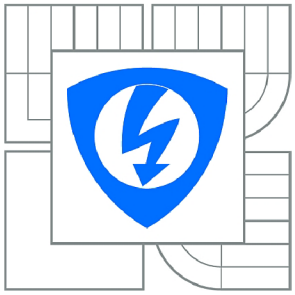
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF CONTROL AND INSTRUMENTATION

APLIKACE V OS ANDROID PRO TINY6410 SDK

AN ANDROID APPLICATION FRO TINY6410 SDK

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

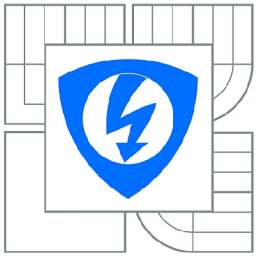
AUTOR PRÁCE — Bc. JAKUB ARM

AUTHOR

VEDOUCÍ PRÁCE — Ing. TOMÁŠ MACHO, Ph.D.

SUPERVISOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Diplomová práce

magisterský navazující studijní obor
Kybernetika, automatizace a měření

Student:	Bc. Jakub Arm	***ID:***	115139
Ročník:	2	***Akademický rok:***	2013/2014

NÁZEV TÉMATU:

Aplikace v OS Android pro Tiny6410 SDK

POKYNY PRO VYPRACOVÁNÍ:

1. Seznamte se vývojovým kitem Tiny6410 a s operačním systémem Android.
2. Vytvořte metodiku pro změření důležitých parametrů operačního systému Android. Zaměřte se na zjištění doby přepnutí kontextu a určení časové náročnosti paměťových operací.
3. Pomocí vytvořené metodiky proveďte měření doby přepnutí kontextu a časové náročnosti paměťových operací pro systém Android. Naměřené výsledky vyhodnoťte.
4. Na základě výkonnostních parametrů a zkušeností získaných se systémem Android pro danou platformu navrhněte, implementujte a odlaďte software pro ovládání USB osciloskopu Hantek DSO-2150.

DOPORUČENÁ LITERATURA:

Cinar, O.: Android Apps with Eclipse. Apress, 1 edition, June 2012, ISBN-10: 1430244348.

Furber, S.: ARM System-on-Chip Architecture. Addison-Wesley Professional, 2 edition, August 2000, ISBN-10: 0201675196.

Termín zadání:	10.2.2014	***Termín odevzdání:***	19.5.2014

Vedoucí práce: Ing. Tomáš Macho, Ph.D.
Konzultanti diplomové práce:

doc. Ing. Václav Jirsík, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

Abstrakt

Tato práce se zabývá vlastnostmi populárního operačního systému Android nainstalovaného na vývojovém kitu Tiny6410 rodiny FriendlyARM. Operační systém i vývojový kit jsou koncipovány pro nasazení v uživatelských multimediálních embedded aplikací s podporou komunikačních technologií. Cílem této práce je přinést ucelený pohled na tuto sestavu a na této sestavě změřit základní parametry operačního systému, jako je doba přepnutí kontextu a doba alokace či průchodu polem v paměti. Dalším cílem práce je vytvořit na této sestavě uživatelskou aplikaci, která bude ovládat a zobrazovat naměřená data z připojeného USB osciloskopu DSO-2150 firmy Hantek. Tato aplikace bude také demonstrovat použití všech programátorských prostředí, které má vývojář k dispozici, se zaměřením na jejich optimální využití.

Klíčová slova

Android, čas přepnutí kontextu, Tiny6410 SDK, FriendlyARM, OpenGL ES, osciloskop DSO-2150, diplomová práce, VUT Brno.

Abstract

This thesis aims generally at getting positives and negatives of popular operating system Android running on development kit Tiny6410 which is a part of FriendlyARM family. The operating system and development kit are designed to be used in user multimedia embedded applications with communication features. The goal of this thesis is to take a look on this framework and to measure main parameters of the operating system, specifically to measure switching context time and time for allocation and operation of memory block. Next goal is to create user application which will control and monitor connected oscilloscope DSO-2150 by Hantek via USB interface. This application will also demonstrate positives and negatives of all programming enviroments, which are available, for user application development with consideration of their optimal use.

Keywords

Android, switching context time, Tiny6410 SDK, FriendlyARM, OpenGL ES, oscilloscope DSO-2150, master's thesis, VUT Brno.

Bibliografická citace:

ARM, J. Aplikace v OS Android pro Tiny6410 SDK. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2014. 68 s. Vedoucí diplomové práce Ing. Tomáš Macho, Ph.D..

Prohlášení

„Prohlašuji, že svou diplomovou práci na téma Aplikace v OS Android pro Tiny6410 SDK jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.“

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne: **12. května 2014** .....

podpis autora

Poděkování

Děkuji vedoucímu diplomové práce Ing. Tomášovi Machovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

V Brně dne: **12.května 2014** .....

podpis autora

Obsah

1 ÚVOD

V dnešním světě automatizace je kladen velký důraz na inovaci. Právě inovace vede k vývoji lepších a efektivnějších technologií, které jsou jednou z dnešních významných komodit každého státu či společenství.

Každý funkční celek automatizace je postaven na symbióze hardwarové a softwarové části za účelem dosažení maximálního synergického efektu. Je tedy přínosné věnovat více úsilí optimalizaci a zdokonalování softwarové části, když už hardwarovou část máme víceméně danou.

Základní „tvůrčí“ součástkou každého elektrického embedded zařízení je bezesporu procesor, a to v jakékoliv podobě. Při použití vyšších vybavenějších procesorů je vhodné namísto specializovaného firmware použít jako základní softwarovou vrstvu operační systém. Právě operační systém je hlavním vykonavatelem požadovaných operací a správcem daného procesoru. Softwarově zpřístupňuje všechny jeho implementované funkce a optimalizuje způsob jejich využití. Jedním z operačních systémů pro embedded zařízení je systém Android.

Android je otevřený a v dnešní době populární operační systém, který je nejčastěji nasazován hlavně v mobilních zařízení – smartphone, PDA, navigace či tablet. Tento systém je zejména populární hlavně díky přívětivému uživatelskému prostředí, díky své schopnosti přehrávat běžný multimediální obsah či podporou standardních komunikačních a moderních interaktivních technologií, jako jsou dotykové obrazovky. Z hlediska vývojáře uživatelských aplikací je systém populární, jelikož lze naportovat na procesory mnohých architektur, přičemž zachovává jednotné rozhraní pro ovládání implementovaných funkcí, jako je např. podpora dotykové obrazovky, kamera či podpora WiFi technologií. V neposlední řadě je populární, protože umožňuje vývojářům rychlé vytváření aplikací ve standardním programovacím prostředí, a tím zmenšuje jejich náklady.

1.1 Cíl práce

Cílem práce je seznámit se s operačním systémem Android, který je nainstalovaný na vývojovém kitu Tiny6410, změřit dobu přepnutí kontextu a vytvořit aplikaci, která bude ovládat připojený USB osciloskop DSO-2150.

V první části této práce je cílem prozkoumat vlastnosti této sestavy, resp. operačního systému. Jedním z významných parametrů systému je čas přepnutí kontextu, což je jedna z nejčastějších operací systému. Přítomnost této funkce systému zajišťuje pseudoparalelní běh procesů na jednojádrovém procesoru, a tedy zdánlivě paralelní souběh spuštěných aplikací a uživatelského ovládání. Tato část práce se tedy bude hlavně zabývat měřením tohoto parametru.

Dalším parametrem je čas reakce na externí přerušení, tedy čas, za jaký systém vykoná obslužnou rutinu a předá CPU zpět plánovači. Tento parametr je závislý na

architektuře systému přerušení, taktovací frekvenci a softwarové architektuře systému přerušení. U tohoto parametru budou uvedeny způsoby jeho měření.

Posledním měřeným parametrem je čas operace s externí pamětí, tedy čas průchodu určitého paměťového bloku a vykonání standardní operace. Tento parametr zahrnuje režie přístupu a operace s externí pamětí a je závislý na optimalizaci operací s externí pamětí a na frekvenci paměťové sběrnice.

Jednotlivými měřeními tedy získáme přehled o vlastnostech systému a možnostech jeho použití. Stejně tak získáme přehled o možnostech kitu Tiny6410 a procesoru S3C6410.

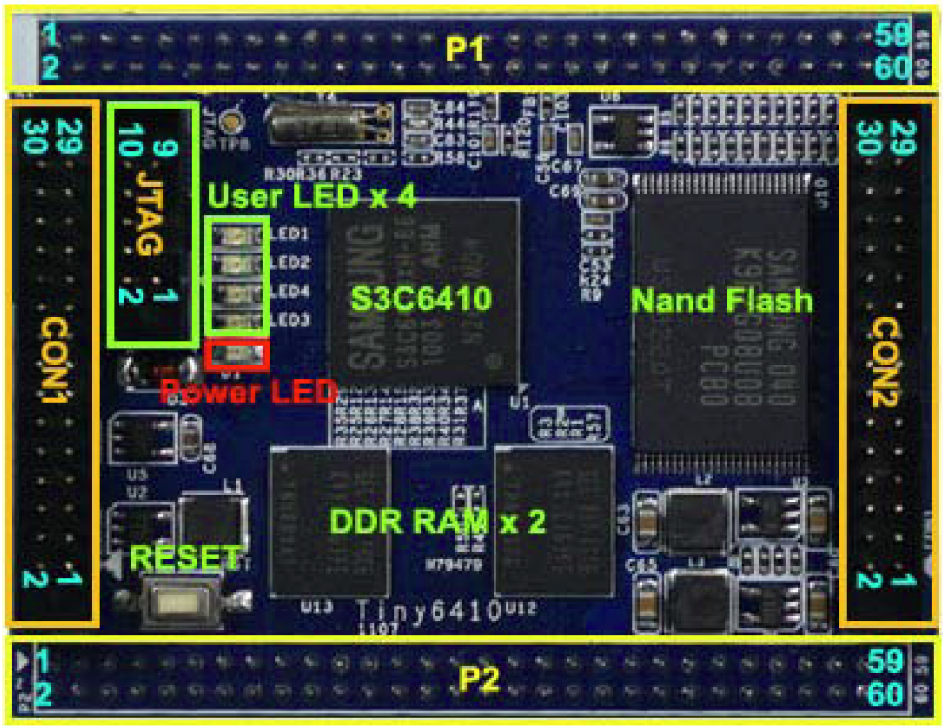
Druhá část práce se zabývá vytvořením uživatelské aplikace, která bude sloužit jako rozhraní pro USB osciloskop DSO-2150. Aplikace bude zobrazovat naměřené průběhy z osciloskopu, se kterým bude komunikovat pomocí USB sběrnice, a nastavovat parametry osciloskopu dle pokynů od uživatele. Tato aplikace bude demonstrovat multimediální a komunikační možnosti operačního systému Android.

2 VÝVOJOVÝ KIT TINY6410

Tiny6410 v1.2 je jeden z vývojových kitů rodiny FriendlyARM. Tento kit je určen pro vývojové účely, pro seznámení vývojářů s možnostmi čipu S3C6410, který je postaven na jádře architektury ARM11. Je tvořen deskou plošného spoje (SDK board), která obsahuje standardní hardware pro různá komunikační rozhraní (USB, UART, ethernet, aj.), testovací tlačítka a senzory, sloty pro paměťové karty a připojenou desku s vlastním procesorem s pamětí, tzv. CPU board.

2.1 CPU board

Srdcem kitu je CPU board, což je samostatný plošný spoj obsahující obvod S3C6410, dynamickou a statickou paměť, stabilizátor napětí a rozhraní pro připojení periferií. Výčet součástí CPU boardu shrnuje následující tabulka.



Obr. 2-1 CPU board Tiny6410 [1]

Procesor	Samsung S3C6410, 533 MHz
RAM	2x 128 MB DDR, typ K9F2G08U0A
Flash	2 GB NAND Flash MLCV, 32bit bus
Regulátor napětí	Vstupní napětí je deklarováno 2,0 až 6,0 V
Reset	Resetuje procesor
LED diody	4 zelené pro uživatelské využití, 1 červená signalizující napájení
Konektor P1	Signály pro LCD, A/D, SDIO2, USB, TVOUT0, přerušení, aj.

Konektor P2	Signály pro sériovou linku, SPI1, I2C, SD card, aj.
Konektor CON1	GPIO, A/D, SPI0, TVOUT1
Konektor CON2	CMOS kamera, GPIO
JTAG	Konektor pro programování a debugování technologií J-Link

Tab. 2-1 Součásti CPU boardu [1]

2.2 S3C6410

Výkonnou součástkou CPU boardu je SoC (System on Chip) firmy Samsung s typovým označením S3C6410. Jedná se o elektronickou součástku, která v jednom integrovaném obvodu sdružuje 16/32-bit RISC mikroprocesor založený na architektuře ARM11 v6, paměťový subsystém a periferní subsystémy pro komunikaci a zpracování signálů spojené vnitřní 64/32-bit sběrnicí.

S3C6410 je zaměřen na embedded aplikace s podporu multimédií, mobilních či jiných standardních komunikací, proto je vybaven subsystémy pro komunikaci po 2G a 3G sítích, subsystémy pro zpracování multimédií, podporou pro televizní výstup a řadičem pro TFT LCD displeje.

Sdružením těchto subsystémů do jednoho čipu se součástka stává konkurenceschopná z hlediska ceny a rychlosti vývoje embedded zařízení určených pro tuto aplikaci. To je dáno tím, že komunikace mezi subsystémy a systém napájení je z elektrického hlediska již dána od výrobce a uživatel pouze využije nainstalovanou sadu zapojených subsystému ve své konkrétní aplikaci.

Blokové schéma subsystémů S3C6410 je na obrázku v příloze č. 6.

2.2.1 Vybavení

S3C6410 je vybaven mikroprocesorovým subsystémem postaveným na architektuře ARM. Dalšími důležitými subsystémy jsou multimediální a komunikační. Všechny subsystému spolu komunikují pomocí vnitřní sběrnice typu AHB či APB.

2.2.1.1 Mikroprocesorový subsystém

Mikroprocesorový subsystém je založen na architektuře mikroprocesoru ARM1176JZF-S, tedy Harvardské architektury. Obsahuje oddělené paměti o velikosti 16 kB typu cache pro data a instrukce a oddělené paměti o velikosti 16 kB typu TCM (Tightly-coupled memory) pro data a instrukce. Obě cache paměti mají možnost 64bitového adresování. Cache paměti pracují v neblokujícím módu.

CPU obsahuje Arm Thumb (ARMv6) instrukční sadu, technologii Jazelle, která umožňuje přímé vykonávání Java byte kódu a navíc řadu SIMD (Single Instruction Multiple Data) DSP instrukcí, které vykonávají operace s 16 nebo 8 bitovými daty v 32bitových registrech. Mikroprocesor je taktován frekvencí 533 MHz, přičemž maximální dovolená frekvence taktování je 667 MHz.

Významné rysy mikroprocesoru :

- vnitřní sběrnice dle standardu AMBA (Advanced microcontroller bus architecture), obsahuje sběrnice APB (Advanced Peripheral Bus), AHB (Advanced High-performance Bus) a AXI (Advanced Extensible Interface)
- 8-stupňové zřetězení instrukcí (pipe-lining) – dovoluje zpracovávání více instrukcí zároveň
- předpovídač (branch prediction) – jednotka snažící se odhadnout, který skok se vykoná v instrukcích if-else, aby se mohla načíst další instrukce metodou pipe-lining
- return stack – zásobník pro adresy při předpovídání volání adres a návratů
- cache paměti v neblokujícím módu s Hit Under Miss optimalizací (cache dovoluje číst data z cache v době, kdy získává jiná požadovaná data z vyšších pamětí)
- možnost nastavení „rychlé“ obsluhy přerušení
- koprocesor CP14 pro podporu technologie Jazelle a CP15, který se stará o celkovou správu a konfiguraci systému, cache pamětí, TCM pamětí a MPU (Memory Protection Unit), také zpřístupňuje cache paměti pro debugování a monitoruje výkon systému
- instrukční a datová jednotka správy paměti MMU (Memory Management Unit) - stará o překlad logické adresy na fyzickou, o ochranu přístupu do paměti, přičemž využívá nejdříve jednotlivé Micro TLB (Translation Look-aside Buffer) struktury, poté v případě neúspěchu vužije Main TLB strukturu
- VFP (Vector Floating-Point) - koprocesor pro aritmetické operace s single-precision a double-precision čísly dle standardu ANSI/IEEE 754/1985, je možno pracovat s čísly vektorově (funkce neon), kdy koprocesor používá SIMD instrukce, a tedy dokáže v jedné instrukci zpracovat až 8 operandů typu single-precision

8-stupňový pipe-lining se skládá z těchto kroků [3] :

1. Fe1 – načtení instrukce
2. Fe2 – předpovídání skoků
3. De – dekódování instrukce
4. Iss – výběr instrukce z fronty a čtení registrů
5. Sh – vykonání operací posunu/generování adresy
6. ALU – vykonání operací ALU jednotky/první krok přístupu do cache paměti dat
7. Sat – saturace výsledků typu integer/druhý krok přístupu do cache paměti dat
8. WB – zápis výsledku

Při chybné predikci skoku se zmaří 6 hodinových cyklů.

Mikroprocesor se může nacházet v osmi operačních módech, přičemž jeden je uživatelský s aktivními omezeními. Ostatní módy jsou privilegované s aktivním

omezením pouze na zdroje. Každý mód má svoje specifické registry. Operační módy jsou [3] :

- User – provádí se normální operace
- Fast interrupt – obsluhování „rychlých“ přerušení
- Interrupt – obsluhování všech ostatních přerušení
- Supervisor – chráněný mód operačního systému
- Abort – zákaz přístupu do paměti
- System – privilegovaný mód operačního systému
- Undefined – ve frontě se nachází neplatná instrukce
- Secure monitor – aktivní funkce ochran na čipu

2.2.1.2 Paměťový subsystém

Díky paměťovému subsystému, má mikroprocesor optimalizované rozhraní pro připojení externích pamětí, a to pomocí dvou portů. Jeden port slouží pro připojení dynamické paměti DRAM a druhý slouží pro připojení statické paměti Flash/ROM. Subsystém je koncipován jako matice pamětí s velkou šířkou dat, a tedy propustností, což je vyžadováno pro nasazení v mobilních komunikací.

Subsystém obsahuje dva nezávislé kanály pro paměti s rozhraním :

- SROM
    - SRAM/ROM/NOR Flash – datová sběrnice 8bit nebo 16bit, adresová sběrnice max 27bit
    - OneNAND Flash – datová sběrnice 16bit
    - NAND Flash – podpora SLC i MLC
    - CF
- DRAM
    - SDRAM – 16/32-bitová datová sběrnice
    - Mobile SDRAM – pouze 32bitová datová sběrnice s propustností 133 Mbps/pin (při 133MHz)
    - DDR/Mobile DDR – 16/32-bitová datová sběrnice s propustností 266 Mbs/pin

2.2.1.3 Subsystémy pro akceleraci multimédií

S3C6410 obsahuje hardwarovou podporou dekódování a enkódování formátů MPEG4, H.264/H.263, vykreslování 3D grafiky pomocí ovladačů OpenGL ES 1.1 a 2.0 a hardwarovou podporu operací pro 2D vykreslování, jako jsou převracení a změna rozměrů. Pro video výstup je mikroprocesor vybaven hardwarovou podporou pro televizní výstup TV OUT. Pro získání obrazu je v subsystému obsažena podpora pro vstup z CMOS kamery. Subsystémy zpracovávají multimédia v těchto oblastech :

- kamera – změna rozlišení vstupu až z 4096x4096, vytvoření výstupu v RGB 16/18/24-bit, operace rotace a zrcadlení, digitální zoom, přímý výstup na LCD pomocí MSDMA
- multi format codec – podpora enkódování a dekódování MPEG4, H.264, H.263, VC1, kontrola bit-ratu, synchronizace a porcování dat (RVLC)
- JPEG codec – enkódování a dekódování, dekomprese a komprese
- 2D graphics – zrychlení vykreslování linek, pixelů a textů, podpora rotace obrázků s rozlišením až 2048x2048
- 3D graphics – hardwarová podpora 3D vykreslování pomocí ovladačů OpenGL ES, vykreslování až 75,8 Mp/s na frekvenci 133 Mhz, akcelerátor napojen přímo na sběrnice AHB (32bit) a na 2 kanály sběrnice AXI (64bit)
- TV video encoder – zpracování video výstupu formátů NTSC a PAL s rozšířenými obrazovými funkcemi, jako je změna rozlišení či kontrastu

2.2.1.4 Řadič pro TFT LCD

Řadič se stará o video výstup na LCD displeje. Podporovaná rozlišení jsou 800x480, 640x480 a 320x240. Podporuje také změnu jasu. Řadič se také stará o získání pozice kliknutí na připojeném dotykovém displeji.

2.2.1.5 Audio subsystémy

S3C6410 je vybaven rozhraním pro připojení audio vstupu a výstupu. Zpracování signálů je umožněno standardním řadičem AC97, jenž podporuje jeden výstup, jeden vstup a jeden mikrofonní vstup s maximální frekvencí 48 kHz.

Dalšími podporovanými audio rozhraními jsou PCM serial a IIS-Bus, které mohou pracovat bez zásahu mikroprocesoru pomocí technologie DMA (Direct Memory Access).

2.2.1.6 Komunikační subsystémy

S3C6410 disponuje hardwarovou podporou standardních průmyslových sběrnic I2C, SPI a UART. Také podporuje populární uživatelskou sběrnici USB v režimu Host a také duálním režimu HS OTG, a to ve všech třech rychlostních kategorií.

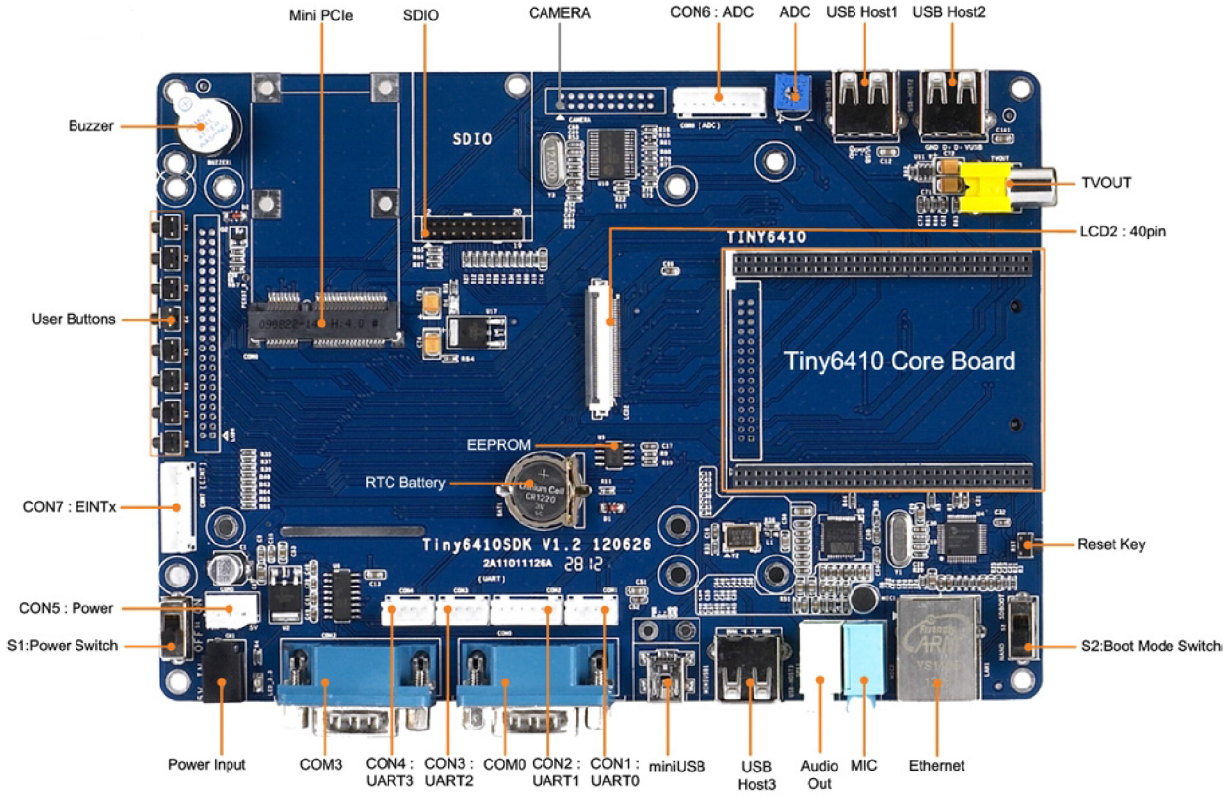
Internetové komunikační prostředky, jako je ethernet a WiFi, jsou podporovány pouze z pozice hardwarové podpory zprostředkovatelských komunikačních sběrnic (SDIO, HIS). Po těchto sběrnicích komunikuje procesor se specializovaným čipem, který implementuje komunikační stack až na úroveň linkové vrstvy. Poté je komunikační dráha ještě upravena PHY čipem, který implementuje fyzickou vrstvu standardu IEEE802.3. V případě WiFi komunikace je implementace provedena pomocí externího SDIO WiFi modulu.

2.2.1.7 Security subsystém

Tento subsystém umožňuje hardwarovou akceleraci výpočtu šifer (AES, 3DES), výpočtu hash funkcí (SHA-1) a generování náhodných čísel. Architektura tohoto subsystému umožňuje zpracovávat výpočty dávkově za pomocí vyrovnávacích bufferů a za nečinnosti CPU díky podpoře DMA. Data se přenášejí pomocí mechanismu zabezpečeného DMA, tedy SDMA (Security Direct Memory Access).

2.3 SDK board

Největší součástí kitu je SDK board. Je to plošný spoj, který obsahuje patice pro připojení CPU boardu a rozšiřuje jeho vybavení ve smyslu periférií. To znamená, že přes patici jsou na piny procesoru zapojeny periferní konektory přímo (UART, LCD) či přes standardní převodníky (PHY pro ethernet, AC97 pro audio výstup). Seznam instalovaných periférií je v následující tabulce.



Obr. 2-2 SDK board Tiny6410 [13]

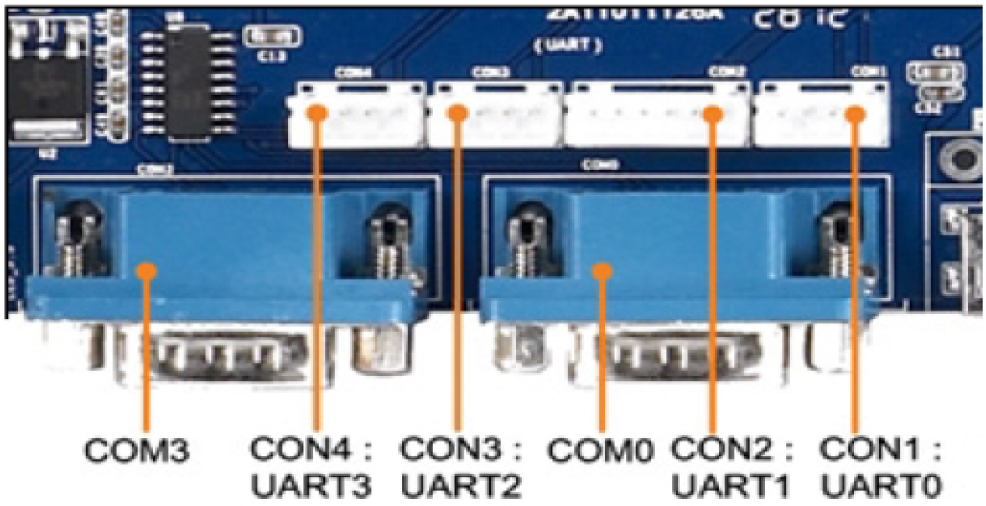
8 uživatelských tlačítek	Tlačítka jsou zapojena na piny SoC, mohou pracovat v interrupt módu
Mini USB	USB Slave
USB Host	USB Host z SoC je přes USB HUB rozdělen na 4 USB, z nichž pro 3 jsou osazeny konektory

Ethernet	10/100M MB ethernet s konektorem RJ45
Audio I/O	Standardní audio vstup a výstup na 3,5mm Jack
SD card	Slot pro paměťovou SD kartu
Serial	4 konektory pro sériovou linku na TTL úrovních, z toho 2 jsou vyvedeny na DB9 konektor
TV-OUT	Video výstup RCA
SDIO rozhraní	Patice pro SD WiFi modul, který také obsahuje SPI a I2C rozhraní
LCD	Různé patice pro připojení LCD touchpanelů, vestavěn touchpanel H43 4,3“
Buzzer	PWM výstup na piezokrystal
IR	Infračervený přijímač
Teplotní sensor	Teplotní čidlo Road DS18B02
ADC	Potenciometr, jehož jezdec je připojen na analogový vstup SoC, pro AD konverzi
RTC baterie	Baterie pro hodiny reálného času v procesoru
Regulátor napětí	Regulátor napětí pro napájení desky dodávající napětí 5 V

Tab. 2-2 Součásti SDK boardu [1]

2.3.1 Sériové komunikační sběrnice

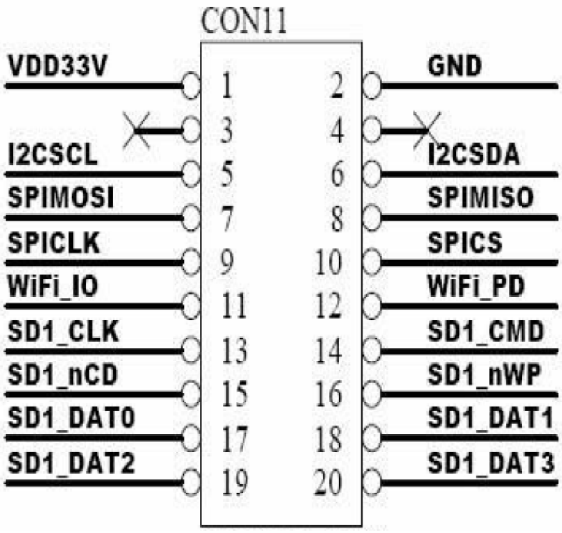
SDK board obsahuje čtyři konektory pro sériovou linku v TTL úrovních. Kit dále obsahuje dva konektory DB9 jako standardní rozhraní pro sériovou linku. Tyto konektory jsou rozšířením stávajících portů sériové linky pomocí převodníků úrovní MAX3232.



Obr. 2-3 Sériové porty SDK boardu [13]

2.3.2 SDIO rozhraní

SDIO rozhraní umožňuje připojení dalších modulů, např. SD WiFi modulu, který také obsahuje porty pro sběrnice I2C a SPI. Rozhraní je realizováno konektorem CON11.



Obr. 2-4 Popis konektoru CON11 pro SDIO rozhraní [13]

2.3.3 Uživatelská tlačítka

Uživatelská tlačítka slouží jako univerzální diskrétní vstupy. Tyto vstupy jsou také vyvedeny na konektor CON7. Napojení tlačítek na piny procesoru a piny konektoru CON7 shrnuje následující tabulka :

Tlačítko	K1	K2	K3	K4	K5	K6	K7	K8
CON7	1	2	3	4	5	6	7	8
pin	GPN0	GPN1	GPN2	GPN3	GPN4	GPN5	GPL11	GPL12

Tab. 2-3 Napojení uživatelských tlačítek na CON7 a procesor [13]

2.3.4 Analogové vstupy

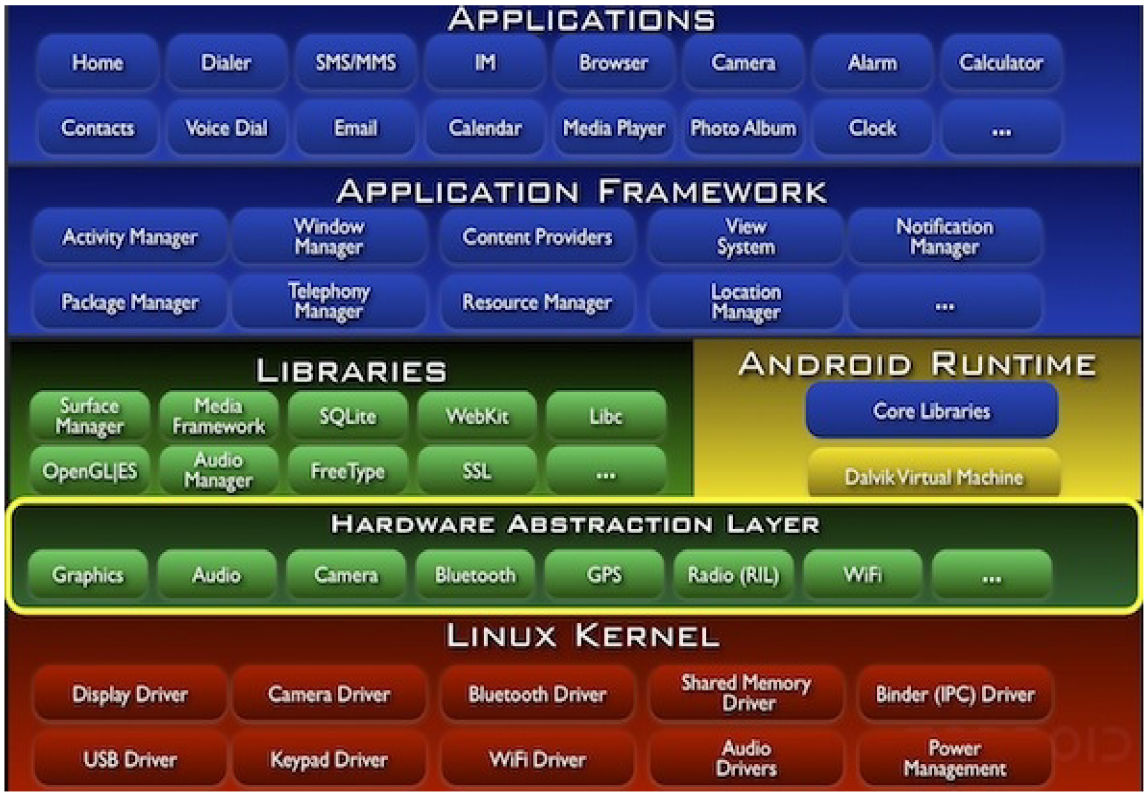
Procesor obsahuje 8 A/D vstupních kanálů, ale pouze jeden A/D převodník. Z toho AIN4 – AIN7 jsou použity na získání informace místa stisku na dotykové obrazovce. Vstup AIN1 je přímo napojen potenciometr W1 na SDK boardu. Ostatní A/D vstupy jsou vyvedeny na konektor CON6.

3 ANDROID

Android je open-source softwarový systém vyvinutý zejména pro mobilní embedded zařízení, jako jsou mobilní telefony, PDA či tablety. Za tímto systémem stojí seskupení firem Open Handset Alliance. Je určen pro použití na procesorech postavených na jádrech ARM, Power Architecture či x86. Snaha vývojářů systému je vytvořit univerzální prostřední pro vývojáře mobilních aplikací, které nezávisí na použité platformě a zpřístupňuje hardwarové součásti daného zařízení pomocí jednotného rozhraní.

3.1 Architektura

Architektura systému Android je rozdělena do vrstev :



Obr. 3-1 Architektura systému Android [4]

Linuxové jádro (Linux kernel)

Systém běží na linuxové jádře. Toto jádro je monolytického typu a obsahuje následující součásti :

- plánovač procesů
- ovladače pro hardware daného zařízení
- nástroje pro správu sdílené paměti
- základní nástroje pro zajištění meziprocesové komunikace IPC (Interproccess Communication) pocházející z Be0S (OpenBinder)

- správa napájení

Každá verze Androidu může obsahovat linuxové jádro jiné verze. To je shrnuto v následující tabulce :

Verze Androidu	Veze linuxového jádra
1.0	2.6.25
1.5 (Cupcake)	2.6.27
1.6 (Donut)	2.6.29
2.2 (Froyo)	2.6.32
2.3 (Gingerbread)	2.6.35
3.0 (Honeycomb)	2.6.36
4.0.X (Ice Cream Sandwich)	3.0.1
4.1/4.2 (Jelly Bean)	3.0.31

Tab. 3-1 Verze Androidu a linuxového jádra [4]

Základní vrstva (Native Libraries Layer)

Tato vrstva rozšiřuje ovladače jádra o knihovny, ve kterých je nadefinováno, jak se má se zařízením pracovat. Tedy definuje funkce a datové typy proměnných pro práci se zařízeními. Dále vrstva obsahuje exekuční prostředí pro Android aplikace, tzv. Android runtime. Výkonnou část Android runtime je tzv. Dalvik virtual machine, což je sada programů, která tvoří virtuální stroj, na kterém běží vlastní aplikace. Tento systém spouštění uživatelských aplikací ve virtuálním stroji se nazývá *sandboxing* [5]. Sandbox spoléhá na oddělení procesů v linuxovém jádře, přidává další bezpečnostní stupeň oddělení procesů a implementuje mechanismy pro zabezpečení zpřístupnění systémových a uživatelsky citlivých API funkcí systému. Virtuální stroj Dalvik je naprogramován v C a C++ vzhledem k tomu, že běží na linuxovém jádře, ale vytváří speciální runtime prostředí pro uživatelské aplikace, které interpretuje speciální byte kód (Dalvik bytcode) uložený v souborech typu DEX.

Abstraktní vrstva (Application Framework Layer)

Je to vrstva, ve které jsou naprogramovány základní aplikace, které se starají o správu určitého celku. Vlastní uživatelské aplikace komunikují pouze s těmito aplikacemi. Vrstva zpřístupňuje třídy a rozhraní pro ovladače, definuje programové rozhraní pro grafickou část aplikace a implementuje služby, které poskytují správu na daném hardware. Jedná se tedy o vrstvu, která tvoří rozhraní daného zařízení se službami, které poskytuje systém android, a která je viditelná z vrstvy uživatelských aplikací předinstalovaných či externě dodaných.

Aplikační vrstva (Applications)

Jedná se o vrstvu, která je tvořena předinstalovanými aplikace pro systém Android, jako je seznam kontaktů či editor SMS zpráv, a jinými externě doinstalovanými aplikacemi, které uživatel potřebuje.

3.2 Vytváření aplikací

Aplikace v systému Android jsou tvořeny *APK* souborem, což je balíček exekučních souborů a souborů se zdroji. Exekuční soubory obsahují Dalvik byte-code instrukce, které jsou vykonávány virtuálním strojem. Vývojáři aplikací mají k dispozici SDK (Standard Development Toolkit) nástroje pro vývoj aplikací standardně v jazyce Java. GUI (Graphic User Interface) část aplikace je řešena pomocí značkovacího jazyka v *XML* formátu. Celá aplikace je z Java prostředí při kompilaci přeložena pomocí SDK do DEX formátu.

Další možností je použít NDK (Native Development Toolkit) sadu nástrojů pro vývoj aplikací v jazyce C++. Tento způsob ale není preferován a je určen pro velmi časově náročné aplikace jako např. herní enginy. Je využíváno technologie JNI (Java Native Interface), která zajišťuje propojení nativního kódu a aplikace běžící v sandboxu na virtuálním stroji. Tato konstrukce ovšem vyžaduje zkompilování nativního kódu s podporou JNI. V Java aplikaci se funkce z nativní knihovny využijí nebo se v nativním prostředí spouští virtuální stroj, kde se spouští části aplikace napsané v Javě.

Každá aplikace musí obsahovat tzv.*maniest file*, který informuje systém o provozních parametrech aplikace a o komponentách, které chce aplikace využívat.

3.2.1 Základní komponenty aplikace

Aplikace pro systém Android je složena ze základních komponent, o které se starají správci komponent tohoto systému. Aplikace může volat komponentu v jiné aplikaci, pokud má oprávnění, ta ale běží v procesu aplikace s volanou komponentou. O přenos dat mezi procesy se stará systém pomocí objektů *Intent*, což jsou zprávy, které mohou obsahovat odkazy na data. Aplikace může obsahovat tyto základní komponenty :

- Activity – jedná se o samostatnou obrazovku s GUI, je možné využívat i jiné Activity z jiných aplikací, pokud je to povoleno.
- Service – je to komponenta běžící na pozadí bez GUI vykonávající náročné operace, Activity může nastartovat proces, připojit se na něj a ovládat ho.
- Content provider – spravuje aplikační data, která ukládá do persistentního uložiště (SQL databáze, souborový systém, web). Umožňuje ostatním aplikacím přístup k těmto datům, pokud mají příslušné oprávnění. Data v Content provideru se z cizí aplikace žádají pomocí *Content resolver*, což přidá mezi aplikaci a data abstraktní vrstvu, která zaručuje vyšší bezpečnost.

- Broadcast receiver – je to v podstatě brána aplikace příjem notifikace událostí (Intent) v systému a posílání událostí do systému. Jedná se nevizuální komponentu, avšak může vytvořit notifikaci ve status baru.

3.2.2 Manifest file

Při spuštění aplikace v systému Android, potřebuje systém znát všechny komponenty, které aplikace obsahuje a parametry s jakými má být aplikace spuštěna. O to se stará *manifest* soubor *AndroidManifest.xml*. Jedná se o soubor s následujícími informacemi ve formátu XML :

- deklaruje minimální požadovaný API level knihoven v zařízení (systémových nebo z třetí strany)
- specifikuje oprávnění, které aplikace potřebuje k běhu (např. čtení kontaktů)
- deklaruje specifické ovladače, které aplikace potřebuje k běhu (např. Kamera)
- jiné parametry (rozlišení, velikost oken, aj.)

3.2.3 Application resources

Android aplikace obsahuje soubory s kódem ale také všechny soubory zdrojů, které využívá, jako jsou obrázky, soubory textových řetězců, grafika obrazovek, aj. Tyto soubory zdrojů jsou členěny do příslušných adresářů a jejich seznam je v souboru *R.java*, přičemž každý soubor je zastoupen identifikátorem *ID*, pomocí kterého se přistupuje ke zdroji v kódových souborech aplikace.

Tento přístup má výhody např. v upravování zdrojů nezávisle na kódové části aplikace jako jsou textové řetězce (podpora jiných jazyků v aplikaci) či obrazovek. Grafická stránka obrazovky (layout) je definována v souboru formátu XML, kde jsou nadeklarovány použité prvky obrazovky (widget), jejich vlastnosti a rozmístění. Namapování událostí (event) je možno řešit dvěma způsoby. A to v kódové části nadefinováním obsluhy události (listener) při vytváření obrazovky nebo zadáním názvu obsluhovací funkce do parametru příslušné události ve widgetu textově.

3.2.4 Native development

Pro účely naší práce se omezíme na první možnost vývoje části aplikace v nativním prostředí. Budeme tedy volat z Java prostředí funkce z knihovny naprogramované v nativním C++ prostředí. Při připojení knihovny v Java prostředí se musí v tomto prostředí nadeklarovat použité nativní funkce [11]. Tyto funkce se systém automaticky snaží navázat na funkce v knihovně pomocí syntaxe jména funkce. Při neúspěchu se po zavolání funkce objeví chyba *java.lang.UsatisfiedLinkError*. Druhou možností je implementovat v knihovně funkce *JNI_OnLoad* a *JNI_OnUnLoad*, které systém automaticky spouští při načtení knihovny. Pomocí funkce *JNI_OnLoad* programátor

naváže nativní funkce k funkcím v Java prostředí, přičemž definuje tzv. signature funkce, tedy její předpis pomocí JNI symbolické syntaxe. Tím se zajistí správné typové kompatibility a zamezí neočekávanému chování aplikace.

3.3 Plánování a správa procesů

Systém Android spoléhá na linuxový plánovač procesů. Standardním linuxovým plánovačem je CFS (Completely Fair Scheduler), což je preemptivní plánovač, který přiřazuje každému procesu stejný časový interval (Time slice) CPU. Avšak procesům s vyšší prioritou (low 19 až high -20) přiřazuje time slice častěji.

Dalvik runtime spoléhá na tento plánovač a mapuje priority dle následující tabulky :

Thread priority	Java name	Android name	Unix priority
1	MIN_PRIORITY	ANDROID_PRIORITY_LOWEST	19
2		ANDROID_PRIORITY_BACKGROUND + 6	16
3		ANDROID_PRIORITY_BACKGROUND + 3	13
4		ANDROID_PRIORITY_BACKGROUND	10
5	NORM_PRIORITY	ANDROID_PRIORITY_NORMAL	0
6		ANDROID_PRIORITY_FOREGROUND	-2
7		ANDROID_PRIORITY_DISPLAY	-4
8		ANDROID_PRIORITY_URGENT_DISPLAY + 3	-5
9		ANDROID_PRIORITY_URGENT_DISPLAY + 2	-6
10	MAX_PRIORITY	ANDROID_PRIORITY_URGENT_DISPLAY	-8
		ANDROID_PRIORITY_AUDIO	-16
		ANDROID_PRIORITY_URGENT_AUDIO	-19
		ANDROID_PRIORITY_HIGHEST	-20

Tab. 3-2 Mapování priorit v Android systému [8]

Při spuštění komponenty aplikace se vytvoří nový linuxový proces s prioritou 0 a všechny komponenty jsou vykonávány v jednom threadu (main thread či UI thread). Pokud už ale proces aplikace běží, jsou spouštěné komponenty spuštěny ve stejném threadu jako běžící proces. Po ukončení běhu komponent není ukončen proces aplikace, ale systém ho udržuje dále v běhu kvůli spuštěným jiným komponentám aplikace (např. Service) nebo kvůli načteným zdrojům (cached resources). Systém ukončuje procesy v případě nedostatku operační paměti metodou priorit. To znamená, že nejdříve ukončí procesy s nejnižší prioritou, přičemž priorita procesu se rovná alespoň nejvyšší prioritě ze spuštěných komponent. Pokud mají dva procesy stejné priority, ukončí se dřív ten, který má tuto prioritu delší dobu.

Jak bylo výše zmíněno všechny komponenty běží v jednom threadu, tedy i grafická stránka aplikace běží v tomto threadu. Je tedy zřejmé, že není dobré zatěžovat tento thread dlouhotrvajícími operacemi. Navíc pokud bude GUI v nečinnosti více jak 5 s, zobrazí se dotaz ANR (application not responding) na ukončení aplikace z důvodu dlouhé odezvy. Vlákna v tomto systému nejsou implicitně thread-safe, to znamená, že při přístupu k proměnným v jiném vlákně může dojít k neočekávanému chování z důvodu špatné synchronizace vláken.

Při vytvoření nového vlákna, běží toto vlákno pseudoparalelně se stejnou prioritou jako vlákno, které ho vytvořilo, a při přístupu do GUI threadu je nutno použít metodu *post* v daném widgetu. Další možnost provedení náročné operace na pozadí je využití třídy *AsyncTask*. Použitím této třídy se kód stává přehlednější a thread-safe.

3.4 Systémový časovač

Časovač je v informačním pojetí většinou hardwarová součástka, která je naprogramovaná k tomu, aby v pravidelných intervalech vyvolávala přerušení CPU. V tomto časovém přerušení se obnovují hodnoty čítačů časovačů systému. Operační systém má většinou k dispozici více typů časovačů. Jednotlivé časovače v systému se liší parametry – rozlišení, granularita – a tím i použitím.

Časovač typu RTC (Real Time Clock) implementuje reálný čas ve smyslu sekund, minut, hodin, dnů, měsíců a roků. Rozlišení časovače v prostředí Android je 1 ms a časovač se volá pomocí funkce *currentTimeMillis()*. Tento časovač počítá od definovaného počátku 1.ledna 1970.

Časovače typu PIT (Programmable Interval Counter) poskytují reálný čas s rychlejší dobou obnovování. Tyto systémové časovače mají velké rozlišení až v řádech nanosekund. U těchto časovačů je důležitým parametrem granularita, tedy nejmenší možná diference dvou po sobě jdoucích hodnot časovače při opakovaném volání funkce pro získání jeho hodnoty [10]. V systému Android je příkaz pro volání hodnoty takového časovače *System.nanoTime()*. Systém vrátí hodnotu nejpřesnějšího časovače, který má k dispozici s rozlišením v nanosekundách. Tento příkaz v Androidu v podstatě volá unixovou funkci *clock_gettime()*, která má ale méně nároků na svoje vykonání. Pokud je potřeba ještě více zefektivnit funkci pro získání hodnoty časovače, je potřeba přímý přístup k časovači [10].

4 MĚŘENÍ PARAMETRŮ SYSTÉMU

V této kapitole bude rozebráno měření parametrů systému Android, a to čas pro přepnutí kontextu, čas reakce na přerušení a čas práce s dynamickou a statickou pamětí. K tomu je k dispozici kit s nainstalovaným systémem Android dodaný výrobcem kitu verze 2.3.4 s linuxovým jádrem 2.6.36-FriendlyARM. Měřicí software je vytvořen v programu Eclipse Indigo 22.2.1 s pluginemSDK tools 22.2.1 pro vývoj aplikací pro systémy Android.

4.1 Switching context time

Prvním měřeným parametrem je čas přepnutí kontextu (switching context time).

4.1.1 Definice

Přepnutí kontextu (context switch) je operace CPU přerušení vykonávání jednoho vlákna či procesu a následné obnovení vykonávání jiného vlákna či procesu od místa, kde bylo naposledy pozastaveno. Tato operace se skládá z úkonů přerušení aktuálně vykonávaného vlákna, uložení stavu procesoru (context) do paměti, načtení stavu procesoru pro jiné vlákno, obnovení registrů procesoru dle tohoto stavu a nastavení programového čítače na instrukci, která se má vykonat [14].

Tento proces zahrnuje ještě vymazání instrukční pipe-line procesoru a obnovení TLB tabulky. Tyto dílčí režie přepnutí kontextu tvoří tzv. přímé režie. Přepnutí kontextu dále zahrnuje režie pro sdílení cache paměti mezi vlákny, tzv. nepřímé režie. Těmito režiemi se dále nebudeme zabývat, jelikož jsou zanedbatelné v případě, že se ve vláknech nepřistupuje k rozsáhlým datovým strukturám [15].

4.1.2 Metody měření

V této kapitole bude rozebráno měření času přepnutí kontextu mezi dvěma vlákny v rámci jednoho procesu na jednojádrovém procesoru. Měření bude provedeno ve standardním Dalvik prostředí a v native prostředí, jak s využitím API volání pro získání přesného času, tak měřením vytvořeného pulsu na výstupu procesoru pomocí osciloskopu.

Při této metodice měření je nezbytně nutné, aby se vykonaly thready hned po sobě, tedy aby scheduler zařadil tato vlákna za sebe. To je možné obecně docílit nastavením nejvyšší priority těmto vláknům. V Dalvik prostředí to můžeme uskutečnit pomocí metody *setPriority()* v třídě *Thread*. V native prostředí je možné nastavit prioritu vláken a zároveň strategii (policy) scheduleru pomocí API funkce standardu POSIX *pthread_setschedparam()*. Nastavitelné strategie scheduleru jsou [12] :

- SCHED_OTHER – standardní nastavení pro normální uživatelské procesy, scheduler přepíná vlákna po časových úsecích, četnost chodu vlákna závisí na prioritě
- SCHED_FIFO – vlákno nebude přepnuto preempcí, jeho běh se ukončí skončením, čekáním na synchronizační objekt či manuálním pokynu k reschedule
- SCHED_RR – scheduler přepíná pouze mezi těmito real-time vlákny s ohledem na jejich nastavenou prioritu

Pro naše účely se zdá být výhodné nastavit strategii scheduleru na SCHED_FIFO, abychom byl zajištěn běh vláken po sobě. Bohužel nastavení strategie scheduleru a prioritu procesu v native prostředí vyžaduje root oprávnění, které jako běžná aplikace nemáme. Navíc systém Android standardně nepodporuje získání root oprávnění. Pro získání root oprávnění v aplikaci v systému Android je možné použít upravenou verzi systému (root verze), nebo real-time patch Preempt_RT, který již umožňuje nastavit prioritu vláken a nastavuje scheduler do real-time módu.

4.1.3 Nejistoty měření

Významnou soustavnou chybou měření je, že scheduler přepíná thready po time slice, přičemž prvotně vyřizuje požadavky na přerušení IRQ (Interrupt ReQuest). Tento vliv je částečně minimalizován na jednojádrovém procesoru pouze měřením v relativně klidovém stavu s co nejmenším počtem běžících procesů, tedy navrhnutím měřicí rutiny s co nejmenší časovou náročností a provedením více měření s výběrem nejmenší naměřené hodnoty, u které je předpokládáno, že je touto chybou nejméně zatížena.

Nejistota pramenící z nepřesnosti systémového časovače (jitter) je zanedbána, jelikož není možné ji pomocí aplikace smysluplně vyčíslit. Nejistota pramenící z granularity systémového časovače, která se pohybuje do hodnoty 2 μs [10-kap.Granularity] je změřena (příloha č. 5), přičemž při tomto i všech dalších měření je zanedbána režie volání funkce *clock_gettime* pro získání hodnoty časovače, která se pohybuje kolem hodnoty 1 μs [10-kap.Performance]. Hodnota granularity časovače je použita jako zdroj nejistoty B dle vzorců (1). Tato nejistota bude také základem pro kombinovanou nejistotu dle vzorce (2), jelikož jako výsledky měření budou brány nejmenší naměřené hodnoty bez nejistoty A.

$$u_B = \sqrt{\sum u_{Bz}^2} \quad , \quad u_{Bz} = \frac{\Delta z_{max}}{\chi} \tag{1}$$

$$u = 2.\mathrm{u}_c = 2\sqrt{u_A^2 + u_B^2} \tag{2}$$

V každém z uvedených měření bude udávána nejnižší naměřená hodnota, průměrná hodnota dle vzorce (3) a výběrová směrodatná odchylka dle vzorce (4), která je základem k určení nejistoty A.

$$\bar{\tau} = \frac{1}{N}\sum_{i=1}^{N} \tau_i \tag{3}$$

$$s_\tau = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N} (\tau_i - \bar{\tau})^2} \tag{4}$$

4.1.3.1 Měření granularity časovače

Než budou provedena jednotlivá měřením, bude změřena granularita časovače metodou popsanou v [10-kap.Granularity]. Toto měření je zpracováno v příloze č. 5.

Změřil jsem granularitu časovače v nativním prostředí systému Android τ = 1,228 μs a z toho vyčíslil kombinovanou nejistotu pro měření s tímto časovačem u = 0,710 μs.

4.1.4 Diagram měření

Všechna měření switching context času jsou realizována dle schématu v příloze č. 4.

4.1.5 Měření v Dalvik prostředí

V tomto prostředí jsou vlákna synchronizována pomocí kritické sekce. Také je nastavena měřicím vláknům nejvyšší priorita, aby byla vyšší šance, že se budou vykonávat po sobě. Pro získání přesného času je volána funkce *System.nanoTime()*. Toto měření je zpracováno v příloze č.1.

Výsledek měření je τ = (80,12±0,71)μs. Naměřený čas je teoreticky složen z času přepnutí kontextu mezi vlákny, režií virtuálního stroje, ve kterém měření probíhalo, a režií volání synchronizačních a časových funkcí. Režii virtuálního stroje je možné odstranit měřením v native prostředí, kdy aplikace běží přímo v uživatelském prostoru linuxového jádra. Tímto se zabývá následující měření.

4.1.6 Měření v native prostředí

V tomto prostředí jsou vlákna synchronizována pomocí semaforu. Vlákna mají základní prioritu, kterou nelze změnit. Pro zjištění přesného času je volána API funkce *clock_gettime(CLOCK_MONOTONIC,...)*. Toto měření je zpracováno v příloze č. 2.

Výsledek měření je τ = (34,93±0,71)μs. Naměřený čas je teoreticky složen z času přepnutí kontextu mezi vlákny a režií volání synchronizačních a časových funkcí. Vliv režie volání časových funkcí a granularity časovače je malý, jak je uvedeno výše. Přesto bude v následujícím měření tento vliv odstraněn tím, že bude vytvořen na výstupu pulse, jehož délka by měla reprezentovat čas přepnutí kontextu.

4.1.7 Měření v native prostředí pomocí osciloskopu

Oproti měření v předchozí kapitole je měřen interval přepnutí vláken pomocí osciloskopu, kdy v prvním vlákně je nastaven výstup a v druhém je výstup shozen. Tím

vznikne pulse, jehož délka je změřena osciloskopem. Jako výstup je použita LED dioda v CPU boardu, jelikož je možné ji ovládat pomocí API funkce systému *ioctl* na virtuálním souboru */dev/leds*. Dodaná verze systému pro daný kit obsahuje sice klasické GPIO (General-purpose Input/Output), ale není zde podpora ovladačů ze strany dodaného systému. Toto měření je zpracováno v příloze č. 3.

Výsledek měření je $\tau = 44{,}60\ \mu s$. Naměřený čas je teoreticky složen z času přepnutí kontextu mezi vlákny a režií volání synchronizačních a souborových funkcí. V porovnání s předchozím měřením je zde značný rozdíl. Tento rozdíl může být způsoben užším výběrem naměřených hodnot, tedy 10 oproti 1000. Další rozdíl oproti předchozímu měření je, že toto měření bylo prováděno jednorázově, zatímco předchozí měření bylo provedeno dávkově. Toto měření bylo ze softwarového hlediska realizováno v prostoru jádra (kernel space) systému na rozdíl od předchozích měření.

4.1.8 Statistické měření

Další možnou metodou měření switching context času je změřit čas např. 10 000 cyklů kyvadlového posílání bytu mezi dvěma thready pomocí meziprocesového komunikačního kanálu (pipe), dále změřit čas téhož úkonu při navázání pipe na ten samý thread a vypočítat switching context time dle vzorce uvedeného v [15-kap.2.1]. Touto metodou je možné dosáhnout přesnějšího výsledku, bereme-li v úvahu, že se při měření obou časů projeví nahodilé chyby měření v podobě obsluhy přerušení a běhu jiných vláken s normálním rozložením. Statisticky se tedy všechny nahodilé chyby měření odečtou. Toto měření již není předmětem této práce.

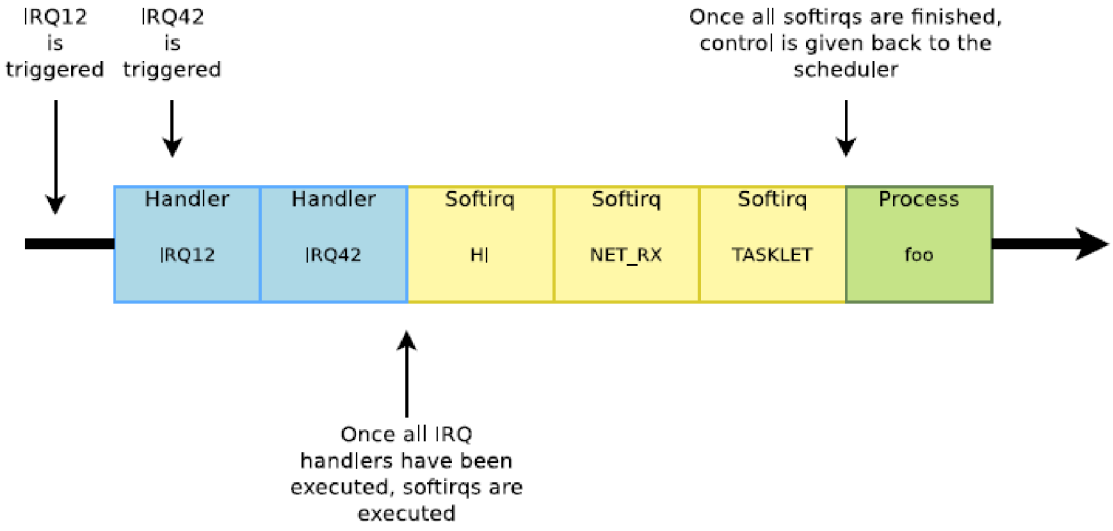
4.1.9 Měření pomocí tracer funkcí jádra

Linuxové jádro, na kterém systém běží, lze za pomoci jeho dostupných zdrojových kódů a speciálního toolchainu překompilovat s vlastním nastavením. V tomto nastavení je možné přidat do jádra specifickou funkci *tracer*. Pro měření switching context time se jako nejvýhodnější jeví funkce *Trace process context switches and events*, která by měla zaznamenat volání definovaných funkcí s časovou značkou volání (timestamp) včetně funkcí scheduleru. Zpracováním výstupu této funkce by potom bylo možné přesněji určit switching context time. Problém by mohl nastat v případě, že scheduler začne vykonávat proces (funkci), u které není nadefinován atribut (prvních pět bytů funkce) pro tracování či skočí do přerušené funkce.

4.2 Měření reakce na přerušení

Reakce na přerušení (interrupt) je vykonání obslužné rutiny při výskytu hardwarového přerušení IRQ procesoru. CPU testuje příchod hardwarového přerušení po každé instrukci vykonané v user mode. Při výskytu přerušení vykonává CPU obsluhy přerušení (interrupt handle) přednostně, v pořadí dle čísla IRQ. Nejdříve uloží svůj

současný stav, začne vykonávat obslužné rutiny všech příchozích přerušení, poté začne případně vykonávat softwarová přerušení subsystémů jádra (soft_irq), poté vykonává procesní funkce s daty nadeklarované během obsluhy přerušení (tasklet), nakonec načte uložený stav a předá vykonávání instrukcí zpět scheduleru. Proces vykonávání přerušení shrnuje následující obrázek :



Obr. 4-2 Stack obsluhy interruptů v linux systému [19]

IDT (Interrupt Descriptor Table) je implementována jádrem systému Android, tedy linuxovým jádrem. Vlastní obslužné rutiny jsou implementovány ovladači nacházející v prostoru jádra (kernel space). V systému linux je možné si zaregistrovat obslužné rutiny přerušení, a to pomocí funkce *request_irq(...)*. Odebrání vlastní funkce přerušení zajišťuje funkce *free_irq(...)*. To je avšak možné pouze v kernel space. V uživatelském prostoru (user space) tedy není možné zaregistrovat obsluhu přerušení [16]. Navíc systém Android nepodporuje zaregistrování obslužných rutin přerušení v kernel space funkcemi v user space. Možná řešení tohoto problému jsou :

- rekompilace jádra – implementace knihoven, které zpřístupní požadované funkce a rekompilace jádra nebo použití jiného již upraveného komunitního jádra
- vytvoření driverů – vytvoření driverů s vlastní obsluhou přerušení volného vstupu [17], který za běhu originálního systému implementujeme do jádra příkazem *insmod driver.ko* z příkazové řádky (shell) systému s root oprávněním. Toto oprávnění máme automaticky při připojení debuggovacího nástroje *adb* k zařízení přes USB rozhraní.
- namapováním fyzického adresového prostoru do virtuální paměti procesu pomocí funkce *mmap(...)* získáme přímý přístup k paměti, který můžeme číst, avšak zapisovat do něj pomocí funkce *msync(...)* můžeme pouze s root oprávněním [18], které naše aplikace nemůže za běžných podmínek získat. Poté

bychom teoreticky mohli změnit adresu skoků v IDT na naši vlastní obslužnou rutinu v případě, že to běžící jádro dovolí.

- využitím tracer funkce kernelu *Interrupt-off latency* můžeme získat čas, za který CPU obslouží všechna přerušení. Tato funkce ale vrací jen maximální naměřenou hodnotu, tedy pravděpodobně vrátí čas, za který se vykoná obsluha souběhu nejvíce přerušení. Tato funkce je do jádra implementovaná při kompilaci, pokud se nastaví.

Za stávající situace jsou dvě možnosti měření doby reakce systému na přerušení. První možnost je číst v blokujícím režimu virtuální soubor seznamu aktivních přerušení s vyhodnocením změny hodnoty počtu přerušení nebo číst v blokujícím režimu virtuální soubor ovladače vstupů s vyhodnocením nástupné hrany vstupu [20]. V tomto případě je ale vnesena do měření soustavná metodická aditivní chyba způsobená řízením exekuce procesů schedulerem pomocí preempce.

4.3 Měření času operace s pamětí

V této kapitole jsou změřeny časy operací se statickou (BSS a data) a dynamickou (heap) pamětí.

Static je segment paměti, která uchovává proměnné aplikace deklarované jako globální či statické. Tyto proměnné jsou alokovány při startu programu a uvolněny při jeho ukončení [23]. Zpomalení této paměti je určeno hlavně segmentací a stránkováním [22].

Heap je segment paměti, ve kterém si může funkce dynamicky alokovat paměť, např. pomocí API funkce *malloc(...)* a po ukončení jejího používání se dealokuje pomocí API funkce *free*(...) [23]. Velikost této paměti závisí na fyzických možnost paměti a díky virtualizaci i na fyzických možnostech pevných externích uložišť. Tato paměť se vyznačuje nedeterministickým přístupem hlavně díky časté fragmentaci [22].

Změřil jsem (příloha č. 7) časové režie průchodu pole ve statické a dynamické paměti systému Android v klasickém Java i v nativním prostředí. Všechna tato měření jsou zatížena chybou metody, a to preempcí scheduleru a obsluhou příchozích přerušení. Časové režie operací se statickou a dynamickou pamětí jsou v jednotlivých prostředích srovnatelné. V Java prostředí je časová režie značně vyšší. Je to dáno tím, že Java prostředí disponuje rozšířenými funkcemi pro správu paměti, jako je její ochrana a automatické uvolňování (Garbage Collector).

V Java prostředí umisťuje compiler všechny nelokální proměnné do dynamické JVM heap paměti [24]. Klíčovým slovem *static* se označují proměnné, které jsou společné všem instancím třídy. Vývojář tedy nemá možnost zvolit umístění proměnné do statické paměti v Java prostředí. To samé platí pro vytváření programu v native prostředí. Klíčovým slovem *static* programátor sděluje compileru, že proměnná je vidět pouze v rámci daného souboru, popř. společná pro všechny instance třídy. Proměnná je ale umístěna v heap paměti [25]. Pokud by bylo potřeba umístit proměnnou do jiné části

paměti, musel by se upravit LDS script, který linkeru říká, do jaké části paměti které proměnné na dané architektuře namapovat.

V obou prostředích jsem zvlášť změřil časovou režii alokace dynamické paměti, která je v native prostředí dle měření nedeterministická a u menších alokovaných celků nezanedbatelná v porovnání s vlastním průchodem alokované paměti. V Java prostředí roste s velikostí alokované paměti.

5 APLIKACE DSO2150

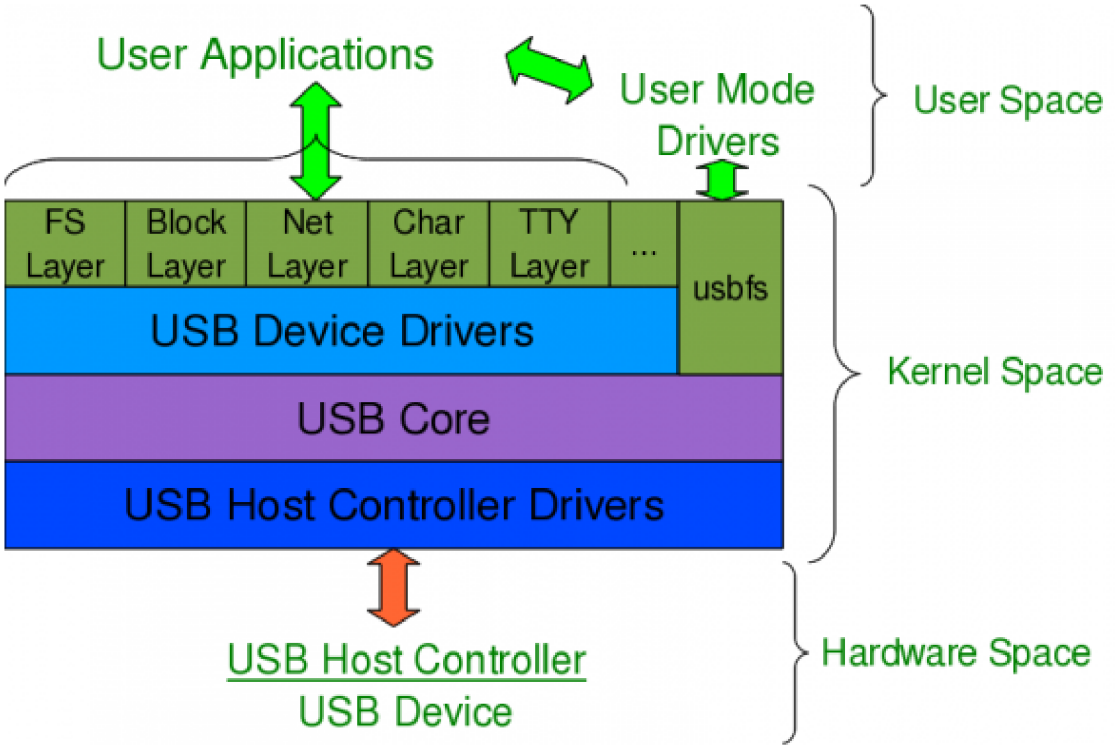
Vytvoření aplikace DSO2150 je druhým hlavním bodem této práce. Tato aplikace bude sloužit jako obrazovka k danému USB osciloskopu typu DSO2150. Aplikace bude zobrazovat signály naměřené osciloskopem a bude umožňovat nastavení osciloskopu, tedy nastavení časové základny, nastavení zobrazovaného velikosti napětí měřeného signálu či nastavení triggeru (úroveň, typ hrany a pozice). Aplikace bude zobrazovat pouze měřené signály z obou kanálů jako funkci napětí na čase $u=f(t)$.

Koncept aplikace jsem navrhl na základě poznatků o systému Android, které jsou uvedeny v předchozích kapitolách a také s ohledem na cílovou platformu. Ze strany operačního systému Android verze 2.3.3 není implementována plná podpora komunikace pomocí USB rozhraní (ta je implementována až od verze 3.0). První možnost je portovat Android vyšší verze na platformu Tiny6410. Zvolil jsem druhou možnost, a to implementovat USB podporu pomocí ovladače (driver).

Koncept celé aplikace se tedy sestává ze tří částí. První část je tvořena ovladačem pro USB komunikaci, který bude spouštěn v prostoru jádra systému Android, tedy v linuxovém jádře. Tento driver bude komunikovat s vytvořenou aplikací pomocí standardního protokolu typu znakového zařízení (character device). Vlastní aplikace se bude sestávat z části vytvořené v native prostředí, kde se budou vykonávat hlavně operace s pamětí. Tato část tvoří můstek mezi ovladačem a GUI částí aplikací. GUI část aplikace bude vytvořena ve standardním java prostředí a bude zajišťovat interakci s uživatelem.

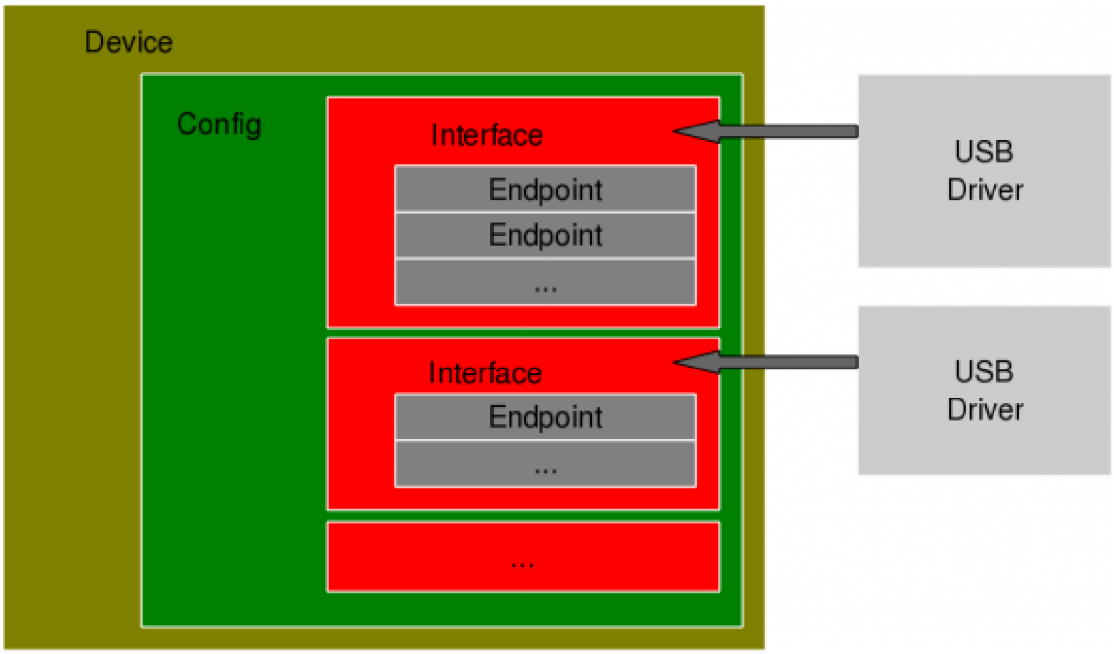
5.1 USB kernel driver

Linuxové jádro již v sobě obsahuje základní ovladače pro práci s USB (USB Core). Jedná se ale jen o ovladače umožňující detekci připojení zařízení a získání parametrů USB připojení. Tato vrstva je již v systému Android implementována. Vlastní ovladače pro komunikaci se specifickým zařízením (USB Device Drivers) se musí pro každé specifické zařízení doprogramovat. To většinou zajišťuje výrobce zařízení ve spolupráci s výrobcem OS. Tyto specifické ovladače komunikují s aplikacemi v user space pomocí standardizovaného rozhraní. Toto rozhraní se volí v závislosti na typu zařízení. TTY rozhraní je tvořeno konzolí, character rozhraní je koncipováno jako textové vstupně výstupní rozhraní. Rozhraní mezi USB ovladači a uživatelskou aplikací tvoří vrchní vrstvu stacku USB ovladačů jádra. Celou situaci shrnuje následující obrázek :



Obr. 5-1 Stack USB ovladačů linuxového jádra [26]

Každé USB zařízení je v systému reprezentováno abstraktní třídou tohoto zařízení (device), která obsahuje dostupné třídy konfigurací (config) podporované zařízením. Konfigurace obsahuje třídy rozhraní (interface), pomocí kterého zařízení komunikuje. Rozhraní jsou definovány sadou přípojných bodů (endpoint), což představuje adresu pro komunikaci typu roura (pipe). Celou situaci shrnuje následující obrázek :



Obr. 5-2 USB description [26]

V závislosti na typu a vlastností přenášených dat se endpointy dělí na :

- control – pro přenos řídících dat, obsažen v každém zařízení s adresou 0
- interrupt – pro rychlý přenos krátkých shluků dat
- bulk – pro přenos velkých shluků dat s nižšími přenosovými frekvencemi
- isochronous – pro přenos většího množství dat se zaručeným rychlým přenosem, ale nezaručenou integritou dat

Vytvořený ovladač má za cíl zpřístupnit základní operace, jako jsou zápis a čtení na jednotlivé typy endpointů. Pro naše účely stačí, když ovladač zpřístupní operace zápis a čtení na endpointy typu bulk a control (viz obr. 5-3). Tyto funkce jsou zpřístupněny pomocí rozhraní, které je charakterizováno funkcemi *read(...)* pro přenos dat z ovladače, *write(...)* pro přenos dat do ovladače a *ioctl(...)* pro vykonávání příkazů.

Pomocí funkcí *write(...)* a read*(...)* jsou zavolány požadovaná USB funkce. Tyto USB funkce a případně jejich parametry jsou zvoleny pomocí funkce *ioctl(...)*. Poté funkce *write(...)* zavolá USB příkaz pro zápis a jako parametr *data* předá zadané pole dat. Funkce *read(...)* naopak zavolá USB příkaz pro čtení a vrátí pomocí parametru odpověď od zařízení.

Všechny USB funkce, které jsou volány pracují v blokujícím režimu z hlediska volající funkce. Volající funkce tedy čeká na uskutečnění komunikace a poté vykonává další příkazy. Funkce *write(...)* a *read(...)* pracují tedy v blokujícím režimu. Tato skutečnost sice popírá výhody paralelismu, ale pouze na úrovni ovladačů v jádře systému. Volající funkce v GUI aplikaci mohou volat tyto funkce ve zvláštním vlákně. Výhodou blokujícího režimu je, že volající funkce má ihned po zavolání odezvu, zda se vyskytla chyba komunikace, a to z jakékoliv příčiny.

Typ endpointu	**Adresa**
Control	0x00
Bulk IN	0x86
Bulk OUT	0x02

Tab. 5-1 USB interface osciloskopu DSO2150

V operačním systému linux se ovladače zavádí při startu systému. Další možností je zavést driver do systému pomocí příkazu *insmod DSO2150.ko* z příkazové řádky. K tomu je potřeba být v systému přihlášen a mít root práva, čehož je možno docílit spuštěním debuggovací konzole *adb* pro systém Android.

Až bude ovladač zaveden v systému je možné úspěšně spustit aplikaci DSO2150. Po připojení USB osciloskopu systém automaticky vyhledá nejvhodnější driver. Dle PID a VID identifikátorů USB zařízení bude zavolána funkce *probe(...)* v ovladači. Když funkce vyhodnotí správné zařízení, pro které je driver vytvořen, oznámí tuto skutečnost systému pomocí návratové hodnoty funkce a systém přidělí připojenému zařízení tento

driver. Po odpojení USB zařízení je zavolána funkce *disconnect(...)* pro ukončení práce se zařízením a pro uvolnění nepotřebných proměnných.

Driver je tedy koncipován jako softwarový most mezi voláním standardních funkcí pro komunikaci pomocí USB rozhraní z jádra systému a aplikací běžící v uživatelském prostoru, která tyto funkce volá. Po zavedení ovladače do systému je tímto vyřešen problém omezených práv uživatele, pod kterým je v systému spuštěna vlastní GUI aplikace, pro přístup k USB rozhraní systému. UML diagram funkcí driveru :

DSO2150 driver

```
struct usb_dso;
static struct usb_class_driver dso_class;
static struct file_operations dso_fops;
static struct usb_driver dso2150_driver;
```

```
// Funkce implementující ovládání driveru pomocí rozhraní character device driver
static void dso_delete(struct kref *kref);
static ssize_t dso_read(struct file *file, char __user *buffer, size_t count, loff_t *ppos);
static ssize_t dso_write(struct file *file, const unsigned char *user_buffer, size_t count, loff_t *ppos);
static int dso_open(struct inode *inode, struct file *file);
static int dso_release(struct inode *inode, struct file *file);
static long dso_ioctl(struct file *file, unsigned int cmd, unsigned long arg);
//
// Funkce zastřešující volání funkcí pro obecnou komunikaci s USB zařízením
int bulkWrite(struct usb_dso* dso2150, unsigned char *data, unsigned int length);
int bulkRead(struct usb_dso* dso2150, unsigned char *data, unsigned int length);
int bulkReadMulti(struct usb_dso* dso2150, unsigned char *data, unsigned int length);
int controlWrite(struct usb_dso* dso2150, unsigned char request, unsigned char requestType,
 unsigned char *data, unsigned int length, int value, int index);
int controlRead(struct usb_dso* dso2150, unsigned char request, unsigned char requestType,
 unsigned char *data, unsigned int length, int value, int index);
//
// Funkce implementující rozhraní standardního USB driveru pro linuxová jádra
static void dso_disconnect(struct usb_interface *interface);
static int dso_probe(struct usb_interface *interface, const struct usb_device_id *id);
//
// Funkce implementující standardní rozhraní obecného driveru pro linuxová jádra
static int __init usb_dso_init(void);
static void __exit usb_dso_exit(void);
```

Obr. 5-3 UML diagram funkcí driveru

5.2 Native část aplikace

Tato část aplikace je napsána v native prostředí, tedy v jazyce C++. Je tvořena sadou funkcí, které tvoří framework pro ovládání osciloskopu pomocí výše uvedeného ovladače. Funkce zastřešují volání funkcí ovladače a vytváření zpráv s příkazy. Funkce tedy definují rozhraní pro efektivní ovládání zařízení a implementují protokol pro USB komunikaci se zařízením. Směrem dolů v softwarovém stacku komunikuje tato část aplikace s vytvořeným USB ovladačem pomocí character device rozhraní, tedy pomocí funkcí *open(...)*, *read(...)*, *write(...)*, *ioctl(...)* a *close(...)*. Směrem nahoru komunikuje s GUI částí aplikace pomocí JNI rozhraní.

5.2.1 JNI

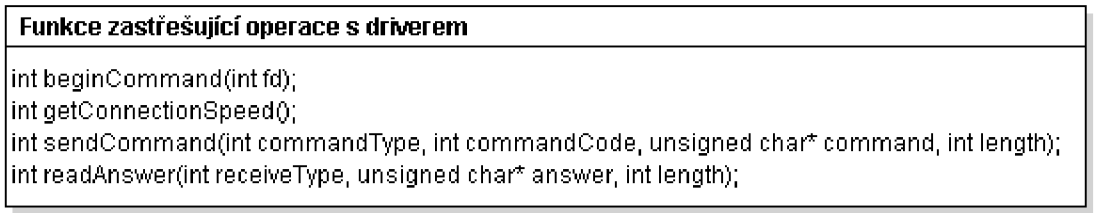
Samotná aplikace, tedy její převážná část včetně GUI rozhraní, je vytvořena ve standardním prostředí pro programování aplikací pro systém Android, tedy v jazyce Java. Aby mohla aplikace využívat funkce z native prostředí, musí být tyto funkce slinkovány do dynamické nebo statické knihovny. Při přilinkování dynamické knihovny v prostředí operačního systému Linux je zapotřebí všechny funkce v cílovém prostředí znovu nadeklarovat a propojit deklarace s funkcemi v knihovně.

Druhou možností je zkompilovat funkce jako statickou knihovnu s příponou SO. Pro volání funkcí z native prostředí existuje rozhraní JNI (Java Native Interface), pomocí něhož si virtuální stroj sám zjišťuje adresy funkcí a spravuje paměť, která je potřeba při volání funkcí. Toto rozhraní je obsaženo přímo v Java prostředí a již integrováno v Java runtime, a tedy i v Dalvik runtime.

Při použití JNI je třeba upravit C++ kód, a to používat JNI datové typy (hlavně při definování parametrů funkcí) a specificky deklarovat funkce (v názvu musí být obsaženo jméno volající activity) nebo použít funkce *JNI_OnLoad(...)* a *JNI_OnUnload(...)* pro spárování funkcí z native do Java prostředí. Po nadeklarování funkcí v Java prostředí lze pak tyto funkce volat, jako by byly v Java prostředí. Správu paměti při volání funkce spravuje runtime.

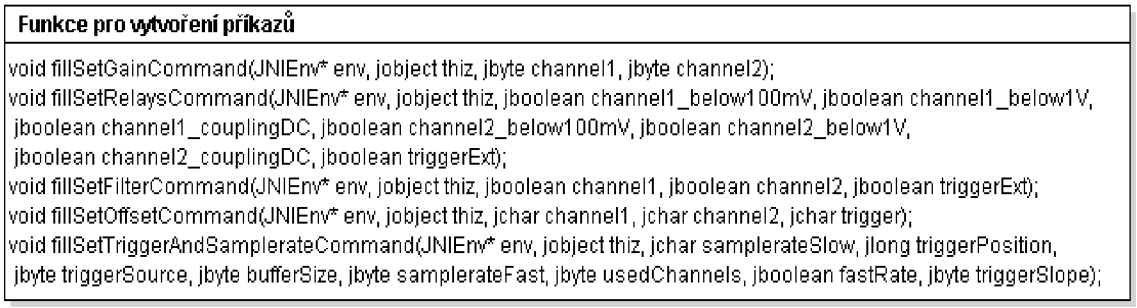
5.2.2 Framework

Funkce implementované v native prostředí lze rozdělit na tři typy. První sada funkcí zastřešuje volání funkcí character device driveru. Tyto funkce lze volat pouze z tohoto prostředí a ošetřují případné vzniklé chyby jak v driveru tak i při samotném volání funkcí *open(...)*, *read(...)*, *write(...)*. Tyto funkce jsou vypsány v následujícím UML diagramu :



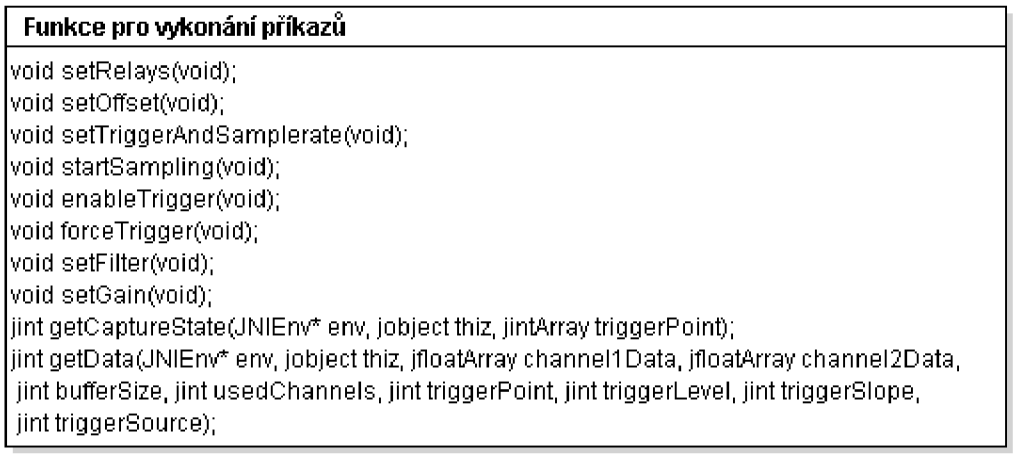
Obr. 5-4 UML diagram funkcí zastřešující operace s driverem

Druhá sada funkcí slouží k naplnění zpráv příkazů pro ovládání osciloskopu. Parametry těchto funkcí definují obsah příkazů, vlastní naplnění úseku paměti příkazu definuje protokol pro komunikaci se zařízením. Funkce mají za cíl urychlit operace s pamětí, jelikož v native prostředí jsou operace s pamětí vykonány rychleji než v Java prostředí (viz kap. 4.3). Tyto funkce jsou vypsány v následujícím UML diagramu :



Obr. 5-5 Funkce pro vytvoření příkazů z parametrů

Třetí sada funkcí implementuje rozhraní pro efektivní ovládání osciloskopu. Parametry funkcí zpřístupňují nastavitelné parametry příkazů. Tyto funkce zastřešují funkce pro ovládání driveru a odchytávají jejich chyby. Dále tyto funkce vypisují přenesené příkazy do logu, pokud je tato funkce povolena. Tyto funkce jsou vypsány v následujícím UML diagramu :



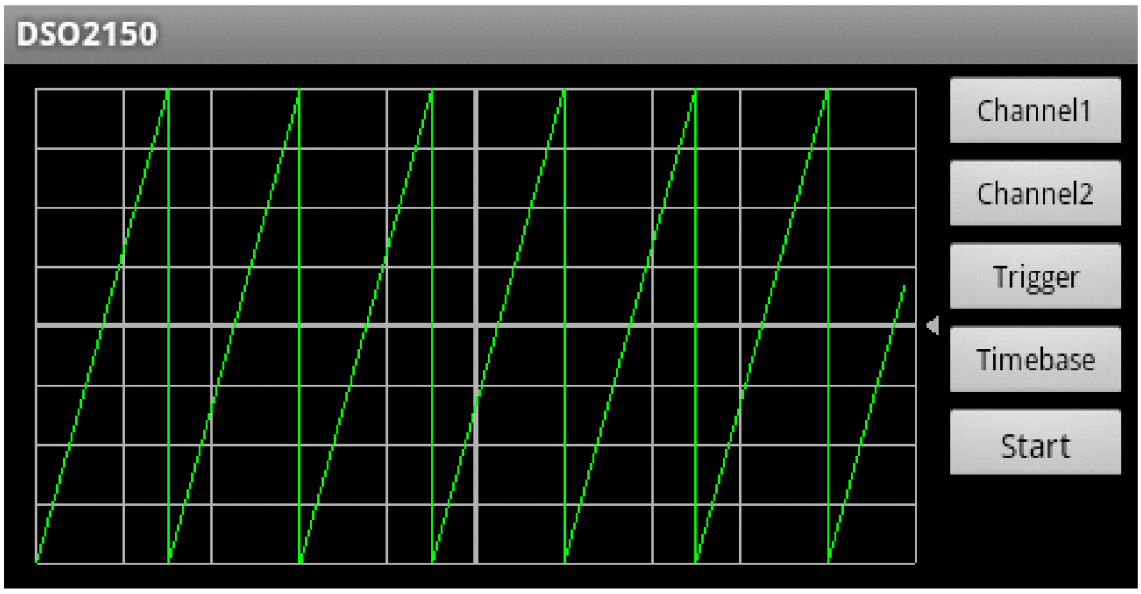
Obr. 5-6 Funkce pro poslání příkazů do zařízení pomocí driveru

Důležitá součást těchto funkcí je implementace synchronizačního mechanismu, který zabraňuje chybám při náhodném volání těchto funkcí z různých vláken (thread) aplikace, a tedy zpřístupňuje USB komunikaci v podobě volání funkcí driveru pouze jedné funkci v daném čase. Tím je zajištěna thread-safe vlastnost tohoto frameworku, resp. ovládání driveru zařízení. Jako synchronizační mechanismus je zde použit semafor, který je deklarován jako globální proměnná, tedy vždy přístupná. Vlastní mechanismus spočívá v zablokování semaforu před voláním funkcí pro komunikaci s driverem a uvolněním semaforu po jejím vykonání či vyskytnuvší se chybě.

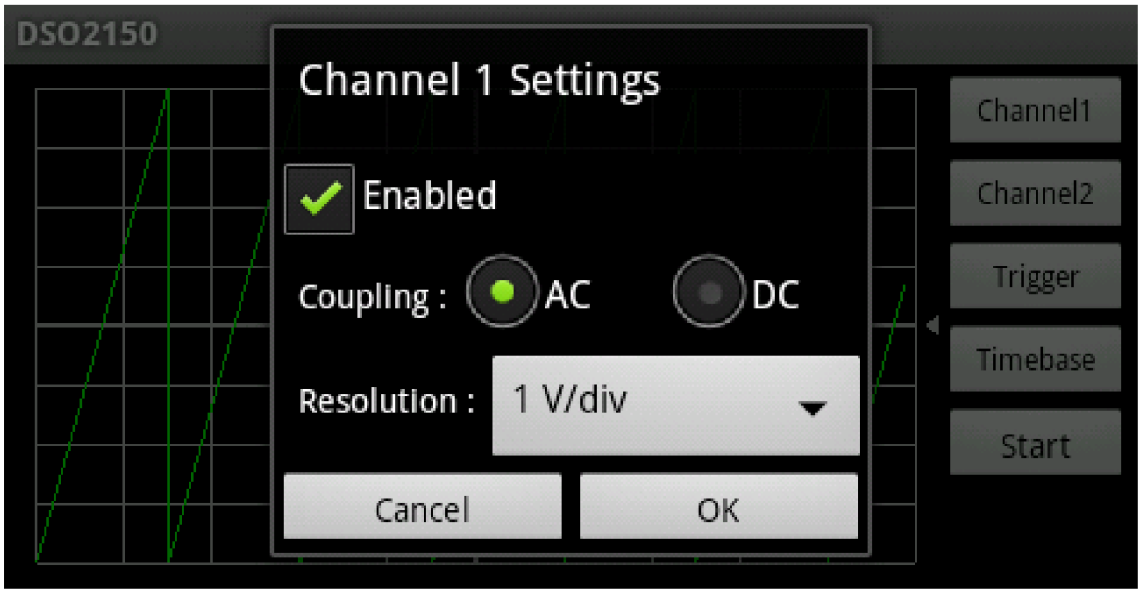
5.3 GUI část aplikace

GUI část aplikace slouží k interakci s uživatelem. Tedy pro zobrazování naměřených dat a pro přijímání příkazů k nastavení osciloskopu pomocí uživatelsky přívětivého rozhraní. Tato část je naprogramována v Java prostředí, tedy ve standardním prostředí pro vytváření aplikací pro systém Android.

Aplikace se skládá z hlavní obrazovky (activity) a dialogových oken, které jsou určeny pro nastavování jednotlivých parametrů. Na hlavní obrazovce je dominantní komponenta (widget) samotná obrazovka osciloskopu, která zobrazuje naměřené signály z osciloskopu dle zvolených parametrů. Pravá strana je vyplněna tlačítky, která zobrazují dialogy s nastavením parametrů. Pro kanál jsou nastavitelné parametry rozsah zobrazení, vstupní vazba a zapnutí kanálu. Pro trigger jsou nastavitelné parametry mód, reakce na hranu a zdroj. Poslední parametr k nastavení je časová základna. Parametry a možnosti nastavení shrnuje tabulka 5-2. Posledním tlačítkem (tlačítko „Start“) se zapne snímání, a to automatické nebo jednorázové.

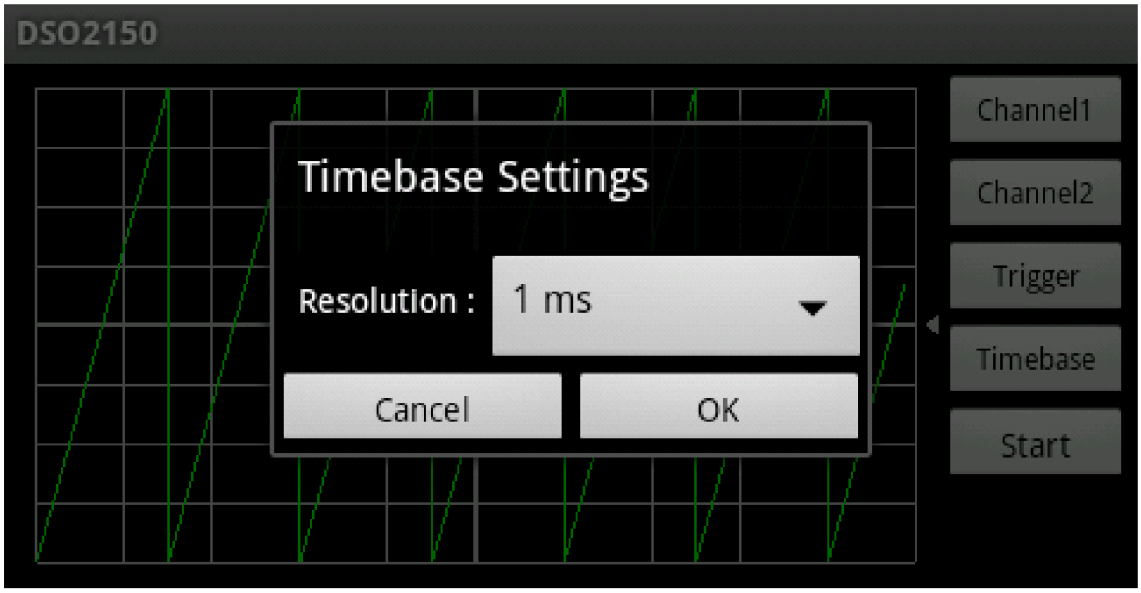


Obr. 5-7 Hlavní obrazovka aplikace DSO2150



Obr. 5-8 Dialog s nastavením parametrů kanálu 1

Obr. 5-9 Dialog s nastavením parametrů triggeru



Obr. 5-10 Dialog s nastavením časové základny

V následující tabulce jsou shrnuty všechny parametry k nastavení a možnosti jejich nastavení :

Celek	Parametr	Možnosti
Kanál	Povolení	ano, ne
	Vazba	AC, DC
	Rozlišení	10mV, 20mV, 50mV, 100mV, 200mV, 500mV, 1V, 2V, 5V
Trigger	Mód	automatický, jednorázový
	Zdroj	kanál 1, kanál 2, externí trigger
	Hrana	vzestupná, sestupná

Obecné	Časová základna	10µs, 20µs, 40µs, 100µs, 200µs, 400µs, 1ms, 2ms, 4ms, 10ms, 20ms, 40ms, 100ms, 200ms, 400ms

Tab. 5-2 Seznam parametrů a možností k nastavení

Osciloskop je v tomto prostředí brán jako objekt třídy *DSO2150*. Tato třída tvoří poslední úroveň abstrakce pro ovládání osciloskopu. Osciloskop je pomocí této třídy ovládán voláním příslušných funkcí. Třída *DSO2150* obsahuje třídy *DSO215OChannel*, *DSO2150Trigger* a *DSO2150Relays*, které vytvářejí objektové rozhraní pro ovládání a nastavování osciloskopu. Tyto podtřídy uchovávají nastavené parametry a při změně automaticky pomocí mateřské třídy vydají pokyn k poslání příslušného příkazu do osciloskopu. UML diagramy těchto tříd jsou v příloze č.8.

5.3.1 Zobrazovač signálů

Vlastní zobrazování signálů je zajištěno pomocí widgetu, který umožňuje přímo zobrazovat grafický výstup ze standardního prostředí OpenGL, resp. OpenGL ES 1.0.

5.3.1.1 OpenGL ES

OpenGL ES je část OpenGL API pro počítačové vykreslování 2D a 3D grafiky pro embedded systémy. Většina čipů, které jsou určeny pro embedded aplikace, jsou vybaveny jednotkou GPU (Graphic Processing Unit) nebo mají hardwarovou podporu OpenGL operací. Díky tomu nemusí vytvářet obraz CPU a vykreslování je rychlejší než při použití softwarového vykreslování GDI (Graphics Device Interace). OpenGL nabízí přístup přímo k hardwarovým vlastnostem, což je vykoupeno většími nároky na práci programátora, než při použití GDI.

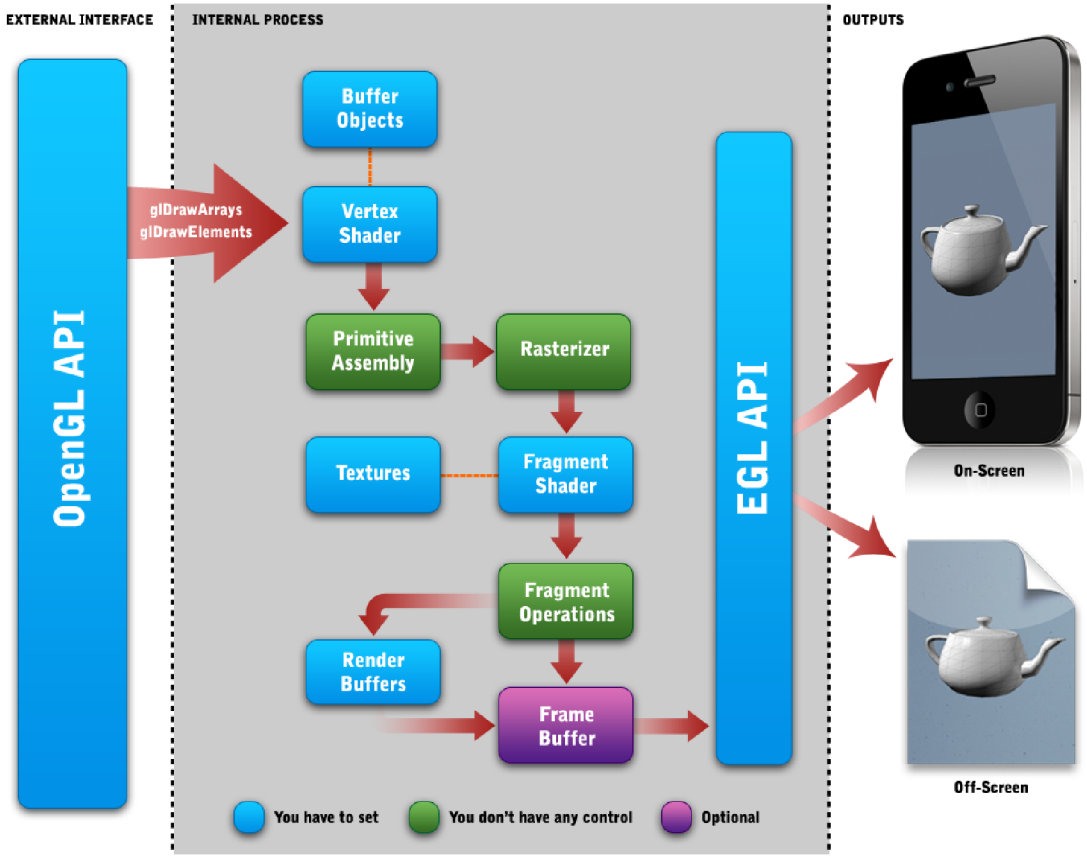
Z hlediska geometrie prostoru je OpenGL koncipován jako 3D prostor s počátkem [0; 0; 0], v němž se nachází vykreslené objekty a pozorovatel (eye). Obrazovka widgetu je tedy vykreslována dle nastavení směru a polohy pozorovatele tak, že se promítne obraz 3D prostoru do 2D prostoru v daném zorném poli pozorovatele (frustrum).

OpenGL je postaveno na těchto konceptech : primitives, buffers a rasterize. Koncept primitives definuje základní geometrické útvary, a to bod, úsečka a trojúhelník. Z těchto základních útvarů jsou složeny všechny vykreslované útvary.

Koncept buffers definuje typy paměťových celků :

- buffer objects – definuje objekt v prostoru a je uchováván na serverové části OpenGL rozhraní
- render buffer – dočasné úložiště obrázku, který vznikne vyrenderováním jednoho objektu s danými parametry
- frame buffer – je úložiště obrázku, který vznikne složením jednotlivých render bufferů a který v podstatě tvoří výstup OpenGL procesu

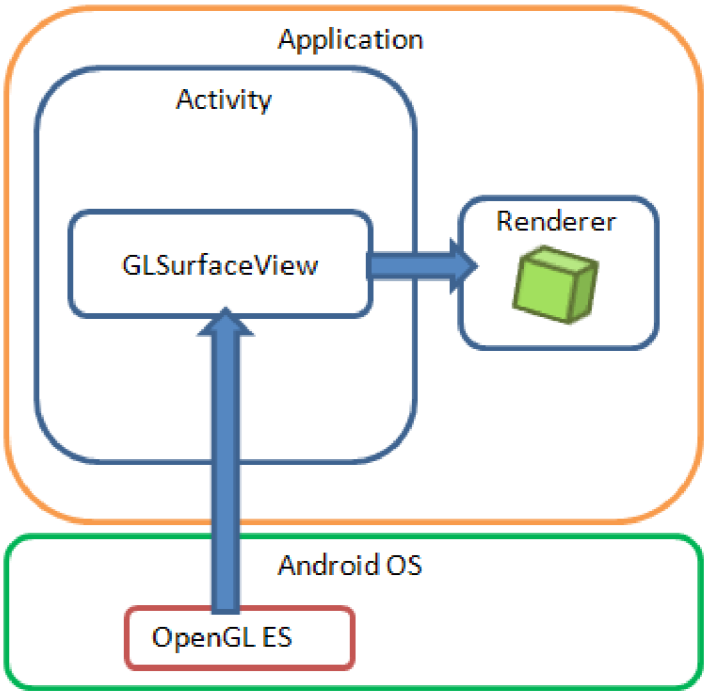
Koncept rasterize definuje proces získání 2D obrázku z nadefinovaných 3D objektů s parametry jako stavový automat. Tento stavový automat je typu pipeline. K dispozici je standardní fixed pipeline se standardním nastavením rozšířených parametrů scény a objektů a programmable pipeline, který využívá tzv.shaders, což jsou mikroprogramy zajišťující rozšířené vykreslování osvětlení, průhlednosti, aj.



Obr. 5-11 OpenGL proces rasterizace [27]

OpenGL je multiplatformní rozhraní, které poskytuje všem programovacím prostředím stejné API rozhraní. Výsledný výstup z OpenGL je zobrazován v systému Android pomocí widgetu *GLSurfaceView*. Tento widget, jako každý jiný zachytává vstup od uživatele. Definice objektů a operace s nimi, jako je rotace a transformace, je zajištěno pomocí vnitřní komponenty *Renderer*. Ta také definuje čtvrtý rozměr – čas, a to tím, že překreslí novou změněnou scénu.

Komponenta *GLSurfaceView* komunikuje s OpenGL ovladačemi implementované v runtime systému Android, které obstarávají vykonání požadovaných operací na hardwaru. Komponenta zapouzdřuje třídu *Renderer*, pomocí které programátor definuje v jakém čase se provede daná operace s objektem či scénou. Tato třída také spravuje četnost překreslování v závislosti na nastavení. A to automatické překreslování s danou frekvencí nebo manuální na externí pokyn programátora. Způsob implementace OpenGL scény do Android aplikace shrnuje obrázek 5-12.



Obr. 5-12 Schéma implementace OpenGL v OS Android [28]

5.3.1.2 Vykreslování v GLScope

Při implementaci komponenty *GLSurfaceView* je třeba odvozením vytvořit cílový widget *GLScope* a naprogramovat požadované funkce. V konstruktoru *GLScope* je třeba inicializovat všechny potřebné třídy, které definují jednotlivé grafické objekty :

- *GLSignal* – definuje naměřený signál
- *GLGrid* – definuje mřížku obrazovky osciloskopu
- *GLDragger* – definuje posuvník offsetu umístěný vpravo od mřížky
- *GLRenderer* – třída pro operace s objekty.

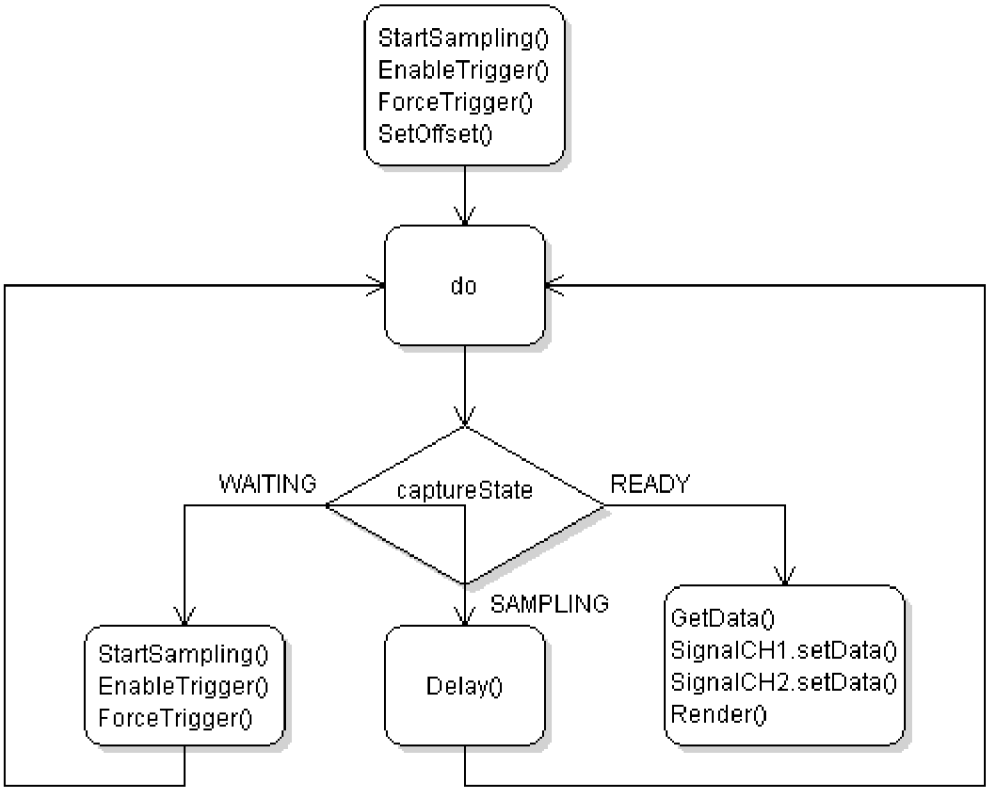
UML diagramy těchto tříd jsou v příloze č.9. V třídě *GLScope* je implementován časovač, který v pravidelných intervalech čte naměřená data z osciloskopu a předává je třídě *GLSignal* k zobrazení. Diagram této funkce je na obrázku 5-13. Perioda volání funkce pomocí časovače je nastavena na 100ms. Tomu odpovídá frekvence překreslování 10 fps. Tuto periodu lze zmenšit, a to i díky sníženým časovým nárokům na vlastní vykreslení obrazu.

Z hlediska USB komunikace je možno při rychlosti Full Speed dosáhnout teoreticky rychlost 12 Mb/s. Při jednom volání funkce se pošle ze strany PC celkem 5 příkazů. Před každým příkazem je nutno poslat kontrolní zprávu zařízení o velikosti 13 B, aby bylo nastaveno na přijímání příkazu po bulk endpointu. Každý příkaz má tedy i s režií protokolu velikost 30 B. Ze strany PC je při vykonání jednoho cyklu funkce třeba 150 B. V případě, že jsou data připravena v zařízení, jsou poslána. V jednom cyklu funkce je vyčítán postupně celý buffer, který má velikost 10240 B. Dohromady se v

jednom cyklu funkce přenese maximálně 10390B pomocí USB rozhraní. Maximální teoretický počet volání této funkce v jedné sekundě je tedy

$$\frac{12\text{Mb}/s}{10390\text{B}*8}=144\,s^{-1} \tag{5}$$

Tomu odpovídá maximální teoretická rychlost překreslování 144f ps a perioda 7 ms. Zmenšováním periody překreslování se také zvyšují nároky na výkon procesoru a operačního systému. Vzhledem k tomu, že je USB kanál využíván i jinými částmi programu v předem nedefinovaných časech a četnosti, je vhodné volit rychlost překreslování co nejmenší. Z uživatelského hlediska bohatě postačí rychlost 25 fps, což je rychlost videa. Pro zachycení nečekaných neperiodických naměřených hodnot je z hlediska uživatele vhodné zvolit nižší hodnotu periody překreslování, aby tyto anomálie byl schopen uživatel alespoň vizuálně zachytit. Na základě těchto poznatků byla zvolena výše uvedená rychlost překreslování 10 fps, tedy perioda volání funkce pro překreslení 100ms.



Obr. 5-13 UML diagram funkce pro vyčítání dat v pravidelných intervalech

5.3.1.3 Ovládání osciloskopu

Pokyn k vzorkování a následný sběr dat je ovládán pomocí výše zmíněné funkce pro vyčítání naměřených dat. Funkce pracuje ve dvou režimech dle nastavení triggeru. V automatickém režimu se v každém cyklu funkce vyčte buffer naměřených hodnot. Ty jsou již naměřeny bez ohledu na přítomnost podmínky triggeru. To je zajištěno příkazem *ForceTrigger()*. Zobrazovaný signál ale nebude bez podmínky triggeru stabilní.

V jednorázovém režimu se po splnění podmínky triggeru zobrazí naměřený signál na obrazovce, a poté je vykonán další cyklus funkce až na pokyn uživatele.

Parametry měření se nastavují pomocí výše zmíněných dialogů. Po potvrzení dialogu je ihned dán pokyn k odeslání nastavovacího příkazu do osciloskopu. Toto nastavování probíhá z hlediska komunikace mezi aplikací a osciloskopem asynchronně. Kolizi se synchronní částí vyčítání dat je zabráněno již v nativní části aplikace pomocí synchronizačního mechanismu typu semafor. V této aplikaci se v podstatě jedná o kritickou sekci, která umožňuje využití zdroje vždy pouze jednomu vláknu, přičemž ostatní vlákna čekají na uvolnění tohoto zdroje.

Parametr offset kanálu a triggeru je ovládán pomocí posuvníku tvaru šipky vpravo od mřížky obrazovky. Táhnutím šipky se nastavuje tento parametr. Změna parametru má okamžitou odezvu.

6 ZÁVĚR

Úkolem diplomové práce bylo seznámit se s kitem Tiny6410, operačním systémem Android, na této sestavě změřit základní parametry operačního systému a vytvořit demonstrační aplikaci pro ovládání osciloskopu DSO-2150.

Při seznamování se s operačním systémem Android nainstalovaném na kitu Tiny6410 jsem zjistil, že tento operační systém je vhodný pro nasazení v uživatelských embedded aplikacích. Procesorový čip tohoto kitu je určen výrobcem spíše pro multimediální aplikace s komunikačními možnostmi, tedy pro komunikační embedded zařízení s grafickým výstupem a různorodým vstupem od uživatele, čemuž odpovídá i jeho vybavení. Síla tohoto systému spočívá také v jeho jednotnosti a portovatelnosti. Tedy na každé hardwarové platformě má vývojář uživatelských aplikací k dispozici stejné rozhraní komunikačních a multimediálních knihoven. Obecně pro potřeby automatizace bych Android použil jako operační systém pro HMI zařízení, kde se nejvíce uplatní jeho přednosti.

První částí práce je zaměřena na měření výkonu systému Android na daném kitu. Přitom jsem se hlavně zaměřil na parametr switching context time, který udává časovou režii systému na dané architektuře základní operace přepnutí mezi vykonáváním dvou vláken či procesů. Tento parametr je závislý na optimalizaci systému, na nastavení vnitřního správce procesů (plánovač) a také frekvenci vnitřní sběrnice CPU jednotky procesoru a její architektuře. Popsal jsem celkem tři metody měření tohoto parametru, přičemž jsem se nejvíce zabýval metodou měření pomocí dvou vláken. Při této metodě je stěžejní zajistit běh vláken hned po sobě, přičemž je měřen čas, který uběhne, než dojde k synchronizaci a započetí vykonávání druhého vlákna. Jelikož je tento parametr měřen na jednojádrovém a jednovláknovém procesoru, je měření zatíženo chybou přičtením času obsluhy příchozích přerušení. Další zdroj metodické aditivní chyby je vlastní přepínání plánovače systému, resp. výběr běžícího vlákna. S danou konfigurací jádra systému není možné skutečně zajistit, že se po synchronizaci z prvního vlákna vybere k běhu právě naše druhé vlákno a nebude přerušeno plánovačem do doby volání funkce pro získání časového rozdílu mezi přepnutím těchto vláken.

Při měření jsem zanedbal časové režie volání API funkcí pro získání hodnoty časovače a časové režie funkcí pro synchronizaci vláken. Změřil jsem pouze zdroj nejistoty vyplývající z granularity časovače.

Z výsledků měření switching context time a při uvážení výše uvedených skutečností jsem určil jako nejpřesnější hodnotu switching context time systému Android na kitu Tiny6410 $\tau = (34{,}93 \pm 0{,}71)\mu s$, tedy nejmenší naměřenou hodnotu přepnutí vláken v tzv. native prostředí systému.

Dalším významným parametrem systému je čas reakce na přerušení. Popsal jsem způsoby měření tohoto parametru a jako nejlepší způsob jsem vybral importování vlastního driveru do jádra za běhu systému, který v obsluze přerušení nastaví signál na

výstupu obvodu, s jehož pomocí osciloskopem je určen čas reakce CPU na příchozí přerušení jako rozdíl mezi signálem pro vyvolání přerušení a signálem nastaveným v obsluze přerušení.

Posledním zjišťovaným parametrem byla časová režie alokace a operace nad polem hodnot alokovaným ve statické a dynamické paměti. Zjistil jsem, že v Java prostředí se globální proměnné mapují automaticky do JVM dynamické paměti a že v native prostředí se také proměnné standardně mapují do heap paměti. Dále jsem zjistil, že s přibývající velikostí pole se rozdíl časové režie alokace paměti v native a Java prostředí zvyšuje.

V této práci jsem zjistil, že v případě vývoje časově kritičtějších aplikací v systému Android je vhodné aplikaci rozdělit na dvě části, a to na časově kritickou část vytvořenou v native prostředí v programovacím jazyce C++ a na uživatelskou část vytvořenou ve standardním prostředí v programovacím Java. Tímto je využito silných vlastností obou vývojových prostředí. Tohoto poznatku je využito při programování aplikace, která je předmětem druhé části této práce.

Úkolem druhé části bylo vytvořit demonstrativní aplikaci, která bude sloužit k ovládání a zobrazování naměřených dat z osciloskopu DSO-2150. Tento osciloskop je připojen k Tiny6410 pomocí USB rozhraní. K tomu jsem musel vytvořit ovladače do jádra systému v programovacím jazyce C, který zpřístupňuje funkce pro komunikaci přes USB rozhraní do native prostředí aplikace. A to jednak z výše uvedeného důvodu výhodnosti rozdělení aplikace, jednak z důvodu absence USB podpory v Java prostředí ze strany OS Android verze 2.3.3, která je naportována na kit Tiny6410.

Vlastní aplikace je tedy rozdělena do dvou částí. Do nativní části jsem implementoval funkce pro ovládání vytvořeného ovladače. Dále funkce, které v paměti vytvoří požadovaný příkaz dle získaného protokolu pro osciloskop. A nakonec funkce pro získání naměřených dat z osciloskopu a jejich zpracování pro zobrazení.

GUI část aplikace, kterou jsem vytvořil ve standardním vývojovém prostředí, obsahuje dialogy s nastavením osciloskopu a zobrazovač signálů. Dialogy tvoří uživatelské rozhraní mezi uživatelem a třídou pro ovládání osciloskopu. Zobrazovač jsem vytvořil s využitím standardních grafických knihoven OpenGL ES. Díky této technologii je možné dosáhnout vyšších frekvencí překreslování, jelikož čip obsažený v kitu má hardwarovou podporu OpenGL operací, a tím se snižují nároky na procesor.

Přínosem této práce je změření času přepnutí kontextu na procesoru postaveným na architektuře ARM11 s frekvencí 533 MHz, změření času alokace a průchodu pole v paměti, ukázka vytvoření uživatelské aplikace pro operační systém Android, ukázka vytvoření ovladače USB zařízení do jádra tohoto systému, ale také ucelení informací o vývojovém kitu Tiny6410 a operačním systému Android.

Seznam použitých zdrojů

[1] Industrial ARMWorks. Tiny6410 Hardware spec [on line]. Dostupné na URL: http://www.andahammer.com/assets/Docs/Tiny6410HWSpec.pdf

[2] Samsung. S3C6410x User's manual. 22 August 2008. Rev.1.1. 1378 p. Dostupné na URL: http://s3c6410kits.googlecode.com/files/S3C6410X_UM_Rev1.10_080822.pdf

[3] Davey I.,Oliveri P. The ARM11 Architecture. Spring 2009. 34p. Dostupné na URL: http://www.cs.virginia.edu/~skadron/cs433_s09_processors/arm11.pdf

[4] Edureka. The Beginner's Guide to Android: Android architecture. 2 January 2013. [cit-3-10-2013]. Dostupné na URL: http://www.edureka.in/blog/beginners-guide-android-architecture/

[5] Varga J.,Kostrecová E. OS Android – Architecture and Security. Sdělovací technika 10/2013.

[6] Android. Application Fundamentals. [cit-21-10-2013]. Dostupné na URL: http://developer.android.com/guide/components/fundamentals.html

[7] Varga J.,Kostrecová E. Android applications – desing and security. Sdělovací technika 9/2013.

[8] Hackborn D. Android Scheduling. [cit-23-10-2013]. Dostupné na URL: https://github.com/keesj/gomo/wiki/AndroidScheduling

[9] Android. Processes and Threads. [cit-23-10-2013]. Dostupné na URL: http://developer.android.com/guide/components/processes-and-threads.html

[10] Gamasutra. Getting High Precision Timing on Android. [cit-30-10-2013]. Dostupné na URL: http://www.gamasutra.com/view/feature/171774/getting_high_precision_timing_on_.php

[11] Wikipedie. Java Native Interface. [cit-31-10-2013]. Dostupné na URL: http://cs.wikipedia.org/wiki/Java_Native_Interface

[12] Bar M. The Linux Scheduler. 1 April 2000. [cit-6-11-2013]. Dostupné na URL: http://www.linuxjournal.com/article/3910?page=0,0

[13] Minidevs. Tiny6410 User Manual v2.0. 22 March 2001. 24str. Dostupné na URL: http://armdevs.googlecode.com/files/Tiny6410_User_manual.pdf

[14] The Linux Information Project. Context Switch Definition. 25 October 2004, updated 28 May 2006. [cit-13-11-2013]. Dostupné na URL: http://www.linfo.org/context_switch.html

[15] Li Ch., Ding Ch., Shen K. Quantifying The Cost of Context Switch. June 2007. 4 p. [cit-20-11-2013]. Dostupné na URL: http://www.cs.rochester.edu/u/cli/research/switch.pdf

[16] O'Reily. Linux Device Drivers, chapter 10 – Interrupt Handling.258 p. 21 Januar 2005. Dostupné z URL: http://lwn.net/images/pdf/LDD3/ch10.pdf

[17] Coffey T., O'Shaughnessy A. Write a Linux Hardware Device. [cit-25-11-2013]. Dostupné na URL: http://www.networkcomputing.com/unixworld/tutorial/010/010.txt.html

[18] Johnson M. User-space device drivers. [cit-25-11-2013]. Dostupné na URL: http://www.tldp.org/LDP/khg/HyperNews/get/devices/fake.html

[19] Adeneo embedded. Linux Kernel and Android Development Class. December 5, 2012. 742p. Dostupné n URL: http://www.aerian.fr/cours/Esiee/I5/OV5-INP/ESIEE%20-%20Linux%20-%20Class.pdf

[20] Brake C. How to implement an interrupt driven GPIO input in Linux. January 10, 2009. [cit-27-11-2013]. Dostupné na URL: http://bec-systems.com/site/281/how-to-implement-an-interrupt-driven-gpio-input-in-linux

[21] FriendlyARM Co., Ltd. User's Guide to Tiny6410 Android. March 28, 2011. 81 p. Dostupné na URL: http://www.geeetech.com/Documents/ARM-AVR%20Series/ARM11/tiny6410/GET%20User's%20Guide%20to%20Tiny6410%20System%20Installation.pdf

[22] Kučera P. Operační systémy reálného času VIII. - Správa paměti. 29 str. Dostupné na URL: http://taceo.eu/lectures/mrts/data/09_sprava_pameti_cz.pdf

[23] Funkhouser T. Memory Allocation. Princeton University, Computer Science 217. Spring 2002. 15 p. Dostupné na URL: http://www.cs.princeton.edu/courses/archive/spring02/cs217/lectures/memory.pdf

[24] Oliveira S. Which one is faster: Java heap or native memory? 29 November 2012. Dostupné na URL: http://mentablog.soliveirajr.com/2012/11/which-one-is-faster-java-heap-or-native-memory/

[25] Silva V. Pro Android Games. ISBN: 9781430226475. Apress. 2009.

[26] Pugalia A.K. Device Drivers, Part 11. 1 October 2011. [cit-12-2-2014]. Dostupné na URL: http://www.linuxforu.com/2011/10/usb-drivers-in-linux-1/

[27] Bomfim D. All about OpenGL ES 2.x-(part 2/3). February 4, 2014. Dostupné na URL: http://blog.db-in.com/all-about-opengl-es-2-x-part-2/

[28] Beatcraft. About OpenGL. March 23, 2012. Dostupné na URL: http://labs.beatcraft.com/en/index.php?Android%20%2F%20Using%20OpenGL%20on%20Android

Seznam použitých zkratek a symbolů

OS	operační systém
SDK	sada nástrojů pro vytváření aplikací ve standardním prostředí
PDA	elektronický osobní asistent
USB	universální komunikační sběrnice
UART	interface pro komunikaci po sériové lince
CPU	jednotka vykonávající instrukce
ARM	architektura procesorů
RISC	architektura mikroprocesorů s redukovanou instrukční sadou
RAM	operační paměť s náhodným přístupem
NAND	struktura tvořící logický člen negativního součinu
MLCV	pokročilá technologie čtení stavu buněk paměti
LED	světelná dioda
LCD	displej z tekutých krystalů
SDIO	vstupně výstupní rozhraní pro paměťová zařízení
TVOUT	rozhraní pro výstup na televizi
I2C	multi-masterová sériová počítačová sběrnice
SPI	sériové periferní rozhraní
GPIO	vstup či výstup pro obecné použití
CMOS	technologie výroby transistorů jako logických prvků
JTAG	technologie pro debugování firmware
AHB	vysokorychlostní vnitřní sběrnice
APB	vnitřní sběrnice pro komunikaci se zařízeními
SIMD	instrukce obsahující více dat
TCM	malá rychlá paměť umístěná co nejblíže místu využití
DSP	procesor pro zpracování digitálních signálů
AMBA	standard pro architekturu vnitřních sběrnic
AXI	vnitřní rozšířená sbrnice
MPU	jednotka ochrany paměti
MMU	jednotka správy paměti
TLB	tabulka často překládaných adres
VFP	jednotka pro zpracování čísel s pohyblivou desetinnou čárkou
ALU	aritmeticko-logická jednotka

DMA	metoda přímého přístupu do paměti
HS OTG	vysokorychlostní USB protokol s pevně nedefinovaným masterem
PHY	fyzické rozhraní pro ethernetovou komunikaci
PWM	pulsně-šířková modulace
TTL	logika postavená na transistorech
IPC	meziprocesorová komunikace
DEX	přípona souborů v Dalvik kódu
APK	přípona souborů celé aplikace pro OS Android
NDK	sada nástrojů pro vytváření aplikací v native prostředí
GUI	grafické uživatelské rozhraní
API	rozhraní systému využívané aplikacemi
XML	přípona souborů značkovacího jazyka
JNI	rozhraní mezi nativním a standardním prostředím
CFS	nastavení plánovače pro přiřazování stejného času všem stejně
RTC	hodiny reálného času
PIT	programovatelný časovač
TTY	rozhraní založené na terminálu
PID	USB identifikátor produktu
VID	USB identifikátor zařízení
UML	označení souborů modelovacího jazyka
SO	přípona souborů pro sdílené knihovny
OpenGL ES	grafické prostředí OpenGL pro embedded zařízení
GPU	jednotka pro vykonávání grafických operací
GDI	rozhraní pro ovládání softwarového vykreslování
HMI	zařízení sloužící jako rozhraní mezi člověkem a strojem

Seznam příloh

Příloha č.1 : Měření switching context time v Dalvik prostředí

Popis :

Měření switching context time v Java (Dalvik) prostředí. V pomocném vlákně se sejme počáteční reálný čas pomocí funkce *System.nanoTime()*, po synchronizaci pomocí kritické sekce s hlavním vláknem se sejme koncový reálný čas. Switching context time je určen diferencí koncového a počátečního času. Provede se 1000 měření.

Seznam přístrojů :

- Tiny6410 kit

Software :

```
public class TheTask extends Thread {
    static long startTime = 0;
    Object lock;
     public TheTask(Object lock){
           this.lock = lock;
     }
     public void run(){
           synchronized(lock)
           {
                 startTime = System.nanoTime();
                 lock.notify();
           }
     }

     public static double measure()
     {
           Object lock = new Object();
        long endTime = 0;
        Thread.currentThread().setPriority(MAX_PRIORITY);
         Thread task = null;

           try
           {
                 task = new TheTask(lock);
                 synchronized(lock)
                 {
                       task.start();
                       lock.wait();
                       endTime = System.nanoTime();
                 }

           }
           catch(InterruptedException e)
           {}
        return  (double)(endTime - startTime)/1000;
    }
}
```

Měření :

$\bar{\tau} = 150{,}21\,\mu s$, $s_\tau = 118{,}50\,\mu s$, $min_\tau = 80{,}12\,\mu s$

Závěr :

Změřil jsem switching context time ve standardním prostředí systému Android jako τ = 150,21 μs.

Příloha č. 2 : Měření switching context time v native prostředí

Popis :

Měření switching context time v native prostředí. V pomocném vlákně se sejme počáteční reálný čas pomocí funkce *clock_gettime(CLOCK_MONOTONIC,..)*, po synchronizaci pomocí semaforu s hlavním vláknem se sejme koncový reálný čas. Switching context time je určen diferencí koncového a počátečního času. Provede se 1000 měření.

Seznam přístrojů :

- Tiny6410 kit

Software :

```
static sem_t semaphore;
static timespec start;
static timespec end;
void *StartThread(void* arg)
{
    struct timespec sleepTime;
    struct timespec remainingSleepTime;
    sleepTime.tv_sec=0;
    sleepTime.tv_nsec=500;
    clock_gettime(CLOCK_MONOTONIC, &start);
    sem_post(&semaphore);
    nanosleep(&sleepTime,&remainingSleepTime);
}

jlong measure()
{
    sem_init(&semaphore, 0, 0);
    pthread_t thread1;
    int rc = pthread_create(&thread1, NULL, StartThread, (void*)NULL);
    if (rc)
    {
          return -1;
    }
    sem_wait(&semaphore);
    clock_gettime(CLOCK_MONOTONIC, &end);
    sem_destroy(&semaphore);
    int ret = 0;
    pthread_join(thread1, (void**)&ret);
    return end.tv_nsec - start.tv_nsec;
}
```

Měření :

$\bar{\tau} = 45{,}51\,\mu s$, $s_\tau = 5{,}60\,\mu s$, $min_\tau = 34{,}93\,\mu s$

Závěr :

Změřil jsem switching context time v nativním prostředí systému Android s použitím systémového časovače jako $\tau = 34{,}93\ \mu s$.

Příloha č. 3 : Měření switching context time v native prostředí pomocí osciloskopu

Popis :

Měření switching context time v native prostředí pomocí LED výstupu (výstup pracuje v negativní logice). V pomocném vlákně se výstup nastaví, po synchronizaci s hlavním vláknem se výstup shodí. Tento čas je měřen osciloskopem.

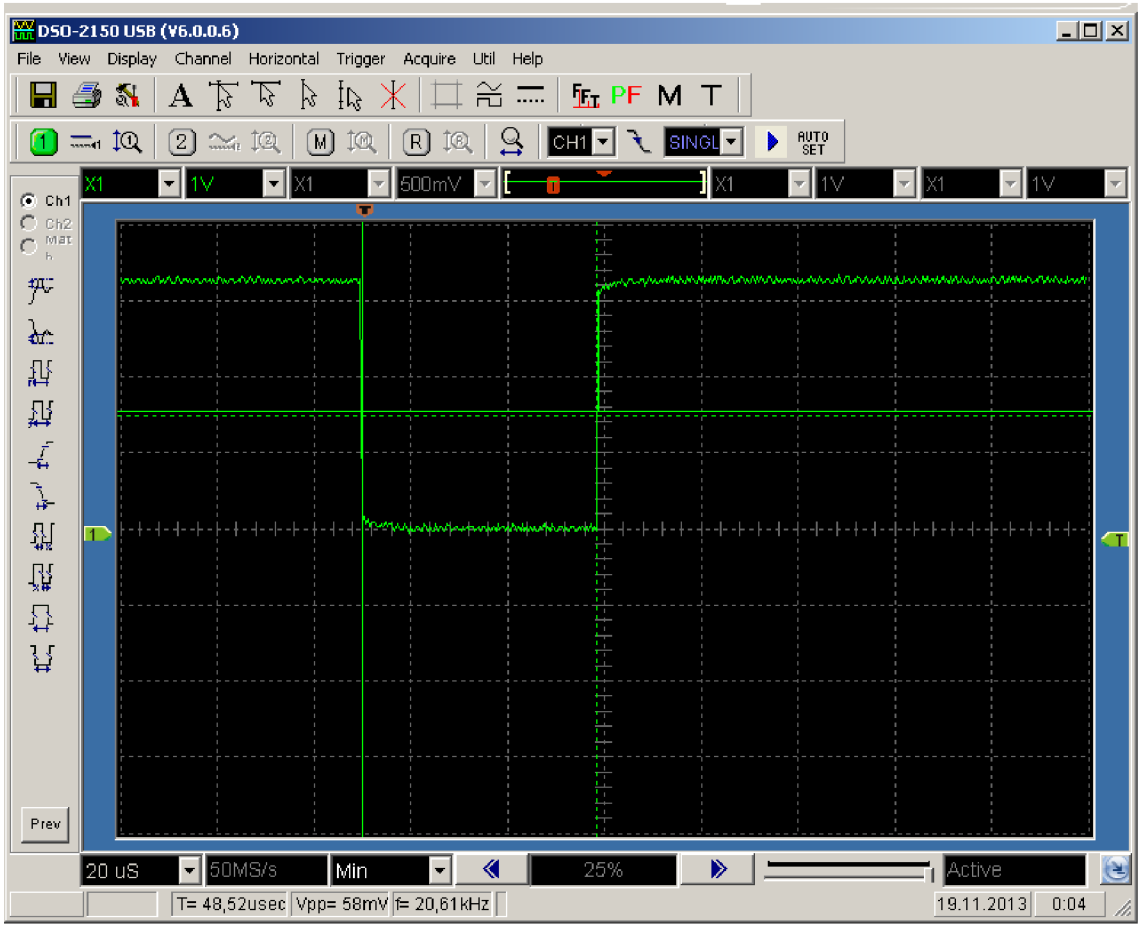
Seznam přístrojů :

- osciloskop Hantek DSO-2150 USB, v.č.4964836
- Tiny6410 kit

Software :

```
static sem_t semaphore;
static FILE* fd = NULL;
void *StartThread(void* arg)
{
    struct timespec sleepTime;
    struct timespec remainingSleepTime;
    sleepTime.tv_sec=0;
    sleepTime.tv_nsec=500;
    fd = fopen("/dev/leds", "r");
    ioctl(fileno(fd), 1, 1);
    sem_post(&semaphore);
    nanosleep(&sleepTime,&remainingSleepTime);
    fclose(fd);
}
jlong measure()
{
    sem_init(&semaphore, 0, 0);
    fd = fopen("/dev/leds", "r");
    pthread_t thread1;
    int rc = pthread_create(&thread1, NULL, StartThread, (void*)NULL);
    if (rc)
    {
          return -1;
    }
    sem_wait(&semaphore);
    ioctl(fileno(fd), 0, 1);
    fclose(fd);
    sem_destroy(&semaphore);
    int ret = 0;
    pthread_join(thread1, (void**)&ret);
    return 0;
}
```

Obrazovka osciloskopu :



Obr. 3-1 Obrazovka osciloskopu při měření

Měření :

Číslo	1	2	3	4	5	6	7	8	9	10
τ [μs]	48,52	46,41	46,11	47,92	48,82	49,42	53,04	47,01	44,60	48,82

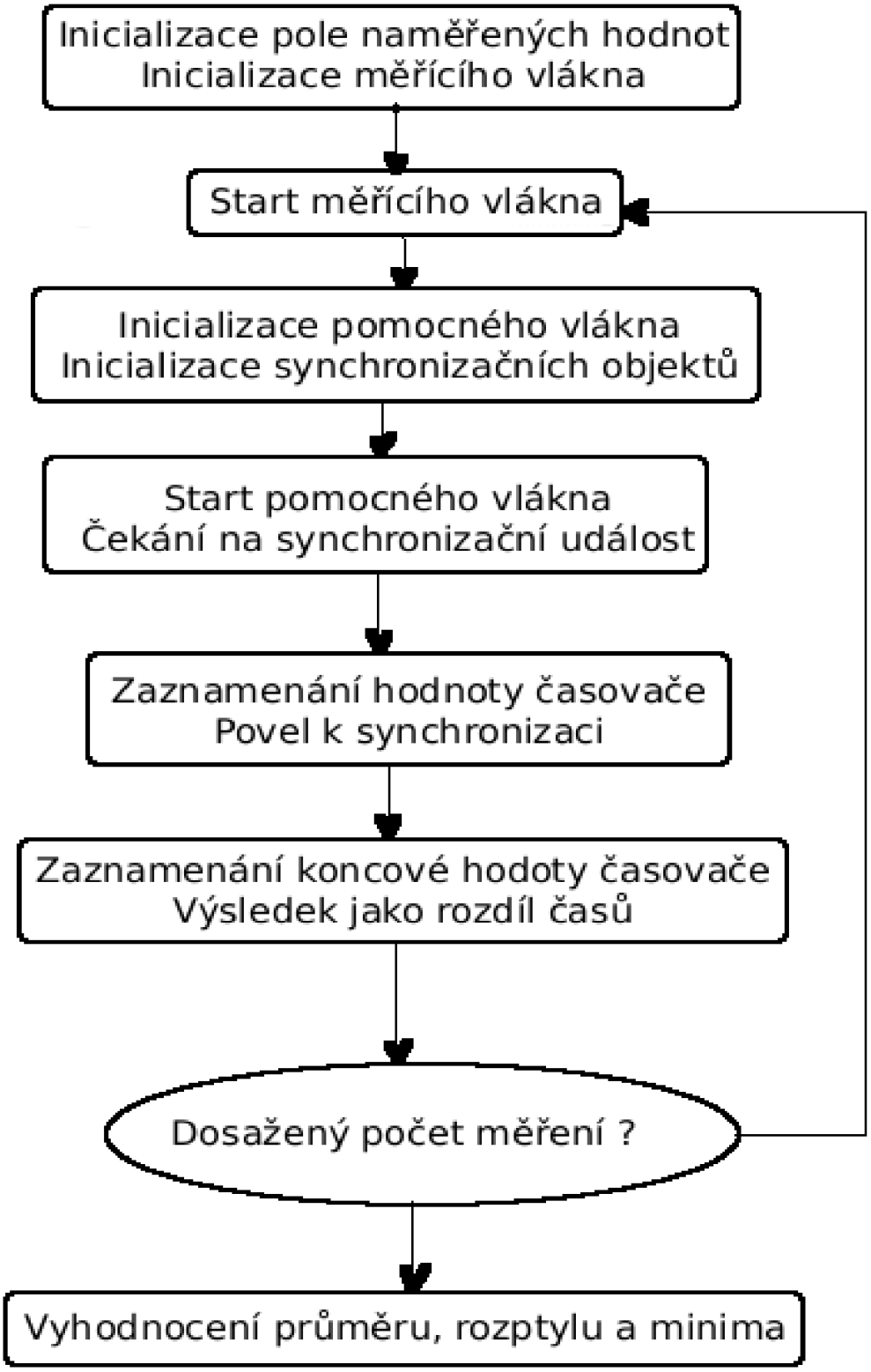
Tab. 4-1 Hodnoty měření switching context time pomocí osciloskopu

$$\bar{\tau} = 48{,}12\,\mu s \quad , \quad s_\tau = 2{,}34\,\mu s \quad , \quad min_\tau = 44{,}60\,\mu s$$

Závěr :

Změřil jsem switching context time v nativním prostředí systému Android s použitím externího osciloskopu jako τ = 44,60 μs.

Příloha č. 4 : Stavový diagram měření switching context time



Obr. 4-1 Stavový diagram měření switching context time

Příloha č. 5 : Měření granularity časovače

Popis :

Měření granularity časovače v native prostředí. Několikrát za sebou je volána funkci pro získání diference hned po sobě jdoucích volání API funkce *clock_gettime(CLOCK_MONOTONIC,..)* pro získání aktuální hodnoty časovače. V případě nulového výsledku se nestihla hodnota časovače obnovit. Provede se 1000 měření.

Seznam přístrojů :

- Tiny6410 kit

Software :

```
static timespec start;
static timespec end;
jlong measure()
{
    while(true)
    {
          clock_gettime(CLOCK_MONOTONIC, &start);
          clock_gettime(CLOCK_MONOTONIC, &end);
          if (end.tv_nsec != start.tv_nsec)
                break;
    }
    return end.tv_nsec - start.tv_nsec;
}
```

Měření :

$\bar{\tau} = 1{,}228\,\mu s \quad , \quad s_\tau = 0{,}175\,\mu s \quad , \quad min_\tau = 1{,}052\,\mu s$

Výpočty :

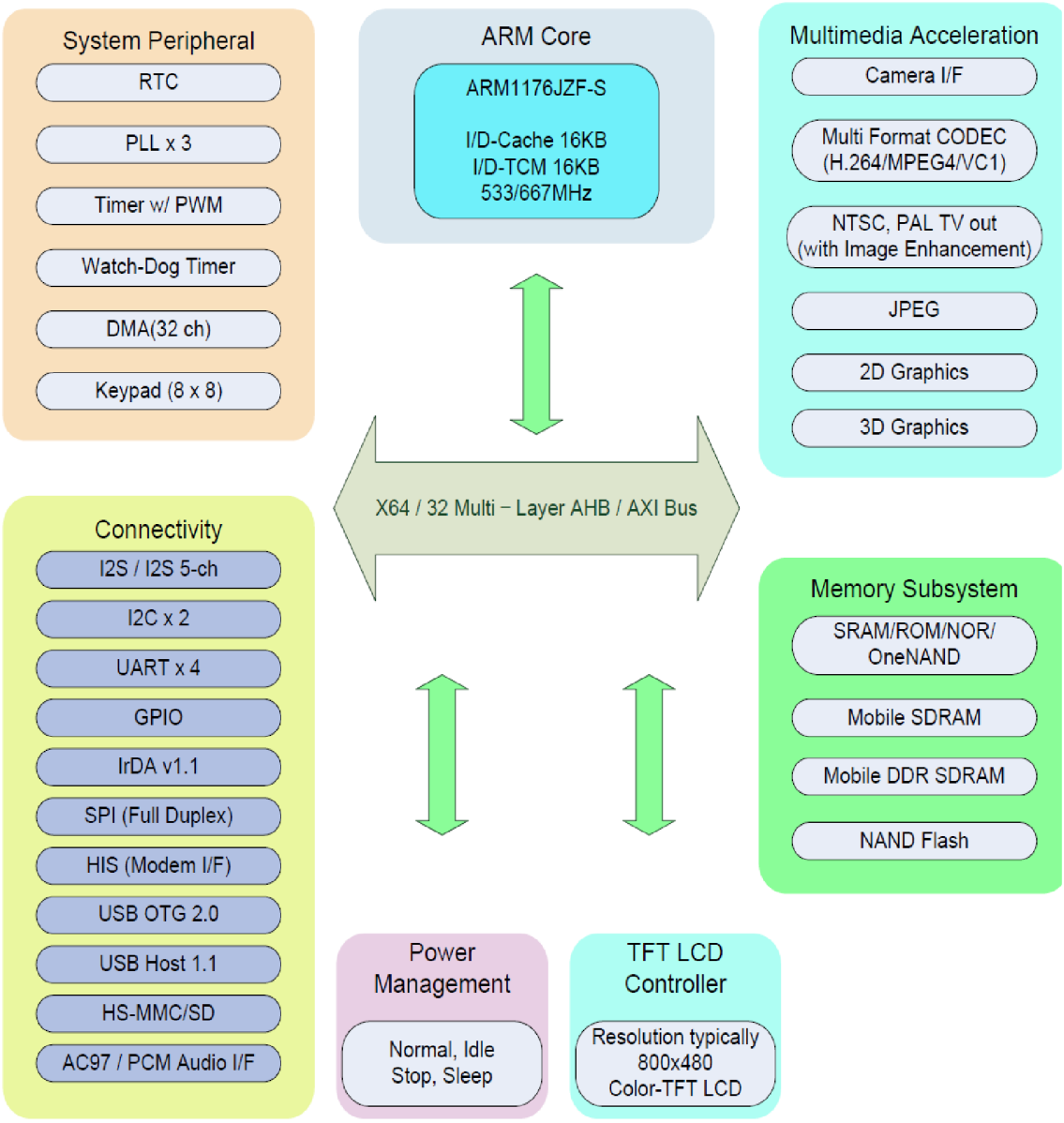
$$u_{Bz} = \frac{\Delta z_{max}}{\chi} = \frac{0{,}614}{\sqrt{3}} = 0{,}355\,\mu s \quad , \quad u_B = \sqrt{\sum u_{Bz}^2} = 0{,}355\,\mu s$$

$$u = 2.\mathrm{u}_c = 2\sqrt{u_A^2 + u_B^2} = 0{,}710\,\mu s$$

Závěr :

Změřil jsem granularitu časovače v nativním prostředí systému Android τ = 1,228 μs a z toho vyčíslil kombinovanou nejistotu pro měření s tímto časovačem u = ,710 μs.

Příloha č. 6 : Blokové schéma subsystémů S3C6410



Obr. 2-2 S3C6410 blokové schéma [2]

Příloha č. 7 : Měření času operace s pamětí

Popis :

Změří se čas, který je potřeba pro zapsání hodnoty typu double (8 byte) ve for-cyklu do každého prvku pole v paměti statické (pole deklarováno jako static) a dynamické (pole inicializováno těsně před měřením), a to v native i Java prostředí. Tento čas bude změřen pro různě velká pole a výsledky budou vyneseny do grafu. Měření časů bude provedeno vždy jen po startu aplikace pro každou hodnotu zvlášť, aby se neuplatňoval značný časový přínos cache pamětí a také proto, že velikost statického pole musí být známa při spuštění aplikace. Také bude změřena časová režie dynamické alokace.

Seznam přístrojů :

- Tiny6410 kit

Software v Java prostředí :

```
public static int MATRIX_CAPACITY = 200;
static double static_matrix[] = new double[MATRIX_CAPACITY];
double[] dynamic_matrix;
public void ledOff(View v) {
          long startTime = 0;
          long endTime = 0;
          long dynamic_time = 0;
          long static_time = 0;
          long allocation_time = 0;
          int i = 0;

          startTime = System.nanoTime();
          dynamic_matrix = new double[MATRIX_CAPACITY];
          endTime = System.nanoTime();
          allocation_time = endTime - startTime;

          startTime = System.nanoTime();
          for(i = 0; i < MATRIX_CAPACITY; i++)
                dynamic_matrix[i] = 0;
          endTime = System.nanoTime();
          dynamic_time = endTime - startTime;

          startTime = System.nanoTime();
          for(i = 0; i < MATRIX_CAPACITY; i++)
                static_matrix[i] = 0;
          endTime = System.nanoTime();
          static_time = endTime - startTime;
          //result : dynamic_time, static_time, allocation time
    }
```

Měření v Java prostředí :

Kapacita [-]	100	200	500	1000	2000	5000	10000	20000
Dynamic [ms]	0,059	0,127	0,881	2,798	2,633	5,491	3,692	4,233
Static [ms]	0,051	0,099	0,308	2,316	2,118	3,198	2,972	3,629
Alokace [ms]	0,034	0,095	0,101	0,125	0,153	0,338	0,750	1,234

Tab. 4-2 Měření času práce s pamětí v Java prostředí

Software v native prostředí :

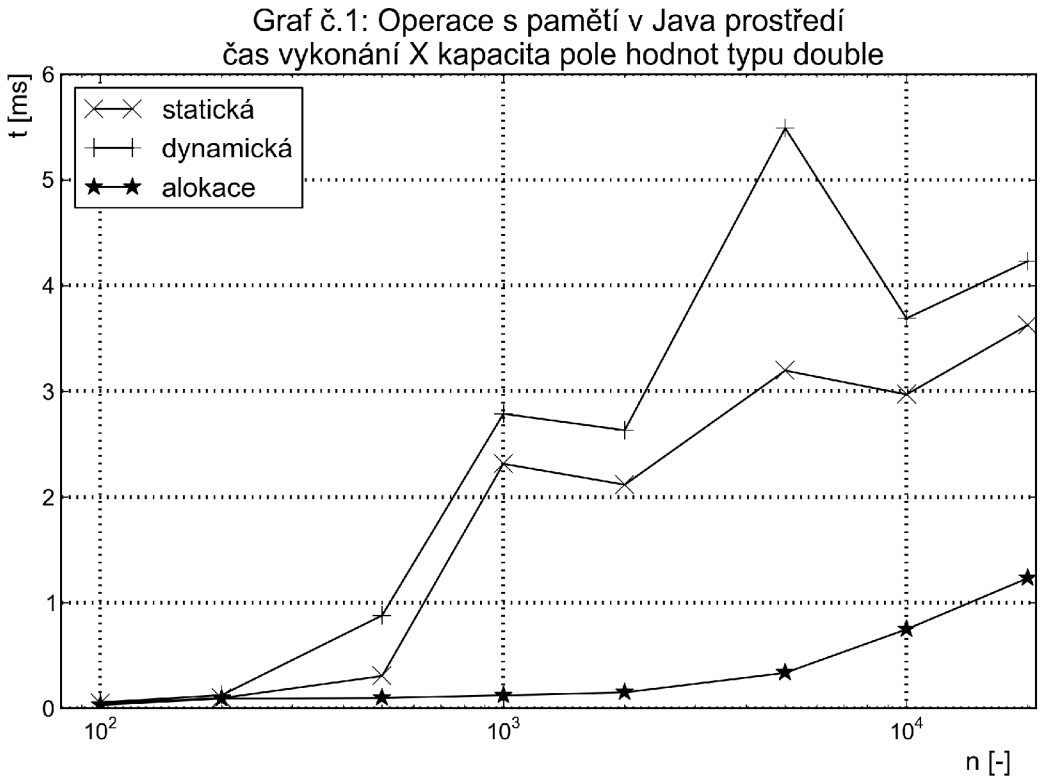
```
const int MATRIX_CAPACITY = 20000;
static double static_matrix[MATRIX_CAPACITY] = {0};
jlong measure_static()
{
    int i = 0;
    clock_gettime(CLOCK_MONOTONIC, &start);
    for(i = 0; i < MATRIX_CAPACITY; i++)
          static_matrix[i] = 0;
    clock_gettime(CLOCK_MONOTONIC, &end);
    return end.tv_nsec - start.tv_nsec;
}

jlong measure_dynamic()
{
    int i = 0;
    clock_gettime(CLOCK_MONOTONIC, &start);
    double* dynamic_matrix = (double*)malloc((unsigned long
          int)sizeof(double)*MATRIX_CAPACITY);
    clock_gettime(CLOCK_MONOTONIC, &end);
    //result : allocation time

    clock_gettime(CLOCK_MONOTONIC, &start);
    for(i = 0; i < MATRIX_CAPACITY; i++)
          dynamic_matrix[i] = 0;
    clock_gettime(CLOCK_MONOTONIC, &end);
    return end.tv_nsec - start.tv_nsec;
}
```

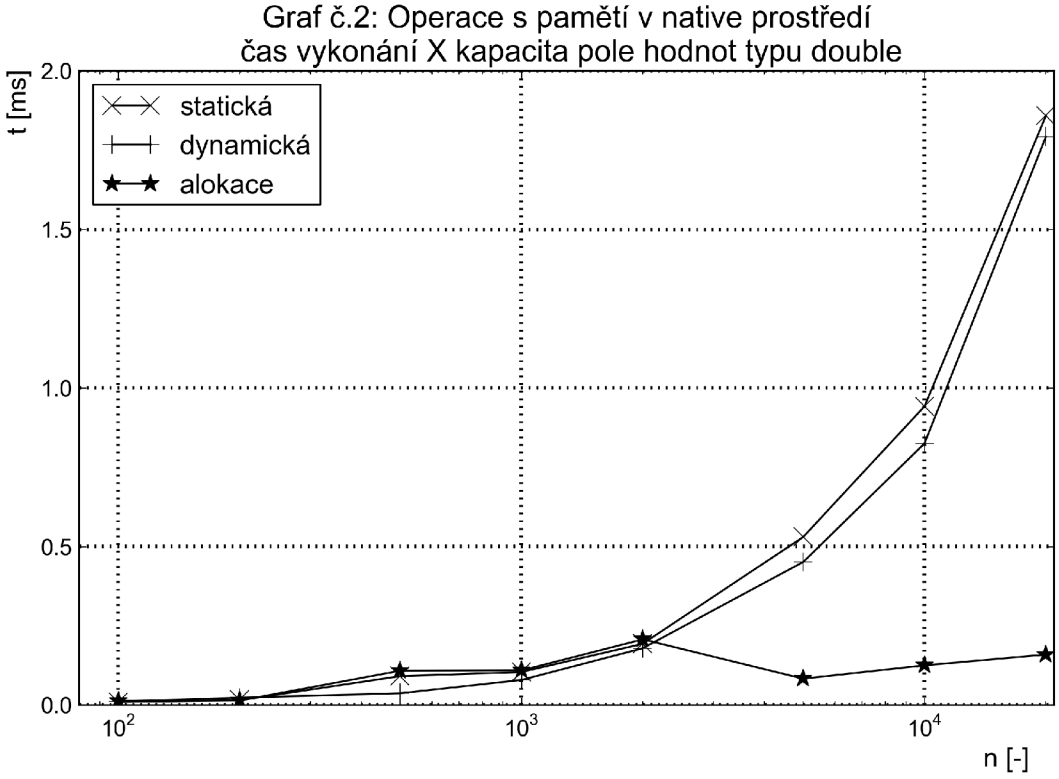
Měření v native prostředí :

Kapacita [-]	100	200	500	1000	2000	5000	10000	20000
Dynamic [ms]	0,012	0,024	0,038	0,080	0,180	0,452	0,826	1,793
Static [ms]	0,010	0,016	0,092	0,105	0,194	0,531	0,943	1,859
Alokace [ms]	0,014	0,015	0,109	0,110	0,208	0,084	0,126	0,160

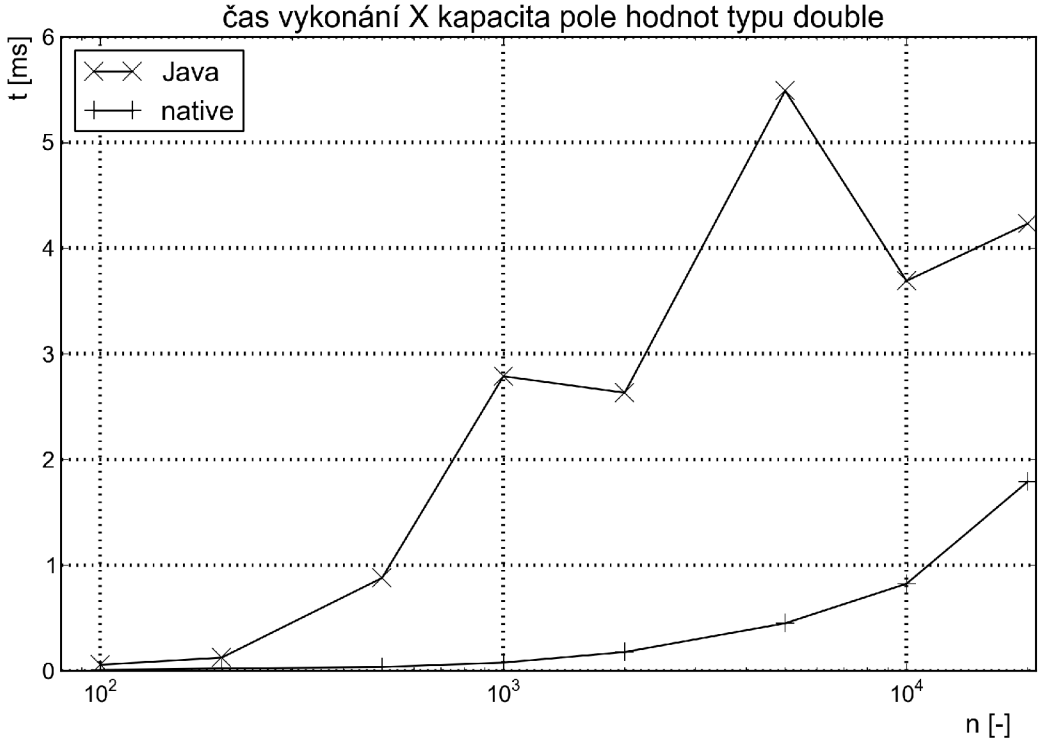
Tab. 4-2 Měření času práce s pamětí v native prostředí

Grafy :

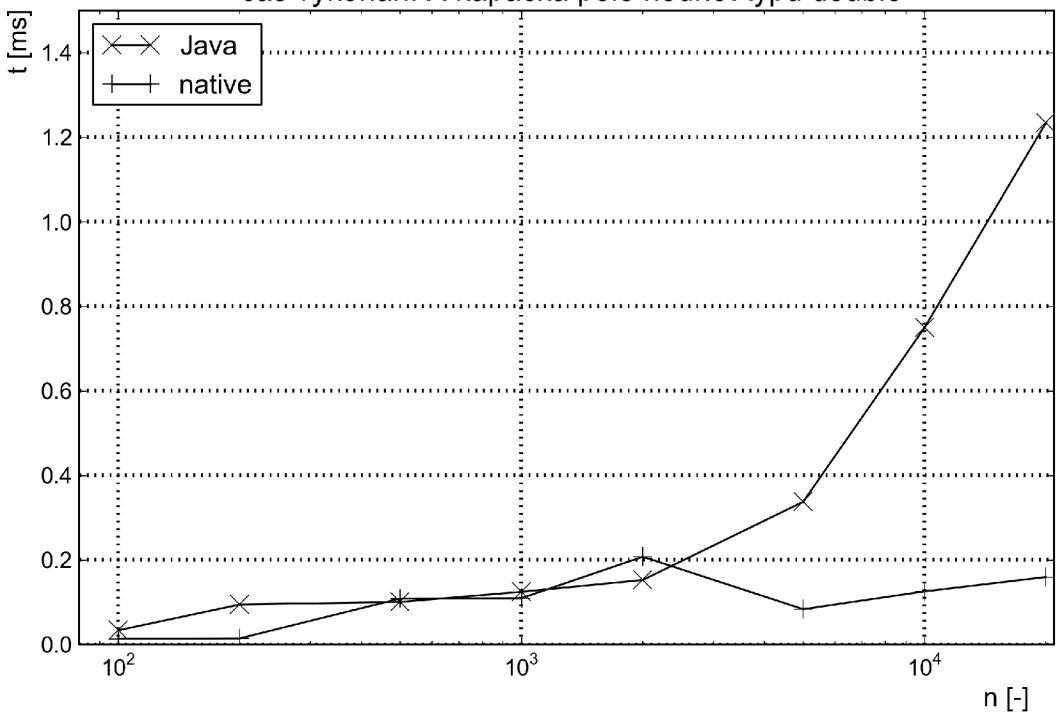

Graf č.1: Operace s pamětí v Java prostředí
čas vykonání X kapacita pole hodnot typu double
t [ms]
n [-]
statická
dynamická
alokace
0
1
2
3
4
5
6
10^2
10^3
10^4


Graf č.2: Operace s pamětí v native prostředí
čas vykonání X kapacita pole hodnot typu double
t [ms]
n [-]
statická
dynamická
alokace
0.0
0.5
1.0
1.5
2.0
10^2
10^3
10^4

Graf č.3: Operace s dynamickou pamětí
čas vykonání X kapacita pole hodnot typu double



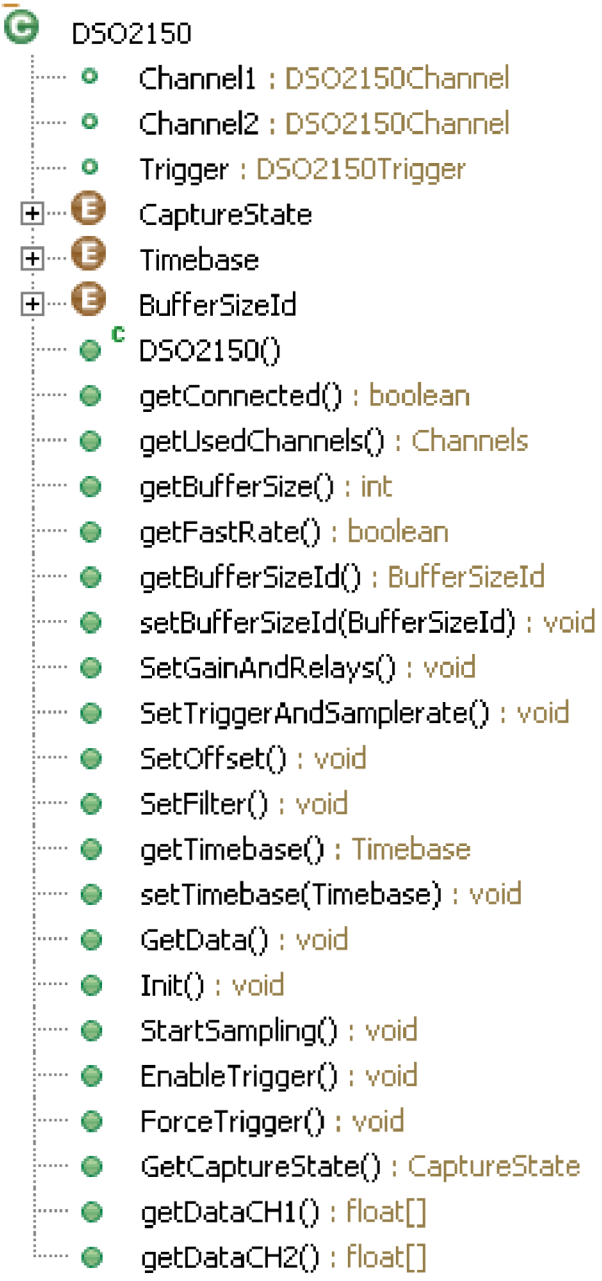
Graf č.4: Alokace dynamické paměti
čas vykonání X kapacita pole hodnot typu double



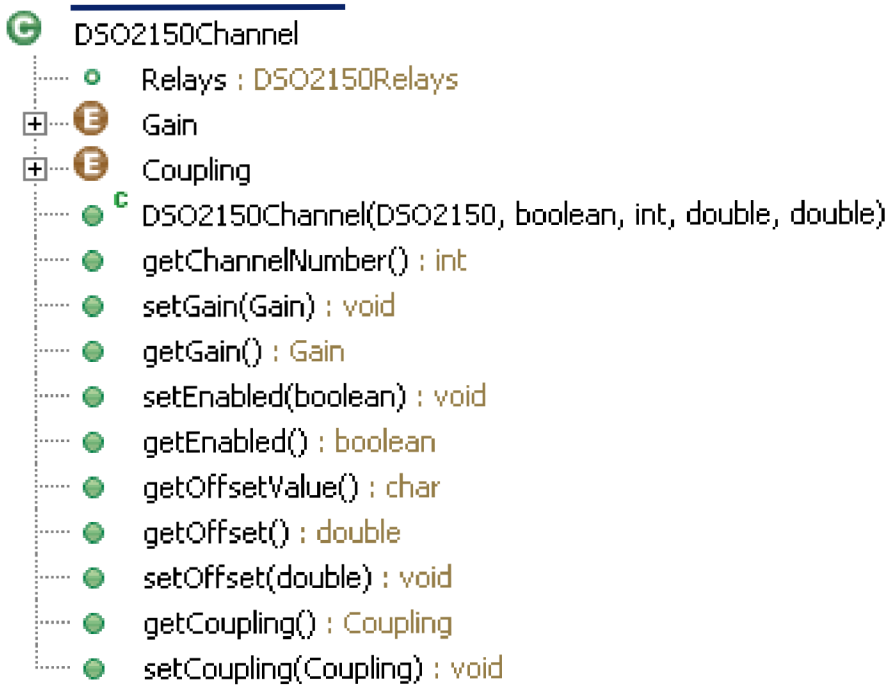
Závěr :

Změřil jsem časové režie průchodu pole ve statické a dynamické paměti systému Android v klasickém Java i v nativním prostředí a režii alokace dynamické paměti. Časové režie operací se statickou a dynamickou pamětí jsou v jednotlivých prostředích srovnatelné. V Java prostředí je časová režie značně vyšší. Časová režie alokace dynamické paměti je v native prostředí dle měření nedeterministická a u menších alokovaných celků nezanedbatelná v porovnání s vlastním průchodem alokované paměti. V Java prostředí roste s velikostí alokované paměti.

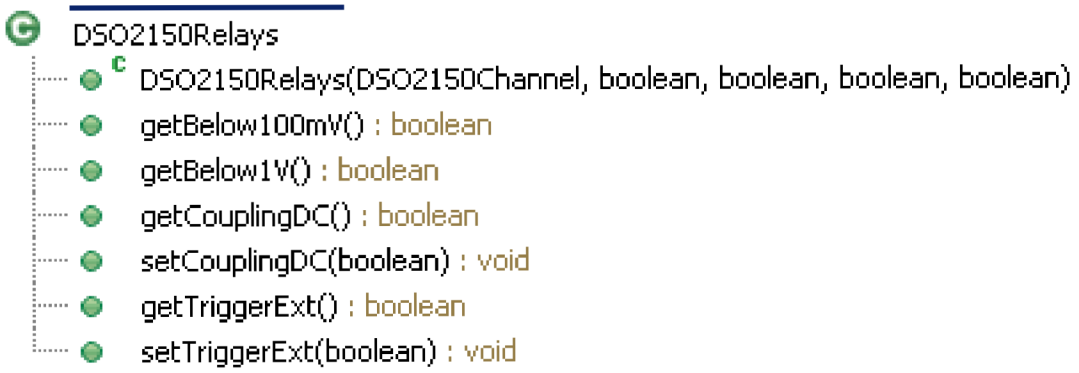
Příloha č. 8 : UML diagramy GUI části DSO2150



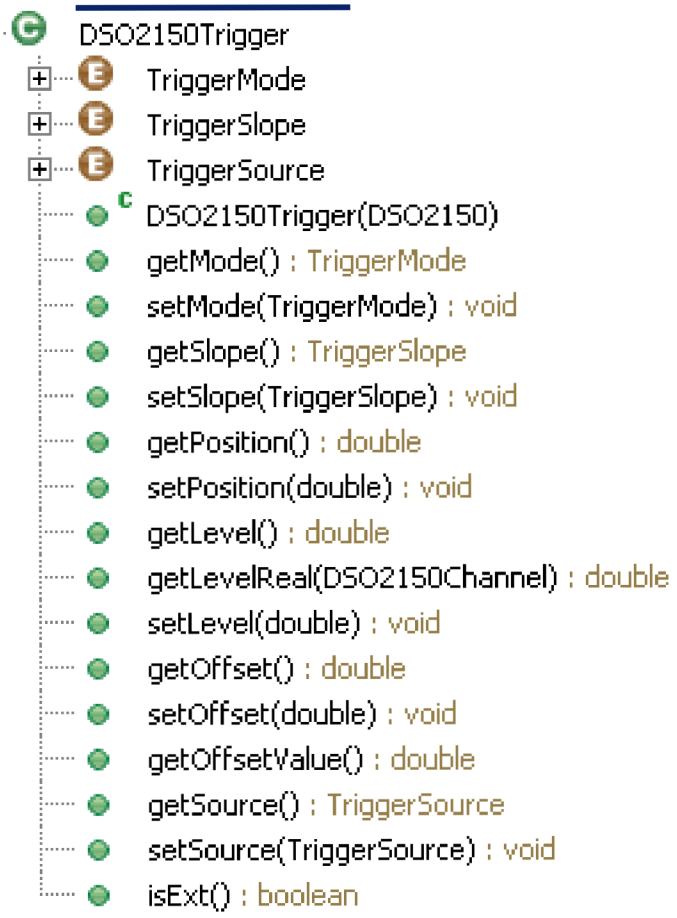
Obr. 5-14 UML diagram třídy DSO2150



Obr. 5-15 UML diagram třídy DSO2150Channel

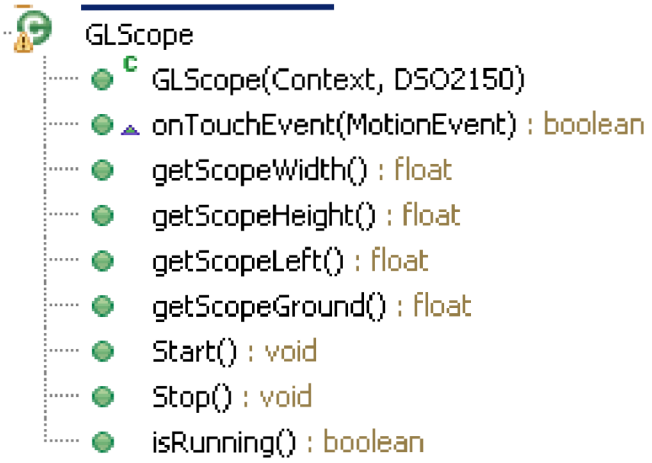


Obr. 5-16 UML diagram třídy DSO2150Relays

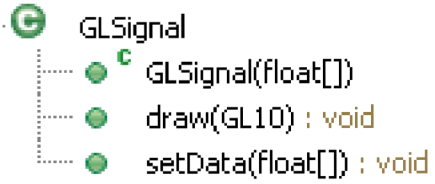


Obr. 5-17 UML diagram třídy DSO2150Trigger

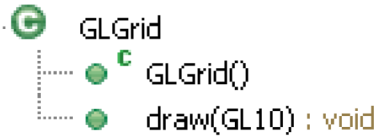
Příloha č. 9 : UML diagramy tříd OpenGL ES zobrazovače



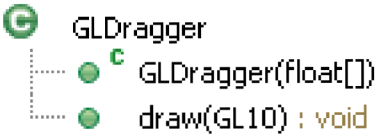
Obr. 5-18 UML diagram třídy GLScope



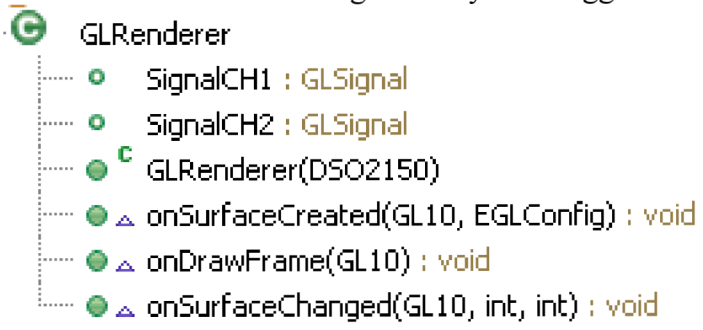
Obr. 5-19 UML diagram třídy GLSignal



Obr. 5-20 UML diagram třídy GLGrid



Obr. 5-21 UML diagram třídy GLDragger



Obr. 5-22 UML diagram třídy GLRenderer