VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta informačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2017 Bc. Jan Havran

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

KOMUNIKACE NA ČIPU ADSP-SC58X

COMMUNICATION ON THE ADSP-SC58X CHIP

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JAN HAVRAN
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN VIKTORIN
SUPERVISOR

BRNO 2017

Zadání diplomové práce/19991/2016/xhavra13

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Havran Jan, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Komunikace na čipu ADSP-SC58x**
Communication on the ADSP-SC58x Chip
Kategorie: Vestavěné systémy

Pokyny:

1. Seznamte se s rodinou čipů ADSP-SC58x od společnosti Analog Devices obsahující DSP a ARM na jednom čipu.
2. Seznamte se s vybraným vývojovým kitem obsahujícím tento čip a nastudujte dostupná prostředí a další vybavení pro vývoj.
3. Navrhněte způsob komunikace mezi DSP a ARM části čipu, který umožní kromě synchronizace také výměnu dat.
4. Navržený způsob implementujte a demonstrujte funkčnost na ukázkové aplikaci. Odměřte parametry vaší implementace.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího rozšíření.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Viktorin Jan, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Tato práce se zabývá návrhem komunikace mezi jádry SHARC a ARM na platformě ADSP-SC58x, konkrétně mezi bare-metal a Linuxovými aplikacemi na čipu ADSP-SC589. Pro komunikaci jsou nastíněny současné možnosti synchronizace a přenosu dat, jako MCAPI, MDMA nebo využití sdílené paměti. Dále jsou navrženy a implementovány způsoby komunikace s využitím těchto prostředků.

Abstract

This projects describes the design of communication between SHARC and ARM cores on ADSP-SC58x platform, concretely between bare-metal and Linux applications on ADSP-SC589 chips. There are outlined several available technologies for data transfer, such as MCAPI, MDMA or shared memory. There are also designed and implemented new communication principes based on current implementations of these technologies.

Klíčová slova

Linux, ARM, SHARC, SoC, meziprocesorová komunikace, MCAPI, DMA

Keywords

Linux, ARM, SHARC, SoC, inter-processor communication, MCAPI, DMA

Citace

HAVRAN, Jan. *Komunikace na čipu ADSP-SC58x.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Viktorin Jan.

Komunikace na čipu ADSP-SC58x

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jana Viktorina. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Havran
23. května 2017

Poděkování

Chtěl bych poděkovat Ing. Janu Viktorinovi za mimořádnou ochotu a cenné rady, které mi byly poskytnuty nejen při tvorbě této práce.

Obsah

Kapitola 1

Úvod

Výpočetní technika dnes najde využití v řadě profesí – od nemocnic přes vojenství až po zábavní průmysl. Díky rozvoji technologií se čím dál častěji objevují malá zařízení, která jsou výkonná a zachovávají si nízkou spotřebu. Ostatně nízká spotřeba je v určitých segmentech průmyslu, jako mobilní telefony a různá vestavěná zařízení, důležitá. Z důvodu zvyšování výkonu při zachování nízké spotřeby se čím dál častěji obvody integrují do jediného čipu. Takové čipy mohou mít integrovány procesory, paměti a různé řadiče. Na čipu se tak nachází celý systém – nazývá se System on Chip.

Dalším způsobem jak snížit spotřebu je použít specializované obvody. Ačkoliv procesory pro obecné použití obsahují technologie pro optimalizaci určitého typu úlohy (například instrukce SIMD), nedosahují takové efektivity jako čipy, které byly vyvinuté pro konkrétní využití – například specializované obvody ASIC. Zároveň může být někdy nutné, aby si i takto optimalizované čipy udržely jistou možnost flexibility při tvorbě aplikací. To splňují například signální procesory DSP.

Existují ale situace, kdy je vedle specializovaných procesorů potřeba i obecný procesor – například pro snadnější komunikaci s vnějším prostředím. Objevují se tak platformy, které mají integrovány jak obecné procesory, tak specializovanější obvody typu DSP. Mezi takové platformy se řadí i rodina čipů ADSP-SC58x od firmy Analog Devices. Tyto čipy obsahují vedle specializovaných DSP procesorů (SHARC) pro práci s algoritmy optimalizovanými na audio úlohy také obecný procesor (ARM) pro komunikaci s okolním prostředím, ať už se jedná o počítačovou síť nebo různé připojitelné periférie. Pro takovéto platformy je velmi důležité vyřešit komunikaci mezi DSP a obecnými procesory, která bude dosahovat dostatečné přenosové rychlosti (zvládne přenášet data z výkonných DSP procesorů) a umožní synchronizaci procesorů. Právě návrhem komunikace se zabývá tato práce.

Kapitola 2 se zabývá popisem softwarových a hardwarových technologií používaných ve vestavěných zařízeních. V kapitole 3 je popsán proces bootování a tvorby ovladačů pro Linuxové jádro. Po hardwarové stránce je blíže rozebrána platforma ADSP-SC58x v kapitole 4. V následující kapitole 5 jsou popsány postupy vývoje aplikací pro SHARC jádra, bootování Linuxu a jeho softwarová podpora.

Současné prostředky pro komunikaci mezi ARM a SHARC jádry jsou nastíněny v kapitole 6, kde jsou rovněž popsány nové způsoby komunikace využívajících stávajících prostředků. Závěrečná kapitola 7 shrnuje zhodnocení výsledků práce a další možný vývoj tohoto projektu.

V návaznosti na semestrální projekt byl využit popis softwarových a hardwarových prostředků zvoleného vývojového kitu pro návrh a implementaci nových způsobů komunikace.

Kapitola 2

Vestavěná zařízení

V této kapitole jsou popsány procesory ARM, které bývají použity ve vestavěných systémech jako obecné procesory. Dále jsou uvedeny některé používané technologické platformy a jejich srovnání. Nakonec je představen standard MCAPI sloužící pro komunikaci a synchronizaci právě takových obvodů.

2.1 Procesory ARM

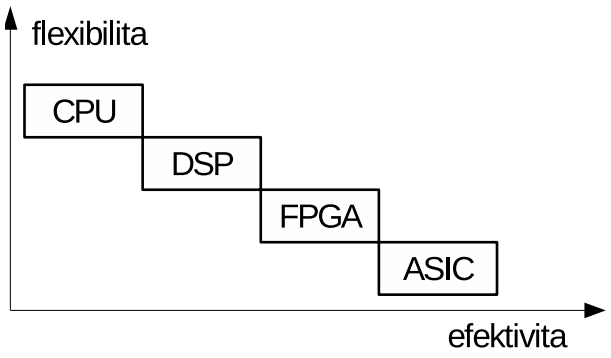
ARM[1] procesory jsou často nasazovány ve vestavěných zařízeních, kde je spotřeba procesoru klíčovým faktorem. Tyto procesory vyvíjí a licencuje výrobcům hardware firma ARM Holdings. Na rozdíl od běžně používaných procesorů x86 od firem jako AMD a Intel patří ARM procesory mezi RISC[2] architekturu. Tato architektura je oproti CISC[3] architektuře, jak ostatně vyplývá z názvu, specifická redukovanou instrukční sadou, která je hardwarově implementovaná automatem. Díky tomu je procesor jednodušší ve smyslu velikosti čipu a podporu jeho instruční sady lze snadněji implementovat pro různé překladače. RISC architektura také disponuje velkým počtem univerzálních registrů. Další rozdíl je v přístupu k paměti: RISC architektura tuto operaci umožňuje pouze instrukcemi *Load* a *Store*, což, mimo jiné, zjednodušuje práci kompilátorům při optimalizaci.

Procesory ARM mají podobně jako x86 procesory (AMD/Intel) speciální instrukce pro práci s multimediálními daty (jako je zvuk či video). Pro ARM procesory se tato technologie nazývá NEON a rozšiřuje instrukční sadu o 64 a 128 bitové instrukce, které umí pracovat s 8, 16, 32 a 64 bitovým celými čísly a 32 bitovými čísly s plovoucí desetinnou čárkou dle standardu IEEE [12].

2.2 Další technologické platformy

Kromě procesorů pro obecné použití, mezi které patří například ARM, existuje mnoho dalších výpočetních technologických platforem, které se od sebe liší obtížností návrhu, flexibilitou a efektivitou (porovnání na obrázku 2.1). Takové platformy bývají obvykle nasazovány spolu s obecnými procesory. Následuje popis vybraných platforem.

[1] Advanced RISC Machine
[2] Reduced Instruction Set Computing
[3] Complex Instruction Set Computing



Obrázek 2.1: Porovnání technologických platforem z hlediska efektivity a flexibility

DSP

Jedná se o mikroprocesory specializované na zpracování digitálních signálů, které byly typicky získány převodem z analogových signálů pomocí A/D převodníků. DSP procesory mají zpravidla podporu SIMD instrukcí pro zpracování multimediálních dat. Ačkoliv jsou procesory optimalizované na konkrétní typ algoritmů, tak si stále zachovávají určitou flexibilitu.

FPGA

FPGA[4] jsou integrované obvody obsahující hardwarově programovatelné bloky, díky čemuž je možné upravovat už vyrobený obvod. Tyto programovatelné bloky jsou umístěny do matice propojů, kde mohou být různě propojovány a vytvářet tak konkrétní logiku. Omezujícím faktorem FPGA je obvykle množství zdrojů a složitá konfigurace. Naopak jeho výhodou je paralelizace a rychlost hardwarově implementovaných algoritmů. FPGA obvody se dají programovat v jazycích VHDL[5] a Verilog. V posledních letech se také objevují nástroje, s jejichž pomocí lze obvody programovat v jazyce C.

ASIC

ASIC[6] jsou integrované obvody navržené pro řešení konkrétních úloh. Díky tomu jsou výpočetně efektivní, ale málo flexibilní, což je jejich nevýhoda oproti FPGA. Proto je při vývoji ASIC procesorů nutné věnovat velké prostředky na ověřování návrhu.

2.3 Direct Memory Access

Vestavěná zařízení zpravidla obsahují několik periférií, které potřebují komunikovat s hlavním procesorem. Konfigurace periférií může probíhat pomocí přístupu do izolovaných vstupů/výstupů instrukcemi IN/OUT (tyto vstupy/výstupy mají vlastní adresový prostor), nebo v případě namapování registrů do paměťového prostoru pomocí standardních instrukcí pro práci s pamětí. Pro dlouhé datové přenosy jsou ale tyto způsoby komunikace nevhodné – vytěžují totiž procesor, který musí každý takový přenos realizovat. Pro tyto účely se

[4] Field-programmable gate array
[5] VHSIC Hardware Description Language
[6] Application-specific integrated circuit

využívá principu přímého přístupu do paměti – DMA[7], kdy se o paměťové přenosy stará DMAC[8] jednotka specializovaná pro tyto účely. Tato jednotka má přístup do hlavní paměti a může tak samostatně řídit datové přenosy – procesor tak v době přenosu může vykonávat užitečnou činnost [19].

Paměťové přenosy lze rozdělit na dva přístupy:

- Bus mastering – kromě procesoru mohou řídit přenosy přes sběrnici také periférie. Periferie zahajuje přenosy požadavkem na přidělení sběrnice. Jakmile je sběrnice periferii přidělena, tak může bez spolupráce procesoru přenášet data po sběrnici.
- Central DMA – ke sběrnici má přístup specializovaný obvod – DMA řadič (DMAC). Přenos dat tak probíhá přes tento řadič, kdy periférie nebo procesor nastaví parametry přenosu a samotná data pak přenáší DMAC.

DMAC lze zpravidla konfigurovat dvěma způsoby.

- Pomocí registrů – přenosy se nastavují přes přístupné registry DMA řadiče.
- Pomocí deskriptorů – konfigurace přenosu se uloží do systémové paměti a řadiči se poté předá pouze ukazatel na tuto paměť.

Některé DMA řadiče mohou být řízeny specializovanými instrukcemi, přenos je v takovém případě popsán mikroprogramem. Příkladem takové jednotky je PL330 [13].

2.4 MCAPI

Multicore Communications API je rozhraní spravované nadací *The Multicore Association*. Tento standard byl navržen pro komunikaci a synchronizaci zařízení (procesorů, DSP, ASIC nebo například FPGA) distribuovaných na společné desce nebo čipu. MCAPI je platformě nezávislé – MCAPI tedy nepopisuje konkrétní implementaci pro konkrétní systém [14].

MCAPI rozlišuje koncepty jednotlivých prostředků:

- **Doména** – tvoří skupinu nad jedním nebo více uzly. Pro běžné účely stačí jedna doména.
- **Uzel** – uzly mohou být vlákna, jádra nebo celý operační systém.
- **Endpoint** – endpointy jsou koncové body komunikace. Pro zasílání dat slouží vždy jeden endpoint jako zdroj a druhý jako cíl.

Před zahájením komunikace je potřeba nejdříve vytvořit endpointy. Pokud spolu chtějí dva uzly komunikovat, tak každý vytvoří svůj endpoint a získá endpoint svého protějšku (k tomu potřebuje znát doménu a číslo uzlu protějšku).

Po vytvoření endpointů lze zahájit komunikaci. Ta může být buď blokující, nebo neblokující. Pro blokující komunikaci platí, že operace skončí chybou, nebo po dokončení přenosu. U neblokující operace jsou nastaveny parametry přenosu a daná operace ihned skončí. V takovém případě tedy není jasné, zdali byl přenos dokončen či nikoliv. Pro tyto účely MCAPI poskytuje funkci `mcapi_wait()`, která čeká na dokončení přenosu (u této funkce lze nastavit timeout, tedy maximální dobu po kterou se má čekat), funkci `mcapi_test()` vracející informaci o tom, zda byl přenos dokončen, případně funkci `mcapi_cancel()` pro zrušení přenosu.

Data mohou být přenášena dvěma možnými způsoby: pomocí kanálů nebo zpráv [14].

[7] Direct Memory Access
[8] DMA Controller

Kanály

Kanály vyžadují point-to-point spojení pro přenos dat mezi dvěma endpointy. Rozhraní poskytuje funkce pro zaslání a přijetí dat a také pro otestování, zda jsou nějaká data připravena pro přijetí. Lze tak před voláním funkce pro příjem dat nejprve zjistit, zdali jsou vůbec nějaká k dispozici. Kanály se dále dělí na dva typy:

- **Paketový kanál** – přenáší vektorová data různé délky. Tato délka je dána konkrétní implementací.
- **Skalární kanál** – přenáší skalární data o velikosti 8, 16, 32 nebo 64 bitů.

Zprávy

Pro zasílání zpráv mezi dvěma endpointy není potřeba narozdíl od kanálů vytvářet spojení. Stejně jako pro kanály, i zde jsou funkce pro zaslání, přijetí a zjištění dostupnosti nějaké zprávy.

Kapitola 3

Linux

Linuxové jádro je svobodný systém s otevřenými zdrojovými kódy, na jehož vývoji se podílí kromě velikého množství softwarových a hardwarových firem (Google, Intel, AMD, Samsung nebo Microsoft) také komunita. Linux může být svobodně upravován, a tak vzniklo velké množství modifikací, jsou vytvářeny distribuce, které kromě Linuxového jádra přidávají i celou řadu svobodného i nesvobodného softwaru.

Linuxové jádro se řadí mezi monolitická jádra – jádro operačního systému pracuje v kernel space a má přímý přístup k hardwaru. Uživatelské aplikace využívají pro svůj běh prostředky poskytované jádrem, jenž vytváří nad hardwarem další vrstvu.

3.1 Linux pro vestavěná zařízení

Linux podporuje velké množství platforem jako jsou x86, SPARC, PowerPC nebo ARM. Díky tomu je Linux dostupný nejen na desktopech nebo serverech, ale právě i ve vestavěných zařízeních.

3.1.1 DeviceTree

Vestavěná zařízení bývájí zpravidla navržena pro konkrétní účel, proto se od sebe liší nejen použitými typy obvodů, ale díky licenční politice ARM si mohou firmy svoje integrované obvody dále upravovat. Vzniká tak velké množství konfigurací, kdy se používají různě konfigurované periférie, implementované obvody nebo například registry bývají mapovány na různé adresy, a to i přes to, že se může jednat o stejné komponenty. Tyto problémy se snaží řešit DeviceTree[1], což jsou stromově strukturované soubory využívané k popisu hardwaru na konkrétním čipu. Linuxové ovladače tak nemusejí mít pro každou konfiguraci periférie speciální kód, ale informace, jako počet použitých přerušení nebo adresy registrů, získají z DeviceTree.

Každý DeviceTree soubor je tvořen z uzlů (nodes) tvořící stromovou strukturu, kde kořenovým uzlem je uzel "/". Úplná cesta konkrétního uzlu je tedy tvořena kořenovým uzlem, dále rodičovskými uzly daného uzlu a nakonec samotným uzlem, přičemž uzly jsou odděleny znakem "/". Každý uzel je pak tvořen svými vlastnostmi (properties) nebo poduzly (potomky). Vlastnost je tvořena dvojicemi názvem vlastnosti a její hodnotou [17]. DeviceTree formát existuje v textové (přípona .dts) a binární (přípona .dtb) variantě. Převod mezi těmito dvěma formáty lze provézet pomocí DeviceTree kompilátoru (dtc):

[1] http://www.devicetree.org

```
$ dtc -I dts -O dtb <VSTUPNÍ SOUBOR>.dts > <VÝSTUPNÍ SOUBOR>.dtb

$ dtc -I dtb -O dts <VSTUPNÍ SOUBOR>.dtb > <VÝSTUPNÍ SOUBOR>.dts
```

DeviceTree je využíván Linuxovým jádrem pro identifikaci zařízení a runtime konfiguraci [20]. Například pomocí *compatible* lze určit jaký ovladač bude použit pro zařízení definované daným uzlem – takový ovladač bude mít stejný název pro tuto vlastnost. Jádro obsahuje podpůrné funkce (předpona *of_**[2]) v hlavičkových souborech *include/linux/of*.h*. Tyto funkce mohou být využity například pro získání čísla přerušení pro dané zařízení (uzel), přidělené části adresového prostoru atd. Pro architekturu ARM jsou DeviceTree soubory uloženy v podadresáři *linux/arch/arm/boot/dts/* zdrojového kódu Linuxového jádra.

3.1.2 Bootování

Bootování vestavěných systémů se může odlišovat mezi výrobci. Po zapnutí systému začne bootovací procesor (například na ADSP-SC58x architektuře se jedná o ARM procesor) zpravidla načítat instrukce ze speciální paměti – Boot ROM. Tato paměť obsahuje první instrukce, které procesor vykoná. Na některých systémech bývají také přítomny různé přepínače, kterými lze pro Boot ROM API konfigurovat zdroj bootování a jeho parametry [10, s. 3160]. Po načtení Boot ROM konfigurace je zahájeno bootování aplikace uložené na zadané periférii. Takovou aplikací pro ARM procesory může být například zavaděč U-Boot[3].

U-Boot

U-Boot je open source zavaděč používaný ve vestavěných systémech. Podporuje několik architektur včetně procesorů ARM. Stejně jako Linux pracuje s DeviceTree, který používá pro runtime konfiguraci pro konkrétní zařízení. U-Boot podporuje zavádění systému z různých periférií jako SD karty, ethernetu, sériové linky nebo flash paměti. Pro komunikaci s uživatelem je použito rozhraní příkazové řádky, odkud může být U-Boot řízen a také mohou být volány různé podprogramy [16]. Aplikace lze pro U-Boot vyvíjet dvěma různými způsoby:

- **Samostatná aplikace** – aplikace, která může být samostatně přeložena a nahrána do U-Bootu za běhu (například přes TFTP). Pro spuštění této aplikace lze použít příkaz *go*.
- **Příkaz** – speciální aplikace, která je překládána spolu s U-Bootem. Tato aplikace je součástí výsledného U-Boot obrazu a je spouštěna stejně jako ostatní příkazy U-Boot. U-Boot příkazy mohou rovněž obsahovat nápovědu k programu.

3.1.3 Buildroot

Samotné Linuxové jádro poskytuje správu prostředků, ovladače a prostředí pro běh aplikací – nikoliv samotné aplikace. Proto k němu bývají nainstalovány další aplikace podobně jako v Linuxových distribucích. Pro tyto účely vznikl projekt Buildroot.

Buildroot poskytuje nástroje, které se snaží vývojáři usnadnit proces kompilace jádra, Unixových nástrojů (například BusyBox[4]), přidávání balíčků (aplikací) a jejich záplat a konfiguraci systému pro vestavěná zařízení. Zdrojové kódy pro cílovou vestavěnou platformu

[2] OpenFirmware

[3] Universal Bootloader

[4] www.busybox.net - jedná se o jediný spustitelný soubor, kterému se při spuštění zadá jako parametr podporovaný příkaz, který bude BusyBox svým chováním napodobovat

lze překládat pomocí křížového překladu (cross-compilation), díky němu lze zdrojový kód pro jednu platformu přeložit na jiné platformě (například na výkonnějším desktopu nebo serveru). Pro konfiguraci překladu je podobně jako v Linuxu používán nástroj Kconfig [2].

3.2 Tvorba ovladačů

Pro Linux, stejně jako pro ostatní operační systémy, jsou nezbytně nutné různé ovladače sloužící pro správu paměti, přerušení, souborových systémů nebo různých periférií. Linux poskytuje pro tyto ovladače své API[5]. Pro toto API existuje dokumentace ve formě zdrojových kódů a komentářů (taková dokumentace je psána ve speciálním formátu – *kernel-doc*[6]), dále lze použít textovou dokumentaci v adresáři *Documentation* zdrojového kódu Linuxu nebo například využít webovou stránku *free-electrons*[7], kde lze online prohledávat Linuxový zdrojový kód.

3.2.1 Adresový prostor

Narozdíl od vývoje běžné aplikace lze při vývoji ovladačů v Linuxu rozlišovat adresový prostor na dva základní typy: uživatelský (user space) a jaderný (kernel space) adresový prostor.

V uživatelském adresovém prostoru běží běžné (uživatelské) aplikace, které mají dostupnou standardní knihovnu jazyka C. Tyto aplikace nemají přístup k hardwaru systému a její požadavky jako například práce se soubory jsou zpracovány právě jádrem. Stejně tak samotný běh je jádrem hlídán, a tak se i jádro postará například o neuvolněná paměť po skončení programu. Díky tomu chyba v uživatelské aplikaci nemá takové fatální následky jako v případě jaderného programu. Naproti tomu ovladače mají k hardwaru a systémovým prostředkům přímý přístup (přestože by pro tyto operace mělo být použito API poskytované jádrem). Kvůli tomu může být chybná manipulace s prostředky fatální následky pro stabilitu systému.

Jádro využívá speciálních režimů některých procesorů, v případě ARM procesorů se jedná celkem o dvě úrovně – privilegovaný a neprivilegovaný (například x86 používají 4 úrovně). Uživatelské aplikace běží v neprivilegovaném režimu, zatímco jaderné ovladače v privilegovaném.

Celkově lze při práci s jádrem rozlišovat 5 typů adres [15, s. 413]:

- **Uživatelská virtuální adresa** – adresa používaná aplikací, která je spuštěna v uživatelském prostoru.
- **Fyzická adresa** – adresa používaná při komunikaci mezi procesorem a systémovou pamětí a používá se pro ni datový typ *phys_addr_t*.
- **Adresa sběrnice** – používá se při komunikaci mezi různými periférními zařízeními a pamětí (může být shodná s fyzickou adresou). Tato adresa bývá používáná při konfiguraci DMA přenosů (datový typ *dma_addr_t*).

[5] Application Programming Interface

[6] Dokumentace k tomu formátu je dostupná na https://www.kernel.org/doc/Documentation/kernel-doc-nano-HOWTO.txt.

[7] lxr.free-electrons.com

- **Logická jaderná adresa** – běžná adresa používaná jádrem Linuxu. Logickou adresu lze alokovat například voláním funkce *kmalloc()*. Logická adresa je mapována 1:1 vůči fyzické adrese.
- **Virtuální jaderná adresa** – opět adresa využívaná jádrem, ale tentokrát nemusí být tato adresa mapovaná na fyzickou adresu. Virtuální adresu lze alokovat například voláním funkce *vmalloc()*.

3.2.2 Moduly

Linuxové jádro poskytuje po svém zavedení funkcionalitu pro obsluhu daného hardwaru a softwaru. Tuto funkcionalitu lze ale za běhu rozšířit o nové funkce díky modulům.

Moduly jsou programy běžící v jaderném režimu. Tyto programy, narozdíl od uživatelských aplikací, neobsahují funkci *main()*, která slouží jako vstupní bod do programu, jejíž ukončením skončí i daná aplikace. Jaderné programy namísto toho obsahují funkce pro načtení modulu (tato funkce se určí makrem *module_init*) a pro odstranění modulu (makro *module_exit*). Moduly pracují jako událostmi řízený program, kdy si tento program v jádře registruje operace, které bude obsluhovat. Nemají tak hlavní smyčku, ale reagují pouze na tyto události [21].

Pro zavedení modulu slouží aplikace *insmod*, případně *modprobe* a pro odstranění modulu aplikace *rmmod*, případně *modprobe* s parametrem "-r". *Modprobe* narozdíl od *insmod* řeší i závilosti, proto může kromě požadovaného modulu načíst i moduly, na kterých závisí. Pro vypsání aktuálně natažených modulů lze použít aplikaci *lsmod*.

Kromě maker pro určení inicializační a ukončovací funkce je poskytováno Linuxovým jádrem mnoho dalších. Mezi nimi například makra pro krátký popis modulu `MODULE_DESCRIPTION`, jméno autora modulu `MODULE_AUTHOR` nebo například verzi modulu `MODULE_VERSION` [15, s. 30]. Důležitím makrem je také určení licence modulu `MODULE_LICENSE`, neboť tomuto modulu budou poskytnuty funkce API jádra kompatibilní licence (tyto poskytované funkce jsou označené makry `EXPORT_SYMBOL`).

3.2.3 Ovladače zařízení

Jádro poskytuje také API pro ovladače zařízení, například hlavičkový soubor `device.h` poskytuje speciální makro:

```
module_driver(__driver, __register, __unregister, ...)
```

které nastaví inicializační (`__register()`) a ukončovací (`__unregister()`) funkce s využitím maker `module_init` a `module_exit`. Oběma funkcím bude předáno `__driver` jako parametr.

Toto makro je dále použito v hlavičkovém souboru `platform_device.h`, které poskytuje makro:

```
module_platform_driver(__platform_driver)
```

Toto makro nedělá nic jiného, než že použije makro `module_driver` s parametrem `__platform_driver`. Zde je ale vyžadováno, aby tento parametr byl speciálního typu, konkrétně struktury `platform_driver`. Tato struktura totiž obsahuje (mimo jiné) ukazatele na funkce *probe()* a *remove()* a strukturu `device_driver`, pomocí které lze definovat položku `compatible`. Jádro totiž potřebuje vědět, pro které zařízení má použít jaký ovladač. Jak bylo popsáno v sekci 3.1.1, jádro při prohledávání DeviceTree hledá odpovídající ovladače,

které bude možné zavést do systému. Jednou z takových možností, jak spárovat ovladač a periférii definovanou v DeviceTree jsou právě položky `compatible`. Při nalezení odpovídajícího ovladače je zavolána funkce *probe()* příslušného ovladače.

3.2.4 Znaková zařízení

Na uživatelskou aplikaci může být kladen požadavek, kdy bude potřebovat komunikovat s jaderným modulem (například user space ovladače). Ale jak bylo popsáno v sekci 3.2.1, uživatelská a jaderná aplikace pracují s jiným adresovým prostorem. Pro takové účely lze používat například blokové zařízení (block devices) nebo znakové zařízení (char devices), což jsou speciální soubory spravované některým jaderným modulem. Takový soubor lze rozpoznat například z prvního sloupce výpisu příkazu *ls -s*, kde písmeno "b"značí blokové a "c"znakové zařízení. Tyto soubory se nachází zpravidla v adresáři */dev*.

Každé zařízení má přidělenou dvojici čísel MAJOR a MINOR. MAJOR číslo značí ovladač, který dané znakové zařízení spravuje a MINOR označuje konkrétní zařízení. Tyto čísla lze zobrazit opět příkazem *ls -l* [15, s. 44].

Ovladač znakového zařízení si při registraci daného zařízení registruje operace, které bude obsluhovat. Mezi tyto operace patří [15, s. 50]:

- **open()** – funkce volaná při pokusu o otevření znakového zařízení (systémové volání *open*). Ovladač se může rozhodnout požadavek na otevření zamítnout pomocí návratové hodnoty.
- **release()** – funkce volaná při zavření souborového popisovače (systémové volání *close*). Jádro se postará o to, aby tato funkce byla volána až při posledním volání *close* – například v důsledku forknutí procesu [15, s. 60].
- **read()** – odpovídá systémovému volání *read*. Jedním z argumentů je ukazatel na paměť v uživatelském prostoru, kam mají být nakopírována požadovaná data. Jelikož ale ovladač pracuje v jaderném adresovém prostoru, je nutné provézt kopii dat mezi těmito adresovými prostory pomocí funkce *copy_to_user*.
- **write()** – slouží pro přenos dat z uživatelské aplikace do ovladače (systémové volání *write*). Obdobně jako operace *read*, i tato operace má ukazatel na paměť v uživatelském prostoru. Tentokrát je ale tato paměť určena pro zápis dat do zařízení, proto se použije funkce *copy_from_user*.
- **ioctl()** – slouží pro zadávání příkazů ovladači. Každý ovladač si může specifikovat vlastní příkazy, kterými je pak uživatelskou aplikací řízen.

Uživatelské aplikace tak zápisem, čtením nebo jinými operacemi s těmito znakovými soubory mohou komunikovat s konkrétním jaderným modulem, který má pro takováto volání registrovány své funkce.

Kapitola 4

Platforma ADSP-SC58x

Jednou z rodin nízkopříkonových čipů jsou SoC platformy ADSP-SC58x. Jedná se o procesory firmy Analog Devices s integrovanými CPU a DSP jádry do jednoho čipu, konkrétně ARM a SHARC. V této kapitole je popsán hardware rodiny procesorů ADSP-SC58x, nabízených vestavěných kitů, podporované periférie a bootovací proces.

4.1 Procesory ADSP-SC58x

ADSP-SC58x je rodina procesorů od firmy Analog Devices obsahující 1 či 2 jádra SHARC a jádro ARM Cortex-A5. Dokumentace Analog Devices označuje (čísluje) tato jádra následovně:

- Core 0 (ARM)
- Core 1 (SHARC 1)
- Core 2 (SHARC 2)

Procesory ARM Cortex-A5 jsou 32bitové procesory podporující instrukční sadu ARMv7 a technologie jako NEON pro akceleraci obrazu/audia, Jazelle pro hardwarové vykonávání Java bajtkódu nebo podporu FPU[1]. ARM jádro v procesorech ADSP-SC58x pracuje na maximálním kmitočtu 450 MHz a má k dispozici datovou a instrukční L1 cache o velikosti 32 KiB a 256 KiB L2 cache [10].

SHARC jsou DSP procesory optimalizované pro práci s audio a FP daty[2]. Tyto procesory obsahují DAI[3], vícenásobné interní sběrnice pro omezení úzkého hrdla propustnosti a vestavěnou paměť SRAM. Různé ADSP-SC58x procesory mají různou konfiguraci SHARC jader. Jejich rozdíly zobrazuje tabulka[4] 4.1 [10].

Procesory ADSP-SC58x také podporují různou škálu periférií a sběrnic. Rozdíly jsou zobrazeny v tabulce 4.2.

4.1.1 L1 SRAM

Každé procesorové jádro SHARC obsahuje vlastní, různě velikou SRAM paměť (porovnání velikostí v různých ADSP-SC58x procesorech je v tabulce 4.1). SHARC tuto paměť může

[1] Floating-point unit – jednotka vykonávající operace s čísly s plovoucí řádovou čárkou
[2] Floating-point. SHARC procesory umí pracovat v režimech 32, 40 i 64 bitů
[3] Digital audio interface
[4] pro velkou šířku tabulky byly názvy procesorů ADSP-SC58x zkráceny na SC58x

SHARC	SC582	SC583	SC584	SC587	SC589
max. frekvence	450 MHz	450 MHz	450 MHz	450 MHz	450 MHz
počet jader	1	2	2	2	2
L1 SRAM / jádro	640 KiB	384 KiB	640 KiB	640 KiB	640 KiB

Tabulka 4.1: Porovnání vlastností jednotlivých SHARC procesorů

Rozhraní	SC582	SC583	SC584	SC587	SC589
PCIe 2.0	–	–	–	–	1×
GPIO pinů	80	80	80	102	102
USB 2.0 (host/device/OTG)	1×	1×	1×	1×	1×
USB 2.0 (host/device)	–	–	–	1×	1×
Ethernet 10/100/1000 Mb	1×	1×	1×	1×	1×
Ethernet 10/100 Mb	–	–	–	1×	1×
Pouzdro	349-BGA	349-BGA	349-BGA	529-BGA	529-BGA

Tabulka 4.2: Porovnání vlastností ADSP-SC58x procesorů z hlediska podpory periférií a sběrnic

použít jak pro data, tak pro kód, přičemž doba přístupu SHARC jádra k této paměti trvá jediný cyklus. Celá SRAM paměť SHARC jádra je mapována dvakrát – jednou do soukromého adresového prostoru, kam má přístup pouze vlastní SHARC jádro, a podruhé do multiprocesorového adresového prostoru, odkud je přístupná pro druhé SHARC jádro a také pro ARM [10] [1].

Z tabulky 4.3 lze vidět rozložení této SRAM paměti. SRAM paměť jednotlivých jader ADSP-SC589 procesoru netvoří jeden celistvý paměťový blok velikosti 640 KiB, ale je rozdělena do 4 paměťových bloků – první dva bloky o velikosti 192 KiB a další dva o velikosti 128 KiB.

Každý SHARC procesor je se svojí SRAM pamětí propojen dvěma slave porty, přičemž pomocí obou portů lze do jednotlivých paměťových bloků bezkonfliktně přistupovat jak přímo z jádra, tak pomocí DMA. Adresy jednotlivých portů jsou rovněž zaneseny v tabulce 4.3.

4.2 Vývojový kit ADZS-SC589 EZ-KIT

Společnost Analog Devices nabízí pro své procesory ADSP-SC58x dva vývojové kity:

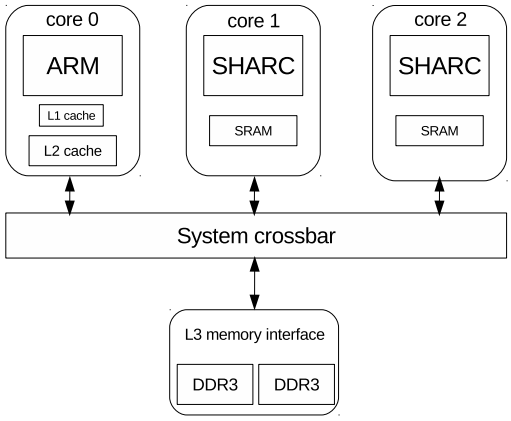
- **ADZS-SC584** – vývojový kit pro procesory ADSP-SC582/3/4
- **ADZS-SC589** – vývojový kit pro procesory ADSP-SC587/9

Zde bude popsán vývojový kit ADZS-SC589 s čipem ADSP-SC589, který obsahuje oproti ADZS-SC584 více periférií a větší DDR3 paměť. Tento vývojový kit obsahuje (nejedná se o úplný výčet):

- čip ADSP-SC589 s jedním ARM jádrem a dvěma SHARC jádry
- 512 MiB paměti DDR3

jádro	port	blok paměti	začátek	konec
SHARC1	Slave 1	Blok 0	0x28240000	0x28270000
		Blok 1	0x282C0000	0x282F0000
		Blok 2	0x28300000	0x28320000
		Blok 3	0x28380000	0x283A0000
	Slave 2	Blok 0	0x28640000	0x28670000
		Blok 1	0x286C0000	0x286F0000
		Blok 2	0x28700000	0x28720000
		Blok 3	0x28780000	0x287A0000
SHARC2	Slave 1	Blok 0	0x28A40000	0x28A70000
		Blok 1	0x28AC0000	0x28AF0000
		Blok 2	0x28B00000	0x28B20000
		Blok 3	0x28B80000	0x28BA0000
	Slave 2	Blok 0	0x28E40000	0x28E70000
		Blok 1	0x28EC0000	0x28EF0000
		Blok 2	0x28F00000	0x28F20000
		Blok 3	0x28F80000	0x28FA0000

Tabulka 4.3: Multiprocesorový adresový prostor SRAM paměťových bloků SHARC jader (ADSP-SC589).



Obrázek 4.1: Schéma úrovní pamětí pro ARM a SHARC jádra

- 16 MiB SPI Flash paměti
- 1× Ethernet 10/100
- 1× Ethernet 10/100/1000
- 2× Micro-USB 2.0 (1× je varianta OTG)
- UART simulované skrze Micro-USB
- PCIe
- SD/MMC konektor pro paměťové karty

- AD/DA převodníky s celkem 12 RCA (cinch) konektory – 8 výstupních a 4 konfigurovatelné jako vstup nebo výstup

Pro převod analogových signálů na číslicové je na desce integrován obvod ADAU1979, který obsahuje 4 konfigurovatelné A/D převodníky připojené ke 4 vstupním RCA konektorům. Dále je přítomen obvod ADAU1962A, který poskytuje převod 12 digitálních signálů na 12 analogových výstupů. Tento převodník používá ve výchozím stavu 8 RCA konektorů, lze ale převodník nakonfigurovat i pro využití dalších 4 konektorů, které jsou společné s obvodem ADAU1979. Celkem je tedy možné využít až 12 analogových výstupů.

Oba převodníky (DA, AD) lze konfigurovat jak pomocí SPI rozhraní, tak přes I2C.

Vývojový kit obsahuje také několik tlačítek a přepínačů [5, s. 35]:

- **SW1** – polohou přepínače se nastavuje, z jaké periférie bude procesor bootovat:
  - 0 – Bez bootování
  - 1 – SPI2 master boot
  - 2 – SPI2 slave boot
  - 6 – LP0 slave boot
  - 7 – UART0 slave boot
- **SW2** – tlačítko pro resetování procesoru a některých periférií
- **SW3/4** – GPIO tlačítka
- **SW6/7** – přepínání mezi konfiguracemi pro různé počty procesorů (vývojových kitů) a nastavení přípojných emulátorů

K vývojovému kitu je rovněž možné připojit emulátory pro ladění ICE-1000 a ICE-2000. Tyto emulátory lze propojit s hostitelským počítačem pomocí rozhraní USB a ladit tak přes emulovaný JTAG[5] běh procesorů ARM a SHARC nebo přistupovat k SPI Flash paměti.

4.3 System Protection Unit (SPU)

SPU jednotka poskytuje prostředky pro ochranu systémových zdrojů před nežádoucím přepsáním/přístupem. SPU jednotka pracuje neustále a nelze ji vypnout. Tato jednotka má přiděleno jediné přerušení číslo 241 [6, s. 189, 193].

SPU poskytuje pro periférie tyto ochrany:

- **Globální zámek** – řídicí registry některých periférií lze chránit před zápisem. Takové registry obsahují speciální bit, jehož nastavením se povolí zámek pro daný registr. Dále SPU obsahuje globální zámek vztahující se na všechny takové registry. Do registru nelze zapisovat, jeli povolen globální zámek SPU a zároveň zámek konkrétního registru.
- **Ochrana periférií před mastery** – nastavením SPU lze zakázat přístup Core Masterů (ARM a SHARC jádra) nebo System Masterů (PCIE, DBG, ETR a Memory DMA) ke konkrétním periferiím.

[5] Joint Test Action Group

- **Ochrana periférií před nezabezpečenými přístupy** – pro každou periférii lze nastavit, zdali do jejích registrů může být zapisováno pouze zabezpečenými přístupy nebo i nezabezpečenými.

Všechny tyto mechanismy jsou aplikovatelné i na samotné SPU, přičemž platí, že pokud se nastavením zakáže zápis do odpovídajících SPU registrů, pak takové nastavení nemůže být až do systémového resetu měněno.

Registry

Následuje popis registrů SPU jednotky:

- **SPU_CTL** (Control Register) – konfigurací tohoto registru lze povolit globální zámek (zápisem hodnoty 0xAD) a zámek samotného registru. V případě, že jsou oba zámky povoleny, tak až do resetu není další manipulace s tímto registrem možná. Dále lze nastavit, aby v případě nepovoleného pokusu (vzhledem k aktuální konfiguraci SPU) o zápis do registru periférie bylo vyvoláno přerušení z SPU (ID 241).
- **SPU_SECURECHK** (Secure Check Register) – čtením z tohoto registru lze v závilosti na vyčtené hodnotě určit, zdali se z registru četlo zabezpečeným přístupem (hodnota 0xFFFFFFFF) nebo nezabezpečeným (0x00000000).
- **SPU_SECURECTL** (Secure Control Register) – umožňuje (v kombinaci s globálním zámkem) zakázat zápis do SPU_SECUREC a SPU_SECUREP registrů všech periférií, dále vyvolání přerušení v případě porušení přístupu k registru z hlediska zabezpečení. Pomocí tohoto registru lze také vynulovat MSEC nebo SSEC bity (registr SPU_SECUREP) všech periférií.
- **SPU_SECUREC[n]** (Secure Core Registers) – existuje pro každé SHARC jádro. V registru lze nastavit jediný bit, který určuje, zdali může být s L1 pamětí daného jádra manipulováno i pomocí nezabezpečených přístupů.
- **SPU_SECUREP[n]** (Secure Peripheral Register) – existuje pro každou periférii. Lze modifikovat pouze dva bity:
  - MSEC: daná periférie bude generovat zabezpečené/nezabezpečené přístupy.
  - SSEC: daná periférie bude (ne)chráněna před nezabezpečenými přístupy.
- **SPU_STAT** (Status Register) – stavový registr, z kterého lze vyčíst informace o chybách nebo stav globálního zámku.
- **SPU_WP[n]** (Write Protect Register n) – každá periferie má tento registr. Nastavením bitů tohoto registru lze zakázat přístup jednotlivých Core a System Masterů k paměťově mapovaným registrům konkrétní periférie. Tento registr existuje i pro SPU – pokud se všem masterům zakáže přístup, tak nastavení SPU nemůže být až do resetu změněno.

4.4 Správa přerušení

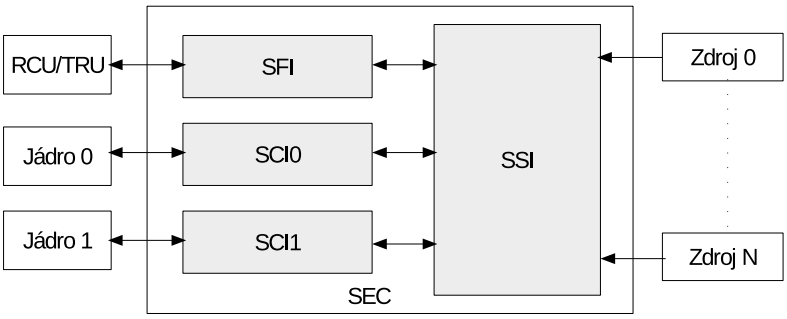
Čipy ADSP-SC58x disponují dvěmi řadiči přerušení: GIC[6] pro ARM jádro a SEC[7] pro SHARC jádra.

4.4.1 System Event Controller (SEC)

Řadič SEC má kromě přerušení pro SHARC jádra také na starosti veškerou správu událostí (mezi ně patří také přerušení a různá selhání systému), včetně konfigurace všech zdrojů událostí a nastavení priorit [6, s. 239].

Řadič SEC je tvořen několika bloky, jak je znázorněno na obrázku 4.2:

- **SEC source interface (SSI)** – řízení zdrojů událostí. Každý zdroj události obsahuje několik registrů pro konfiguraci (například nastavení priority zdroje události) a zjištění stavu události. Zdrojem události může být například jednotka DMA. Každý zdroj má svoje ID, v literaruře označované jako SID [6, s. 258].
- **SEC core interface (SCI)** – rozhraní pro propojení událostí mezi SEC a SHARC jádry. Rovněž řeší priority událostí určených pro tato jádra.
- **SEC Fault Interface (SFI)** – rozhraní propojující chybové události se zbytkem systému.



Obrázek 4.2: Blokové Schéma SEC

Tento řadič přerušení může být zakázán (narozdíl od SPU, které vypnout nejde). Pro správnou funkci musí být tedy SEC nejprve povoleno pomocí globálního řídícího SEC registru. SEC dovoluje zpracovávat přerušení dvěma způsoby [6, s. 264]:

- pro přerušení SHARC jádra skrze rozhraní SCI
- pro vyvolání chyby skrze rozhraní SFI

Pro první možnost, tedy vyvolání přerušení v jádře procesoru, je třeba nakonfigurovat požadovaný zdroj události tak, aby při jejím výskytu bylo vyvoláno přerušení. Také je třeba nastavit, kterému jádru má být přerušení určeno. Dále je potřeba povolit přerušení pro zvolené jádro. Toto vše lze nastavit skrze SEC registry, které jsou mapované v paměťovém prostoru. Přerušení může být rovněž vyvoláno softwarově.

[6] Generic Interrupt Controller
[7] System Event Controller

Nastavení přerušení jako chybové události je provedeno podobným způsobem. Rovněž je třeba nakonfigurovat požadovaný zdroj události, tentokrát ale jako zdroj selhání. Dále je potřeba nakonfigurovat chování při vyvolání takového přerušení – například pro reset systému.

Událost vyvolaná v některém z nakonfigurovaných zdrojů je zachycena SSI rozhraním, které na základě nastavení zdroje přesměruje přerušení do určitého SCI (pokud je zdroj nastaven jako přerušení pro konkrétní jádro), případně SFI (pokud je zdroj nastaven jako selhání). SEC dále vybere přerušení s nejvyšší prioritou (pokud má více aktivních přerušení stejnou prioritu, vybere se zdroj s nejnižším SID). Jakmile jádro skončí s obsluhou přerušení, zapíše do registru SEC_END SID zpracovaného přerušení.

SEC rovněž podporuje seskupování zdrojů přerušení pomocí nastavení příslušných bitů v řidícím registru každého zdroje.

Všechna dostupná přerušení a spouštěče pro SEC jsou v tabulce 4.4 [6, s. 241].

Název	ID přerušení	Název	ID spouštěče
SEC0_ERR	0	SEC0_FAULT	92

Tabulka 4.4: Seznam přerušení (SEC) a spouštěčů (TRU) pro SEC

4.4.2 Generic Interrupt Controller (GIC)

Řadič přerušení GIC pro ARM jádro umí obsluhovat všechna přerušení ze SEC zdrojů (254 SPI[8] přerušení). Mimo to podporuje ještě softwarová přerušení (celkem 8 SGI[9] přerušení). Jednotlivá přerušení mohou být navíc konfigurována jako normální, nebo zabezpečená.

Logika GIC je rozdělena do dvou bloků. Každý tento blok disponuje vlastními registry [6, s. 292]:

- **GICPORT0** (distribuční blok) – umožňuje konfiguraci jednotlivých přerušení (povolení/zakázání přerušení, nastavení priorit, citlivost na hranu nebo úroveň), přeposlat SIG do jádra procesoru a poskytuje mechanismus pro softwarovou změnu stavu čekání jednotlivých přerušení.
- **GICPORT1** (rozhraní pro ARM) – tento blok umožňuje maskovat přerušení, potvrzovat přerušení a definovat preemptivní chování procesoru ohledně přerušení.

4.5 Trigger Routing Unit (TRU)

Procesor ADSP-SC589 obsahuje jednu Jednotku TRU[10] umožňující synchronizaci nebo řízení toku událostí, aniž by procesor musel zasahovat. Existují dva druhy triggerů [6, s. 309]:

- **Master trigger** – označuje zdroj triggeru, tedy periférie generující událost spouštěče (trigger). Master triggerem může být kromě periférie také softwarový trigger daný TRU_MTR registrem.

[8] Shared Peripheral Interrupt
[9] Software Generated Interrupt
[10] Trigger Routing Unit

- **Slave trigger** – cíl triggeru, tedy periférie, která obdrží událost od spouštěče.

.

Pro správnou funkčnost spouštěčů musí být nakonfigurován trigger master i slave. Některé periférie mohou být konfigurovány jako master i jako slave, jiné pouze jako master nebo pouze jako slave. O propojení mezi zdrojem (master) a cílem (slave) se stará právě jednotka TRU.

TRU může být využita pro softwarové řízení toku (viz sekce 4.5.2) nebo například pro synchronizaci DMA kanálů (sekce 4.6.5). Pro tyto účely může být využito vlastnosti TRU, kdy master:slave triggery nemusí být mapovány 1:1, ale na jeden trigger master může připadat více slavů.

TRU jednotka má k dispozici celkem 10 přerušení (tabulka 4.5) [6, s. 254, 258], kterým odpovídá 10 slave triggerů (tabulka 4.6) [6, s. 248, 249], přičemž platí, že aktivace jednoho z těchto triggerů vyvolá odpovídající přerušení.

Jméno	Přerušení	Jádro	Jméno	Přerušení	Jádro
TRU0_SLV0	248	0	TRU0_SLV6	138	1
TRU0_SLV1	249	0	TRU0_SLV7	139	1
TRU0_SLV2	250	0	TRU0_SLV8	140	2
TRU0_SLV3	251	0	TRU0_SLV9	141	2
TRU0_SLV4	136	1	TRU0_SLV10	142	2
TRU0_SLV5	137	1	TRU0_SLV11	143	2

Tabulka 4.5: Seznam všech přerušení jednotky TRU

Název	Slave TRU	Název	Slave TRU
TRU0_SLV0	84	TRU0_SLV6	90
TRU0_SLV1	85	TRU0_SLV7	91
TRU0_SLV2	86	TRU0_SLV8	92
TRU0_SLV3	87	TRU0_SLV9	93
TRU0_SLV4	88	TRU0_SLV10	94
TRU0_SLV5	89	TRU0_SLV11	95

Tabulka 4.6: Seznam všech slave triggerů jednotky TRU (TRU0Interrupt Request)

4.5.1 Registry

TRU obsahuje celkem 5 registrů (z toho jeden je dostupný vícekrát). Zde je jejich popis [6, s. 310]:

- **TRU_ERRADDR** (Error Address Register) – obsahuje adresu prvního registru, který způsobil chybu přístupu do dalších TRU registrů. Tato chyba mohla nastat buď snahou o zápis do TRU_MTR nebo TRU_SSR[n] registrů, pokud tyto registry byly chráněny proti zápisu pomocí zámku (více o této problematice je popsáno v sekci 4.3), nebo pokud je přistupováno na nevalidní adresu. Pokud nastane více chyb, tak v tomto registru bude uložena pouze adresa prvního registru, u kterého byla chyba

detekována. Pro resetování TRU_ERRADDR registru je třeba vynulovat příslušné bity v TRU_STAT registru.

- **TRU_GCTL** (Global Control Register) – řídicí registr. Obsahuje zámky pro tento a TRU_MTR registr, dále umožňuje vypnout/zapnout nebo resetovat TRU jednotku.
- **TRU_MTR** (Master Trigger Register) – umožňuje softwarové generování (master) triggerů. Je rozdělen na 4 sekce po 8 bitech, přičemž do každé sekce může být zápisem trigger master ID spuštěn trigger master. Tento registr je možné zamknout pomocí řídicího registru. O využití tohoto registru pro generování triggerů pojednává sekce 4.5.2.
- **TRU_SSR[n]** (Slave Select Register) – jediný registr, který se v TRU vyskytuje vícekrát. Každý slave trigger má svůj vlastní TRU_SSR registr (seznam TRU slave registrů je v tabulce 4.6). Kromě bitu pro zamknutí registru je zde také 8 bitové pole sloužící pro spárování tohoto slave triggeru s libovolným master triggerem – to se provede zápisem master trigger ID.
- **TRU_STAT** (Status Information Register) – tento status registr má definované pouze dva bity. Jeden indikuje chybu při špatném použití adresy TRU registrů, další indikuje pokus o zápis do TRU registru chráněného zámkem. Adresa registru, která zapříčinila tuto chybu, je uložena v registru TRU_ERRADDR (resetování tohoto registru se provede vynulováním obou bitů v TRU_STAT registru).

4.5.2 Synchronizace pomocí softwarových triggerů

Triggery je možné využít jako řešení mnoha synchronizačních problémů. Jedním z nich je synchronizace procesorů, přičemž pro tuto úlohu využijeme kromě triggerů také mechanismu přerušení (přerušení bylo popsáno v sekci 4.4). Jak je vidět v tabulce 4.5, přerušení TRU jednotky pro ADSC-589 jsou rozděleny do tří skupin – každé jádro má k dispozici čtyři přerušení. Pro vyvolání některého z těchto přerušení lze použít TRU slave triggery z tabulky 4.6. Například pro vyvolání přerušení na jádře 0 (ARM) lze použít přerušení TRU0_SLV0 a jemu odpovídající slave trigger stejného názvu. Jako master trigger je vhodné zvolit některý softwarový trigger (například SOFT0_MST) z tabulky 4.7 [6, s. 244, 245], neboť ostatní triggery jsou spárované s některou periférií a můžou tak být využívány pro jiné účely.

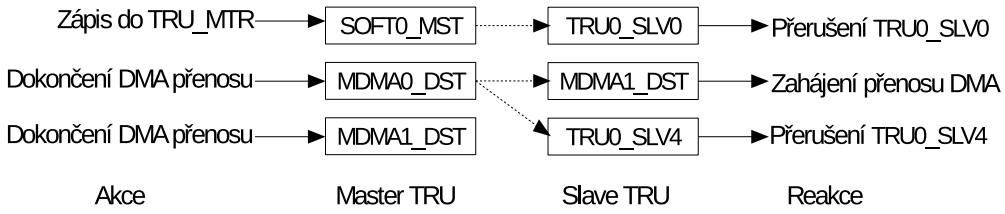
Název	Master TRU	Název	Master TRU
SOFT0_MST	93	SOFT3_MST	96
SOFT1_MST	94	SOFT4_MST	97
SOFT2_MST	95	SOFT5_MST	98

Tabulka 4.7: Seznam softwarově řízených trigger masterů (Software-driven Trigger)

Pro softwarové vyvolání přerušení na ARM jádře je třeba učinit následující kroky (tok událostí je znázorněn na ilustraci 4.3:

- alokovat si přerušení TRU0_SLV0 na ARM jádře
- povolit TRU v registru TRU_GCTL

- do TRU_SSR registru slave triggeru (TRU0_SLV0) uložit ID softwarového master triggeru (SOFT0_MST) a tím spárovat master se slavem
- zapsat do TRU_MTR registru ID softwarového master triggeru (SOFT0_MST) a tím skrze TRU a SEC vyvolat přerušení TRU0_SLV0



Obrázek 4.3: Řízení událostí pomocí TRU. Propojení master a slave triggerů je konfigurováno skrze TRU_SSR registry.

4.6 Direct Memory Access (DMA)

DMA jednotka disponuje několika kanály, které mohou přenášet data mezi pamětí a periférií/pamětí. Přenášené buffery mohou být konfigurovány jako 1D nebo 2D pole, přičemž lze nastavit počet řádků/sloupců, velikost jednoho přenosu a zarovnání řádků. DMA jednotka rovněž může vyvolat přerušení a informovat tak o dokončení přenosu nebo chybách.

Dostupnou DMA jednotku je možné konfigurovat dvěmi různými způsoby, které budou popsány v následujících sekcích: konfigurace pomocí zápisů do registrů a konfigurace pomocí deskriptorů v paměti.

4.6.1 DMA kanály

Pro přenosy je dostupných celkem 45 kanálů, které lze rozdělit do dvou skupin.

- Peripheral DMA (PDMA)
- Memory DMA (MDMA)

PDMA

PDMA kanály slouží pro přenos mezi periférií a pamětí. Periférní rozhraní podporuje následující šířky slova: 8b, 16b, 32b. Seznam všech PDMA kanálů zobrazuje tabulka 4.8 [6, s. 720, 722].

MDMA

MDMA kanály jsou určené pro přenosy dat z jedné části paměti do jiné. Paměťové rozhraní podporuje následující šířky slova: 8b, 16b, 32b, 64b, 128b. Tyto kanály jsou rozděleny do streamů, kdy každý stream obsahuje dva kanály – zdroj a cíl. Pro jeden MDMA přenos je nutno použít kanály ze stejného streamu – kanály mezi streamy nelze kombinovat. Seznam všech MDMA kanálů (8 kanálů, tedy celkem 4 streamy) zobrazuje tabulka 4.9 [6, s. 720, 722].

DMA ID	DMA kanál	DMA ID	DMA kanál
0	SPORT0_A_DMA	22	SPI0_TXDMA
1	SPORT0_B_DMA	23	SPI0_RXDMA
2	SPORT1_A_DMA	24	SPI1_TXDMA
3	SPORT1_B_DMA	25	SPI1_RXDMA
4	SPORT2_A_DMA	26	SPI2_TXDMA
5	SPORT2_B_DMA	27	SPI2_RXDMA
6	SPORT3_A_DMA	28	EPPI0_CH0_DMA
7	SPORT3_B_DMA	29	EPPI0_CH1_DMA
10	SPORT4_A_DMA	30	LP0_DMA
11	SPORT4_B_DMA	31	HAE0_TXDMA
12	SPORT5_A_DMA	32	HAE0_RXDMA_CH0
13	SPORT5_B_DMA	33	HAE0_RXDMA_CH1
14	SPORT6_A_DMA	34	UART1_TXDMA
15	SPORT6_B_DMA	35	UART1_RXDMA
16	SPORT7_A_DMA	36	LP1_DMA
17	SPORT8_B_DMA	37	UART2_TXDMA
20	UART0_TXDMA	38	UART2_RXDMA
21	UART0_RXDMA	41	FFTA0_TXDMA
		42	FFTA0_RXDMA

Tabulka 4.8: Seznam PDMA kanálů

DMA ID	DMA kanál	DMA ID	DMA kanál
8	MDMA0_SRC	39	MDMA2_SRC
9	MDMA0_DST	40	MDMA2_DST
18	MDMA1_SRC	43	MDMA3_SRC
19	MDMA1_DST	44	MDMA3_DST

Tabulka 4.9: Seznam MDMA kanálů

4.6.2 Registry

Do adresového prostoru procesoru je namapováno několik registrů, pomocí kterých lze nastavit datové přenosy. Zde je jejich popis [6, s. 764]:

- **DMA_ADDRSTART** – slouží k nastavení adresy přenášeného bufferu pro daný kanál. Podle toho, jestli je daný kanál nastaven jako zdroj nebo cíl, se bude jednat o zdrojový buffer nebo cílový buffer.
- **DMA_ADDR_CUR** – obsahuje aktuálního adresu bufferu. Na začátku přenosu obsahuje adresu zadanou přes registr DMA_ADDRSTART, v průběhu přenosu se hodnota mění v závislosti na přenášených datech.
- **DMA_CFG** – konfigurační registr. Pomocí tohoto registru lze nastavit parametry DMA přenosu, jako směr daného kanálu (zdroj/cíl), velikost přenášeného slova pro paměťové a periférní přenosy nebo například velikost struktury pro deskriptorovou

konfiguraci. Dále je zde možnost povolit vyvolání přerušení nebo triggeru po dokončení přenosu řádku/bloku dat.

- **DMA_STAT** – stavový registr obsahuje informace o šířce paměťové/periférní sběrnice a o stavu, případně chybách, DMA jednotky.

Následuje seznam registrů omezujících propustnost pro registrové přenosy:

- **DMA_BWLCNT** – tímto registrem lze omezit paměťové přenosy. Hodnota udává počet cyklů které musí uběhnout, než se zahájí další přenos.
- **DMA_BWLCNT_CUR** – obsahuje aktuální hodnotu omezení propustnosti. Na začátku obsahuje hodnotu DMA_BWLCNT a každým cyklem se dekrementuje. Po dosažení hodnoty 0 dojde k zahájení dalšího datového přenosu a hodnota registru se opět nastaví na hodnotu DMA_BWLCNT.
- **DMA_BWMCNT** – udává maximální počet cyklů daných pro jednotku k dokončení.
- **DMA_BWMCNT_CUR** – obsahuje počet zbývajících cyklů pro daný deskriptor pro dokončení. Hodnota se nastavuje na hodnotu z registru DMA_BWMCNT po každém zápisu do registru DMA_CFG nebo po dokončení práce jednotky.

Dále jsou přítomny registry pro nastavení deskriptorových přenosů:

- **DMA_DSCPTR_CUR** – obsahuje adresu příštího deskriptoru, který se načte. Před zahájením deskriptorového přenosu je třeba tento registr nastavit na adresu paměti s odpovídajícími deskriptory.
- **DMA_DSCPTR_NXT** – specifikuje adresu, odkud se načte další deskriptor. Pouze pro mód seznamu deskriptorů.
- **DMA_DSCPTR_PRV** – obsahuje ukazatel na předchozí deskriptor.

Následují registry popisující velikost a inkrement přenosů pro 1D a 2D přenosy.

- **DMA_XCNT** – počet prováděných přenosů. Pro 2D přenosy určuje hodnota registru velikost vnitřní smyčky.
- **DMA_XCNT_CUR** – počet zbývajících přenosů. Na počátku má tento registr hodnotu DMA_XCNT. V případě 2D přenosů se registr nastavuje na hodnotu z registru DMA_XCNT po každém přeneseném řádku.
- **DMA_XMOD** – určuje velikost inkrementu (v bajtech) registru DMA_ADDR_CUR po každém přenosu. Při práci s 2D polem je jediný rozdíl, že poslední inkrement není o velikost registru DMA_XMOD, ale DMA_YMOD.
- **DMA_YCNT** – počet přenášených řádků pro 2D přenosy (určuje tedy velikost vnější smyčky).
- **DMA_YCNT_CUR** – počet zbývajících řádků (pouze pro 2D přenosy).
- **DMA_YMOD** – určuje o jakou velikost je třeba inkrementovat registr DMA_ADDR_CUR po dekrementaci registru DMA_YCNT_CUR (určuje tedy zarovnání řádků pro 2D pole).

4.6.3 Register-based konfigurace

Jednou z možností konfigurace DMA přenosů je pomocí zápisu do dostupných namapovaných registrů. Před každým přenosem je tedy nutné ve všech potřebných registrech nastavit adresy, velikosti přenosů, inkrementy a směr kanálů. Konfiguraci skrze registry lze provézt dvěma módy [6, s. 744]:

- **Stop mód** – DMA přenos proběhne pouze jednou – po provedení nastavených přenosů se daný kanál zastaví.
- **Autobuffer mód** – v tomto módu probíhají DMA přenosy cyklicky – po provedení všech nastavených přenosů pro daný kanál se znovu nastaví počáteční adresy bufferu a přenos pokračuje dále.

4.6.4 Komunikace pomocí deskriptorů

Nevýhodou registrových konfigurací je, že před každým přenosem je nutné konfiguraci nahrát do příslušných registrů. Pro deskriptorovou konfiguraci se nastavení uloží do paměti, na kterou se DMA jednotce předá pouze ukazatel. V případě opakujících se DMA přenosů tak stačí nastavit konfiguraci pouze jednou. Pro zahájení přenosu se provede uložení adresy paměti s uloženou konfigurací do registru DMA_DSCPTR_CUR a následně se provede nastavení módu a povolení kanálu v registru DMA_CFG [6, s. 745].

Stejně jako registrová konfigurace, tak i deskriptorová může pracovat v různých módech:

- **Pole deskriptorů** – v tomto módu jsou deskriptory v paměti uloženy přímo za sebou – není tak třeba nastavovat adresu dalšího deskriptoru. Obsah deskriptorů spolu s odpovídajícími offsety je zobrazen v tabulce 4.10.
- **Seznam deskriptorů** – pro tento mód není vyžadováno, aby byly deskriptory uloženy přímo za sebou. Naproti tomu musí každý deskriptor obsahovat ukazatel na svého následovníka. Obsah deskriptorů spolu s odpovídajícími offsety je zobrazen v tabulce 4.11.

Offset	Registr
0x00	DMA_ADDRSTART
0x04	DMA_CFG
0x08	DMA_XCNT
0x0C	DMA_XMOD
0x10	DMA_YCNT
0x14	DMA_YMOD

Tabulka 4.10: Offsety a odpovídající registry pro pole deskriptorů.

4.6.5 Synchronizace MDMA přenosů

Detekovat konec DMA přenosu a provézt následně další akci lze několika způsoby – zde budou diskutovány tři různé varianty, které využívají detekce dokončení přenosu pomocí přerušení nebo triggerů. Obě tyto detekce mohou být nastaveny pro jednorozměrné i dvourozměrné přenosy.

Offset	Registr
0x00	DMA_DSCPTR_NXT
0x04	DMA_ADDRSTART
0x08	DMA_CFG
0x0C	DMA_XCNT
0x10	DMA_XMOD
0x14	DMA_YCNT
0x18	DMA_YMOD

Tabulka 4.11: Offsety a odpovídající registry pro seznam deskriptorů.

Polling

První možnost, jak zjistit stav dokončení DMA operace, je pomocí stavového registru. Ten má jeden bit pojmenovaný jako IRQDONE označující dokončení signalizovaného přenosu. Stačí tak cyklicky zjišťovat stav tohoto bitu a poté, co bude detekováno dokončení přenosu, tak jej zápisem jedničky vynulovat [6, s. 779].

Přerušení

Další možností je využít některé z MDMA přerušení, jejichž úplný výčet je v tabulce 4.12 [6, s. 255, 257]. Každý MDMA kanál obsahuje dva typy přerušení – jeden pro signalizaci chyby přenosu a druhý pro dokončení přenosu (takového, jenž byl právě nastaven pro vyvolání přerušení). Pro synchronizaci tedy stačí vhodně nastavit cílový MDMA kanál (jeho řídicí registr) na vyvolání přerušení a nastavit cílovou aplikaci pro odchycení tohoto přerušení.

Jméno	DMA přerušení		Jméno	DMA přerušení	
	Fin	Err		Fin	Err
MDMA0_SRC	172	217	MDMA2_SRC	168	221
MDMA0_DST	173	218	MDMA2_DST	169	222
MDMA1_SRC	174	219	MDMA3_SRC	166	223
MDMA1_DST	175	220	MDMA3_DST	167	224

Tabulka 4.12: Přerušení Memory DMA kanálů. Fin označuje číslo přerušení vyvolané po dokončení DMA přenosu, Err znamená číslo přerušení zapříčiněné chybou během DMA přenosu.

Triggery

Posledním popsaným mechanismem pro detekci dokončení přenosů budou triggery. Jelikož MDMA obsahuje kromě master triggerů i slave (viz tabulka 4.13) [6, s. 244, 249], tak je možné DMA přenosy nastavit tak, aby dokončení jednoho přenosu odstartovalo nový přenos. Pro tyto účely jsou v řídicím registru DMA kanálů položky pro nastavení čekání na vstupní trigger (slave) a pro spuštění výstupního triggeru (master) po dokončení přenosu řádku/bloku dat. Opět platí, že master a slave triggery je třeba spárovat pomocí TRU_SSR registru příslušného DMA kanálu (jak je popsáno v sekci 4.5).

Název	TRU		Název	TRU	
	Master	Slave		Master	Slave
MDMA0_SRC	74	102	MDMA2_SRC	78	106
MDMA0_DST	75	103	MDMA2_DST	79	107
MDMA1_SRC	76	104	MDMA3_SRC	80	108
MDMA1_DST	77	105	MDMA3_DST	81	109

Tabulka 4.13: Seznam MDMA slave a master triggerů

Ilustrace 4.3 zobrazuje situaci, kdy dokončení přenosu MDMA0 streamu zahájí přenos MDMA1 streamu. Dokončení MDMA1 streamu nevyvolá žádnou akci, jelikož žádný TRU_SSR registr nemá přiděleno master TRU tohoto streamu. Aktivaci DMA master triggeru musí být v daném DMA řídicím registru povolena, stejně tak čekání na DMA slave trigger.

4.7 Universal Asynchronous Receiver/Transmitter (UART)

Dostupný UART modul podporuje plně duplexní přenos, různé délky přenášeného slova a stop bitů a různou rychlost přenosu. ADSP-SC589 procesor má zabudované 3 UART moduly (UART0, UART1 a UART2), přičemž každý UART modul má přidělen vlastní DMA kanál (seznam těchto DMA kanálů je v tabulce 4.8). Pro UART0 navíc platí, že je na kitu připojen na FTDI převodník z USB na UART (tento čip podporuje mimo jiné také hardwarové i softwarové řízení toku) [5, s. 21]. UART modul má dále vlastní triggery (pro RX a TX DMA kanály), přerušení (opět RX a TX kanály, dále pak stavové přerušení) a registry [6, s. 1491].

Modul umí pracovat ve třech různých operačních módech:

- UART – komunikace pomocí UART standardu, pro který je možné konfigurovat následující vlastnosti:
  - 5-8 datových bitů
  - 1 start bit a 1 nebo 2 stop bity (pro 5bitové datové slovo ještě 1 1/2 stop bitu)
  - lichá, sudá a sticky parita, případně bez parity
- IrDA[11] – dalším podporovaným standardem je sériová komunikace odpovídající fyzické vrstvě známé jako IrDA SIR
- MDB (Multi-Drop Bus) – posledním podporovaným módem je komunikace v multidrop bus sítích. Tato komunikace obsahuje navíc adresový bit, pomocí něhož lze kromě dat (adresový bit je nastaven na hodnotu 0) posílat i adresy (adresový bit nastavený na logicku 1)

UART modul může přenášet data s využitím jádra nebo DMA jednotky (pro každý UART jsou dostupné bufferované RX a TX kanály). Pro přenos lze také zapnout automatickou nebo manuální hardwarovou kontrolu toku (hardware flow control), která využívá signály *UART_RTS* a *UART_CTS* pro řízení rychlosti přenosu dat.

[11] Infrared Data Association

4.7.1 Registry

UART poskytuje pro řízení a zjišťování stavu přenosu několik registrů [6, s. 1516-1540]:

- **Clock Rate Register** – obsahuje konfiguraci pro řízení hodin. Honota spodních 16 bitů určuje děličku (divisor) systémových hodin (SCLK) a nejvyšší bit (EDBO) určuje povolení další předděličky s hodnotou 16.
- **Control Register** – přes řídicí registr lze povolit UART modul, nastavit paritu, počet stop bitů nebo hardwarovou kontrolu toku.
- **Status Register** – z tohoto registru lze vyčíst o chybách přenosu, připravenosti dat, stav adresového bitu atd.
- **Scratch Register** – registr obsahující 8 bitů pro libovolné využití (nijak neovlivňuje chování UART modulu).
- celkem tři registry pro povolení, zakázání a zjištění stavu přerušení (přerušení pro RX/TX, zaplnění nebo vyprázdnění bufferů a jiné).
- nakonec jsou k dispozici registry pro čtení/zápis bufferu, čítače přenesených dat v každém směru a registr pro nastavení adresy v případě MDB módu.

4.7.2 Rychlost přenosu

Rychlost přenosu UART modulu je odvozena od systémových hodin (SCLK) a obsahu registru *Clock Rate Register*. Bitová rychlost se odvodí podle následujícího vztahu [6, s. 1497]:

$$BitRate = SCLK/16^{1-EDBO} \times divisor$$

kde *EDBO* a *divisor* jsou hodnoty z CRR registru.

CRR registr a tedy rychlost přenosu lze nastavit ručně, nebo lze použít automatickou detekci rychlosti. Pro tuto detekci se využít časovač, pomocí kterého lze změřit periodu mezi dvěmi logickými jedničkami. Při měření doby mezi start a stop bity datového slova se použijí data o hodnotě 0. Hodnota divisor se pak vypočítá [6, s. 1499]:

$$Divisor = perioda/16^{1-EDBO} \times sirka$$

kde *perioda* je změřená perioda časovačem a *sirka* je šířka (počet bitů) měřeného slova (včetně start bitu).

4.8 Bootování

Několik jader dvou různých typů, různé módy bootování a počet periférií, ze kterých lze bootovat, komplikuje bootovací proces na platformně ADSP-SC58x . V následujících sekcích bude popsán proces bootování od popisu bootovacího souboru, přes bootování jader, až po povolení jader.

4.8.1 Formát spustitelných souborů

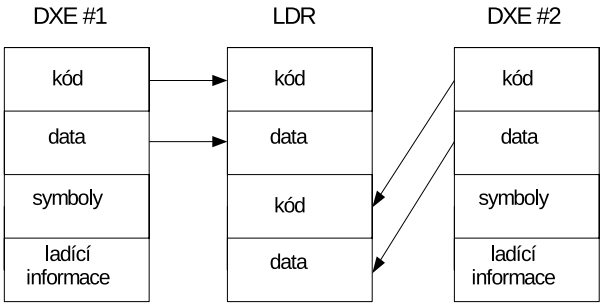
Součástí ABI[12] ARM architektury je i formát spustitelných souborů. ARM architektura, stejně jako mnoho dalších (AMD64, PowerPC, SPARC) podporuje standard ELF[13], používaný převážně Unixovými operačními systémy.

Formát z hlediska spustitelné aplikace (dá se na něj nahlížet i z hlediska linkování) obsahuje hlavičku ELF, programovou hlavičku a sekce. Sekce mohou být datové, kódové, ale například také sekce pro ladící informace [22].

Pro jádra SHARC jsou používány soubory ve formátu DXE, který je generován SHARC kompilátorem, a jehož formát je identický s formátem ELF [3, s. 2] [4, s. A-5].

4.8.2 Boot Stream

Boot API procesoru umí bootovat ze souborů formátu LDR (tedy nebootuje přímo ze souborů formátu DXE a ELF), což je kontejner obsahující spustitelné soubory pro jednotlivá jádra ARM a SHARC. LDR kontejner je, stejně jako DXE a ELF, rozdělen do několika sekcí. Tento soubor vznikne tak, že se spojí některé sekce z dílčích spustitelných souborů formátu DXE/ELF (lze použít jenom jedno z jader, ale také všechny tři – tedy ARM a dvě SHARC jádra). LDR ale narozdíl DXE/ELF neobsahuje sekce symbolů a ladicích informací. Obrázek 4.4 zobrazuje variantu LDR souboru a dvou DXE souborů, ze kterých byl vytvořen [4, s. 1-16].



Obrázek 4.4: Sekce dvou DXE souborů a výsledného LDR souboru

4.8.3 Bootování procesoru

ADSP-SC58x procesory podporují bootování z následujících periférií (nastavení příslušných přepínačů je popsáno v sekci 4.2 [8, s. 1]:

- SPI2 Flash master – procesor bootuje z flash paměti na adrese 0 v roli master.
- SPI2 Flash slave – bootování z externího master SPI zařízení.
- LP0 slave – bootování v roli master z externího link portu 0.
- UART0 slave – UART0 je připojen na FTDI čip (převodník mezi USB a UART). V tomto módu se UART pokusí detekovat rychlost přenosu. Jelikož se jedná o slave mód, tak UART čeká na první znak (v tomto případě znak @ (0x40), na kterém bude

[12] Application Binary Format
[13] Executable and Linkable Format

probíhat měření) – viz sekce 4.7.2. Poté, co bude detekována rychlost, odpoví UART slave čtyřmi bajty ve formátu [6, s. 3140]:

```
0xBF, UART_CLK[15:8], UART_CLK[7:0], 0x00
```

kde `UART_CLK` je registr UART modulu. 2. bajt tedy určuje děličku systémových hodin a 3. bajt případné povolení další předděličky o hodnotě 16, jak je popsáno v 4.7.1.

Poté, co jsou odeslány tato data je připraven UART slave přijmout LDR bootovací soubor.

Existují dva možné způsoby, jak bootovat procesor:

- Power-On Reset
- ROM API

Power-On Reset

Po resetu procesor načítá instrukce z Boot ROM (ROM paměť poskytující API, mimo jiné, pro bootování – nachází se přímo na čipu). Boot ROM procedura poté načte a nabootuje LDR soubor nacházející se na vybrané periférii (viz výše). Během této fáze bootování jsou nabootována všechna jádra, jejichž spustitelné soubory se nachází v LDR souboru. Platí tu ale omezení, že pouze ARM procesor (core 0) je uvolněn z resetu – uvolnění SHARC jader z resetu je nutné provézt z ARM jádra [6, s. 3113].

ROM API

ROM API je další možností, jak nabootovat procesor. Narozdíl od metody Power-On Reset jsou jádra bootována za běhu, přičemž ROM API je zavoláno některým z již běžících jader. Bootování jader je poskytováno voláním funkce adi_rom_Boot(), jejíž funkční prototyp je následující (pro úplný popis parametrů viz literatura [6, s. 3148]).

```
void *adi_rom_Boot(void *pBootStreamAddress,
    uint32_t dFlags,
    int32_t dBlockCount,
    ROM_BOOT_HOOK_FUNC *pCallHook,
    uint32_t dBootCommand);
```

s následujícím významem parametrů [6, s. 3148]:

- **pBootStreamAddress** – ukazatel na boot stream. Například pro SPI flash boot se jedná o adresu LDR souboru ve flash paměti.
- **dFlags** – příznaky pro bootování, jako například nastavení bootovacího módu jako slave, resetování jádra po návratu z funkce *adi_rom_Boot* atd.
- **dBlockCount** – počet bloků, které se mají načíst.
- **pCallHook** – funkce, která bude volána během bootování (může být NULL).
- **dBootCommand** – nastavení zdroje (SPI, UART, ...) a parametry pro tento zdroj (nastavení hodin pro UART, výběr zařízení pro SPI, ...)

Pro tuto metodu platí rovněž jistá omezení:

- Pokud je ROM API voláno jádrem ARM, tak tímto voláním mohou být bootována všechna tři jádra (core 0, core 1, core 2).
- Pokud je ROM API voláno jádrem SHARC 1, tak mohou být bootována pouze SHARC jádra (core 1, core 2).
- Pokud je ROM API voláno jádrem SHARC 2, tak může být voláno opět jenom toto jádro (core 2).

4.8.4 Uvolnění jádra z resetu

Uvolnění jádra z reset stavu je možné skrze registry RCU[14] jednotky. Uvolnění jádra n se provede následujícím postupem (pro detailní popis příslušných registrů viz [6, s. 3093]). V tomto postupu jsou pro jednoduchost ilustrace zanedbány čekání na potvrzení nastavení daných registrů:

1. Vynulování CR[n] bitu RCU_CRSTAT registru
2. Nastavení SI[n] bitu RCU_SIDIS registru pro zakázání systémových rozhraní
3. Nastavení CR[n] bitu RCU_CRCTL registru pro vstupu jádra do reset stavu
4. Nastavení SI[n] bitu registru RCU_SIDIS pro povolení systémových rozhraní
5. Vynulování CR[n] bitu RCU_CRCTL registru pro uvolnění jádra z reset stavu

[14] Reset Control Unit

Kapitola 5

Vývoj aplikací pro platformu ADSP-SC589

Pro platformu ADSP-SC589 existuje řada nástrojů pro vývoj aplikací pro SHARC a ARM jader. Zde bude diskutovány některé z nich, pro ARM architekturu konkrétně tvorba aplikací pro operační systém Linux včetně postupu jeho bootování.

5.1 Vývojové nástroje

Pro vývoj aplikací na čipy ADSP-SC589 poskytuje firma Analog Devices několik nástrojů pro operační systémy Windows i Linux. Zde budou popsány některé z nich.

5.1.1 CCES

CrossCore Embedded Studio je vývojové prostředí pro procerory Blackfin, SHARC a ARM vyvinuté firmou Analog Devices. Jedná se o licencované prostředí založené na projektu Eclipse. Toto prostředí je dostupné pro platformy Windows a Ubuntu (podporovaná verze 14.04). Součástí tohoto prostředí je dokumentace, různé aplikační knihovny (například MCAPI), debugger pro ladění ARM/SHARC aplikací přes emulátor ICE-1000/ICE-2000 nebo kompilátor pro ARM a SHARC jádra, přičemž kompilátor pro SHARC jádra je v současné verzi (2.3.0) dostupný pouze pro operační systémy Windows.

Spolu s CCES jsou instalovány některé pomocné nástroje. Jedním z takových nástrojů je `elfloader`, který vytváří z ELF/DXE souborů LDR kontejner. Tento kontejner může být vytvořen z jednoho až tří spustitelných souborů (jeden soubor odpovídá jednomu jádru), přičemž u všech spustitelných souborů, kromě posledního, je třeba pomocný nástroj informovat, že se nejedná o poslední soubor, a to parametrem `-NoFinalTag`. Dále je třeba definovat zdroj bootování parametrem "-b"(viz sekce 4.8.3). Možné varianty jsou: `SPI` (nebo `SPIMASTER`), `SPISLAVE`, `LPSLAVE`, `UARTSLAVE` [11, s. 159]. Následující příklad vytvoří LDR soubor `priklad.ldr` z ELF souboru (arm.elf) pro core0 a DXE souboru (sharc1.dxe) pro core1 [8, s. 5]:

```
elfloader.exe -proc ADSP-SC589 -si-revision 0.1 -b spimaster -f binary \
      -width 8 -bcode 1 -init init -core0=arm.elf -core1=sharc1.dxe \
      -NoFinalTag=arm.elf -o priklad.ldr
```

Dalším nástrojem je `cldp`[1]. Tento nástroj spustí program na procesoru, který přes debugovací rozhraní stáhne LDR soubor a nahraje jej do flash paměti [7, s. 3].

Následující příkaz nahraje soubor sharc0.ldr do flash paměti na offset 0x40000 přes debugger ICE-1000 a následně jej z této paměti přečte a porovná se zdrojovým souborem (pro programování flash paměti je použit ovladač `sc589_w25q128fv_dpia_Core1.dxe` dodávaný spolu s vývojovým prostředím):

```
cldp.exe -proc ADSP-SC589 -core 1 \
     -emu 1000 -driver sc589_w25q128fv_dpia_Core1.dxe \
     -cmd prog -erase affected -format bin -offset 0x40000 -file sharc0.ldr \
     -cmd compare -format bin -offset 0x40000 -file sharc0.ldr
```

5.1.2 Toolchain

Toolchain je kolekce vývojových nástrojů. Mezi nejznámější patří GNU Toolchain, který obsahuje mimo jiné tyto nástroje:

- kompilátor jazyka (gcc)
- assembler (as)
- linker (ld)
- disassembler a analyzátor objektových souborů (objdump)
- debugger (gdb)
- nástroj pro odstranění symbolů z objektových souborů (strip)

Tyto nástroje dle konfigurace podporují křížovou kompilaci (cross-compile), což znamená, že nástroj generuje kód jiné architektury, než ze které byl spuštěn. Toolchainy se od sebe mohou v mnoha věcech lišit, pro jejich odlišení se používá následující konvence [18, s. 3]:

`arch[-vendor][-os]-abi`

Význam je následující (včetně příkladů):

- arch – cílová architektura (arm, x86_64)
- vendor – poskytovatel toolchainu (linaro)
- os – operační systém (linux, none pro bare-metal[2])
- abi – aplikačně binární rozhraní (eabi, gnueabi)

Například arm-linux-gnueabi-gcc značí GCC kompilátor vytvářející spustitelné soubory pro ARM architekturu s operačním systémem Linux s GNU EABI rozhraním.

Následuje popis některých Linux Toolchainů pro vývoj Linuxových nebo bare-metal aplikací.

[1] Command-Line Device Programmer

[2] aplikace běžící přímo na cílovém hardwaru, bez operačního systému

CCES Toolchain

S CCES nástrojem je také dodáván toolchain pro vývoj bare-metal aplikací. Jeho nástroje jsou instalovány v adresáři `/opt/analog/cces/2.3.0/ARM/arm-none-eabi`. Tento toolchain má tedy podobu: `arm-none-eabi`. Kromě obvyklých nástrojů (gcc, strip, objdump, ld a jiných) obsahuje toolchain také nástroj OpenOCD[3]. Tento nástroj je určen pro ladění aplikací přes připojený JTAG nebo pro pro práci s pamětí (například flash paměť) na integrovaném čipu.

Linux-addin Toolchain

K CCES nástroji lze pro vývoj Linuxových aplikací doinstalovat balíček `Linux-Addin for ADSP-SC5xx` (dostupný pouze pro operační systém Linux). V tomto balíčku jsou obsaženy standardní vývojové nástroje, jenž mají podobu `arm-linux-gnueabi`. Balíček je instalován v adresáři `/opt/analog/cces-linux-add-in/1.0.1/ARM/arm-linux-gnueabi`.

GNU Toolchain

Pro vývoj aplikací pro platformu ARM se také používá GNU Toolchain. Tento svobodný software lze nainstalovat ve většině Linuxových distribucí (jmenná konvence se zde může trochu lišit).

Buildroot Toolchain

V případě, že je vytvářen Linuxový systém nástrojem Buildroot, tak lze využít i toolchain vygenerovaný tímto nástrojem. Takový toolchain je generován pro stejné ABI a platformu, pro jakou je vytvářen Linuxový systém. Toolchain má podobu `arm-buildroot-linux-uclibcgnueabihf` a je standardně vygenerován v podadresáři `host/usr/bin/` daného systému.

5.2 Podpora čipů ADSP-SC58x v open source projektech

Firma Analog Devices prostřednictvím balíčku `Linux-Addin for ADSP-SC5xx` přináší podporu čipů ADSP-SC58x do projektů Linux, Buildroot a U-Boot (podpora těchto čipů se zatím nestala součástí daných projektů).

U-Boot

U-Boot od Analog Devices je distribuován formou zdrojových kódů i binárního souboru ve formátu LDR. Jelikož je LDR formát využíván pouze čipy od firmy Analog Devices a tento formát není podporován v běžně dostupných nástrojích, je pro kompilaci U-Bootu nutné použít CCES/Linux Addin nástroje.

Buildroot

Buildroot, jako sada vývojových nástrojů, je rovněž distribuován formou zdrojových kódů. Jelikož se ale jedná o projekt spravující sadu distribuovaných nástrojů, programů a balíčků,

[3] Open On-Chip Debugger

lze poměrně snadno přenést potřebné konfigurace a balíčky pro podporu ADSP-SC58x platformy do oficiální verze Buildrootu (upstreamu). Stačí k tomu použít vytvořený defconfig soubor, případně přenést požadované balíčky s následnou úpravou konfigurace.

Linux

Operační systém dodávaný jako součást Buildroot projektu, kde jsou dostupné jak zdrojové kódy, tak zkompilovaná verze Linuxového jádra (včetě příslušného souborového systému a dtb souboru). Tato implementace vychází z verze 4.0.0 (viz bootovací výpis) a přidává podporu mimo jiné meziprocesorové komunikace ICC.

5.3 Zprovoznění Linuxu na platformě ADSP-SC589

Tato sekce se zabývá krátkým popisem možnostmi bootování Linuxu na jádře ARM, včetně nahrání U-Boot zavaděče na paměť flash.

5.3.1 Nahrání U-Bootu do paměti flash přes ICE-1000

Na kitu ADZS-SC589 není U-Boot ve výchozím stavu přítomen. Jeho instalaci je třeba provézt pomocí emulátoru ICE-1000 následnovně: je třeba připojit USB do ICE-1000 pro emulaci JTAG, dále napájení a v případě nahrávání z Linuxu další USB pro emulaci UART a ethernet. Ve Windows lze provést nahrání U-Bootu přes program *cldp*, v případě Linuxu je situace o něco složitější. Je nutné nahrát spustitelnou verzi U-Bootu do RAM paměti pomocí OpenOCD a GDB, poté jej spustit a pomocí sériového rozhraní (emulovaného přes USB) se připojit na běžící U-Boot. Nakonec se přes ethernet stáhne U-Boot v LDR kontejneru, který se pomocí spuštěného U-Bootu nahraje do flash paměti.

5.3.2 Bootování

V případě, že je U-Boot nainstalovaný na vybrané bootovací periférii, tak je při zapnutí automaticky spuštěn.

Spolu s U-Bootem mohou být bootovány i aplikace pro SHARC jádra. V sekci 4.8.3 byly popsány dva módy: Power-On Reset (SHARC aplikace musí být součástí stejného LDR souboru jako U-Boot) a zavoláním adi_rom_Boot() z ROM API.

Bootování SHARC jader

Nabootovaná SHARC jádra zůstávají v reset stavu. Uvolněním jader z reset stavu se provede postupem popsaným v 4.8.4. U-Boot, resp. Linux, od Analog Devices obsahují speciální aplikace k tomu určené: icc (spuštění: *icc enable <CORE_ID>*), resp. corecontrol (spuštění: *corecontrol –start <CORE_ID>*), kde parametr <CORE_ID> znamená:

- 0 – core 0 (ARM)
- 1 – core 1 (SHARC1)
- 2 – core 2 (SHARC2)

U-Boot: bootování Linuxu

Pomocí U-Bootu lze dále spouštět další aplikace, jako například Linux. V závilosti na konfiguraci U-Bootu je pro Linux generováno několik souborů, které je třeba nahrát do paměti RAM cílového zařízení (příklad adresy této paměti je uveden v závorkách):

- **uImage** – výsledný obraz ve formátu pro U-Boot (0xC2000000)
- **rootfs.cpio.uboot** – filesystém (0xC0000000)
- **sc58x-ezkit-hpc.dtb** – device tree pro kit SC589 (0xC4000000)

Tyto soubory lze do paměti RAM nahrát z několika periférií/rozhraní následujícími příkazy:

- **SD karta** – pro SD kartu je souborový systém uložen na kartě, proto jej netřeba předem nahrávat do paměti:
  ```
  ext2load mmc 0:1 0xc4000000 sc58x-ezkit-hpc.dtb
  ext2load mmc 0:1 0xc2000000 uImage
  ```
- **TFTP** – stažení přes ethernet
  ```
  tftp 0xC2000000 uImage
  tftp 0xC0000000 rootfs.cpio.uboot
  tftp 0xC4000000 sc58x-ezkit-hpc.dtb
  ```
- **Flash paměť** – načtení z flash paměti, kam byl soubor dříve uložen. Příkaz read je ve formátu: <RAM ADRESA> <FLASH OFFSET> <VELIKOST>. Nahrání potřebných souborů skrze flash paměť vypadá následovně (flash offset a velikost souboru se liší dle obsahu flash paměti a velikosti dotyčných souborů):
  ```
  sf probe 2:1
  sf read 0xc4000000 0x4f0000 0x5000
  sf read 0xC0000000 0xb00000 0x4e0000
  sf read 0xC2000000 0x500000 0x5e0000
  ```

Jakmile jsou soubory uloženy v paměti RAM, lze spustit bootování jádra následujícími příkazy:

```
set bootargs root=/dev/mtdblock2 rw rootfstype=jffs2 clkin_hz=(25000000) \
      console=ttySC0,57600 mem=224M
bootm 0xC2000000 0xC0000000 0xC4000000
```

Pro SD kartu, kde není souborový systém nahrát do paměti, je příkaz mírně odlišný:

```
bootm 0xC2000000 - 0xC4000000
```

Kapitola 6

Komunikace mezi ARM a SHARC procesory

Přenášení dat a synchronizace mezi ARM a SHARC aplikacemi je možné provádět několika způsoby. Tato kapitola popisuje již existují mechanismy, ale také návrh nových komunikačních principů pro platformu ADSP-SC589.

6.1 Existující mechanismy komunikace

Komunikace mezi jednotlivými jádry je možná na platformě ADSP-SC58x dvěma způsoby:

- využít MCAPI implementaci pro SHARC a ARM (pro operační systém Linux) jádra.
- komunikovat pomocí pamětí SRAM a DDR, přičemž data lze přenášet pomocí procesoru, nebo využít DMAC jednotku a přenášet data některými z MDMA kanálů.

6.1.1 Dostupná implementace MCAPI standardu

Společnost Analog Devices implementovala MCAPI pro SHARC i ARM jádra. Podpora SHARC jader je implementovaná knihovními funkcemi, které jsou distribuovány jako součást vývojového prostředí CCES.

Pro ARM procesory je dodávána implementace pro operační systém Linux skrze balíček *Linux-Addin for ADSP-SC5xx* pro vývojové prostředí CCES. MCAPI je implementováno jako user space knihovna, která využívá jaderný ovladač ICC[1] (s ovladačem komunikuje pomocí znakového zařízení */dev/icc*).

Jak bylo řečeno v sekci 2.4, tak velikost zprávy a paketu není definována standardem MCAPI, ale konkrétní implementací. Pro SHARC jádra byla maximální velikost zprávy i paketu nastavena na 256 B, pro ARM jádra (v rámci Linuxového ovladače) na 1024 B. Společnost Analog Devices dále určila, že jednomu jádru (MCAPI/ARM) odpovídá jeden MCAPI uzel. MCAPI knihovna pro Linux navíc neimplementuje plně standard MCAPI, a tak například blokující varianta přenosu zpráv zůstala neimplementována. Stejně tak nejsou implementovány čekací funkce *mcapi_wait_any* a *mcapi_wait*.

[1] Inter-Core Communication

Implementace MCAPI na platformě ADSP-SC589

V této sekci bude popsán princim implementace MCAPI standardu firmy Analog Devices na platformě ADSP-SC589. Zde popsané principy byly vyčteny přímo ze zdrojových kódů implementace MCAPI na platformách SHARC a ARM (Linux).

Pro implementaci MCAPI standardu byla využita paměť L2, jejíž využití je znázorněno v tabulce 6.1.

banka	Využití	Velikost	začátek	konec
banka1	ICC	4 KiB	0x20080000	0x20087FFF
	MCAPI ARM	4 KiB		
	MCAPI SHARC1	4 KiB		
	MCAPI SHARC0	4 KiB		
	ARM	16 KiB		
banka2	ARM	32 KiB	0x20088000	0x2008FFFF
banka3	ARM	32 KiB	0x20090000	0x20097FFF
banka4	ARM	32 KiB	0x20098000	0x2009FFFF
banka5	SHARC1	32 KiB	0x200A0000	0x200A7FFF
banka6	SHARC1	32 KiB	0x200A8000	0x200AFFFF
banka7	SHARC0	32 KiB	0x200B0000	0x200B7FFF
banka8	SHARC0	32 KiB	0x200B8000	0x200BFFFF

Tabulka 6.1: Rozdělení L2 paměti (256 KiB) pro použití MCAPI.

Detail využití prvních 4 KiB paměti (část ICC) je znázorněn na obrázu 6.1. ICC část paměti je využita pro přenos zpráv mezi všemi jádry. Tato paměť je strukturována do několika úrovní, jejichž význam je následující:

- channel: sdílený kanál mezi dvěmi jádry (tento kanál nemá nic společného s kanálem MCAPI standardu). Jelikož platforma ADSP-SC589 obsahuje celkem 3 jádra, tak jsou vytvořeny celkem 3 kanály:
  - 0: kanál mezi ARM a SHARC 0 jádry
  - 1: kanál mezi ARM a SHARC 1 jádry
  - 2: kanál mezi SHARC jádry
- msgq: fronta zpráv. Každý kanál obsahuje čtyři fronty zpráv. MCAPI standard umožňuje zprávám nastavovat prioritu, pro tyto účely jsou vytvořeny dvě fronty, každá pro jinou prioritu:
  - msgq[0][x] - zprávy s vysokou prioritou
  - msgq[1][x] - zprávy s normální prioritou

  Pro každou prioritu dále existují dvě fronty zpráv pro oba směry komunikace. ID jádra (v implementaci odpovídá ICC ID číslu MCAPI uzlu) určuje, která fronta bude pro dané jádro použita jako přijímací (rx), a která jako odesílací (tx). Jádro, které má na daném kanálu vyšší číslo používá fronty následujícím způsobem:
  - msgq[x][0] - odesílací fronta (tx)

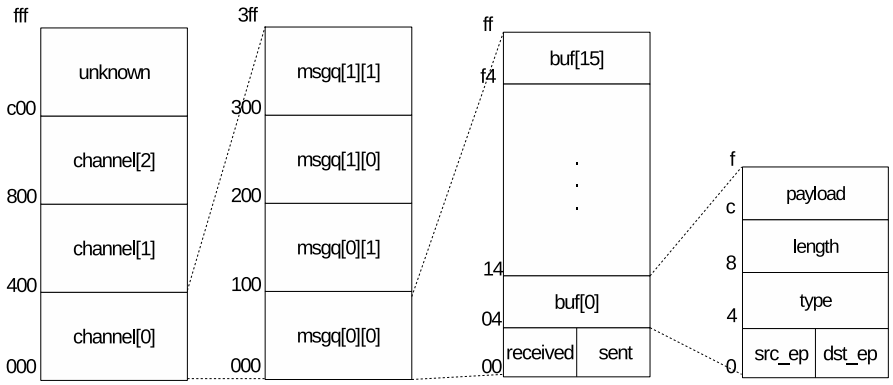
  - msgq[x][1] - přijímací fronta (rx)

  Druhé jádro (s nižším číslem) bude tyto fronty používat opačným směrem.

- sent/received: udává počet odeslaných, respektive přijatých zpráv - jejich rozdíl udává počet dostupných zpráv ve frontě, přičemž ukazatel na zprávu ve frontě se vypočítá:

  ```
  ukazatel = pocet % velikost fronty
  ```

- buf: sdílené buffery mezi jádry pro jednotlivé fronty (pro každou frontu připadá 16 bufferů). Tento buffer neobsahuje přenášená data, nýbrž se jedná o další strukturu:
  - payolad: fyzická adresa na přenášenou zprávu v paměti.
  - length – délka přenášené zprávy.
  - type – typ zprávy (viz sekce 2.4).
  - src_ep – endpoint zdroje.
  - dst_ep – endpoint cíle.



Obrázek 6.1: Rozdělení ICC části L2 paměti pro MCAPI implementaci na platformě ADSP-SC589

Jádra při zaslání nové zprávy informují cílové jádro pomocí přerušení. Tato přerušení jsou vyvolána softwarově způsobem, jak bylo popsáno v sekci 4.5.2. Každé jádro má pevně přidělené přerušení, kterým mu je signalizováno zaslání nové zprávy. Toto přerušení je vyvoláno pomocí softwarových slave triggerů, které jsou spárovány s master triggery dle tabulky 6.2 (číslo odpovídající přešení lze získat z tabuly 4.5, číslo slave triggeru z tabulky 4.6 a master triggeru z tabulky 4.7).

Jádro	**Přerušení**	**Slave trigger**	**Master trigger**
0	TRU0_SLV3	TRU0_SLV3	SOFT3_MST
1	TRU0_SLV7	TRU0_SLV7	SOFT4_MST
2	TRU0_SLV11	TRU0_SLV11	SOFT5_MST

Tabulka 6.2: Synchronizace jader pomocí přerušení a spouštěčů.

Všechna jádra tedy na platformě ADSP-SC589 používají pro komunikaci s ostatními jádry fronty (dvě úrovně priorit) umístěné v L2 paměti. Tyto fronty obsahují počet odeslaných (*send*) a přijatých (*received*) zpráv, přičemž hodnotu odeslaných zpráv upravuje pouze odesilatel (příjemce tuto hodnotu pouze čte), zatímce hodnotu přijatých zpráv upravuje příjemce. Při odeslání zprávy se zároveň vygeneruje přerušení pomocí TRU, které je přiděleno konkrétnímu jádru. Inicializaci, jako například nastavení TRU_SSR registrů (viz sekce 4.5), provádějí všechna jádra.

ICC ovladač – softwarová implementace v Linuxovém jádře

Linuxová implementace MCAPI standardu se skládá z user space knihovny (libmcapi) a Linuxového ICC[2] ovladače.

User space knihovna se dále dělí na několik vrstev:

- *mcapi.c* – základní implementace MCAPI, jejíž funkce při požadavku (čtení/odeslání zprávy atd) pouze zkontrolují validitu požadavku (validní priority, endpointy, velikost přenášeného buffer, ...). Pokud je požadavek v pořádku, předají jej další vrstvě.
- *mcapi_trans_stub.c* – poskytuje pomocné funkce pro ověřování validity požadavků využívané souborem *mcapi.c*. Při obdržení požadavku MCAPI knihovny předají tento požadavek další vrstvě.
- *trans_impl_dev.c* – poslední vrstva user space knihovny. V této vrstvě jsou příchozí požadavky z vyšších vrstev zpracovány a předány Linuxovému ICC ovladači skrze znakové zařízení (pomocí volání funkcí *ioctl*).

Linuxový ovladač (*linux/drivers/staging/icc/core/protocol.c*) přistuje k L2 paměti přidělené pro meziprocesorovou komunikaci a stará se o signalizaci/detekci zpráv mezi ARM jádrem a protějšími SHARC jádry pomocí přerušení. Pro každý kanál, kde je na jedné straně ARM jádro (tedy kanály 0 a 1), je spuštěno samostatné vlákno obsluhující požadavky pro konkrétní kanál.

Ovladač zároveň komunikuje s user space implementací knihovny MCAPI pomocí znakového zařízení a systémového volání *ioctl*. Jak bylo popsáno v sekci 3.2.1, user space a kernel space aplikace pracují s různým adresovým prostorem. Pokud tedy přijde požadavek na přenos bufferu z/do user space aplikace, musí dojít ke kopírování dat mezi těmito adresovými prostory. K této kopii dochází pokaždé (pokud MCAPI zprávy nejsou prázdné), tudíž zde nastává velká režie při přenosu.

MCAPI implementace si současně udržují vlastní buffery (které nejsou přístupné ostatním jádrům). Pokud přijde nová zpráva, která nemůže být vložena do ICC fronty, která je plná, tak je ponechána v paměti vyhrazené daným uzlem (jádrem).

6.1.2 MDMA

Jak bylo popsáno v sekci 4.6, DMAC dostupné na architektuře ADSP-SC58x obsahuje 4 MDMA streamy, přičemž streamy mají různou propustnost. Jak ukazuje tabulka 6.3, jednotlivé streamy lze rozdělit do tří skupin dle propustnosti [9, s. 10].

V sekci 4.1.1 bylo uvedeno, že SRAM paměť jednotlivých SHARC jader je dostupná i v meziprocesorovém adresovém prostoru. Proto je možné MDMA použít nejen pro přenos mezi pamětí DDR, ale také pro přenosy mezi SRAM paměťmi SHARC jader navzájem

[2] Inter-Core Communication

MDMA stream	hloubka FIFO	Rychlost	SHARC propustnost
0	128, 64	nízká	450 MiB/s
1	128, 64	nízká	450 MiB/s
2	128, 64	střední	900 MiB/s
3	128, 64	vysoká	1500 MiB/s

Tabulka 6.3: Porovnání výkonnosti přenosu různých MDMA streamů mezi dvěmi SHARC jádry

nebo pro přenos mezi pamětí SRAM a a DDR. Při studiu této problematiky byl vytvořen Linuxový ovladač testující rychlost přenosu mezi pamětmi DDR a L1 SRAM. V sekci 4.6.3, respektive 4.6.4, byly vysvětleny tři možné způsoby řízení DMA – pomocí registrů, pole deskriptorů a seznamu deskriptorů. Tabulka 6.4 zobrazuje naměřené výsledky přenosu bloku dat o velikosti 192 KiB. Bohužel se při implementaci ovladače nepodařilo zprovoznit deskriptorové přenosy, proto jsou naměřeny pouze přenosy konfigurované pomocí registrů.

Blok 192 KiB		**velikost slova**					
		8b	**16b**	**32b**	**64b**	**128b**	**256b**
Stream 0, 1 [MB/s]	**Mem → Mem**	11	22	44	87	107	107
	Mem → SHARC1	34	67	107	107	107	107
	SHARC1 → SHARC2	57	107	107	107	107	107
	SHARC2 → Mem	39	76	107	107	107	107
Stream 2 [MB/s]	**Mem → Mem**	11	22	44	87	173	172
	Mem → SHARC1	39	76	152	214	214	213
	SHARC1 → SHARC2	57	114	215	215	215	215
	SHARC2 → Mem	46	90	178	213	215	215
Stream 3 [MB/s]	**Mem → Mem**	11	22	44	87	172	174
	Mem → SHARC1	42	82	163	214	214	213
	SHARC1 → SHARC2	72	143	214	215	215	215
	SHARC2 → Mem	50	97	193	214	215	215

Tabulka 6.4: Rychlost DMA přenosu dat řízeného pomocí registrů mezi DDR pamětí (Mem) a SHARC L1 pamětí.

6.2 Návrh komunikace

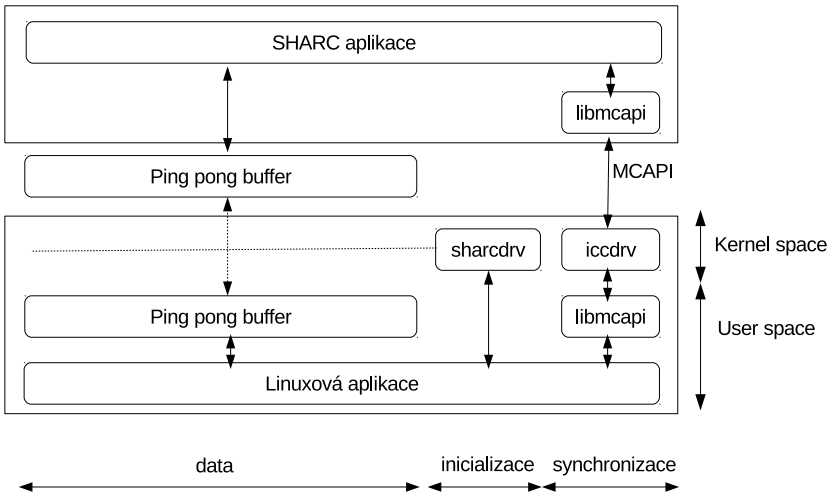
V této sekci budou diskutovány různé návrhy komunikace mezi ARM a SHARC procesorem a jejich implementace.

6.2.1 Přenos velkého množství dat

Pro přenos velkého množství dat se ukázala současná implementace MCAPI jako neefektivní z důvodu velké režie při přenosu a značného omezení velikosti bufferu (256 B). Proto byla navržena výměna velkého množství dat přes dva (ping pong) buffery (implementace se ale dá rozšířit na více bufferů). Jedná se o dva buffery ve sdílené paměti, ke kterým má přístup

ARM i SHARC jádro. Každý z těchto bufferů plní jednu roli – jeden je v jednu chvíli čtecí, druhý zapisovací. Jakmile je čtecí buffer vyprázdněn a zapisovací naplněn, prohodí se jejich role (pomocí MCAPI zpráv).

Tento způsob komunikace je zobrazen na obrázku 6.2. MCAPI zde slouží pro inicializaci a synchronizaci jader. Při inicializaci jsou vytvořeny dva velké buffery ARM jádrem, jejichž adresy jsou zaslány i SHARC jádru.



Obrázek 6.2: Komunikace mezi jádry SHARC a ARM

Softwarová implementace

Pro tento model komunikace byla implementována aplikace přenášející data z SHARC1 jádra do ARM jádra. SHARC jádro bylo zvoleno jako zdroj z důvodu, že se jedná o DSP jádru sloužící pro zpracování audio dat. Taková data mohou být například přenášena do Linuxové aplikace.

Z obrázku 6.2 je patrné, že SHARC aplikace přistupuje k bufferům přímo. Pro Linuxovou aplikaci toto neplatí, jelikož user space aplikace nemá přístup k fyzickém adresám systémové paměti (jak je vysvětleno v sekci 3.2.1). Pro tyto účely byl navržen jednoduchý ovladač, který umí naalokovat potřebné ping pong buffery a namapovat je do user space adresového prostoru. Pro komunikaci mezi jádrem a aplikací bylo vytvořeno znakové zařízení, které přijímá požadavky od user space aplikace pomocí volání *ioctl*.

Další komunikace už probíhá bez účasti tohoto modulu. Linuxová aplikace zašle fyzické adresy SHARC jádru pomocí MCAPI zpráv. Poté jsou MCAPI zprávy využívány pouze k informování aplikace o prohození rolí ping pong bufferů.

Pro funkčnost aplikace je striktně vyžadováno, aby aplikace nezapisovala data do bufferu, do kterého zapisuje druhé jádro – aplikace zná adresy obou bufferů.

Možná vylepšení

V současné implementaci jsou vytvořeny dva buffery, které jsou dostupné oběma jádrům a tak nedochází ke kopii dat. Na SHARC jádro můžou být ale kladeny vysoké nároky (bude provádět náročný výpočet), že přístup do DRR paměti bude SHARC jádro citelně

zpomalovat a SHARC tak nebude například stíhat zpracovat všechna příchozí data. Pro řešení tohoto problému je možné vytvořit ping pong buffery na L1 paměti SHARC jádra. Tato paměť je fyzicky přítomna přímo na SHARC jádru (jak je patrné z ilustrace 4.1) a přístup k ní je velmi rychlý (naopak ale tato paměť má značně omezenou velikost). ARM jádro by mohlo mít kopii bufferů v alokované DDR paměti (ping pong buffery by tak existovaly dvakrát – jednou v L1 SHARC paměti a jednou v DDR). Kopie těchto bufferů může být prováděna pomocí DMA přenosů, kdy SHARC jádro naplní buffer, zahájí přenos a v době přenosu může zaplňovat další buffer (rychlost přenosu pomocí DMA byla změřena a zapsána do tabulky 6.4). Pro synchronizaci DMA kanálů lze využít TRU jednotky popsané v sekci 4.6.5.

6.2.2 Synchronizace, přenos malého množství dat

Jak bylo prokázáno v sekci 6.1.1, současná (zejména Linuxová) implementace MCAPI trpí velkou režií při přenosu a také tím, že některé funkce tohoto standardu nebyly implementovány. Při návrhu komunikace se nabízela možnost vylepšení Linuxové implementace standardu MCAPI, ale ukázalo se to jako neefektivní, neboť MCAPI je pro jednoduchou synchronizaci (nebo přenos malého množství dat) zbytečně komplikované.

Dalším problémem bylo, že se nepodařilo efektivně navrhnout takové řešení pro Linuxovou implementaci, která by netrpěla velkou režií, jako současná implementace. Tento problém je patrný zejména u standardních MCAPI zpráv, tedy při použití funkcí *mcapi_msg_send()* (viz funkční prototyp 6.1) a *mcapi_msg_recv()* (funkční prorotyp 6.2). Diskutovaným parametrem je *buffer*. Jak je vidět, při zasílání je funci předán user space buffer (jedná se o user space knihovnu), který Linuxový ICC ovladač nemá nikde namapován (o tomto bufferu nemusel mít dosud žádnou informaci). Toto by se dalo řešit tak, že by si user space aplikace vytvořila vlastní buffery, které by měla za pomoci jiného ovladače namapovány na user space buffery, ale zároveň by k nim měla i fyzické adresy. V takovém případě by ICC ovladač mohl s adresami bufferů manipulovat jako s fyzickými adresami a tak by odpadla režie při vytváření kopií jejich obsahů.

Listing 6.1: Funkce mcapi_msg_send()

```
void mcapi_msg_send(
        MCAPI_IN mcapi_endpoint_t send_endpoint,
        MCAPI_IN mcapi_endpoint_t receive_endpoint,
        MCAPI_IN void * buffer,
        MCAPI_IN size_t buffer_size,
        MCAPI_IN mcapi_priority_t priority,
        MCAPI_OUT mcapi_status_t * mcapi_status);
```

Při příjmu MCAPI zpráv může sice ovladač mapovat příchozí buffery do user space prostoru, ale tyto buffery by zůstaly neuvolněné až do zavolání ukončovací funkce MCAPI (pro MCAPI zprávy neexistuje funkce, která by informovala ovladač o tom, že buffer již není dál používán).

Listing 6.2: Funkce mcapi_msg_recv()

```
void mcapi_msg_recv(
        MCAPI_IN mcapi_endpoint_t receive_endpoint,
        MCAPI_OUT void * buffer,
        MCAPI_IN size_t buffer_size,
```

```
        MCAPI_OUT size_t * received_size,
        MCAPI_OUT mcapi_status_t * mcapi_status);
```

V případě MCAPI kanálů je situace lepší jen v jednom směru. Jak je vidět z definice funkce *mcapi_pktchan_recv()* (prototyp viz 6.3), při příjmu zpráv z kanálu je získána **adresa** user space buffer s přijatými daty (této funkci se tedy nepředává adresa bufferu, na který mají být data **nakopírována**). Dokonce je dostupná i funkce *mcapi_pktchan_release()* (úplný prototyp 6.4), díky které je možné informovat ovladač o tom, že získaný buffer již není používán. Bohužel ale funkce pro odeslání zprávy přes kanál *mcapi_pktchan_send()* (prototyp 6.5) trpí stejným nedostatkem jako standardní MCAPI zpráva, a sice že je této funkci předávána adresa bufferu, kam mají být nakopírována data. Tedy opět buffer, o kterém nemá ovladač žádné informace.

Listing 6.3: Funkce mcapi_pktchan_recv()

```
void mcapi_pktchan_recv(
        MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
        MCAPI_OUT void ** buffer,
        MCAPI_OUT size_t * received_size,
        MCAPI_OUT mcapi_status_t * mcapi_status);
```

Listing 6.4: Funkce mcapi_pktchan_release()

```
void mcapi_pktchan_release(
        MCAPI_IN void * buffer,
        MCAPI_OUT mcapi_status_t * mcapi_status);
```

Listing 6.5: Funkce mcapi_pktchan_send()

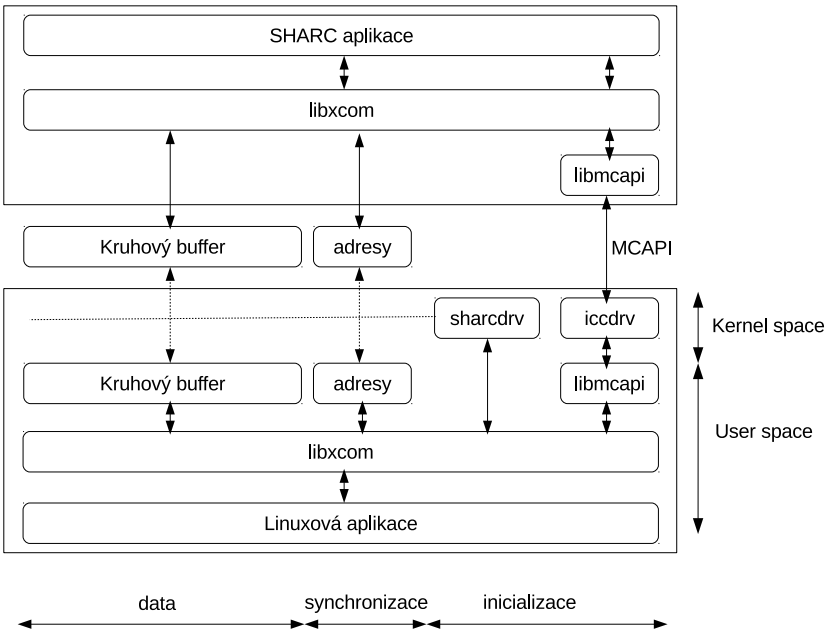
```
void mcapi_pktchan_send(
        MCAPI_IN mcapi_pktchan_send_hndl_t send_handle,
        MCAPI_IN void * buffer,
        MCAPI_IN size_t size,
        MCAPI_OUT mcapi_status_t * mcapi_status);
```

Návrh komunikace

Jak bylo popsáno 6.1.1, při posílání MCAPI zpráv dochází k velké režii (především kopírování mezi user space a kernel space adresovými prostory). V předešlé sekci bylo zároveň popsáno, že omezení kopie mezi těmito adresovými prostory by byla možná jen s výraznými úpravami do současné implementace. Navíc by toho nešlo dosáhnout za použití MCAPI standardu – buď by musel být tento standard upraven, nebo by vedle MCAPI standardu bylo nutné využít nové techniky jako alokace bufferů v Linuxovém ovladači a namapování do user space.

Z těchto důvodů bylo navrženo nové komunikační API (pracovním názvem *xcom*), jehož princip je zobrazen na obrázku 6.3. Je zde vidět, že user Linuxová space space aplikace i SHARC aplikace komunikují přes stejnou knihovnu – tyto aplikace nemusí řešit konfiguraci bufferů ani MCAPI komunikaci.

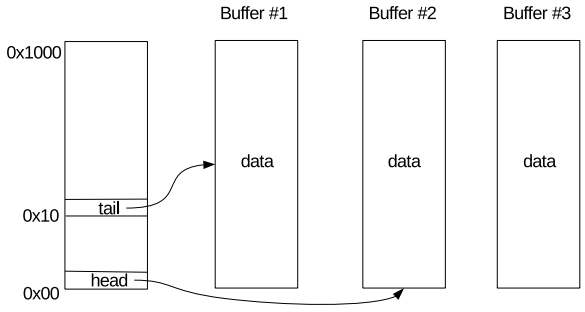
Pro výměnu dat slouží kruhové buffery, které ale nemusí ležet v paměti za sebou. Adresy těchto bufferů si musí pamatovat Linuxová i SHARC implementace této knihovny. 3 buffery buffery jsou zobrany na ilustraci 6.4. Na této ilustraci je vidět i speciální buffer, který



Obrázek 6.3: Komunikace mezi jádry SHARC a ARM

obsahuje ukazatele *head* a *tail*. Tyto ukazatele obsahují adresy prvního volného bufferu (*head*) a posledního obsazeného (*tail*). K těmto bufferům a adresám má přístup Linuxová i SHARC implementace knihovny.

Jeden cyklický buffer slouží pro komunikaci pouze jedním směrem, kdy jedno jádro do těchto bufferů zapisuje (a posouvá ukazatel *head*) a druhé jádro z nich čte (pousouvá ukazatel *tail*). Implementace nikdy nedovolí, aby mohl *head* předběhnout *tail*. Tyto ukazatele, stejně jako funkce kruhových bufferů, jsou před Linuxovou a SHARC aplikací schovány za *xcom* knihovnou.

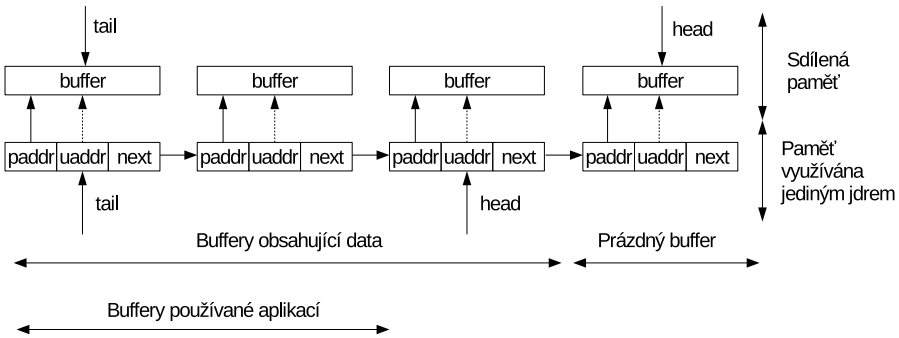


Obrázek 6.4: Kruhový buffer

Alokaci kruhových bufferů a inicializaci ukazatelů *head* a *tail* řeší implementace na ARM jádře. Jakmile jsou tyto buffery připraveny, jsou jejich adresy zaslány SHARC jádru pomocí MCAPI zpráv – MCAPI se tak využívá pouze při inicializaci.

Softwarová implementace

Knihovna *xcom* vytváří pro přenos dat dva kruhové buffery – jeden pro přenos dat z SHARC jádra na ARM a druhý naopak. Jak bylo řečeno v sekci, buffery nemusí ležet za sebou v paměti a zároveň o jejich umístění není ve sdílené paměti žádná informace (kromě head a tail bufferů). Ilustrace 6.5 zobrazuje jak jsou buffery uloženy ve sdílené paměti a jaké informace si o nich uchovává *xcom* knihovna. Zde je ilustrována konkrétně Linuxová verze, která obsahuje fyzickou a namapovanou user space adresu, dále ukazatel na další buffer v kruhovém seznamu.



Obrázek 6.5: Kruhový buffer

Knihovna *xcom* poskytuje aplikacím buffery uspořádané jako FIFO frontu. Práce s těmito buffery pro čtení a zápis fungují stejným způsobem. Aplikace si nejdříve od knihovny *xcom* získá adresu prvního bufferu se kterým může pracovat operací *pop* (v případě čtení prvního plného bufferu, v případě zápisu prvního prázdného bufferu). Knihovna tak ví, že s tímto bufferem pracuje aplikace a nebude tak s ním nijak manipulovat. Jakmile s bufferem dokončí aplikace požadovou operaci (zpracuje jeho data), předá tento buffer zpátky knihovně operací *push* – při této operaci nepředává aplikace adresu bufferu, jelikož tuto informaci knihovna nepotřebuje (s buffery je manipulováno režimem FIFO). Dále je dostupná ještě operace pro zjištění dostupných dat (operace *avail*).

Kromě sdílených ukazatelů *head* a *tail* si knihovna musí uchovávat ještě ukazatel, který rozděluje dostupné buffery ještě na části, kterou si aplikace může alokovat, a kterou už si alokovala. Na ilustraci 6.5 je konkrétně zobrazena situace, kde cyklický buffer používán jako čtení. Ukazatel *head* ve sdílené paměti tak ukazuje na první prázdný buffer, ukazatel *head* v paměti knihovny ukazuje na první neprázdná buffer dostupný pro alokaci a ukazatele *tail* na poslední neprázdný buffer (který je zároveň alokován uživatelskou aplikací).

Pro alokování bufferů a jejich namapování do user space paměťového prostoru byl vytvořen jednoduchý jaderný modul (narozdíl od SHARC jádra nemá Linuxová verze knihovny *xcom* přístup k fyzickým adresám paměti). Tento modul při svém spuštění vytvoří znakové zařízení, na kterém pomocí operace *ioctl* zpracovává a reaguje na požadavky knihovny *xcom*.

Bylo změřeno, že tímto způsobem si jádra **teoreticky** mohou za jednu sekundu vyměnit 884496 zpráv – toto měření probíhalo tak, že se se zapisovanými a čtenými daty nijak nemanipuvalo (aplikace si vyžádala buffer a ihned ho odeslala). Měření proběhlo ještě jednou, tentokrát bylo simulováno zapisování dat (odesílatel si alokoval buffer, zaplnil ho celý daty pomocí funkce *memcpy* a tento buffer vrátil) – čtení dat simulováno nebylo. Zde byly naměřeny dva odlišné výsledky závisející na tom, které jádro provádělo zápis.

V případě SHARC jádra jako odesílatele bylo vyměněno 7042 bufferů za jednu sekundu, což při velikosti 4096 B na jeden buffer odpovídá rychlosti 27,5 MiB/s. V případě ARM jádra jako odesílatele bylo vyměněno 21276 bufferů, což odpovídá rychlosti 83,1 MiB/s. MCAPI implementace pro srovnání dosahovala přenosové rychlosti 5,5 MiB/s.

Možná vylepšení

Jak se ukázalo, MCAPI implementace je při přenosu zpráv neefektivní – dochází k velké režii při kopírování dat. Pokud by se došlo k závěru, že MCAPI nebude na cílové platformě nijak využíváno, tak by se mohla využít L2 paměť pro ukládání sdílených ukazatelů *head* a *tail*. Tím by se manipulace s těmito ukazateli ještě zrychlila.

Současná Linuxová implementace knihovny *xcom* je hlavně v user space. Synchronizační část by se ale mohla přesunout do Linuxového ovladače, kde by mohly být využity pro synchronizaci triggery, jako jsou nyní využívány MCAPI implementací.

Dalším možným vylepšení je vytvoření DeviceTree uzlu pro implementovaný Linuxový ovladač. Tento uzel by odpovídal jednomu SHARC jádru – pro čipy ADSP-SC589 by tak byly vytvořeny uzly dva.

Kapitola 7

Závěr

V této práci byl analyzován hardware platformy ADSP-SC58x, dostupné DMA kanály a možnosti jejich konfigurace, typy pamětí pro ARM i SHARC jádra a bootovací proces dostupných jader. Dále byly popsány prostředky pro vývoj aplikací pro jádra SHARC a ARM, včetně softwarové podpory operačního systému Linux poskytnuté firmou Analog Devices. Byly také diskutovány možné technologie pro výměnu dat mezi jádry čipu ADSP-SC589, jako MDMA a MCAPI. Práce se zabývala také detaily implementace jednotlivých technologií, včetně určitých nedostatků Linuxové implementace standardu MCAPI.

Pro bootování SHARC jader byla vytvořena jednoduchá U-Boot aplikace, která s využitím Boot ROM umí bootovat programy z LDR souborů.

Pro přenos rozměrných dat byla implementována aplikace využívající dva buffery (ping pong) ve sdílené paměti, kde SHARC jádro do bufferů zapisuje a ARM jádro (Linuxová aplikace) z nich čte. Tyto buffery jsou napamovány do user space pomocí vytvořeného jaderného modulu. Jádra jsou o výměně role bufferů informováno pomocí MCAPI zpráv. Jako jedno z možných vylepšení do budoucna je alokace bufferů v L1 SHARC paměti, která je fyzicky přístupná přímo na SHARC jádře, a dále kopie dat z těchto bufferů do DDR paměti pomocí DMA přenosů.

Byla zde také detailně popsána současná implementace MCAPI pro Linux. Jak bylo uvedeno, tato implementace je neefektivní zejména kvůli velké režii při zaslání MCAPI zpráv. Proto byla navržena nová knihovna (*xcom*) pro Linux (ARM) i SHARC, která MCAPI využije pouze pro inicializaci. Pro Linuxovou implementaci jsou buffery využívané touto knihovnou mapované z jaderného modulu, který byl pro tyto účely vytvořen. Výměna dat pak probíhá přes sdílenou paměť. Bylo změřeno, že tato implementovaná knihovna dosahuje lepší propustnosti než knihovna MCAPI. Jako možné vylepšení se naskytuje možnost využít triggery pro synchronizaci jader, kdy by jedno jádro mohlo informovat svůj protějšek o dostupnosti nových zpráv. V takovém případě by se na komunikaci musel podílet ve větší míře i Linuxový ovladač. Pokud by se pro sdílené ukazatele *head* a *tail* v kruhovém bufferu použila paměť L2, tak by se práce s těmito buffery mohla zrychlit. Toto řešení by si ale vyžadovalo podrobnou analýzu využití L2 paměti ostatními aplikacemi, aby nedošlo ke konfliktu. Případně by mohla být odstraněna implementace MCAPI, díky čemu by mohla být využita část L2 paměti původně využívaná touto implementací.

Současná implementace rovněž dokáže detekovat chyby jen v omezené míře – toto by bylo vhodné do budoucna vylepšit. Stejně tak by bylo vhodné přidat nový parametr pro všechny *xcom* funkce, do kterého by byl uložen stav *xcom* operací.

Literatura

[1] Using MCAPI/MDMA for ADSP-SC58x Dual-SHARC Audio Talkthrough. Říjen 2015.

[2] *The Buildroot user manual*. 2016, [Online; navštíveno 9.5.2017].
URL https://buildroot.org/downloads/manual/manual.pdf

[3] Analog Devices: *Understanding and Using Linker Description Files on SHARC Processors*. Leden 2007, [Online; navštíveno 04.01.2017].
URL http://www.analog.com/media/en/technical-documentation/application-notes/ee_069.pdf

[4] Analog Devices: *VisualDSP++ 5.0 Loader and Utilities Manual*. Říjen 2008, [Online; navštíveno 04.01.2017].
URL http://www.analog.com/media/en/dsp-documentation/legacy-software-manuals/VisualDSP5.0_loader_Rev2.1.pdf

[5] Analog Devices: *ADSP-SC589 EZ-Board® Evaluation System Manual*. Květen 2015.

[6] Analog Devices: *ADSP-SC58x SHARC Processor Hardware Reference*. Červen 2015.

[7] Analog Devices: *Implementing Second-Stage Loader on ADSP-BF707 Blackfin® Processors*. Červen 2015.

[8] Analog Devices: *Tips and Tricks Using the ADSP-SC58x/ADSP-2158x Processor Boot ROM*. Září 2015.

[9] Analog Devices: *Using MCAPI/MDMA for ADSP-SC58x Dual-SHARC Audio Talkthrough*. Říjen 2015.

[10] Analog Devices: *ADSP-SC582/583/584/587/589/ADSP-21583/584/587*. Červen 2016.

[11] Analog Devices: *Linux Add-in for CrossCore Embedded Studio*. Únor 2016.

[12] ARM: *Cortex-A5 Technical Reference Manual*. 2016.

[13] Arm Limited: *PrimeCell DMA Controller (PL330)*. 2007.

[14] Association, T. M.: Multicore communications API working group. 2016, [Online; navštíveno 07.01.2017].
URL http://www.multicore-association.org/workgroup/mcapi.php

[15] Corbet, J.; Rubini, A.; Kroah-Hartman, G.: *Linux Device Drivers*. O'Reilly Media, Inc, Leden 2005.

[16] DENX Software Engineering: *The Universal Boot Loader (Das U-Boot)*. Květen 2012, [Online; navštíveno 9.5.2017].
URL http://www.denx.de/wiki/publish/U-Bootdoc/U-Bootdoc.pdf

[17] devicetree.org: Devicetree Specification. Květen 2016, [Online; navštíveno 15.04.2017].
URL http://www.devicetree.org/specifications-pdf

[18] Dutta, P.: Design of Microprocessor-Based Systems. Říjen 2012, [Online; navštíveno 08.01.2017].
URL http://web.eecs.umich.edu/~prabal/teaching/eecs373-f12/notes/notes-toolchain.pdf

[19] Kotásek, Z.: Periferní zařízení: studijní opora. 2007, [Online; navštíveno 12.12.2016].
URL http://www.fit.vutbr.cz/study/courses/IPZ/public/.cs

[20] Likely, G.: The Linux usage model for device tree data. 2013, [Online; navštíveno 27.04.2017].
URL https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt

[21] Martínek, T.: Ovladače zařízení (Linux). 2015.

[22] The SCO Group: *System V Application Binary Interface*. Červen 2013, [Online; navštíveno 04.01.2017].
URL http://www.sco.com/developers/gabi/latest/contents.html

Příloha A

Obsah přiloženého paměťového média

CD obsahuje následující soubory a adresáře:

- br2-external/ – adresář s konfiguračními soubory pro Buildroot.
- src/ – adresář se zdrojovými kódy diplomové práce.
- doc/ – adresář se zdrojovými kódy dokumentace této práce v jazyce LaTeX.
- dp_havran.pdf – tato dokumentace v elektronické podobě.

Příloha B

Manuál

Implementace byla testována na vývojovém kitu ADSZ-SC589 EZ-KIT s dostupným U-Bootem. Pro bootování jader byla vytvořena jednoduchá aplikace *bootrom* pro U-Boot ve verzi standalone i jako příkaz. Tato aplikace je ve formě patchů pro U-Boot verzi od firmy Analog Devices uložena v adresáři *src/bootrom*. Bootování SHARC jádra se provede zavoláním aplikace *bootrom* s offsetem jako jediným parametrem (tento parametr udává offset ve flash paměti, na které je uložen program pro SHARC jádro).

Implementované aplikace (jaderné moduly) je třeba kompilovat vůči verzi Linuxu od firmy Analog Devices (U-Boot i Linux jsou součástí balíku *Linux-Addin for ADSP-SC5xx* od firmy Analog Devices). Pro použití Linuxu s Buildrootem jsou přiloženy konfigurační soubory v adresáři *br2-external*.

Pro spuštění implementace ping pong bufferů (adresář *src/buf*) je nejprve nutné povolit SHARC jádro příkazem *corecontrol –start <CORE_ID>*. Poté je nutné nahrát modul *sharcdrv.ko* a spustit user space aplikaci.

Pro testování knihovny *xcom* (adresář *src/xcom*) není třeba manuálně povolovat SHARC jádro, neboť to provede knihovna automaticky. Opět zde platí, že je nejdříve nutné nahrát modul *sharcdrv.ko* a poté spustit user space aplikaci, která tento modul bude využívat skrze knihovnu *xcom*.

Pro otestování rychlosti MDMA kanálů (adresář *src/mdma*) stačí nahrát zkompilovaný jaderný modul. Tento modul testuje všechny MDMA streamy, proto může být žadoucí nepouštět jiné moduly pracující s těmito kanály (ke konfliktu nedojde, jen daný stream nebude otestován).