VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KNIHOVNA PRO PRÁCI S VIDEEM PRO PLATFORMU ANDROID

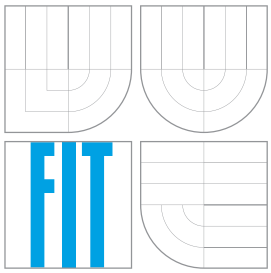
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ SLAVOTÍNEK

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KNIHOVNA PRO PRÁCI S VIDEEM PRO PLATFORMU ANDROID

VIDEO LIBRARY FOR ANDROID PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ SLAVOTÍNEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ALEŠ LÁNÍK

BRNO 2014

Abstrakt

Práce se zabývá zpracováním videa na platformě Android. Konzultovány jsou knihovny pro kódování a dekódování multimediálního obsahu, které lze na této platformě použít. Součástí je návrh a implementace knihovny, která bude takový nativní kodek – FFmpeg – zastřešovat a poskytovat multimediálním Java aplikacím jednoduché a efektivní rozhraní.

Abstract

This thesis deals with processing of video content on Android platform. Libraries for video encoding and decoding usable on this platform are described first. One of these libraries – FFmpeg – is used as core to design and implement middleware video-processing library. This library will provide simple but effective API for multimedia Java applications.

Klíčová slova

kódování videa a zvuku, dekódování videa a zvuku, Android, JNI, FFmpeg

Keywords

video and audio encoding, video and audio decoding, Android, JNI, FFmpeg

Citace

Tomáš Slavotínek: Knihovna pro práci s videem pro platformu Android, diplomová práce, Brno, FIT VUT v Brně, 2014

Knihovna pro práci s videem pro platformu Android

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Aleše Láníka.

.....
Tomáš Slavotínek
28. května 2014

Poděkování

Touto formou děkuji vedoucímu Ing. Aleši Láníkovi za poskytnuté rady a podporu při tvorbě této práce.

Obsah

Kapitola 1

Úvod

Oblast pořizování a zpracování multimediálního obsahu doznává, stejně jako jiné oblasti informatiky a elektrotechniky, prudký rozvoj. V domácích „amatérských“ podmínkách jsme tak schopni pořizovat, zpracovávat a prostřednictvím internetu zveřejňovat pohyblivý obraz vysoké kvality doplněný několikakanálovým zvukem. To vše v případě potřeby i v reálném čase.

K pořízení audiovizuálního záznamu typicky použijeme zařízení navržené speciálně k tomuto účelu – např. videokameru, fotoaparát, webovou kameru či software pro snímání obrazovky počítače. Tento záznam můžeme na běžném osobním počítači upravovat, stříhat, slučovat s jinými záznamy apod. Hotový produkt, pokud je určen pro veřejnost, je možné zveřejnit na některém z video portálů nebo rovnou v reálném čase streamovat.

Většina z nás u sebe nosí zařízení, které je dostatečné vybavené pro vykonání všech výše uvedených kroků. Může se jednat o chytrý mobilní telefon, tablet nebo jinou formu kapesního či přenosného počítače. Tyto systémy bývají vybaveny poměrně kvalitním snímačem pro záznam statického i pohyblivého obrazu a zvuku, výkonným často vícejádrovým procesorem, operační pamětí s kapacitou dosahující až několika gigabytů, trvalým úložištěm s kapacitou o řád větší a několika technologickými řešeními pro připojení k internetu.

Tato práce se zabývá možnostmi zpracování videa na zařízeních spadajících do této kategorie – zařízeních vybavených systémem Android. Práce si klade za cíl prozkoumat oblast existujících knihoven pro zpracování videa vhodných pro platformu Android a navrhnout zastřešující rozhraní pro multimediální (Java) aplikace. Součástí práce je rovněž zhodnocení výsledků a výkonnosti výsledného řešení na různém hardware. Od čtenáře se očekává znalost programovacích jazyků Java a C (případně C++).

Výhodou jsou rovněž základní znalosti z oblasti vývoje aplikací pro platformu Android, práce však v patřičných kapitolách poskytuje čtenáři odkazy na literaturu a jiné zdroje, které je možné použít k nastudování dané problematiky.

Kapitola 2

Platforma Android

Android [10] je operační systém založený na Linuxovém jádře, který je určený zejména pro mobilní zařízení jako jsou chytré telefony a tablety, ale i pro televizní přijímače, herní konzole a jinou elektroniku.

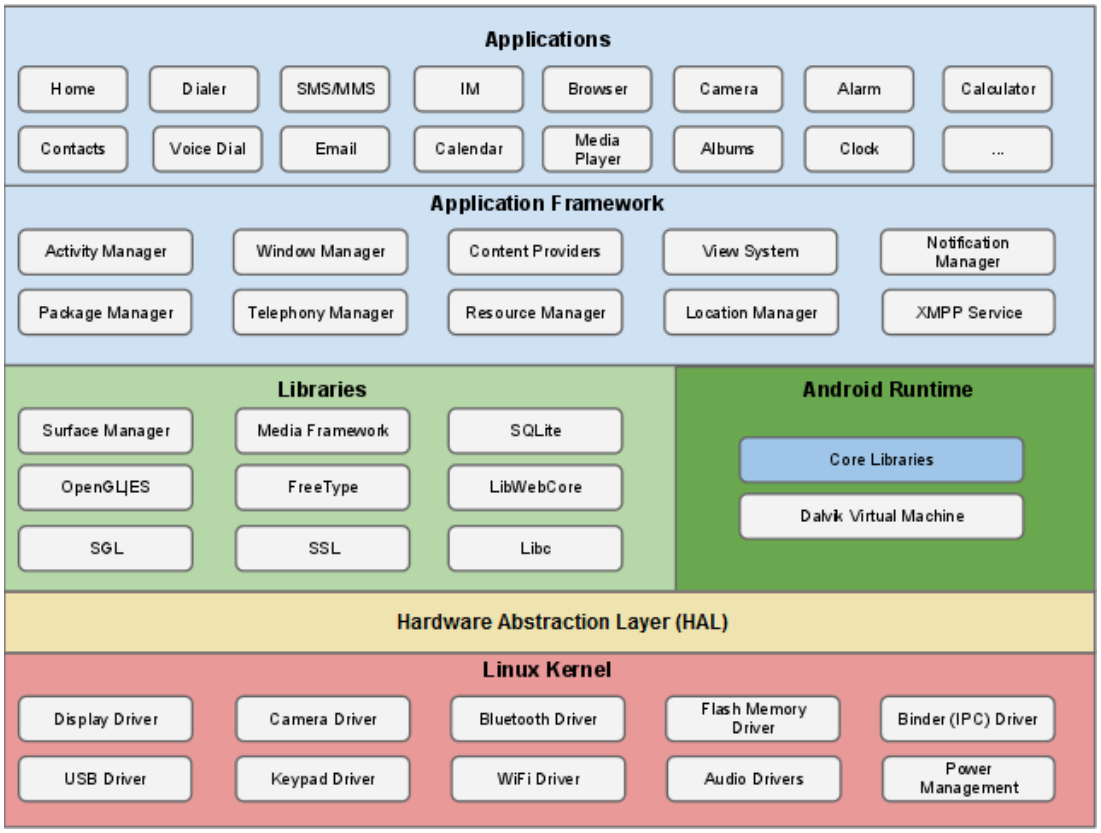
Historií platformy Android a jejím popisem z pohledu vývojáře se zabývá řada publikací a jiných zdrojů — například Allen Grant (2013) [28] nebo Miroslav Ujbányai (2012) [29]. V následujícím textu se tedy budeme zabývat pouze oblastmi nezbytnými pro výklad pozdějších pasáží práce.

2.1 Úvod do platformy Android

Obrázek 2.1 zachycuje architekturu platformy Android ve velmi zjednodušené podobě. Přítomny jsou však všechny základní vrstvy a komponenty systému. Jak již bylo řečeno, základem systému je Linuxové jádro — pro starší verze systému předcházející Android 4.0 (*Ice Cream Sandwich*; ICS) to byl kernel 2.6, pro současné verze je to pak kernel 3.4 (stav z roku 2013). Mimo základní služby operačního systému poskytuje kernel také pokročilou správu napájení a obsahuje ovladače pro jednotlivé komponenty cílového zařízení (jako například ovladač displeje, zvuku, kamery apod.).

Mezi jádrem a vyššími úrovněmi architektury se nachází vrstva *Hardware Abstraction Layer* (HAL). Tato vrstva zakrývá specifické metody přístupu k jednotlivým hardwarovým zařízením a navenek poskytuje sadu jednotných aplikačních rozhraní (Application Programming Interface; API). Tato abstrakce umožňuje, aby vyšší vrstvy systému mohly být do značné míry nezávislé na rozdílné hardwarové konfiguraci jednotlivých zařízení.

Nad kernelem a HAL se nachází další nativní vrstva (napsaná v jazyce C) obsahující sadu systémových knihoven a oblast nazvanou *Android Runtime*. Android Runtime zastřešuje kromě knihoven jádra také instance virtuálního stroje *Dalvik Virtual Machine* (DVM), ve kterých potom běží jednotlivé aplikace. Tyto aplikace jsou většinou psány v jazyce Java a následně je vygenerován byte-kód ve formátu Dalvik dex-code. DVM disponuje just-in-time kompilátorem, který se za běhu stará o překlad byte-kódu pro konkrétní cílovou architekturu. Protože mobilní zařízení typicky disponují omezenými prostředky (pamětí, výpočetním výkonem, kapacitou baterie apod.) ve srovnání např. se stolními počítači, byly při návrhu virtuálního stroje uskutečněny změny proti klasickému konceptu. Složitost DVM byla zredukována tak, aby zabíral co nejméně operační paměti i úložného prostoru. Virtuální stroj je registrově orientovaný, na rozdíl od Java Virtual Machine (JVM), který je orientovaný zásobníkově. Tím odpadá režie spojená se zpracováním zásobníkových instrukcí



Obrázek 2.1: Vrstvy platformy Android. [11]

a je možné přímo mapovat lokální proměnné na registry DVM. Cílem tohoto řešení bylo dosáhnout vyššího výkonu interpretu a jednodušší instrukční sady. Při přímém srovnání výkonu JVM se starší verzí DVM se však pozitivní vliv tohoto přístupu nepodařilo prokázat a výsledky mluvily spíše ve prospěch virtuálního stroje Javy.

Pro výpočetně náročné a časově kritické aplikace není zpracovaní interpretovaného jazyka v prostředí virtuálního stroje příliš vhodné, zejména pak na výpočetně limitované mobilní platformě. Z tohoto důvodu je stejně jako v klasickém JVM i v DVM systému Android k dispozici rozhraní *Java Native Interface* (JNI). Toto rozhraní umožňuje psát kritické části aplikace (nebo veškerý výkonný kód) v nativním jazyce — zejména lze použít vyšší programovací jazyk C/C++, ale je možné využít i jazyk symbolických instrukcí (JSI). Mimo to je díky této vlastnosti možné na platformě Android využít řadu veřejně dostupných projektů z nejrůznějších oblastí. Ty jsou nezřídka psány právě v jazyce C či C++ a lze je často použít s minimálními úpravami nebo dokonce zcela bez zásahu do původního projektu.

Nástroji pro vývoj a překlad nativního kódu i kódu psaného v jazyce Java, které jsou dostupné pro platformu Android, se bude zabývat kapitola 2.2.

Postoupíme-li ve schématu z obrázku 2.1 o úroveň výše dostaneme se k vrstvě nazvané *Application Framework*. Jak název napovídá, funkcí této vrstvy je poskytnout podporu – framework pro uživatelské i systémové aplikace. Toho je dosaženo sadou knihoven a jejich API. K dispozici jsou funkce známé z jiných prostředí jako je Window Manager a View System pro správu oken a GUI operací, Activity Manager pro řízení událostí a životního

cyklu aplikace apod. Mimo to je poskytnut přístup ke speciálním fukcím jako je Location Manager pro získání informací o aktuální poloze zařízení. Popis jednotlivých tříd aplikačního frameworku je nad rámec této práce. Podrobné informace lze v případě potřeby získat například z [9]. Kapitola 5 (Návrh) se pouze okrajově zmíní o některých třídách využitých při implementaci knihovny.

V další a již poslední vrstvě se nacházejí samotné aplikace. Může se jednat o software, který je součástí systému daného zařízení (jako např. Home screen, aplikace telefon a zprávy apod.) nebo o uživatelem instalované aplikace získané prostřednictvím služby Google Play nebo z jiného zdroje (typicky v podobě instalačního balíčku apk).

2.2 Vývojářské nástroje

Vývoj aplikací pro platformu Android je podpořen řadou veřejně dostupných nástrojů. Většina z nich je součástí oficiálního balíčku *Software Developement Kit* (SDK). Pro podporu vývoje aplikací v nativním jazyce je navíc k dispozici *Native Developement Kit* (NDK). Součástí prvně jmenovaného balíčku je i vývojové prostředí Android Developer Tools (ADT), které je založeno na projektu Eclipse. Nově (stav z roku 2013) je také dostupné nové prostředí Android Studio, které je od počátku vystavěno specificky pro potřeby vývoje pro Android. V následujících podkapitolách se stručně seznámíme s jednotlivými balíčky a nástroji.

2.2.1 Software Developement Kit (SDK)

Balíček Android SDK obsahuje všechny základní nástroje potřebné pro vývoj aplikací v jazyce Java.

Součástí je komplexní vývojové prostředí Android Developer Tools, což je v podstatě vývojová platforma Eclipse s předinstalovanými ADT rozšířeními pro Android.

Nezbytnou součástí SDK je kompilátor – Android DX Compiler. Ten se stará o převod zkompilovaných Java .class zdrojových kódů na spustitelný `.dex` (Dalvik Executable) formát.

Dále je v balíčku dostupný emulátor mobilních zařízení včetně obrazu generické verze systému Android. Pomocí nástroje Android Virtual Device Manager lze vytvářet a spravovat virtuální zařízení pro emulátor. Navíc je možné instalovat definiční soubory skutečných zařízení, což umožňuje nejen vytvářet virtuální stroje specifických vlastností, ale rovněž je usnadněno nahrávání aplikace do cílového zařízení a její vzdálené ladění. Tyto definiční soubory, které nejsou součástí SDK, jsou dostupné přímo od výrobce konkrétního přístroje.

Kromě výše uvedených součástí disponuje SDK řadou dalších nástrojů, zejména pro ladění kódu, ale např. i pro vytváření roztažitelných grafických prvků apod. Kompletní popis nástrojů dostupných v Android SDK je možné získat z [9].

2.2.2 Native Developement Kit (NDK)

Jak bylo řečeno, Android SDK umožňuje vývoj aplikací psaných v jazyce Java. Pro vývoj aplikací obsahujících nativní kód, je však třeba kromě SDK použít samostatný balíček nazvaný Android NDK.

Ten obsahuje knihovny a hlavičkové soubory nezbytné pro navázání nativního kódu na zbytek aplikace psaný v jazyce Java. Podporována je standardní verze jazyka C a minima-

listická varianta C++ (jsou dostupné pouze základní knihovny a chybí podpora výjimek a RTTI[1]).

Součástí NDK jsou i nástroje nezbytné pro překlad a linkování kódu pro některou z podporovaných architektur. Pro každou architekturu musí existovat tzv. Application Binary Interface (ABI), který přesně definuje, jakým způsobem probíhá interakce mezi vygenerovaným strojovým kódem a cílovým systémem.
V současné verzi NDK (release 9) jsou dostupné ABI pro následující architektury:

- ARM (instrukční sada alespoň ARMv5TE, rozšířená podpora pro ARMv7)
- IA-32/x86 (instrukční sada alespoň Pentium Pro, optimalizace pro Intel Atom)
- MIPS (instrukční sada MIPS32, alespoň revize r1)

Hardwarovým architekturám (ARM) se věnuje samostatná podkapitola 2.4.

Jako hostitelská platforma pro vývoj nativního kódu jsou podporovány operační systémy Microsoft Windows, Linux a Mac OS X. Pod systémem Windows není již potřeba dodatečně instalovat prostředí Cygwin jako tomu bylo u starších verzí Android NDK. Přímo v NDK je obsažena sada nástrojů a skriptů, které umožňují překlad bez nutnosti instalovat další software.

2.3 Nativní aplikace

Aplikace pro Android může být psána jako hybridní, kdy je část programu nativní kód a část je psaná v jazyce Java, nebo se může jednat o zcela nativní program.

2.3.1 Zcela nativní aplikace

Základem každé klasické aplikace pro Android musí být instance třídy `Activity`, která mimo jiné řeší obsluhu vytvoření aplikace, reakce na události apod. Implementace této třídy se normálně nachází v hlavním zdrojovém Java souboru. U zcela nativní aplikace tento soubor a tedy i příslušná Activity zcela chybí.

Tato situace je v Android SDK 9 (Android 2.3) a vyšších vyřešena zavedením speciální třídy `NativeActivity`, jejíž implementace je provedena v nativním (C/C++) kódu.

Z hlediska této práce není tento přístup dobře využitelný, proto se jím nebudeme dále zabývat. Informace k tomuto způsobu psaní aplikací a k `NativeActivity` lze najít v [9].

2.3.2 Hybridní aplikace

Hybridní aplikace obsahuje jak nativní tak Java kód. Jak již bylo řečeno v kapitole 2, v nativním kódu jsou typicky psány výpočetně náročné nebo časově kritické části aplikace. Rovněž lze tímto způsobem do aplikace zahrnout jiný projekt psaný v jazyce C/C++ nebo JSI.

Podpora pro hybridní aplikace je k dispozici již od Android SDK 3 (Android 1.5).

[1] RTTI – Run-Time Type Information, získávání informace o datovém typu za běhu programu

2.3.3 Struktura a konfigurace projektu

Budeme-li respektovat klasickou adresářovou strukturu projektu pro Android, zdrojové kódy nativní části aplikace typicky umisťujeme do podadresáře `jni`. V tomto adresáři je rovněž umístěn soubor `Android.mk`, který definuje způsob, jakým mají být přeloženy jednotlivé nativní části projektu. Výstupem překladu je jeden nebo více modulů (knihoven) ve formátu spustitelném na cílové architektuře. Pro výstupy překladu je automaticky vytvořen adresář `libs` s podadresáři pojmenovanými dle názvu architektury. Dále je vytvořen adresář `obj`, kde se nachází objektové soubory, které jsou meziproduktem vznikajícím při překladu. Soubor `Android.mk` plní podobnou funkci jako klasický soubor `Makefile`. Umožňuje mimo jiné vybrat zdrojové soubory, cesty k hlavičkovým souborům, standardní a jiné statické knihovny a definovat název výsledného modulu (knihovny). Následující text je ukázkou obsahu tohoto souboru s popisem jednotlivých definic:

```
LOCAL_MODULE := mylibrary                # název knihovny
LOCAL_SRC_FILES := mylibrary.c           # zdrojové soubory
LOCAL_C_INCLUDES := $(LOCAL_PATH)/ffmpeg # podadr. se soubory pro inkluzi
LOCAL_STATIC_LIBRARIES := library.so     # vlastní statické knihovny
LOCAL_LDLIBS := -lm -llog                # standardní knihovny
```

V případě komplexnějšího projektu může být vhodné použít kromě souborů `Android.mk` i konfiguraci společnou pro celou nativní část aplikace, která se umisťuje do souboru `Application.mk`. Překlad lze spustit voláním příkazu `ndk-build` (pod systémem Windows je tím vykonán specifický dávkový soubor `ndk-build.cmd`).

2.3.4 Java Native Interface

Java Native Interface (JNI) je rozhraní umožňující navázat nativní kód na kód běžící ve virtuálním stroji JVM. Toto rozhraní je obousměrné – to znamená, že lze z Java aplikace volat nativní kód nebo z nativní aplikace volat kód běžící ve virtuálním stroji. Častější je první případ a i my se zaměříme na tuto variantu.

Aby mohl být kód na obou stranách JNI navzájem provázán, bylo třeba definovat jednoznačná pravidla pro transformaci názvů funkcí, pro práci s datovými typy atd. V následujícím textu si popíšeme základní pravidla platná při použití jazyka C.

V rámci dané Java třídy je třeba uvodit deklaraci nativní metody klíčovým slovem `native`, čímž je specifikováno, že se definice nemá hledat na straně Javy a očekává se, že je implementace obsažena v nativním kódu. V kódu psaném v jazyce C je potom potřeba funkci implementující danou metodu pojmenovat tak, že její název bude složen z názvu balíčku, názvu třídy a nakonec názvu dané metody. Následuje ukázka kódu zachycující situaci na straně jazyka Java:

```
package java.pkg.name;

public class JavaClass
{
  native void MethodName();
}
```

Zde je potom vidět odpovídající funkce na straně nativního kódu v jazyce C:

```
JNIEXPORT void JNICALL java_pkg_name_JavaClass_MethodName( JNIEnv * env,
    jobject obj )
{
```

```
  // Telo funkce
}
```

Vytvoření nativních hlavičkových a zdrojových souborů se správně pojmenovanými funkcemi lze dosáhnout i automaticky pomocí generátorů. Pro jazyk C je to generátor nazvaný `javah`, který je součástí Android NDK. Programu stačí předat jako parametr název třídy, pro kterou se mají soubory generovat.

Podíváme-li se na výše uvedenou ukázku kódu, všimneme si, že funkce na straně jazyka C disponuje parametry, které původní metoda neobsahuje. Tyto dva speciální parametry jsou přítomny pro všechny funkce komunikující přes JNI. První parametr je ukazatel na strukturu `JNIEnv`, která reprezentuje samotné rozhraní JNI. Použití tohoto parametru bude vysvětleno později. Druhý parametr je typu `jobject` a jedná se o ukazatel na vnitřní reprezentaci Java objektu, ke kterému nativní metoda přísluší. Jak je vidět, navíc přibyly direktivy `JNIEXPORT` a `JNICALL`. Ty jsou pro většinu hostitelských prostředí prázdné a nejsou jako takové nezbytně nutné, ale jejich použití se doporučuje. První uvedený zajišťuje, aby došlo ke správnému exportování funkce z modulu a `JNICALL` pak definuje korektní způsob volání takové funkce (definována jako `__stdcall` pro Windows).

Kromě specifického pojmenování funkcí je třeba definovat způsob, jakým se bude zacházet s reprezentací datových typů, které se mezi Javou a nativním jazykem mohou lišit. Pro jazyk C jsou tedy zavedeny ekvivalenty všech jednoduchých datových typů. Jejich jméno je tvořeno jednoduše přidáním písmene „j“ před název daného typu v jazyce Java. Například celočíselný datový typ `int` z Javy je tak v C dostupný jako `jint`. Seznam všech dostupných typů a jejich nativních alternativ je dostupný na [25]. Kromě jednoduchých datových typů je třeba zajistit i dostupnost složených objektů. Toho je dosaženo zajištěním přístupu k vnitřní reprezentaci daného objektu. Aby se předešlo případným konfliktům v přístupu z nativního a interpretovaného kódu nebo přístupu k již neexistujícím objektům je zavedena sada přístupových funkcí, které jsou dostupné přes (již zmíněný) první parametr `JNIEnv` nativní funkce.

Aby byly všechny tyto datové typy, struktury a přístupové funkce pro jazyk C dostupné, je třeba zajistit inkluzi hlavičkového souboru `jni.h`.

Na straně Java kódu se musíme ještě postarat o načtení nativní části aplikace voláním metody `System.loadLibrary()`, které jako parametr předáme název dané knihovny.

Následující ukázka kódu zachycuje kompletní implementaci velmi jednoduché hybridní aplikace. Nejprve Java kód:

```
package java.pkg.name;

public class JavaClass
{
  static
  {
    System.loadLibrary( "NativeMath" );
  }

  native double NativePow( double base, double exp );
}
```

A nyní protějšek na straně jazyka C:

```
#include <jni.h>
#include <cmath>

JNIEXPORT jdouble JNICALL java_pkg_name_JavaClass_NativePow(
  JNIEnv * env, jobject obj, jdouble base, jdouble exp )
{
  return ( jdouble )pow( ( double )base, ( double )exp );
}
```

Je třeba mít na paměti, že podobná jednoduchá nativní implementace, jako v uvedeném příkladu, nemusí být výkonově efektivní, protože komunikace přes JNI a s tím spojené akce znamenají poměrně značnou režii navíc. Tato pak může zcela zastínit výkonnostní výhodu nativního kódu, nebo ještě hůře, způsobit ztrátu proti implementaci v jazyce Java. Dále je třeba uvážit potenciální rizika programování v jazyce jako je C – nutnost manuálního uvolňování paměti, rizika přetypování a práce s ukazateli, což jsou techniky, které mohou snadno vést na chyby a pro programátory pracující výhradně v jazyce Java se mohou jevit nepřirozené.

2.4 Hardware

Doposud jsme se bavili o platformě Android jako o software, avšak abychom mohli tento software provozovat, musíme mít k dispozici i adekvátní hardware. Nemůže se pochopitelně jednat o hardware libovolný a zejména kvůli přítomnosti nativního kódu v implementované knihovně budeme nuceni se jeho specifiky zabývat, abychom byly schopni takový kód přeložit do co možná nejefektivnější podoby.

Tato kapitola proto bude úvodem do vybraných hardwarových architektur, které jsou v současnosti typickým prostředím pro provozování systému Android. Získané znalosti budou použity jako základ v pozdějších pasážích práce, zejména pak v kapitole 6.4, která se prakticky věnuje samotnému překladu jedné z multimediálních knihoven.

Jak již bylo uvedeno v podkapitole 2.2.2 věnované NDK, mezi podporované architektury patří ARM, IA-32/x86 a MIPS. Jedná se o navzájem velmi rozdílná prostředí, kdy každá z architektur existuje v mnoha různých variantách a revizích, o konkrétních modelech procesorů nemluvě. Nicméně na základě poznatků, se kterými bude čtenář seznámen v kapitole 5.2, můžeme zúžit zaměření této práce na architekturu ARM. Na procesorech této architektury jsou totiž založena téměř všechna zařízení cílové skupiny (více viz kapitola 5.2).

Následující podkapitoly čerpají převážně z referenčních manuálů architektur ARM [1], [3], [4], [6] a [7], není-li uvedeno jinak. Pro přístup k uvedeným manuálům může být vyžadováno přihlášení.

2.4.1 Rodina procesorů ARM

Pod označením ARM se skrývá celá rodina architektur mikroprocesorů, s uplatněním především v mobilních, multimediálních a vestavěných zařízeních. Procesory ARM mají v současnosti (2013/2014) dominantní postavení na poli „chytrých“ mobilních telefonů a tabletů, a to nejen těch se systémem Android, ale i přístrojů prodávaných se systémy iOS, Windows Phone a jinými. Tyto procesory nachází své uplatnění i v aplikacích jako jsou přenosné

počítače, herní konzole, digitální televizní přijímače nebo set-top boxy.

Zkratka ARM (*Acorn RISC Machine*) v sobě původně nesla označení společnosti, která za touto architekturou stojí – Acorn Computers, dnes ARM Holdings, výklad zkratky byl však později změněn na *Advanced RISC Machine*. Společnost Acorn začala na návrhu této architektury pracovat v roce 1983 a první mikroprocesor založený na rané verze této architektury (ARMv1) přišel o necelé dva roky později (ARM1, rok 1985). Označení ARM v sobě rovněž ukrývá typ instrukční sady, kterou tyto procesory disponují – RISC (Reduced Instruction Set Computing, redukovaná instrukční sada). RISC oproti CISC (Complex Instruction Set Computing, komplexní instrukční sada) typicky disponuje menším počtem instrukcí a jejich variant, což je částečně kompenzováno větším počtem vnitřních registrů. Takovýto návrh umožňuje velmi efektivní provádění instrukcí bez přítomnosti mikrokódu – instrukce není nutné rozkládat na mikroinstrukce a ty teprve vykonávat, ale je možné, aby funkcionalita instrukcí byla „zadrátována“ přímo v obvodech mikroprocesoru. Tím se rovněž snižuje složitost samotného jádra (počet tranzistorů) a mimo jiné klesá i jeho příkon, což je jedna z vlastností, která činí tyto procesory vhodné pro použití v mobilních a vestavěných systémech.

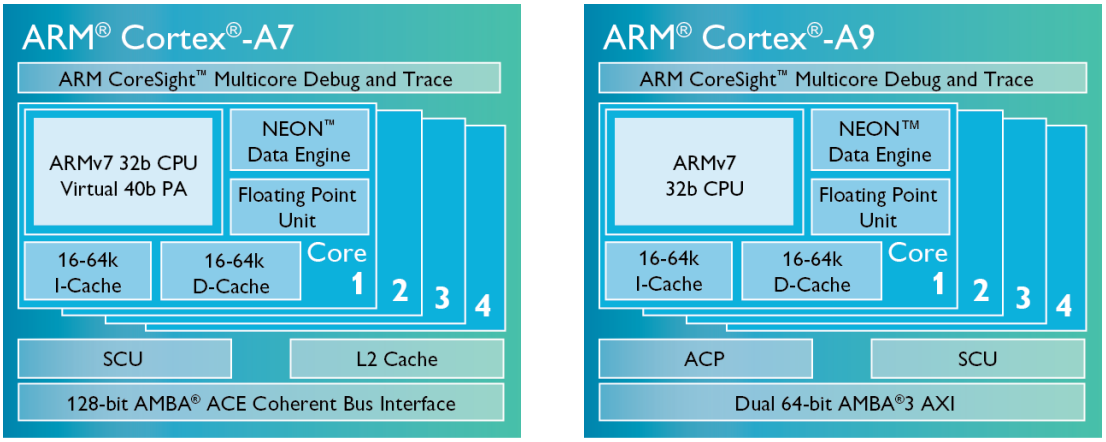
Pomineme-li historické architektury ARMv1 a ARMv2, které byly 32/26 bitové, a teprve nastupující ARMv8-A, která je 64/32 bitová, jsou všechny ostatní ARM procesory 32 bitové. Až do verze ARMv3 byla tato architektura z hlediska rozložení paměti typu *little-endian* (nejméně významný byte se nachází na nejnižší adrese). Pozdější revize jsou typu *bi-endian* a umožňují práci v režimech *little* i *big-endian*. Jak lze očekávat, všechny varianty procesorů rodiny ARM disponují základními typy instrukcí – aritmeticko-logické instrukce pro práci s čísly s pevnou řádovou čárkou, řídící instrukce a instrukce pro přesuny. Co však není u architektur ARM pevně dáno, je přítomnost různých jednotek doplňujících samotné CPU a rozšiřujících jeho schopnosti a výkon. Dostupnost určité jednotky (a její konkrétní varianty) není závislá pouze na revizi architektury ARM, což lze vzhledem k postupnému vývoji očekávat, ale jejich přítomnost se liší i mezi jednotlivými jádry stejné architektury. Je to pochopitelné, jelikož rodina ARM klade důraz na flexibilitu a nabízí výrobcům hardware možnost vytvořit čip „šitý na míru“ konkrétní aplikaci. Spolu s CPU a jeho podpůrnými prostředky může být v jednom čipu integrována i řada jiných komponent a periferií. Hovoříme o takzvaných SoC (System on Chip), kdy prakticky celý počítač je realizován v jednom integrovaném obvodu.

Je tedy patrné, že zařízení založená na procesorech ARM jsou velmi rozmanitým prostředím, což může komplikovat realizaci a nasazení aplikace obsahující nativní kód, který je pochopitelně v přeložené podobě na cílové architektuře závislý. V takovém případě bude tedy nutné zjistit, na jakých revizích této architektury jsou zařízení cílové skupiny založena. Implementujeme-li výpočetně náročnou aplikaci (což multimediální knihovna je) budeme zřejmě chtít dosáhnout co nejvyššího výkonu využitím volitelných rozšíření a podpůrných jednotek, kterými mohou být tyto procesory vybaveny. Jak je patrné ze statistik uvedených v kapitole 5.2, v mobilních přístrojích se systémem Android se uplatňuje zejména architektura ARMv7 (profil ARMv7-A) a ve velmi malé míře také starší ARMv6. K těmto architekturám, jejich profilům a rozšířením si tedy uvedeme bližší informace.

2.4.2 Architektury ARMv6 a ARMv7

Architektury ARMv6 a ARMv7 se liší zejména rozdílnými rozšířeními instrukční sady, jak si uvedeme dále. Naprostá většina procesorů architektury ARMv6 jsou pouze jednojádrové (výjimku tvoří ARMv6K, jádro ARM11 MPCore). Většina čipů ARMv7 naproti tomu existuje ve variantách s jedním až čtyřmi jádry. Součástí návrhu ARMv7 jsou rovněž takzvané *profily architektury*. Definovány jsou profily A, R a M, které jsou uváděny jako sufix za číselným označením architektury – např. ARMv7-A. Každý z profilů se zaměřuje na jiný segment cílového trhu. Profil A (Application) definuje architekturu s virtuální pamětí (VMSA, Virtual Memory System Architecture) a zaměřuje se na výkonná zařízení, na kterých bude provozován plnohodnotný operační systém. Profil R (Real-time) specifikuje stroj s chráněnou pamětí (PMSA, Protected Memory System Architecture) a cílí na aplikace vyžadující deterministické časování operací a nízkou latenci při obsluze přerušení. Poslední profil – M (Microcontroller) byl navržen s ohledem na použití v běžných vestavěných systémech. Tento profil je rovněž založen na paměťovém systému typu PMSA.

Z hlediska zaměření této práce je pro nás nejvýznamnější architektura a profil ARMv7-A. Na procesorech této třídy je založena většina cílových zařízení a patří sem velmi populární jádra jako např. výkonný Cortex-A9 nebo low-endový Cortex-A7. Zjednodušené blokové schéma těchto jader je vidět na obrázku 2.2.



Obrázek 2.2: Blokové schéma jader ARM Cortex-A7 a Cortex-A9. [15]

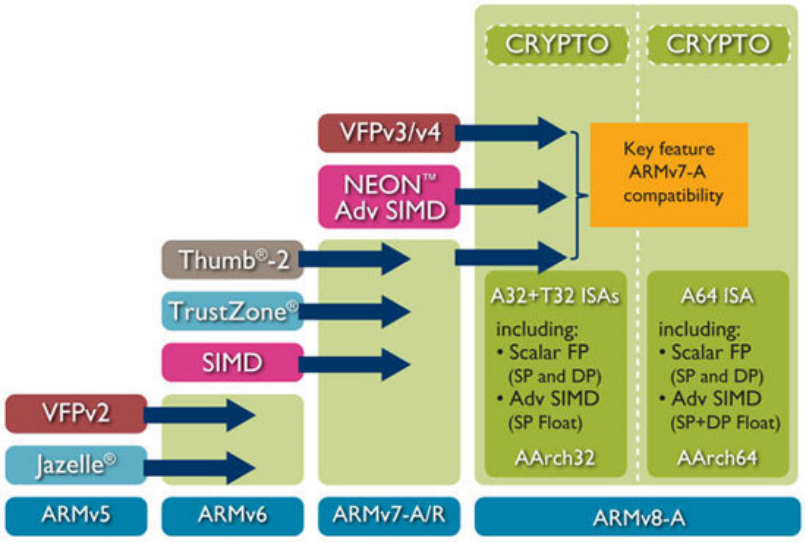
Je důležité si všimnout, že Cortex-A7 i A9 podporují vícejádové konfigurace (1 až 4 jádra). Tato jádra pracují v režimu symetrického multiprocesingu (SMP). Součástí každého z jader je kromě samotného CPU i vyrovnávací paměť první úrovně (Level 1, L1 cache) pro přednačtení a uchování instrukcí a dat. Je-li součástí jádra i některý z volitelných koprocesorů (VFPv3 a NEON, viz dále), má každé jádro k dispozici vlastní jednotku tohoto typu. Jádro Cortex-A9 je proti A7 samo o sobě výkonnější, navíc je možné ho rozšířit o jednotku grafického akcelerátoru Mali-624. Součástí základní varianty A7 je řadič vyrovnávací paměti druhé úrovně (Level 2, L2 cache) a až 1 MB této paměti. A9 proti tomu nabízí volitelný řadič paměti L2 cache (není na obrázku) vhodný pro konkrétní použití a disponující až 8 MB paměti. Obě jádra podporují instrukční sady ARM, Thumb-2 a zmíněná rozšíření VFPv3 a NEON. [15]

O těchto vlastnostech si nyní řekneme více.

2.4.3 Instrukční sady ARM a jejich rozšíření

Jedním ze specifik architektur ARM počínaje verzí ARMv3 je podpora více navzájem výrazně odlišných instrukčních sad. Za nativní můžeme považovat 32 bitovou sadu ARM, která je dostupná na všech procesorech ARM. Počínaje ARMv3 je k dispozici alternativní sada instrukcí s označením Thumb. Tato sada je 16 bitová a byla do návrhu začleněna jako prostředek k dosažení větší hustoty kódu – podmnožina 32 bitových instrukcí ARM je kódována na 16 bitové instrukce Thumb. Jedním z omezení instrukcí Thumb je nemožnost využít některé registry procesoru. Navíc zde figuruje nezanedbatelná dodatečná režie, způsobená právě překladem na instrukce ARM. Výkonovou penalizaci Thumb instrukcí do značné míry řeší inovovaná sada Thumb-2, která je dostupná v některých revizích architektury ARMv6 a všech revizích ARMv7. Jelikož jsou sady Thumb a Thumb-2 méně náročné na vstupně/výstupní kanál procesoru, mohou být v některých situacích výkonnějším řešením ve srovnání se sadou ARM (to platí zejména pro Thumb-2). Rozšířením sady Thumb-2 je ThumbEE (Thumb Execution Environment), které je standardní součástí ARMv7-A a volitelné pro ARMv7-R. ThumbEE přináší změny, které umožňují efektivní provádění byte-kódu jazyků Java, C# a dalších.

Mimo tyto instrukční sady byla architektura ARM postupně obohacována o různá volitelná rozšíření. Tento vývoj je ve zjednodušené podobě zachycen na obrázku 2.3.



Obrázek 2.3: Vývoj architektur ARM od ARMv5 do ARMv8. [15]

Je patrné, že rozšíření představená jako doplněk určité architektury, se stávají součástí základní specifikace architektury následující. Situace však není tak jednoduchá. Některá rozšíření, která se následovně stala standardem, mohou být v pozdějších revizích zahrnuta pouze v základní podobě a ne vždy ve stejně výkonově výhodné implementaci jako u revizí předcházejících. Často je výkon takového řešení degradován ve prospěch nově představeného rozšíření, které lze použít jako alternativu. Jelikož je nově představené rozšíření volitelné, nemusí být v konkrétním čipu k dispozici a to se může při některých aplikacích projevit výraznou výkonovou penalizací.

Jedním z rozšíření je Jazelle nebo též Jazelle DBX (Direct Bytecode eXecution), které poskytuje hardwarovou podporu pro interpretaci byte-kódu jazyku Java. Jazelle tak umožňuje výrazné zrychlení vybraných operací, které by jinak musely být prováděny softwarovou im-

plementací virtuálního stroje JVM. Toto rozšíření bylo představeno s architekturou ARMv5TEJ („J“ zde značí pravě přítomnost Jazelle) a bylo typickou součástí architektur ARMv6. V jádrech založených na ARMv7 byla však již poskytnuta pouze minimální podpora bez praktické akcelerace v hardware. Místo toho se doporučuje využít již zmíněného rozšíření instrukční sady Thumb-2 označovaného jako ThumbEE nebo též Jazelle RCT (Runtime Compilation Target).

Významným rozšířením je sada instrukcí umožňující práci s matematickým koprocesorem, označovaná jako VFP (Vector Floating Point). VFP poskytuje instrukce pro práci s čísly s plovoucí řádovou čárkou. Podporována je jak jednoduchá (single) tak dvojitá přesnost (double precision). Jednotka nepodporuje paralelní práci s vektory, jak by název mohl naznačovat. Vektorové operace jsou prováděny sekvenčně, nejedná se tedy o pravé rozšíření typu SIMD (viz dále). Sada VFP se poprvé objevila v rámci architektury ARMv5 – jednalo se o revize VFPv1 a VFPv2. Podpora dvojité přesnosti zde byla volitelná. Revize VFPv3 přinesla dvojnásobný počet registrů (32 64 bitových). Tato verze je k dispozici na architektuře ARMv7-A, kam spadají populární jádra Cortex-A8 a Cortex-A9. Jádro A9 obsahuje VFPv3 v plnohodnotné variantě, A8 disponuje VFPv3 pouze v provedení VFPlite, které je přitom výrazně pomalejší. [12] Zatím poslední verzí této sady je VFPv4, která přichází s instrukcemi typu Multiply-accumulate („vynásob a shrň“) a podporou pro čísla s poloviční přesností (half precision). VFPv4 je součástí jader Cortex-A12 a Cortex-A15.

Pod označením NEON nebo též Media Processing Engine se skrývá koprocesor a rozšíření instrukční typu SIMD. Instrukce typu SIMD (Single Instruction, Multiple Data) umožňují provádět danou operaci nad více daty současně – toto chování je vhodné např. pro práci s vektory. NEON je 64/128 bitová instrukční sada umožňující pracovat s celočíselnými typy o délce 8, 16, 32 a 64 bitů, typy s pohyblivou řádovou čárkou s poloviční a jednoduchou přesností. Dvojitá přesnost není rozšířením podporována a může být vhodnější pro tento typ operací využít rozšíření VFPv3 je-li dostupné. Spektrum nabízených instrukcí je uzpůsobeno zejména operacím typickým pro práci s multimediálním obsahem – zpracování obrazu a zvuku, interaktivní grafika (hry). Jsou-li obě rozšíření NEON a VFPv3 na daném jádře dostupná, jejich registry se navzájem částečné kryjí, s čímž je třeba počítat zejména při psaní kódu v jazyce symbolických instrukcí. Ve většině případů, kdy není nevyžadována dvojitá přesnost čísel s plovoucí řádovou čárkou, dosahuje implementace s využitím sady NEON vyššího výkonu ve srovnání s obdobným řešením založeným na VFPv3. [2]

Kapitola 3

Knihovny pro práci s videem

Tato kapitola se zabývá rozborem knihoven pro práci s multimediálním obsahem – pohyblivým videem, zvukem a doplňujícím obsahem jako jsou titulky.

Konzultovány jsou zejména veřejně dostupné, nekomerční řešení, z nichž bude vybrána vhodná varianta, která bude následně použita jako základní stavební kámen při návrhu a implementaci zastřešující knihovny pro platformu Android. Zaměříme se na multiplatformní knihovny použitelné jak na platformě PC (Microsoft Windows, Linux, Mac OS X) tak na mobilních zařízeních s operačním systémem Android, čímž bude výrazně rozšířena oblast použitelnosti navrhovaného řešení. Základními požadovanými schopnostmi je možnost dekódování i kódování obrazových a zvukových dat. U jednotlivých řešení budeme dále sledovat rozsah poskytovaných možností a kvalitu jejich implementace. Zejména se zaměříme na škálu podporovaných multimediálních kontejnerů a kodeků.

Svými vlastnostmi se jako nejvíce zajímavé jeví zejména tyto projekty:

- FFmpeg
- libav
- OpenMAX

Jejich popisu, srovnání a výběru jedné z nich se budou věnovat následující podkapitoly.

3.1 FFmpeg

Projekt FFmpeg [17] je komplexní multiplatformní řešení poskytující prostředky pro přehrávání, konverzi, záznam a streamování audiovizuálního obsahu.

Jedná se o sadu knihoven a nástrojů, která je udržována skupinou nazvanou *FFmpeg team*. Projekt je dostupný v podobě zdrojových kódů nebo již v přeložené podobě a v současnosti je šířen pod licencí GNU Lesser General Public License (LGPL) verze 2.1. Na některé volitelné součásti se však vztahuje licence GNU General Public License (GPL) verze 2 případně licence jiná[1].

Mezi hlavní přednosti FFmpegu patří velmi rozsáhlá podpora nejrůznějších kodeků pro audio i video. Z nejpopulárnějších jsou to např. MPEG-2, MPEG-4, H.264 a VC-1. Seznam nejpoužívanějších kodeků zahrnutých v FFmpegu je zanesen ve srovnávací tabulce

[1] Součásti poskytované pod licencí GPL v2 lze vyjmout z překladu. Pokud jsou tyto ponechány vztahuje se licence GPL v2 na celý balík FFmpeg.

3.1. Pro naprostou většinu formátů je implementován alespoň dekodér, u vybraných pak i kodér. Úplný seznam kodérů a dekodérů je možné dohledat na stránkách projektu v sekci *Documentation*, *General Documentation*. Další důležitou vlastností tohoto balíčku je schopnost manipulovat s širokou škálou multimediálních kontejnerů pomocí takzvaných muxerů a demuxerů. Za zmínku rovněž stojí také sada podpůrných nástrojů a filtrů.

Je na místě však poznamenat, že některé nástroje a kodeky nejsou příliš robustní a podpora pro méně časté formáty a jejich exotické varianty je spíše na úrovni minimální implementace. Nástrojům a knihovnám se blíže věnuje samostatná podkapitola 3.1.1.

FFmpeg je dostupný pro nejrůznější architektury jako x86 (IA-32 i AMD64), ARM (v5, v7 aj.), PowerPC, MIPS, DEC Alpha a další. Z operačních systémů jsou podporovány Linux, FreeBSD, OpenBSD, NetBSD, Microsoft Windows, ale i méně známé a dnes již nepříliš populární systémy jako např. IBM OS/2. Kompletní seznam aktuálně podporovaných architektur a systémů lze najít přímo na stránkách projektu v sekci *FATE*.

Kodek	FFmpeg	libav
Video		
MPEG-2	ano	ano
MPEG-4	ano	ano
H.264	ano	ano
(SMPTE) VC-1	ano	ano
QuickTime	ano	ano
RealVideo	1-4	1-4
WMV	7,8,9	7,8,9
IntelIndeo	2-5	2-5
Audio		
AAC	ano	ano
AC-3	ano	ano
ATRAC	1,3,3+	1,3,3+
FLAC	ano	ano
MPEG	MP1,2,3	MP1,2,3
RealAudio	ano	ano
PCM	ano	ano
WMA	1,2,další	1,2,další

Tabulka 3.1: Tabulka nejpopulárnějších kodeků dostupných v balíčcích FFmpeg a libav. [17] [18]

3.1.1 Nástroje a knihovny

Jak již bylo zmíněno, součástí FFmpegu je i sada podpůrných **nástrojů**. Tyto však nemusí být k dispozici pro všechny architektury a systémy, zejména u uživatelských sestavení se často některé nebo všechny nástroje vynechávají pro jejich nadbytečnost pro specifické použití, nebo pro jejich nekompatibilitu s cílovou platformou. Některé z nástrojů si popíšeme:

- **ffmpeg** - Jedná se o textovou aplikaci jejímž prostřednictvím lze (pomocí parametrů) využívat ostatní součásti balíčku pro kódování, dekódování, konverzi a to nejen z/do souboru, ale i z/do hardwarového zařízení.

- **ffplay** - Multimediální přehrávač využívající knihovny FFmpegu. Přehrávač vyžaduje pro překlad/spuštění sadu knihoven SDL, což je jeden z důvodů proč nemusí být dostupný pro všechna prostředí.
- **ffserver** - Jedná se aplikaci, která umožňuje streamování multimediálních datových toků prostřednictvím počítačových sítí.
- **ffprobe** - Utilita, která v textové podobě vypíše informace o specifikovaném multimediálním souboru či toku.

Jádrem balíčku FFmpeg jsou však **knihovny**. Následuje seznam a stručný stručný popis nejdůležitějších z nich:

- **libavcodec** - Jedna z nejvýznamnějších součástí obsahující veškeré kodeky.
- **libavformat** - Knihovna umožňující práci s multimediálními kontejnery (muxery a demuxery).
- **libavfilter** - Knihovna poskytující nejrůznější filtry.
- **libpostproc** - Implementace postprocessingu pro video obsah.
- **libswresample** - Knihovna určená k převzorkování zvukových záznamů.
- **libswscale** - Knihovna pro změnu rozlišení a barvové reprezentace videa.

Některé důležité funkce poskytované knihovnami libavformat, libavcodec, libswscale a libswresample budou včetně praktického použití popsány v kapitolách 5 a 6 (Návrh a Implementace).

3.2 libav

Balíček s názvem libav [18] vznikl v roce 2011 odštěpením od projektu FFmpeg. S FFmpegem sdílí drtivou většinu vlastností a schopností, avšak projekty (a aplikace na nich založené) nejsou přímo kompatibilní. Libav se snaží řešit některé problémy, se kterými se potýká projekt rodičovský. Mimo jiné jsou to problémy spojené s legální distribucí některých součástí.

Prakticky vše co bylo v předchozí kapitole řečeno o projektu FFmpeg lze vztáhnout i k libav. Jak je patrné z tabulky 3.1 i v oblasti podpory nejpoužívanějších kodeků jsou obě řešení přímo srovnatelné.

3.2.1 Nástroje a knihovny

Sada nástrojů a knihoven je v případě libav velmi podobná jako u projektu FFmpeg:

- **avconv** - Obdoba nástroje ffmpeg umožňující kódování, dekódování, konverzi a další operace s multimediálními soubory a datovými toky.
- **avplay** - Velmi jednoduchý multimediální přehrávač založený na schopnostech libav.
- **avserver** - Alternativa k utilitě ffserver, určená ke streamování audia i videa.

- **avprobe** - Nástroj poskytující nejrůznější informace o definovaném multimediálním souboru či toku.

Srovnatelné s FFmpegem jsou i poskytované knihovny:

- **libavcodec** - Obsahuje samotné kodeky.
- **libavformat** - Sada muxerů a demuxerů.
- **libavfilter** - Audio/video filtry.
- **libpostproc** - Funkce realizující postprocessing pro video obsah.
- **libavresample** - Sada funkcí pro převzorkování zvukových záznamů (obdoba libswresample z FFmpegu).
- **libswscale** - Funkce pro změny rozlišení videa a barvové konverze.

Popisu funkcí poskytovaných jednotlivými knihovnami projektu libav se v rámci této práce věnovat nebudeme. Funkce a principy jejich použití jsou však srovnatelné s funkcemi FFmpegu, kterým se věnují pozdější kapitoly 5 a 6 (Návrh a Implementace).

3.3 OpenMAX

OpenMAX [20] nebo též Open Media Acceleration je sada programových rozhraní pro jazyk C určená pro práci s multimediálním obsahem nejrůznějšího typu – zejména zvuku a statického i pohyblivého obrazu. Projekt se snaží poskytnout několik standardních rozhraní, která usnadňují a sjednocují vývoj kodeků a aplikací pro práci s tímto typem obsahu. Důraz je kladen na podporu přenosných a vestavěných zařízení, po kterých v dnešní době požadujeme zpracování velkých objemů multimediálního obsahu. V prostředí systému Android je OpenMAX využít multimediálním enginem Stagefright (Android verze 2.0 a vyšší), který nabízí zejména softwarové kodeky pro nejrozšířenější formáty, na některých zařízeních je možné využít i hardwarový kodek či implementovat vlastní řešení jako zasuvný modul (pro vrstvu OpenMAX IL, viz dále).

OpenMAX je otevřené multiplatformní „royalty-free“[2]. řešení, které spolu s jinými knihovnami (jako např. OpenGL či OpenCL) spravuje nevýdělečné konsorcium známé jako Khronos Group.

Rozhraní, která OpenMAX poskytuje, jsou dle použití rozdělena do třech vrstev:

- Application Layer (OpenMAX AL),
- Integration Layer (OpenMAX IL),
- Development Layer (OpenMAX DL).

3.3.1 OpenMAX AL

Vrstva s označením Application Layer se chová jako rozhraní mezi samotnou aplikací a multimediálním prostředím specifickým pro cílovou platformu. To umožňuje snazší portování aplikací pro různé cílové platformy. Toho je dosaženo poskytnutím sady standardizovaných objektů a metod pro manipulaci s nimi. Základem OpenMAX AL je podpora pro multimediální přehrávače a aplikace pro zachytávání obsahu tohoto typu.

[2] „Royalty-free“ označuje intelektuální vlastnictví, které je poskytováno zdarma bez jakýchkoliv licenčních či jiných poplatků.

3.3.2 OpenMAX IL

Rozhraní vrstvy Integration Layer slouží jako abstrakce jednotlivých bloků systému. Blokem zde může být hardware, software či kombinace obojího. OpenMAX IL umožňuje inicializaci, řízení a uvolnění jednotlivých bloků.

3.3.3 OpenMAX DL

Development Layer je sadou nízkoúrovňových rozhraní stojících mezi hardware konkrétního zařízení a multimediálními kodeky nebo dalšími nástroji zejména pro zpracování signálů a videa. Hardwarem zde může být klasický procesor (CPU), signálový procesor (DSP), speciální akcelerátor apod. V rozhraní je vystavena podpora pro několik oblastí zpracování multimédií. Tyto oblasti jsou v souvislosti s OpenMAX DL označovány jako domény a každá z nich obsahuje několik specifických subdomén (např. několik nejpoužívanějších kodeků spadajících do dané domény). Tabulka 3.2 shrnuje všechny domény a subdomény přítomné v rozhraní OpenMAX DL.

Doména		**Subdoména**	
Název	**Význam**	**Název**	**Význam**
AC	Audio Coding	MP3	MP3
		AAC	AAC
VC	Video Coding	COMM	Common
		M4P2	MPEG4 Part 2
		M4P10	MPEG4 Part 1
IP	Image Processing	PP	Pre- a Post-processing
		CS	Color Space Conversion
		BM	Bitmap Manipulation
IC	Image Coding	JP	JPEG codec
SP	Signal Processing	—	—

Tabulka 3.2: Domény a subdomény rozhraní OpenMAX DL. [20]

3.4 Volba knihovny

Všechny uvedené knihovny jsou v nějaké formě dostupné pro platformu Android. OpenMAX jako takový není „hotovým“ řešením, ale jedná se spíše o jakousi vrstvu mezi hardwarovými či softwarovými kodeky a ostatními součástmi multimediálního software.

I když toto řešení počítá s podporou většiny nejdůležitějších kodeků, tyto kodeky nemusejí být součástí konkrétního enginu vystavěného na základě OpenMAX (tj. Stagefright pro Android), a realizace takového enginu se může na jednotlivých platformách lišit. Z tohoto hlediska se jeví zajímavější knihovny FFmpeg a libav, které je možné konfigurovat na míru danému použití a takto přichystané řešení je použitelné na nejrůznějších platformách.

Rozdíly mezi FFmpeg a libav nejsou z našeho pohledu příliš významné oba balíčky disponují prakticky totožnou sadou kodeků a muxerů/demuxerů. V současné době je FFmpeg populárnější a více diskutované řešení a u libav není zcela jasné, zda se podaří vyřešit problémy, kvůli kterým došlo k odštěpení od FFmpegu. Z tohoto důvodu bude při řešení

použit FFmpeg, avšak principy popsané v rámci této práce lze aplikovat i na projekt založený na libav. Při návrhu bude přihlédnuto k možnosti záměny vnitřní knihovnu za jinou příbuznou, tak aby byla provedená změna zejména navenek co nejvíce transparentní - tj. aby se obešla pokud možno beze změn ve veřejném rozhraní navrhované knihovny. Změny ve vnitřní implementaci by pochopitelně byly nezbytné, avšak jejich rozsah by v případě přechodu na příbuzné řešení (libav) nebyl příliš velký (zejména by se jednalo o změnu názvů funkcí, definic a struktur s občasnými odlišnostmi v parametrech funkcí).

Kapitola 4

Dostupná řešení

Pro práci s multimediálním obsahem v prostředí virtuálního stroje JVM a platformu Android je dostupných několik řešení příbuzných k tématu této práce. Tato kapitola se některým z nich stručně věnuje.

4.1 jjmpeg

Projekt *jjmpeg* [13] si stejně jako tato práce klade za cíl vytvořit mezivrstvu mezi balíčkem FFmpeg a aplikací psanou v jazyce Java pro virtuální stroj JVM. Řešení poskytuje funkční rozhraní pracující na vyšší úrovni oproti funkcím poskytovaným jednotlivými moduly FFmpegu, čímž zavádí určitou míru abstrakce a usnadňuje tak vývoj samotné aplikace.

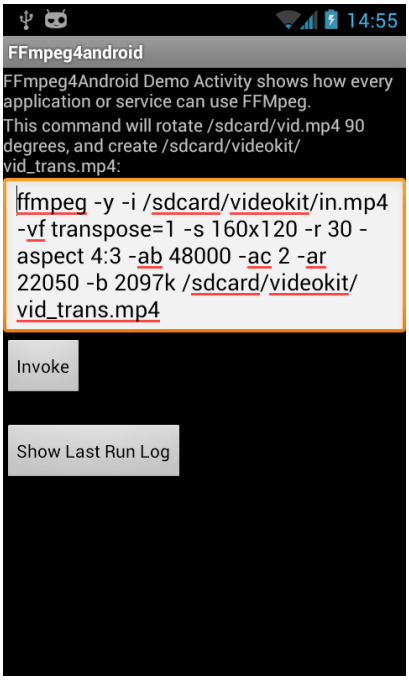
Projekt je bohužel ponechán v rozpracované fázi a není již delší dobu aktualizován či udržován. Jako základ využívá starou verzi knihovny FFmpeg a bohužel nedisponuje dostatečnou dokumentací, což znesnadňuje jak její použití, tak případný další vývoj. Zdrojové kódy projektu i binární soubory jsou však k dispozici ke stažení a to pod licencí LGPLv3 (nebo pozdější).

Odnoží projektu je *jjmpeg-android*, který se zaměřuje na poskytnutí stejné funkcionality v prostředí systému Android. Tato větev se stále nachází v rané fázi vývoje a projekt rovněž není již delší dobu aktualizován. Zdrojový kód je poskytován v rámci licence GPLv3 (nebo pozdější).

Vzhledem ke stavu, v jakém se projekt nachází, nebude jjmpeg při realizaci práce využit jako její základ ani nebudou přímo použity jeho součásti. Může se však stát hodnotným zdrojem informací při řešení specifických problémů v rámci návrhu a implementace – v takovém případě bude tato skutečnost uvedena na příslušném místě v textu práce (případně ve zdrojovém kódu samotném).

4.2 FFmpeg 4 Android a příbuzné

Aplikace *FFmpeg 4 Android* [16] od vývojářů *JY Team NetComps LTD* je dostupná přímo přes distribuční službu Google Play. Jedná se o knihovnu FFmpeg (nyní ve verzi 2.0) přeloženou pro systém Android s optimalizacemi pro vybrané architektury. Tato knihovna je zastřešena jednoduchou aplikací, která umožňuje zadávání textových příkazů, které jsou následně vykonány příslušnou částí FFmpegeu. Rozhraní aplikace je zachyceno na obrázku 4.1.



Obrázek 4.1: Uživatelské rozhraní aplikace FFmpeg 4 Android.

Další informace a příklad použití lze najít na domovské stránce projektu [16]. Součástí aplikace bohužel není API pro Java aplikace, které by umožňovalo přímou spolupráci nadřazené aplikace a FFmpegu v reálném čase.

K dispozici je několik obdobných řešení jako např. *FFmpeg for Android* rovněž umožňující zadávání příkazů pro FFmpeg. Z dalších aplikací jmenujme *FFmpeg Media Encoder* a *ffmpeg codec*, které poskytují jednoduché GUI pro dávkové zpracování videa (konverzi).

4.3 Práce Marka Vyorala

Vedoucím práce jsem byl upozorněn na diplomovou práci Marka Vyorala, která se zabývá podobnou tématikou.

Marek Vyoral ve své práci „Editor videa pro platformu Android“ [30] rovněž využil knihovnu FFmpeg jako základ pro multimediální aplikaci. Jeho práce poskytuje několik zajímavých poznatků, avšak autor se zaměřuje spíše na aplikaci samotnou nežli na tvorbu univerzálního rozhraní a knihovny pro FFmpeg.

Jeho práce však byla přínosným zdrojem v prvních fázích práce s knihovnou FFmpeg, protože na jednom místě poskytuje ucelené informace týkající se této problematiky.

Kapitola 5

Návrh

Tato kapitola se věnuje návrhu zastřešující knihovny pro balíček FFmpeg. Vyvíjená knihovna byla pracovně pojmenována *MediaLib* a pod tímto názvem se na ni budeme v následujícím textu odkazovat. Obsahem první podkapitoly je specifikace požadavků plynoucích ze zadání práce a poznatků získaných dosavadní analýzou problému. Následuje definování cílové platformy v rámci rodiny operačních systémů Android. Pozdější části se již týkají návrhu veřejného aplikačního rozhraní a vnitřní struktury samotné knihovny. Poslední podkapitola bude popisovat návrh jednoduché aplikace určené pro demonstraci činnosti knihovny MediaLib.

5.1 Požadavky na knihovnu

Jak již bylo uvedeno, knihovna MediaLib bude fungovat jako prostředník mezi uživatelskou aplikací a balíčkem FFmpeg.

Od výsledného řešení jsou požadovány tyto základní schopnosti:

- Dekódování videa (zvuku, a případně jiného obsahu),
- kódování videa (a zvuku),
- možnost upravit data obrazového snímku.

Kódováním zde rozumíme akceptování obrazových (případně zvukových) dat ve formátu RAW (raw data; surová data uložená jako sekvence bytů bez metadat), zakódování dat s využitím některého z podporovaných kodeků a zápis dat do výstupního kontejneru. Proces kódování by mělo být možné parametrizovat, co se volby kodeku, kontejneru a dalších nastavení týče. Dekódování je proces opačný, kdy očekáváme data v některém z podporovaných formátů, tento formát automaticky rozpoznáme, vybereme vhodný kodek a pomocí něj data dekódujeme. Před předáním dat nadřazené aplikaci případně ještě provedeme další vhodné úpravy, jako například změnu rozlišení a barevné hloubky obrazových dat či úpravu počtu kanálů u dat zvukových. V režimu dekódování dále požadujeme podpůrné funkce jako je skok na určitou pozici v nahrávce, získání časové značky a dalších informací.

Knihovna by rovněž měla umožnit úpravy obrazového snímku za účelem aplikace grafických efektů, filtrů apod.

Důležitým aspektem řešení je jeho výkon, tak aby bylo možné knihovnu v režimu dekodéru použít jako základ pro multimediální přehrávač, aniž by docházelo k výpadku snímků

nebo přeskakování zvukových rámců. V režimu kódování bychom chtěli, aby byla knihovna prakticky použitelná pro zpracování videa na běžně dostupných mobilních zařízeních (specifikace viz kapitola 5.2).

Na základě konzultace s vedoucím práce byly vybrány kodeky, jejichž podpora bude od výsledného řešení vyžadována. Jedná se v současnosti o jedny z nejpoužívanějších video a audio kodeků balíčku FFmpeg – konkrétně MPEG-2, MPEG-4 a H.264 pro video a MPEG a AC-3 pro audio. Lze však očekávat, že knihovna bude podporovat práci s naprostou většinou kodeků a kontejnerů podporovaných FFmpegem.

5.2 Cílová platforma

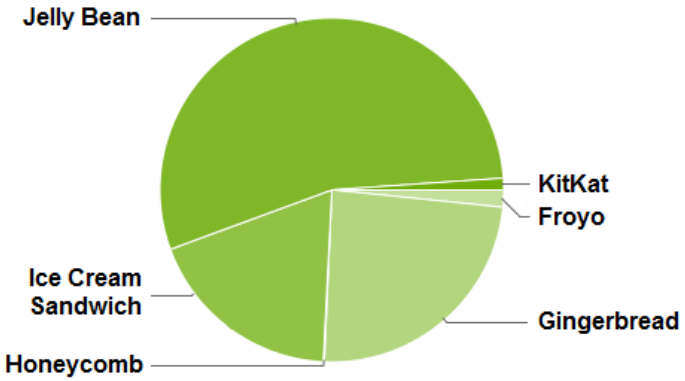
Platforma Android je softwarově i hardwarově velmi rozmanitá. V rámci návrhu je třeba přesněji definovat platformu, na kterou se při návrhu zaměříme. Případné změny této specifikace, které vyplynou ve fázi implementace a testování, budou spolu s odůvodněním uvedeny v kapitole 6 (Implementace) a zhodnoceny v kapitole 8 (Závěr).

5.2.1 Softwarová platforma

Verze	Jméno	API	Podíl [%]
2.2	Froyo	8	1,6
2.3.3 - 2.3.7	Gingerbread	10	24,1
3.2	Honeycomb	13	0,1
4.0.3 - 4.0.4	Ice Cream Sandwich	15	18,6
4.1.x	JellyBean	16	37,4
4.2.x		17	12,9
4.3		18	4,2
4.4	KitKat	19	1,1

Tabulka 5.1: Zastoupení jednotlivých verzí systému Android ke konci roku 2013. [9][8]

Tabulka 5.1 a obrázek 5.1 zachycují relativní zastoupení jednotlivých verzí systému Android mezi uživateli a to ke konci roku 2013 [9][8], kdy probíhalo rozhodování o cílové platformě realizované knihovny. Verze starší než 2.2 zde nejsou zahrnuty, protože se jedná již o velmi zastaralé revize systému a jejich procentuální zastoupení je menší než 1. Jelikož by hardware přístrojů s takto starou verzí systému (2009) velmi pravděpodobně zdaleka nedostačoval k provozu realizovaného software, můžeme si je dovolit zanedbat. Podobná situace se týká i tabletové verze 3.2, která má zastoupení rovněž menší než 1 %. Jak je vidět, drtivá většina zařízení disponuje alespoň Androidem ve verzi 2.3.3 Gingerbread. Tato verze bude při návrhu a implementaci považována za minimum, přičemž verze 2.2 může být podporována, pokud to nebude znamenat implementační komplikace nebo výkonovou či funkční penalizaci pro knihovnu jako celek. Při testování knihovny a demonstrační aplikace se zaměříme mimo verzi 2.3.x na verze 4.0 Ice Cream a 4.1.x/4.2.x Jelly Bean, jelikož se jedná o verze s nejvyšším zastoupením. Nové verze 4.3 a 4.4 nemají zatím přílišné zastoupení, avšak lze očekávat, že tato situace se změní v jejich prospěch, proto bude zajištěna i jejich podpora. Na všech podporovaných verzích se pokusíme kromě testování i o měření výkonu implementovaného řešení a vzájemné porovnání takto získaných dat, čemuž se bude věnovat samostatná kapitola 7.



Obrázek 5.1: Graf zastoupení jednotlivých verzí systému Android ke konci roku 2013. [9][8]

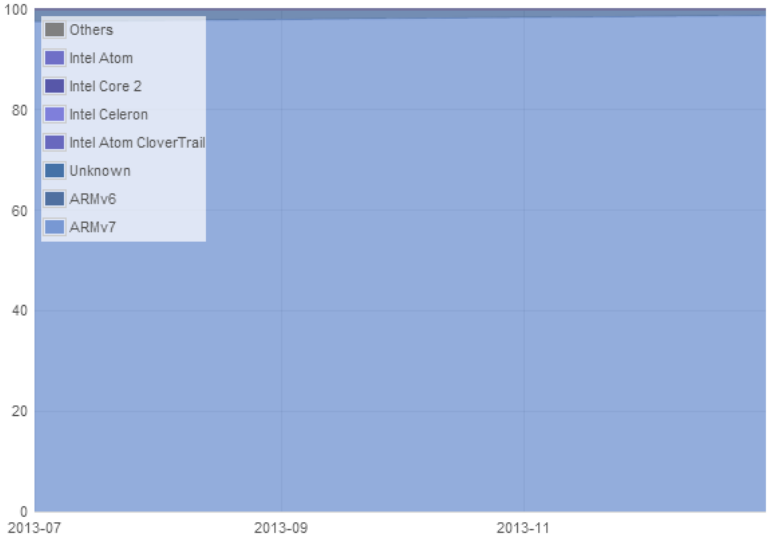
5.2.2 Hardwarová platforma

Dalším aspektem při zacílení projektu je volba podporovaného a doporučeného hardware. Vzhledem k faktu, že aplikace bude obsahovat nativní kód, bude třeba rovněž stanovit podporované architektury mikroprocesorů.

Statistická data týkající se hardware mobilních zařízeních se systémem Android není tak snadné získat, jako tomu bylo v případě software. Následující text vychází ze statistik poskytovaných společností Unity Technology [26] a z obecného povědomí o aktuálním stavu trhu s mobilními zařízeními.

Stav k 3. 2014:

Architek-tura	Podíl [%]
ARMv7	98,8
ARMv6	1,2
ostatní	~0



Obrázek 5.2: Poměrné zastoupení architektury procesorů na přelomu let 2013/2014. [26]

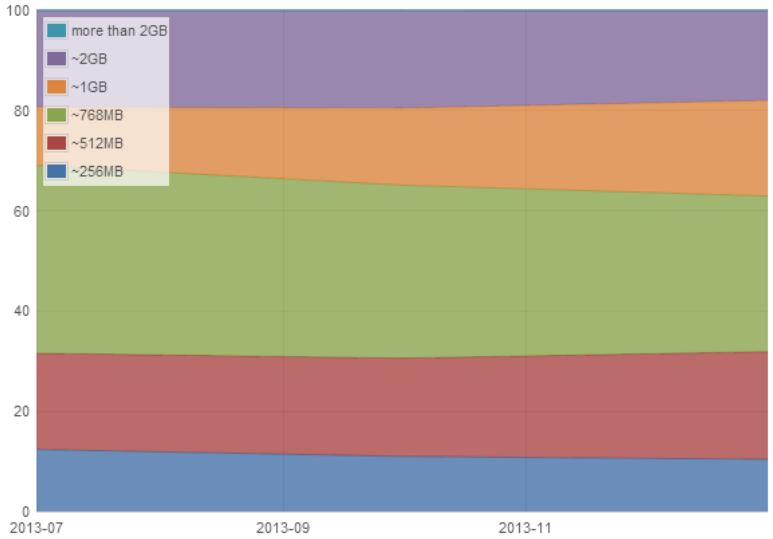
Společnost Unity Technology, která se pohybuje v oblasti herních technologií nejen pro mobilní zařízení, poskytuje na svých webových stránkách řadu statistik týkajících se hardwarového vybavení přístrojů se systémy Android, iOS, Windows Phone a jinými. Vzhledem k oblasti působnosti této společnosti, lze očekávat že nashromážděná data nemusí reprezentovat situaci na celém trhu od zařízení spadajících do kategorie low-end až po zařízení nejvyšší třídy. Výsledky budou spíše reprezentovat situaci v „horní polovině“ trhu, což však nečiní tato data nepoužitelná pro práci z oblasti zpracování multimédií, jelikož i zde

lze očekávat zaměření na podobný segment. Statistiky, které Unity Technology poskytuje, je možné filtrovat dle operačního systému, proto se zde zveřejněné výsledky budou týkat výhradně platformy Android. Jak je patrné z obrázku 5.2, prakticky 100 % podíl na trhu patří zařízením založeným na architektuře ARM. Z tohoto celku mají dominantní postavení (98,8 %) procesory rodiny ARMv7, na druhém místě se potom umístila architektura ARMv6 avšak se zastoupením pouhých 1,2 %. Na základě těchto informací se v rámci této práce zaměříme právě na architekturu ARMv7. Bude však učiněna snaha poskytnout nativní kód v optimalizované podobě i pro jiné architektury, výkonově výhodná rozšíření instrukční sady a hardwarové akcelerátory/kodeky. Základní informace o architektuře ARM byly poskytnuty v kapitole 2.4. Překlad kódu a jeho optimalizace pro tyto procesory bude diskutována v kapitole 6.4.

Jak již bylo uvedeno, práce cílí zejména na zařízení spadající alespoň do současné střední nebo vyšší střední třídy, které disponují dostatečnými prostředky pro uspokojivou práci s videem. Konkrétní hardwarové požadavky budou stanoveny až ve fázi implementace a testování.

Stav k 3. 2014:

Kapacita [MB]	Podíl [%]
~256	10,4
~512	21,5
~768	31,1
~1024	19,1
~2048	18,0
jiná	~0



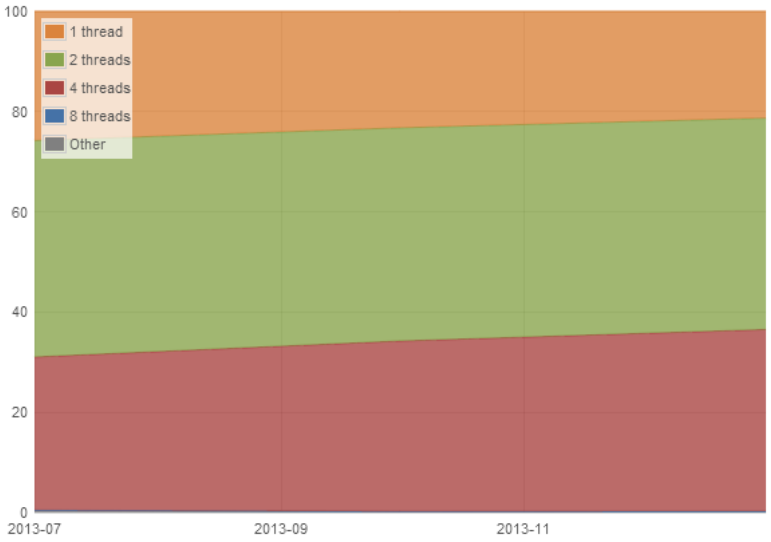
Obrázek 5.3: Kapacita operační paměti zařízení se systémem Android na přelomu let 2013/2014. [26]

Současnou situaci na poli mobilních zařízení z pohledu kapacity operační paměti a počtu logických jader (vláken) procesoru zachycují obrázky 5.3 a 5.4. V případě procesorů ARM odpovídá počet logických jader počtu fyzickému. Je patrné, že téměř 80 % zařízení je vybaveno dvou nebo čtyř jádrovým procesorem. Přičemž trend naznačuje poměrně stálou pozici dvoujádrových procesorů, které lze v současnosti považovat za mainstream. Patrný je rovněž postupný úpadek zařízení vybavených jediným jádrem a pozvolný nástup procesorů s jádry čtyřmi. Knihovnu se tedy budeme snažit optimalizovat pro dvou a čtyřjádrové procesory a to zejména vhodnou distribucí časově náročného kódu mezi více paralelních vláken. Vícejádrové procesory ARMv7 těchto zařízení jsou typicky taktovány kolem 1 GHz nebo výše, při vývoji a testování se tedy zaměříme zejména na tuto výkonovou kategorii. Co se kapacity operační paměti týče, většina přístrojů (přibližně 90 %) disponuje alespoň 512 MB, při návrhu se tedy bude počítat alespoň s touto kapacitou. Je však pravděpodobné že knihovnu bude možné provozovat i na přístrojích s operační pamětí menší kapacity.

Stejně jako v případě náročnosti na výkon procesoru bude i v oblasti požadavků na

Stav k 3. 2014:

Počet vláken	Podíl [%]
1	21,3
2	42,2
4	36,3
8	0,1
jiný	0,1



Obrázek 5.4: Počet logických jader se systémem Android na přelomu let 2013/2014. [26]

paměť pochopitelně záležet na konkrétním použití knihovny, použitém multimediálním materiálu a také na množství prostředků, které zabírá samotný systém a ostatní procesy.

5.3 Bloková struktura

Na obrázku 5.5 je znázorněno blokové schéma jednotlivých částí systému využívajícího knihovnu MediaLib včetně jejich rozložení mezi vrstvu nativního kódu a kódu pro virtuální stroj. Z obrázku a předchozích kapitol je zřejmé, že se bude jednat o aplikaci hybridní.

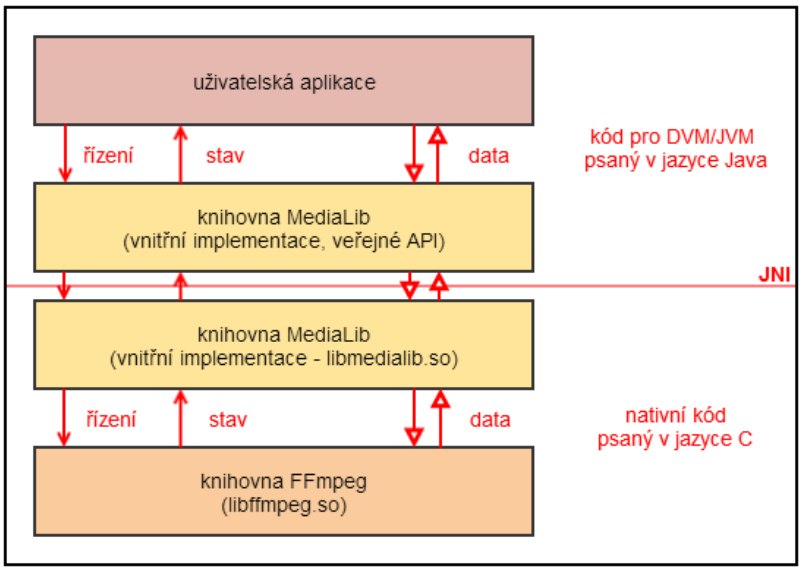
Horní polovina obrázku obsahuje součásti psané v jazyce Java, které běží ve virtuálním stroji JVM/DVM. Nachází se zde pochopitelně uživatelská aplikace a část knihovny MediaLib psaná v jazyce Java včetně jejího veřejného API.

Ve spodní polovině obrázku se pak nachází převážná část kódu knihovny – nativní komponenty psané v jazyce C. Kromě nativní části MediaLib ve formě knihovny `libmedialib.so` je zde rovněž samotný balíček FFmpeg, který je taktéž zkompilovaný do podoby knihovny s názvem `libffmpeg.so`.

Obě poloviny jsou navzájem propojeny rozhraním Java Native Interface (JNI), kterému se věnovala kapitola 2.3.4.

Červené šipky na obrázku znázorňují komunikaci a tok dat mezi jednotlivými funkčními bloky. Ve velmi zjednodušené podobě je tak zachycena možnost aplikace zadávat příkazy knihovně MediaLib (šipka „řízení“), která buďto provádí přímo jejich zpracování, nebo je požadavek transformován a předán jako příkaz či sekvence příkazů knihovně FFmpeg. Nadřazené aplikaci je rovněž nutné hlásit stav provedené operace a stav celé knihovny MediaLib (šipky „stav“). Kromě řízení a zpětné vazby je třeba zajistit tok vlastních multimediálních dat mezi jednotlivými bloky systému, což je na obrázku vyznačeno jako „data“.

Známe již tedy blokovou strukturu systému, v následujících kapitolách se tedy můžeme zaměřit na podrobný návrh a popis jednotlivých součástí a způsoby jejich komunikace.



Obrázek 5.5: Schéma jednotlivých částí systému využívajícího knihovnu MediaLib.

5.4 Obecný návrh rozhraní a činnosti knihovny

Tato kapitola se věnuje návrhu a popisu veřejného aplikačního rozhraní (API), které bude jako jediné dostupné pro aplikace vystavené nad MediaLib. Je důležité, aby toto rozhraní umožnilo splnit veškeré požadavky kladené na knihovnu jako takovou, přitom se však budeme snažit, aby bylo jeho použití v aplikaci co možná nejsnadnější a dostatečně intuitivní. Rovněž budou diskutovány principy vnitřní činnosti této knihovny.

Dle požadavků specifikovaných v zadání práce a na základě zkušeností získaných při práci s knihovnou FFmpeg jsme schopni sestavit seznam základních operací, které musí rozhraní poskytovat. Tyto operace budou následně transformovány do podoby funkcí veřejného rozhraní.

Rozhraní knihovny MediaLib bude poskytovat následující operace:

- inicializace a uvolnění knihoven MediaLib a FFmpeg a jejich prostředků,
- otevření a uzavření multimediálního souboru,
- získání informací o souboru a jeho obsahu,
- nastavení parametrů kódování/dekódování,
- volba datového toku (video a audio stopy),
- získání a vložení snímku (s možností manipulace pomocí filtrů),
- získání a vložení zvukového rámce,
- nastavení a získání pozice v (primárním) datovém toku,
- hlášení stavových statistických údajů.

Knihovna by měla umožňovat práci s více soubory současně – proto bude vhodné rozhraní rozdělit na množinu funkcí vztahujících se k celé knihovně a funkce umožňující práci na úrovni souborů.

Se soubory bude možné pracovat v jednom ze tří základních režimů:

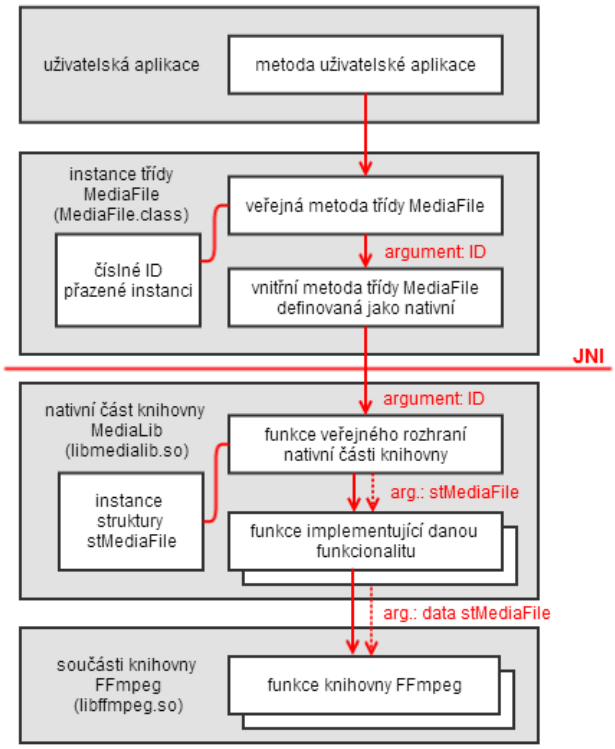
- režim dekódování – *Decoder*,
- režim kódování – *Encoder*,
- režim přehrávač – *Player*.

První dva režimy vycházejí ze specifikace dané zadáním práce. Poslední režim – přehrávač – byl do návrhu zpětně promítnut v procesu prototypování. Tento režim umožňuje efektivní přehrávání multimediálního obsahu v reálném čase, s možností předem stanovit oblast pro výstup videa a použitím třídy `AudioTrack` pro výstup zvuku. Bližší informace k tomuto režimu a odůvodnění proč byl zaveden budou poskytnuty dále v této kapitole a kapitole 5.7 věnující se třídě `AudioTrack`.

Jak již bylo uvedeno v předchozích odstavcích, rozhraní bude kvůli podpoře více otevřených souborů rozděleno na dva celky. Prakticky pak bude API tvořeno veřejnými metodami dvou tříd jazyka Java, které byly v rámci návrhu pojmenovány `MediaLib` a `MediaFile`. Třída `MediaLib` bude poskytovat metody vztahující se ke knihovně samotné – půjde zejména o její inicializaci a uvolnění. Další, neméně důležitá, metoda bude sloužit k otevírání multimediálních souborů. S touto akcí bude spojeno vytvoření instance třídy `MediaFile`, která od tohoto okamžiku slouží jako rozhraní pro práci s daným souborem, až do okamžiku jeho opětovného uzavření. Obě třídy budou kromě veřejného rozhraní obsahovat i malou část vnitřní funkcionality, její převážná část se však bude nacházet v nativní části aplikace. Zde budou knihovna i multimediální soubor realizovány strukturami jazyka C a sadou funkcí. Pro lepší orientaci si můžeme i tyto struktury nyní pojmenovat – `stMediaLib` reprezentující knihovnu jako celek a `stMediaFile` reprezentující multimediální soubor. Struktura `stMediaLib` bude uchovávat informaci o stavu celé knihovny. Veškeré informace vztažené ke konkrétnímu souboru budou pak součástí struktury `stMediaFile`. Zde budou kromě stavu udržovány i informace o paměti, která obsahuje vlastní obrazová, zvuková či jiná pomocná data příslušející danému souboru.

Máme tedy navržen způsob realizace knihovny i multimediálního souboru na obou stranách rozhraní JNI. Dále je třeba vyřešit propojení těchto stran. V případě knihovny samotné je situace poměrně jednoduchá, jelikož se nepředpokládá přítomnost více „instancí“ současně. Můžeme tedy přímo mapovat jedinou instanci třídy `MediaLib` v jazyce Java na jedinou instanci nativní struktury `stMediaLib`. Pro multimediální soubory budeme však muset navrhnout mechanismus, který zajistí, aby byla instance třídy `MediaFile` provázána s odpovídající instancí struktury `stMediaFile`. K tomuto účelu je možné použít jedinečnou informaci (např. číselnou), která bude při otevírání multimediálního souboru na některé straně rozhraní JNI vygenerována a druhé straně předána. Nativní struktura i Java třída si tuto informaci uchová. Kdykoliv pak dojde ke komunikaci přes JNI z třídy `MediaFile`, bude nativní funkci jako jeden z argumentů předána i ona unikátní informace. Tuto informaci použijeme pro výběr patřičné instance struktury `stMediaFile`. Konkrétní způsob implementace bude vybrán a popsán v kapitole 6.1.

Obecný princip činnosti celého systému při volání některé z metod rozhraní `MediaFile` je zachycen na obrázku 5.6. Pro jednoduchost zde není zachycen proces zpětného navracení



Obrázek 5.6: Princip činnosti systému využívajícího knihovnu MediaLib.

stavové informace (případně dat) uživatelské aplikaci. V horní části obrázku je znázorněna uživatelská aplikace volající některou z metod instance třídy `MediaFile`. Jedná-li se o metodu, která vyžaduje komunikaci s nativní částí knihovny, je z privátní proměnné získána hodnota unikátního identifikátoru a ta je při volání nativní funkce spolu s dalšími argumenty předána na druhou stranu rozhraní JNI (na obrázku znázorněno vodorovnou červenou čarou). Zde je volání zachyceno příslušnou exportovanou funkcí, která se mimo jiné na základě hodnoty obdrženého identifikátoru postará o získání odpovídající instance struktury `stMediaFile`. Dále je již vykonána funkcionalita, která byla uživatelskou aplikací požadována. V případě potřeby můžou být k tomuto účelu volány funkce poskytované jednotlivými součástmi knihovny FFmpeg (spodní část obrázku). Výsledkem prováděných operací budou většinou data, která budou po patřičných úpravách předána přes jednotlivé vrstvy zpět aplikaci. K předání těchto informací přes rozhraní JNI může dojít prostřednictvím některého z argumentů funkce, nebo alternativním způsobem, který využívá dlouhodobě udržované „reference“ na cílový Java objekt. Těmto technikám a jejich konkrétnímu použití se budeme věnovat v kapitole 6.1.

5.5 Návrh veřejného rozhraní

Nyní když máme obecně definovány operace, které bude knihovna MediaLib nabízet a známe rovněž princip činnosti jednotlivých vrstev této knihovny včetně způsobu jejich spolupráce, můžeme se v návrhu posunout do další fáze. Věnovat se budeme metodám realizujících jednotlivé operace knihovny. Zde popsaný návrh vycházel z doposud uvede-

ných poznatků a výsledné formy nabyl na základě procesu prototypování. V následujících odstavcích se budeme věnovat specifikaci rozhraní těchto metod a jejich chování. Souvisejícím tématem je návrh propojení metod s knihovnou FFmpeg, této oblasti bude věnována kapitola 5.6.

5.5.1 Rozhraní třídy MediaLib

Nejprve se zaměříme na metody spadající pod společnou část rozhraní knihovny – tedy pod třídu `MediaLib`. Následující blok pseudokódu zachycuje jejich deklaraci ve zjednodušené podobě:

```
MediaFile(); // konstruktor
void Init();
void Release();
String GetInfo();
MediaFile openFile( String filePath, Options options );
```

Metoda `Init()` bude především zajišťovat nastavení stavových proměnných na výchozí hodnoty a načtení potřebných knihoven (nativní části MediaLib a knihovny FFmpeg) do paměti. Volání této metody musí být učiněno jako první krok před jakoukoliv další činností s knihovnou MediaLib. Tuto operaci bude pravděpodobně možné začlenit do konstruktoru třídy `MediaLib`, nevyvstane-li při implementaci nějaká překážka.

Pokud aplikace hodlá práci s knihovnou ukončit, je vhodné po uzavření všech souborů (viz dále) volat metodu `Release()`. Ta zajistí korektní ukončení práce s nativními knihovnami. Uvolnění těchto knihoven bude ponecháno v režii systému Android, jelikož rozhraní Android API neposkytuje žádnou metodu, která by tuto akci na základě požadavku vyvolala.

Jako prostředek pro získání informací o knihovně MediaLib a použitém balíčku FFmpeg lze použít metodu `GetInfo()`. Ta v textové podobě vrátí verzi obou knihoven a v případě FFmpegu i řadu dalších informací vztahujících se k použitému sestavení tohoto balíčku, včetně seznamu dostupných komponent apod.

Klíčovou funkci rozhraní třídy `MediaLib` plní metoda `openFile()`, která slouží k otevření multimediálního souboru. Nedojde-li v průběhu otevírání souboru k chybě, bude vytvořena nová instance třídy `MediaFile` a ta bude na základě principů popsaných v minulé kapitole provázána s nově vytvořenou instancí struktury `stMediaFile` na straně nativního kódu. Další úkony související s otevíráním souboru jsou již prováděny v rámci těchto nových instancí. Objekt třídy `MediaLib` se již pouze postará o předání patřičného unikátního identifikátoru instanci třídy `MediaFile`. Dalším krokům prováděným při otevření souboru se budeme věnovat v následující části kapitole, zaměřené na třídu `MediaFile`.

Dojde-li při otevírání souboru k neočekávané chybě, bude nadřazená aplikace o této skutečnosti informována generováním výjimky, jak je v prostředí jazyka Java zvykem. Tento způsob bude použit i u ostatních metod veřejného rozhraní, kde hrozí, že některá z dílčích operací může selhat.

Rozhraní metody pro otevření souboru obsahuje dva argumenty. Textový argument `filePath` slouží pro předání plné cesty a názvu multimediálního souboru. Argument `options` je abstrakcí množiny různých parametrů, které budou použity k přizpůsobení procesu otevírání souboru konkrétním požadavkům aplikace. Nejvýznamnějším prvkem této množiny bude parametr definující režim, ve kterém bude soubor otevřen – půjde o již zmíněné režimy dekódování, kódovaní a režim přehrávač. Další parametry spadající do této množiny budou specifikovány v rámci implementace.

5.5.2 Rozhraní třídy MediaFile

První metodou třídy `MediaFile`, které se budeme věnovat, je její konstruktor. Návrh rozhraní předpokládá, že o vytváření instancí této třídy se bude starat výhradně nadřazená část rozhraní – třída `MediaLib`. Konstruktor tedy nebude určen pro veřejné použití uživatelskou aplikací.

```
MediaFile( int ID, Options options ); // konstruktor
```

Jak je vidět z uvedeného pseudokódu, konstruktor bude očekávat dva argumenty. Prvním argumentem (ID) je identifikátor unikátní pro danou instanci. Tento identifikátor bude konstruktoru předán instancí třídy `MediaLib`, jak bylo popsáno výše. Argument `options` je i zde abstrakcí argumentů použitých pro parametrizaci procesu otevření souboru (množina obsahuje zejména prvek nesoucí informaci o zvoleném režimu práce). Po inicializaci stavových proměnných bude konstruktor přes rozhraní JNI informovat nativní část knihovny, aby provedla zbývající úkony související s otevřením multimediálního souboru. Dle zvoleného pracovního režimu půjde především o výběr vhodného muxeru resp. demuxeru pro manipulaci s multimediálním kontejnerem. Dojde-li v rámci některé operace konstruktoru k selhání či výjimce, bude o tom informována nadřazená třída (tedy `MediaLib`) rovněž formou výjimky.

Po úspěšném otevření souboru je nad ním možné prostřednictvím rozhraní třídy `MediaFile` provádět různé operace. Množina platných operací se bude lišit v závislosti na definovaném režimu práce.

V režimech dekodér a přehrávač by první operací měl být výběr aktivních datových toků. K tomu bude sloužit metoda `SelectStream()`.

```
void SelectStream( Type type, int ID, boolean primary );
int GetStreamCount( Type type );
```

Prvním parametrem metody `SelectStream()` je typ stopy, jejíž výběr provádíme. Podporovány budou audio a video stopy. Druhý parametr je číselný identifikátor požadované stopy. Poslední parametr umožňuje nastavit danou stopu jako primární (primární stopa bude využita pro získávání časových značek a dalších informací).

Počet stop daného typu uvnitř souboru bude možné získat voláním metody `GetStreamCount()`.

Je-li nastaven režim kodér je třeba před další činností do souboru vložit alespoň jeden datový tok. To umožňuje následující metoda:

```
int InsertStream( Type type, Options options );
```

Parametr `type` zde opět udává typ stopy (audio či video). Abstraktní parametr `options` bude sloužit k definici dodatečných informací o toku jako např. použitý kodek. Návratovou hodnotou je číselný identifikátor nového toku. V tomto režimu je rovněž možné použit informační metodu `GetStreamCount()` popsanou dříve.

Klíčovou operací v režimech kodér a dekodér je vložení resp. získání snímku (příp. zvukového rámce). Tuto funkcionalitu budou zastávat následující metody:

```
void GetVideoFrame( VideoData video );
void GetAudio( AudioData audio, AudioInfo info );

void InsertVideoFrame( VideoData video );
void InsertAudio( AudioData audio, AudioInfo info );
```

Z názvu metod lze odhadnout jejich funkcionalitu. Metody pracující s obrazovou informací, nevyžadují žádné dodatečné informace, jelikož pracují vždy právě s jedním snímkem a formát dat je předem jednorázově nastaven (viz dále). Parametry zvukové informace jsou rovněž definovány před jejich získáváním či vkládáním, avšak na rozdíl od obrazu, zde se může délka informace (počet zvukových rámců) pro jednotlivá volání metody lišit. Je to dáno povahou zvukových dat, která jsou většinou uložena po blocích pro určitý počet obrazových snímků.

Nositelem informace o délce předaných dat resp. počtu předaných rámců je argument `info`. K předání resp. získání obrazové a zvukové informace pak slouží parametry `video` a `audio`. V režimu přehrávač budou zvuková data získávána automaticky, volání metody `GetAudio` v tomto režimu nebude platnou operací. Volba formátu zvuku je v tomto režimu v režii třídy `AudioTrack` (viz kapitola 5.7).

Nacházíme-li se v režimu dekódování či kódování, parametry výstupních resp. vstupních audio a video dat jsou jednorázově nastaveny použitím těchto metod:

```
void InitVideoFrame( Options options );
void InitAudio( Options options );
```

V případě videa bude možné definovat rozlišení obrazu (jeho výšku a šířku v pixelech) a datový formát. U zvuku pak půjde o nastavení vzorkovací frekvence, množství informace na vzorek, počet kanálů a případně další parametry, které budou specifikovány ve fázi implementace.

Ne vždy si aplikace při dekódování či přehrávání vystačí se sekvenčním přístupem k multimediálnímu obsahu, proto jsou k dispozici metody umožňující „skok“ na konkrétní pozici v záznamu.

```
int SeekToFrame( int frame, Options options );
int SeekToTime( int time, Options options );
```

První funkce (`SeekToFrame()`) umožňuje specifikovat požadovanou pozici číslem snímku, druhá (`SeekToTime()`) potom očekává jako parametr časovou značku. Je-li poskytnutá informace platná, dojde k patřičné změně pozice. Toto nastavení nemusí být vždy zcela přesné (dle formátu stopy a zvolené metody nastavení pozice), proto obě funkce vrací hodnotu odpovídající skutečně nastavené pozici. Pomocí parametru `options` je možné dále upravit chování těchto metod – zejména bude možné definovat, zda se má přejít na nejbližší snímek odpovídající dané značce nebo zda stačí skok na nejbližší klíčový snímek. Důvod, proč je vhodné poskytnout oba typy přístupu, bude vysvětlen později při popisu vnitřní činnosti knihovny.

Nedílnou součástí rozhraní knihovny by měly být i funkce pro získání stavových informací. Kromě čísla a času aktuálního snímku budou mimo jiné k dispozici metody pro získání celkového času a celkového počtu snímků. Jelikož se tyto informace mohou lišit pro jednotlivé toky obsažené v souboru, bude možné parametrem `ID` definovat vztažnou stopu.

```
int GetTotalFrameCount( int ID );
int GetTotalTime( int ID );
int GetFrameId( int ID );
int GetTime( int ID );
```

Pro režim přehrávač bude třeba aplikaci nabídnout možnost spustit, pozastavit a ukončit přehrávání. Tyto metody se budou starat zejména o obsluhu třídy `AudioTrack`. Navrhovaný formát těchto funkcí je následující:

```
void Play();
void Pause();
void Stop();
bool IsPlaying();
```

Formát a chování těchto metod bude upřesněno ve fázi implementace. Naposled uvedená metoda (`IsPlaying()`) poskytne zpětnou vazbu o tom, zda momentálně probíhá přehrávání.

Ukončí-li aplikace práci se souborem, použije metodu `CloseFile()`, která se postará o uvolnění využité paměti a uzavření multimediálního souboru. Příslušná instance třídy `MediaFile` bude od tohoto okamžiku považována za neplatnou.

5.5.3 Rozhraní pro obrazové filtry

Dle zadání má být při návrhu brán ohled na možnost aplikace grafických filtrů. Filtry psané v jazyce Java mají při současném návrhu přímý přístup k obrazovým datům jak při dekódování, tak při kódování. Tyto filtry mohou rovněž využít doposud popsané rozhraní k získání dodatečných parametrů jako rozměry obrazu, jeho formát, aktuální časovou značku apod.

Jelikož grafické filtry mohou být výpočetně poměrně náročné, bylo by vhodné poskytnout podporu i pro filtry v nativním kódu. Takový kód je totiž typicky daleko efektivnější a je rovněž možné využít knihoven pro zpracování obrazu jako např. OpenCV (Open Source Computer Vision) [19]. Podpora pro tento typ filtrů bude implementována v podobě externí knihovny. Tato knihovna bude ze strany Java kódu zastřešena třídou `FilterLib`, která umožní získat ukazatele na nativní funkce implementující jednotlivé filtry. Rozhraní `MediaFile` bude rozšířeno o možnost předání těchto ukazatelů. Proces kódování a dekódování bude doplněn o dodatečný krok, který v případě potřeby zajistí přímé volání funkce realizující požadovanou grafickou operaci. Tímto postupem se vyhneme nadbytečné a neefektivní komunikaci přes rozhraní JNI. Důležité je vhodně navrhnout rozhraní pro nativní funkce, jelikož bude společné pro všechny filtry.

Filtrům budou poskytnuta vstupní data a oddělený paměťový prostor pro uložení výstupních dat. Toto oddělení vstupu a výstupu je vhodné, jelikož ne všechny filtry mohou číst a přímo modifikovat poskytnutou paměťovou oblast (tzv. režim práce *in situ*). Mimo to budou k dispozici informace o rozměrech obrazu, formátu dat (shodný pro vstup i výstup), čísle snímku (pro časově závislé filtry) a dodatečných uživatelských parametrech, které bude možné nastavit v době běhu aplikace. Následuje ukázka navrhované formy rozhraní nativních filtrů ve formě pseudokódu:

```
void filter( VideoData dataIn, VideoData dataOut, int width, int height,
    int frameId, Params params );
```

Poslední parametr `params` zde seskupuje uživatelské a případně další parametry, které budou blíže specifikovány ve fázi implementace.

Ukazatel na funkci s tímto rozhraním a celkový počet funkcí bude možné získat přes zmíněné rozhraní třídy `FilterLib`. K tomuto účelu budou sloužit následující dvě metody:

```
int GetFilterCnt();
int GetFilter( int nId );
```

Funkce `GetFilter()` přímo vrátí ukazatel na nativní funkci, který bude možné instanci třídy `MediaFile` předat a případně opět zrušit touto sadou metod:

```
void SetVideoFilter( int nFilterAddr, Params params );
void ClearVideoFilter();
```

Argument `params` zde zachycuje množinu uživatelských parametrů, kterými lze za běhu ovlivnit chování filtru.

Aplikace více filtrů (jejich řetězení) není tímto minimalistickým návrhem přímo podporována, ale ani omezena – řetězení filtrů je možné implementovat v rámci nativní knihovny obsahující samotné filtry.

5.6 Propojení s knihovnou FFmpeg

Rozhraní popsané v předchozí kapitole bude kromě vlastní implementace zapouzdřovat operace se součástmi knihovny FFmpeg, které budou výkonným jádrem většiny poskytovaných metod. Jelikož operace poskytované knihovnou FFmpeg jsou poměrně nízkoúrovňové, bude v rámci jedné metody rozhraní `MediaLib` typicky třeba vykonat řadu dílčích operací knihovny FFmpeg, postarat se o jejich správnou součinnost a vhodně připravit vstupy a výstupy těchto operací (data).

Aby bylo možné uskutečnit optimální návrh funkcionality jednotlivých metod, bylo nutné předem analyzovat a testovat možnosti poskytované knihovnou FFmpeg. Tato kapitola bude shrnovat poznatky získané v průběhu tohoto procesu a vzhledem k výše řečenému se bude obsahově pohybovat na rozmezí kapitol Návrh a Implementace. Nebudeme se zde věnovat implementačním detailům, budou však použity názvy konkrétních funkcí FFmpegu a jejich návaznost na veřejné rozhraní popsané výše. Kapitola 6 (Implementace) nebude již opakovat zde uvedená fakta a zaměří se spíše na kód, který jednotlivé celky propojuje, a kód, který realizuje dodatečnou funkcionalitu neposkytovanou FFmpegem, příp. funkcionalitu specifickou pro platformu Android.

Ze sady součástí, které FFmpeg poskytuje, budeme využívat především `libavformat` pro práci s multimediálními kontejnery a `libavcodec` poskytující rozhraní pro vnitřní kodeky. Dále využijeme schopností součásti `libswscale` pro konverzi obrazových dat a `libswresample` pro převzorkování dat zvukových.

Před použitím některých součástí FFmpegu je třeba provést jejich tzv. registraci. Toho lze dosáhnout voláním metod `av_register_input_format()`, `av_register_output_format()` a dalších (s patřičnými parametry). Hodláme-li využít větší množství formátů a dalších součástí nebo není-li rozsah využité funkcionality předem pevně stanoven, je možné registrovat všechny součásti, které tento proces vyžadují. K tomu slouží funkce `av_register_all()`, která bude v našem případě volána při inicializaci knihovny metodou `Init()` nebo samotným konstruktorem třídy `MediaLib`.

Dále se zaměříme na operace spadající do procesu **dekódování**. První z nich je otevření vstupního souboru. K otevření multimediálního kontejneru lze použít funkci ze sady `libavformat` nazvanou `avformat_open_input()`. Použitím této funkce získáme tzv. kontext daného multimediálního souboru, který budeme dále využívat jako prostředek pro práci s ním. Pro většinu formátů máme nyní k dispozici informace o dostupných stopách. Pro formáty, které neobsahují explicitní hlavičku s těmito informacemi (jako např. MPEG), je nutné přečíst několik rámců a z nich tuto informaci získat. O to se postará funkce `avformat_find_stream_info()`. Přečtené rámce jsou bufferovány pro budoucí použití při samotném dekódování, žádná užitečná informace tedy není použitím této funkce ztracena. Tyto funkce jsou z pohledu rozhraní naší knihovny zastřešeny metodou `OpenFile()`.

Nyní již lze uskutečnit výběr požadovaných stop metodou `SelectStream()`, která musí mimo jiné učinit dvě důležitá volání součásti `libavcodec`. Prvním je vyhledání vhodného kodeku pro danou stopu funkcí `avcodec_find_decoder()`. Je-li vhodný kodek v aktuálním sestavení a konfiguraci FFmpegu nalezen, funkce vrací strukturu reprezentující daný kodek. Tato struktura však pouze nese informace o nalezeném kodeku, není však ještě jeho platnou „instancí“. Pro získání instance resp. kontextu kodeku, je třeba volat funkci `avcodec_open2()`.

Po úspěšném otevření souboru je možné získávat rámce a po výběru stopy můžeme zahájit i jejich dekódování (přísluší-li k danému toku). Způsob získání rámce je společný pro audio i video. Dosáhneme toho voláním funkce `av_read_frame()`, která v případě úspěchu vrací strukturu zapouzdřující daný paket.

Další kroky se liší v závislosti na typu stopy. Pro **video** stopu (metoda `GetVideoFrame()`) je dekódování získaného paketu vyvoláno funkcí `avcodec_decode_video2()`. Dekódovaný snímek může být v jiném formátu (typicky YUV) nežli je formát požadovaný (systém Android využívá jako výchozí formát pro bitmapová data ARGB8888). Rovněž rozlišení snímku může být odlišné od rozlišení požadovaného. V takovém případě musíme zajistit konverzi získaných dat. FFmpeg, konkrétně jeho součást `libswscale`, nám k tomuto účelu nabízí funkci sws_scale(). Před jejím použitím je však nutné získat tzv. sws kontext nastavený na požadovaný vstupní a výstupní formát a rozměry. O nastavení sws kontextu se stará funkce `sws_getContext()` a o jeho opětovné uvolnění pak funkce `sws_freeContext()`. Nezmění-li se parametry vstupního ani výstupního obrazu je efektivní tento kontext udržet a žádat jeho uvolnění až po dekódovaní všech snímků (v našem případě až při uzavření souboru).

Pro takto získaný snímek bychom chtěli znát i časovou značku. V závislosti na kontejneru a kodeku nemusí být tato značka pro každý snímek přímo dostupná. O její získání ze stopy či kontejneru (případně její výpočet) se stará funkce `av_frame_get_best_effort_timestamp()`.

Dekódování **audia** (metoda `GetAudio()`) probíhá na základě obdobného principu jako v případě obrazových dat, avšak použité funkce a pochopitelně implementační detaily se liší. Samotné dekódování je realizováno funkcí `avcodec_decode_audio4()`. Funkce typicky vrátí několik zvukových rámců dohromady (dle použitého formátu, kodeku a nastavení enkodéru). Jako v případě videa i zde je většinou formát rámců odlišný od formátu požadovaného. O převzorkování se postará funkce `swr_convert()`, jejíž kontext je nutné nejprve inicializovat (`swr_alloc_set_opts()`) s použitím vhodných parametrů a nakonec uvolnit (`swr_free()`).

Nyní se budeme věnovat režimu kódování multimediálního obsahu. Metoda `OpenFile()` se v tomto případě bude chovat odlišně. Nejprve se pokusí vytvořit kontext výstupního souboru voláním funkce `avformat_alloc_output_context2()` součásti `libavformat`. Tato funkce stanoví použitý formát na základě přípony předaného názvu souboru. Nepodaří-li se najít odpovídající formát je možné funkci zavolat s jinými parametry pro vynucení určitého typu kontejneru. O samotné otevření souboru v požadovaném režimu (v našem případě zápis) se stará funkce `avio_open`, která je rovněž součástí `libavformat`. Před otevřením souboru je třeba definovat datové toky, které budou v souboru obsaženy (viz dále). Po otevření souboru je pro některé formáty nutné vložit hlavičku, která mimo jiné obsahuje informace právě o dostupných tocích. K tomu slouží funkce `avformat_write_header()`.

Definice nového toku pro kontext výstupního souboru spočívá ve volání funkce `avformat_new_stream()`. Následují nám již známé operace vyhledání vhodného enkodéru

(`avcodec_find_encoder()`) a otevření nalezeného kodeku (`avcodec_open2()`).

Zápisu obrazových dat předchází již popsaný proces konverze s využitím funkcí součásti `libswscale`. Dalším krokem je kódování těchto dat funkcí `avcodec_encode_video2()`, která již vrací paket pro výstupní soubor. K zápisu paketu dochází voláním funkcí `av_write_frame()` nebo `av_interleaved_write_frame()`. První uvedená funkce v případě více stop (audio i video) očekává, že dodaná data jsou ve správném pořadí vzhledem k jejich časovým značkám. Druhá funkce naproti tomu dokáže tato data v případě potřeby uložit do bufferu a zajistit jejich správné časové prokládání.

Proces kódování zvuku spočívá v naplnění rámce obdrženými daty. K tomu lze využít funkci `avcodec_fill_audio_frame()`. Tento rámec bude dále třeba kódovat do výstupního formátu. To zajistí funkce `avcodec_encode_audio2()`, která obdobně jako u videa, vrací paket určený pro zápis do výstupního souboru.

Před uzavřením souboru je nutné korektně ukončit daný soubor. Tento proces se může opět lišit pro různé typy kontejneru, je proto vhodné použít funkci `av_write_trailer()`, která by měla zajistit zápis správných informací na konec souboru. Vlastní uzavření souboru je vyvoláno funkcí `avio_closep()`.

Z významných operací uveďme ještě posun v rámci souboru (seek). Jelikož se z pohledu multimediálního obsahu nejedná o triviální operaci, nabízí FFmpeg odpovídající funkci – `av_seek_frame()`. Ta by měla zajistit správné chování vzhldem k použitému kontejneru a formátu stop. Proces nastavení pozice je možné dodatečně parametrizovat. Při popisu rozhraní MediaLib jsme si uvedli dva způsoby nastavení pozice (seek). Přesnější způsob je vyhledání absolutně nejbližšího snímku vzhledem k zadané pozici. Tato operace však není triviální. Je nezbytné vyhledat nejen nejbližší předcházející klíčový snímek, ale rovněž je nutné krokovat až do požadované pozice a průběžně dekódovat jednotlivé snímky, protože požadovaný snímek nemusí (v závislosti na formátu) vždy obsahovat veškerá obrazová data. Je patrné, že tato operace bude poměrně pomalá a výpočetně náročná. Ne vždy však vyžadujeme naprostou přesnost při nastavování pozice – většina multimediálních přehrávačů v takovém případě nastavuje pozici na nejbližší dostupný klíčový snímek, což je druhý způsob nastavení pozice. Tato operace je mnohem rychlejší a méně náročná. Oba způsoby jsou podporovány funkcí `av_seek_frame()`, přičemž přesnější „seekování“ lze aktivovat nastavením vlajky `AVSEEK_FLAG_ANY`.

Pomocným funkcím FFmpegu pro práci s pamětí (alokace a uvolnění), transformaci časových a jiných informací apod. se zde věnovat nebudeme. Budou-li tyto funkce z hlediska implementace významné, budou popsány v příslušné kapitole.

5.7 Třída AudioTrack a její použití

Třída *AudioTrack* je dostupná prostřednictvím rozhraní Android API verze 3 nebo vyšší. Knihovna MediaLib v režimu *Player* ji bude využívat k přehrávání zvukové stopy v reálném čase. Tato kapitola čerpá převážně z dokumentace třídy AudioTrack dostupné na webu *Android Developers* [9].

K zahrnutí této třídy přímo do návrhu knihovny MediaLib došlo opět na základě procesu prototypování, kdy bylo při přímém přístupu k této třídě z nativního kódu dosaženo lepších výsledků, nežli v případě, kdy byla obsluha AudioTracku v režii uživatelské aplikace (psané v jazyce Java). Dále je v tomto případě možné, aby se knihovna sama starala i o synchronizaci s případnou obrazovou stopou (viz režim *Player*). Tato úprava přitom nijak

neomezuje možnosti použití zvukové části knihovny MediaLib jiným způsobem. Není tak pochopitelně omezena ani možnost vytvoření přehrávače, který se sám bude starat o synchronizaci a výstup zvukové informace (ať již využitím třídy AudioTrack či jiného řešení).

AudioTrack umí zajistit výstup poskytnutých audio dat prostřednictvím zvukového vybavení daného zařízení. Třída se rovněž stará o synchronizaci takových dat vzhledem k reálnému času – data je tedy možné třídě předávat asynchronně, ta jsou dočasně uschována ve vnitřním bufferu a v patřičný okamžik přehrána.

Třída nabízí dva základní režimy práce. *Statický* (static) a režim *streamovaní* (streaming). Statický režim je vhodný zejména pro zvuky s kratší a předem známou délkou trvání. Taková data totiž obvykle nezabírají velký paměťový prostor, je tedy možné (a většinou i vhodné) je do paměti nahrát celá najednou (např. ve fázi inicializace aplikace) a potom je přehrávat na požádání s minimální režií a latencí. Typicky se statický režim využívá například při realizaci akustické odezvy uživatelského rozhraní (UI, User Interface) a rovněž při přehrávání krátkých zvukových efektů ve hrách.

Režim streamování je naproti tomu zaměřen zejména na přehrávání dlouhých datových toků, které je nepraktické nebo i nemožné jednorázově nahrát do paměti. Problémem zde nemusí být jen omezená kapacita paměti, ale i povaha dat samotných, ta totiž mohou do zařízení přicházet z externích zdrojů v podobě nepřetržitého datového toku, který není k dispozici jako celek a není známa jeho konečná délka. Z uvedených poznatků je patrné, že tento režim bude pro použití v rámci knihovny MediaLib vhodnější.

Oba režimy využívají stejnou sadu metod, přičemž požadovaný režim činnosti je definován jako jeden z parametrů konstruktoru. Veřejný konstruktor je definován v následující podobě:

```
AudioTrack(int streamType, int sampleRateInHz, int channelConfig, int
    audioFormat, int bufferSizeInBytes, int mode);
```

První argument (`streamType`) definuje kategorii, do které bude přehrávaný zvuk spadat v rámci systému Android – tedy jestli se jedná o vyzvánění, systémový zvuk, hudbu, budík apod. Pro jednotlivé kategorie jsou připraveny odpovídající definice – např. `STREAM_MUSIC` pro hudbu. Toto nastavení obvykle definuje i to, kterým z dostupných ovládacích prvků v nastavení systému Android bude hlasitost přehrávání ovlivněna. Argument `sampleRateInHz` slouží k předání vzorkovací frekvence vstupních dat. Další v pořadí je parametr `channelConfig`, který umožňuje specifikovat konfiguraci zvukových kanálů. Toto nastavení by mělo opět odpovídat formátu vstupních dat. Je však třeba mít na paměti, že ne všechna nastavení musí být mobilním zařízením podporována. Jak lze tuto situaci řešit si uvedeme později v této kapitole. Typickými nastaveními jsou *mono* (`CHANNEL_OUT_MONO`) nebo *stereo* (`CHANNEL_OUT_STEREO`). Argumentem `audioFormat` je nutné třídě předat formát v jakém jsou vstupní data uložena. Podporován je pouze formát PCM (Pulse-code modulation) v 8 a 16 bitové variantě (`ENCODING_PCM_8BIT` a `ENCODING_PCM_16BIT`). Jsou-li vstupní data uložena v jiném formátu je třeba je nejprve dekódovat. Argument `bufferSizeInBytes` definuje velikost vnitřního bufferu instance třídy AudioTrack. Poslední argument `mode` slouží k nastavení režimu práce (viz výše).

Je-li některé z nastavení `sampleRateInHz`, `channelConfig`, `audioFormat` mimo schopnosti daného zařízení, měla by i tak být instance třídy AudioTrack vytvořena. Výsledné nastavení však bude odlišné od toho požadovaného. Je tedy nutné metodami `getSampleRate()`, `getChannelConfiguration()`, `getAudioFormat()` zjistit skutečně nastavenou vzorkovací frekvenci, konfiguraci kanálů a zvukový formát. S využitím těchto hodnot je třeba provést

konverzi do požadovaného formátu. K tomuto účelu použijeme konverzní funkce dostupné v balíčku FFmpeg.

Dalším problémem může být volba vhodné velikosti vnitřního bufferu (parametr `bufferSizeInBytes` konstruktoru). Ve statickém režimu je velikost bufferu typicky nastavena na hodnotu shodnou s celkovou velikostí přehrávaných dat. V režimu streamování je situace složitější. Je nutné nastavit minimálně hodnotu nezbytnou pro úspěšné vytvoření AudioTracku, tu lze získat voláním funkce `getMinBufferSize()`. Jak samotná dokumentace uvádí a jak jsem měl možnost si ověřit v praxi, tato velikost nemusí být dostatečná pro plynulé přehrávání. Doporučuje se nastavení velikosti bufferu alespoň na dvojnásobek minimální hodnoty, čímž efektivně dojde k aktivaci dvojitého bufferování (double-buffering). V našem případě bude hodnota nastavena na několikanásobek minimální velikosti, jelikož spousta multimediálních formátů dodává zvukové rámce po blocích vždy jednou pro několik video snímků dopředu. Tímto se minimalizuje režie spojená s předáváním dat třídě AudioTrack.

K předání zvukových dat třídě AudioTrack slouží metody `writeData()`:

```
int write(short[] audioData, int offsetInShorts, int sizeInShorts)
int write(byte[] audioData, int offsetInBytes, int sizeInBytes)
```

Obě metody mají obdobné parametry, liší se pouze datovým typem použitým pro data samotná a hodnoty zbývajících parametrů jsou vztažena k použitému datovému typu. Jelikož v ostatních částech návrhu využíváme typ `byte` pro uložení vlastních dat a součásti FFmpegu s tímto typem rovněž pracují, použijeme druhou variantu metody `write()`, která typ `byte` očekává.

Proces přehrávaní je možné ovládat metodami `play()`, `pause()` a `stop()`, které realizují funkce obvyklé pro běžný multimediální přehrávač – tedy zahájení, pozastavení a ukončení přehrávání. Třída AudioTrack nabízí množství dalších metod, sloužících např. pro ovládání hlasitosti, změnu formátu dat za běhu apod.

Bližší informace lze nalézt na webu *Android Developers* [9].

5.8 Návrh demonstrační aplikace

Součástí této práce je i návrh a realizace ukázkové aplikace, která bude využívat schopnosti poskytované knihovnou MediaLib. Tato aplikace byla vytvářena již ve fázi prototypování, kdy sloužila pro testování implementované funkcionality. Hlavním účelem aplikace však bude demonstrace základních funkcí knihovny – tj. dekódování a kódování obrazu a zvuku. Pro demonstraci dekódování v reálném čase bude v aplikaci připraven jednoduchý multimediální přehrávač, který bude umožňovat práci se zvukovým i obrazovým materiálem (popř. kombinaci obojího). K dispozici budou základní ovládací prvky pro zahájení/zastavení přehrávače a prvek pro posun v nahrávce. Mimo tuto interaktivní funkcionalitu bude aplikace vybavena prostředky umožňujícími překódování připravených multimediálních souborů mezi různými formáty. Sada těchto souborů bude pevně dána (a poskytnuta spolu s aplikací), jelikož mimo demonstraci překódování bude tato funkce rovněž sloužit k měření a následnému srovnání výkonu výsledného řešení na různých zařízeních. Aplikace bude rovněž demonstrovat možnost práce s více soubory současně, jelikož při překódování souboru bude manipulovat se vstupním a výstupním souborem (tj. nezávislými instancemi třídy `MediaFile`) v jeden okamžik.

Ukázková aplikace bude realizována jako samostatný celek (projekt) a knihovna MediaLib na ní nebude nijak závislá. Aplikace bude určena pro stejnou skupinu cílového hardware i software jako tato knihovna.

Jelikož aplikace samotná není jádrem této práce, blíže se jejímu návrhu věnovat nebudeme. Později v kapitole 6.3 si uvedeme základní vnitřní stavbu a způsob činnosti aplikace. Uveden bude rovněž popis poskytované funkcionality a uživatelského rozhraní.

Kapitola 6

Implementace

6.1 Implementace knihovny MediaLib

Tato kapitola by čtenáři měla usnadnit orientaci ve zdrojovém kódu knihovny MediaLib a osvětlit některá specifika její implementace.

Implementace samotné knihovny se nachází v adresáři `src/MediaLib` na přiloženém disku DVD. Zde je možné nalézt veškeré zdrojové, konfigurační a pomocné soubory. Součástí je i projektový soubor pro prostředí Android ADT. Při vývoji knihovny bylo použito toto softwarové vybavení:

- Android SDK pro systém Windows (x86-64) (verze aktuální k 17.9.2013),
- Android ADT (Android Developer Tools) Build: v22.2.1 (součást SDK),
- Android NDK r9 pro systém Windows (x86-64),
- Microsoft Windows 7 Professional (x86-64),
- Java SE Runtime Environment (build 1.7.0_51-b13).

V tomto prostředí byla vyvíjena samotná knihovna i demonstrační aplikace. Výjimku zde tvoří balíček FFmpeg, který byl do podoby knihovny pro systém Andorid překládán v prostředí systému Linux. Překladu knihovny FFmpeg a podrobnému popisu použitých nástrojů se věnuje samostatná kapitola 6.4.

Implementace knihovny se skládá z nativního a Java kódu, který je dle zvyklostí rozložen mezi podadresáře `jni` a `src`.

6.1.1 Nativní část

Nativní kód psaný v jazyce C je dále rozdělen do několika samostatných souborů. V souboru `api.c` jsou umístěny veškeré nativní funkce, které jsou přímo volány z virtuálního stroje přes rozhraní JNI. Tento soubor tvoří spíše jakýsi interface a implementace zde obsažená se omezuje převážně na převod datových typů rozhraní JNI na typy běžně používané v jazyce C. Dále zde dochází k uzamykání datových oblastí obdržených přes rozhraní JNI, tak abychom mohli tyto oblasti v nativním kódu modifikovat bez vzniku kolizí s kódem běžícím ve virtuálním stroji. Po provedení těchto kroků je volána funkce z některého z dalších souborů – ta již realizuje požadovanou operaci. Jakmile je daná operace dokončena,

kód v souboru `api.c` se postará o opětovné uvolnění uzamčených dat, aby nebyl přístup virtuálního stroje k těmto datům nadále blokován.

Struktura `stMediaFile` reprezentující multimediální soubor je definována v souboru `mediafile.h` a v souboru `mediafile.c` se nachází funkce pro její inicializaci. Tato struktura obsahuje veškeré kontexty knihovny FFmpeg vztahující se ke konkrétnímu otevřenému multimediálnímu souboru. Mimo to se zde nachází stavové informace a buffery pro dočasné uložení informací při konverzi obrazu a převzorkování zvuku.

Soubor `core_io.c` obsahuje implementaci funkcí pro otevření (`openFile()`) a uzavření (`closeFile()`) souboru. Tyto funkce se pouze postarají o alokaci instance struktury `stMediaFile` a její uvolnění. Další operace se liší v závislosti na režimu práce (kódování či dekódování) a jejich kód se nachází v souborech kodéru či dekodéru. Funkce `openFile()` nastavuje do globální proměnné adresu nové instance struktury `stMediaFile`. Tato hodnota je okamžitě vyzvednuta z Java kódu a uložena v příslušné instanci třídy `MediaFile` na druhé straně rozhraní JNI. Od tohoto okamžiku je při každém dalším volání nativní funkce předána tato hodnota jako první parametr funkce. Daná funkce provede přetypování na ukazatel a provede jednoduchou kontrolu, zda tento pointer opravdu směřuje na instanci struktury `stMediaFile`. K této kontrole je využita funkce `checkMediaFilePtr()`, která porovná 4 první bajty na cílové adrese se vzorovou hodnotou, která by měla být shodná pro všechny platné instance této třídy. Tento proces je vhodný zejména ve fázi vývoje knihovny, kdy se může stát, že předaná hodnota nebude díky programátorské chybě správná a bez kontroly by vedla na fatální a obtížně laditelné selhání.

Implementace dekodéru je rozložena do třech souborů s prefixem `dec_`. V souboru `dec_io.c` se nachází významná funkce `readPacket()`, která se stará o načítání rámců ze souboru. Jejím jádrem je volání funkce `av_read_frame()` knihovny FFmpeg, tak jak bylo popsáno v kapitole 5.6. Získaný rámec může obsahovat audio, video či jinou informaci. Jelikož mohou být tyto informace ve vstupním souboru různě prokládány, jsou v implementaci zavedeny samostatné fronty pro zvukové a obrazové rámce. Zde jsou rámce nachystány pro funkce video a audio dekodéru. Rámce určitého typu jsou shromažďovány pouze pokud byla zvolena stopa daného typu funkcí `selectStream()`, v případě opačném jsou zahazovány.

Video dekodér se nachází v souboru `dec_video.c`. Hlavními funkcemi jsou zde `decodeVideo()` a `scaleVideoFrame()`, které se starají o dekódování video rámce a konverzi výstupních dat do požadovaného formátu. K tomuto účelu jsou použity dříve popsané funkce balíčku FFmpeg. Před odblokováním obrazového bufferu muže být ještě volána funkce grafického filtru, byl-li předem nastaven příslušný pointer. Ukazatel na funkci filtru je definován následujícím způsobem:

```
void ( * filterFunc )( uint8_t *, uint8_t *, int, int, int64_t, float,
    float );
```

Tomu musí pochopitelně odpovídat i formát funkce, která filtr implementuje (v samostatné knihovně). První dva parametry jsou ukazatele na vstupní a výstupní data. Následují dva celočíselné parametry udávající výšku a šířku obrazu v pixelech. Následuje parametr pro předaní čísla snímku a jako poslední jsou k dispozici dva parametry typu `float` pro předání uživatelských nastavení filtru. Předaná obrazová data jsou vždy ve formátu ARGB8888 – tedy 4 bajty pro každý obrazový bod. Nastavení ukazatele na filtr probíhá prostřednictvím funkce `setVideoFilter()`, která kromě ukazatele samotného, umožňuje i nastavení dvou zmíněných uživatelských parametrů. Všechny tyto informace jsou uchovány v příslušné instanci struktury `stMediaFile`. Rovněž je jednorázově alokován dodatečný buffer pro výstupní data filtru, aby tato časově náročná operace nemusela být prováděna při práci s každým snímkem obrazu. Aktuálně použitý filtr je kdykoliv možné deaktivovat voláním

funkce `clearVideoFilter()`, která vynuluje ukazatel na filtr a uvolní alokovanou paměť.

Dekodér zvuku je implementován v souboru `dec_audio.c`. Dříve popsaný princip dekódování a převzorkování zvuku je realizován funkcemi `decodeAudio()` a `convertAudio()`. Posledním krokem je zápis samotných dat. Cíl, kam budou tato data zapsána se liší dle aktuálního pracovního režimu. V režimu dekodér proběhne zápis do paměťové oblasti předané přes rozhraní JNI. V režimu přehrávač jsou data zapsána přímo do vnitřního bufferu instance třídy `AudioTrack` a to v rámci funkce `writeAudio()`. Aby byl takový přímý zápis možný, je nutné nejprve získat referenci na instanci `AudioTracku` a její metody, o což se stará funkce `initAudioTrack()`. Přístup k Java třídám z nativního kódu je poměrně komplikovaný a bližší informace k této problematice lze nalézt například v [22].

Funkce pro nastavení a získání aktuální pozice spolu s funkcemi pro získání celkového času a počtu snímků se nacházejí v již zmíněném souboru `enc_io.c`. Veškeré funkce pracující s časovou značkou nebo číslem snímku, využívají jako prostředek k uložení těchto informací 64 bitový celočíselný datový typ `uint64_t`. Větší číselný rozsah je nezbytný, jelikož tyto parametry dosahují pro delší záznamy značně vysokých hodnot (nad rámec 32 bitového integeru). Datové typy pracující s plovoucí řádovou čárkou zde nejsou vhodným řešením, jelikož jejich použití vedlo k degradaci přesnosti časových značek, což se negativně projevilo zejména u dlouhých snímků.

Součásti související s **kodérem** jsou uloženy v souborech uvozených `enc_`. Oproti návrhu v této části přibyla funkce `insertHeader()`, která zajistí zápis hlavičky s informacemi o obsažených stopách. Tento krok nelze sloučit s funkcí pro otevření souboru, jelikož v okamžiku jejího volání nejsou ještě informace o těchto stopách známy. Tyto informace jsou získány až voláním funkcí `insertVideoStream()` a `insertAudioStream()`. Ty zajistí uložení všech potřebných parametrů do instance struktury `stMediaFile`, odkud jsou čerpány při zápisu hlavičky a hlavně pak při zápisu dat samotných.

Proces kódování zvuku a obrazu se shoduje postupem uvedeným v kapitole Návrh. Kódování zvukové informace probíhá ve funkci `insertAudio()`, která se nachází v souboru `enc_audio.c`. Obrazový kodér má jádro v souboru `enc_video.c` a vložení snímku je vyvoláno funkcí `insertVideoFrame()`. K zápisu audio i video rámců je použita dříve zmíněná funkce FFmpegu `av_interleaved_write_frame()`, která se na rozdíl od `av_write_frame()` postará i o správné prokládání zvukové a obrazové informaci v rámci výstupního souboru.

6.1.2 Část psaná v jazyce Java

Část knihovny psaná v jazyce Java je v porovnání s nativní částí mnohem jednodušší a méně rozsáhlá. Nachází se zde implementace tříd `MediaLib` a `MediaFile`, které typicky pouze zastřešují komunikaci přes rozhraní JNI. Metody dostupné přes veřejné rozhraní těchto tříd se starají o zachycení návratového kódu nativní metody. V případě, že tento kód signalizuje selhání dané operace, postará se zapouzdřující metoda o generování výjimky `MediaLibException`. Její implementace se nachází v souboru `MediaLibException.java`. Implementace výjimky je rozšířena o uložení chybového kódu, který ji vyvolal. Tento kód je možné zpětně získat v uživatelské aplikaci voláním metody `GetCode()`.

Oproti návrhu je rozhraní třídy `MediaFile` rozšířeno o alternativní metody pro získání a vložení obrazového snímku. Jedná se o metody nazvané `GetVideoFrameBitmap()` a `InsertVideoFrameBitmap()`. Argumentem těchto metod jsou data zapouzdřená třídou `Bitmap`. Tento způsob předání dat se v některých situacích jeví jako efektivnější, jelikož se vyhneme dodatečnému kopírování dat do/z bufferu třídy `Bitmap`, která je pod systémem Android velmi často používána pro práci s obrazovými daty. Původní metody

`GetVideoFrame()` a `InsertVideoFrame()` očekávají a vrací data jako pole bajtů.

Implementace třídy `MediaFile` v souboru `MediaFile.java` obsahuje několik pomocných funkcí, které nejsou součástí veřejného rozhraní. Do této kategorie spadá funkce `getChannelCfg()`, která převádí informaci o rozložení zvukových kanálů (mono, stereo, 5.1 apod.) mezi formátem využívaným třídou `AudioTrack` a formátem použitým v knihovně FFmpeg. Rovněž se zde nachází vnitřní metoda `initAudioTrack()`, která se postará o správnou inicializaci a vytvoření instance třídy `AudioTrack`. Tato metoda je vyvolána pouze v případě, kdy byl soubor otevřen v režimu přehrávač.

6.2 Implementace filtrů a jejich rozhraní

Formát funkce, která implementuje filtr, byl uveden v předchozí kapitole. Zde se budeme věnovat rozhraní, které umožňuje předat ukazatel na takovou funkci z nativní uživatelské knihovny přes rozhraní JNI na stranu Java kódu. Odtud je možné tuto hodnotu předat instanci třídy `MediaFile`, která pointer předá své nativní implementaci, kde může být filtr přímo volán v procesu kódování či dekódování.

Uživatelská knihovna s filtry je na straně Java kódu reprezentována třídou `FilterLib`. Ta se v rámci svého konstruktoru postará o získání počtu filtrů implementovaných v nativní části a o získání ukazatelů na jednotlivé tyto funkce. Tyto informace si uchová ve vnitřních proměnných a uživatelská aplikace je může získat voláním metod `GetFilterCnt()` a `GetFilter()`.

Nativní část knihovny obsahuje mimo filtry i dvě dodatečné funkce `getFilterCnt()` a `getFilterPointers()`, které Java třídě `FilterLib` poskytují informaci o počtu filtrů a ukazatele na ně. Implementaci těchto funkcí a filtrů samotných lze nalézt v souboru `jni/filters.c`. Tento soubor obsahuje dva jednoduché ukázkové filtry. První z nich `sepiaFilter()` implementuje tzv. „sépiový efekt“. Jádrem filtru je výpočet nových hodnot barevných složek modelu RGB:

```
r2 = r * 0.393 + g * 0.769 + b * 0.189;
g2 = r * 0.349 + g * 0.686 + b * 0.168;
b2 = r * 0.272 + g * 0.534 + b * 0.131;
```

Nové hodnoty jsou ještě shora omezeny na hodnotu 255. Tento postup je proveden pro všechny obrazové body.

Druhý ukázkový filtr (`grayscaleFilter()`) umožňuje převod obrazu do odstínů šedi. Nové hodnoty RGB jsou u tohoto filtru vypočteny následujícím způsobem:

```
r2 = g2 = b2 = 0.299 * r + 0.587 * g + 0.114 * b;
```

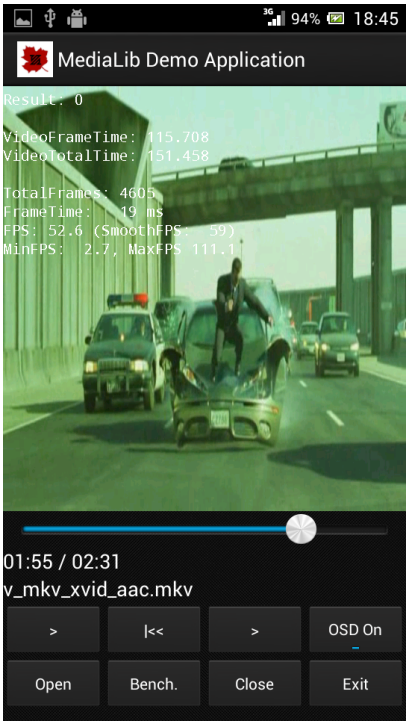
Tento výpočet je jako v případě filtru prvního opakován pro všechny body obrazu.

6.3 Implementace demonstrační aplikace

Implementace demonstrační aplikace se nachází v samostatném adresáři `src/MediaLibDemo`. V uvedeném adresáři je dostupný samostatný projektový soubor pro vývojové prostředí ADT. Aplikace se skládá z kódu psaného výhradně v jazyce Java.

Základní třídou je zde `MainActivity`, jako v případě většiny aplikací pro systém Android. Tato třída přijímá veškeré požadavky či události předané systémem a stará se o jejich obsluhu. V režii třídy `MainActivity` je rovněž vytvoření grafického rozhraní aplikace (UI) a reakce na uživatelský vstup. Rozložení jednotlivých prvků UI a jejich vlastnosti jsou definovány v souboru `main.xml`, který lze nalézt v podadresáři `res/layout`. K provázání těchto

prvků s obslužnými metodami dochází v metodě `onCreate()`. Zde jsou rovněž inicializovány ty prvky rozhraní, jejichž stav (např. popisek) se za běhu aplikace mění. Hlavní uživatelské rozhraní aplikace je zachyceno na obrázku 6.1.

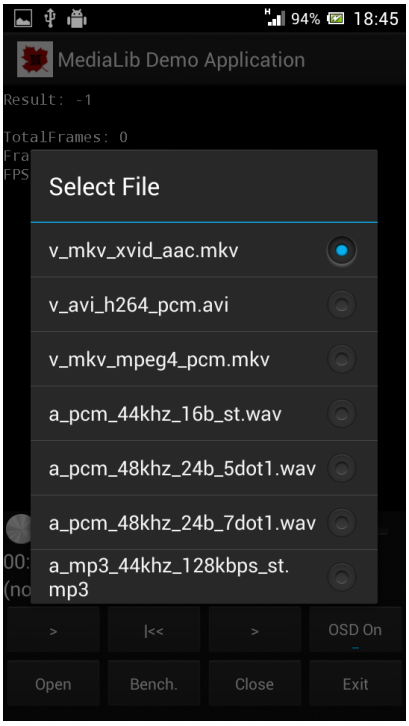


Obrázek 6.1: Hlavní obrazovka uživatelského rozhraní demonstrační aplikace MediaLib-Demo.

Největší plochu zabírá oblast pro zobrazení snímku videa, která je implementována třídou `FrameView`. Tato třída se mimo grafický výstup stará i o veškerou komunikaci s knihovnou MediaLib. Jejímu popisu se budeme věnovat později.

Ostatní části rozhraní jsou založeny na standardních komponentách poskytovaných systémem Andorid. Pod video oblastí se nachází komponenta `SeekBar`, která slouží k zobrazení pozice v přehrávaném záznamu. Aktualizace stavu této komponenty probíhá v samostatném vlákně v pravidelných časových intervalech. Uživatel může aktuální stav ručně změnit o čemž je informována třída `FrameView`, která dále žádá knihovnu MediaLib o změnu pozice v nahrávce metodou `seekToTime()`. `SeekBar` je doplněn textovým výstupem (komponenta `TextView`), který uživateli podává informaci o uplynulém a celkovém čase nahrávky. Tato informace je aktualizována stejným způsobem jako `SeekBar`.

Ve spodní části UI se nachází sada tlačítek realizovaných komponentami typu `Button`. První řada tlačítek slouží k ovládání přehrávače. Tlačítko nejvíce vlevo umožňuje spuštění a pozastavení přehrávání. O těchto událostech informuje třídu `FrameView` voláním metody `playMode()`. Druhé tlačítko zleva umožňuje skok na začátek nahrávky, což je funkcionalita obdobná přetažení ovládacího prvku komponenty `SeekBar` úplně na začátek. Další tlačítko umožňuje ruční posun o jeden snímek vpřed (je-li přehrávání pozastaveno). Po jeho stisku dochází k vyvolání metody `nextFrame()` třídy `FrameView`. Poslední tlačítko tohoto řádku slouží ke zobrazení či skrytí statistických informací zobrazovaných v oblasti snímku. Mezi tyto informace patří zejména počet snímků za vteřinu (Frames Per Second, FPS) a související údaj – doba trvání jednoho snímku v milisekundách.



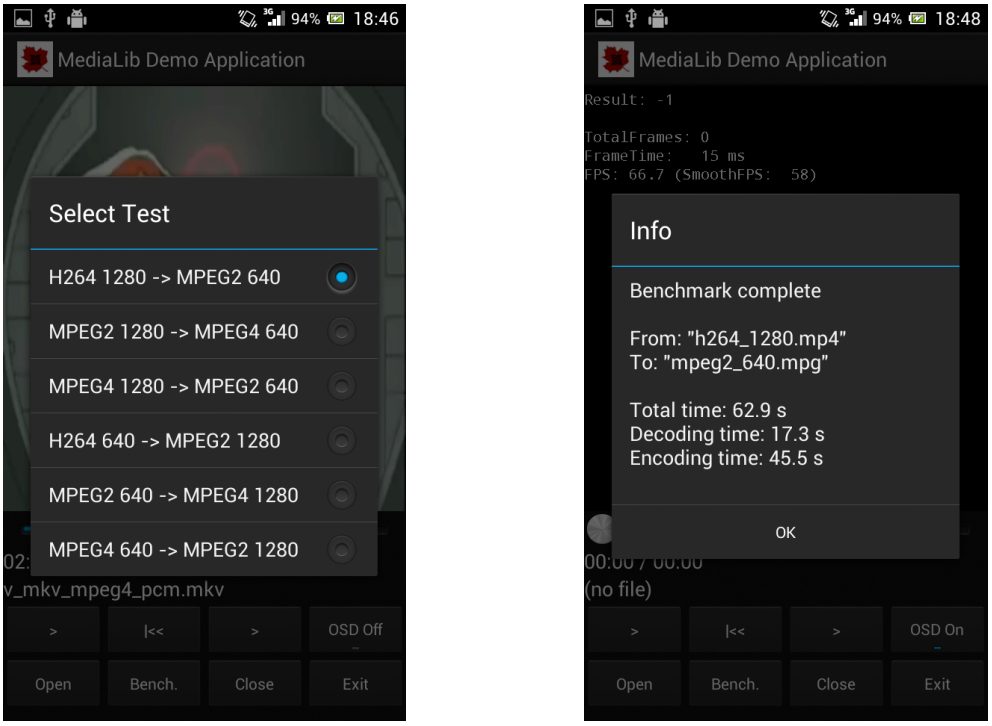
Obrázek 6.2: Dialogové okno pro výběr multimediálního souboru.

První tlačítko spodní řady (s popiskem „Open“) slouží k výběru přehrávaného souboru. Demonstrační aplikace umožňuje výběr z několika přiložených multimediálních souborů. Dialog pro výběr souboru je zachycen na obrázku 6.2. Výběrem kterékoliv položky dojde k volání metody `openFile()` třídy `FrameView`. Jako parametr je předána cesta k požadovanému souboru. Třída `FrameView` žádá knihovnu MediaLib o otevření příslušného souboru. Dojde-li k chybě, třída `FrameView` o této skutečnosti informuje třídu `MainActivity`. Hlavní třída se postará o vytvoření dialogového okna s chybovým hlášením. V případě, že operace uspěje, je aplikace připravena k zahájení přehrávání.

Tlačítko „Bench.“ (zkratka pro „Benchmark“) slouží k vyvolání testu výkonu. Test spočívá v převodu video záznamu mezi různými formáty a rozlišeními. Uživateli je opět nabídnuto několik předdefinovaných možností (viz obrázek 6.3).

Výběrem některé z položek dojde k vyvolání metody `benchmark()` třídy `FrameView`. Tato metoda je do značné míry nezávislá na ostatních částech třídy `FrameView`, které se týkají implementace přehrávače. Prvky pro ovládání playbacku tedy nemají vliv na proces testování výkonu. Je možné spustit testování výkonu i v okamžiku, kdy je přehráván jiný soubor. To ovšem zapříčiní zkreslení výsledků benchmarku, jelikož dekódování snímků pro přehrávač zabere část celkového výpočetního výkonu daného zařízení. Výsledky benchmarku jsou uživateli prezentovány formou informačního dialogu. Poskytnuty jsou tři časové údaje (v sekundách) – čas strávený dekódováním vstupního souboru, čas strávený kódováním výstupu a rovněž časový údaj zachycující celkovou dobu běhu benchmarku (viz obrázek 6.3). Je-li zařízení připojeno k počítači s debugovacím software, jsou tyto informace tisknuty i do výstupního logu, odkud je lze lze získat v textové podobě.

Ve spodní části UI se nachází ještě dvě další tlačítka. Tlačítko „Close“ lze použít k uzavření přehrávaného souboru, o což se postará metoda `closeFile` třídy `FrameView`, která požadavek na uzavření souboru předává knihovně MediaLib. Poslední tlačítko („Exit“)



Obrázek 6.3: Dialogové okno pro výběr varianty benchmarku (vlevo) a informační dialog s výsledky testu (vpravo).

slouží k uzavření demonstrační aplikace. V případě že je otevřen nějaký soubor, dojde před ukončením běhu aplikace k jeho uzavření způsobem popsaným výše.

Třída `FrameView`, která tvoří jádro přehrávače, je založena na komponentě `SurfaceView`. Tato komponenta je vhodná právě pro zobrazení grafických informací, které se v čase rychle mění. Na rozdíl od klasické třídy `View`, neprobíhá občerstvování obsahu v hlavním UI vlákně, ale ve vlákně vytvořeném speciálně pro instanci třídy `SurfaceView`. Tímto způsobem je možné dosáhnout vyšší obnovovací frekvence grafické oblasti a současně není blokována interaktivita ostatních částí UI.

Kromě dříve uvedených metod pro ovládání přehrávače a práci se soubory, disponuje třída `FrameView` několika metodami privátními. Důležitá je zejména metoda `run()`, která se stará o obnovování oblasti video snímku. V jejím těle je nejprve obrazová oblast uzamčena metodou `lockCanvas()`, tak aby při přístupu k jejímu obsahu nedocházelo ke kolizím s hlavním UI vláknem. Pokud je spuštěno přehrávání, tak v nekonečné smyčce získáváme přírustky reálného času a je-li vhodný okamžik k vykreslení dalšího snímku požádáme o jeho získání knihovnu MediaLib voláním metody `GetVideoFrameBitmap()`. Získaná informace je uložena do předpřipravené bitmapy a ta je potom bez dalších úprav předána plátnu třídy `FrameView` metodou `drawBitmap()`. Plátno je následně uvolněno metodou `unlockCanvasAndPost()` čímž je umožněno vykreslení jeho obsahu. O zpracování zvuku se zde nestaráme, jelikož knihovna MediaLib pracuje v režimu přehrávač a tento proces je v režii třídy `AudioTrack`, která je rovněž synchronizována proti reálnému času.

Metoda `benchmark()` třídy `FrameView` pracuje nad nezávislou dvojicí souborů. V jejím těle dochází k otevření vstupního i výstupního souboru metodou `OpenFile()`. Dále je proveden výběr obrazové stopy ze vstupu (`SelectStream()`) a přidáni nové obrazové

stopy do výstupního souboru (`InsertVideoStream()`). Parametry výstupního snímku jsou nastaveny dle zvolené varianty benchmarku a do výstupního souboru je zapsána hlavička s informacemi o stopě (metoda `InsertHeader()`). Nyní je již spuštěn proces překódování, kdy ve smyčce získáváme snímky ze zdrojového souboru `GetVideoFrameBitmap()` a přímo je vkládáme do souboru cílového `InsertVideoFrameBitmap()`. Před a za těmito metodami získáváme reálný čas, který je následně použit pro výpočet celkového času stráveného danou operací. Cyklus je přerušen, jakmile zdrojový soubor ohlásí, že neobsahuje další snímky. V tomto případě zavoláme metodu `InsertVideoFrameBitmap()` s hodnotou `null` místo bitmapy s daty. Tímto dáváme knihovně MediaLib vědět, že má zapsat případné bufferované informace do souboru. Na závěr uzavřeme oba soubory metodou `CloseFile()`. Třídě `MainActivity` zpřístupníme výsledky benchmarku a ta se potom postará o jejich zobrazení.

6.4 Překlad FFmpeg pro Android

Jedním z prvních problémů, který bylo v rámci implementace nezbytné vyřešit, byl překlad balíčku FFmpeg pro použití v systému Android. FFmpeg podporuje řadu cílových platforem a překlad pro většinu z nich je přímočarý proces. Na systém Android se při vývoji FFmpeg bere čím dál tím větší ohled, avšak stále není jeho podpora na takové úrovni jako v případě ostatních systémů. Zejména samotná fáze překladu a linkování není zcela automatizována a vyžaduje vhodně nastavené překladové prostředí a dodatečné zásahy do projektu.

Existuje několik předpřipravených řešení, která usnadňují překlad FFmpegu pro Android. Některá z nich jsou přímo dostupná z oficiálních wiki stránek projektu FFmpeg [21]. Je však třeba mít na paměti, že většina uvedených postupů a hotových řešení není založena na aktuální verzi FFmpegu ani aktuální verzi vývojářských nástrojů pro Android. Proto bude princip a přesný postup překladu popsán i zde, aby čtenář získal povědomí o jednotlivých krocích vedoucích k získání knihovny „šité na míru“ konkrétnímu použití na systému Android. Takto získané znalosti bude možné aplikovat i při překladu budoucích verzí FFmpegu s využitím poslední dostupné verze Android NDK.

Postup zde uvedený je jedno z více možných řešení. Jedná se o překlad přímo v adresářovém stromu zdrojových kódů systému Android, což není pro tento proces nezbytně nutné, avšak jedná se o způsob flexibilnější a umožňuje mimo jiné snadné přidání doplňujících knihoven, s jejichž volitelnou podporou FFmpeg počítá – zejména pak jmenujme projekt VideoLAN x264 [27], což je populární enkodér pro formát H.264/MPEG-4 AVC. Tento postup vychází z návodů [5] a [14] dostupných z wiki stránek FFmpegu a projektu *FFmpeg for Android* [23].

Proces překladu bude vyžadovat následující softwarové prostředky:

- operační systém unixového typu (použit systém Linux, Ubuntu 13.10 AMD64),
- Android NDK (použita verze r9c),
- balíček se zdrojovými kódy FFmpegu (použita verze 2.2 a 2.2.1).

Veškerý použitý software je dostupný online, avšak bude poskytnut (kromě operačního systému) i na disku přiloženému k této práci.

Instalace Android NDK spočívá v rozbalení získaného archivu do libovolného (vhodného) umístění, na které se budeme v následujícím textu odkazovat označením `$NDK`. Zdrojové kódy projektu FFmpeg je potom třeba rozbalit do adresáře `$NDK/sources`. Zde by

tedy měl být vytvořen podadresář `ffmpeg-X.Y.Z`, kde X.Y.Z označuje číslo verze (např. `ffmpeg-2.2`). Z názvu adresáře můžeme toto číslo verze pro jednoduchost odstranit, tak že ve výsledku získáme adresářovou strukturu `$NDK/sources/ffmpeg`, která již přímo obsahuje jednotlivé složky a soubory projektu.

V kořenovém adresáři projektu FFmpeg je třeba upravit soubor `configure`. Je nutné modifikovat chování při překladu tak, aby název souboru, který je výstupem překladu neobsahoval za příponou ještě sufix s číslem verze – tento formát názvu souboru totiž není nástroji při překladu pro systém Android podporován. Konkrétně je tedy třeba následující řádky

```
SLIBNAME_WITH_MAJOR='$(SLIBNAME).$(LIBMAJOR)'
LIB_INSTALL_EXTRA_CMD='$$(RANLIB)␣"$(LIBDIR)/$(LIBNAME)"'
SLIB_INSTALL_NAME='$(SLIBNAME_WITH_VERSION)'
SLIB_INSTALL_LINKS='$(SLIBNAME_WITH_MAJOR)␣$(SLIBNAME)'
```

modifikovat do této podoby

```
SLIBNAME_WITH_MAJOR='$(SLIBPREF)$(FULLNAME)-$(LIBMAJOR)$(SLIBSUF)'
LIB_INSTALL_EXTRA_CMD='$$(RANLIB)␣"$(LIBDIR)/$(LIBNAME)"'
SLIB_INSTALL_NAME='$(SLIBNAME_WITH_MAJOR)'
SLIB_INSTALL_LINKS='$(SLIBNAME)'
```

Následuje vytvoření skriptu, který vyvolá vlastní překlad. V adresáři `$NDK/sources` tedy vytvoříme textový soubor s názvem `build_android.sh`. Jednoduchá verze tohoto skriptu může vypadat následovně:

```
#!/bin/bash
NDK=
SYSROOT=$NDK/platforms/android-9/arch-arm/
TOOLCHAIN=$NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/linux-x86_64
function build_ffmpeg
{
./configure \
    --prefix=$PREFIX \
    --enable-shared \
    --disable-static \
    --disable-ffmpeg \
    --disable-ffplay \
    --disable-ffprobe \
    --disable-ffserver \
    --disable-avdevice \
    --disable-doc \
    --disable-symver \
    --cross-prefix=$TOOLCHAIN/bin/arm-linux-androideabi- \
    --target-os=linux \
    --arch=arm \
    --enable-cross-compile \
    --sysroot=$SYSROOT \
    --extra-cflags="-O3␣-fpic␣-fasm␣-$ADDI_CFLAGS" \
    --extra-ldflags="$ADDI_LDFLAGS" \
    $ADDITIONAL_CONFIGURE_FLAG
make clean
make
make install
}
PREFIX=$(pwd)/android/arm
build_ffmpeg
```

Důležité části skriptu si popíšeme v následujících podkapitolách.

6.4.1 Nastavení cílové architektury a platformy

Proměnná `NDK` musí reflektovat skutečné umístění kořenového adresáře Android NDK, její hodnotu je tedy třeba individuálně nastavit.

Následující text se bude věnovat překladu kódu napsaného v jazyce C a cílovou architekturou budou procesory založené na architektuře ARM (viz kapitola 2.4.1).

Nastavení `SYSROOT` se odvíjí od cílové platformy a architektury. Platforma se nenastavuje dle čísla verze systému Android, ale na základě revize API rozhraní (API Level). Informaci o tom jakou revizí API rozhraní disponuje konkrétní verze systému, lze nalézt v aktuální podobě na webu Android Developers [9]. V našem případě je cílovou platformou Android 2.3 Gingerbread, který poskytuje API Level 9 – čemuž odpovídá hodnota `android-9` (dostupné možnosti lze zjistit nahlédnutím do podadresáře `$NDK/platforms`. Cílovou architekturou může být ARM, x86 nebo MIPS, čemuž odpovídají hodnoty `arch-arm`, `arch-x86` a `arch-mips` v daném pořadí.

Proměnná `TOOLCHAIN` definuje cestu k sadě nástrojů (tzv. toolchain), která má být při překladu kódu pro cílovou architekturu použita. Odvíjí se od použitého hostitelského systému (zde 64-bitová verze systému Linux) a pochopitelně od samotné cílové architektury. V uvedeném případě bude tedy použita sada s označením `arm-linux-androideabi-4.8`, kde číselný sufix (`4.8`) označuje číslo verze použitého překladače (typicky by měla být zvolena poslední dostupná verze). Poslední část definované cesty `linux-x86_64` říká, že bude použito sestavení nástrojů pro 64-bitovou verzi Linuxu. Seznam dostupných sad je opět možné získat nahlédnutím do příslušného adresáře (`toolchains`), přičemž dostupné možnosti se budou lišit v závislosti na tom, pro který hostitelský systém je určen použitý balíček Android NDK.

6.4.2 Konfigurace balíčku FFmpeg

Volba komponent, které budou součástí výsledného sestavení knihovny FFmpeg, probíhá pomocí parametrů příkazu `./configure`. Určitou komponentu lze povolit definováním parametru `-enable-NAME`, kde *NAME* odpovídá jejímu názvu. Vyloučit komponentu z překladu lze použitím parametru `-disable-NAME`. Seznam jednotlivých komponent jako jsou kodeky, kontejnery, utility apod. a jejich názvů lze nalézt v dokumentaci projektu FFmpeg [22]. V dokumentaci lze také získat informaci o tom, zda je daná komponenta ve výchozím nastavení povolena či zakázána.

Součástí balíčku je i sada nástrojů, jejichž stručnému popisu se věnovala podkapitola 3.1.1. Sestavujeme-li knihovnu pro systém Android je nezbytné většinu těchto utilit vynechat, protože v současné době nejsou v tomto prostředí použitelné a typicky kvůli chybějícím závislostem ani přeložitelné (přehrávač *ffplay* je například závislý na sadě knihoven SDL). Pro tuto práci není žádný z nástrojů nijak významný, proto jsou *ffmpeg*, *ffplay*, *ffprobe* a *ffserver* zakázány.

Dalším důležitým nastavením je volba výstupu překladu. Pokud hodláme FFmpeg využívat v podobě knihovny, jako v našem případě, máme na výběr mezi knihovnou statickou nebo dynamickou. Tato volba ovlivní nejen formát a způsob použití knihovny, ale rovněž může mít vliv na volbu licence, pod kterou může být projekt využívající knihovnu FFmpeg šířen. Statickou knihovnu lze získat použitím definice `--enable-static`, dynamickou použitím `--enable-shared`, přičemž obě možnosti lze uvést současně. Tento projekt a uvedený příklad využívá FFmpeg v podobě dynamické knihovny.

Uvedený skript obsahuje ještě několik dalších důležitých parametrů příkazu `./configure`. Jedním z nich je definice cílového systému – FFmpeg nenabízí speciální volbu pro An-

droid, místo toho je volbou `--target-os=linux` nastaven systém Linux, na kterém je Android založen. Volbou –arch=arm skriptu starajícímu se o překlad FFmpegu sdělujeme, že cílovou architekturou je ARM. Jelikož překládáme FFmpeg v rámci stromu zdrojových kódů operačního systému, je třeba tento typ sestavení povolit o což se starají parametry `--enable-cross-compile` a `--cross-prefix`.

6.4.3 Parametry překladače a optimalizace

Poslední tři parametry v ukázkovém skriptu slouží k definovaní dodatečných argumentů určených pro samotný překladač. Tyto parametry lze mimo jiné použít k optimalizaci výstupního kódu, čemuž se budeme věnovat v této podkapitole. Od čtenáře se očekává znalost hardwarové architektury ARM (základní informace z této oblasti poskytuje kapitola 2.4).

Výše uvedený skript je sice v praxi použitelný, avšak byl z důvodu přehlednosti výrazně zjednodušen. Chybí zde mimo jiné právě parametry zajišťující optimalizaci kódu pro výkon a pro konkrétní architekturu. Úplný skript, použitý v překladu FFmpegu pro tento projekt, lze nalézt v elektronické příloze této práce.

Balíček Android NDK využívá k překladu kompilátory ze sady GCC (GNU Compiler Collection). Konkrétně se jedná o nástroj *gcc* pro překlad kódu psaného v jazyce C a nástroj *g++* pro jazyk C++. Zde se budeme věnovat překladu s využitím programu *gcc* a optimalizaci pro architekturu ARM. Dále uvedená nastavení jsou překladači předána pomocí parametru `--extra-cflags`, který budeme dále označovat zjednodušeně jako *CFLAGS*.

Nejprve se budeme věnovat parametrům CGLAGS, které nám umožní bezproblémový překlad knihovny FFmpeg pro Android. Jelikož výstupem překladu bude dynamická knihovna, je na místě překladači sdělit, aby generoval kód nezávislý na pozici (Position-independent code – PIC), což je forma kódu, kterou tyto knihovny vyžadují. Toho dosáhneme definicí `-fpic`. Dále je třeba použít parametr `-fasm`, který zajistí, že překladač bude akceptovat klíčové slovo `asm` v místě, kde je vložen kód psaný v jazyce symbolických instrukcí (JSI). Toto je nezbytné, jelikož některé součásti FFmpegu právě tímto způsobem kód JSI vkládají. Rovněž doplníme definici `ANDROID` pro preprocesor, čímž dojde ke správnému použití pasáží kódu specifických pro Android. Překládáme-li knihovnu mimo režim ladění je vhodné rovněž přidat i definici `NDEBUG` (v opačném případě použijeme `DEBUG`). Tyto definice lze pro preprocesor přichystat v rámci CFLAGS a to přidáním prefixu `-D` před název dané definice – tedy v našem případě `-DANDROID` a `-DNDEBUG`.

Prvním krokem při optimalizaci programu pro rychlost je povolení některé z předdefinovaných úrovní optimalizace. Toho dosáhneme přidáním definice `-O1`, `-O2`, `-O3` nebo `-Ofast` do CFLAGS. Vyšší číslo zde značí vyšší stupeň optimalizace výstupního kódu pro rychlost. Detaily k jednotlivým definicím lze nalézt v dokumentaci projektu GCC [24]. Parametr `-Ofast` má účinek shodný s `-O3`, navíc však aktivuje optimalizace matematických operací. Obdobného efektu lze dosáhnout definicí `-O3` v kombinaci s `-ffast-math`. Je třeba mít na paměti, že tato skupina optimalizací není zcela v souladu s pravidly a specifikacemi matematických funkcí, tak jak jsou definovány dle standardů IEEE a ISO. Většinou je tedy vhodné řádně testovat chování kódu generovaného s tímto parametrem (při ladění se na to může díky odlišné konfiguraci překladače snadno zapomenout a případný problém se tak může projevit až ve finálním optimalizovaném sestavení programu).

Nyní se dostáváme k optimalizacím specifickým pro procesory z rodiny ARM. Zdaleka nejdůležitější je zde specifikování správné architektury dle cílového zařízení. Chceme-li obsáhnout širokou škálu těchto zařízení a přitom zachovat pokud možno co nejvyšší stu-

peň optimalizace pro každé z nich, nevyhneme se opakovanému sestavení knihovny s různými parametry překladu. Mezi architektury s nejvýraznějším zastoupením patří ARMv7 a ARMv6. Každá z těchto dvou architektur existuje v několika variantách – pro ARMv7 můžeme zmínit ARMv7-A, která zahrnuje velmi populární jádra Cortex-A5, A7, A8, A9 a další. Překladač *gcc* pro ARM umožňuje specifikovat požadovanou architekturu volbou `--march`. Tato volba definuje instrukční sadu, ze které budou vybírány instrukce při generování výstupního kódu. Můžeme zvolit například základní architekturu ARMv7 volbou `--march=arm7`. Aplikaci obsahující takový nativní kód pak bude možné spustit na všech variantách této architektury – tedy např. ARMv7-A, ARMv7-R a ARMv7-M, v kódu však nebudou použity instrukce specifické pro jednotlivé tyto varianty. Zvolíme-li konkrétní variantu (např. ARMv7-A volbou `--march=arm7-a`) využijeme i specifická rozšíření instrukční sady, což může být výhodné z hlediska výkonu, avšak takový kód pravděpodobně nebude možné spustit na procesorech založených na jiné variantě ARMv7. Při volbě architektury tedy záleží na škále zařízení, na která cílíme a na tom, do jaké míry je pro nás důležitá míra optimalizace.

Kromě volby specifické architektury (instrukční sady) je možné v procesu optimalizace pokročit ještě o stupeň výše a to specifikací konkrétního mikroprocesorového jádra. To nám umožňuje parametr `-mtune`. Volba určitého jádra přitom neomezí použitelnost kódu pouze na daný procesorech, kompatibilita s ostatními jádry spadajícími pod stejnou architekturu zůstane zachována. Parametr překladači pouze říká, pro které jádro se má pokusit kód „vyladit“. Provádění programu na procesoru s tímto jádrem pak bude pravděpodobně optimálnější v porovnání s ostatními jádry kategorie. Vyladění kódu pro určité jádro se však může negativně podepsat na rychlosti jeho provádění na jádrech ostatních (ve srovnání se situací, kdy parametr nebyl vůbec použit). Je dobré proto znát hardwarové vybavení cílových zařízení, optimalizovat kód pro jádro s největším procentuálním zastoupením, a přesvědčit se, zda vliv optimalizace nemá přílišný dopad na ostatní zařízení. Výhodnou volbou rovněž může být nastavení optimalizace na hodnotu `-mtune=generic-arch`, která by se měla postarat o optimalizaci pro co nejširší mix populárních jader dané architektury. Chování tohoto parametru závisí na použité verzi překladače, v každé verzi může tedy být zvýhodněna jiná podmnožina jader v rámci architektury.

Další možností optimalizace je volba typu použitých instrukcí. Jak bylo uvedeno, procesory ARM disponují instrukcemi typu ARM a Thumb (v různých revizích). Nelze předem říct, která z nich bude výkonově výhodná, záleží to na použitém hardware a na povaze překládaného kódu. Výsledky se tedy budou lišit případ od případu a pro konkrétní nastavení je třeba se rozhodnout na základě testování. Jako výchozí jsou použity instrukce typu ARM, přepnutí překladače do režimu generování instrukcí typu Thumb spočívá v použití parametru `-mthumb`. Implicitně je zakázáno instrukce ARM a Thumb v rámci jednoho programu kombinovat, toto chování je však možné změnit parametrem `-mthumb-interwork`.

Více informací o parametrech `march`, `mtune`, `mthumb` a dalších souvisejících je možné získat v dokumentaci *gcc* [24] v sekci *ARM Options* věnované nastavením specifickým pro procesory ARM.

Disponuje-li cílová architektura některým z volitelných koprocesorů VFP či NEON, je vhodné zajistit přeložený kód ve verzi optimalizované pro využití těchto rozšíření instrukční sady. Parametrem `-mfpu` lze překladači sdělit, aby tyto instrukce využil. Podporu koprocesoru VFP v určité verzi, například VFPv3 pro ARMv7-A, lze aktivovat nastavením parametru na hodnotu `-mfpu=vfpv3`. Procesory architektury ARMv7-A (a vyšší) mohou být volitelně vybaveny také SIMD koprocesorem NEON. Tento procesor značně urychluje nejen práci s vektory, ale i operace s čísly s pevnou a hlavně plovoucí řádovou čárkou.

Nevyžadujeme-li podporu dvojité přesnosti, lze očekávat, že kód s povoleným rozšířením NEON bude rychlejší i ve srovnání s optimalizací pro VFPv3. K aktivaci instrukcí NEON sice stačí nastavit parametr `-mfpu=neon`, avšak aby došlo k urychlení i o operací s plovoucí řádovou čárkou, je třeba definovat i `-funsafe-math-optimizations`. Tento parametr je nezbytný, jelikož specifikace NEON není zcela v souladu se standardem IEEE 754. Při použití tohoto nastavení je vždy vhodné ověřit, zda se vygenerovaný kód chová korektně. Pokud je výše uvedená skupina parametrů rozšířena o `-ftree-vectorize`, překladač se bude snažit paralelizovat některé skalární operace využitím instrukcí pro práci s vektory. V některých případech tak lze dosáhnout dodatečného zvýšení efektivity kódu. Doplňující informace lze získat v [24] a [2], ze kterých tato část textu čerpá.

Kapitola 7

Testování a měření výkonu

7.1 Použitá zařízení

Nedílnou součástí této práce je rovněž testování vytvořené knihovny a aplikace (dále jen „aplikace“) v různých hardwarových a softwarových podmínkách.

Pro tyto testy a měření výkonu byla použita zařízení spadající do cílové skupiny definované v kapitole 5.2. Použité přístroje lze zařadit do současné střední a vyšší třídy (stav k první polovině roku 2014). Tomuto zařazení odpovídá jejich hardwarové vybavení. Celkem byla při testování použita čtyři zařízení – tři mobilní telefony (Sony Xperia P, Lenovo A760, Google Nexus 5) a jeden tablet (Google Nexus 7). Jejich hardwarové a softwarové vybavení je uvedeno v tabulce 7.1. Informace o jednotlivých zařízeních byla získána z webových stránek výrobců.

Název zařízení	**Verze sys. Android**	**Kapacita RAM**	**CPU**	**GPU**
Sony Xperia P	4.1.2	1 024 MB	ARMv7-A 2x Cortex-A9 1 000 MHz	ARM Mali-400MP1
Lenovo A760	4.1.2	1 024 MB	ARMv7-A 4x Cortex-A5 1 200 MHz	Qualcomm Adreno 203
Google Nexus 5	4.4.2	2 048 MB	ARMv7-A 4x Krait 400 2 300 MHz	Adreno 330
Google Nexus 7	4.4.2	1 024 MB	ARMv7-A 4x Cortex-A9 1 300 MHz	Nvidia Geforce ULP

Tabulka 7.1: Specifikace zařízení použitých pro testování a měření výkonu.

Aplikaci bylo možné bez potíží spustit na všech uvedených zařízeních. Tato zařízení byla rovněž dostatečně výkonná pro přehrávání ukázkových nahrávek v reálném čase, bez trhání či jiných potíží.

Všechny použité přístroje jsou založeny na architektuře ARMv7-A. Zařízení vyhovujících parametrů založené na ARMv6 se nepodařilo pro testování získat. Použitá zařízení byla v době testování vybavena systémem Android verze 4.1.2 a 4.4.2. Testování pod starší

revizí systému proto proběhlo pouze v emulátoru. Virtuální zařízení bylo nakonfigurováno tak, aby využívalo obraz systému Android verze 2.3.3. V tomto prostředí aplikace pracovala korektně.

7.2 Měření výkonu

K měření výkonu implementovaného řešení na různém hardware byla využita funkce „Benchmark“ vestavěná do demonstrační aplikace. Tato funkce realizuje překódování videa mezi různými formáty a rozlišeními. Měření proběhlo na všech přístrojích uvedených v minulé kapitole. Na každém z nich byly spuštěny všechny nabízené testy (celkem 6). Parametry jednotlivých testů jsou uvedeny v tabulce 7.2. Všechna vstupní videa mají shodnou délku trvání – 15 sekund. Počet snímků za vteřinu je rovněž pro všechny vzorky stejný – 25 FPS.

Název testu	**Vstup**		**Výstup**	
	Formát	Rozlišení	Formát	Rozlišení
H264 1280 → MPEG2 640	H.264	1280×720	MPEG2	640×480
MPEG2 1280 → MPEG4 640	MPEG2	1280×720	MPEG4	640×480
MPEG4 1280 → MPEG2 640	MPEG4	1280×720	MPEG2	640×480
H264 640 → MPEG2 1280	H.264	640×480	MPEG2	1280×720
MPEG2 640 → MPEG4 1280	MPEG2	640×480	MPEG4	1280×720
MPEG4 640 → MPEG2 1280	MPEG4	640×480	MPEG2	1280×720

Tabulka 7.2: Specifikace jednotlivých použitých testů.

Jelikož je systém Android multitaskingovým prostředím a testovaná aplikace nebyla jediným spuštěným procesem, opakovali jsme každý test 10 krát, přičemž výsledky jednotlivých běhů se mírně lišily. Veškeré zde uvedené údaje jsou proto průměrnými hodnotami vypočtenými z údajů získaných při jednotlivých bězích. Časy strávené čtením a dekódováním vstupního souboru jsou uvedeny v tabulce 7.3. Doba strávená kódováním a zápisem do výstupního souboru je pro jednotlivá zařízení uvedena v tabulce 7.4. Poslední tabulka (7.5) pak uvádí celkovou délku trvání jednotlivých benchmarků.

Název testu	**Zařízení**			
	Xperia P	A760	Nexus 5	Nexus 7
H264 1280 → MPEG2 640	16,3	7,5	3,0	10,5
MPEG2 1280 → MPEG4 640	18,9	9,2	3,9	10,3
MPEG4 1280 → MPEG2 640	12,7	7,6	5,4	9,1
H264 640 → MPEG2 1280	18,7	15,5	9,2	12,4
MPEG2 640 → MPEG4 1280	22,3	12,6	10,9	12,3
MPEG4 640 → MPEG2 1280	23,7	14,7	10,3	12,2

Tabulka 7.3: Výsledky výkonnostních testů na jednotlivých zařízeních. Hodnoty platné pro fázi dekódování (v sekundách).

Na základě naměřených hodnot můžeme učinit několik poznatků. Předně můžeme srovnat náročnost procesu kódování a dekódování pro stejný kodek a rozlišení. Pro velké množství případů zabralo kódování 5 krát až 10 krát více času nežli proces dekódování. Toto je pochopitelné, jelikož ve většině případů je důležitější jak je kodek efektivní při přehrá-

vání (dekódování). S přihlédnutím k tomuto faktu je většina kodeků navržena a rovněž implementace bývá na straně dekodéru kvalitnější (z hlediska efektivity kódu).

Dále můžeme hodnotit použitelnost implementované knihovny pro přehrávání multimediálních záznamů v reálném čase. Vidíme, že dekódování 15 sekundové nahrávky zabralo ve většině případů méně času v porovnání s dobou jejího trvání. Výjimku zde tvoří testy spuštěné na telefonu Xperia P, kde byly časy dekódování pro většinu formátů větší než je délka nahrávky.

Zajímavé se může zdát, že dekódování videa s nižším rozlišením zabralo více času ve srovnání s videem ve vyšším rozlišení a stejném formátu. Toto je způsobeno fází převzorkování video snímku, která byla v případě těchto benchmarků učiněna již na straně dekodéru. Zvětšení rozlišení snímku z 640×480 na 1280×720 bodů je operace časově náročnější nežli postup opačný.

Název testu	**Zařízení**			
	Xperia P	A760	Nexus 5	Nexus 7
H264 1280 → MPEG2 640	34,6	23,1	9,5	26,0
MPEG2 1280 → MPEG4 640	44,5	21,9	22,4	40,8
MPEG4 1280 → MPEG2 640	26,8	22,7	15,2	20,7
H264 640 → MPEG2 1280	77,8	60,7	38,9	58,8
MPEG2 640 → MPEG4 1280	102,2	58,2	55,4	73,6
MPEG4 640 → MPEG2 1280	112,5	60,9	43,5	60,2

Tabulka 7.4: Výsledky výkonnostních testů na jednotlivých zařízeních. Hodnoty platné pro fázi kódování (v sekundách).

Získaná data můžeme použít i k porovnání výkonu jednotlivých zařízení. Přístroje vybavené výkonnějším procesorem taktovaným na vyšší frekvenci zde dosáhly nižších časů. Z použitých zařízení disponuje nejvýkonnějším procesorem Google Nexus 5, který dosáhl s velkým předstihem nejnižších časů. Telefon Lenovo A760 a tablet Google Nexus 7 dosáhly ve většině testů srovnatelných výsledků. Telefon Xperia P je vybaven pouze dvoujádrovým procesorem taktovaným na 1 GHz, což jsou parametry výrazně horší ve srovnání se zbytkem použitých zařízení. Tento fakt se odrazil i v naměřených hodnotách, které jsou jednoznačně nejvyšší.

Název testu	**Zařízení**			
	Xperia P	A760	Nexus 5	Nexus 7
H264 1280 → MPEG2 640	50,9	30,7	12,6	36,6
MPEG2 1280 → MPEG4 640	63,5	31,1	26,3	51,2
MPEG4 1280 → MPEG2 640	39,5	30,3	20,7	29,9
H264 640 → MPEG2 1280	96,6	76,2	48,1	71,2
MPEG2 640 → MPEG4 1280	124,8	70,8	66,4	85,9
MPEG4 640 → MPEG2 1280	136,3	75,7	53,8	72,5

Tabulka 7.5: Výsledky výkonnostních testů na jednotlivých zařízeních. Celkové časy (v sekundách).

Kapitola 8

Zhodnocení a závěr

Cílem práce bylo seznámit se s nativními knihovnami pro práci s multimediálními soubory, získat znalosti o platformě Android a pro tento systém navrhnout a implementovat zastřešující multimediální knihovnu.

Platformou Android a vývojem aplikací pro ni se zabývala první část této práce. Důraz je kladen zejména na vývoj nativní části aplikace a specifika s tím spojená. Čtenáři jsou dále poskytnuty informace o hardwarové stránce platformy Android, které jsou významné z hlediska překladu nativního kódu pro cílová zařízení.

Ve druhém tématickém celku práce jsou diskutovány dostupné multimediální knihovny pro platformu PC a zařízení se systémem Android. Text se zaměřuje na nejpopulárnější řešení – balíčky FFmpeg a libav. Popsány jsou jednotlivé součásti a jejich použití. Přihlédnuto je rovněž k existujícím projektům s podobným zaměřením jako má tato práce.

Jako základ knihovny byl vybrán balíček FFmpeg. Po podrobné analýze jeho fungování a použití byl učiněn prvotní návrh knihovny MediaLib. Na základě toho byl implementován její prototyp včetně jednoduché demonstrační aplikace. Zkušenosti získané v rámci tohoto procesu byly zpětně promítnuty do další iterace návrhu. Tento návrh je náplní třetího tematického celku této diplomové práce. Na základě nového návrhu byla vystavena finální podoba knihovny MediaLib. Ta splňuje požadavky kladené zadáním práce – umožňuje dekódovat i kódovat multimediální obsah a poskytuje rozhraní pro aplikaci grafických filtrů. Implementaci je věnována samostatná kapitola. Ta mimo jiné poskytuje i informace týkající se překladu knihovny FFmpeg pro platformu Android.

V kapitole „Testování a měření výkonu“ je zhodnocena praktická výkonnost implementované knihovny na různém hardware. K předvedení schopností knihovny a samotnému měření výkonu je využita demonstrační aplikace, která je rovněž součástí řešení. Tato aplikace prokázala, že výsledné řešení je použitelné i pro přehrávání multimediálního obsahu v reálném čase.

Výsledné řešení bude po jeho odevzdání a obhajobě zveřejněno v podobě zdrojových kódů, nebude-li to v rozporu s licenčním ujednáním.

Literatura

[1] ARMv5 Architecture Reference Manual [online]. https://silver.arm.com/download/download.tm?pv=1073121, 2005 [cit. 28. května 2014].

[2] Introducing NEON [online]. http://infocenter.arm.com/help/topic/com.arm.doc.dht0002a/DHT0002_Aintroducing_neon.pdf, 2009 [cit. 28. května 2014].

[3] ARMv6-M Architecture Reference Manual [online]. https://silver.arm.com/download/download.tm?pv=1102513, 2010 [cit. 28. května 2014].

[4] ARMv7-M Architecture Reference Manual [online]. https://silver.arm.com/download/download.tm?pv=1111932, 2010 [cit. 28. května 2014].

[5] How to Build FFmpeg for Android [online]. http://www.roman10.net/how-to-build-ffmpeg-for-android/, 2011 [cit. 28. května 2014].

[6] ARMv7-AR Architecture Reference Manual [online]. https://silver.arm.com/download/download.tm?pv=1299246, 2012 [cit. 28. května 2014].

[7] ARMv8-A Architecture Reference Manual [online]. https://silver.arm.com/download/download.tm?pv=1541018, 2012 [cit. 28. května 2014].

[8] Android Developers na Archive.com [online]. https://web.archive.org/web/20131202013657/http://developer.android.com/about/dashboards/index.html, 2013 [cit. 28. května 2014].

[9] Android Developers [online]. http://developer.android.com/, 2013 [cit. 28. května 2014].

[10] Android [online]. http://www.android.com, 2013 [cit. 28. května 2014].

[11] Android Security Overview [online]. http://source.android.com/devices/tech/security/, 2013 [cit. 28. května 2014].

[12] ARM Assembly language [online]. http://www.shervinemami.info/armAssembly.html, 2013 [cit. 28. května 2014].

[13] Domovská stránka projektu jjmpeg [online]. https://code.google.com/p/jjmpeg/, 2013 [cit. 28. května 2014].

[14] How to Build ffmpeg with NDK r9 [online]. http://www.roman10.net/how-to-build-ffmpeg-with-ndk-r9/, 2013 [cit. 28. května 2014].

[15] ARM - Processors [online]. http://www.arm.com/products/processors/index.php, 2014 [cit. 28. května 2014].

[16] Domovská stránka projektu FFmpeg 4 Android [online]. http://ffmpeg4android.netcompss.com/, 2014 [cit. 28. května 2014].

[17] Domovská stránka projektu FFmpeg [online]. http://www.ffmpeg.org/, 2014 [cit. 28. května 2014].

[18] Domovská stránka projektu libav [online]. http://www.libav.org/, 2014 [cit. 28. května 2014].

[19] Domovská stránka projektu OpenCV [online]. http://opencv.org/, 2014 [cit. 28. května 2014].

[20] Domovská stránka projektu OpenMAX [online]. http://www.khronos.org/openmax/, 2014 [cit. 28. května 2014].

[21] FFmpeg Bug Tracker and Wiki [online]. http://trac.ffmpeg.org/wiki, 2014 [cit. 28. května 2014].

[22] FFmpeg Documentation [online]. http://www.ffmpeg.org/documentation.html, 2014 [cit. 28. května 2014].

[23] FFmpeg for Android [online]. http://sourceforge.net/projects/ffmpeg4android/, 2014 [cit. 28. května 2014].

[24] GCC online documentation [online]. http://gcc.gnu.org/onlinedocs/, 2014 [cit. 28. května 2014].

[25] Java Native Interface Specification [online]. http://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/jniTOC.html, 2014 [cit. 28. května 2014].

[26] Unity - Mobile (Android) Hardware Stats [online]. http://stats.unity3d.com/mobile/index-android.html, 2014 [cit. 28. května 2014].

[27] VideoLAN x264 [online]. http://www.videolan.org/developers/x264.html, 2014 [cit. 28. května 2014].

[28] Grant, A.: *Android 4 Průvodce programováním mobilních aplikací*. Albatros Media, 2013, iSBN 978-80-251-3782-6.

[29] Ujbányai, M.: *Programujeme pro Android*. Grada, 2012, iSBN 978-80-247-3995-3.

[30] Vyoral, M.: *Editor videa pro platformu Android*. Diplomová práce, FIT VUT v Brně, 2011.