat, jestli přehrávaný signál odpovídá generovanému.

V případě, že přehrávání probíhá v pořádku při celkové délce ALSA bufferu max. 250 ms, zařízení je dle mého názoru schopné provozovat IP-telefonii. Při bezchybném přehrávání až při delším bufferu než je 250 ms, je zařízení pro VoIP nevhodné kvůli velkému zpoždění, které by při hovoru vznikalo. Tímto způsobem lze najít minimální možnou délku bufferu, při které je přehrávání bezchybné. Toto nastavení lze pak následně uplatňovat v aplikaci provozující VoIP, kde se délka ALSA bufferu podílí velkou mírou na celkovém zpoždění hovoru.

Přehrávané hodnoty musí být generovány tak, aby na sebe jednotlivé částí bufferu[5] navazovaly a zároveň následující část bufferu neobsahovala stejná data jako část předchozí. Tyto požadavky zobrazuje následující obrázek Obr. 11 (zdroj vlastní).

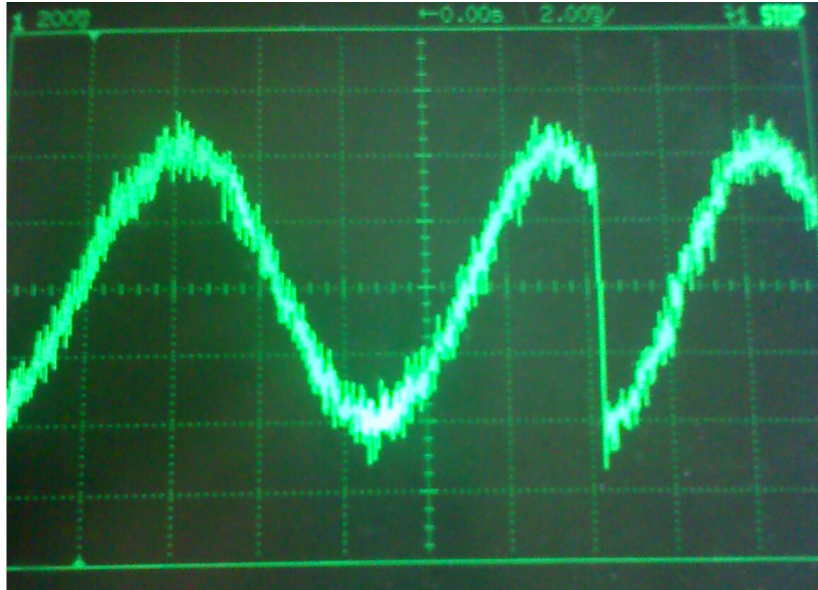


Obr. 11) Generování testovacích dat

V případě přehrávání stejných period dat, tak jak je vidět na vrchní části Obr. 11, se může přehrávání jevit jako bezchybné při jakékoli celkové délce bufferu. Data již jednou zapsaná do ALSA bufferu tam zůstávají, dokud nejsou přepsány novými. Při přehrávání stejných dat nemusí být výpadek zřetelný, protože tam již odpovídající data mohla být zapsána předtím. Jako přílohu E přikládám zdrojový kód jednoduchého programu, který byl pro testování výkonu již několikrát použit. Program přehrává sinusoidu, lze nastavit délku bufferu a vzorkovací frekvenci.

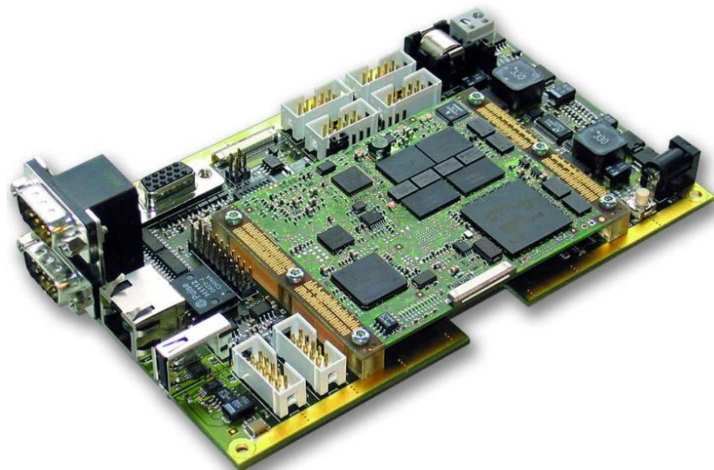
Záměrem společnosti RADOM, s.r.o. bylo provozovat IP-telefonii na vestavěném systému EXM32 s CPU modulem SH7760 (200 MHz, 64 MB SDRAM, více [30]). CPU modul SH7760, který není žádným způsobem optimalizovaný na podobnou činnost se nakonec ukázal jako nedostatečně výkonný. Nejmenší buffer, který byl s tímto modulem možné nastavit, měl 4 části (periody) po cca 80 ms. I při této délce bufferu, která není pro VoIP vhodná, nebylo přehrávání bezchybné. Při přehrávání docházelo k pravidelným výpadkům, které se při testu projevovaly dobře slyšitelným praskáním. Jakým způsobem vypadal jeden z výpadků je vidět na následujícím obrázku Obr. 12 (zdroj vlastní), který byl zachycen pomocí osciloskopu.

[5] Buffer pro přehrávání je rozdělen na několik stejných částí, mezi kterými je periodicky přepínáno (více např. kap. 3.1)



Obr. 12) Přehrávání SH7760

Z obrázku je patrné, že se nestihla přehrát celá část (perioda) bufferu. To mělo za následek přerušení pravidelnosti sinusoidy a nechtěné praskání. Bylo nutné upustit od původního záměru použití tohoto CPU modulu a pro IP-telefonii zvolit modul výkonnější. Z možné nabídky pro vestavěný systém EXM32 byl vybrán CPU modul s označením AU1250 (500 MHz, 128 MB SDRAM, více [30]). S tímto modulem přehrávání probíhalo bezproblémově s délkou bufferu cca 84 ms. Tato délka bufferu je běžná i u PC. Společnost RADOM, s.r.o. se nadále rozhodla pro IP-telefonii použít CPU modul AU1250. Následující obrázek Obr. 13 zobrazuje pro lepší představu vestavěný systém EXM32 s CPU modulem SH7760 (zdroj [34]).



Obr. 13) EXM32, SH7760 starterkit

5.2.2 Testování spolehlivosti zvukového systému

Dostatečný výkon vestavěného zařízení není jediným aspektem pro jeho použití. Velice důležitá je samozřejmě spolehlivost daného HW a SW. Z hlediska spolehlivosti je vhodné otestovat, zda je zvukový systém schopen několikahodinového přehrávání a naopak schopen přehrávání krátkého, ale neustále opakovaného.

Oba typy přehrávání lze důkladně otestovat pomocí jednoduchých testů. Pro dlouhodobé přehrávání stačí např. přehrávat nějaký náhodný zvuk po několik hodin (doporučuji alespoň 12 hod.) při občasné kontrole, zda se opravdu přehrává.

```
aplay -r 8000 -c 1 -f S16_LE /dev/urandom
```

Tento příkaz zajistí nekonečné přehrávání náhodného zvuku. Přehrávat se bude s nastavením 8000 Hz, mono, dvoubajtový vzorek. K otestování krátkého opakovaného přehrávání může posloužit následující jednoduchý skript.

```
#!/bin/sh

for ((  i = 0 ;  1;  i++  ))
do
      echo "Round $i"
      aplay short_test_sound
      sleep 1
done
```

Tento skript donekonečna přehrává např. 2 sec dlouhý zvukový soubor s názvem „short_test_sound" a mezi opakováním dělá 1 sekundovou pauzu. Jako dostatečný test spolehlivosti považuji alespoň 10000 úspěšných opakování. Jako u předchozího testu je nutné občas kontrolovat, zda se opravdu přehrává.

Jestli vše vypadá v pořádku, ale nic se nepřehrává či nenahrává, doporučuji zkontrolovat správné nastavení směšovače (viz kap. 3.3.4). V případě jiných problémů mohu poradit pouze experimentovat či problém konzultovat s dodavatelem vestavěného systému nebo jiným odborníkem na danou problematiku. U vestavěného systému EXM32 s CPU modulem AU1250 se objevil problém s opakovaným přehráváním. Průměrně asi při dvaceti opakování přehrávání zamrzlo a pro další korektní funkčnost musel být zvukový systém restartován. To byl fatální problém, který zásadně omezoval použitelnost systému. Prakticky to znamenalo, že každé cca dvacáté spuštění zvukové aplikace selže a následně musí být proveden restart celé služby. Po konzultacích s vedoucím práce se podařilo problém nakonec nečekaně jednoduše vyřešit. Bylo zjištěno,

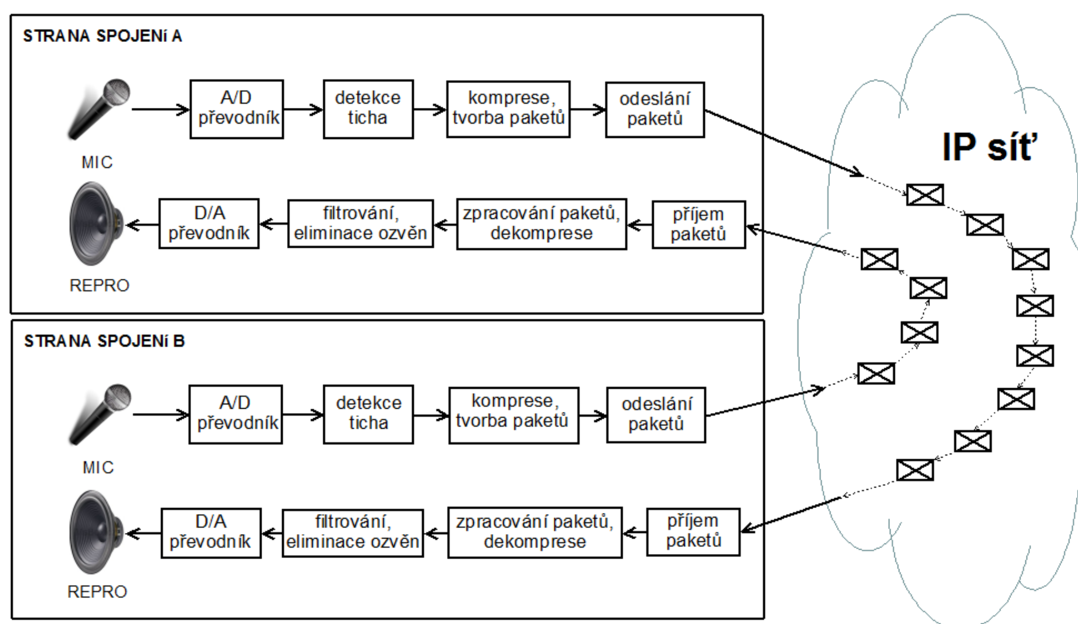
že při nekonečném přehrávání ticha na pozadí k zamrzávání zvukového systému nedochází. Přehrávání ticha lze docílit spuštěním jednoduchého příkazu po naběhnutí systému, ale průměrně si vyžádá zhruba 2 % výkonu CPU modulu AU1250. Samotný příkaz může vypadat např. následovně.

```
aplay -r 8000 -c 1 -f S16_LE /dev/zero &> /dev/null &
```

6. Návrh aplikace pro VoIP

6.1 Obecný princip IP-telefonie

Princip fungování IP-telefonie, respektive informační tok procesů, které při IP-telefonii vznikají, by mohl vypadat tak, jak je zobrazeno na Obr. 14 (zdroj vlastní).



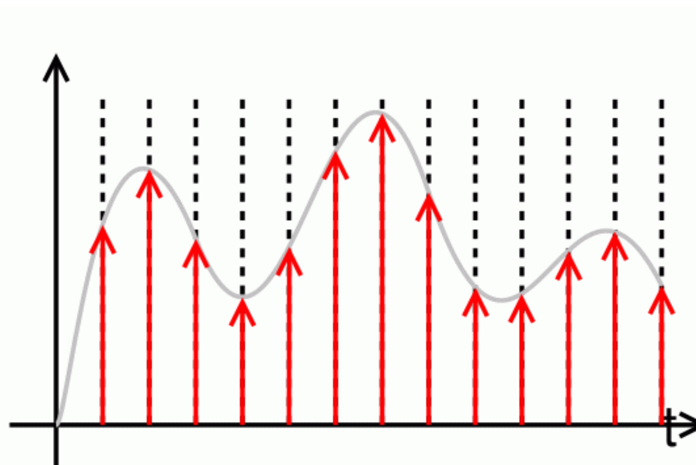
Obr. 14) Informační tok procesů při IP-telefonii

Analogový signál z mikrofonu je převáděn na digitální pomoci zařízení, které se nazývá A/D převodník. Jak se postupuje při tomto převodu bylo nastíněno v kapitole o digitalizaci (2.2). Může následovat detekce ticha, aby nedocházelo ke zbytečnému zpracovávání a přenosu nepodstatných dat. Samotná detekce ticha je založena na zkoumání hodnot malého úseku dat o délce např. 20 milisekund. V případě, že byl úsek vyhodnocen jako ticho, je pozdržen, aby bylo ticho ověřeno i v následujícím úseku. Bylo-li ticho detekováno i ve druhém úseku, je první úsek zahozen. Nebylo-li ticho detekováno, jsou úseky normálně zpracovány[6]. Dále jsou úseky zkomprimovány pomocí nějakého vhodného kodeku pro řeč a odeslány na druhou stranu spojení. Samotné odeslání se uskuteční nejlépe nějakým protokolem postaveným nad UDP, který je navržen pro komunikaci v reálném čase.

[6] Více o detekci ticha lze najít např. ve zdroji [2].

Druhá strana odeslaný paket přijme, zpracuje ho a provede dekomprimaci. Vhodný protokol komunikace zajistí, že nebudou zpracovávány pakety, které nepřišly včas, aby nedocházelo k nárůstu zpoždění. Může následovat aplikace digitálních filtrů a případná eliminace vlastních ozvěn. Digitální eliminace vlastních ozvěn není ovšem triviální záležitostí. Proto také často bývá řešena pomocí speciálního obvodu. Zajímavý projekt zabývající se eliminací ozvěn, kde lze o tomto problému nalézt i další informace, je zdroj [25].

Nakonec je z digitálních diskrétních hodnot rekonstruován spojitý analogový signál, který přehrají reproduktory. Zařízení zabývajícím se opačným procesem, než je digitalizace, se nazývá D/A převodník. Princip rekonstrukce analogového signálu zobrazuje obrázek Obr. 15, který pochází ze zdroje [24].



Obr. 15) Rekonstrukce analogového signálu

Červené šipky naznačují diskrétní digitální hodnoty, které jsou pomocí vhodných metod převedeny na spojitý analogový signál, jenž diskrétní hodnoty interpoluje.

Mezi oběma komunikujícími stranami bývá většinou vedené ještě jedno spojení, které není na obrázku Obr. 14 zobrazeno. Toto spojení bývá využíváno k řízení relace a sledování kvality datového toku.

6.2 Architektura aplikace

6.2.1 Shrnutí požadavků

Ze zadání práce vyplývá, že navržená aplikace má umožňovat plně duplexní hovor ve srovnatelné kvalitě, jako má hovor přes mobilní telefon. Aplikace tedy musí nutně provozovat následující činnosti:

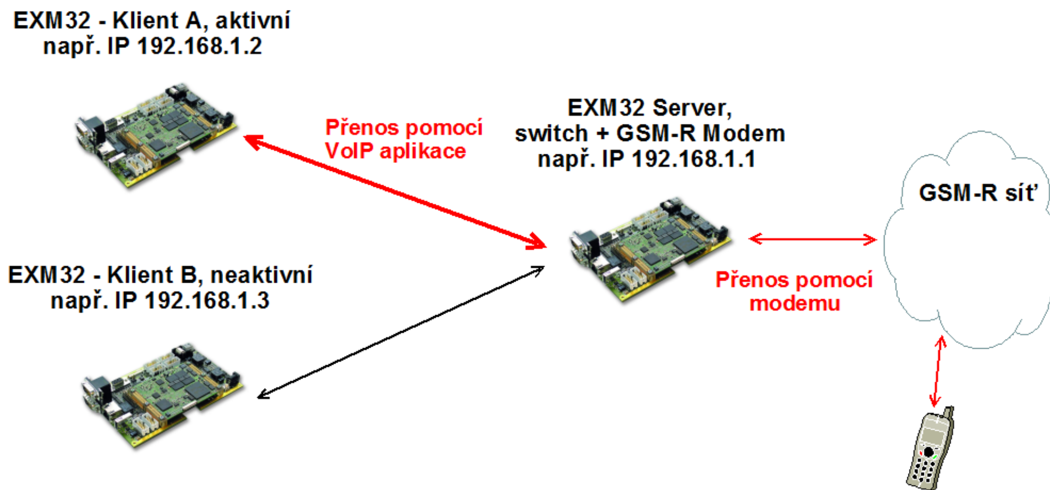
- příjem zvukových dat z IP sítě,
- přehrávání přijatých zvukových dat,
- nahrávání zvukových dat,
- odesílání nahraných zvukových dat do IP sítě.

Minimálně činnosti přehrávání a nahrávání zvukových dat musí probíhat současně. Z tohoto vyplývá, že navržená aplikace bude muset být více vláknová nebo bude nutné použít více procesů. Vzhledem ke snadnějšímu sdílení paměti je více vláknová aplikace vhodnější volbou.

Primární využití aplikace spočívá v přenosu zvuku v malé síti typu Ethernet, která je navíc převážně málo vytížená. Aplikace není přímo určená pro komunikace skrze velké sítě, jako je Internet. Tato podstatná informace bohužel přímo nevyplývá ze zadání práce, ale byla upřesněna zadavatelem práce společností RADOM, s.r.o. Díky této skutečnosti není nutné implementovat či používat poměrně složité protokoly sloužící pro přenos multimediálních dat IP sítí v reálném čase (viz kap. 2.3.3). Pro přenos zvukových dat bude plně dostačovat samotný protokol UDP (viz kap. 2.3.4), což přináší zjednodušení.

Primární využití VoIP aplikace zobrazuje následující obrázek Obr. 16 (zdroj vlastní). O přenos zvuku od klienta na server se stará VoIP aplikace, o přenos zvuku dále se stará modem.

Pro další zjednodušení nebude v architektuře ani v implementaci aplikace počítáno s částí, která by se zabývala eliminací ozvěn. Eliminace ozvěn nebude součástí výsledné aplikace a bude řešena jinou cestou.

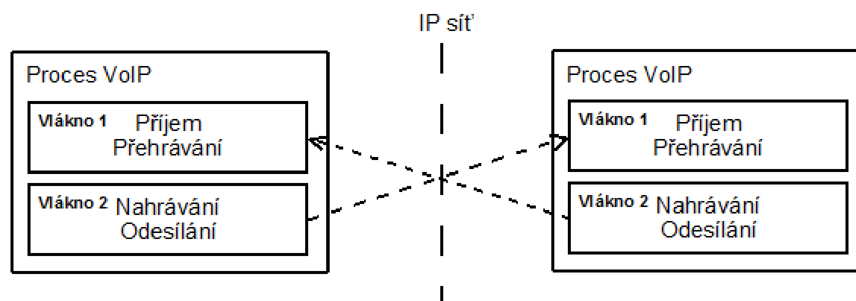


Obr. 16) Primární využití VoIP aplikace

Přestože zadavatel práce upřesnil, že k přenosu zvuku bude využito konzolové aplikace, je vhodné ukrýt logiku VoIP do knihovny, nad kterou může být velice jednoduše postavena jak grafická tak konzolová aplikace.

6.2.2 Reprezentace v paměti a vlákna

Počet vláken aplikace je otázkou. V určení počtu vláken se nabízí více možností. Z funkčního hlediska je možné v jednom vlákně zvuková data přijímat a následně je přehrávat. Zatímco ve druhém vlákně je možné naopak nahrávat a následně odesílat. Tuto situaci zobrazuje následující obrázek Obr. 17 (zdroj vlastní).

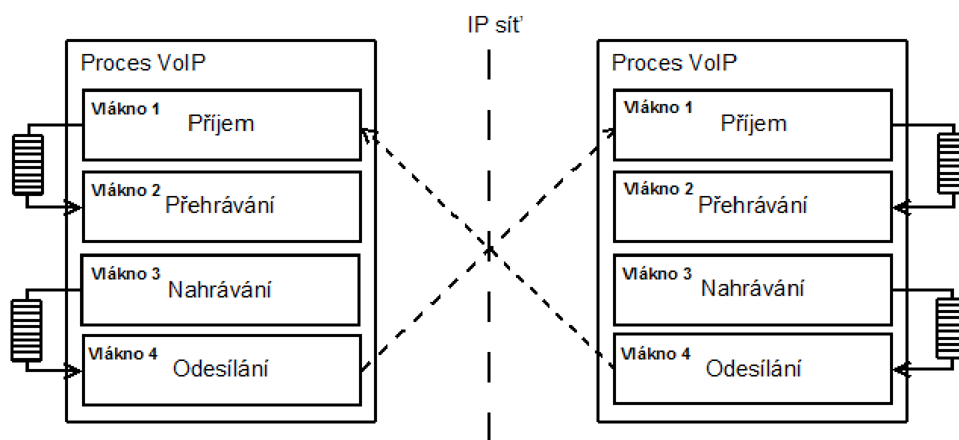


Obr. 17) Architektura, dvě vlákna

Lze také uvažovat o třech vláknech, kde by nahrávání a odesílání bylo v jednom vlákně a příjem s přehráváním v dalších dvou samostatných vláknech. Přijatá data by se do vlákna pro přehrávání předávala přes sdílený buffer. Tato myšlenka vychází z předpokladu, že nahrávání a odesílání bude v celku bezproblémové. Zatímco u příjmu a přehrávání mohou nastávat různé problémy jako: nedostatek či přebytek dat pro přehráva-

ní, s tím související nutnost prodlužování či zkracování stávajících dat, případná aplikace digitálních filtrů, atd. Proto se zdá být vhodnější činnosti příjem a přehrávání rozdělit a provozovat je paralelně.

Další možností je vytvoření samostatného vlákna pro každou ze základních činností. Tuto situaci zobrazuje obrázek Obr. 18 (zdroj vlastní). Jedno vlákno provádí příjem zvukových dat ze sítě a ukládá je do sdíleného bufferu pro přehrávání. Vlákno číslo dvě přijatá data ze sdíleného bufferu přehrává. V pořadí vlákno číslo tři provádí nahrávání a ukládání nahraných dat do sdíleného bufferu pro odesílání. Poslední čtvrté vlákno obstarává odesílání dat ze sdíleného bufferu.



Obr. 18) Architektura, čtyři vlákna, dva sdílené buffery

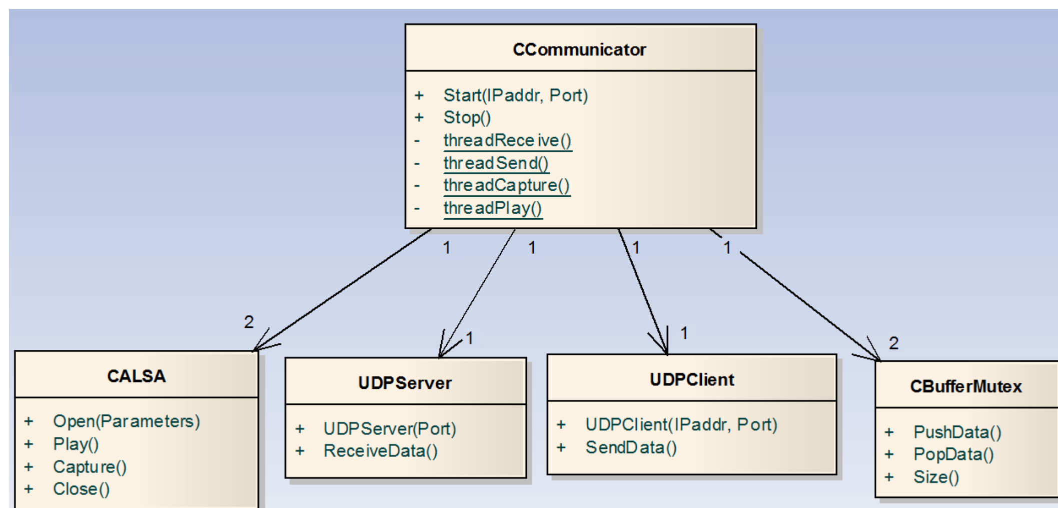
První varianta se dvěma vlákny se zdá být na první pohled jednodušší a díky ušetřeným vláknům i šetrnější na systémové zdroje. Ovšem pro větší efektivitu, možnosti a přehlednost jsou dle mého názoru vhodnější varianty se třemi a čtyřmi vlákny. Pro další vývoj je zvolená architektura se čtyřmi vlákny a dvěma sdílenými buffery.

6.2.3 Návrh tříd a jejich UML diagram

V návrhu aplikace bude využit objektově orientovaný přístup. Aplikace vytvořená pomocí OOP může být teoretiky o něco málo pomalejší oproti aplikaci vytvořené strukturovaným programováním, ale za to lze využít výhod jako: abstrakce, zapouzdření, dědičnost, polymorfismus. Výhody OOP dle mého názoru jednoznačně převažují.

Základní myšlenkou návrhu je vytvořit třídy, které budou poskytovat základní služby VoIP aplikace. Tedy třídu umožňující přehrávání či nahrávání. Vytvořit třídy pro příjem a odesílání pomocí protokolu UDP. Dále pak třídu, která zapouzdří pole či jiný

datový kontejner a bude moci být využita jako sdílený buffer. Všechny tyto třídy budou součástí další třídy, která všechny jejich vlastnosti zapouzdří a umožní jejich cílené využití. Základní myšlenku shrnuje obrázek Obr. 19 (zdroj vlastní).

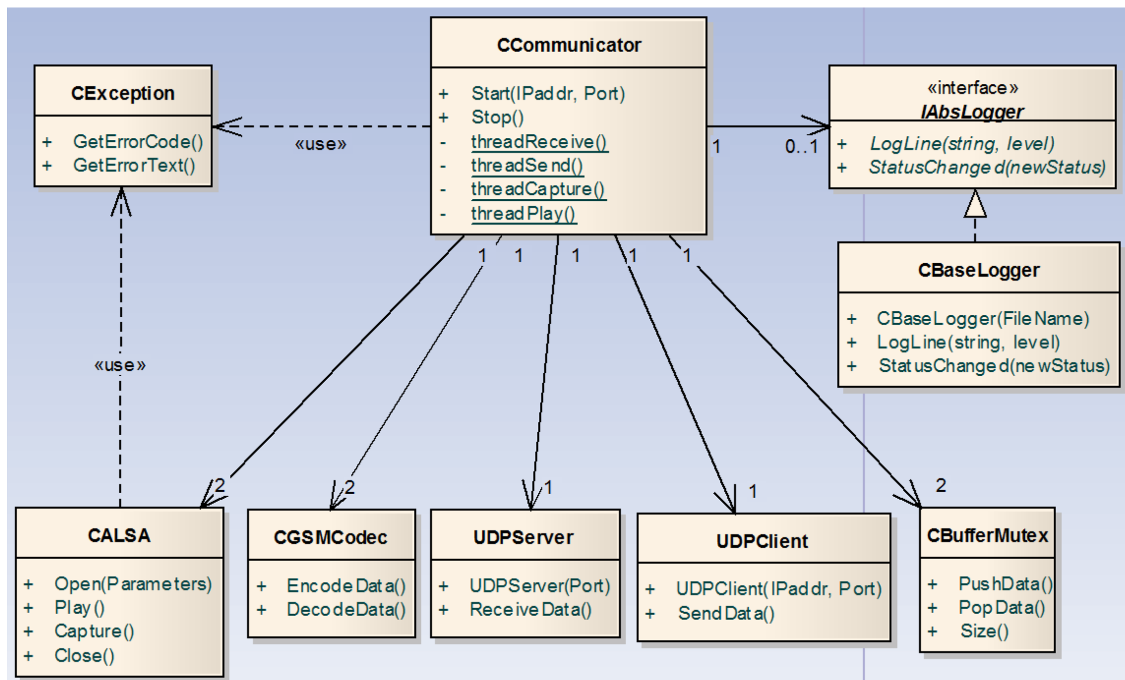


Obr. 19) Základní návrhový diagram tříd

Třída „CCommunicator" se skládá ze dvou tříd „CALSA", jedné třídy „UDPServer", jedné třídy „UDPClient" a dvou tříd „CBufferMutex". Dále třída „CCommunicator" obsahuje čtyři vlákna, která využívají instance předchozích tříd (viz kap. 6.2.2). Hlavní účel třídy „CALSA" popisují zobrazené operace, které umožňují otevření zvukového zařízení, přehrávání či nahrávání a uzavření otevřeného zařízení. Od třídy „UDPServer" se především očekává příjem dat na požadovaném portu. Naopak třída „UDPClient" by měla poskytovat operace, které zajistí odeslání dat na zadanou IP adresu a číslo portu. Poslední vyobrazená třída s názvem „CBufferMutex" bude využita jako sdílený buffer. Třída musí umožňovat vkládání bloku dat na konec bufferu a naopak odebírání bloku dat ze začátku bufferu. U těchto procesů musí být zajištěna vzájemná exkluzivita z důvodu předpokládaného používání instancí této třídy z více vláken.

Ze zadání práce jasně vyplývá, že aplikace má ke komprimaci přenášených zvukových dat využít kodek GSM. Do návrhu tedy přibude další třída, která bude umožňovat komprimaci a dekomprimaci pomocí tohoto zvukového kodeku, který je navržen pro kódování mluvené řeči. Dále do předchozího návrhu přibude také třída, které zapouzdřuje i popisuje případné chyby a některé ostatní třídy ji budou k nahlášení vzniklé chyby používat. Poslední změny v návrhu umožní logování aplikace dle daných potřeb.

Pokročilejší podobu návrhového diagramu tříd zachycuje obrázek Obr. 20 (zdroj vlastní).



Obr. 20) Pokročilý návrhový diagram tříd

Rozhraní (Interface) „IAbsLogger“ obsahuje dvě metody[7]. První metoda slouží k logování zadaného řetězce a jako parametr také přijímá důležitost daného sdělení. Důležitost bude později definována jako nějaký výčtový typ. Druhá metoda slouží k informování, že došlo ke změně ve stavu komunikace např. inicializace, spojeno, atd. Stav bude později také definován jako nějaký výčtový typ. Realizace rozhraní „IAbsLogger“ třída „CBaseLogger“ musí definovat obě metody rozhraní. Tato třída bude sloužit jako základní třída určená k logování a všechny poskytnuté informace bude ukládat do zadaného souboru. Díky této konstrukci je později možné podle potřeb vyměnit třídu „CBaseLogger“ za třídu jinou. Jiná třída musí opět definovat metody z rozhraní a se vstupními parametry může naložit úplně jinak. Např. v případě GUI (Graphical user interface) aplikace může měnit podle stavu komunikace nějaký nápis či ikonu atd.

Předpokládá se, že navržené třídy budou uzavřeny do statické knihovny[8]. Tato knihovna bude využita při vývoji jednoduchých aplikací, ať už grafických či konzolových, které budou volat metody třídy „CCommunicator“ s požadovanými parametry.

[7] Pro jazyk C++ je nutné nahradit interface abstraktní třídou s čistě virtuálními metodami.
[8] Binární kód statická knihovny je při linkování přidán do výsledné aplikace.

Tento přístup demonstruje následující obrázek Obr. 21 (zdroj vlastní). Ten znázorňuje několik různých aplikací, které jsou založeny na stejné statické knihovně.

konzolová aplikace pro IP-telefonii RADOMVoIP	konzolová aplikace2 pro IP-telefonii RADOMVoIP2	grafická aplikace pro IP-telefonii RADOMVoIPGUI
statická knihovna libRADOMVoIP.a třida CCommunicator Start(IP,port); Stop();	**statická knihovna libRADOMVoIP.a** třida CCommunicator Start(IP,port); Stop();	**statická knihovna libRADOMVoIP.a** třida CCommunicator Start(IP,port); Stop();

Obr. 21) Použití statické knihovny

7. Implementace navržené aplikace

Samotné aplikace i třídy byly vytvořeny v jazyce C++. Při implementaci bylo kromě příslušných manuálových stránek OS Linux využito především zdrojů [8], [26] a [27].

7.1 Implementace navržených tříd

Za předpokladu znalosti jazyka C++ byly již v předchozím průběhu práce poskytnuty téměř všechny potřebné informace pro implementaci navržených tříd umožňující VoIP v OS Linux. Jako poslední zbývá poskytnout základní informace o programování s vlákny, protože aplikace byla v kapitole 6 navržena za pomocí více vláken. Další podkapitoly se již budou zabývat samotným popisem implementace navržených tříd (viz Obr. 20).

7.1.1 Vlákna v OS Linux

V operačním systému Linux se používají vlákna splňující normu POSIX. V programu, který používá vlákna je nutné direktivou „include" zahrnout hlavičkový soubor `pthread.h` a linkovat se statickou knihovnou `libpthread.a`. Pro práci s vlákny existuje celá řada funkcí. Dle mého názoru většinou dostačují funkce následující:

- `pthread_create()` – vytvoření vlákna,
- `pthread_join()` – čekání na ukončení daného vlákna (spojení dvou vláken),
- `pthread_mutex_init()` – inicializace struktury mutexu, vysvětleno dále v textu,
- `pthread_mutex_lock()` – uzamčení mutexu,
- `pthread_mutex_unlock()` – odemknutí mutexu,
- `pthread_mutex_destroy()` – zničení struktury mutexu.

Mezi jednotlivými vlákny se často přistupuje ke shodné paměti (např. nějaké pole sdílených hodnot). Vlákna probíhají paralelně a proto je většinou nutné zajistit konzistenci sdílených dat. Jednoduchým nástrojem pro zajištění konzistence je právě

MUTEX (mutual exclusion – vzájemná exkluzivita). Při správném použití mutexu je zajištěno, že daný kód provádí vždy jen jedno vlákno. Použití mutexu vypadá typicky následovně.

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, 0);
/* nejaky kod */
pthread_mutex_lock(&mutex);
/* kriticka sekce */
pthread_mutex_unlock(&mutex);
/* nejaky kod */
pthread_destroy(&mutex);
```

Vlákno, které první vstoupí do kritické sekce, mutex uzamkne a ostatní vlákna jsou blokována[9], dokud není mutex opět odemknut. Odemknutí mutexu musí být provedeno, jinak by ostatní vlákna čekala do nekonečna. Jak může vypadat jednoduchý program s vlákny v jazyce C/C++, ukazuje následující konstrukce.

```
/* funkce vlakna */
void* thread_function(void* param){
      /* nejaka cinnost */
      return 0;
}

int main()
{
      /*nejaky kod*/
      pthread_t thread;
      pthread_create(&thread, 0, thread_function, (void*)param);
      /* nejaka cinnost */
      pthread_join(thread, 0);
      /*nejaky kod*/
      return 0;
}
```

Ve funkci vlákna bývá obvykle nějaký cyklus. Jako přílohu F přikládám jednoduchý program, ve kterém dvě vlákna inkrementují sdílenou hodnotu. Kritická sekce je ošetřena pomocí mutexu. Více lze opět najít v příslušných manuálových stránkách OS Linux. Jiný zdroj je např. [31].

7.1.2 Implementace třídy pro použití kodeku GSM

V návrhu byla tato třída pojmenována „CGSMCodec“. V kapitole 2.2.1 bylo představeno použití tohoto kodeku v OS Linux. Pro použití v objektovém programu je

[9] Funkcí pthread_mutex_try_lock() je možné provést neblokující uzamčení. Je-li kritická sekce volná, mutex je uzamčen, jinak volání funkce ihned skončí.

vhodné vše potřebné ke kódování pomocí tohoto kodeku jednoduše zabalit do jedné třídy. Samotné zapouzdření do třídy je triviální a nebude dále rozebíráno.

7.1.3 Implementace třídy pro sdílený buffer

Třída, která byla v návrhu pojmenována „CBufferMutex“, má umožňovat vkládání bloku dat na konec bufferu a odebírání bloku dat ze začátku bufferu. K tomuto lze výhodně využít datový kontejner z STL[10] s názvem „Double-ended Queues“. Kontejner se sám stará o správu paměti a umožňuje požadované operace. Aby bylo možné třídu využít mezi více vlákny, jako sdílený buffer, je pro klíčové operace využit mutex (viz kap. 7.1.1). Veškeré metody prakticky zapouzdřují metody kontejneru s použitím mutexu. Např. metoda pro vložení bloku dat nakonec bufferu vypadá následovně.

```
void CBufferMutex::PushDataBack(const unsigned char *inData, const
unsigned int count)
{
	pthread_mutex_lock(&m_mutex);
	m_buffer.insert(m_buffer.end(),inData,inData+count);
	pthread_mutex_unlock(&m_mutex);
}
```

7.1.4 Implementace tříd pro UDP komunikaci

UDP komunikaci se bylo poměrně detailně věnováno v kapitole 2.3.4 a v příloze B. Jedná se o jednoduché zapouzdření uvedených principů. Tyto principy byly zapouzdřeny tak, že pro odesílání třídou „UDPClient“ stačí vytvořit instanci a zadat číslo portu s IP adresou. Následně je již možné voláním metody `SendData()` data odesílat. Pro příjem dat třídou „UDPClient“ stačí vytvořit instanci a zadat číslo portu. Pak je již možné přijímat data voláním metody `RecvData()`. Oproti ukázce v příloze B bylo využito systémového volání `select()`, které zajistí návrat z metody po vypršení zadaného časového limitu. Pomocí tohoto systémového volání je možné kontrolovat i více soketů než jeden. Toho může být výhodně využito např. při vytváření nějakého kompaktnějšího serveru. Zdrojový kód v metodě pro příjem dat vypadá zhruba následovně.

```
int UdpServer::ServerRecvData(int usec, int sec, char *recvBuff,
sockaddr_in *clientInfo)
{
	timeval tv;
	fd_set set_socket;
	socklen_t addrlen;
```

[10] C++ STL (Standard Template Library) je generická kolekce tříd a algoritmů, které ulehčují běžné programátorské úkony.

```
	FD_ZERO(&set_socket);
	FD_SET(m_socket, &set_socket);
	tv.tv_sec = sec;
	tv.tv_usec = usec;
	addrlen = sizeof(sockaddr_in);

	int rc_select = select(m_socket+1, &set_socket, 0, 0, &tv);
	if (rc_select > 0)
		return recvfrom(m_socket, recvBuff,
				MAX_PACKET_SIZE, 0, clientInfo, &addrlen);
	else if (rc_select == -1)
		return -1;

	return 0;
}
```

7.1.5 Implementace třídy výjimek

Ve třídách, kde jsou očekávány větší potíže, na které je potřeba reagovat, je vhodné použít místo klasických návratových hodnot poněkud kompaktnější systém výjimek. Pro tento účel byla implementována třída s názvem „CException“, která zapouzdřuje číselný a textový popis chyby. Výjimek je využito ve třídě „CALSA“ pro přehrávání a nahrávání, dále pak ve třídě „CCommunicator“, která vše celkově zapouzdřuje. Výsledná uživatelská aplikace, ať už GUI nebo konzolová, bude dostávat chybová hlášení skrze tuto třídu.

7.1.6 Implementace třídy pro přehrávání a nahrávání

Tato třída, která byla v návrhu pojmenována jako „CALSA“, zapouzdřuje principy a postupy pro programování zvukové aplikace v ALSA API. Těmto postupům bylo věnováno dostatek prostoru v kapitole 4. Klíčovými jsou metody pro otevření zařízení a pro přehrání či nahrání. V případě, že dojde v těchto metodách k chybě, na kterou nemůže být na této úrovni reagováno, dojde k vyhození výjimky. Metoda pro otevření zařízení přijímá následující parametry: přehrávání/nahrávání, počet kanálů, vzorkovací frekvence, formát vzorku, název ALSA zařízení. Lze využít defaultních hodnot. Pro datové transfery do ALSA bufferu byl zvolen klasický prokládaný a neblokující přístup. Konstrukce metody pro otevření je zhruba následující.

```
void CAlsa::Open(param...)throw(CException){
	//inicializace
	Init();
	//pokus o otevreni zarizeni
	int returnCode = snd_pcm_open(&m_handle, device, mode, 0);
	if (returnCode < 0)
	{
```

```
            std::string err_str("snd_pcm_open(): ");
            err_str += snd_strerror(returnCode);
            CException exception(returnCode, err_str);
            throw exception;
      }
      //pokus o nastaveni hw parametru
      SetHwParameters(channels, rate, format);
      //pokus o nastaveni sw parametru
      SetSwParameters();
      //open ok
}
```

Filozofie v metodách zajišťující nahrání či přehrání dat je téměř shodná. Např. metoda pro přehrání přijímá jako parametr ukazatel na data a jejich počet. Zadaná data jsou přehrávána následujícím algoritmem, který zajišťuje přehrání požadovaných dat v případě, že nenastane fatální chyba. Takto je algoritmus postaven z důvodu, že se pokaždé nemusí podařit zapsat najednou požadovaný počet dat.

```
//public metoda
void CAlsa::PlayData(unsigned char* buffer, unsigned int count)
throw(CException){
      unsigned int playFrames= count/m_frameSize;
      unsigned int playedFrames= 0;

      while(playedFrames < playFrames)
            playedFrames += Play(buffer+playedFrames*m_frameSize,
                                        playFrames-playedFrames);
}
//private metoda
int CAlsa::Play(unsigned char* buffer, unsigned int frames)
throw(CException){
      int returnCode = snd_pcm_writei(m_handle, buffer, frames);
      /* REAKCE NA PRIPADNE CHYBY, osetreni a vyhozeni vyjimky */
      //vraceni poctu skutecne zapsanych bajtu
      return returnCode;
}
```

7.1.7 Implementace tříd pro logování

Implementace tříd pro logování spočívá ve vytvoření rozhraní „IAbsLogger“ a jeho realizace třídy „CBaseLogger“. V programovacím jazyce C++ žádné rozhraní neexistují, ale je možné použít abstraktní třídu bez členských atributů s čistě virtuálními metodami. Taková třída má pak stejné vlastnosti jako rozhraní v čistě objektových jazycích (Java, C#, atd.). Jak vypadá deklarace třídy „IAbsLogger“ ukazuje následující kód. Vzhledem k tomu, že se ve skutečnosti jedná o třídu, je název změněn na „CAbsLogger“. Popis úrovně logu a možné stavy (výčtový typ „CCommunicatorStatus“) se nachází v následující kapitole 7.1.8.

```
class CAbsLogger
{
public:
	virtual void LogLine(const char * line, int logLevel) = 0;
	virtual void StatusChanged(CCommunicatorStatus newStatus) = 0;
	virtual ~CAbsLogger() {}
};
```

Třída „CBaseLogger“ dědí od abstraktní třídy a je nucena definovat čistě virtuální metody. Samotná implementace metod pro logování není složitá. Zadané údaje jsou spolu s časovým razítkem přidány do daného souboru. Aby nedošlo ke ztrátě žádné informace při nekorektním ukončení celé aplikace, je nutné soubor pokaždé otevřít a uzavřít. Dále je očekáváno, že logování bude probíhat z více vláken, proto jsou klíčové operace doplněny mutexem (viz kap. 7.1.1). Jak vypadá jedna z metod třídy „CBaseLogger“ pro logování ukazuje následující kód.

```
void CBaseLogger::LogLine(const char * line, int logLevel)
{
	if (logLevel == 1 || m_debugLog == true)
	{
		pthread_mutex_lock(&m_mutex);
		//pridani retezce line nakonec log souboru
		m_file= fopen(m_logFileName.c_str(), "a");
		if (m_file != 0)
		{
			time_t now = time(0);
			char str_time[64];
			memset(str_time, '\0', 64);
			strftime(str_time, 63, "%H:%M:%S", localtime(&now));
			fprintf(m_file, "%s: %s\n", str_time, line);
			fclose(m_file);
		}
		pthread_mutex_unlock(&m_mutex);
	}
}
```

V případě, že je splněna vstupní podmínka, je otevřen soubor, jehož název byl zadán v konstruktoru třídy „CBaseLogger“. Je získán aktuální čas a ve vhodném formátu převeden na řetězec, který je spolu s daným údajem uložen na konec souboru. Soubor je následně uzavřen. Celý tento blok je uzamknut pomocí mutexu.

7.1.8 Implementace zapouzdřující třídy

Třída, která vše zapouzdřuje, se nazývá „CCommunicator“. Tato třída obsahuje jako členské atributy třídy ostatní (viz Obr. 20) a poskytuje metody `Start()`, `Stop()` pro samotné spouštění a zastavování komunikace. Třída také obsahuje členskou proměnou, ve které je uložen aktuální stav komunikace. Rozlišuje se mezi několika stavy:

komunikace zastavena, inicializace, komunikace připravena (dosud nebylo navázáno spojení), spojeno (probíhá hovor), komunikace se ukončuje. Mezi stavy komunikace připravena a spojeno je přepínáno podle počtu přijatých bajtů. Mezi oběma stranami komunikace není vedeno žádné další spojení.

Metoda `Start()` přijímá několik parametrů, z nichž má většina defaultní hodnoty. Jsou to parametry: IP adresa, číslo portu, instance třídy k logování a název zařízení pro nahrávání a přehrávání. V těle metody je provedena inicializace, jsou vytvořeny instance členských tříd a vytvořeny vlákna (viz Obr. 18). V případě chybných parametrů nebo jiných chyb je vyhozena výjimka s příslušným popisem. V metodách třídy jsou prováděny logy dvou úrovní: 0 – poskytuje podrobné ladící informace, 1 – poskytuje běžné informace o průběhu (např. ztráta spojení atd.). Také je logována každá změna stavu komunikace. S těmito informacemi může uživatelská aplikace naložit díky vhodné architektuře (viz kap. 6.2.3) dle libosti.

Samotná logika aplikace je ukryta uvnitř jejich vláken. Funkce vláken jsou privátní statické metody třídy. Činnosti jednotlivých vláken budou popsány samostatně, ale všechna vlákna v sobě mají následující konstrukci.

```
void *CCommunicator::thread_xxx(void *arg){
	//pretypovani, parametr vlakna byl operator this
	CCommunicator* m_class = reinterpret_cast<CCommunicator*>(arg);
	//zalogovani start
	if (m_class->m_logger != 0)
		m_class->m_logger->LogLine("Xxx start", 1);
	/*NEJAKY KOD */
	//hlavni smycka - opakuj dokud neni nastaveno stop*/
	while(m_class->m_stop==false){
		/*NEJAKA CINNOST*/
		//pripadne uspani, na vhodnou dobu
		usleep(xxx);
	}
	/*NEJAKY KOD*/
	//zalogovani konec
	if (m_class->m_logger != 0)
		m_class->m_logger->LogLine("Xxx end", 1);
	return 0;
}
```

7.1.9 Implementace vláken pro nahrávání a odesílání

Koncepce aplikace je postavena tak, že tyto dvě vlákna jsou jednodušší než zbylé. Ve vnitřní smyčce vlákna pro nahrávání jsou po sobě volány následující činnosti: nahrání bloku zvukových dat o velikosti vhodné pro kodek GSM (320 bajtů) za použití

instance objektu třídy „CALSA“, komprimace dat kodekem GSM pomocí třídy „CGSMCodec“, vložení komprimovaných dat do sdíleného bufferu pro odesílání třídy „CBufferMutex“. Ve smyčce vlákna pro odesílání je kontrolováno, jestli ve sdíleném bufferu je dostatek dat na odeslání. V případě, že ano, jsou data odeslána pomocí instance třídy „UDPClient“. Jako nejvhodnější velikost paketu se ukázalo být 33 bajtů, což je velikost zkomprimovaného bloku dat, kterou produkuje kodek GSM.

7.1.10 Implementace vlákna pro příjem dat

Toto vlákno má na starosti nejvíce úkonů. Nejprve obsahovalo v hlavní smyčce úkony: přijmutí dat s časovým limitem 50 ms pomocí instance třídy „UDPServer“, dekódování dat pomocí instance třídy „CGSMCodec“, vložení dat do sdíleného bufferu pro přehrávání třídy „CBufferMutex“. Později se ukázalo, že je nutné přidat úkon další. Jedná se o logiku postavenou nad množstvím dat, které jsou v bufferu pro přehrávání. V případě, že jsou pod určitou mez, jsou data jemně prodlužována. Naopak v případě, že přebývají, jsou jemně zkracována.

Tato logika je nutná hned z několika důvodů. Data po síti nemusí pokaždé přijít včas. Zvuková zařízení, která spolu komunikují, nejsou zpravidla naprosto stejně rychlá (nejsou vyrobeny tak přesně). Parametry jsou sice nastaveny shodně, ale na jednom zařízení je reálná délka periody o maličko jiná než na druhém. Dále může na jedné straně dojít k nějaké chybě, která má za následek zpoždění prováděných procesů. Je tedy nutné pružně udržovat množství dat v bufferu na optimální hladině. V případě, že jich je moc, zvyšuje se zpoždění hovoru. Naopak, když je jich málo, hrozí podtečení bufferu, což má za následek nepříjemné zvukové defekty v hovoru.

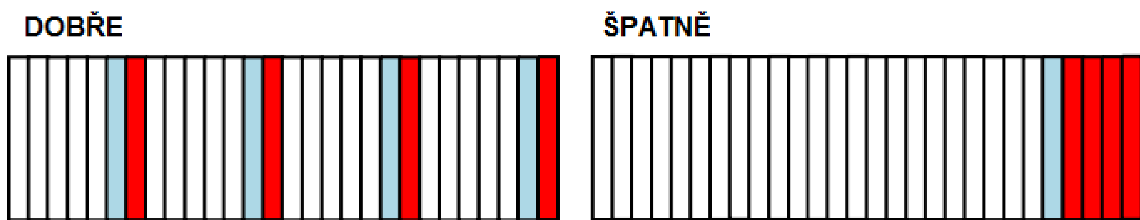
S předchozím souvisí několik otázek:

1. Jak získat přesné obsazení ALSA bufferu?
2. O kolik a jak lze upravovat data, aby to nebylo příliš slyšitelné?
3. Jak určit meze pro doplňování?

Kolik dat je v ALSA bufferu, by mělo jít získat pomocí API funkcí `snd_pcm_avail_update()` a `snd_pcm_delay()`. Jejich použití však nemohu doporučit. Získané údaje byly z nějakého mně neznámého důvodu dosti pochybné. Úspěšně byla aplikována jiná strategie. ALSA buffer udržovat neustále plný a optimální hla-

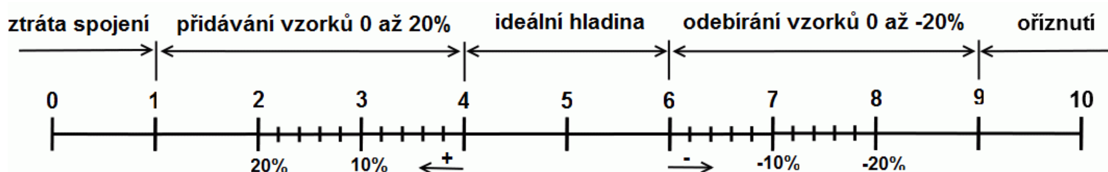
dinu dat udržovat ve sdíleném bufferu pro příjem, který je možné mít bez problému plně pod kontrolou. Výhodné je nastavit ALSA buffer na 4 periody a dalších několik period udržovat ve vlastním bufferu. Čtyři periody ALSA bufferu zajistí dostatečně dlouhý buffer pro datové transfery zvukového systému, další periody ve vlastním bufferu zase prostor pro udržování celkového množství dat na určité hladině.

Sérií pokusů bylo zjištěno, že zkracovat či prodlužovat blok data více než o 20 % je již příliš slyšitelné. Je-li potřeba prodloužit blok dat např. o 10 vzorků, není přidáno 10 vzorků na konec bloku, ale duplikované vzorky jsou rozděleny rovnoměrně na celý blok. Správné duplikování vzorků je zobrazeno na obrázku Obr. 22 (zdroj vlastní). Červenou barvou jsou označeny duplikované vzorky. Modrou barvou vzorky, ze kterých se duplikuje. Tímto způsobem je zajištěno, že úprava bude minimálně slyšitelná.



Obr. 22) Duplikování vzorků

Poslední otázkou je určení mezí (množství dat v bufferu pro přehrávání), při kterých se budou nově obdržené bloky dat modifikovat. Bylo provedeno několik dlouhých hovorů v řádech hodin mezi různými PC i vestavěnými systémy bez jakéhokoli prodlužování či zkracování dat. Z analýzy obsazení bufferu byla vypracována strategie, která se zdá být pro hovor výhodná. Tuto strategii zobrazuje následující obrázek Obr. 23 (zdroj vlastní).



Obr. 23) Strategie přidávání a odebírání vzorků v bufferu

V případě, že je ve sdíleném bufferu pro přehrávání 4 až 6 period dat, nově příchozí bloky nejsou modifikovány. Je-li v bufferu méně než 4 periody, jsou nově příchozí data doplňována. Každá pětina periody za hranicí čtyř, znamená prodlužování bloku

o další 2 % (max. 20 %). Při množství dat větším než 6 period dochází ke zkracování nových bloků dat. Platí, že každá pětina periody za hranicí šesti znamená zkracování bloku o další 2 % (max. 20 %). V případě většího množství dat v bufferu než je 9 period, jsou nejstarší data odebrána tak, aby zůstalo právě 9 period. Takto je zajištěno, že nenastane žádné větší zpoždění. Klesne-li hladina bufferu pod jednu periodu, pravděpodobně se jedná o ztrátu spojení.

Pro shrnutí, ve vlákně pro příjem je tady prováděno: příjem dat, dekódování dat, případná modifikace nového bloku dat podle nastíněné strategie, vložení dat do sdíleného bufferu pro přehrávání.

7.1.11 Implementace vlákna pro přehrávání

Na začátku tohoto vlákna, ještě před hlavní smyčkou (viz kód v kap. 7.1.8), je čekáno, až bude hladina dat ve sdíleném bufferu pro přehrávání na ideální hladině (viz Obr. 23). Následně je ALSA buffer z inicializačních důvodů naplněn až po okraj tichem a v hlavní smyčce začne být přehráváno pomocí instance objektu „CALSA“. Metoda třídy pro přehrání vlákno blokuje, dokud se jí nepodaří zapsat do ALSA bufferu požadovaný počet dat. Tím je hned od začátku držen ALSA buffer téměř plný. Při plném ALSA bufferu se snižuje riziko podtečení bufferu. Velikost bloku dat, který se zapisuje do ALSA bufferu (přehrává se), musí být volen tak, aby se na jeho příjem z IP sítě nečekalo příliš dlouho, ale zároveň nesmí být ani příliš malý (roste režie). Na většině systémů umožňujících bezproblémový provoz IP-telefonie se dá vhodná velikost pro přehrání odvodit z délky jedné části ALSA bufferu (periody), což je většinou cca 21 ms. Tato délka při klasickém nastavení pro VoIP (8000 Hz, mono) znamená hodnotu cca 170 vzorků (většinou 340 bajtů). V případě, že dojde při přehrávání k výjimce, je její výskyt uložen do logu a neprodleně se pokračuje v přehrávání, které nesmí být přerušeno.

7.1.12 Implementace vlákna pro kontrolu spojení

Při návrhu aplikace nebylo s tímto vláknem počítáno. Při vývoji se ovšem zjistilo, že je vhodné použít vlákno další, které by kontrolovalo stav spojení a v případě jeho ztráty resetovalo určité procesy v aplikaci. Tento postup byl zvolen z následujících důvodů. V případě, že došlo k výpadku, respektive chodí dat málo nebo dokonce žádná, klesne hladina dat v ALSA bufferu na minimum a přehrávání je přerušeno. To je samo

o sobě naprosto v pořádku, ale je-li spojení po výpadku obnoveno, počáteční nízká hladina bufferu způsobí špatnou kvalitu hovoru. Ovšem nestačí znova nabufferovat data na ideální hladinu, výpadek mohl být také důsledkem špatné funkčnosti nějaké části komunikace (přehrávání, nahrávání, komprese, dekomprese, příjem, odesílání), pro jistotu jsou po výpadku veškeré tyto činnosti resetovány. Tím je zajištěna správná kvalita hovoru i po obnově výpadku. Samotný reset je poměrně rychlý (v desítkách ms).

Pro shrnutí, vlákno pro kontrolu spojení je vytvořeno již v konstruktoru zapouzdřující třídy „CCommunicator". V případě, že probíhá komunikace a bylo dosaženo spojení (jsou přijímána data), je kontrolováno, zda neklesla hladina ve sdíleném bufferu pro přehrávání pod jednu periodu (viz Obr. 23). Klesne-li, jsou zavolány metody třídy „CCommunicator" `Stop()` a `Start()`, které zařídí automatický reset komunikace. Ukončení tohoto vlákna je provedeno až při ničení instance třídy.

7.1.13 Shrnutí implementace navržených tříd

Celková logika a funkčnost je schovaná ve vláknech třídy „CComunicator". Celkem bylo použito pět vláken, které při své činnosti používají zbylé implementované třídy (viz Obr. 20). Po výpadku spojení je zaručeno opětovné korektní navázání hovoru. Na systémech, kde je možné nastavit požadované parametry, je maximální možné zpoždění dané délkou 13 period (4 x ALSA buffer + 9 x vlastní buffer), což je cca 270 ms. Očekávaná průměrná hodnota zpoždění je zhruba 190 ms. Požadované parametry lze samozřejmě správně nastavit na většině PC a i na vestavěném systému EXM32 s CPU modulem AU1250, pro který je vyvíjená aplikace primárně určena.

Aby bylo možné implementované třídy využít tak, jak je zobrazeno na obrázku Obr. 21, je nutné ze zdrojových kódů tříd vytvořit binární objekty, ze kterých lze následně sestavit statickou knihovnu. To je možné provést pomocí speciálního projektu v nějakém vývojovém prostředí (např. Eclipse[11]) nebo „ručně" následujícím způsobem.

```
#zkompiluje vsechny cpp soubory v adresari do binarnich objektu
g++ -Wall -O2 -D_REENTRANT -c *.cpp
#vytvori ze vsech bin. objektu v adresari knihovnu libRADOMVoIP.a
ar -r libRADOMVoIP.a *.o
```

[11] Dle mého názoru jedno z nejlepších open-source vývojových prostředí. Možné získat na url „http://www.eclipse.org/downloads/".

7.2 Implementace konzolové aplikace

Samotná konzolová aplikace umožňující IP-telefonii je velice jednoduchá. Spočívá ve využití třídy „CCommunicator“ zkompilované do knihovny. Aplikace přijme jako argument IP adresu a zavolá metodu `Start()`. Klíčový blok tohoto krátkého programu je následující.

```
CCommunicator comunicator;
CBaseLogger* logger = new CBaseLogger();
try{
	comunicator.Start(IP, 10001, logger, "default", "default");
}catch(CException& ex){
	cout << ex.GetErrorCode() << ": " << ex.GetErrorText() << endl;
	return 0;
}
```

Aby bylo možné bez problému ovládat aplikaci běžící na pozadí, je využito signálů[12]. Signály jsou v aplikaci nastaveny tak, aby se na signál `SIGINT` provedlo korektní ukončení a na signál `SIGUSR1` uložení základních informací o běhu programu do souboru s názvem „RADOMVoIP.info“. Toto je docíleno použitím několika systémových volání. Na začátku aplikace je následující kód.

```
struct sigaction new_actions;
//presmerovani signalu do funkce Action()
new_actions.sa_handler = Action;
new_actions.sa_flags = SA_RESTART;
//SIGKILL -- nelze presmerovat
//SIGSTOP -- nelze presmerovat
//SIGUSR1 -- info do souboru
int signal_ar[] = {SIGHUP, SIGINT, SIGQUIT, SIGILL, SIGABRT, SIGFPE,
SIGSEGV, SIGPIPE, SIGALRM,SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT,
SIGTSTP, SIGTTIN, SIGTTOU};
int signals = sizeof(signal_ar)/sizeof(int);
setSignal(new_actions, signal_ar, signals);
```

V případě, že proces obdrží nějaký ze signálů uvedených v poli `signal_ar`, je zavolána funkce `Action()`. V těle této funkce je reagováno pouze na signál `SIGUSR1`, ostatní signály jsou ignorovány. Je nutné ovšem přesměrovat veškeré signály, i když na ně není reagováno. Nepřesměrovaný signál by způsobil ukončení aplikace, což není žádoucí. Samotné přiřazení akce danému signálu probíhá ve funkci `setSignal()`. Kód funkcí `Action()` a `setSignal()` vypadá následovně.

[12] Nástroj pro komunikaci mezi procesy v OS Linux. Více např. zdroj [32].

```
void setSignal(struct sigaction& action, int *signals_ar, int signals)
{
	for(int i=0; i<signals; i++)
		sigaction(signals_ar[i], &action, 0);
}
void Action(int sig)
{
	//na sigusr1 vypsat info, jinak nic
	if (sig == SIGUSR1)
	{
		 ofstream fout("RADOMVoIP.info");
		 fout << "********IP/PORT**********" << endl;
		//ulozeni ruznych informaci.....
	}
}
```

Po spuštění komunikace metodou `Start()`, tak jak bylo na začátku kapitoly ukázáno, je běžící aplikace pozastavena pomocí dalšího systémového volání. Ovšem komunikace běží dál v jiných vláknech. Systémové volání `sigsuspend()` umožňuje pozastavení běhu aplikace, dokud není přijat signál, který nevyhovuje nastavené masce. Maska je nastavena tak, aby nevyhovoval právě signál `SIGINT`, který uspanou aplikaci probudí. Následující kód aplikace běžící komunikaci korektně ukončí a aplikace skončí. Pozastavení běhu aplikace je zajištěno následujícím kódem.

```
//nastaveni masky a pozastaveni dokud neprijde SIGINT
sigset_t mask;
sigemptyset(&mask);
setMask(mask, signal_ar, signals);
sigsuspend(&mask);
//kod funkce na nastaveni masky
void setMask(sigset_t& mask, int *signals_ar, int signals)
{
	//Na sigint konec
	for(int i=0; i<signals; i++)
		if (signals_ar[i] != SIGINT)
			sigaddset(&mask, signals_ar[i]);
}
```

Signál lze procesu poslat příkazem `kill`. Detailní informace o jednotlivých systémových voláních nebo příkazech lze získat opět v příslušných manuálových stránkách OS Linux (např. man 2 sigsuspend, man kill, atd.).

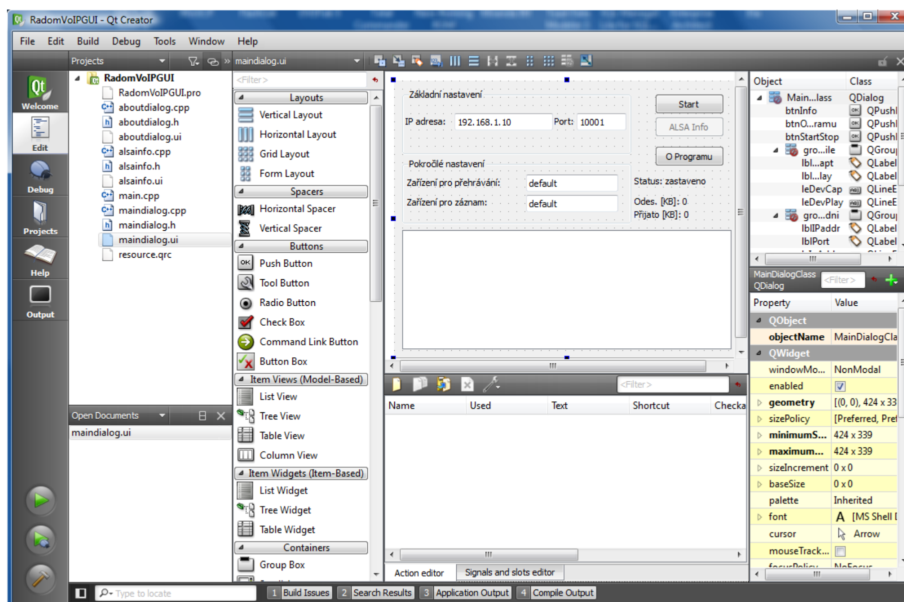
7.3 Implementace GUI aplikace pro testovací a prezentační účely

Vývoj grafické aplikace sice není součástí zadání práce, ale GUI aplikace se skvěle hodí pro testovací a prezentační účely. Implementace GUI aplikace byla o něco náročnější než implementace aplikace konzolové, ale prakticky se jedná také pouze o

vytvoření rozhraní pro použití tříd ukrytých do knihovny umožňující VoIP v OS Linux (implementace tříd kap. 7.1).

Grafická aplikace byla vytvořena pomocí C++ „frameworku" Qt4 ve vývojovém prostředí „Qt Creator"[13]. Prostředí je zobrazeno na obrázku Obr. 24 (zdroj vlastní). Při vývoji grafické aplikace byla využita dobře zpracovaná dokumentace pro Qt zdroj [33].

Po založení projektu jsou jednotlivé komponenty jako tlačítka, input boxy, atd. jednoduše přetaženy metodou „drag and drop" na výsledný formulář. Vlastnosti každé komponenty je možné konfigurovat pomocí editoru vlastností. Během několika kliknutí myši lze vygenerovat veškerý kód pro např. reakci na zmáčknutí tlačítka. Do tzv. slotu, který je speciální metodou třídy navrženého formuláře, lze psát kód, který se provede po dané události (např. zmáčknutí tlačítka).



Obr. 24) Qt Creator

Díky dobře zvolené architektuře tříd (viz Obr. 20, str. 52) bylo možné velice elegantně vyřešit logování grafické aplikace. Poslední logy jsou zobrazovány do textového okna aplikace a celkový log je dostupný ve formě souboru. Dále změna stavu hovoru (inicializace, připraveno, spojeno, atd.) je zobrazena v textovém popisku a v titulku dialogového okna. Toho bylo dosaženo jednoduše následujícím způsobem.

[13] Kompletní vývojové prostředí je momentálně dostupné na url: „http://www.qtsoftware.com/downloads".

```
class MainDialog : public QDialog, CAbsLogger
{
	Q_OBJECT
public:
	//...
private:
	//CAbsLogger
	void LogLine(const char * line, int logLevel);
	void StatusChanged(CCommunicatorStatus newStatus);
	//...
private slots:
	//...
};
```

Hlavní třída dialogu dědí od třídy „CAbsLogger“ a musí předefinovat dané metody. Aby do předefinovaných metod proudily informace o běhu komunikace z třídy „CCommunicator“ a jejich vláken, je předán při spuštění komunikace jako parametr ukazatel na instanci hlavního dialogu pomocí operátoru `this`. To ukazuje následující kód, který je součástí reakce na zmáčknutí tlačítka pro start komunikace.

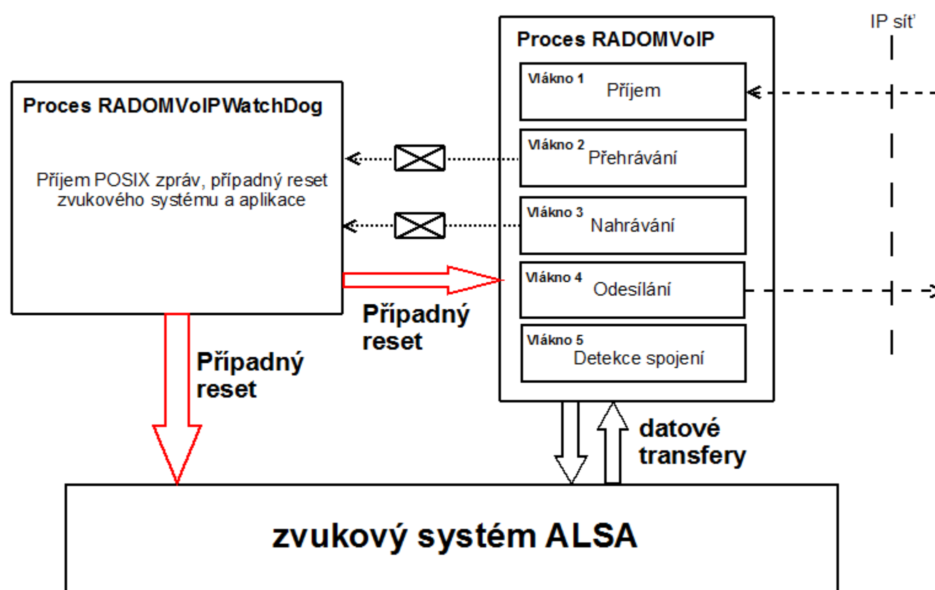
```
logStart();
//start communicator
m_communicator = new CCommunicator();
try{
	m_communicator->Start(IPaddr, portNum, this, devPlay, devRec);
}catch(CException& ex){
	QMessageBox::critical(this, "Error",
	                        QString::fromUtf8(ex.GetErrorText()));
	m_communicator->Stop();
	delete m_communicator;
	m_communicator=0;
	logEnd();
	return;
}
```

V metodách pro logování jsou upravovány grafické komponenty podle příchozích údajů tak, jak bylo popsáno (přidáváni logů do textového okna, atd.). Samotné ovládání grafické aplikace zobrazené na obrázku Obr. 25 je velice snadné. Prakticky stačí vyplnit IP adresu a zmáčknout tlačítko „Start“. Aplikace zobrazuje aktuální počet přijatých a odeslaných bajtů. Také je možné si zobrazit informace o nastavených ALSA HW a SW parametrech (viz kap. 4.1). V případě, že nastane nějaká zásadní chyba, je uživatel informován oknem s popisem dané chyby.

Obr. 25) Vyvinutá GUI aplikace

7.4 Implementace kontrolního mechanismu

V kapitole 5.2.2 byl zmíněn problém se stabilitou zvukového systému pro vestavěný systém EXM32 s CPU modulem AU1250. Při otevírání zvukového zařízení se občas stávalo, že aplikace pravděpodobně vinou ovladače zamrzla. Aplikace i zvukový systém musely být pro obnovu činnosti restartovány. Problém se zdá být vyřešen neustálým přehráváním ticha na pozadí, ale dal důvod ke vzniku kontrolního mechanismu, který by hlídal kritické aspekty komunikace (nahrávání a přehrávání). Představu o kontrolním mechanismu zobrazuje následující Obr. 26 (zdroj vlastní).



Obr. 26) Kontrolní mechanismus

Spolu s aplikací pro komunikaci běží další proces, který očekává pravidelné ohlášení od vláken komunikace pro nahrávání a přehrávání. V případě, že ohlášení nenastane do určitého časového limitu, je zvukový systém i aplikace pro komunikaci resetována. To znamená, že hlídací proces tzv. „watchdog“ spouští a v případě potřeby resetuje samotnou VoIP komunikaci. Tento mechanismus je založený na POSIX zprávách[14]. Hlídací proces „watchdog“ vytvoří perzistentní frontu pro zprávy a spustí proces pro komunikaci, který v hlavních smyčkách vláken pro nahrávání a přehrávání odesílá zprávy v předem definovaném formátu do vytvořené fronty. Hlídací proces neustále ve smyčce zprávy přijímá a ukládá si, kdy přijal poslední zprávu od daného vlákna. V případě, že je překročen časový limit, je proveden zmiňovaný reset. Při implementaci byly využity následující funkce pro práci se zprávami:

- `mq_open()` – vytvoření nebo otevření fronty zpráv,
- `mq_send()` – odeslání zprávy,
- `mq_timedreceive()` – příjem zprávy s časovým limitem,
- `mq_close()` – uzavření fronty zpráv,
- `mq_unlink()` – odstranění perzistentní fronty.

Detailní popis těchto funkcí lze opět získat v manuálových stránkách OS Linux (např. man mq_open()). Do metody `Start()` třídy „CCommunicator“, která je součástí vyvinuté statické knihovny pro vývoj VoIP aplikací v OS Linux, přibyl parametr, který určuje, jestli má být použit hlídací mechanismus. V případě, že ano, je fronta otevřena a ve vláknech pro nahrávání i přehrávání jsou odesílány zprávy. Na začátek hlavních smyček vláken přibyl následující kód.

```
//odesilani kontrolnich zprav
if (m_class->m_useWatchDog == true)
      if ( mq_send(m_class->m_queueId, (char*)&msg_buf,
                                    sizeof(SMsg), 0)  == -1 )
            if (m_class->m_logger != 0)
                  m_class->m_logger->LogLine("mq_send() error", 1);
```

Je-li použit hlídací mechanismus „odešli zprávu“, případné selhání odeslání je logováno. Samotná struktura zprávy je následující.

[14] Informace o POSIX frontě zpráv lze získat např. v manuálových stránkách OS Linux příkazem „man mq_overview“. Pro používání je nutné mít aktivovanou podporu v jádře.

```
enum CommWatchDogMsgType{
	//ohlaseni nebo zacatek sledovani
	msg_report = 0,
	//zastaveni sledovani
	msg_stopWatch = 1
};
//struktura zpravy
struct SMsg{
	int id;
	CommWatchDogMsgType type;
};
```

První položka struktury označuje, od jakého vlákna byla zpráva přijata. Vlákna odesílají zprávy, každé se svým `id`. Druhá položka určuje, jestli se jedná o ohlášení nebo začátek či konec sledování. Tento hlídací mechanismus byl doimplementován pouze do konzolové aplikace. Algoritmus procesu „watchdogu" by se dal shrnout do následujících kroků:

- vytvoření fronty,
- načtení konfigurace ze souboru (příkaz ke spuštění konzolové aplikace pro VoIP, příkaz pro reset zvukového systému),
- spuštění aplikace pro VoIP,
- hlavní smyčka (příjem zpráv, kontrola časového limitu, případný reset),
- uzavření a smazání fronty zpráv.

Aplikace hlídacího mechanismu se ukončí korektně při obdržení signálu `SIGINT` (viz kap. 7.2) stejně jako konzolová aplikace pro IP-telefonii. Pro pohodlné ovládání celkového konceptu pro VoIP (hlídací mechanismus, aplikace pro hovor) byly vytvořeny skripty `start.sh`, `stop.sh` , `status.sh`. První skript se postará o spuštění procesů, druhý o jejich bezpodmínečné ukončení a třetí vypíše, zda jsou procesy spuštěny. Spuštění procesů ve skriptu pro start vypadá následovně.

```
PID_WATCHDOG=$(ps -o pid -C RADOMVoIPWatchDog -no-heading)
#spusti pouze neni-li uz spusten, watchdog spusti samotny RADOMVoIP
if [[ ${PID_WATCHDOG} == "" ]] ; then
	bin/RADOMVoIPWatchDog > /dev/null &
fi
```

V případě, že již proces „watchdog" není spuštěn, provede se spuštění procesu hlídacího mechanismu na pozadí (ten spustí VoIP aplikaci), výstupy jsou zahazovány. Taktika skriptu pro ukončení je následující: pokus o korektní ukončení aplikace VoIP

odesláním signálu `SIGINT`, násilné ukončení aplikace pro VoIP odesláním signálu `SIGKILL`, pokus o korektní ukončení „watchdog“ aplikace, násilné ukončení „watchdog“ aplikace. Následuje klíčový blok skriptu pro ukončení, odeslání signálu `SIGINT` VoIP aplikaci a následně čekání, jestli se ukončí. Další bloky skriptu jsou obdobné.

```
if [[ ${PID_COMMUNICATOR} != "" ]] ; then
      #stopping RADOMVoIP communicator, max 2 sec
      kill -s INT ${PID_COMMUNICATOR}
      COMM_END=0
      for ((  i = 0 ;  i<20;  i++  ))
      do
            bin/usleep 100000
            PID_COMMUNICATOR_TMP=$(ps -o pid -C RADOMVoIP --no-heading)
            if [[ ${PID_COMMUNICATOR_TMP} == "" ]] ; then
                  echo "RADOMVoIP end $i"
                  COMM_END=1
                  break
            fi
      done
      #nasleduje pripadne nasilne ukonceni...
fi
```

8. Závěr

Cílem práce bylo vytvořit aplikaci, která bude moci být použita pro přenos zvuku v OS Linux na vestavěných systémech EXM32. Proto byla vytvořena knihovna pro vývoj VoIP aplikací, jejíž funkčnost byla úspěšně ověřena pomocí dvou aplikací, které jsou nad knihovnou postavené. Jedná se o aplikaci konzolovou, primárně určenou pro nasazení společností RADOM, s.r.o., dále pak o aplikaci s grafickým uživatelským rozhraním, určenou k testování a prezentačním účelům.

Přes značné problémy, které se při vývoji objevily:

- špatná dokumentace zvukového API,
- nedostatečný výkon původního CPU modulu,
- fatální nespolehlivost zvukového systému.

Byla vyvinuta konzolová aplikace, u které lze předpokládat úspěšné nasazení do provozu. Obě výsledné aplikace splňují veškeré na ně kladené požadavky. Především nabízí podobnou kvalitu i zpoždění jako hovor přes mobilní telefon. Navíc nad aplikací konzolovou byl pro jistotu postaven hlídací mechanismus, který zajišťuje automatické restartování aplikace i zvukového systému v případě detekovaných problémů.

V diplomové práci i při samotném vývoji byla pootevřena některá témata, jejichž řešení dalece přesahuje rozsah této práce. Jedná se především o:

- implementaci speciálních protokolů pro přenos multimediálních dat a řízení relace,
- softwarové řešení eliminace ozvěn,
- možnosti a aplikace digitálních filtrů.

Je pravděpodobné, že některé z uvedených témat budou v budoucnu řešena a dodatečně implementována do stávající knihovny či aplikace. Především možnosti digitální filtrace jsou pro zadavatele společnost RADOM, s.r.o. velice zajímavé.

Seznam použitých zdrojů

1. PETERKA, Jiří. *Počítačové sítě : Lekce č. 5: Základy datových komunikací* [online]. v. 3.3. Praha : Univerzita Karlova, 2007 [cit. 2009-04-13]. Dostupný z WWW: <http://www.earchiv.cz/l218/slide.php3?&l=5&me=1>.

2. KLIMO, Martin, et al. *Teória IP telefonie*. 1. vyd. Žilina : Žilinská univerzita, 2009. 398 s. ISBN 978-80-8070-915-0.

3. WOODARD, Jason. *Speech Coding* [online]. Southampton : University of Southampton , neznámý [cit. 2009-04-13]. Dostupný z WWW: <http://www-mobile.ecs.soton.ac.uk/speech_codecs/index.html>.

4. 4Front Technologies. *4Front Technologies : Digital Audio Solutions* [online]. [1996] , March 25, 2009 [cit. 2009-04-13]. Dostupný z WWW: <http://www.opensound.com/>.

5. *Open Sound System* [online]. 2008 , 29. 11. 2008 [cit. 2009-04-13]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Open_Sound_System>.,

6. *Advanced Linux Sound Architecture* [online]. 2008 , 9. 12. 2008 [cit. 2009-04-13]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Advanced_Linux_Sound_Architecture>.

7. *Advanced Linux Sound Architecture (ALSA) project homepage* [online]. 2008 , 21 January 2008 [cit. 2009-04-13]. Dostupný z WWW: <http://www.alsa-project.org/main/index.php/Main_Page>.

8. *ALSA project - the C library reference* [online]. 2009 [cit. 2009-04-13]. Dostupný z WWW: <http://www.alsa-project.org/alsa-doc/alsa-lib/>.

9. TRANTER, Jeff . *Introduction to Sound Programming with ALSA* [online]. October 1st, 2004 [cit. 2009-04-13]. Dostupný z WWW: <http://www.linuxjournal.com/article/6735>.

10. Linux Kernel Organization. *The Linux Kernel Archives* [online]. neuvedeno , 2009-04-02 [cit. 2009-04-13]. Dostupný z WWW: <http://www.kernel.org/>.

11. *Musical Instrument Digital Interface* [online]. 2009 , 9. 3. 2009 [cit. 2009-04-13]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Musical_Instrument_Digital_Interface>.

12. IWAI, Takashi . *http://www.alsa-project.org/~tiwai/lk2k/archtect.gif* [online]. 25-Sep-2000 [cit. 2009-03-13]. Dostupný z WWW: <http://www.alsa-project.org/~tiwai/lk2k/archtect.gif>.

13. IWAI , Takashi. *Image:Alsa basic structure.jpg* [online]. 2007 [cit. 2009-04-13]. Dostupný z WWW: <http://en.opensuse.org/Image:Alsa_basic_structure.jpg>.

14. PETERKA, Jiří. *Počítačové sítě : Lekce č. 4: Rodina protokolů TCP/IP* [online]. v. 3.3. Praha : Univerzita Karlova, 2007 [cit. 2009-04-13]. Dostupný z WWW: < http://www.earchiv.cz/l218/slide.php3?l=4&me=1>.

15. *GNU ccRTP* [online]. 2006 [cit. 2009-04-14]. Dostupný z WWW: <http://www.gnu.org/software/ccrtp/>.

16. *Voip-Info.org* [online]. 2003-2008 [cit. 2009-04-14]. Dostupný z WWW: <http://www.voip-info.org/>.

17. BANERJEE, K.. *Introduction to Internet Multimedia* [online]. 2005 , March 22, 2006 [cit. 2009-04-14]. Dostupný z WWW: <http://geocities.com/intro_to_multimedia/index.html>.

18. *Session Initiation Protocol* [online]. 2009 [cit. 2009-04-14]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Session_Initiation_Protocol>.

19. MATTHEW, Neil , STONES, Richard . *Beginning Linux Programming : Second Edition*. 2nd edition. [s.l.] : Wrox Press Ltd., 2000. 804 s. ISBN 1−861002−97−1.

20. MITCHELL, Mark , SAMUEL, Alex , OLDHAM, Jeffrey . *Advanced Linux Programming*. [s.l.] : New Riders Publishing, 2001. 368 s. ISBN 0-7357-1043-0.

21. LAUTRBACH, Petr. *Jádro systému* [online]. neuvedeno [cit. 2009-04-14]. Dostupný z WWW: <http://www.fi.muni.cz/~kas/p090/referaty/2009-jaro/st/kernel.html>.

22. Gentoo Foundation, Inc.. *Instalace ALSA pro Gentoo* [online]. neuvedeno [cit. 2009-04-14]. Dostupný z WWW: <http://www.gentoo.org/doc/cs/alsa-guide.xml>.

23. KRÁTKÝ, Robert. *Na co se často ptáme: ALSA - II : Konfigurace pomocí souboru .asoundrc*. Použití pluginů. Mixování zvukových proudů z více zdrojů. [online]. 22. 12. 2004 [cit. 2009-04-14]. Dostupný z WWW: <http://www.abclinuxu.cz/clanky/multimedia/na-co-se-casto-ptame-alsa-ii>.

24. *D/A převodník* [online]. 14. 3. 2009 [cit. 2009-04-20]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Soubor:Sampled.signal.svg>.

25. *Open Source Line Echo Canceller (OSLEC)* [online]. neuvedeno , 14-Apr-2009 [cit. 2009-04-20]. Dostupný z WWW: <http://www.rowetel.com/ucasterisk/oslec.html>.

26. *C++ Reference* [online]. neuvedeno [cit. 2009-04-26]. Dostupný z WWW: <http://www.cppreference.com/wiki/>.

27. Cplusplus.com. *Cplusplus.com* [online]. v2.2.1. 2000-2009 [cit. 2009-04-26]. Dostupný z WWW: <http://www.cplusplus.com/>.

28. KEGEL, Dan. *Building and Testing gcc/glibc cross toolchains* [online]. 2003 , 7 Dec 2006 [cit. 2009-04-26]. Dostupný z WWW: <http://kegel.com/crosstool/>.

29. MSC Vertriebs GmbH. *MSC EXM32-Au1200 Linux Guide*. [s.l.] : [s.n.], last update 31-may-2007. 21 s.

30. MSC Vertriebs GmbH. *Embedded Computer Modules/COMs : EXM32 Modules* [online]. neuvedeno [cit. 2009-05-01]. Dostupný z WWW: <http://www.msc-ge.com/frame/en/produkte/com/exm32/module_overview.html>.

31. IPPOLITO, Greg . *POSIX thread (pthread) libraries* [online]. 2002, 2003, 2004 [cit. 2009-05-01]. Dostupný z WWW: <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>.

32. PODHOLA , Martin . *Signály a procesy 2* [online]. Středa, 14. prosinec 2005 [cit. 2009-05-09]. Dostupný z WWW: <http://www.linuxexpres.cz/praxe/signaly-a-procesy-2>.

33. Nokia Corporation. *Qt Reference Documentation* [online]. 2009 [cit. 2009-05-09]. Dostupný z WWW: <http://doc.trolltech.com/4.5/index.html>.

34. MSC Vertriebs GmbH. *MSC EXM32 AU1200 WINDOWS CE Starter Kit* [online]. 2008 [cit. 2009-05-12]. Dostupný z WWW: <http://www.msc-toolguide.com/http-www-msc-ge-com-frame-en-379-www/msc-exm32-sh7760-qnx-starter-kit-1.html>.

35. Linphone.org. *Linphone* [online]. 2007 [cit. 2009-05-12]. Dostupný z WWW: <http://www.linphone.org/>.

36. *Ekiga* [online]. neuvedeno [cit. 2005-05-12]. Dostupný z WWW: <http://www.gnomemeeting.org/>.

37. TeamSpeak Systems GmbH. *TeamSpeak* [online]. neuvedeno [cit. 2009-05-12]. Dostupný z WWW: <http://www.teamspeak.com/>.

38. Skype. *Skype* [online]. 2009 [cit. 2009-05-12]. Dostupný z WWW: <http://skype.com/intl/cs/>.

Příloha A – soubor gsm.h použitelný v jazyce C++

```
/*
 * Copyright 1992 by Jutta Degener and Carsten Bormann, Technische
 * Universitaet Berlin.  See the accompanying file "COPYRIGHT" for
 * details.  THERE IS ABSOLUTELY NO WARRANTY FOR THIS SOFTWARE.
 */

#ifndef	GSM_H
#define	GSM_H

#ifdef __cplusplus
extern "C" {
#endif

#ifdef __cplusplus
#	define	NeedFunctionPrototypes	1
#endif

#if __STDC__
#	define	NeedFunctionPrototypes	1
#endif

#ifdef _NO_PROTO
#	undef	NeedFunctionPrototypes
#endif

#ifdef NeedFunctionPrototypes
#   include	<stdio.h>		/* for FILE * 	*/
#endif

#undef GSM_P
#if NeedFunctionPrototypes
#	define	GSM_P( protos )	protos
#else
#	define	GSM_P( protos )	( /* protos */ )
#endif

/*
 *	Interface
 */

typedef struct gsm_state *	gsm;
typedef short		  	gsm_signal;		/* signed 16 bit */
typedef unsigned char		gsm_byte;
typedef gsm_byte 		gsm_frame[33];		/* 33 * 8 bits	 */

#define	GSM_MAGIC		0xD		  	/* 13 kbit/s RPE-LTP */

#define	GSM_PATCHLEVEL		10
#define	GSM_MINOR		0
#define	GSM_MAJOR		1

#define	GSM_OPT_VERBOSE		1
#define	GSM_OPT_FAST		2
#define	GSM_OPT_LTP_CUT		3
#define	GSM_OPT_WAV49		4
#define	GSM_OPT_FRAME_INDEX	5
#define	GSM_OPT_FRAME_CHAIN	6
```

```c
extern gsm  gsm_create    GSM_P((void));
extern void gsm_destroy GSM_P((gsm));

extern int  gsm_print   GSM_P((FILE *, gsm, gsm_byte  *));
extern int  gsm_option  GSM_P((gsm, int, int *));

extern void gsm_encode  GSM_P((gsm, gsm_signal *, gsm_byte  *));
extern int  gsm_decode  GSM_P((gsm, gsm_byte   *, gsm_signal *));

extern int  gsm_explode GSM_P((gsm, gsm_byte   *, gsm_signal *));
extern void gsm_implode GSM_P((gsm, gsm_signal *, gsm_byte   *));

#undef GSM_P

#ifdef __cplusplus
} /* closing brace for extern "C" */
#endif

#endif /* GSM_H */
```

Příloha B – UDP komunikace (příjem dat)

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
	int sockfd;//file descriptor soketu
	//struktrura pro ulozeni na ktere se bude prijimat
	sockaddr_in my_address;
	//vytvoreni (otevreni) socketu
	sockfd = socket(AF_INET, SOCK_DGRAM, 0);
	if (sockfd == -1)
	{
		printf("socket() error: %s\n", strerror(errno));
		return -1;
	}
	//nastaveni adresy, na ktere se bude prijimat
	my_address.sin_family = AF_INET;
	my_address.sin_addr.s_addr = htonl(INADDR_ANY);
	my_address.sin_port = htons(2626);
	//prirazeni adresy soketu
	int rc = bind(sockfd, (struct sockaddr *)&my_address,
	                                      sizeof(my_address));
	if (rc == -1)
	{
		printf("connect() error: %s\n", strerror(errno));
		return -2;
	}
	//prijmuti a vypsani pokusneho retezce
	char buff[256];
	memset(buff, '\0', 256);
	rc = recv(sockfd, buff, sizeof(buff), 0);
	if (rc == -1)
	{
		printf("recv() error: %s\n", strerror(errno));
		return -3;
	}
	printf("Prijato: %s", buff);
	//ukonceni (zavreni soketu)
	close(sockfd);

	return 0;
}
```

Příloha B – UDP komunikace (odeslání dat)

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
	int sockfd;//file descriptor soketu
	sockaddr_in server_address;//struktrura pro ulozeni adresy
	//vytvoreni (otevreni) socketu
	sockfd = socket(AF_INET, SOCK_DGRAM, 0);
	if (sockfd == -1)
	{
		printf("socket() error: %s\n", strerror(errno));
		return -1;
	}
	//nastaveni adresy
	server_address.sin_family = AF_INET;
	server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
	server_address.sin_port = htons(2626);
	//specifikovani kam mam tento soket defaultne zasilat data
	int rc = connect(sockfd, (struct sockaddr *)&server_address,
	                                            sizeof(server_address));
	if (rc == -1)
	{
		printf("connect() error: %s\n", strerror(errno));
		return -2;
	}
	//odeslani pokusneho textu
	const char* str = "Ahoj";
	rc = send(sockfd, str, strlen(str), 0);
	if (rc == -1)
	{
		printf("send() error: %s\n", strerror(errno));
		return -3;
	}
	//ukonceni (zavreni soketu)
	close(sockfd);
	printf("Odeslani probehlo uspesne\n");

	return 0;
   }
```

Příloha C – přehrání náhodného zvuku

```
#include <stdio.h>
#include <stdlib.h>
#include <alsa/asoundlib.h>
#include <limits.h>

int main(int argc, char *argv[])
{
	int err;
	/* handle zarizeni */
	snd_pcm_t *pcm_handle = NULL;
	/* nazev zarizeni */
	const char *device_name = "default";
	/* otevreni zarizeni */
	err = snd_pcm_open(&pcm_handle, device_name,
	                                     SND_PCM_STREAM_PLAYBACK, 0);
	/* kontrola chyby */
	if (err < 0)
	{
		printf ("cannot open audio device %s (%s)\n", device_name,
		                                     snd_strerror (err));
		pcm_handle = NULL;
		return 0;
	}
	/* nastaveni HW parametru */
	snd_pcm_hw_params_t *hw_params;
	snd_pcm_hw_params_malloc (&hw_params);
	snd_pcm_hw_params_any (pcm_handle, hw_params);
	snd_pcm_hw_params_set_access (pcm_handle, hw_params,
	                                SND_PCM_ACCESS_RW_INTERLEAVED);
	snd_pcm_hw_params_set_format (pcm_handle, hw_params,
	                                SND_PCM_FORMAT_S16_LE);
	unsigned int rrate= 44100;
	snd_pcm_hw_params_set_rate_near (pcm_handle, hw_params, &rrate, NULL);
	snd_pcm_hw_params_set_channels (pcm_handle, hw_params, 2);
	snd_pcm_uframes_t period_size= 940;
	snd_pcm_hw_params_set_period_size_near (pcm_handle, hw_params,
	                                      &period_size, NULL);
	snd_pcm_uframes_t buffer_size = 8*period_size;
	snd_pcm_hw_params_set_buffer_size_near (pcm_handle, hw_params,
	                                      &buffer_size);
	err = snd_pcm_hw_params (pcm_handle, hw_params);
	snd_pcm_hw_params_get_period_size(hw_params, &period_size, 0);
	snd_pcm_hw_params_get_buffer_size(hw_params, &buffer_size);
	unsigned int period_time;
	snd_pcm_hw_params_get_period_time(hw_params, &period_time, 0);
	snd_pcm_hw_params_free (hw_params);
	if (err < 0)
	{
		printf ("cannot set hw parames (%s)\n", snd_strerror (err));
		pcm_handle = NULL;
		return 0;
	}
	/* nastaveni zakladnich SW parametru */
	snd_pcm_sw_params_t *sw_params;
	snd_pcm_sw_params_malloc (&sw_params);
	snd_pcm_sw_params_current (pcm_handle, sw_params);
```

```
        snd_pcm_sw_params_set_start_threshold(pcm_handle, sw_params,
                                                 buffer_size-period_size);
        snd_pcm_sw_params_set_avail_min(pcm_handle, sw_params, period_size);
        snd_pcm_sw_params(pcm_handle, sw_params);
        snd_pcm_sw_params_free (sw_params);
        /* zobrazeni infa */
        printf("Set rate: %u Hz, period size: %lu frames, period time: %u us,
               buffer size: %lu frames\n", rrate, period_size, period_time,
                                                       buffer_size);
        /* buffer pro jednu periodu */
        short playBuffer[period_size];
        /* vypocet poctu opakovani, prehrani jedne periody */
        int loops= (5*1000000)/ period_time;
        printf("Playing random sound for 5 sec...\n");
        int i;
        /* vygenerovani jedne periody */
        srand(time(NULL));
        for(i=0; i<period_size; i++)
        {
               playBuffer[i]= rand()%SHRT_MAX/2;
               if (rand()%2==0) playBuffer[i]= -playBuffer[i];
        }
        /* prehravani po dobu 5 sec */
        for(i=0; i<loops; i++)
        {
               err= snd_pcm_writei(pcm_handle, playBuffer, period_size);
               if (err == -EPIPE)
               {
                      snd_pcm_prepare(pcm_handle);
                      printf("underrun\n");
               }else if (err < 0)
               {
                      snd_pcm_prepare(pcm_handle);
                      printf("unknow error\n");
               }
        }
        /* stop alsa */
        printf("Stop playing...\n");
        snd_pcm_drain(pcm_handle);
        snd_pcm_close(pcm_handle);

        return 0;
}
```

Příloha D – přehrání náhodného zvuku pomocí registrované callback funkce

```
#include <stdio.h>
#include <stdlib.h>
#include <alsa/asoundlib.h>
#include <limits.h>

snd_pcm_uframes_t period_size= 940;

//callback funkce pro prehravani
void playCallback(snd_async_handler_t *pcm_callback)
{
	snd_pcm_t *pcm_handle= snd_async_handler_get_pcm(pcm_callback);
	short *playBuffer= (short*)
			snd_async_handler_get_callback_private(pcm_callback);
	snd_pcm_sframes_t avail;
	//uprava pozice ukazatelu bufferu
	avail = snd_pcm_avail_update(pcm_handle);
	if (avail >= period_size)
	{
		snd_pcm_writei(pcm_handle, playBuffer, period_size);
		//uprava pozice ukazatelu bufferu
		avail = snd_pcm_avail_update(pcm_handle);
	}
}

int main(int argc, char *argv[])
{
	int err;
	/* handle zarizeni */
	snd_pcm_t *pcm_handle = NULL;
	/* nazev zarizeni */
	const char *device_name = "default";
	/* otevreni zarizeni */
	err = snd_pcm_open(&pcm_handle, device_name,
					SND_PCM_STREAM_PLAYBACK, 0);
	/* kontrola chyby */
	if (err < 0)
	{
		printf ("cannot open audio device %s (%s)\n", device_name,
						snd_strerror (err));
		pcm_handle = NULL;
		return 0;
	}
	/* nastaveni HW parametru */
	snd_pcm_hw_params_t *hw_params;
	snd_pcm_hw_params_malloc (&hw_params);
	snd_pcm_hw_params_any (pcm_handle, hw_params);
	snd_pcm_hw_params_set_access (pcm_handle, hw_params,
						SND_PCM_ACCESS_RW_INTERLEAVED);
	snd_pcm_hw_params_set_format (pcm_handle, hw_params,
						SND_PCM_FORMAT_S16_LE);
	unsigned int rrate= 44100;
	snd_pcm_hw_params_set_rate_near (pcm_handle, hw_params, &rrate, NULL);
	snd_pcm_hw_params_set_channels (pcm_handle, hw_params, 2);
```

```
        snd_pcm_hw_params_set_period_size_near (pcm_handle, hw_params,
                                                 &period_size, NULL);
        snd_pcm_uframes_t buffer_size = 8*period_size;
        snd_pcm_hw_params_set_buffer_size_near (pcm_handle, hw_params,
                                                 &buffer_size);
        err = snd_pcm_hw_params (pcm_handle, hw_params);
        snd_pcm_hw_params_get_period_size(hw_params, &period_size, 0);
        snd_pcm_hw_params_get_buffer_size(hw_params, &buffer_size);
        unsigned int period_time;
        snd_pcm_hw_params_get_period_time(hw_params, &period_time, 0);
        snd_pcm_hw_params_free (hw_params);
        if (err < 0) {
              printf ("cannot set hw parames (%s)\n", snd_strerror (err));
              pcm_handle = NULL;
              return 0;
        }
        /* nastaveni zakladnich SW parametru */
        snd_pcm_sw_params_t *sw_params;
        snd_pcm_sw_params_malloc (&sw_params);
        snd_pcm_sw_params_current (pcm_handle, sw_params);
        snd_pcm_sw_params_set_start_threshold(pcm_handle, sw_params,
                                                 buffer_size-period_size);
        snd_pcm_sw_params_set_avail_min(pcm_handle, sw_params, period_size);
        snd_pcm_sw_params(pcm_handle, sw_params);
        snd_pcm_sw_params_free (sw_params);
        /* zobrazeni infa */
        printf("Set rate: %u Hz, period size: %lu frames, period time: %u us,
              buffer size: %lu frames\n", rrate, period_size, period_time,
                                                 buffer_size);
        /* buffer pro jednu periodu a jeho inicializace */
        short playBuffer[period_size];
        memset(playBuffer, 0, period_size*sizeof(short));
        /* priprava zarizeni */
        snd_pcm_prepare(pcm_handle);
        /* zapsani inicializacnich dat*/
        snd_pcm_writei(pcm_handle, playBuffer, period_size);
        int i;
        /* vygenerovani jedne periody */
        srand(time(NULL));
        for(i=0; i<period_size; i++){
              playBuffer[i]= rand()%SHRT_MAX/2;
              if (rand()%2==0) playBuffer[i]= -playBuffer[i];
        }
        /* registrace callback funkce */
        snd_async_handler_t *pcm_callback;
        snd_async_add_pcm_handler(&pcm_callback, pcm_handle, playCallback,
                                                               playBuffer);
        /* zacatek prehravani */
        snd_pcm_start(pcm_handle);
        printf("Playing (any key + enter for exit)...\n");
        getchar();
        printf("Stop playing...\n");
        /*stop playback*/
        snd_async_del_handler(pcm_callback);
        snd_pcm_drop(pcm_handle);
        snd_pcm_close(pcm_handle);

        return 0;
}
```

Příloha E – přehrávání sinusoidy

```
/**********************************************
Prehrava sinusoidu po dobu 10 sec. Lze nastavit
rate, pocet period a velikost periody ve framech
***********************************************/
#include <stdio.h>
#include <stdlib.h>
#include <alsa/asoundlib.h>
#include <limits.h>
#include <math.h>

void genSin(short* buffer, int buffer_size, double* phase, double phase_size)
{
	double valueRad= (*phase);
	double stepRad= (phase_size != M_PI) ? phase_size : M_PI+0.89;

	int i;
	for(i=0; i<buffer_size; i++)
	{
		double tmp= (sin(valueRad) * SHRT_MAX/4);
		*(buffer+i)= (short)(sin(valueRad) * SHRT_MAX/4);
		valueRad += stepRad;
	}
	*phase= valueRad;
}

int main(int argc, char *argv[])
{
	unsigned int rrate = 48000;
	snd_pcm_uframes_t period_size = 0;
	int periods = 0;

	if (argc > 2)
	{
		if (strcmp(argv[1],"-r")==0)
			rrate= atoi(argv[2]);
		else if (strcmp(argv[1],"-psize")==0)
			period_size= atoi(argv[2]);
		else if (strcmp(argv[1],"-pcount")==0)
			periods= atoi(argv[2]);

		if (argc > 3)
		{
			if (strcmp(argv[3],"-r")==0)
				rrate= atoi(argv[4]);
			else if (strcmp(argv[3],"-psize")==0)
				period_size= atoi(argv[4]);
			else if (strcmp(argv[3],"-pcount")==0)
				periods= atoi(argv[4]);

			if (argc > 5)
			{
				if (strcmp(argv[5],"-r")==0)
					rrate= atoi(argv[6]);
				else if (strcmp(argv[5],"-psize")==0)
					period_size= atoi(argv[6]);
				else if (strcmp(argv[5],"-pcount")==0)
					periods= atoi(argv[6]);
```

```
            }
        }
}else printf("Test ALSA sin (-r rate, -psize period size, -pcount
                                            buffer periods)\n\n");

if (argc > 2)
        printf("Engaged rate: %i Hz, period size: %i frames, buffer
                                 periods: %i\n", rrate, period_size, periods);
else printf("Engaged rate: %i Hz, period size: auto, buffer periods:
                                                    auto\n", rrate);

int err;
/* handle zarizeni */
snd_pcm_t *pcm_handle = NULL;
/* nazev zarizeni */
const char *device_name = "default";
/* otevreni zarizeni */
err = snd_pcm_open (&pcm_handle, device_name,
                                      SND_PCM_STREAM_PLAYBACK, 0);
/* kontrola chyby */
if (err < 0)
{
        printf ("cannot open audio device %s (%s)\n", device_name,
                                            snd_strerror (err));
        pcm_handle = NULL;
        return 0;
}
/* nastaveni HW parametru */
snd_pcm_hw_params_t *hw_params;
snd_pcm_hw_params_malloc (&hw_params);
snd_pcm_hw_params_any (pcm_handle, hw_params);
snd_pcm_hw_params_set_access (pcm_handle, hw_params,
                                      SND_PCM_ACCESS_RW_INTERLEAVED);
snd_pcm_hw_params_set_format (pcm_handle, hw_params,
                                      SND_PCM_FORMAT_S16_LE);
snd_pcm_hw_params_set_rate_near (pcm_handle, hw_params, &rrate, NULL);
snd_pcm_hw_params_set_channels (pcm_handle, hw_params, 1);
snd_pcm_hw_params_set_period_size_near (pcm_handle, hw_params,
                                      &period_size, NULL);
snd_pcm_uframes_t buffer_size = periods*period_size;
snd_pcm_hw_params_set_buffer_size_near (pcm_handle, hw_params,
                                      &buffer_size);
unsigned int period_time;
snd_pcm_hw_params_get_period_time(hw_params, &period_time, 0);
err = snd_pcm_hw_params (pcm_handle, hw_params);
snd_pcm_hw_params_free (hw_params);
if (err < 0)
{
        printf ("cannot set hw parames (%s)\n", snd_strerror (err));
        pcm_handle = NULL;
        return 0;
}
/* nastaveni zakladnich SW parametru */
snd_pcm_sw_params_t *sw_params;
snd_pcm_sw_params_malloc (&sw_params);
snd_pcm_sw_params_current (pcm_handle, sw_params);
snd_pcm_sw_params_set_start_threshold(pcm_handle, sw_params,
                                            buffer_size-period_size);
snd_pcm_sw_params_set_avail_min(pcm_handle, sw_params, period_size);
```

```
	snd_pcm_sw_params(pcm_handle, sw_params);
	snd_pcm_sw_params_free (sw_params);
	/* zobrazeni infa */
	printf("Set rate: %i Hz, period size: %i frames, period time: %i us,
		buffer size: %i frames\n", rrate, period_size, period_time,
								buffer_size);
	/* buffer pro jednu periodu */
	short playBuffer[period_size];
	memset(playBuffer, 0, period_size*sizeof(short));
	/* inicializace parametru pro sinusoidu */
	double phase= 0;
	int step= 64;
	while(period_size%step==0) step+= 8;
	int parts= period_size/step;
	int mod= period_size%step;
	double phase_size= (2*M_PI)/step;
	printf("Sins per period: %i\n", parts);
	printf("Phase size: %f\n", phase_size);
	printf("Step: %i Mod: %i\n", step, mod);
	/* pocet period za 10 sec */
	int loops=(10 * 1000000)/period_time;
	/* buffer na 10 sec */
	short sinBuf[period_size*loops];
	/* vygenerovani 10 sec */
	printf("Generating sin for 10 sec...\n");
	int i;
	for(i=0; i<loops; i++)
	{
		/* vygenerovani jedne periody */
		int j;
		for(j=0; j<parts; j++)
			genSin(sinBuf+(i*period_size)+(j*step), step, &phase,
									phase_size);
		genSin(sinBuf+(i*period_size)+(j*step), mod, &phase,
									phase_size);
	}
	/* prehrani vygenerovane sinusoidy */
	printf("Playing sin for 10 sec...\n");
	for(i=0; i<loops; i++)
	{
		memcpy(playBuffer, sinBuf+(i*period_size),
							period_size*sizeof(short));
		err= snd_pcm_writei(pcm_handle, playBuffer, period_size);
		if (err == -EPIPE)
		{
			snd_pcm_prepare(pcm_handle);
			printf("underrun\n");
		}else if (err < 0) printf("unknow error\n");
	}
	printf("Stop playing...\n");
	snd_pcm_drop(pcm_handle);
	snd_pcm_close(pcm_handle);

	return 0;
}
```

Příloha F – ukázka použití a synchronizace vláken

```
/* dve vlakna soucasne inkrementuji do ITERATIONS */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define ITERATIONS 10000000

volatile int count = 0;
pthread_mutex_t mutex;

/* ceka do zmeny casu na novou sekundu -- nejvyse jednu sekundu */
static void wait_thread(void)
{
    time_t start_time = time(NULL);

    while (time(NULL) == start_time)
    {
        /* do nothing except chew CPU slices for up to one second */
    }
}

void *ThreadAdd(void *a)
{
      int i, tmp;
      /*cekani na preklopeni casu na novou sekundu, vlakna zacnou soucasne*/
      wait_thread();

      for (i = 0; i < ITERATIONS; i++) {
            /* uzamknuti kriticke sekce mutexem,
            bude provadet vzde jen jedno vlakno*/
            pthread_mutex_lock(&mutex);

            tmp = count; /* copy the global count locally */
            tmp = tmp + 1;      /* increment the local copy */
            count = tmp; /* store the local value into the global count */

            /* odemknuti kriticke sekce */
            pthread_mutex_unlock(&mutex);
      }
      return 0;
}

int main(int argc, char *argv[])
{
      pthread_t tid1, tid2;

      /* inicializace mutexu */
      pthread_mutex_init(&mutex, NULL);

      /* spusteni dvou vlaken */
      if (pthread_create(&tid1, NULL, ThreadAdd, NULL)) {
            fprintf(stderr, "ERROR creating thread 1\n");
            return EXIT_FAILURE;
      }
      if (pthread_create(&tid2, NULL, ThreadAdd, NULL)) {
            fprintf(stderr, "ERROR creating thread 2\n");
```

```
        return EXIT_FAILURE;
    }

    /* cekani na dokonceni vlaken */
    if (pthread_join(tid1, NULL)) { /* wait for the thread 1 to finish */
        fprintf(stderr, "ERROR joining thread 1\n");
        return EXIT_FAILURE;
    }
    if (pthread_join(tid2, NULL)) { /* wait for the thread 2 to finish */
        fprintf(stderr, "ERROR joining thread 2\n");
        return EXIT_FAILURE;
    }

    /* kontrola vysledku */
    if (count < 2 * ITERATIONS)
        fprintf(stderr, "BOOM! count is %d, should be %d\n", count,
                2 * ITERATIONS);
    else
        printf("OK! count is %d\n", count);

    /* zniceni mutexu */
    pthread_mutex_destroy(&mutex);

    return 0;
}
```