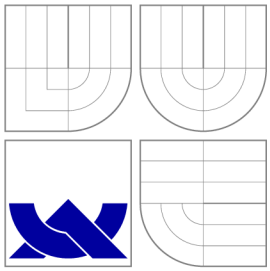
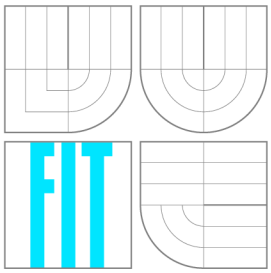


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

FRAMEWORK PRO VÝVOJ APLIKACÍ NA PLATFORMĚ ARM

APPLICATION DEVELOPMENT FRAMEWORK FOR THE ARM PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETR BUCHTA
AUTHOR

VEDOUCÍ PRÁCE — Ing. VAŠÍČEK ZDENĚK, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Tato diplomová práce se zabývá návrhem řešení a realizací frameworku, poskytující základní prostředky pro vývoj aplikací na studijním vývojovém kitu FITkit Minerva. První část této práce je věnována návrhu a implementaci datového kanálu mezi PC a kitem, pro jehož implementaci bylo zvoleno USB rozhraní. Dalším bodem je vytvoření komunikačního rozhraní mezi mikrokontrolérem, postaveným na jádře ARM Cortex M-4, a hradlovým polem FPGA Xilinx Spartan-6. Na straně FPGA byl následně vytvořen systém pro připojení HW komponent, které pomocí implementovaného rozhraní mohou komunikovat s mikrokontrolérem, například pro účely HW akcelerace. V rámci diplomové práce taktéž vznikl systém umožnující programování a ladění FPGA obvodu z vývojového prostředí Xilinx ISE bez nutnosti použití originálního JTAG adaptéru. Toho bylo docíleno využitím protokolu XVC, který umožňuje použití vlastního adaptéru, jež byl v tomto případě implementován jako součást softwaru mikrokontroléru.

Abstract

This master's thesis focuses on designing and implementing a framework that would offer basic program resources for application development on hardware platform FITkit Minerva. First part of this work focuses on designing a data channel between PC and the kit for which the USB interface was used. Next part focuses on implementing a channel between an ARM based microcontroller and FPGA Xilinx Spartan-6. That led to creating a special system inside FPGA that allows adding new HW components that communicate with the microcontroller, which can be used for implementing HW acceleration techniques. Another outcome of this work is a debugging interface that allows to program and debug FPGA using development environment Xilinx ISE without a need of the original Xilinx JTAG adapter. This was achieved by using the XVC protocol that allows to create a custom JTAG adapter that in this case was implemented in the software of the microcontroller.

Klíčová slova

FITkit Minerva, Freescale Kinetis K60, Xilinx Spartan-6, FTDI, XVC

Keywords

FITkit Minerva, Freescale Kinetis K60, Xilinx Spartan-6, FTDI, XVC

Citace

Petr Buchta: Application Development Framework for the ARM Platform, diplomová práce, Brno, FIT VUT v Brně, 2015

Application Development Framework for the ARM Platform

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Zdeňka Vašíčka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Buchta
May 27, 2015

Poděkování

Děkuji panu Ing. Zdeňku Vašíčkovi za odborné vedení této práce, za cenné rady a připomínky

Contents

Chapter 1

Úvod

Vývojové platformy jsou v oblasti vestavěných systému (Embedded Systems) velice důležitým prostředkem. Značně urychlují prvotní fázi seznamování se s použitou platformou pomocí studia již hotových řešení, a tak významně urychlují fázi vývoje konkrétních aplikací. Fakulta informačních technologií (FIT) Vysokého učení technického v Brně vyvinula v minulosti vývojovou platformu FITkit, která dává studentům možnost realizace jak softwarových, tak hardwarových projektů a prohlubuje tak jejich praktické znalosti, které jsou na technické škole důležitou součástí studia. Původní vývojový přípravek FITkit byl založený na 16bitovém mikrokontroléru, hradlovém poli FPGA a řadě periférií, které jsou z dnešního pohledu vlivem rychlého tempa vývoje v oblasti technologií již zastaralé. Z tohoto důvodu vznikla nová verze této studijní vývojové platformy s označením FITkit Minerva. Ta je postavena na moderních komponentách, jako je 32bitový mikrokontrolér založený na ARMovém jádře a hradlové pole Xilinx Spartan-6. Prostředky kitu tak reflektují aktuální trendy ve světě vestavěných systémů, což studenům umožňuje získat mnohem praktičtější zkušenosti.

Předložená diplomová práce se věnuje vývoji základního programového vybavení pro platformu FITkit Minerva, které bude poskytovat řešení pro komunikaci kitu s PC a mikrokontrolérů s FPGA. Jedná se tedy o základní komunikační vrstvy, které jsou důležité pro vývoj aplikací a funkčnost vývojového kitu jako celku. Na tomto řešení budou dále implementovány prostředky pro programování FPGA obvodu tak, aby nebylo nutné použití externího adaptéru. Z finančních důvodů není totiž možné disponovat stejným množstvím programovacích adaptérů jako vývojových kitů. Původně byl kit byl navržen tak, že programování FPGA obvodu bude možné pomocí univerzálního USB kontroléru FTDI VNC2, který je již na desce osazen. Nicméně se ukázalo, že toto řešení nedosahuje uspokojivých výsledků, jelikož například programování flash paměti FPGA obvodu trvá průměrně tři čtvrtě minuty. Jedním z cílů diplomové práce je tedy vytvořit efektivnější alternativu.

Práce je členěna do jedenácti kapitol. Druhá kapitola se věnuje popisu vývojového kitu a možnostem jeho propojení s PC. Jsou zde shrnuty možné varianty, které se běžně za tímto účelem používají v praxi, spolu s rozborem jejich výhod a limitací pro použití v tomto konkrétním případě. Následující kapitola rozebírá způsob komunikace mezi PC a kitem. Za tímto účelem bylo zvoleno řešení emulace obvodu FTDI FT232R a je zde popsán způsob, jakým obvody firmy FTDI komunikují s hostitelským zařízením. Na tuto část dále navazuje kapitola zaměřená na mikrokontrolér (MCU) založený na jádře ARM Cortex-M4 a rozbor hlavních komponent MCU, které byly v této práci použity. Následuje popis FPGA obvodu a rozbor dostupných prostředků pro účely jeho programování.

Další část práce je věnována popisu vytvořeného řešení. V kapitole šest jsou popsané jed-

notlivé softwarové části mikrokontroléru, které jako celek vytvářejí komunikační rozhraní mezi PC, MCU a FPGA. Na tuto kapitolu navazuje popis řešení na straně FPGA, kde bylo vytvořeno rozhraní pro sběrnici FlexBus, která propojuje mikrokontrolér s FPGA a umožňuje připojení dodatečných komponent, jež vyžadují komunikaci s MCU. Příkladem je SPI kontrolér, který byl vytvořen pro účely programování externí flash paměti FPGA obvodu prostřednictvím mikrokontroléru nebo komponenta pro akceleraci šifrovacího algoritmu DES. Tuto část uzavírá kapitola devět, ve které je demonstrována tvorba aplikací založených na vytvořeném řešení.

Pro programování FPGA byly vytvořeny dvě aplikace, které jsou shrnuty v kapitole deset. Jedna je postavená na zápisu do externí flash paměti, což umožňuje nevolatilní zápis konfigurace FPGA obvodu. Druhým řešením je možnost programování přímo z vývojového prostředí Xilinx ISE bez potřeby použití originálního JTAG adaptéru. Ten byl nahrazen pomocí protokolu XVC (*Xilinx Virtual Cable*), který umožnil vytvoření vlastního adaptérů v softwaru mikrokontroléru. V poslední kapitole je zhodnoceno celkové řešení a možnosti budoucího vývoje.

Chapter 2

Vývojový kit Minerva a dostupná komunikační rozhraní

V této kapitole je stručně popsán použitý vývojový kit a možnosti, které nabízí. Následně jsou zde popsána dostupná rozhraní, kterými kit disponuje pro účely propojení s externími periférierni a která jsou vhodná pro komunikaci s PC. Jsou zde rozebrány možné způsoby komunikace a programování komponent po USB rozhraní spolu s jejich výhodami a omezeními. Cílem této kapitoly je popsat kroky, které vedly k výběru jak efektivního, tak univerzálního řešení pro komunikaci s komponentami vývojového kitu a zároveň popsat základní principy tohoto řešení.

2.1 Stručný popis vývojového kitu

Základem vývojového přípravku FITkit Minerva je mikrokontrolér Freescale Kinetis řady K60 postavený na výpočetním jádře ARM Cortex-M4, a programovatelné hradlové pole Spartan-6 firmy Xilinx. Na tyto komponenty je napojena řada rozšiřujících rozhraní, jejichž propojení je graficky znázorněno diagramem 2.1. Deska kitu je přímo osazená standardními konektory pro komunikační rozhraní Ethernet a USB, grafický výstup HDMI, audio vstup/výstup a slotem pro připojení SD karty.

USB systém kitu je složen z komponent USB rozbočovače, mikrokontroléru (MCU), ladícího rozhraní (MCU-DBG) a programovatelného USB kontroléru (VNC2). MCU obsahuje dedikovaný USB řadič, za pomocí kterého je možné implementovat komunikační kanál založený na libovolném USB protokolu. Tento řadič umožňuje komunikaci v rychlostní kategorii Full Speed (USB FS), což představuje maximální teoretickou přenosovou rychlost 12 Mbps [6].

2.2 Volba rozhraní

Jak vyplývá z přehledu dostupných rozhraní na obrázku 2.1, pro účely propojení kitu s PC se nejlépe hodí ethernetové či USB rozhraní. Podívejme se proto na výhody a nevýhody těchto možností. V případě, kdy hlavním cílem je programování vývojového kitu a výměna aplikačních dat mezi kitem a PC, se sice jeví síťové rozhraní jako funkční, ale ne příliš vhodná volba. USB rozhraní je již použito pro programování mikrokontroléru pomocí komponenty MCU-DBG a zároveň je tak řešeno i jeho napájení. Použití ethernetového rozhraní by tak znamenalo další kabel při používání kitu. Při použití USB taktéž odpadá nutnost

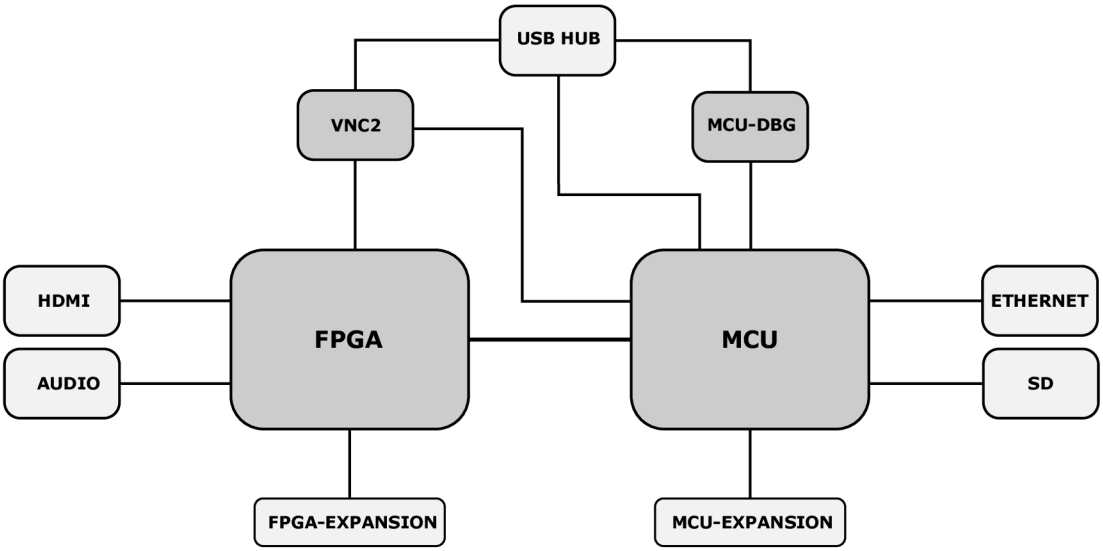


Figure 2.1: Přehled hlavních komponent a jejich propojení

konfigurace síťového rozhraní pro spárování obou stran a s výhodou se dá použít řada standardizovaných komunikačních tříd (viz kapitola 2.3). USB se tedy jeví jako vhodnější a univerzálnější varianta, které v této práci byla dána přednost.

Mimo USB řadič mikrokontroléru by se pro komunikaci po USB dalo využití i komponent MCU-DBG a VNC2, které jsou obě součástí USB systému kitu a zároveň propojeny sériovým rozhraním s MCU. V případě ladícího rozhraní je k dispozici asynchronní sériový kanál (UART) a v případě komponenty VNC2 je k dispozici jak asynchronní UART, tak synchronní SPI [14]. V obou případech lze založit přenos dat mezi MCU a PC po jednom z těchto sériových rozhraní, což eliminuje nutnosti práce s interním USB řadičem MCU. Tato možnost je však značně limitující, jelikož veškerý potenciál přenosové rychlosti USB rozhraní je zde brzděn převodem na rozhraní UART či SPI. Obvod VNC2 byl na desce kitu původně určen pro programování FGPA obvodu, nicméně zkušenosti jsou takové, že není možné dosáhnout přenosové rychlosti větší než 50 kbps, což je nepřijatelná hodnota.

2.3 Možnosti komunikace pomocí rozhraní USB

Specifikace USB definuje tuto sběrnici jako tzv. *host controlled*, což znamená, že všechna zařízení připojena k této sběrnici jsou ovládána jediným hostitelským zařízením, které řídí veškerou komunikaci. Standard definuje názvosloví *host* a *function* pro rozlišení dvou základních prvků topologie USB systému [20]. Termín *function* byl definován pro zpřesnění pojmu *USB device* (*USB zařízení*), který patřičně nerozlišuje roli hosta a připojených zařízení pod jeho správou. V této práci je pojem *USB zařízení* použit pro zařízení v roli *function*.

Každé USB zařízení implementuje určitou funkcionalitu, která je blíže definována pomocí speciálního identifikátoru. Než-li se budeme této problematice věnovat podrobněji, uveďme si, jakým způsobem operační systém pracuje s USB zařízením. Po připojení USB zařízení si hostitelský kontrolér vyžádá jeho identifikaci, na základě které OS zavede patřičný ovladač. Součástí identifikace jsou parametry popisující, zda zařízení implementuje nějakou standardní funkcionalitu. USB specifikace definuje řadu standardních funkčních

tříd (*USB class*), které byly zavedeny z důvodu, aby bylo možné vytvářet univerzální ovladače zařízení. OS má tedy možnost použít standardní ovladače pro celou škálu zařízení, bez nutnosti dodatečné identifikace zařízení uživatelem. Systém USB tříd tedy poskytuje využití jednotného rozhraní pro zařízení od různých výrobců se stejnou či obdobnou funkcionalitou. Jedny z nejčastějších tříd jsou shrnuty tabulkou 2.1. Jelikož jako rozhraní pro propojení vývojového kitu s PC bylo vybráno právě USB, je třeba se zabývat tímto rozdělením USB tříd dále a vybrat takovou, která by nejlépe splňovala zadané požadavky.

ID	**Název třídy**	**Příklad zařízení**
`01h`	Audio	Zvuková karta, mikrofon
`02h`	Communication Device Class (CDC)	Modem
`03h`	Human interface device (HID)	Klávesnice, myš, herní kontroléry
`08h`	Mass Storage (MSC)	USB flash disk, čtečka karet, fotoaparát
`09h`	USB hub	USB rozbočovač
`21h`	Device Firmware Update (DFU)	Zařízeni s možností updatu SW
`FFh`	Vendor specific	Rozhraní ovladače definované výrobcem

Table 2.1: Příklad standardních USB tříd

Cílem diplomové práce je vytvoření řešení pro komunikaci s kitem a programování FPGA obvodu. Z nabídky standardních USB tříd by šly vybrat takové třídy, jejichž kombinací by bylo možné tyto požadavky pokrýt. Například pro aktualizaci softwaru se dá použít třída *Device Firmware Update* a pro vytvoření komunikačního kanálu se již podle názvu nabízí třída *Communication Device Class*. Lze se taktéž vydat cestou vlastního návrhu protokolu (*Vendor specific class*), což ovšem přináší komplikace při tvorbě ovladačů a jejich podpoře v rámci různých operačních systémů. Za těchto okolností je tedy vhodné použít již zavedených řešení, která se již v praxi osvědčila a jsou běžně podporována nejpoužívanějšími operačními systémy. V následujících podsekcích jsou rozebrána taková řešení, která se jeví jako nejvhodnější pro dané použití spolu s výčtem jejich kladů a omezení.

Mass Storage

Jednou z možností, jak realizovat programování FPGA za pomocí USB, je využití standardního *USB Mass Storage* (USB MSC) protokolu, který primárně slouží pro přístup k externímu úložišti dat. Při připojení takového zařízení může operační systém počítače zpřístupnit paměť tohoto zařízení ve formě externího disku, pokud zařízení implementuje nějaký ze souborových sytému a nebo umožňuje jeho vytvoření naformátováním ze strany OS. Základní myšlenkou při využití této funkční třídy při programování komponent jako FPGA či MCU je zpřístupnění programové paměti komponenty formou externího úložiště. Tento přístup vyžaduje kromě USB ovladače na straně mikrokontroléru také implementaci souborového systému (např. FAT12, FAT16), který bude spravovat interní či externí paměť komponenty ve formě souborů. Takto je možné paměť zařízení číst či zapisovat bez potřeby dalšího programového vybavení, jelikož podpora Mass Storage je standardní součástí operačního systému. Pro názornost je tato technika ilustrována obrázkem 2.2, kdy připojené zařízení se v systému objeví jako nový disk, na kterém soubor firmware.bin představuje firmware zařízení. Manipulací se souborem je tedy možné firmware zařízení číst, mazat nebo přepsat.

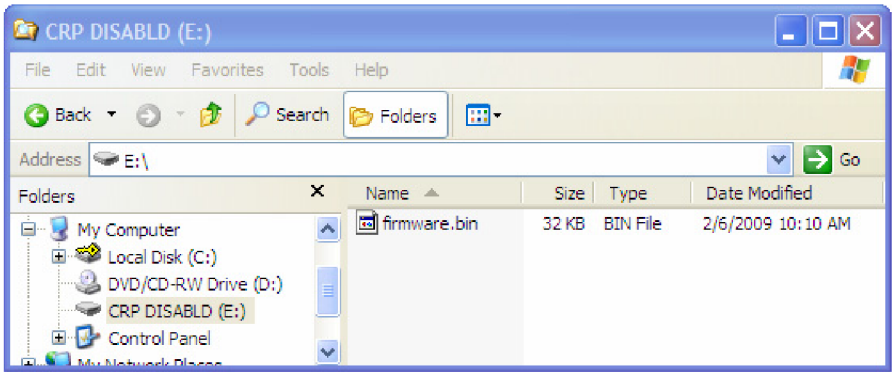


Figure 2.2: Využití třídy Mass Storage pro aktualizaci firmware [18]

Nicméně tento přístup má i své nevýhody a přináší několik komplikací. Ty vznikají především ze snahy použít třídu Mass Storage pro aplikace, na které nebyla primárně navržena, v tomto případě tzv. *In-System Programming* (ISP) (programování bez nutnosti vyjmutí programované komponenty ze systému). Výhoda tohoto přístupu spočívá v tom, že není třeba instalovat další software pro programování. Problematická však může být podpora více operačních systémů. Toto omezení plyne ze způsobu, jakým operační systém pracuje s bloky dat při kopírování na externí disk. Předpoklad je, že bloky jsou zapisovány sekvenčně. V systému MS Windows je toto garantováno při použití souborového systému FAT12, nicméně systémy jako Linux či Mac OS negarantují sekvenční zápis. Pokud se tedy nelze spolehnout na sekvenční zápis, nelze programování komponenty provést bez dodatečného přerovnání dat po skončení zápisu. Podle [18] existují techniky, jak docílit sekvenčního zápisu i na těchto operačních systémech, nicméně každý snaha o obcházení standardního chování OS snižuje uživatelskou přívětivost a komplikuje vývoj.

Device Firmware Upgrade

Vhodnější variantou pro In-System Programming se jeví použití další standardní třídy *USB Device Firmware Upgrade* (USB DFU). Tato třída vznikla pro účely aktualizace softwaru (firmware) v USB zařízeních a nabízí tak způsob provedení ISP, který by měl být nezávislý na operačním systému, výrobci či typu zařízení. Nicméně i přes důležitost manipulace s firmwarem periferních zařízení se tento standard zatím nezdá být široce rozšířený. Při hledání produktů, které tento standard implementují, jsem našel zejména nekomerční open-source projekty. Bohužel aktuálně žádný ze systémů MS Windows neobsahuje standardní ovladače pro tuto třídu. Toto omezení se samozřejmě dá obejít použitím softwaru třetích stran či vlastní implementací [5], [21]. Při testování dostupných ovladačů se však ukázalo, že použití necertifikovaných ovladačů pro MS Windows může být často problematické. Konkrétně u verze Windows 8 bylo potřeba vždy po startu systému ovladač manuálně aktivovat. Vzhledem k nedostatečné podpoře této třídy bylo od jejího použití opuštěno.

Communication Device Class

Vedle programování kitu je dalším cílem vytvoření komunikační linky mezi kitem a PC. Ani jedna ze zmíněných tříd USB MSC a USB DFU však neobsahuje prostředky pro datovou výměnu mezi zařízeními v reálném čase, například za účelem výměny dat, která probíhá periodicky v menších časových intervalech. Pro takové účely by bylo nutné implemen-

tovat další USB třídu, která se svým charakterem hodí pro tyto účely. USB specifikace umožňuje, aby zařízení implementovalo více tříd. V tomto případě se jedná o takzvané kompozitní zařízení, které rozšiřuje svou funkcionalitu nad rámec jediné standardní funkční třídy. Vhodnou třídou pro toto rozšíření k Mass Storage se jeví třída *USB Communication Device Class* (USB CDC). Ta je určena pro zařízení typu modem a obdobná komunikující zařízení. Hojně se tato třída používá pro přechod ze staršího RS-232 rozhraní k USB při zachování softwarové kompatibility na straně PC, kde rozhraní pro komunikaci s připojenou periférií zůstává stejné díky emulaci sériového COM portu [1]. Použitím této techniky by se značně zjednodušila implementace softwaru na straně PC, jelikož rozhraní pro standardní sériový port je součástí knihovního rozhraní nejrozšířenějších operačních systémů. Nicméně operační systém MS Windows neobsahuje standardní ovladače pro tuto emulaci sériového portu a je zde opět zapotřebí použít řešení třetích stran.

Sériové rozhraní obvodů FTDI

V řadě případů standardní USB třídy plně nevyhovují a výrobci pro svá zařízení implementují vlastní způsob komunikace. Jedná se potom o třídu typu vendor-specific. V této sekci se blíže seznámíme s jedním z populárních vendor-specific protokolu, který je použit u masově rozšířených USB převodníků firmy FTDI (Future Technology Devices International Ltd.). Jedná se o hojně používané obvody, které implementují převod mezi USB a řadou jiných standardních rozhraní jako UART, SPI, JTAG atd. Jejich velkou výhodou je široká nabídka podporovaných operačních systémů a rozšířenost ovladačů. Vedle systémů jako Windows, Linux, Mac OS jsou podporovány taktéž i systémy jako Android nebo Windows RT [10].

Výrobce standardně nabízí dva typy ovladačů, kde první varianta vytváří rozhraní mezi převodníkem a PC ve formě virtuálního COM portu (VCP) a druhá varianta implementuje toto rozhraní ve formě knihovny funkcí. Použití standardního COM rozhraní je zejména výhodné při potřebě výměny textových, ale i binárních dat, kdy je možné použít libovolnou aplikaci sériového terminálu a tímto je zajištěna veškerá funkcionalita ze strany PC bez potřeby dalšího softwaru. Aplikace má tak možnost přistupovat k USB zařízení stejným způsobem, jako by přistupovala ke standardnímu sériovému portu. Ovladač ve variantě virtuálního COM portu je u aktuálně podporovaných verzí systému Windows nainstalován automaticky při připojení zařízení a u systému Linux je ovladač součástí jádra. K přímému přístupu k rozhraní převodníku slouží další verze ovladače s označením D2XX, která zpřístupňuje rozhraní FTDI obvodu přes DLL (Dynamic-link library) knihovnu. Aplikační software tak přistupuje k USB zařízení přes sadu funkcí, které umožní ovládat a konfigurovat veškerou funkcionalitu převodníku, která je dostupná [12].

Použitím tohoto protokolu by se výhodně řešila dostupnost ovladačů v rámci nejrozšířenějších operačních systému a tvorba programového rozhraní. Nicméně jako velké omezení protokolu by se mohla zdát jeho uzavřenost. Firma FTDI nikdy nezveřejnila specifikaci tohoto protokolu a tím jeho použití omezila pouze na jejich vlastní produkty. Díky velké popularitě jejich obvodů a jejich masivnímu rozšíření však vznikla i open-source varianta ovladače *libFTDI*, která umožňuje plnohodnotné použití FTDI obvodů včetně studia použitého protokolu [15]. Při použití tohoto otevřeného ovladače spolu s prostředky jako analyzátor USB rozhraní je tedy možné protokol bez problému nastudovat a následně implementovat jako aplikaci mikrokontroléru, která by emulovala chování běžného FTDI obvodu. Takto by se značně zjednodušila implementace softwaru na straně PC, jelikož veškeré potřebné prostředky jsou poskytovány a spravovány firmou FTDI.

Použitím této varianty lze splnit veškeré požadavky, které pro práci s vývojovým kitem byly zadány. Programování komponent lze jednoduše postavit nad rozhraním knihovny D2XX, která implementuje i prostředky pro tzv. *flow-control*, tedy řízení toku dat, které je výhodné, pokud je potřeba zapisovat do paměti, jejíž zápisová rychlost je menší než rychlost přenášených dat [11]. Varianta ovladače VCP (Virtual COM Port) bude umožňovat interakci s kitem pomocí libovolné terminálové aplikace, jako je např. Putty nebo HyperTerminal.

2.4 Popis základních principů komunikace po USB

Jelikož implementace komunikace po USB vyžaduje znalost minimálně nejvyšší protokolové vrstvy USB rozhraní, následující odstavce přibližují tuto problematiku. Vzhledem k robustnosti USB rozhraní není možné zde popsat všechny protokolové vrstvy a principy systému USB. Popis je tedy zaměřen pouze na relevantní části potřebné k implementaci emulace FTDI převodníku. V následujících odstavcích je popsán způsob identifikace zařízení, který je zejména důležitý pro správnou instalaci ovladače, spolu se základními typy USB přenosů. V kapitole 3 je již uveden konkrétní způsob, jakým FTDI obvody využívají těchto prostředků.

Identifikace zařízení

Identifikace každého USB zařízení je provedena na základě předání tzv. USB deskriptorů, které obsahují informace o původu zařízení a jeho konfiguraci. Po výměně těchto informací operační systém provede kontrolu kompatibility zařízení, kde je zejména nutné ověřit číslo revize specifikace USB, rychlostní třídy nebo maximální proudový odběr zařízení. Následně je provedeno vyhledání příslušného ovladače a jeho spárování se zařízením.

Každé USB zařízení obsahuje minimálně následující čtyři deskriptory:

- deskriptor zařízení (Device Descriptor)
- deskriptor konfigurace (Configuration Descriptor)
- deskriptor rozhraní (Interface Descriptor)
- endpoint deskriptor (Endpoint Descriptor)

Vztah deskriptorů je uveden na obrázku 2.3. Ten znázorňuje hierarchii deskriptorů, kdy nejnadřazenějším typem je deskriptor zařízení, který svými parametry dále ovlivňuje počet a význam ostatních deskriptorů. Parametry s prefixem `bNum` určují počet podřazených deskriptorů.

Device Descriptor

Device deskriptor obsahuje obecné informace o zařízení a na rozdíl od ostatních deskriptorů každé zařízení obsahuje pouze jeden tento deskriptor. Jsou v něm obsaženy informace o výrobci zařízení a čísle produktu, tzv. *Vendor ID* a *Product ID* (VID, PID), které spolu unikátně identifikují každé USB zařízení dostupné na trhu. Na základě páru VID:PID dochází k přiřazení ovladače operačním systémem. Dále jsou zde uvedeny parametry, jako je číslo podporované revize USB specifikace, číslo implementované funkční třídy, maximální velikost paketu, počet podporovaných konfigurací zařízení, odkazy na textové deskriptory popisující zařízení textovou formou a další parametry.

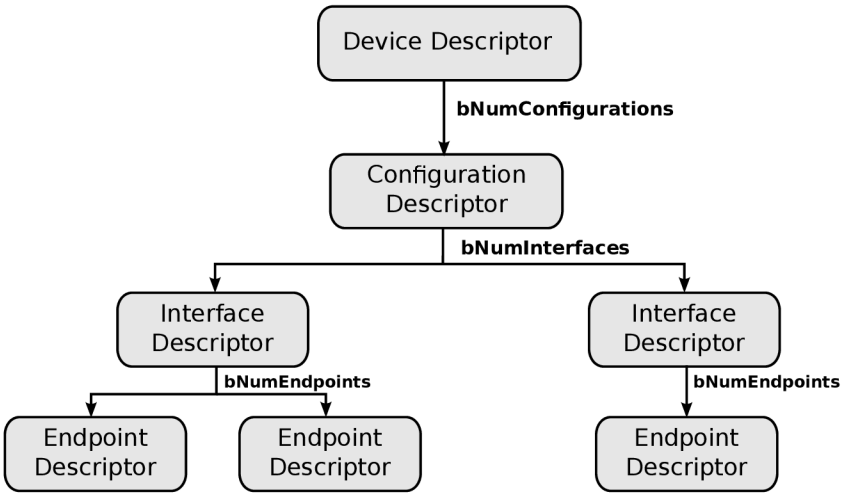


Figure 2.3: Vztah USB deskriptorů

Configuration Descriptor

Konfigurační deskriptor obsahuje informace o způsobu napájení zařízení, jeho maximální odběr a zejména počet rozhraní (interface), které zařízení implementuje. Rozhraní je zde definováno ve smyslu funkce zařízení, kde jedno zařízení může mít například v rámci jedné konfigurace jedno rozhraní pro přenos videa a další pro přenos zvuku. Specifikace dovoluje použití více konfigurací, čímž je zde otevřena možnost pro implementaci odlišné funkcionality v rámci jednoho zařízení. Takto lze například v rámci jedné konfigurace realizovat přenos dat nad pomalejším spolehlivým kanálem a v rámci druhé konfigurace nad rychlejším nespolehlivým kanálem. Výběr konkrétní konfigurace je v režii ovladače zařízení.

Interface Descriptor

Tento deskriptor je použit pro popis jednotlivých funkčních rozhraní v rámci jedné konfigurace. Jsou zde uvedeny identifikátory použité funkční třídy (viz kapitola 2.3), které umožňují přiřazení standardního ovladače. Důležitou položkou je počet použitých endpointů pro dané rozhraní, která rovněž vyjadřuje počet následujících endpoint deskriptorů.

Endpoint Descriptor

Každé USB zařízení používá pro komunikaci tzv. endpointy. Každý endpoint je charakterizován typem přenosů (popsáno dále) a směrem dat. Endpoint číslo nula musí být implementován v každém zařízení, jelikož je použit při procesu enumerace, což je proces konfigurace zařízení při jeho připojení. Počet použitých enpointů plyne z požadavků na zařízení a může jich být celkem 32, tedy 16 vstupních a 16 výstupních. Směr dat se též označuje jako *IN* a *OUT*, kde IN znamená tok dat ze zařízení a OUT směrem do zařízení. Směr dat je tedy značen z pohledu hostitelského zařízení. Každý použitý endpoint (EP) mimo EP0 musí být popsán endpoint deskriptorem, kde jsou uvedeny parametry, jako je směr dat, typ přenosu a maximální velikost paketu.

Typy USB přenosů

Každé USB zařízení implementuje přenos dat použitím minimálně jednoho za čtyř typů datových přenosů, které jsou nad sběrnicí USB definovány. Každý z těchto typů přenosu má odlišnou charakteristiku a jejich kombinací by mělo být možné pokrýt veškeré nároky na přenos dat u většiny periferních zařízení. Jedná se o:

- Control Transfers
- Bulk Transfers
- Interrupt Transfers
- Isochronous Transfers

Obvody firmy FTDI používají ke komunikaci dva z těchto typů, a to *Control Transfers* a *Bulk Transfers*. Jelikož samotný FTDI protokol je postaven na těchto typech přenosu, je vhodné ve stručnosti popsat jejich princip a oblasti použití. Detailní popis lze nalézt v [20], [2].

Control Transfers

Kontrolní přenos (Control Transfer) je typ přenosů, který je určen především pro operace typu příkaz – odpověď, kde nedochází k výměně většího množství dat. Slouží pro zjišťování stavu zařízení a předávání příkazů. Tento typ přenosu je také použit v průběhu celé enumerace zařízení, kde se tímto způsobem předávají deskriptory. Kontrolní přenosy jsou rozděleny na tři fáze: *Setup Stage, Data Stage, Status Stage.*

V první setup fázi host zasílá zařízení požadavek (resp. příkaz) ve formě tzv. SETUP paketu, jehož struktura je znázorněna tabulkou 2.2. Zde jsou vyplněny informace o typu požadavku a jeho parametrech. V bitovém poli `bmRequestType` jsou informace o směru následující datové fáze (Data Stage), kdy zařízení buď odesílá, nebo přijímá data. Vedle této informace je zde uveden i typ požadavku, kde se rozlišuje mezi typy *Standard*, *Class* nebo *Vendor*. V případě FTDI protokolu se jedná o třídu typu Vendor, jelikož není součástí žádné standardní USB třídy, kde by naopak byl použit typ Class. Požadavky typu Standard jsou použity během enumerace zařízení, kdy je zařízení těmito standardními příkazy identifikováno a nakonfigurováno. Pole `bRequest` obsahuje číslo příkazu k provedení a dvojice `wValue`, `wIndex` tvoří parametry tohoto příkazu.

Pole	**Velikost [B]**	**Popis**
`bmRequestType`	1	Bitové pole popisující typ požadavku
`bRequest`	1	Číslo požadavku (request number)
`wValue`	2	První parametr požadavku
`wIndex`	2	Druhý parametr požadavku
`wLength`	2	Velikost dat v datové fázi (Data Stage)

Table 2.2: Popis struktury SETUP paketu

V následující datové fázi (Data Stage) dochází k předání dat ve směru, který byl specifikován v poli `bmRequestType`. Host zde podle směru dat generuje transakci typu IN nebo OUT, čímž zahajuje přenos datových paketů. Následuje fáze potvrzení (Status Stage), kde

je opět podle směru předání dat generována jedna z transakcí IN/OUT, ve které je odeslán datový paket nulové délky stranou, která data v předchozí datové fázi přijímala. Paket nulové délky slouží k signalizaci správného doručení dat. V případě komplikací během přenosu jsou pro tyto případy vyhrazeny prostředky pro signalizaci chyb či zaneprázdněnosti zařízení, nicméně jejich popisu se tento text vzhledem ke stručnosti nevěnuje.

Bulk Transfers

Přenosy typu Bulk jsou použity pro přenos objemných dat, při jejichž přijetí nesmí dojít k jakékoliv ztrátě. Nelze u nich zaručit minimální přenosovou rychlost, jelikož transakce pro tento typ přenosu jsou plánovány host kontrolérem až po uspokojení všech ostatních typů přenosů. Bulk přenosy nejsou tedy vhodné pro real-time aplikace. Jsou použity například pro přenos dat mezi externím diskem nebo tiskárnou. Jejich využitím lze docílit nejvyšší přenosové rychlosti pro aplikace, které vyžadují bezchybný přenos informace. Na rozdíl od typu Control Transfer, popsaného v předchozí sekci, používají Bulk přenosy pouze jednu fázi. Host kontrolér generuje transakci typu IN/OUT v závislosti na směru přenosu, dochází k předání dat a jejich potvrzení přijímanou stranou. Obvody FTDI implementují výměnu veškerých aplikačních dat právě nad přenosy typu Bulk.

2.5 Shrnutí

Cílem této kapitoly bylo vybrat vhodné rozhraní a protokol pro propojení vývojového kitu s PC. Z dostupných rozhraní bylo vybráno rozhraní USB, což se vzhledem k požadavkům na funkcionalitu a dnešnímu rozšíření této sběrnice jeví jako přirozený krok. Pro komunikaci po rozhraní USB byl zvolen protokol od společnosti FTDI, a to hlavně díky značně rozšířené podpoře a programovému vybavení, které tato firma poskytuje. Tato volba by měla ve výsledku zjednodušit vývoj softwaru pro PC a přitom zajistit funkčnost na nejrozšířenějších operačních systémech.

Chapter 3

USB převodníky firmy FTDI a jejich protokol

Tato kapitola se věnuje detailněji popis obvodů firmy FTDI a protokolu, který používají. Firma FTDI nabízí širokou řadu USB obvodů pro převod mezi USB a řadou sériových i paralelních rozhraní. Velice populární a rozšířená je rodina obvodů FTx2xx, která je určená pro převod na standardní sériová rozhraní jako UART, SPI, I^2C či JTAG. Pomocí nich je možné jednoduchým způsobem vytvořit komunikační kanál mezi PC a periferním zařízením. Zařízení komunikuje s FTDI obvodem pomocí jednoho ze sériových protokolů a veškerá aplikační data jsou následně převáděna do kontextu USB rozhraní a přenášena do PC (opačný směr je analogický). Ze strany PC je použito jednotné programové rozhraní, jehož výhody byly popsány v předešlé kapitole (část 2.3).

Konkrétní obvody se liší, kde hlavními parametry jsou zejména typ podporovaného sériového rozhraní, počet kanálů a rychlostní třída USB. Například obvod FT2232H podporuje rychlostní třídu High Speed (až 480 Mbps) a nabízí dva sériové kanály na nichž je možné převádět data z USB na asynchronní UART či synchronní SPI, I^2C nebo JTAG. Naopak obvod FT232R podporuje pomalejší verzi USB Full Speed (až 12 Mbps) spolu s jedním UART kanálem. Obvod FT232R byl vybrán pro potřeby tohoto projektu, jelikož díky své základní funkcionalitě se nejlépe hodí pro studium způsobu komunikace a implementaci kódu pro jeho emulaci.

3.1 Identifikace převodníku FT232R (USB deskriptory)

Identifikace každého USB zařízení je provedena na základě předání tzv. USB deskriptorů, které obsahují informace o původu zařízení a jeho konfiguraci. Na základě těchto informací operační systém vyhodnotí, jaký ovladač pro dané zařízení má použít. Pokud tedy má být pro právě připojené zařízení použit standardní FTDI ovladač, musí se zařízení náležitě identifikovat. Získat popis takového USB deskriptoru lze například připojením jednoho z FTDI převodníků k PC a v systému Linux pak použít standardní aplikaci `lsusb`. V následujících odstavcích jsou uvedeny základní deskriptory obvodu FT232R, jejichž obecný význam byl popsán v předchozí kapitole.

Device Descriptor

Obsah deskriptoru zařízení je v tabulce 3.1. U tohoto deskriptoru jsou důležitá pole `bDeviceClass`, `bDeviceSubClass` a `bDeviceProtocol`, které svými nulovými hodnotami

indikují, že identifikační čísla těchto položek budou uvedena až na úrovni Interface Deskriptoru. Pole `iManufacturer`, `iProduct` a `iSerialNumber` jsou indexy na textové deskriptory (String Desctriptors), které zařízení odesílá na konci enumerace. Pomocí nich může zařízení předat informaci o výrobci a produktu textovou formou. Jejich popis byl zde vynechán a lze jej nalézt v [20].

Pole	Velikost	Hodnota	Popis
`bLength`	1	18	Velikost deskriptoru v bajtech
`bDescriptorType`	1	1	Typ deskriptoru
`bcdUSB`	2	0x0200	Číslo revize USB ve formátu BCD
`bDeviceClass`	1	0	Číslo funkční třídy, tzv. Class Code
`bDeviceSubClass`	1	0	Číslo funkční podtřídy
`bDeviceProtocol`	1	0	Číslo protokolu v rámci dané třídy
`bMaxPacketSize`	1	8	Velikost paketu pro endpoint č. 0
`idVendor`	2	0x0403	Vendor ID
`idProduct`	2	0x6001	Product ID
`bcdDevice`	2	0x0600	Revize produktu
`iManufacturer`	1	1	Index textového deskriptoru výrobce
`iProduct`	1	2	Index TD produktu
`iSerialNumber`	1	3	Index TD sériového čísla
`bNumConfigurations`	1	1	Počet konfigurací zařízení

Table 3.1: Device Descriptor použitého obvodu FTDI FT232R

Configuration Descriptor

V tabulce 3.2 je zobrazen obsah Configuration deskriptoru. Pole `wTotalLength` obsahuje součet velikostí tohoto a všech následujících deskriptorů (Interface Descriptor a Endpoint Descriptor), jelikož je zařízení posílá jako jeden celek. Pole `bmAttributes` kóduje informaci, že zařízení je napájeno z rozhraní USB a podporuje tzv. Remote Wakeup, což je technika šetření spotřeby, kdy USB zařízení umožňuje probudit USB host kontrolér z režimu spánku.

Pole	Velikost	Hodnota	Popis
`bLength`	1	9	Velikost deskriptoru v bajtech
`bDescriptorType`	1	2	Typ deskriptoru
`wTotalLength`	2	32	Velikost všech dalších deskriptorů
`bNumInterfaces`	1	1	Počet dostupných rozhraní
`bConfigurationValue`	1	1	Číslo této konfigurace
`iConfiguration`	1	0	Index TD pro tuto konfiguraci
`bmAttributes`	1	0xA0	Další parametry zařízení
`bMaxPower`	1	90mA	Maximální spotřeba zařízení

Table 3.2: Configuration Descriptor použitého obvodu FTDI FT232R

Interface Descriptor

Tabulka 3.3 obsahuje hodnoty interface deskriptoru, kde podstatné jsou údaje o počtu použitých endpointů a třída implementovaného protokolu. Počet endpointů je popsán polem `bNumEndpoints`. V tomto případě deskriptor signalizuje dva USB endpointy, jejichž popis bude probrán dále. Hodnota 0xFF v polích `bInterfaceClass`, `bInterfaceSubClass`, `bInterfaceProtocol` znamená, že není použita žádná ze standardních USB tříd a implementovaný protokol je Vendor-specific.

Pole	**Velikost**	**Hodnota**	**Popis**
`bLength`	1	9	Velikost deskriptoru v bajtech
`bDescriptorType`	1	4	Typ deskriptoru
`bInterfaceNumber`	1	0	Číslo rozhraní
`bAlternateSetting`	1	0	Číslo alternativního rozhraní
`bNumEndpoints`	1	2	Počet endpointů
`bInterfaceClass`	1	0xFF	Číslo třídy (Class Code)
`bInterfaceSubClass`	1	0xFF	Číslo podtřídy
`bInterfaceProtocol`	1	0xFF	Číslo protokolu
`iInterface`	1	2	Index TD popisující toto rozhraní

Table 3.3: Interface Descriptor použitého obvodu FTDI FT232R

Endpoint Descriptor

V případě obvodu FT232R jsou použity mimo povinný EP0 také EP1 IN a EP2 OUT. Jejich použití je signalizováno dvěma endpoint deskriptory uvedenými v tabulce 3.4. Číslo a směr endpointu jsou kódovány v poli `bEndpointAddress`. Pole `bmAttributes` obsahuje informaci, že oba tyto endpointy používají jako typ přenosu *Bulk Transfer*. Maximální velikost paketu je 64 B, což je také maximum u verze USB Full Speed. Pole `bInterval` je u přenosů typu Bulk Transfer ignorováno.

		EP1	**EP2**	
Pole	**Velikost**	**Hodnota**		**Popis**
`bLength`	1	7	7	Velikost deskriptoru
`bDescriptorType`	1	5	5	Typ deskriptoru
`bEndpointAddress`	1	0x81	0x02	Kódování adresy a směru EP
`bmAttributes`	1	0x02	0x02	Kódování parametrů EP
`wMaxPacketSize`	2	64	64	Maximální velikost paketu
`bInterval`	1	0	0	Polling interval

Table 3.4: Endpoint Descriptors použitého obvodu FTDI FT232R

3.2 Popis FTDI protokolu

Obvody FTDI komunikují s ovladačem na straně PC pomocí sady příkazů. Jejich význam a formát je důležitý pro vytvoření aplikace v mikrokontroléru, která by činnost FTDI obvodu napodobovala. Velká část protokolu je tvořena příkazy pro konfiguraci parametrů převodníku, které bude stačit při emulaci obvodu pouze vhodně potvrzovat, jelikož aplikace mikrokontroléru bude mít zpracování dat ve vlastní režii. V následujících odstavcích jsou popsány nejdůležitější příkazy a způsob přenosu aplikačních dat.

Konfigurace parametrů převodníku

Pro konfiguraci USB–UART převodníku v FTDI obvodech je určena skupina příkazů, které jsou předávány pomocí kontrolních USB přenosů (Control Transfer). Slouží k nastavení parametrů jako rychlost rozhraní, šířka dat nebo řízení toku. Ačkoliv tato funkcionalita není pro naše účely potřebná, můžeme význam těchto příkazu přizpůsobit konkrétním potřebám a nebo zcela změnit jejich význam a využít toho, že ovladač a jeho DLL rozhraní obsahuje rutiny pro jejich vyvolání.

bRequest	**Funkce**
0x00	Reset převodníku
0x01	Změna stavu signálů RTS/DTR
0x02	Nastavení typu řízení toku dat
0x03	Nastavení rychlosti převodníku
0x04	Nastavení parametrů převodníku

Table 3.5: Příkazy pro konfiguraci převodníku FT232R

Každému příkazu náleží i parametr daný polem `wValue`. Jejich možné lze je nalézt v [15]. Pole `wIndex` slouží pro specifikaci kanálu převodníku a v případě FT232R, který obsahuje pouze jeden, je toto pole vždy nulové. Signály RTS/DTR se používají ve spojení s řízením toku (flow-control) a slouží pro signalizaci připravenosti přijímat či odesílat data.

Přístup k EEPROM

Obvod FT232R obsahuje integrovanou paměť typu EEPROM, která umožňuje uživateli měnit některé parametry jako například textový popis zařízení a výrobce. Tuto paměť lze také využít pro uložení uživatelských dat. Příkazy pro práci s EEPROM jsou shrnuty v tabulce 3.6. Adresa daná parametrem `wIndex` adresuje jedno 16bitové slovo.

bRequest	**Funkce**
0x90	Čtení na pozici `wIndex`
0x91	Zápis hodnoty `wValue` na pozici `wIndex`

Table 3.6: Příkazy prací s pamětí FT232R

Při emulaci obvodu je nutné implementovat chování této paměti, jelikož ovladač při připojení zařízení tuto paměť čte pro získání případné dodatečné konfigurace. Pro získání

originálního obsahu paměti obvodu FT232R stačí paměť přečíst například pomocí knihovního rozhraní D2XX a tento obsah použít pro implementaci emulující aplikace v mikrokontroléru.

Přenos aplikačních dat

Přenos veškerých aplikačních dat je založen na přenosech typu Bulk. Ať už je pro zařízení použit ovladač D2XX nebo VCP, formát přenášených dat je vždy stejný. V případě přenášených dat směrem do zařízení se data přenášejí v originální podobně bez jakéhokoliv kódování. Hostitelské zařízení pouze iniciuje novou Bulk transakci typu OUT a odešle požadovaná data. Předání dat v opačném směru, tedy ze zařízení, je realizováno odlišně. Data předaná v každé transakci obsahují jako nultý a první bajt stavové bity informující o stavu zařízení. Tímto způsobem hostitelská strana získává přehled o událostech, jako je výskyt chyby při převodu dat nebo informace o řízení toku. Formát tohoto bitového pole a význam jednotlivých bitů je shrnut tabulkou 3.7.

0. bajt		**1. bajt**	
Bit	**Význam**	**Bit**	**Význam**
0,1,2,3	Nepoužito	1	Overrun Error
4	Clear To Send	2	Parity Error
5	Data Set Ready	3	Framing Error
6	Ring Indicator	4	Break Interrupt
7	Data Carrier Detect	0,5,6,7	Nepoužito

Table 3.7: Význam stavových bitů

Nultý bajt obsahuje tzv. modem status informace, kde se nachází např. příznak Clear To Send (CTS) pro řízení toku. Následující bajt obsahuje informace o chybách a událostech na lince převodníku. Jelikož přenosy typu Bulk neumožňují řízení přenosové rychlosti a nelze tedy upravit přenosovou rychlost na požadovanou hodnotu v případech, kdy zařízení nestačí data přijímat, musí se použít nějaký z prostředků pro řízení toku dat (flow-control). Díky implementaci modem-status bajtu stačí změnit hodnotu signálu CTS v případech kdy zařízení potřebuje více času ke zpracování a předejít tak ztrátě dat. Vysílací strana pak pouze kontroluje stav tohoto signálu před odesláním dalšího bloku dat.

Chapter 4

Mikrokontrolér Kinetis K60

Tato kapitola je věnována popisu mikrokontroléru Freescale Kinetis K60, který tvoří jednu z hlavních částí vývojového kitu. Jsou ze popsány prostředky mikrokontroléru a část je věnována i jádru ARM Cortex-M4, na kterém je toto MCU založeno. Je zde rozebrán USB řadič mikrokontroléru, pomocí kterého bude implementováno komunikační rozhraní mezi PC a vývojovým kitem. Zároveň je zde popsán USB ovladač firmy Freescale, který bude třeba integrovat. Konec kapitoly je věnován popisu systému FlexBus, který je určen pro komunikaci s externími komponentami a v tomto případě je použit pro komunikaci mezi MCU a FPGA.

4.1 Mikroprocesor ARM Cortex-M4

Použitý mikrokontrolér Freescale Kinetis K60DN512 spadá do rodiny mikrokontrolérů Kinetis K60, která je specifická především integrací ethernetového a USB rozhraní. Tato řada je cílená na aplikace vyžadující širokou nabídku komunikačních rozhraní, prostředků pro zpracování analogového signálu, hardwarovou akceleraci kryptografických prostředků a také důraz na minimální spotřebu. Maximální frekvence mikroprocesorového jádra je u tohoto modelu 100 MHz. Paměťové prostředky činí 512 kB ROM a 128 kB RAM.

Z komunikačních rozhraní je možné, mimo již zmíněné, dále využít standardních rozhraní typu UART, SPI, I^2C, I^2S, CAN nebo SDHC. Pro zpracování analogového signálu je k dispozici 16bitový AD převodník, 12-bitový DA převodník, vysokorychlostní komparátor nebo jednotka napěťové reference. Je zde použit komplexní systém správy hodinového signálu, který umožňuje využití interního či externího zdroje signálu a jeho následnou parametrizaci pomocí modulů FLL (Frequency-Locked Loop) či PLL (Phase-Locked Loop). Pro aplikace vyžadující běh na baterie je k dispozici až 10 nízkoodběrových režimů spolu s možností tzv. clock-gating pro odpojení nepotřebných periférií. Pro oblasti vyžadující kryptografické zabezpečení je MCU vybaveno jednotkou pro akceleraci kryptografických algoritmů nebo generátorem náhodných čísel [7] [6].

V MCU řady Kinetis K60 je použit mikroprocesor Cortex-M4 navržený firmou ARM. Jedná se o plně 32bitový procesor, tj. registry, datová i paměťová sběrnice mají šířku 32 bitů. Je zde použit koncept Harvardské architektury, což tedy znamená oddělenou paměť i cestu pro instrukce a data, a tedy prostor pro zvýšení výkonu. Vedle klasických prostředků jsou zde implementovány i jednoduché techniky paralelního zpracování dat jako SIMD (Single Instructin, Multiple Data) instrukce, které umožní efektivní zpracování především v oblasti zpracování signálu.

Architektura Cortex-M4 neimplementuje pouze část mikroprocesoru, ale i důležité periférie jako řadič přerušení, jednotku ochrany paměti (MPU), jednotku pro výpočty v plovoucí řádové čárce (FPU) nebo tzv. debugovací rozhraní a další. Díky tomu, že jsou tyto obvody specifikovány jako součást architektury, odpadá nutnost nahradit tyto obvody výrobcem a vzniká tak cesta pro standardní rozhraní, které usnadňuje práci s různými mikrokontroléry od různých výrobců, založenými na totožné architektuře. V případě řady procesorů ARM Cortex-M vzniklo standardní rozhraní *CMSIS* (Cortex Microcontroller Software Interface Standard) zpřístupňující funkcionalitu komponent této architektury. Díky této snaze o standardní rozhraní je možné psát software s ještě větší mírou přenositelnosti bez ohledu na výrobce MCU [25].

4.2 USB řadič

Použitý mikrokontrolér obsahuje USB řadič (USB Controller) podporující specifikaci USB 2.0 a rozšíření *On-The-Go* (OTG). Podporované rychlostní kategorie jsou Low Speed a Full Speed. Řadič implementuje i podporu pro funkcionalitu v hostitelském režimu, se kterou je OTG rozšíření spojeno. To definuje způsob, kterým se zařízení domluví na tom, kdo bude obstarávat roli hostitelského zařízení a kdo roli připojeného zařízení (klienta). OTG rozšíření se využívá u zařízení, které musí umět oba režimy pro dosažení potřebné funkcionality. Příkladem může být tiskárna, která je v režimu klienta při tisku z PC a v režimu hosta při připojení fotoaparátu pro tisk [6]. V tomto konkrétním případě není hostitelský režim nijak užitečný a následující popis bude zaměřen pouze na režim v roli klienta.

USB kontrolér poskytuje veškerou potřebnou funkcionalitu pro implementaci USB zařízení bez potřeby obsluhovat kteroukoliv z nižších protokolových vrstev USB rozhraní v rámci softwaru. Kontrolér provádí veškerou práci až na transakční úroveň. V jeho režii jsou úlohy, jako je tvorba a zpracování paketů, dodržování správného pořadí paketů pro různé typy přenosu, potvrzování doručení a zejména kontrola poškození dat. Tato funkcionalita představuje nejspodnější vrstvu ze tří vrstev, které USB specifikace definuje pro rozdělení každé implementace USB zařízení. Zbývající dvě nadřazené úrovně jsou již realizovány v softwaru MCU. Software MCU tedy musí obstarat veškerou zbylou funkcionalitu počínaje řízením transakcí (Transaction Layer) spolu s prostředky pro konfiguraci zařízení (USB Device Framework) a samotnou aplikaci charakterizující funkci zařízení [20].

Hlavní částí kontroléru je Serial Interface Engine (SIE), který obstarává veškerý převod aplikačních dat do protokolového formátu USB. Jeho činnost je řízena prostřednictvím datové struktury zvané Buffer Descriptor Table popsané v následující části. Nastavení a řízení kontroléru je zajištěno prostřednictvím sady 43 registru. Jelikož z pohledu implementace je vývoj ovladače kontroléru poměrně náročný na čas, dá se využít prostředků poskytovaných výrobcem. Ten zdarma nabízí k využití tzv. USB Stack, jehož součástí je jak ovladač kontroléru, tak i implementace transakční vrstvy. Pro jejich integraci je potřeba vytvořit mezivrstvu zpřístupňující prostředky MCU pod jednotným rozhraním. Toto je nezbytné, jelikož se jedná o univerzální kód a takto jej lze integrovat do různých prostředí jako například MCU s RTOS (Real-Time Operating System) či bez něj [5].

Buffer Descriptor Table

Hlavním prostředkem pro správu USB transakcí je datová struktura *Buffer Descriptor Table* (BDT). Ta obsahuje tzv. buffer deskriptory pro každý použitý endpoint, jejichž

konfigurací se vytváří nové transakce mezi daným endpointem a hostitelským zařízením. Jedná se o datovou strukturu vytvořenou v paměti mikrokontroléru, nikoliv v registrech MCU. Paměť vyhrazená pro BDT musí být zarovnána na hranici 512 B. Každý použitý endpoint (pro každý směr zvlášť) vyžaduje dva buffer deskriptory, které se rozlišují jako ODD a EVEN (lichý, sudý). Jedná se o techniku tzv. double bufferingu, která umožňuje eliminovat vliv režie procesoru při čekání na uvolnění právě zpracovávaného bufferu a docílit tak větší propustnosti. Jeden buffer deskriptor představuje 8 B paměti a vlivem double bufferingu na každý endpoint v daném směru připadne 16 B. Při využití všech 16 endpointů v každém směru by tedy bylo potřeba 512 B paměti. Adresa konkretního záznamu BD vznikne konkatenací parametrů endpointu (číslo, směr, pořadí BD) s bázovou adresou BDT. Formát výsledné adresy je popsán tabulkou 4.2.

Struktura buffer deskriptoru (BD) je popsána tabulkou 4.1. Prvních 32 b slouží k popisu transakce a následujících 32 b pro adresu alokovaného bufferu pro přenášená data. Význam první poloviny BD se mění podle toho, kterou stranou je záznam BD aktuálně vlastněn. V tabulce 4.1 je popsán význam v případě, kdy byl záznam modifikován jednotkou SIE a předán procesoru ke zpracování. Aktuální vlastník je dán bitem `OWN`. Tento bit představuje prostředek vzájemného vyloučení, který je potřeba, jelikož prvních 32 b BD je modifikováno oběma stranami. Velikost transakce je dána polem `BC` (Byte Count). Zbývající pole `TPID3-0` a `DATA0/1` signalizují, o jaký typ transakce se jedná (Token PID) a zda byl pro přenos dat použit sudý nebo lichý datový paket. Detailní popis významu těchto bitů v je uveden v [20], [6].

K notifikaci o událostech jako přijetí dat v rámci transakce slouží dedikovaný kanál přerušení pro USB kontrolér spolu se stavovým registrem přerušení `USB0_ISTAT` signalizující událost, která toto přerušení vyvolala. V případě události `TOKDNE` bude register `USB0_STAT` obsahovat popis parametrů enpointu, u kterého právě došlo k dokončení transakce specifikované jeho buffer deskriptorem.

31:26	**25:16**	**15:8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
–	BC	–	OWN	DATA0/1	TPID3	TPID2	TPID1	TPID0	0	0
Buffer Address										

Table 4.1: Formát buffer deskriptoru

Pozice	**Význam**
31:9	Bázová adresa BDT zarovnaná na 512 B
8:5	Číslo endpointu
4	Směr endpointu
3	Použití sudého/lichého bufferu
2:0	Vždy nula

Table 4.2: Kompozice adresy buffer deskriptoru

4.3 Kinetis USB Device Stack

Rozhraní USB řadiče je programově dostupné ve formě sady registrů a BDT tabulkou v paměti (viz 4.2), což z pohledu uživatelské aplikace představuje velice nekomfortní a těžkopádný způsob jeho obsluhy. Je tedy potřeba tyto dvě vrstvy oddělit prostřední vrstvou, která zapouzdří obsluhu USB kontroléru do sady funkcí. Tato prostřední vrstva se skládá z ovladače USB kontroléru a logiky, která spravuje příchozí a odchozí datové transakce, jejichž vytvoření bylo iniciováno nadřazenou aplikační vrstvou. Software implementující tyto prvky se často označuje jako tzv. USB Stack, tedy sada funkčních vrstev jejichž vztah by se dal vyjádřit zásobníkovým modelem a jako celek vytvářejí jednotné rozhraní pro uživatelskou aplikaci.

V tomto případě byla vybrána implementace přímo od výrobce mikrokontroléru, společnosti Freescale, která zdarma nabízí implementaci USB ovladače pro své modely řady Kinetis. Jedná se o univerzální software, který byl napsán tak, aby jel bylo možné použít na systémech s různými USB řadiči nebo na systémech s operačním systémem (resp. RTOS) či bez něj. Daní za tuto univerzálnost je však složitější integrace. Chceme-li použít konkrétní kombinaci MCU a OS, je třeba vytvořit dodatečné rozhraní, které by odstiňovalo rysy cílového systému, jako je například práce s pamětí nebo zpracování přerušení.

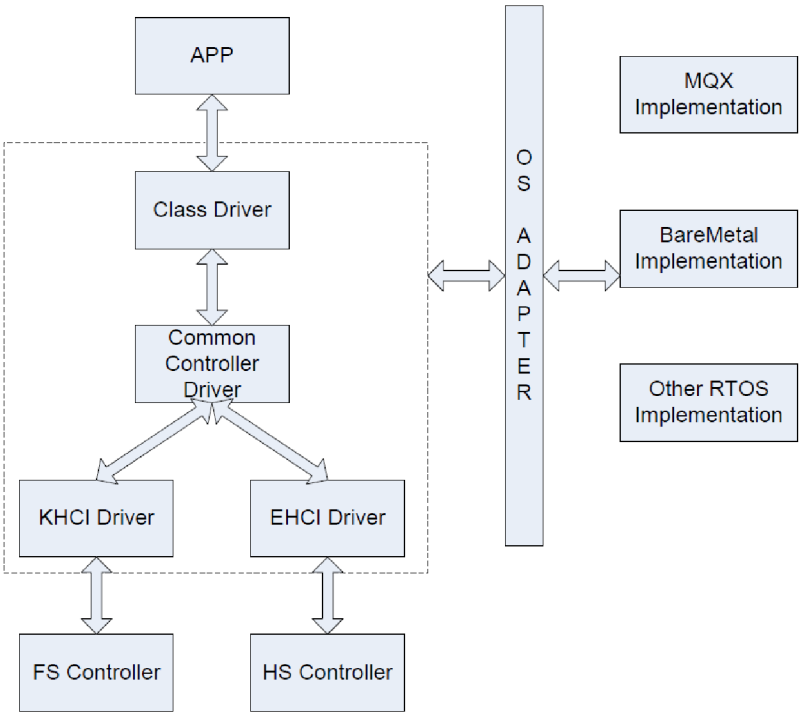


Figure 4.1: Vztah komponent použitého USB ovladače [9]

Struktura použitého zásobníku je znázorněna na obrázku 4.1. Části uvnitř čerchované oblasti tvoří komponenty integrovaného USB zásobníku. Nejspodnější vrstvou je ovladač USB kontroléru. V tomto případě je použit **KHCI Driver**, jelikož použité MCU obsahuje USB kontrolér rychlostní kategorie Full Speed. Nadřazenou vrstvu tvoří **Common Controller Driver**, která se stará o správu datových transakcí a již poskytuje aplikační rozhraní, které je zcela nezávisle na použitém hardware. Vrstva **Class Driver** implemen-

tuje rozhraní pro aplikace využívající nějaké ze standardních USB tříd. V této práci se však žádná taková třída nepoužívá, jelikož FTDI protokol není na žádné z nich postaven, a tak je její funkcionalita nevyužita. V pravé části diagramu je znázorněn vztah USB zásobníku s variantou operačního systému, použitého na cílové platformě. Ať už se jedná o jakoukoliv variantu OS, je jeho rozhraní odděleno vrstvou nazvanou **OS Adapter**, která překládá požadavky USB zásobníku do kontextu cílového OS. Jedná se o funkce jako alokace paměti, nastavení přerušení, funkce pro vzájemné vyloučení a podobně. Varianta bez jakékoliv varianty OS je zde pod názvem **Bare Metal**.

4.4 Rozhraní FlexBUS pro komunikaci s FPGA

Použité MCU Kinetis K60 obsahuje rozhraní FlexBUS, které slouží pro komunikaci s externími obvody jako paměti typu ROM, FLASH, programovatelná logická pole či ostatní obvody, jejichž komunikace je založena na podobném principu. Jedná se o multifunkční paralelní sběrnici, která dovoluje konfiguraci řady parametrů a tím možnost přizpůsobení pro celou řadu externích zařízení. Komunikace je založena na principu master–slave, kde všechny transakce jsou iniciovány ze strany MCU. V případě vývojového kitu FITkit Minerva byla tato sběrnice použita pro propojení MCU s FPGA.

Adresovou i datovou šířku je zde možné zvolit 8, 16, nebo 32 b. Předání adresy a dat je multiplexováno, jelikož adresa i data sdílí stejné signály. Přehled signálů potřebných pro výměnu dat a signalizaci je uveden v tabulce 4.3.

Signál	**Název**
FB_CLK	Clock
FB_AD[0:31]	Address/Data
FB_RW	Read/Write
FB_ALE	Address Latch Enabled
FB_CS#	Chip Select
FB_TA#	Transaction Acknowledged

Table 4.3: Použité FlexBUS signály

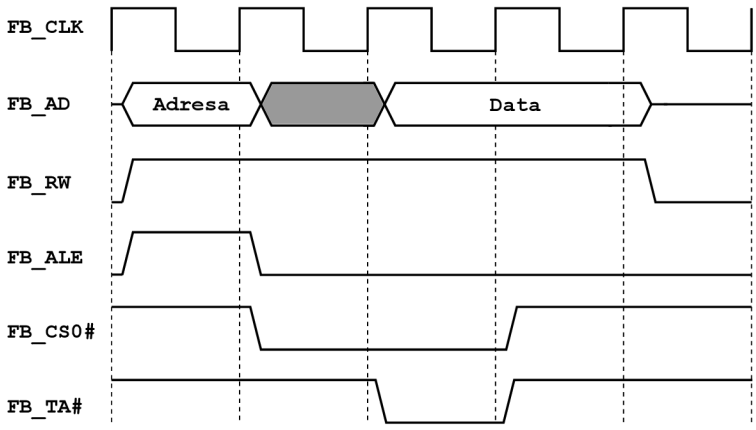


Figure 4.2: Ukázka operace čtení na sběrnici FlexBUS

Průběh čtecí operace z externího zařízení je znázorněn na obrázku 4.2. Komunikace je synchronizována hodinovým signálem `FB_CLK`, který je generován ze strany MCU. Při zahájení čtení je nejprve vystavena adresa na portu `FB_AD[0:31]` spolu se signálem `FB_ALE`, který informuje protější stranu o této události. Signál `FB_RW` je použit pro signalizaci směru transakce, kde log. 1 indikuje čtení. Počet taktů vystavení adresy (Address Hold Time) je nastavitelný parametr. V následujícím taktu po uplynutí vystavení adresy je signál `FB_CS0#` (aktivní v log. 0) použit k vybrání adresované periférie a rovněž k vymezení datové části transakce. Jedná se o jeden ze šesti Chip-Select signálu, což tedy umožňuje připojení až šesti jednotek, nicméně v případě propojení s FPGA byl použit pouze `FB_CS0#`. Čtené zařízení informuje o vystavení dat aktivací signálu `FB_TA#`. Na základě tohoto signálu jsou data v následujícím cyklu přečtena. Ukončení transakce je signalizováno navrácením `FB_CS0#` signálu do neutrální úrovně.

Zápis probíhá zcela analogicky a je znázorněn na obrázku 4.3. Hlavním rozdílem je signál `FB_RW`, který úrovní log. 0 signalizuje zápis dat směrem z MCU. Přijímací strana přečte vystavená data a provede potvrzení aktivací signálu `FB_TA#`. Po potvrzení transakce je podle uživatelského manuálu hodnota vystavených dat nedefinována.

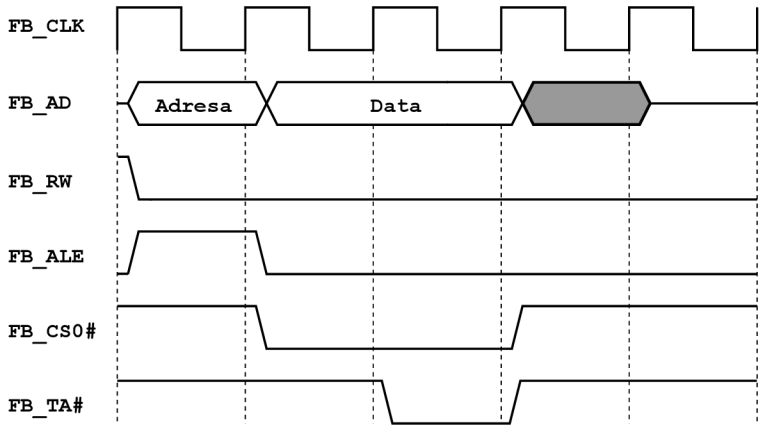


Figure 4.3: Ukázka operace zápisu na sběrnici FlexBUS

Chapter 5

Hradlové pole Spartan-6

Tato kapitola se zabývá popisem programovatelného hradlového pole (FPGA), které je na kitu Minerva použito. Vedle jeho základních parametrů jsou zde popsány části důležité pro možnost programovat FPGA ze strany mikrokontroléru.

5.1 Programovatelné hradlové pole Xilinx Spartan-6

Vývojový kit obsahuje programovatelné hradlové pole od firmy Xilinx řady Spartan-6. Konkrétně se jedná o typ XC6SLX9, což je podle katalogového listu této řady [22] čip s nižším množstvím zdrojů ve srovnání s ostatními obvody Spartan-6. Použité FPGA disponuje 1430 slices, kde každý tento prvek obsahuje čtyři šestivstupé LUT a osm klopných obvodů typu flip-flop. Jejich kombinací lze tedy dosáhnout parametrů 90kb distribuované RAM spolu s 11440 bistabilních KO. Množství dedikované RAM (block RAM) je 32 18kb jednotek. Obvod obsahuje i 16 tzv. DSP Slices obsahujících násobičku, sčítačku a akumulační registr, které jsou tímto cílené na zpracování signálu.

Pro konfiguraci (resp. programování) FPGA je obecně k dispozici několik způsobů, nicméně v našem případě je tato nabídka limitována způsobem, jakým je obvod na kitu zapojen. FPGA není připojeno k MCU žádnými piny, přes které by šlo využít jedno z nabízených sériových či paralelních konfiguračních rozhraní. FPGA je zde nastaveno tak, aby vždy po zapnutí provedlo konfiguraci z externí flash paměti. Jeden ze způsobů, jak měnit konfiguraci FPGA obvodu, je tedy změna obsahu této paměti. Další variantou je rozhraní JTAG. Piny tohoto rozhraní jsou vyvedeny na desce a lze tak použít JTAG adaptér pro programování FPGA. Možnost programování FPGA je jedním z klíčových bodů této práce a tomuto řešení je věnována praktická část.

5.2 Externí flash

Pro nevolatilní uložení konfiguračních dat (bitsreamu) je na kitu použita paměť typu flash, která je připojena k FPGA přes dedikované SPI rozhraní. Kapacita této paměti je 4 Mbity (512 kB) [16]. Podle uživatelské příručky obvodů Spartan-6 je maximální velikost konfigurační sekvence dat 342691 B. Vzhledem k uvedeným velikostem umožňuje kapacita flash paměti uložení pouze jedné konfigurace obvodu. Při generování bitsreamu je možné zvolit použití jednoduché kompresní metody založené na principu RLE (Run-Length Encoding). Tímto lze zmenšit jeho velikost až o 70%. Použité FPGA podporuje tzv. multiboot, kdy je možné provést načtení různých konfigurací z externí flash, nicméně vzhledem k tomu, že

velikost komprese se může v jednotlivých případech značně lišit, je toto využití podstatně limitováno. Komprese dat je tedy výhodná pouze pro zmenšení doby zápisu do paměti.

K rozhraní paměti je připojen i programovatelný USB kontrolér VNC2, který byl původně určen ke změně obsahu této paměti, nicméně přenosová rychlost se ukázala jako nedostatečná (viz. druhá kapitola). Hlavní mikrokontrolér (ARM MCU) bohužel nemá přístup k těmto pinům a pro práci s pamětí je zapotřebí spolupráce s FPGA přes rozhraní FlexBus.

5.3 Interní konfigurační rozhraní ICAP

Řada obvodů Spartan-6 obsahuje interní rozhraní pro přístup ke konfiguračním a stavovým registrům obvodu. Lze přes něj číst aktuální stav či konfiguraci obvodu, případně měnit jeho nastavení. Jedná se o variantu paralelního SelectMAP rozhraní, které je u těchto obvodů určeno ke konfiguraci FPGA z externího zařízení jako je MCU či FPGA. ICAP je s rozhraním SelectMAP kompatibilní ve způsobu komunikace a z velké části i v poskytované funkcionalitě. Nejdůležitějším rozdílem je, že není možné přes ICAP provést konfiguraci FPGA pomocí stejného bitstreamu určeného pro externí flash. ICAP rozhraní dovoluje změnu konfigurace FPGA pouze pomocí částečné rekonfigurace (Partial Reconfiguration), což znamená odlišný a komplikovanější způsob tvorby konfiguračního bitstreamu. Jeho prostřednictvím by tedy bylo možné provádět programování FPGA přímo z MCU, nicméně pouze cestou částečné rekonfigurace.

I přes zmíněné omezení je v tomto případě tato komponenta velice výhodná, jelikož FPGA je na kitu propojeno s MCU pouze přes piny sběrnice FlexBus a MCU tak nemá přístup k žádným z programovacích pinů FPGA obvodu. Například v případě programování externí flash je ICAP důležitý pro provedení resetu FPGA, za účelem načtení nové konfigurace z externí flash paměti. Opeť zde totiž MCU nemá přímý přístup k resetovacímu pinu a ICAP rozhraní se zdá být jedinou možnou cestou, jak provést reset obvodu. Toho lze docílit díky přístupu k řídícím registrům obvodu přes toto rozhraní.

ICAP používá k přenosu dat 16bitovou paralelní sběrnici a několik řídících signálu. Jejich popis je shrnut tabulkou 5.1. Důležitým rysem tohoto rozhraní je pořadí datových bitů, kde nejméně významný bit (lsb) je v každém bajtu na sedmé (poslední) pozici. Toto uspořádání pořadí bitů je ilustrována tabulkou 5.2.

Signál	**Popis**
`CLK`	Clock
`CE`	Clock Enable aktivní v nule
`WRITE`	Zápis/čtení (0/1)
`I[15:0]`	Vstupní data
`O[15:0]`	Výstupní data
`BUSY`	Připravenost dat při čtení

Table 5.1: Použité FlexBUS signály

Nastavení čtení nebo zápisu je provedeno signálem `WRITE`. Jeho změna musí být vždy provedena při neaktivním signálu `CE`. Tento postup změny ze zápisu na čtení je ilustrován obrázkem 5.1. Množiny signálů `I` a `O` jsou zde ilustrovány v jednom řádku.

Jelikož ne vždy je možné dodat nová data během jedné periody hodinového signálu, je

I[15:0]															
8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7

Table 5.2: Pořadí bitů pro rozhraní ICAP

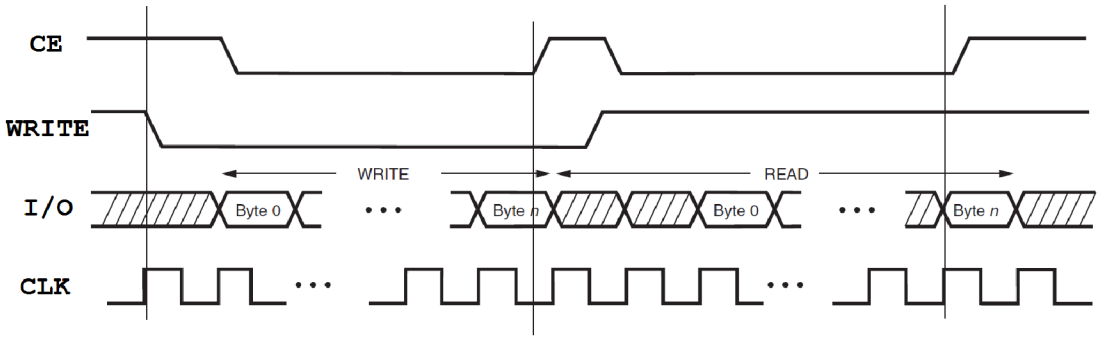


Figure 5.1: Příklad změny signálu `WRITE` ze zápisu na čtení

třeba mít možnost nespojitého zápisu či čtení dat. Jelikož ICAP komponenta potřebuje zdroj hodin i po zápisu všech dat na případné provedení změn v konfiguraci obvodu, je nutné použít signál `CE` (Clock Enable), který bude aktivní pouze když jsou zapisovaná data připravena a vystavena na sběrnici. Princip nespojitého zápisu dat je ilustrován obrázkem 5.2. V bodě 1 je nastaven signál `WRITE` do úrovně signalizující zápis. V bodě 2 jsou k dispozici data k zápisu a signál `CE` se tedy může nastavit do aktivní úrovně. V bodě 4 a 5 jsou data, která byla právě k dispozici navzorkovaná při náběžné hraně hodinového signálu `CLK`. V bodech 6 a 7 se čeká na další vstupní data a signál `CE` je tedy v neaktivní úrovní. Zbytek ilustračního průběhu je analogický.

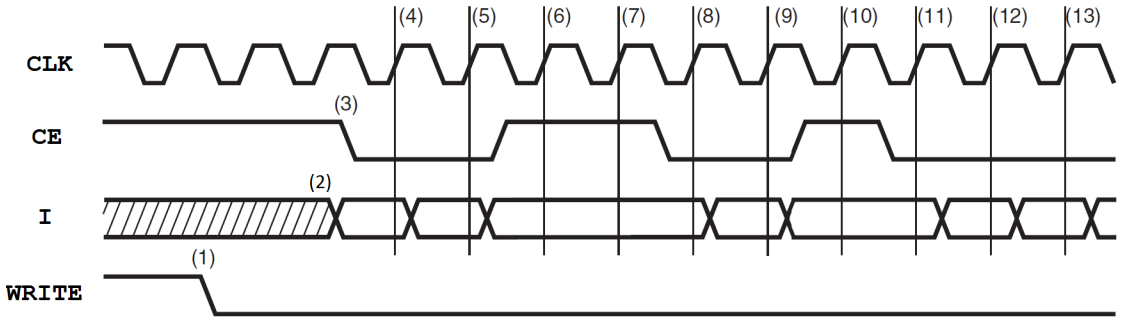


Figure 5.2: Příklad nespojitého zápisu dat do ICAP rozhraní

5.4 Konfigurační pakety

Konfigurace FPGA probíhá na základě vykonávání posloupnosti příkazů, ze kterých je konfigurační bitstream složen. Jejich vykonání spočívá v zápisu nebo čtení konfiguračních registrů obvodu. Konfigurační data jsou složena ze 16bitových slov zapouzdřených do tzv.

konfiguračních paketů. Tento formát dat je použit i pro rozhraní ICAP popsané v předchozí části. Jsou definovány dva typy konfiguračních paketů, jejichž formát se liší v datové šířce pole pro specifikaci velikosti užitečných dat, která daný paket popisuje (payload). První typ slouží pro přenos maximálně 31 16bitových slov, což je dostačující pro změnu obsahu nebo čtení registrů obvodu. Druhý typ je vhodný pro přenos většího počtu dat, tedy zejména pro hlavní konfigurační sekvenci, jelikož je oproti prvnímu typu rozšířen o 32 b pole, ze kterého je použito 28 b pro zakódování velikosti užitečných dat. První typ je dostačující ve všech případech, kdy se nejedná o programování FPGA.

Formát hlavičky konfiguračního paketu typu jedna je uveden v tabulce 5.3. Výčet kódování možných operací je v tabulce 5.4. Velikost užitečných dat je vyjádřena v počtu 16bitových slov. Pole registr specifikuje adresu registru, který se čte nebo modifikuje. Jejích úplný výčet lze nalézt v [24].

Typ [15:13]	**Operace** [12:11]	**Registr** [10:5]	**Počet slov** [4:0]
001	xx	xxxxxx	xxxxx

Table 5.3: Formát hlavičky konfiguračního paketu typ 1

Operace	**Kódování**
NOP	00
READ	01
WRITE	10

Table 5.4: Přehled kódování operací konfiguračního paketu

Příkladem praktického použití je reset FPGA obvodu prostřednictvím rozhraní ICAP. K jeho provedení je třeba provést zápis odpovídající hodnoty do registru `CMD` (Command Register), který leží na adrese `0x05`. Registr `CMD` slouží k řízení konfigurační logiky obvodu, která je řízená množinou příkazů zadávaných přes tento registr. Jedním z této množiny je příkaz `IPROG`, který vykoná restartování obvodu. Kódování této akce prostřednictvím konfiguračních paketů spolu s inicializační sekvencí pro ICAP rozhraní je ilustrováno tabulkou 5.5. První tři zapisovaná slova představují tzv. synchronizační sekvenci, která slouží pro inicializaci nového konfiguračního kontextu. Následuje konfigurační paket, který ve svém záhlaví adresuje registr CMD a ve své datové části kóduje příkaz `IPROG` (`0x0E`). Poslední slovo je paket s NOP operací. Ty jsou při konfiguraci použity k vyplnění stavů, kdy je logika ICAP rozhraní zaneprázdněná zpracováním dat.

0xFFFF	0xAA99	0x5566	0x30A1	0x000E	0x2000

Table 5.5: Restartovací sekvence pro ICAP

Chapter 6

Implementace komunikačních vrstev MCU

V této kapitole je popsána softwarová realizace všech komunikačních vrstev, která jsou v mikrokontroléru zapotřebí. Jedná se o vrstvu USB systému, emulaci FTDI obvodu a prostředků pro komunikaci s FPGA přes rozhraní FlexBus. Veškeré zdrojové kódy byly napsány v jazyce C a pro překlad byl použit kompilátor GNU GCC spolu s jeho nástroji.

6.1 Návrh řešení

Vytvoření komunikačního rozhraní, které by zajišťovalo propojení všech tří stran, tedy PC, FPGA a MCU, je důležitou částí celkové implementace. Diagram 6.1 ilustruje softwarové vrstvy, které je třeba implementovat a jejich vzájemný vztah. Nejspodnější vrstva představuje ovladač USB systému, který tvoří základ pro komunikaci mikrokontroléru s PC. Nad ním je postavena emulace FTDI obvodu, díky které se mikrokontrolér po připojení k PC bude tvářit jako jeden z těchto obvodů a bude možné využít již existující komunikační rozhraní na straně PC. Nad touto vrstvou je postaveno celkové propojení všech tří stran, kde propojení s FPGA je realizováno přes rozhraní FlexBus. Tímto je vyřešena základní komunikace mezi všemi komponentami. Posledním bodem je vytvoření aplikačního rozhraní, které bude zapouzdřovat všechny části vytvořeného systému a poskytovat potřebnou funkcionalitu uživatelským aplikacím. Vytvoření takového aplikačního rozhraní je probráno v samostatné kapitole.

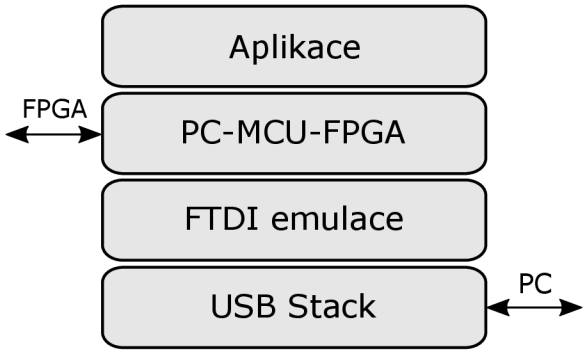


Figure 6.1: Softwarové vrstvy MCU

Důležitým aspektem je volba použití nějaké varianty operačního systému. Ať už se jedná o jednoduchý tasker nebo složitější RTOS, tato volba může významně ovlivnit způsob i výkonnost navrženého řešení. Pro tuto práci byla zvolena cesta bez použití jakékoliv podoby OS, kdy veškeré události jsou zpracovány v kontextu obsluhy přerušení. Výhodou je, že vytvořené řešení nebude vázané na konkrétní implementaci operačního systému. Tato volba taktéž nabízí možnost jednodušší integrace do libovolného OS v případě budoucí potřeby.

Volba frekvence hodinového signálu

Volba hodnoty hlavního hodinového signálu (Master Clock, MC) se odvíjí od použití mikrokontroléru. V našem případě, kdy se jedná o hlavní komponentu vývojového kitu, je vhodné nastavit maximální možnou frekvenci. Mikrokontroléry řady Kinetis K60 mají specifikovanou maximální frekvenci CPU jádra 100 MHz. Frekvence výpočetního jádra je odvozena z MC signálu, stejně jako frekvence hodinového signálu pro ostatní komponenty mikrokontroléru. Obrázek 6.2 je výřez diagramu z uživatelského manuálu, popisující systém pro generování hodinového signálu. Hlavní hodinový signál `MCGOUTCLK` je dále dělen do čtyřech větví, které jsou použity jako zdroj hodin pro všechny komponenty mikrokontroléru.

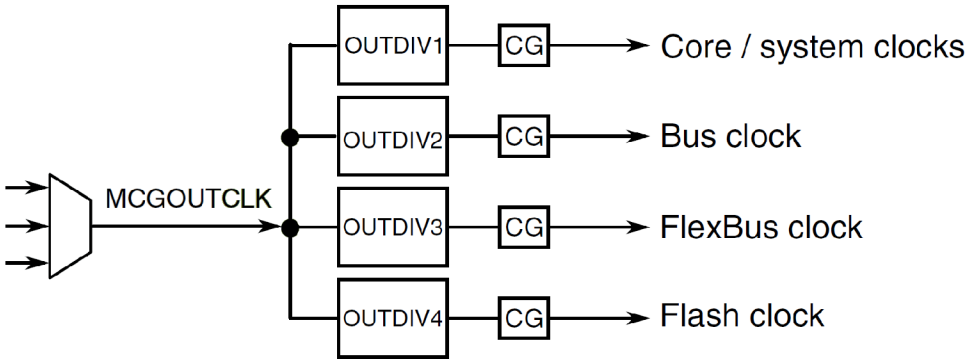


Figure 6.2: Hlavní hodinové signály MCU

Větev Core/System je použita pro taktování CPU a zároveň i USB kontroléru a tedy její hodnota by měla být co nejvyšší a zároveň umožňovat odvození požadované frekvence pro USB kontrolér. USB kontrolér vyžaduje pro svou činnost frekvenci 48 MHz. Na základě tohoto omezení a možností hodinových rozvodů mikrokontroléru jsem zvolil frekvenci 96 MHz.

6.2 Použití USB ovladače

V případě USB ovladače bylo zvoleno použití již hotového řešení od výrobce mikrokontroléru. Ten nabízí USB ovladač Kinetis USB Device Stack, popsaný v kapitole 4. I když se jedná o již hotový produkt, bylo zapotřebí zdrojové kódy nastudovat a integrovat. Jedná se přibližně o 9 tisíc řádků kódu v jazyce C a během vývoje bylo nalezeno i několik chyb, které bylo třeba opravit.

Inicializace USB ovladače

Pokud má naše aplikace běžící na MCU využívat USB, je nutné nejprve inicializovat USB ovladač. Inicializace spočívá ve volání sady funkcí, které se postarají zejména o nastavení registrů USB kontroléru, inicializaci potřebných datových struktur a registraci notifikačních funkcí. Potřebné funkce jsou shrnuty tabulkou 6.1.

`usb_device_init()`	1
`usb_device_register_application_notify()`	2
`usb_device_register_desc_request_notify()`	3
`usb_device_register_vendor_class_request_notify()`	4
`usb_device_postinit()`	5

Table 6.1: Funkce pro základní inicializaci USB komunikace

Funkce `usb_device_init()` provede inicializaci USB kontroléru. Volá se jako první funkce z celého rozhraní, jelikož zde probíhá inicializace základních datových struktur a registrů kontroléru. Následně je třeba provést registraci notifikačních funkcí, které jsou volány během přerušení USB kontroléru za účelem předání popisu události, která toto přerušení vyvolala a jejího případného zpracování. Funkce (**2**) registruje notifikační funkci, která bude přijímat upozornění na události jako je reset před enumerací, dokončení procesu enumerace nebo výskyt chyby. Tato upozornění jsou z pohledu uživatelské aplikace důležitá, jelikož na jejich základě se spouští další fáze ustavení komunikace. Během procesu enumerace musí být uživatelská aplikace informována o tom, o jaké deskriptory má hostitelské zařízení zájem. K registraci funkce, která bude tyto požadavky zpracovávat, slouží volání (**3**). Zaregistrovaná funkce vždy při požadavku na konkrétní deskriptor zařízení obdrží jeho identifikační číslo a předá ukazatel na korespondující datovou strukturu. Funkce (**4**) je použita pro registraci funkce, která bude volána při přijetí dat na endpointu číslo nula, který je implicitně použit pro řídící přenosy (viz kapitola 2.4 o základních principech USB komunikace).

Nakonec je zavolána funkce `usb_device_postinit()`, která provede konečnou fázi inicializace. Ta spočívá zejména v uvedení USB kontroléru do činnosti a povolení přerušení. Jakmile je hostitelským zařízením (PC) detekováno nově připojené zařízení, provede jeho resetování (D+ a D- v nule po dobu delší jak 10 ms), což bude také představovat první notifikační událost.

Inicializace výměny dat

Registrace a konfigurace parametrů jednotlivých endpointů (EP) probíhá v momentě, kdy aplikace obdrží notifikaci o ukončení enumerace. Děje se tak ve funkci, která byla zaregistrována voláním (**2**) z předchozí části. Funkce (**6**) provede inicializaci EP, který je specifikován parametry jako číslo EP, směrem komunikace, typem přenosu a maximální velikostí přenášených dat.

Ke spuštění komunikace na daném EP je po jeho inicializaci ještě potřeba zaregistrovat funkci, která bude volána při dokončení transakce na tomto EP. Při odesílání dat bude zaregistrovaná funkce notifikována o dokončení jejich odeslání a naopak při přijímaní dat bude informována o skutečné velikosti přijatých dat. Tímto způsobem je zajištěna kontrola nad tím, kdy je možné naplánovat další přijetí nebo odeslání dat. Tato registrace se provádí voláním funkce (**7**).

usb_device_init_endpoint()	6
usb_device_register_service()	7
usb_device_send_data()	8
usb_device_recv_data()	9

Table 6.2: Funkce pro inicializaci výměny aplikačních dat

O možnosti vytvořit první datovou transakci je aplikace upozorněna stejným způsobem jako o ukončení enumerace. Po obdržení této události je již možné naplánovat přijetí či odesláni aplikačních dat. K tomu jsou určené funkce (**8**) a (**9**). Nová datová transakce je specifikována číslem EP, na kterém bude probíhat, bufferem pro data a velikostí dat.

6.3 Emulace FTDI obvodu

Implementace FTDI protokolu je realizována nad popsaným USB ovladačem. Je tedy třeba provést jeho inicializaci tak, aby při enumeraci USB zařízení byly předány deskriptory (kapitola 3), které identifikuji mikrokontrolér jako obvod FT232R. Po správné enumeraci následuje inicializace enpointů a zahájení komunikace s ovladačem na straně PC. Jsou vytvořeny tři endpointy, dva pro příjem a odesláni aplikačních dat a jeden pro práci s interní EEPROM, restartování obvodu a konfiguraci jeho převodníku. Tento vztah je znázorněn obrázkem 6.3.

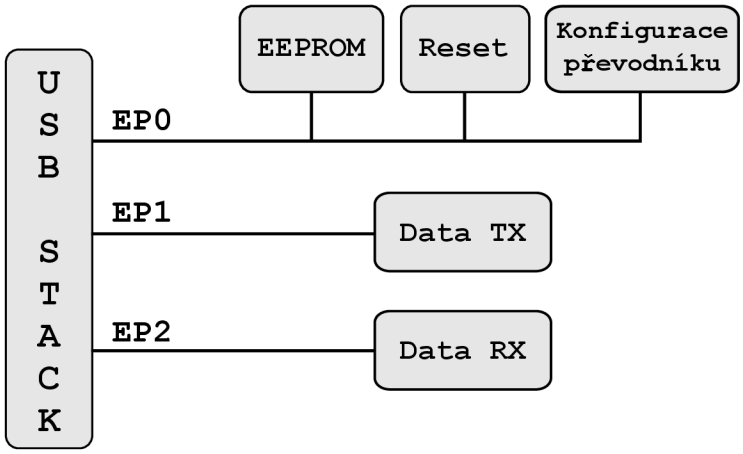


Figure 6.3: Funkce endpointů při emulaci FTDI obvodu

EP0 - konfigurace obvodu

Komunikace na EP0 probíhá pomocí kontrolních přenosů (Control Transfers) a v případě FTDI protokolu jsou použity ke konfiguraci obvodu. Ve fázi inicializace USB zásobníku se notifikační funkce pro EP0 registrovala pomocí volání (**4**). Pro obsluhu příkazů v rámci FTDI protokolu je tedy potřeba, aby tato notifikační funkce zpracovávala veškeré příkazy, které přicházejí od FTDI ovladače na straně PC. Je třeba implementovat obsluhu minimálně takových příkazů, u kterých má ovladač možnost ověření jejich správného vykonání.

V opačném případě by mohl mít ovladač podezření na nefunkční obvod a přerušit komunikaci. Jedná se tedy hlavně o modifikaci EEPROM paměti a nastavení parametrů převodníku, pro které existuje možnost nejen jejich nastavení, ale i čtení jejich aktuální hodnoty.

Příkazy, pro které byla implementována obsluha, jsou uvedeny v tabulce 6.3. Nejdůležitějšími příkazy je reset obvodu a čtení/zápis EEPROM paměti. Ostatní příkazy slouží k obsluze parametrů převodníku a jejich vykonání je podstatné kvůli tomu, že pro některé existuje možnost jejich čtení. Druhým důvodem pro jejich zpracování je to, že v knihovním rozhraní ovladače D2XX existují funkce pro jejich nastavení, které by uživatelská aplikace mohla libovolně použít pro nějaký jiný účel. Například funkce pro nastavení rychlosti převodníku lze použít pro nastavení libovolného parametru uživatelské aplikace. Dalším příkladem může být řízení toku dat mezi PC a MCU pomocí nastavení modemových signálu (RTS, CTS a jiné).

FTDI_RESET_REQUEST
FTDI_READ_EEPROM_REQUEST
FTDI_WRITE_EEPROM_REQUEST
FTDI_SET_LATENCY_TIMER_REQUEST
FTDI_GET_LATENCY_TIMER_REQUEST
FTDI_SET_MODEM_CTRL_REQUEST
FTDI_GET_MODEM_STATUS_REQUEST
FTDI_SET_FLOW_CTRL_REQUEST
FTDI_SET_BAUDRATE_REQUEST
FTDI_SET_DATA_REQUEST

Table 6.3: Implementované příkazy FTDI protokolu

Při zahájení komunikace MCU s PC si nejprve FTDI ovladač vyčte položky EEPROM paměti pro získání dodatečných informací o připojeném obvodu. Z tohoto důvodu je tedy třeba mít konstantní kopii paměti emulovaného obvodu, která bude vždy při této akci použita.

EP2 - příjem dat

Příjem aplikačních dat probíhá na endpointu EP2 pomocí přenosů typu Bulk. FTDI ovladač odesílaná data nekóduje, takže data přijatá na EP2 jsou již ve správném formátu a není zde třeba dalšího zpracování. Velikost bufferu pro příjem dat byla zvolena 256 B. Přijetí nových dat je naplánováno voláním `usb_device_recv_data()`, které vytvoří novou transakci o maximální velikosti dat 256 B. Při přijetí dat je zavolána notifikační funkce, která byla zaregistrována ve fázi inicializace tohoto endpointu. Jejím úkolem je informovat uživatelskou aplikaci o jejich přijetí a naplánovat další transakci opětovným voláním `usb_device_recv_data()`.

Při příjmu je nutné řešit speciální situaci, kdy aplikace na straně PC odešle data o velikosti, která je násobkem 64 B (64 B je maximální velikost jednoho USB paketu u Bulk přenosu). Podle USB specifikace je v tomto případě ukončení přenosu signalizováno pomocí paketu o nulové velikosti užitečných dat (*Zero-Length Packet*, ZL). V případech, kdy není velikost dat násobkem 64 B je přenos dat ukončen v momentě, kdy přišel datový paket o velikosti menší než 64 B. Generování ZL paketu ovšem není v režii USB Host kontroléru,

nýbrž přímo na ovladači zařízení [17]. Ovladač FTDI ovšem v těchto situacích ZL paket negeneruje, jelikož u zařízení jako je USB-UART převodník není členění na jednotlivé datové transakce potřeba (jedná se o jeden spojitý tok dat). Jelikož ZL paket je jediný způsob jak detekovat konec transakce u této délky dat, je potřeba tuto situaci řešit jiným způsobem.

Možným řešením by bylo plánovat transakce pro příjem dat do maximální velikosti 64 B místo požadovaných 256 B. Takto by ale došlo ke snížení propustnosti vlivem větší režie. Například v situaci jako je příjem jedné stránky dat pro naprogramování externí flash paměti FPGA obvodu (velikost stránky 256 B) by bylo potřeba provést čtyři datové transakce a vždy při dokončení jedné z nich manuálně překopírovat příchozí data. USB kontrolér v MCU používá dedikovaný DMA kontrolér pro kopírování příchozích dat a tedy v případě jediné transakce o velikosti 256 B by vše proběhlo bez zásahu procesoru. Jako lepší řešení se ukázal způsob, kdy v případě přenosu dat o této „problematické“ velikosti nejprve PC aplikace informuje MCU aplikaci o této situaci a ta před samotným přenosem nastavení příznak pro kód USB ovladače, že u následující transakce nemá očekávat ZL paket. Obě řešení byla porovnána a zvolené řešení je efektivnější z pohledu implementace i výkonu. Informování probíhá pomocí zápisu do EEPROM paměti (emulované z pohledu MCU) na pozici `90`, kde se zapíše počet celých 64B bloků, ze kterých transakce bude složena. Jelikož přenosy typu Control mají vyšší prioritu než Bulk, je nastavení tohoto příznaku garantováno před samotným přenosem aplikačních dat.

EP1 - odeslání dat

Pro odeslání dat je vyhrazen endpoint EP1. Způsob inicializace je obdobný jako v případě příjmu dat na EP2. Vytvoření transakce pro odeslání je provedeno voláním funkce `usb_device_send_data()`. Po jejich úspěšném odeslání je zavolána notifikační funkce, zaregistrovaná během inicializace, která provede kontrolu přítomnosti dalších dat pro odeslání a podle toho plánuje další odchozí transakci.

Jak již bylo popsáno v kapitole 3, FTDI protokol definuje, že první dva bajty odchozích dat obsahují stavovou informaci.

Skutečnost, že FTDI ovladač očekává v každém příchozím datovém paketu stavovou informaci (viz kapitola 3), má za následek, že FTDI ovladač odstraní všechny 2B slova na pozicích, která jsou násobkem 64 B, jelikož je pokládá za tuto stavovou informaci. V případě 256 B transakce by to tedy byly bajty na pozicích 0, 1, 64, 65, 128, 129, 192, 193. Kvůli tomuto chování je nutné odesílat data z MCU po 64B transakcích, jinak by došlo ke ztrátě dat. Užitečná velikost dat potom činní 62 B v každé odchozí transakci.

Aby měla uživatelská aplikace možnost současného odeslání více jak 62 B v jednom kroku (jedním zavoláním funkce), byla vytvořena varianta vyrovnávacího bufferu, který umožní naplánovat odchozí transakci do velikosti 256 B. Z něj jsou data odesílána do PC po částech o maximální velikosti 62 B, což už ale probíhá mimo režii uživatelské aplikace. Například při čtení externí flash paměti FPGA obvodu, kde stránka má právě velikost 256 B, by práce se 62B úseky mohla působit krkolomně. Aplikací tohoto bufferu dojde ke zjednodušení návrhu aplikací, které budou využívat datového kanálu s PC přes USB. Zároveň se zde plní i funkce vyrovnávací paměti, kdy v případech odesílání menších kousků dat dojde k překrytí naplnění tohoto bufferu a vyřízení předešlé USB transakce.

Pokud nastane situace, kdy je možné odeslat další data, ale žádná nejsou k dispozici (vyrovnávací buffer je prázdný), odešle se pouze samotná stavová informace. Takto má FTDI ovladač přehled o živosti připojeného zařízení.

6.4 Komunikační linka s FPGA pomocí rozhraní FlexBus

Rozhraní FlexBus je použito pro komunikaci s FPGA. FlexBus rozhraní může být provozováno v multiplexovaném 32b, 16b nebo 8b režimu. Z důvodu menšího počtu signálů potřebných pro připojení komponent na straně FPGA, byl zvolen 16bitový režim. Frekvence pro tuto sběrnici byla nakonfigurována na 48 MHz, tedy polovinu hlavního hodinového signálu. Při této frekvenci a datové šířce 16 b se stále jedná o efektivní datový kanál mezi MCU a FPGA. Řešení je navrženo tak, že v případě potřeby, změna ze 16 b na 32 b představuje minimální úpravy kódu.

Adresový prostor mikrokontroléru je navržen tak, že do oblasti od adresy `0x60000000` až po adresu `0xDFFFFFFF` jsou mapovány veškeré operace pro rozhraní FlexBus. Pro naše potřeby byl zvolen úsek `0x60000000` až `0x6FFFFFFF`, který představuje dostatečný rozsah pro adresaci komponent na straně FPGA. FlexBus kontrolér je tedy nastaven tak, aby veškeré operace čtení nebo zápisu v tomto rozsahu byly provedeny prostřednictvím této sběrnice. V kódu se potom jednoduše provádí čtení nebo zápis pomocí dereference ukazatele na požadovanou adresu. Typ ukazatele samozřejmě musí zohledňovat použitou datovou šířku. V tomto případě se tedy jedná o ukazatel na data o velikosti 16 b.

FlexBus umožňuje použití jednoho ze dvou režimů potvrzování konce transakce, kdy je signál sběrnice TA (Transaction Acknowledge) generován buď ze strany FPGA a nebo interně FlexBus kontrolérem po určitém počtu taktů. Při provádění datové transakce je exekuce kódu pozastavena až do příchodu tohoto signálu. Externí potvrzování je zde velice praktické, jelikož ukončení transakce je v režii FPGA a dojde k němu až při dokončení požadovaného výpočtu, který může trvat nedefinovatelnou dobu. Nevýhoda externího potvrzování spočívá v tom, že pokud FPGA neobsahuje konfiguraci s FlexBus řadičem, nikdy nedojde k potvrzení transakce a dojde k zablokování mikrokontroléru na neurčitou dobu (do provedení resetu MCU). Zkušenost během vývoje této práce byla taková, že dostat se do této situace může být velice jednoduché a uvědomit si příčinu značně zdlouhavé. Aby se tedy předešlo zbytečným komplikacím jako je tato, je při spuštění MCU nejprve provedeno nastavení FlexBus kontroléru jako neblokující, následně je provedena kontrola správné konfigurace FPGA a až podle výsledku této kontroly je provedeno požadované blokující nastavení. Kontrola je založena na přečtení identifikace externí flash paměti FPGA obvodu. Výsledek této konfigurace je poznamenán do globální proměnné, aby jej bylo možné v případě potřeby ověřit. Implementace již ale nepočítá s případy, kdy se změní konfigurace FPGA již během činnosti MCU. Pokud nová konfigurace nebude obsahovat rozhraní pro FlexBus, nebo MCU provede transakci v okamžiku, kdy FPGA načítá konfiguraci, dojde k zablokování. Zde se ovšem jedná o obecný problém při komunikaci s externími periférie mi a jeho řešení by znamenalo snížení propustnosti, jelikož by se před každou transakcí muselo provádět ověření správné konfigurace FPGA.

Aby FPGA mělo možnost vyvolat přerušení MCU, byl k tomuto účelu použit pin 28 na portu B. Tento pin je na desce zapojen jako součást FlexBus rozhraní, v roli signálu `OE` (Output Enable). `OS` signál se však v případě připojení pouze jednoho zařízení chová totožně jako signál `CS` (Chip Select) a tedy není potřeba. Pin `B28` byl nastaven s interním pull-up rezistorem a přerušení je tedy citlivé na sestupnou hranu. Použití přerušení se nabízí například v situaci, kdy FPGA provádí výpočet nezávisle na MCU a o dokončení výpočtu informuje pomocí tohoto přerušení.

Chapter 7

Aplikační rozhraní pro komunikaci mezi MCU a PC

Pro použití implementovaného datového kanálu mezi PC a MCU bylo vytvořeno aplikační rozhraní, které poskytne tuto funkcionalitu uživatelským aplikacím jak na straně MCU tak i na straně PC. V této kapitole je popsána jeho implementace a způsob použití. Příklad tvorby aplikace je uveden v samostatné kapitole 9.

7.1 Sériový UART výstup pro ladící účely

Pro ladící účely během vývoje byl použit sériový UART kanál mezi MCU a ladícím rozhraním MCU-DBG (viz 2.1). Toto rozhraní je realizované pomocí dalšího mikrokontroléru, který implementuje ladící OSBDM rozhraní a obsahuje i jeden převodník UART-USB [19]. Tento převodník je propojen s MCU přes jeden z jeho pěti UART kontroléru (UART5).

Rychlost sériového kanálu byla nastavena na 115200 baudů a pro pohodlnější použití byla implementace doplněna o formátovací funkci `printf`. Použitá implementace funkce `printf` má minimální paměťové nároky a zabírá zhruba 2kB kódové paměti. Aby nedošlo při kompilaci kódu v rámci optimalizace nahrazením tohoto volání voláním `puts` ze standardní knihovny (u GCC lze vynutit parametrem `-fno-builtin-printf` [4]), byl vytvořen alias `printf_uart`. Jelikož se jedná o užitečnou funkci, byl tento ladící výstup ponechán jako součást vytvořeného frameworku.

7.2 Aplikační rozhraní na straně MCU

Aplikační rozhraní na straně mikrokontroléru dovoluje uživatelské aplikaci příjem a odeslání dat pomocí USB rozhraní. Tato funkcionalita je založená na již popsaných vrstvách emulace FTDI obvodu a USB ovladače. Jako součást řešení byl vytvořen jednoduchý systém pro směrování příchozích dat, který dovoluje vytvoření logického kanálu pro jednotlivé části aplikace mikrokontroléru. Opět jde o snahu zjednodušit tvorbu aplikací na tomto vývojovém kitu, kdy vytvořením logického kanálu bude uživatelská aplikace notifikováná o příjmu pouze svých dát. Odpadá tedy nutnost řešit podobný problém v případě, kdy je na straně MCU více konzumentů dat. Na tomto principu byly postaveny další části této práce jako je programování FPGA obvodu nebo vytvoření jednoduchého příkazového rozhraní, které budou rozebrány v dalších částech práce.

Příjem dat z PC

Funkce příjmu dat pro konkrétní aplikaci využívá principu tzv. callbacku. Jedná se o notifikační funkci, která je zavolána vždy, když MCU obdrží nová data určená pro konkrétní část aplikace. Callback funkce má dva parametry: velikost příchozích dat a ukazatel na tyto data. Při registraci callbacku je možné nastavit, jestli se má zavolat v kontextu přerušení, tedy přímo z ISR USB systému, nebo nikoliv. Tento způsob volání byl zaveden z důvodu, že pro některé aplikace je výhodné zpracovat data ihned po jejich přijetí ještě v rámci přerušení USB systému. Hlavním důvodem je zamezení vlivu hlavní aplikace mikrokontroléru na systémovou funkcionalitu kitu jako je programování FPGA. Aplikace pro komunikaci s FPGA tak může být o příchozích datech informována přímo z přerušení USB systému a obsloužit příchozí požadavek s prioritou přerušení, která bude pochopitelně vždy vyšší než kontext provádění hlavní aplikace. Pro aplikace, které netvoří část systému, je určené volání jejich callbacku mimo konext přerušení. Slouží k tomu volání `usbio_check()`, které provede kontrolu příchozích dat a zavolání příslušného callbacku. Způsob volání callbacku mimo kontext přerušení je ilustrován obrázkem 7.1.

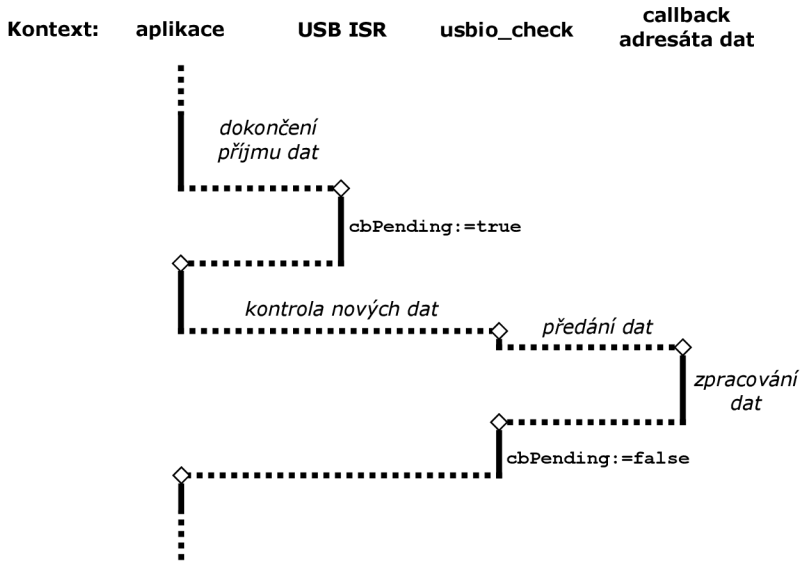


Figure 7.1: Princip volání callbacku mimo kontext přerušení

Volání callbacku adresáta dat v rámci ISR má tu výhodu, že je zde zaručeno zpracování nových dat, před příchodem dat následujících. To je dané tím, že transakce pro nová příchozí data je vytvořena až po návratu z callbacku, kdy již data byla zpracována. Nemůže tedy dojít k jejich přepsání, když jsou data přijímána rychleji než zpracovávána. Další výhoda je minimální latence, kdy není třeba čekat, až z kontextu hlavní aplikace bude zavolána funkce `usbio_check()`. Tento způsob volání je ilustrován obrázkem 7.2. Zpracování dat v kontextu přerušení není ovšem obecně vhodnou variantou jelikož to přináší řadu omezení a potenciálních problému. Proto tedy tento způsob byl použit pouze pro části, které tvoří důležitou součást systému a jejich chod by měl být upřednostněn před hlavní aplikací. Pro všechny ostatní aplikace je určený druhý způsob volání mimo ISR.

Pro zjednodušení komunikace mezi PC a MCU, byla použita i část EEPROM paměti emulovaného FTDI obvodu jako součást komunikačního rozhraní. Pomocí ní se nastavuje aktuální příjemce dat a jsou zde vyhrazeny i tři paměťové položky, které slouží jako dodatečné parametry při vyvolání callbacku. Ty je možné použít pro řízení vytvořené aplikace

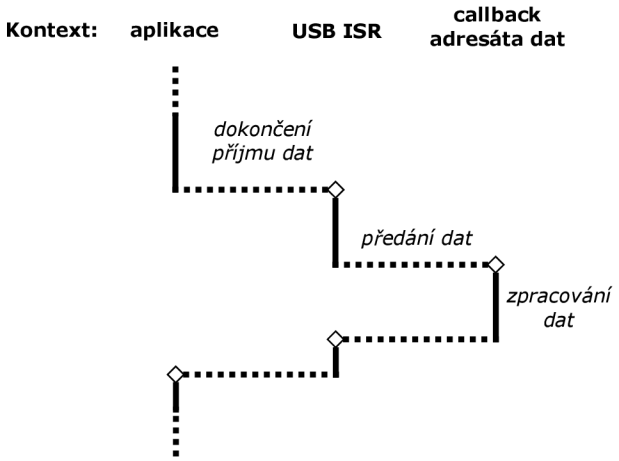


Figure 7.2: Princip volání systémového callbacku v rámci kontextu přerušení

a není tak třeba tuto informaci kódovat spolu s odesílanými daty. Jejich datová šířka je 16 b. Mapování této řídící struktury do virtuální EEPROM je popsáno tabulkou 7.1. Položka `cbPending` na indexu 81 je použita jako příznak pro vyvolání callbacku mimo přerušení. Je mapována do virtuální EEPROM, jelikož zároveň slouží pro synchronizaci odesílání dat z PC. Pokud totiž data nejsou zpracována ihned po jejich přijetí a čeká se na zavolání funkce `usbio_check()`, pro jejich odbavení, musí mít PC aplikace přehled o jejich zpracování. Bez této možnosti by v případě, kdy PC aplikace odesílá spojitý tok dat, mohlo dojít k předčasnému odeslání nových dat a přepsání těch starých.

Položka	Index	Popis
`dest`	80	Aktuální adresát příchozích dat
`cbPending`	81	Příznak čekání na vyvolání callbacku
`p[0]`	82	Pomocný parametr 0
`p[1]`	83	Pomocný parametr 1
`p[2]`	84	Pomocný parametr 2

Table 7.1: Mapování řídících proměnných do virtuální EEPROM

Jelikož v některých případech může být dostačující použit pro komunikaci pouze parametry mapované v EEPROM na indexu 82-84, musí existovat způsob, jak vyvolat callback aplikace bez samotného odeslání dat. Pro tento účel je opět použit příznak `cbPending`. Jeho nastavením ze strany PC aplikace dojde k vyvolání callbacku při následujícím volání `usbio_check()` a nebo ihned po jeho nastavení v případě systémového callbacku spouštěného z kontextu přerušení. Samotný callback může rozpoznat tuto situaci pomocí nulové délky příchozích dat.

Odeslání dat do PC

Způsob odeslání dat je mnohem jednodušší než v případě jejich přijetí. Funkcí `usbio_write()` se provede zápis dat do vyrovnávací pamětí (viz 6.3), ze které se data průběžně odesílají. Jelikož může nastat situace, kdy je tento vyrovnávací buffer již plný, je zápis opakovaně prováděn zhruba po dobu 100 ms a až poté je navrácena chyba. Jedna milisekunda před-

stavuje periodu generování USB rámce a zvolená doba pro opakování zápisu by tedy měla pokrýt stavy, kdy je sběrnice pouze vytížená a nejde o jinou chybu. Jelikož velikost vyrovnávacího bufferu byla nastavena na 256 B, jakékoliv pokusy o zápis většího množství dat budou odmítnuty. V případě potřeby je možné tuto velikost rozšířit bez dodatečných změn v kódu.

Ve spojením s již existující implementací funkce printf byla vytvořena i varinata `printf_usb()`, která odešle svůj naformátovaný výstup po USB rozhraní. Na straně PC je pak možné použít aplikaci terminálu pro zobrazení těchto dat.

7.3 Aplikační rozhraní na straně PC

Aplikační rozhraní na straně PC je založené na knihovním rozhraní FTDI ovladače D2XX. Bylo by možné přímo použít funkce D2XX rozhraní, nicméně bylo by třeba dát si pozor na situace, kdy je třeba synchronizovat odeslání dat. Synchronizace je důležitá, jelikož v některých situacích může dojít k přepsání ještě nezpracovaných dat na straně MCU. Způsob synchronizace je popsán dále v této kapitole včetně příkladu použití. Vytvořené aplikační rozhraní se skládá z několika funkcí, které vznikly zapouzdřením knihovních volání D2XX ovladače a doplněním o zmíněnou synchronizaci. Původní D2XX funkce jsou označeny prefixem `FT_` a jejich upravené varianty jsou označeny prefixem `FTM_`. Přehled vytvořených funkcí je uveden v tabulce 7.2.

Funkce	**Popis**
`FTM_Open()`	Otevření spojení
`FTM_ResetDevice()`	Reset zařízení
`FTM_SetDest()`	Nastavení příjemce dat
`FTM_Write()`	Odeslání dat
`FTM_WriteAsync()`	Odeslání dat bez blokování
`FTM_Read()`	Příjem dat
`FTM_SetParam()`	Nastavení pomocných parametrů
`FTM_GetParam()`	Čtení pomocných parametrů
`FTM_TriggerCallback()`	Explicitní vyvolání callbacku

Table 7.2: Přehled funkcí rozhraní na straně PC

Inicializace

V rámci inicializace je třeba otevřít spojení s emulovaným FTDI obvodem. Funkce `FTM_Open()` je založena na volání `FT_OpenEx()`, které vytvoří spojení s obvodem, který je specifikován sériovým číslem. To bylo u emulovaného obvodu nastaveno na `MINERVA0`. V rámci inicializace je třeba provést i reset logiky FTDI obvodu, kde dojde zejména k vyprázdnění bufferů a vynulování řídících proměnných. Jedná se o krok, který je analogický k práci s realným FTDI obvodem. Funkce `FTM_ResetDevice()` je pouze přejmenování původní funkce `FT_ResetDevice()`.

Nastavení příjemce dat

K nastavení příjemce dat slouží funkce `FTM_SetDest()`. Pomocí tohoto volání se specifikuje, který callback se bude na straně mikrokontroléru volat pro zpracování příchozích dat. Jedná se o zapouzdření funkce `FT_WriteEE()`, která je použita pro zápis do EEPROM paměti FTDI obvodu. V tomto případě jde o zápis na pozici 80, kde je mapována řídící proměnná, která obsahuje hodnotu právě aktivního callbacku.

Výčet destinací je dán aktuální konfigurací na straně mikrokontroléru, kde seznam všech použitých callbacků je reprezentován konstantním polem. Jde tedy o statickou konfiguraci a aktuální výčet destinací je reprezentován pomocí enumerátoru v hlavičkovém souboru aplikačního rozhraní. Aktuální stav je znázorněn tabulkou 7.3.

Příjemce	**ID**	**Popis**
`DEST_TCMD`	0	Příkazové rozhraní
`DEST_FPGA_FLASH`	1	Konfigurační paměť FPGA
`DEST_FPGA_JTAG`	2	JTAG rozhraní FPGA
`DEST_D2XX`	3	Výpis přijatých dat na ladící UART rozhraní
`DEST_MD5`	4	Demonstrační aplikace pro výpočet MD5

Table 7.3: Výčet vytvořených příjemců dat

Většina z uvedených destinací bude popsána dále v této práci. Destinace `DEST_TCMD` je použita pro příkazové rozhraní, které je popsáno v závěru této kapitoly. Další dvě destinace jsou určené pro programování FPGA obvodu, popsáného v kapitole 10. Destinace `DEST_D2XX` byla ponechána jako ladící prostředek, který pouze vypíše přijatá data pomocí volání `printf_uart()`. Destinace `DEST_MD5` vznikla pro demonstraci použití komunikace mezi MCU a PC, které je prezentováno v kapitole 9.

Příjem a odeslání dat

Pro odeslání dat je určená funkce `FTM_Write()`. Jedná se o upravené volání D2XX funkce `FT_Write()`, kde je ošetřená situace s odesláním dat, jejichž velikost je násobkem 64 B. Zároveň je volání doplněno o zmíněnou synchronizaci, kdy dojde k návratu z této funkce až po dokončení provádění callbacku na straně MCU. Tím je zaručeno, že opětovným voláním funkce `FTM_Write()` v době, kdy ještě nejsou zpracována data předchozí, nedojde k jejich přepsání. Maximální velikost odesílaných dat by měla být stejná jako velikost bufferu pro příchozí data na straně MCU, tedy 256 B. Tato velikost je v hlavičkovém souboru rozhraní definována makrem `FTM_MAX_SEND_SIZE`.

Pro příjem dat je určená funkce `FTM_Read()`. Jde pouze o přejmenování původní funkce `FT_Read()`, jelikož zde není třeba žádných modifikací. Hlavními parametry této funkce jsou velikost přijímaných dat a ukazatel na buffer, kam budou přijatá data překopírována. Toto volání je blokující a k návratu dojde až po přijetí požadovaných dat nebo výskytu chyby.

V případech, kdy operaci zápisu bezprostředně následuje operace čtení, je blokující charakter funkce `FTM_Read()` nevýhodný. Proto byla vytvořena varianta `FTM_WriteAsync()`, která v rámci zápisu neobsahuje čekání na dokončení callback funkce. Blokující chování (resp. synchronizace) je založeno na opakovaném čtení příznaku `cbPending` z virtuální EEPROM a tudíž jeho vynecháním v případech, kde není potřeba, může urychlit přenos užitečných dat. Experimentálně však bylo zjištěno, že nejde ani o 10% nárůst přenosové

rychlosti, a tak je z praktického hlediska lepší ve všech situacích použít volání `FTM_Write()` a předejít tak potenciálním chybám.

Použití pomocných parametrů

Pro nastavení parametrů mapovaných do virtuální EEPROM slouží funkce `FTM_SetParam()`. Hlavními argumenty je číslo parametru (0–2) a přiřazovaná hodnota. Pro čtení těchto parametrů je určená funkce `FTM_GetParam()`, která funguje analogicky. V případech, kdy veškerá komunikace s MCU může být založena pouze na těchto třech parametrech a funkce `FTM_Write()` není potřeba, musí se zajistit vyvolání callbacku manuálně. K tomuto slouží funkce `FTM_TriggerCallback()`, která nastaví příznak `cbPending` do aktivní hodnoty a počká na jeho vynulování. Samotný callback může detekovat manuální vyvolání pomocí nulové velikosti příchozích dat. Použití této techniky je demonstrováno v kapitole 9.

7.4 Příkazové rozhraní implementované v MCU

Jako součást vyvíjeného frameworku bylo vytvořeno i příkazové rozhraní, které umožní komunikaci mezi PC a aplikací běžící v mikrokontroléru prostřednictvím sériového terminálu. Zde byla využita dostupnost VCP (Virtual COM Port) ovladače na straně PC, který pro komunikaci s FTDI obvody poskytuje virtuální sériový port. Prostřednictvím terminálu je tedy možné ovládat aplikaci předdefinovanými příkazy. Použití je ilustrováno obrázkem 7.3, na kterém je terminálová aplikace Putty, která je připojená na virtuální COM port přiřazený emulovanému FTDI obvodu v mikrokontroléru. Pro demonstraci použití byly vytvořeny jednoduché příkazy. Příkaz `LED` změní stav LED diody na desce kitu a příkazem `FLASH` dojde k přečtení první stránky flash paměti FPGA obvodu.

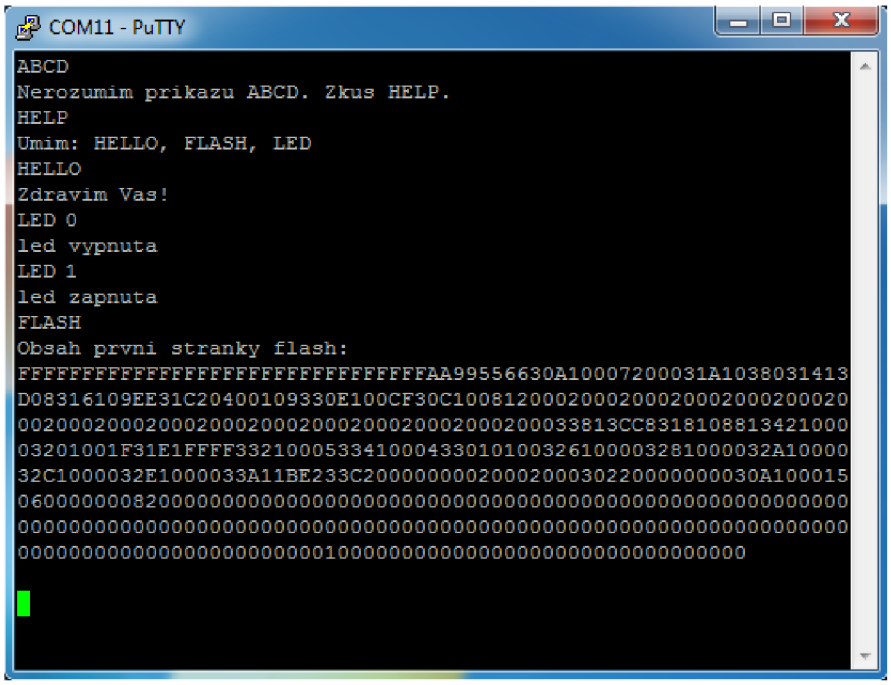


Figure 7.3: Ilustrace použití příkazového rozhraní

Na straně PC nebylo potřeba vyvíjet jakýkoliv dodatečný software, jelikož vše je součástí libovolné terminálové aplikace, podporující komunikaci přes sériový port, a VCP ovladače. Nastavení parametrů virtuálního COM portu může být libovolné, jelikož nikde nedochází k jejich odeslání po reálném sériovém rozhraní (např. UART či RS-232). Na straně MCU byla vytvořena aplikace přijímající data na logickém kanálu `DEST_TCMD`. Na obrázku 9.1 v předešlé části si lze všimnout, že hodnota `DEST_DEFAULT` je rovna `DEST_TCMD`. Při resetu emulovaného FTDI obvodu se pro nastavění aktuálního příjemce dat nastavuje právě tato hodnota. To je z důvodu, že při použití VCP ovladače nemá uživatel možnost měnit příjemce dat a proto se musí jednat o implicitní konfiguraci.

Callback obsluhy terminálu zpracovává příchozí znaky způsobem, kdy veškeré tisknutelné znaky včetně mezery přijme a zbylé odmítne. Mazání již přijatých znaku je provedeno přijetím znaku `DEL` (ASCII `0x7F`), který je typicky generován klávesou `Backspace`. Potvrzení je pomocí znaku `CR` (Carriage Return, ASCII `0x0D`), který je typicky generován klávesou `Enter`. Maximální délka jednoho příkazu je 31 znaků. Všechny přijaté znaky jsou odeslány zpět pro jejich zobrazení na terminálu a následně nahrazeny výstupem standardního volání `toupper()`, které převede malá písmena na velká. Kanál se tedy chová jako standardní terminál a lze se k němu připojit pomocí libovolné terminálové aplikace.

Aplikace mikrokontroléru provádí kontrolu nově příchozího příkazu voláním knihovní funkce `usbio_check()` (viz část 7.2). Pokud došlo k přijetí nového příkazu, je v rámci volání `usbio_check()` zavolána funkce `termcmd_user_callback()` s parametry délky příkazu a ukazatelem na jeho řetězec. Základní implementace funkce `termcmd_user_callback()`, která pouze vypíše přijatý příkaz zpět na terminál, je již v kódu implementována jako tzv. *weak* funkce, což znamená, že pokud ji uživatel nedefinuje ve své aplikaci, použije se tato implementace [4].

7.5 Paměťové nároky implementace

Paměťové nároky vytvořeného řešení jsou uvedeny v tabulce 7.4. Jedná se velikost základního systému, který obsahuje všechny komponenty popsané v této práci, kromě demonstračních aplikací. Součet velikostí `text` a `data` představuje vyžadované množství kódové paměti. Sekce `bss` představuje minimální nároky na RAM. Použité MCU disponuje 512 kB ROM a 128 kB RAM, což představuje paměťové nároky 5,5% na ROM a 2,8% na RAM.

text	data	bss
26692	2248	3640

Table 7.4: Paměťové nároky firmwaru MCU

Chapter 8

Implementace komponent na straně FPGA

V této kapitole je popsán vytvořený FlexBus kontrolér pro propojení FPGA obvodu s MCU, který byl vytvořen tak, aby umožňoval připojení libovolného počtu HW komponent, které vyžadují komunikaci s MCU. Pro potřeby programování FPGA byly vytvořeny komponenty SPI a ICAP rozhraní, pomocí kterých MCU implementuje proces programování FPGA. Jako implementační jazyk byl použit jazyk VHDL spolu s vývojovým prostředím Xilinx ISE 14.7.

8.1 FlexBus kontrolér

Rozhraní vytvořené komponenty FlexBus kontroléru je uvedeno na obrázku 8.1. Jejím úkolem je překlad transakcí na sběrnici FlexBus do takové formy, která umožňuje připojeným komponentám použít jednodušší logiku pro řízení výměny dat. Cílem je zjednodušení návrhu nových komponent pro FPGA a zároveň šetření zdrojů, neboť bez použití FlexBus kontroléru by musely všechny připojené komponenty implementovat stejnou logiku, jako tento kontrolér.

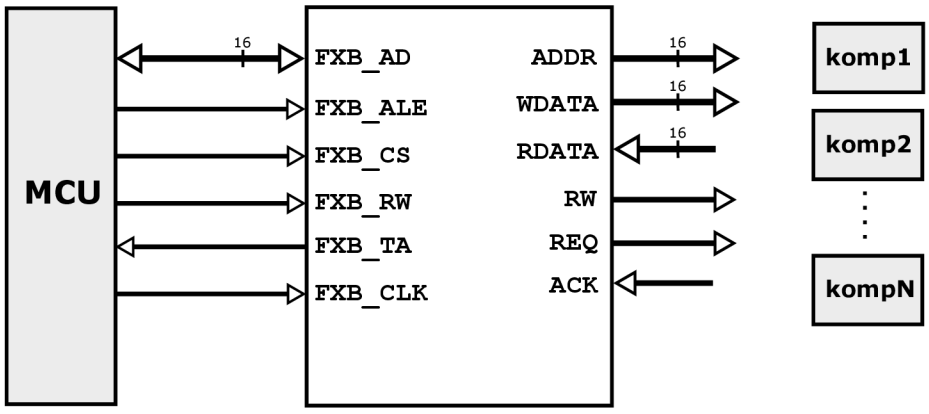


Figure 8.1: Komponenta FlexBus kontroléru

Všechny signály s prefixem `FXB_` jsou součástí FlexBus rozhraní a jsou přímo připojeny k pinům mikrokontroléru. Signály na pravé straně jsou určeny k paralelnímu připojení komponent komunikujících s MCU. Jejich význam je uveden v tabulce 8.1. Podstatným

rozdílem oproti FlexBus rozhraní je, že signály pro adresu a data již nejsou sdílené. Jakmile je detekována nová transakce, její adresa je vystavena na sběrnici `ADDR`, kde zůstane až do ukončení transakce. Signály `WDATA` a `RDATA` jsou určeny pro příchozí a odchozí data. Jejich pojmenování je podle transakce čtení/zápisu na straně MCU. Na signály `WDATA` jsou vystavena zapisovaná data v případě zápisu dat směrem z MCU do FPGA. Na signály `RDATA` jsou vystavena čtená data v případě čtení dat z FPGA do MCU. Veškerá komunikace je iniciovaná ze strany MCU a proto tedy toto rozlišení.

Signál	**Směr**	**Popis**
`ADDR[15:0]`	`OUT`	Adresa transakce
`WDATA[15:0]`	`OUT`	Zapisovaná data směrem MCU→FPGA
`RDATA[15:0]`	`IN`	Čtená data směrem MCU←FPGA
`RW`	`OUT`	Signalizace čtení `1`, zápis `0`
`REQ`	`OUT`	Signalizace nové transakce, aktivní v `1`
`ACK`	`IN`	Signalizace odbavení transakce, aktivní v `0`

Table 8.1: Výčet signálů FlexBus kontroléru

Signalizace nové transakce je realizována pomocí signálu `REQ`. Ten zůstává aktivní až do jejího dokončení. Dokončení transakce je signalizováno signálem `ACK` obsluhující periférií. Při zápisu dat je signál `ACK` aktivován v momentě, kdy adresovaná komponenta data ze sběrnice `WDATA` převzala a již dále nemusí blokovat celé FlexBus rozhraní. Při čtení je signál `ACK` nastaven v momentě, kdy již adresovaná komponenta vystavila požadovaná data na `RDATA` sběrnici a tímto ukončuje operaci čtení. Operace zápisu a čtení je rozlišena signálem `RW`. Po dobu od signalizace nové transakce signálem `REQ` až po dobu signalizace jejího dokončení prostřednictvím signálu `ACK` je celé rozhraní blokováno adresovanou komponentou.

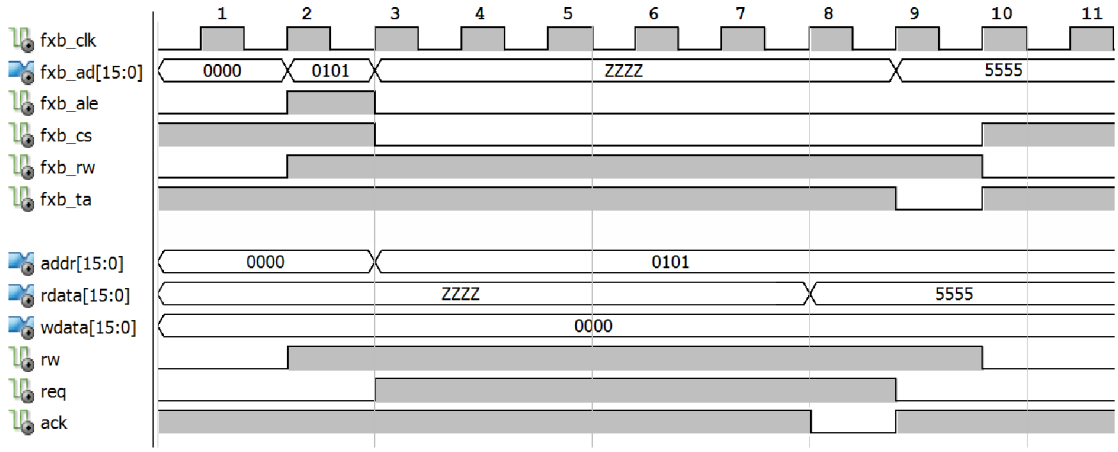


Figure 8.2: Výstup simulace FlexBus kontroléru při operaci čtení

Na obrázku 8.2 je výstup simulace FlexBus kontroléru při zpracování čtecí transakce. V horní části jsou signály rozhraní FlexBus a ve spodní části jsou signály pro signalizaci parametrů transakce a její řízení. V prvním hodinovém taktu neprobíhá na sběrnici žádná aktivita. Nová čtecí transakce je nejprve zahájena vystavením adresy `0x0101` na signálech `FXB_AD` a aktivací signálu `FXB_ALE`. Čtení je ve stejném bodě signalizováno pomocí `FXB_RW`,

který je přímo propojen s výstupem RW. Pří následující náběžné hraně (č. 3) hodinového signálu FXB_CLK je adresa vystavena na výstup ADDR, kde zůstane až do příchodu další transakce. Signály FXB_AD jsou nyní ve stavu vysoké impedance (Z), jelikož se na nich očekává vystavení čtených dat. Ve stejném momentě dochází k signalizaci nové transakce FlexBus kontrolérem, aktivací signálu REQ. Od tohoto momentu je zpracování transakce v režii adresované komponenty, která po vystavení čtených dat potvrdí konec transakce aktivací signálu ACK. V tomto případě zpracování čtených dat 0x5555 trvalo čtyři takty signálu FXB_CLK a v osmém hodinovém taktu dochází k jejich vystavení a potvrzení signálem ACK. V následujícím hodinovém taktu jsou data FlexBus kontrolérem vystavena na piny MCU a potvrzena aktivací signálu FXB_TA. Ukončení transakce je potvrzeno deaktivací FXB_CS.

8.2 Adresace připojených komponent

Každá komponenta implementující komunikaci s MCU je paralelně připojena k rozhraní FlexBus kontroléru. V její režii je při aktivaci signálu REQ rozpoznat svou adresu na sběrnici ADDR, provést zpracování transakce a informovat kontrolér o konci zpracování signálem ACK. Jedná se o jednoduché stavové řízení, které je méně náročné na zdroje, než v případě kdy by byly komunikující komponenty připojeny k rozhraní FlexBus přímo. Jelikož rozpoznání adresy je v režii adresované komponenty, je třeba dát si pozor na situace, kdy je adresována neexistující komponenta. V takovém případě by nikdy nedošlo k potvrzení transakce signálem ACK, což by mělo za následek zamrznutí běhu mikrokontroléru.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
adresa komponenty								položka komponenty							0

Table 8.2: Využití adresy

Z původní 32b adresy bylo použito 16 nejspodnějších bitů, jelikož použití plné délky by bylo ve většině případech zbytečné. Těchto 16 bitů je dále rozděleno tak, že horních osm bitů je určených pro adresaci komponent a spodních osm bitů pro adresaci v rámci komponenty. Při této konfiguraci tedy může být k FlexBus kontroléru připojených až 256 komponent. V rámci nich by bylo možné adresovat až 256 položek, nicméně mikrokontrolér neumožňuje 16b čtení/zápis na lichých pozicích prostřednictvím FlexBus rozhraní, následkem čehož se počet adresovatelných položek uvnitř jedné komponenty redukuje na 128. Využití adresy je ilustrováno tabulkou 8.2.

8.3 Komponenta SPI rozhraní

Pro přístup k externí flash paměti FPGA obvodu prostřednictvím mikrokontroléru byl vytvořen jednoduchý SPI kontrolér. Veškerá logika tohoto obvodu je založena na posuvném registru, pomocí kterého se převádí paralelní data na sériová a naopak. Datová šířka je v tomto případě 8 b. Zapojení této komponenty je znázorněno na obrázku 8.3, který celkově popisuje vytvořený systém uvnitř FPGA. Blok FXBC představuje FlexBus kontrolér.

Z pohledu MCU lze adresovat dvě funkce SPI komponenty, pomocí kterých se řídí veškerá komunikace s flash pamětí (FLASH). Na adrese 0x0000 lze číst nebo měnit stav signálu CS (Chip Select), který je součástí rozhraní připojené paměti. Manipulace s tímto

signálem byla ponechána v režii MCU, jelikož změna jeho stavu závisí na kontextu zpracování a implementace kontroléru je tímto jednodušší. Odeslání dat na SPI rozhraní se provádí zápisem na adresu `0x0002`, kdy dojde k serializaci zapisovaného bajtu a zároveň k přijetí jednoho bajtu směrem z paměti, což reflektuje princip SPI komunikace. Přijatá data jsou po ukončení zápisu uložena v posuvném registru, odkud je lze přečíst čtecí transakcí na stejné adrese `0x0002`. Pro čtení dat je tedy potřeba vždy nějaká data zapsat, což reflektuje princip SPI komunikace. Frekvence zápisu (resp. čtení) dat na SPI rozhraní je polovina hlavního hodinového signálu `FXB_CLK`, tedy 24 MHz. Tato hodnota představuje za daných podmínek maximální možnou frekvenci, jelikož uživatelský manuál použité flash paměti udává maximální frekvenci 25 MHz [16].

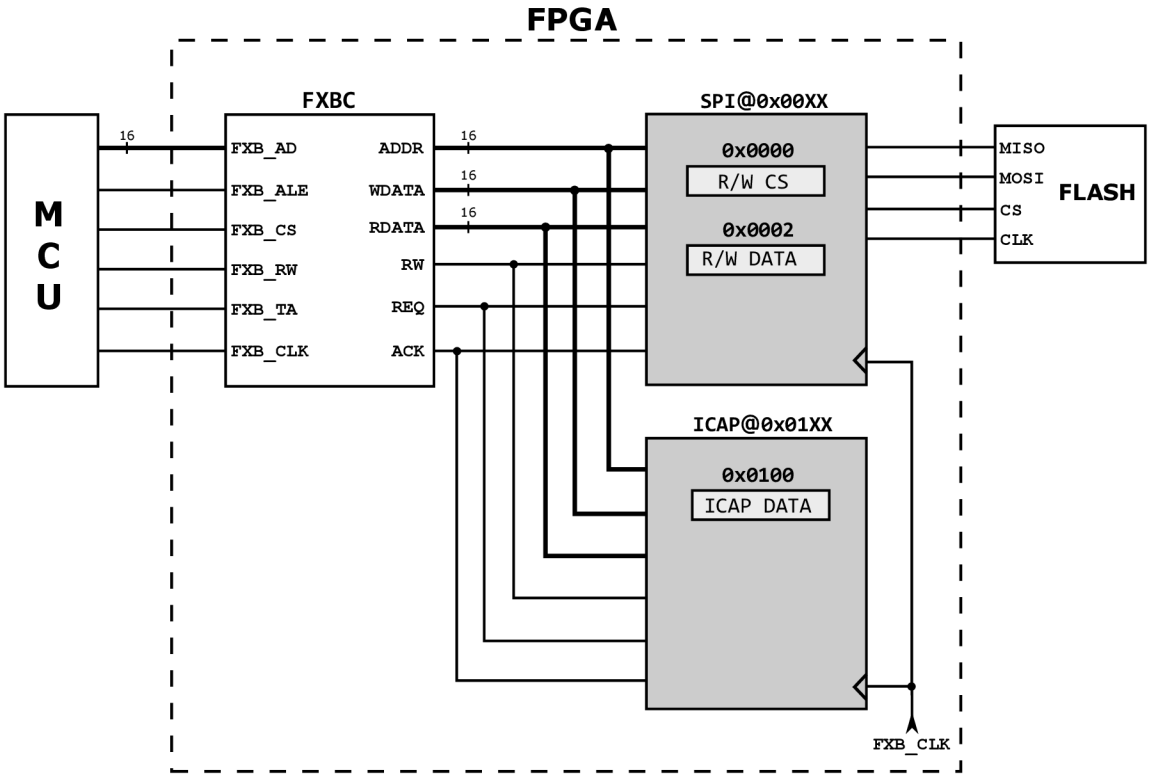


Figure 8.3: Grafické znázornění vytvořeného systému v FPGA

Jelikož se jedná o jednoduchý princip implementace rozhraní mezí MCU a FLASH, probíhá celá komunikace na úrovni jednotlivých bajtů. To samozřejmě zjednodušuje realizaci, ale také degraduje výkon, který je snížen režií generování jednotlivých FlexBus transakcí. Nicméně implementované rozhraní mezi MCU a flash pamětí vzniklo pro účely programování paměti směrem z PC, kde USB kanál s teoretickou propustností 12 Mbps představuje mnohem užší místo v komunikaci.

Přístup k flash paměti je z MCU realizován jako zápis/čtení dat na dvou konkretních adresách. V případě signálu CS jde o adresu `0x60000000` a v případě dat jde o `0x60000002`. Úsek kódu, který přečte stavový registr (SR) flash paměti je znázorněn na obrázku 8.4. Ukazatele `spi_cs` a `spi_data` představují zmíněné adresové pozice. Signál `CS` je aktivní v logické nule a veškerá komunikace začíná jeho aktivací a končí jeho deaktivací. Příkaz pro čtení stavového registru je `0x05`. Modul flash paměti po jeho přijetí vrátí v následujícím zápisu obsah stavového registru. Zápisem libovolných dat (zde `0xFF`) dojde k přenosu

obsahu SR směrem FLASH→FPGA. Přenos obsahu SR směrem FPGA→MCU je realizován čtením na pozici `spi_data`.

```
*spi_cs = 0;
*spi_data = 0x05;
*spi_data = 0xFF;
sr = *spi_data;
*spi_cs = 1;
```

Figure 8.4: Ilustrace komunikace s flash pamětí na straně MCU

8.4 Komponenta ICAP rozhraní

Komponenta pro komunikaci mikrokontroléru s ICAP rozhraním byla vytvořena zejména kvůli potřebě restartovat FPGA obvod, jak již bylo popsáno v části 5.3. Této jednotce byla přiřazena adresa `0x0100`. Při zápisu na tuto adresu dojde k vystavení 16b dat na rozhraní ICAP a provedení zapisovacího cyklu. Provedení resetu FPGA mikrokontrolérem se tedy provede postupným zápisem restartovací sekvence na adresu `0x60000100`, jak je ilustrováno na obrázku 8.5.

```
void fpga_reset(void)
{
    *icap_data = 0xFFFF;    // dummy
    *icap_data = 0xAA99;    // sync
    *icap_data = 0x5566;    // sync
    *icap_data = 0x30A1;    // CMD reg
    *icap_data = 0x000E;    // IPROG
    *icap_data = 0x2000;    // NOOP
}
```

Figure 8.5: Způsob restartování FPGA na straně MCU

8.5 Nároky implementace na zdroje FPGA

Využití nejdůležitějších zdrojů FPGA obvodu je uvedeno v tabulce 8.3. Z uvedených údajů vylívá, že nároky vytvořeného řešení jsou minimální.

Slice Registers	87 z 11400 (1%)
Slice LUTs	75 z 5720 (1%)

Table 8.3: Nároky implementace na zdroje FPGA obvodu

Chapter 9

Příklad tvorby aplikace

Pro demonstraci použití aplikačního rozhraní na straně PC a MCU byla vytvořena aplikace pro výpočet kontrolního součtu MD5. Ta funguje tak, že z PC jsou postupně odesílána data do MCU, kde probíhá výpočet kontrolního součtu a na konci přenosu je výsledek předán zpět do PC. Dalším příkladem je připojení vlastní HW komponenty, implementované v FPGA, do systému FlexBus. Pro tento účel byla vybrána akcelerace šifrovacího algoritmu DES. Kompletní zdrojové kódy obou aplikací jsou součástí realizačního výstupu této práce.

9.1 Implementace MD5 aplikace na straně MCU

Realizace na straně mikrokontroléru spočívá ve vytvoření implementace MD5 algoritmu a vytvoření rozhraní pro interakci s PC aplikací. Pro implementaci MD5 algoritmu byla použita jedna z jeho volně šiřitelných (open-source) variant.

```
typedef enum
{
    DEST_TCMD,
    DEST_FPGA_FLASH,
    DEST_FPGA_JTAG,
    DEST_D2XX,
    DEST_MD5,

    DEST_COUNT,
    DEST_DEFAULT = DEST_TCMD
} UIO_Dest_t;
```

Figure 9.1: Enumerátor datových destinací v souboru `usbio_minerva.h`

Základem implementace je vytvoření nového příjemce dat a registrace jeho callbacku. Hlavičkový soubor `usbio_minerva.h` obsahuje definici enumerátoru, který obsahuje výčet všech destinací. Pro vytvoření nové destinace tedy pouze stačí rozšířit tento výčet o další. Obrázek 9.1 ilustruje stav tohoto enumerátoru po vytvoření nové destinace `DEST_MD5`. Registrace callbacku se provede v souboru `usbio_control.c` vytvořením nového záznamu v globálním poli `usbioCallbacks` (hlavičkový soubor `usbio_minerva.h` je použit jak pro zdrojové kódy na straně MCU tak i na straně PC a proto má jiné pojmenování). Definice pole `usbioCallbacks` po přidání MD5 callbacku je ilustrována na obrázku 9.2. Příznak

`false` vedle ukazatele na funkci `md5_callback()` specifikuje, že tento callback bude volán mimo kontext přerušení.

```
UIO_Callback_t usbioCallbacks[DEST_COUNT] =
{
    //--- system callbacks ---//

    {termcmd_callback, true},   // DEST_TCMD
    {flashprog_callback, true}, // DEST_FPGA_FLASH
    {xvc_callback, true},       // DEST_FPGA_JTAG

    //--- user callbacks ---//

    {d2xx_callback, false},     // DEST_D2XX
    {md5_callback, false}       // DEST_MD5

};
```

Figure 9.2: Struktura `usbioCallbacks` v souboru `usbio_control.c`

Definice callbacku `md5_callback()` je uvedena na obrázku **??**. Funkce s prefixem `MD5_` jsou součástí použité implementace kontrolního MD5 součtu. Tělo callbacku je tvořeno příkazem `switch`, který podle hodnoty pomocného parametru `p0` vybere akci, pro kterou byl callback zavolán. Pomocné parametry jsou součástí globální struktury `usbioControl`,

```
void md5_callback(uint16_t len, uint8_t *data)
{
    uint8_t res[16];

    switch (usbioControl.p[0])
    {
    case MD5_CYCLE
       MD5_Update(&ctx, data, len);
       break;

    case MD5_RESULT:
       MD5_Final(res, &ctx);
       usbio_write(16, res);
       break;

    case MD5_INIT:
    default:
       MD5_Init(&ctx);
    }
}
```

Figure 9.3: Definice funkce `md5_callback()`

která svůj obsah mapuje do virutální EEPROM emulovaného FTDI obvodu. Bez použití pomocných parametrů by se musela konkrétní akce kódovat spolu s daty. Akce `MD5_INIT` provede inicializaci kontrolní struktury `ctx`, která udržuje kontext výpočtu MD5 součtu. Akce `MD5_RESULT` je použita pro ukončení výpočtu a odeslání výsledku zpět do PC. Nakonec akce `MD5_CYCLE` provede samotný výpočet kontrolního součtu, kdy příchozí data, která přicházejí po kouscích o maximální velikosti 256 B, jsou zpracována voláním `MD5_Update()`. Velikost příchozích dat je dána parametrem `len` a jejich umístění v paměti parametrem `data`.

Nesmíme zapomenout na volání `usbio_check()`, které by mělo být periodicky voláno pro kontrolu nově příchozích dat. Vhodným umístěním je například na konci funkce `main()` v rámci nekonečného cyklu.

9.2 Implementace MD5 aplikace na straně PC

Aplikace na straně PC má v režii vytvoření spojení a řízení průběhu výpočtu. Výpočet MD5 součtu se skládá z inicializace, odeslání dat a přijetí výsledku. Kód, který je zde uveden obsahuje pouze volání funkcí, které jsou relevantní pro tento výklad a neobsahuje kontrolu výskytu chyb a ostatní části aplikace.

```
FTM_HANDLE device;

// Initialization
FTM_Open(&device);
FTM_ResetDevice(device);

// Select destination
FTM_SetDest(device, DEST_MD5);

// Initialize new MD5 computation
FTM_SetParam(device, 0, MD5_INIT);
FTM_TriggerCallback(device);
```

Figure 9.4: Inicializační část demonstrační PC aplikace

Část inicializace se skládá z otevření spojení a následného resetu zařízení. Část tohoto úseku kódu je na obrázku 9.3. Samotné zařízení je definováno řídicí strukturou `device` typu `FTM_HANDLE`, která je inicializována voláním `FTM_Open()` a dále použita u všech ostatních `FTM_` volání. Před prvním odesláním dat je třeba zvolit jejich příjemce voláním `FTM_SetDest()`, kde je použita hodnota `DEST_MD5`, definovaná v předchozí části. Těmito kroky je nastavená cesta pro data a je možné provést inicializaci MD5 výpočtu. Jak je patrné z implementace callbacku `md5_callback()` na straně MCU, používá k výběru požadované akce pomocný parametr číslo nula. Funkcí `FTM_SetParam()` se tedy nastaví provedení inicializace výpočtu a následuje manuální spuštění callbacku prostřednictvím `FTM_TriggerCallback()`.

Na obrázku 9.4 je zobrazena část kódu pro odeslání obsahu souboru, ze kterého se počítá kontrolní součet. Před zahájením odesílání je třeba změnit akci callbacku na `MD5_CYCLE`, která představuje aktualizaci kontrolního součtu vlivem nově příchozích dat. V tomto bodě je provedeno samotné odeslání dat po částech o maximální velikosti `FTM_MAX_SEND_SIZE`

(256 B). Jelikož se jedná o kontinuální zápis, který není proložen čtením, je třeba použít blokující variantu zápisu. Proměnná `nb` po zavolání obsahuje počet úspěšně odeslaných bajtů dat.

```
// Set for incoming data
FTM_SetParam(device, 0, MD5_CYCLE);

// Send file content to MCU
while (!feof(file))
{
    bytes = fread(buffer, 1, FTM_MAX_SEND_SIZE, file);
    FTM_Write(device, buffer, bytes, &nb);
}
```

Figure 9.5: Část odeslání dat demonstrační PC aplikace

Závěrečná posloupnost volání je ilustrována obrázkem 9.5, kde je provedeno ukončení výpočtu nastavením akce `MD5_RESULT` a jejího potvrzení vyvoláním callbacku. V rámci této akce mikrokontrolér odeslal do PC výsledný MD5 součet o velikosti 16 B. Jeho přečtení je provedeno funkcí `FTM_Read()` a spojení může být ukončeno pomocí volání `FTM_Close()`.

```
// Read checksum
FTM_SetParam(device, 0, MD5_RESULT);
FTM_TriggerCallback(device);
FTM_Read(device, buffer, 16, &nb);

FTM_Close(device);
```

Figure 9.6: Část přijetí výsledku demonstrační PC aplikace

9.3 Akcelerace algoritmu DES

Pro demonstraci tvorby vlastní HW komponenty, např. pro účely akcelerace, byla vytvořena komponenta implementující šifrovací algoritmus DES. Ta vznikla převážně jako další příklad tvorby komponent na straně FPGA, které komunikují s MCU. Následující popis je cílen hlavně na způsob propojení DES jednotky s rozhraním FlexBus a kompletní implementaci je možné nelézt v přiložených zdrojových kódech.

	Výstup	**Vstup**	**Klíč**	**Režim**
1–16b	0x0200	0x0210	0x0220	0x0230
17–32b	0x0202	0x0212	0x0222	
33–48b	0x0204	0x0214	0x0224	
49–64b	0x0206	0x0216	0x0226	

Table 9.1: Popis rozhraní DES komponenty

Algoritmus DES pracuje s bloky dat o velikost 64 bitů [3]. Jelikož FlexBus kanál mezi mikrokontrolérem a FPGA má šířku 16 bitů, bylo třeba vytvořit rozhraní DES algoritmu tak, aby bylo možné provádět konfiguraci jeho parametrů prostřednictvím 16bitového přístupu. Vytvořené rozhraní se skládá z 13 16bitových registrů, jejichž výčet je uveden v tabulce 9.1. Všechny parametry algoritmu byly rozděleny na 16bitové úseky, jež jsou adresované v rámci DES komponenty. Konfigurační registr na adrese `0x0230` je použit pro nastavení režimu šifrování/dešifrování. Výpočet je spuštěn v okamžiku zápisu posledních 16 bitů vstupních dat na adresu `0x0216` a trvá celkem 16 taktů, během kterých je tato zápisová transakce blokována.

Při provedení experimentálního srovnání se ukázalo HW řešení DES algoritmu desetkrát rychlejší než použitá SW implementace v mikrokontroléru. Zde je však nutné podotknout, že komponenta DES algoritmu byla taktována hodinovým signálem rozhraní FlexBus (48 MHz), aby byla zachována synchronnost systému. Pro zvýšení frekvence by bylo nutné doplnit rozhraní DES komponenty o další logiku, která by řešila synchronizaci dvou různých hodinových domén.

Chapter 10

Programování FPGA

Pro programování FPGA byl původně určen obvod VNC2, nicméně přenosová rychlost tohoto kanálu se ukázala jako nedostačující a externí programovací adaptér byl tak jedinou možností, jak provést rekonfiguraci FPGA v rozumném čase. Toto omezení vedlo k požadavku na vytvoření efektivnějších alternativ, které jsou popsány v této kapitole. První možností jak nakonfigurovat FPGA, je využít externí flash paměť. Tato metoda je však vhodnější spíše pro nevolatilní zápis již odladěné konfigurace, která bude použita vždy po zapnutí kitu. Pro vývoj by bylo vhodnější nevolatilní reprogramování FPGA. V rámci DP byly analyzovány možnosti a navržen a implementován způsob, který dovoluje pracovat s FPGA pomocí standardních nástrojů vývojového prostředí Xilinx ISE.

10.1 Programování externí flash paměti

Jedním ze způsobů jak měnit konfiguraci FPGA obvodu, je změna konfiguračního bitstreamu v externí flash paměti. Veškeré prostředky k implementaci aplikace, která umožní programování externí flash, byly vytvořeny v předchozích částech práce. Na straně FPGA se jedná o SPI kontrolér na rozhraní FlexBus, na straně MCU a PC systém pro výměnu dat po USB. Použitím těchto částí byla vytvořena aplikace pro programování flash paměti FPGA obvodu, která se skládá z PC aplikace a aplikace pro MCU. Obě části byly vytvořeny stejným způsobem jako demonstrační aplikace MD5 součtu v kapitole 9.

Aplikace mikrokontroléru přijímá data na logickém kanále `DEST_FPGA_FLASH`. Jeho callback je volán v rámci přerušení a zpracování příchozích dat tedy není závislé na běhu hlavní aplikace. Data jsou přenášena po blocích o velikosti 256 B, což je zároveň velikost jedné stránky flash paměti. Pro kontrolu správného zápisu je použit prostý součet všech zapsaných dat, kdy na konci zápisu celého bitstreamu jsou všechny stránky na zapisovaných pozicích přečteny a porovnány se součtem vygenerovaným ze zdrojových dat. Jelikož za účelem tohoto ověření jsou data přenášena pouze mezi pamětí a MCU, jedná se o rychlý proces, který trvá zhruba 500 ms při maximální velikosti bitsreamu (∼330 kB).

PC aplikace pak implementuje jednoduché konzolové rozhraní, které umožní provést zápis bitstreamu do paměti, mazání a čtení obsahu paměti, verifikaci správného zápisu a případný restart FPGA.

Na PC sestavě kde probíhal vývoj, dosahovala rychlost zápisu průměrně 800 kbps, což při plné velikosti bitstreamu vyžaduje 3 sekundy pro zápis. Při použití komprimovaného bitstreamu lze dobu zápisu ještě výrazně zkrátit. To je oproti nástroji pracující přes VNC2, který byl původně pro tuto funkci určen, až 16násobné zrychlení.

10.2 Programování FPGA pomocí JTAG rozhraní

JTAG rozhraní FPGA obvodu je možné použít k propojení s PC za účelem programování či ladění. Firma Xilinx nabízí JTAG adaptér *Xilinx Platform Cable USB*, který implementuje potřebný překlad signálů mezi rozhraními USB a JTAG v rychlostní kategorii USB High-Speed, kde je možné využít JTAG rozhraní na frekvenci až 24 MHz. Jedná se však o cenově poměrně náročné řešení, kdy cena v době psaní této práce činní 225 USD. Jako alternativní řešení byla zvolena softwarová implementace vlastního JTAG adaptéru, kdy ve spojení s prostředky vývojového prostředí Xilinx ISE a vytvořeného datového kanálu mezi PC a MCU lze dosáhnout stejné funkcionality jako při použití originálního Xilinx JTAG adaptéru.

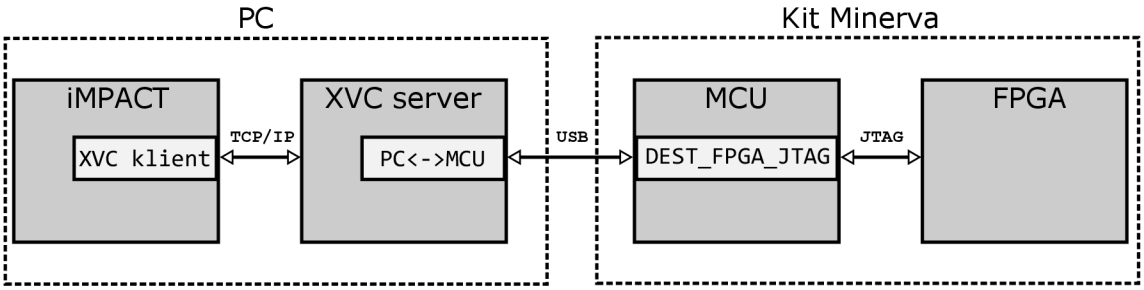


Figure 10.1: Ilustrace vztahu komponent při použití XVC protokolu

Součástí vývojové prostředí Xilinx ISE je aplikace iMPACT, která je mimo jiné určená pro programování FPGA obvodů od tohoto výrobce. Tato aplikace implicitně předpokládá připojení FPGA obvodu prostřednictvím originálního USB kabelu, lze však dodatečným nastavením využít i jiných rozhraní. Jedna z těchto možností je i volba protokolu XVC (Xilinx Virtual Cable), který je postaven na síťovém TCP/IP protokolu. Jedná se o jednoduchý protokol, který kóduje data zasílaná přes JTAG v binární podobě a tvoří tak most mezi programovací aplikací a libovolným JTAG adaptérem, který tyto signály interpretuje.

Navrhl jsem aplikaci, jejíž struktura je znázorněná obrázkem 10.1. Mnou vytvořené části jsou aplikace XVC serveru a softwarové řešení JTAG adaptéru v mikrokontroléru. Aplikace iMPACT je v roli klienta připojeného k serverové aplikaci, která provádí překlad mezi protokolem XVC a protokolem použitého JTAG adaptéru. Ten je implementován jako aplikace v mikrokontroléru, která má v režii interpretaci JTAG signálů na pinech MCU. Tyto piny jsou již přímo připojeny k pinům JTAG rozhraní FPGA obvodu. Aplikace iMPACT s připojeným FPGA přes implementované rozhraní je znázorněna na obrázku 10.2.

Vytvořené řešení umožňuje programování FPGA obvodu stejným způsobem jako při použití originálního JTAG adaptéru od firmy Xilinx a to za nulové náklady. Podstatným rozdílem je však rychlost programování, kdy maximální velikost bitstreamu (∼330 kB) zabere kolem sedmi vteřin, zatímco Xilinx adaptér stejný bitstream dokáže přenést do jedné vteřiny (pouze hrubé orientační srovnání). Efektivnějším řešením by bylo využít ethernetové rozhraní vývojového kitu, kdy by aplikace XVC serveru běžela přímo v mikrokontroléru. Zprovoznění síťové vrstvy vývojového kitu však překračuje jak cíle tak možnosti této práce.

Dodejme, že při používání se vzácně vyskytly situace, kdy komunikace zamrzla na straně aplikace iMPACT. Zde se však zřejmě jedná o nedostatek v implementaci XVC rozšíření a jelikož jde o uzavřený software, nebylo možné tuto příčinu odhalit. Dále bylo vytvořené řešení úspěšně testováno i s novější verzí vývojového prostředí Xilinx Vivado, které by

mohlo do budoucna nahradit aktuálně používané prostředí Xilinx ISE.

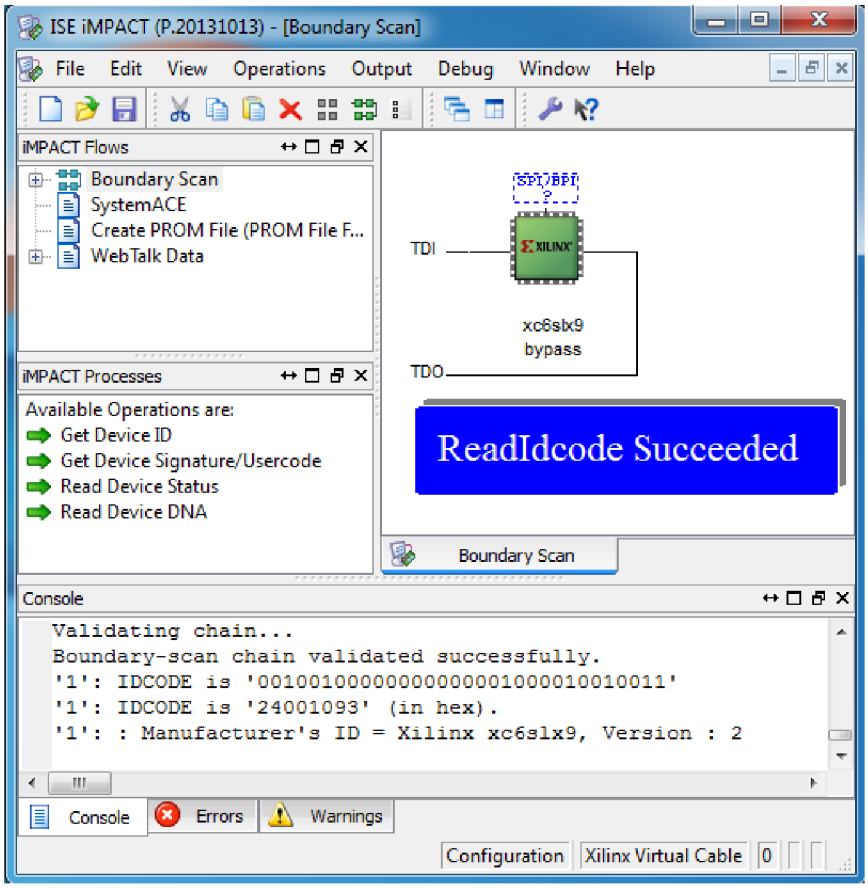


Figure 10.2: Aplikace Xilinx iMPACT s připojeným FPGA kitu přes XVC rozhraní

XVC server

XVC server byl implementován jako konzolová aplikace, která implicitně naslouchá na portu 2542. V aplikaci iMPACT je potřeba provést volbu XVC rozhraní spolu s IP adresou serveru a čísla portu, na kterém server naslouchá. Nastavení se provede v menu *Cable Setup*, kde je potřeba specifikovat položku *Cable Plug-in*. Příklad potřebného nastavení je na obrázku 10.3, kde použití XVC je specifikováno parametry: „`xilinx_xvc host=IP:port disableversioncheck=true`“. Z neznámého důvodu je nutná přítomnost parametru `disableversioncheck` (doporučeno oficiální Xilinx dokumentací [23]), kdy v případě jeho vynechání nedojde k navázání komunikace s XVC serverem.

Po vytvoření spojení spolu obě aplikace komunikují pomocí XVC protokolu, který je složen ze tří příkazů. Jejich výčet je uveden v tabulce 10.1. Příkaz `getinfo` je použit pro získání implementované verze protokolu spolu s maximální velikostí dat, které je server schopen v rámci jednoho příkazu zpracovat. Příkaz `setclk` je použit pro nastavení frekvence hodinového signálu JTAG rozhraní, kdy jako odpověď je očekávána skutečně nastavená frekvence. Pro uspokojení tohoto příkazu lze vrátit libovolnou hodnotu, jelikož má pouze informační charakter bez dalšího vlivu. Veškerá užitečná komunikace pro výměnu JTAG signálů je prostřednictvím příkazu `shift`, který kóduje vstupní datové signály TMS a TDI.

Ty jsou kódovány jako bitové vektory o specifikované délce, kde jednotlivé bity představují stav těchto signálů při náběžné hraně hodinového signálu TCK. Jako odpověď na tento příkaz je bitový vektor výstupního TDO signálu, který byl vzorkován v reakci na aplikaci vstupních signálů TMS a TDI. Pro získání výstupního TDO vektoru jsou přijaté vektory předány dále do mikrokontrolérů, který přijatá data interpretuje.

Příkaz	**Odpověď**
`getinfo:`	„xvcServer_v1.0:8192\n“
`settck:[period in ns]`	1000
`shift:[num bits][tms vector][tdi vector]`	`[tdo vector]`

Table 10.1: Příkazy protokolu XVC

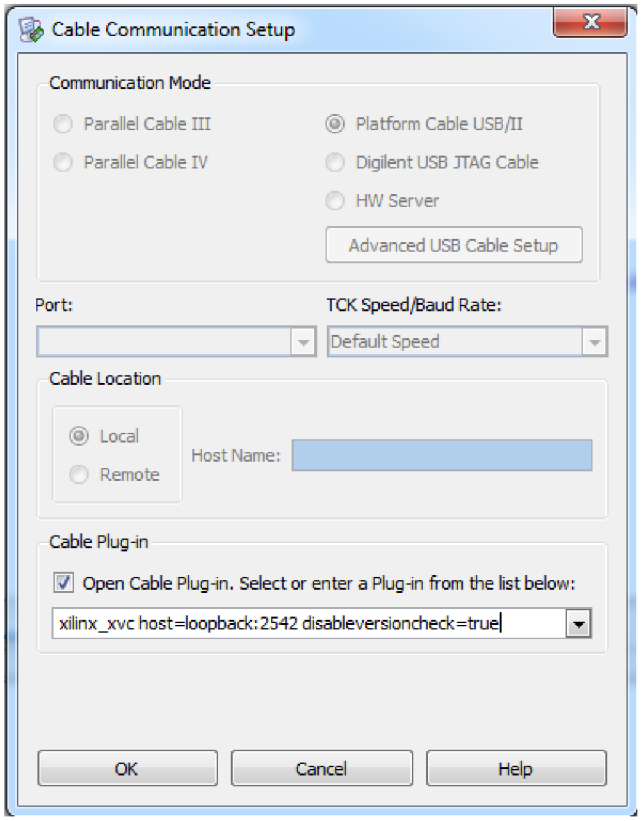


Figure 10.3: Konfigurace Xilinx iMPACT pro použití XVC rozhraní

MCU v roli JTAG adaptéru

Úkolem mikrokontroléru je interpretovat přijaté vektory vstupních JTAG signálů a vzorkovat výstupní TDO signál. Plní tak funkci JTAG adaptéru. Vzhledem k tomu, že MCU není přímo propojeno s JTAG piny FPGA obvodu, je k propojení použít externí GPIO, vyvedený na desce kitu. Způsob propojení je uveden v tabulce 10.2 a ilustrován obrázkem 10.4.

K příjmu dat byl vytvořen nový logický kanál `DEST_FPGA_JTAG`. Interpretace signálů

probíhá tak, že nejprve jsou na příslušné piny vystaveny logické hodnoty TMS a TDI, které jsou po splnění minimální doby vystavení následovány náběžnou hranou generovaného hodinového signálu TCK. Aktivní úroveň TCK je ponechána zhruba 5 us a následuje sestupná hrana, se kterou přichází změna výstupního signálu TDO. Jakmile jsou všechny přijaté bity vektorů takto interpretovány, je navzorkovaný vektor průběhu signálu TDO odeslán zpět do aplikace XVC serveru, který jej použije jako odpověď na aktuální příkaz `shift`.

FPGA	MCU	Lišta P1
TCK	PTA7	27
TMS	PTA9	28
TDI	PTA6	25
TDO	PTA11	26

Table 10.2: Způsob propojení pinů MCU s JTAG rozhraním FPGA

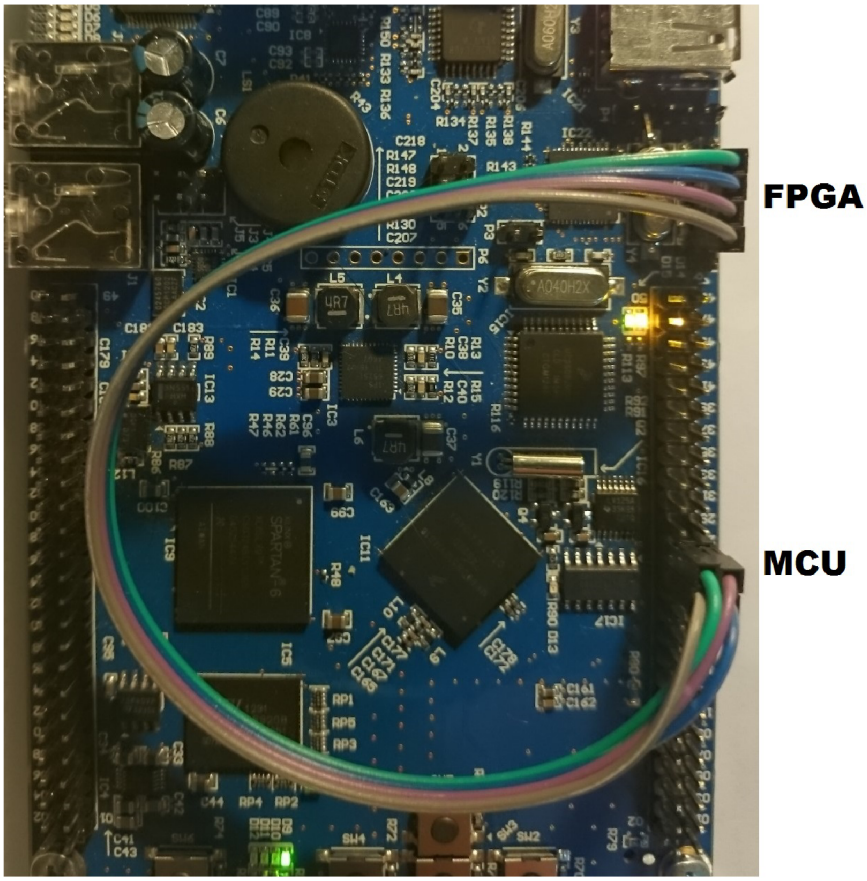


Figure 10.4: Ilustrace propojení MCU s FPGA přes vytvořené JTAG rozhraní

Chapter 11

Závěr

Tato práce si kladla za cíl návrh a implementaci základních prostředků pro vývoj aplikací na studijní vývojové platformě FITkit Minerva. Šlo zejména o vytvoření sady knihovních funkcí zajišťující komunikaci mezi PC, MCU a FPGA, jejichž použitím byly následně vytvořeny prostředky pro programování FPGA obvodu. Pro komunikaci kitu s PC bylo zvoleno USB rozhraní, na kterém bylo vytvořeno zařízení, jež se v OS hlásí jako běžný převodník firmy FTDI. Díky této volbě byla zajištěna přenositelnost mezi nejrozšířenějšími OS, neboť nebylo nutné vytvářet ovladač zařízení, který navíc v prostředí MS Windows musí být certifikovaný, aby jej bylo možné pokládat za důvěryhodný.

Na straně mikrokontroléru byl vytvořen systém pro zjednodušení vývoje aplikací, které potřebují komunikovat s PC. Ten se osvědčil při tvorbě aplikací, které jsou součástí této práce, jako příkazové rozhraní přes sériový terminál nebo prostředky pro programování FPGA. Veškeré aplikační rozhraní je tvořeno pouze několika funkcemi a jejich použití je velice přímočaré. Způsob komunikace s FPGA byl dán již návrhem kitu, kde bylo zvoleno paralelní rozhraní FlexBus. V rámci této práce byl potom vytvořen FlexBus kontrolér pro FPGA, pomocí kterého je možné vytvářet dodatečné komponenty komunikující s MCU po této sběrnici. Jednou z nich je SPI kontrolér, pomocí kterého je možné provádět přístup do externí flash paměti FPGA obvodu a zaručit tak nevolatilní konfiguraci. Další aplikací vytvořenou na základech této práce je virtuální JTAG kanál, s jehož pomocí je možné provádět programování či ladění FPGA přímo z vývojového prostředí Xilinx ISE. Veškeré vytvořené aplikace na straně PC byly vytvořeny tak, aby je bylo možné použít na operačních systémech MS Windows a Linux.

V rámci budoucího zlepšování by si jistě největší pozornost zasloužilo vytvoření síťové vrstvy pomocí ethernetového rozhraní, které je součástí kitu. Pomocí něj by došlo k několikanásobnému zvýšení přenosové rychlosti, což by znamenalo prostor pro celou řadu nových aplikací a zvýšení rychlosti programování FPGA. USB rozhraní však v této práci dostalo přednost, jelikož ladící rozhraní mikrokontroléru již bylo postaveno na USB rozhraní a tedy šlo o logickou volbu.

Přínos této práce by mohl být krátce charakterizován srovnáním situace před zahájením této práce a po dokončení. Na začátku vývoje bylo nejjednodušším řešením pro přenos dat mezi MCU a PC použití externího UART–USB převodníku. Stejně tak pro programování FPGA bylo třeba použít externí JTAG adaptér. Po dokončení této práce již tyto komponenty nejsou potřeba a stačí pouze samotná deska kitu. Navíc veškeré vytvořené prostředky jsou k dispozici pro další vývoj aplikací, které budou dále rozšiřovat edukační hodnotu této platformy.

Bibliography

[1] Atmel Corporation: *Migrating from RS-232 to USB Bridge Specification*, 2004.

[2] Axelson, J.: *USB Complete Fourth Edition : The Developer's Guide*. Lakeview Research, 2009, iSBN 19-3144-808-6.

[3] Chu, P. P.: *FPGA Prototyping Using VHDL Examples*. Wiley-Blackwell, 2008, iSBN 978-0470185315.

[4] Free Software Foundation, Inc.: *GCC online documentation*, [Online], [cit. 2015-04-07].
URL https://gcc.gnu.org/onlinedocs/

[5] Freescale: *Freescale USB Device Stack*, 2012.

[6] Freescale: *K60 Sub-Family Reference Manual*, 2012.

[7] Freescale: *Kinetis Peripheral Module Quick Reference*, 2012.

[8] Freescale: *Kinetis Design Studio v2.0.0 User Guide*, 2014.

[9] Freescale: *USB Stack Device Reference Manual*, 2014.

[10] Future Technology Devices International Ltd.: *Drivers*, [Online], [cit. 2014-12-29].
URL http://www.ftdichip.com/FTDrivers.htm

[11] Future Technology Devices International Ltd.: *AN232B-04 Data Throughput, Latency and Handshaking*, 2006.

[12] Future Technology Devices International Ltd.: *D2XX Programmer's Guide Version 1.3*, 2011.

[13] Future Technology Devices International Ltd.: *FTDI Drivers Installation Guide for Linux*, 2012.

[14] Šimek, V.: *Schéma obvodového zapojení výukového kitu Minerva*. Fakulta informačních technologií VUT Brno, 2013.

[15] Intra2net: *libFTDI - FTDI USB driver with bitbang mode*, [Online], [cit. 2014-12-29].
URL http://www.intra2net.com/en/developer/libftdi/

[16] Micron: *M25P40 Serial Flash Embedded Memory*, 2013.

[17] Microsoft: *Microsoft Developer Network*, [Online], [cit. 2015-03-25].
URL https://msdn.microsoft.com/

[18] NXP Semiconductors: *AN10986 USB In-System Programming with the LPC1300*, 2010.

[19] P&E Microcomputer Systems: *OSBDM User's Design and Troubleshooting Guide.*

[20] USB Implementers Forum, Inc.: *Universal Serial Bus Specification, Revision 2.0*, 2000.

[21] USB Implementers Forum, Inc.: *Universal Serial Bus Device Class Specification for Device Firmware Upgrade*, 2004.

[22] Xilinx: *Spartan-6 Family Overview*, 2011.

[23] Xilinx: *Xilinx Virtual Cable Overview*, 2011.

[24] Xilinx: *Spartan-6 FPGA Configuration User Guide*, 2014.

[25] Yiu, J.: *The Definitive Guide to the ARM Cortex-M3, Second Edition.* Newnes, 2009, iSBN 18-5617-963-X.

Appendix A

Praktické poznámky

A.1 Instalace FTDI D2XX ovladače

Ovladač pro MS Windows i Linux lze volně stáhnout z webových stránek společnosti FTDI. V případě MS Windows je ovladač nainstalován prostřednictvím instalátoru, kde důležitým výstupem jsou soubory `ftd2xx.dll`, `ftd2xx.h`, `ftd2xx.lib`. Pro kompilaci aplikací, které používají tuto DLL, je třeba znát umístění hlavičkového souboru `ftd2xx.h` a linkovacího rozhraní `ftd2xx.lib`. V případě MinGW kompilátoru, který byl použit pro vývoj Minerva frameworku v prostředí Windows, je nejjednodušší tyto soubory překopírovat do adresářů ,,...`MinGW\include`‘‘ resp. ,,...`MinGW\lib`‘‘.

V případě systému Linux jde o manuální instalaci, která je popsána dokumentem [13], nebo v sobuboru `ReadMe.txt`, který je součástí archivu ovladač. Linuxová verze D2XX ovladače bohužel neumožňuje automatické přepnutí z VCP ovladače na D2XX a je třeba tuto akci provést manuálně v situacích, kdy je zaveden VCP ovladač a chceme spustit aplikace založenou na D2XX. Všechny podstatné kroky jsou popsány ve zmíněných dokumentech. Součástí archivu je i spousta příkladů, na kterých lze rychle ověřit úspěšnou instalaci.

A.2 Použití vývojového prostředí Kinetis Design Studio

Kinetis Design Studio (KDS) [8] je volně dostupné vývojové prostředí firmy Freescale určené pro vývoj na mikrokontrolérech Kinetis. Jedná se o upravené vývojové prostředí Eclipse spolu se sadou nástrojů GNU GCC pro ARM a ovladačem ladícího rozhraní OSBDM.

KDS není omezeno pouze na vývoj SW pro mikrokontroléry Kinetis a lze jej použít i pro vývoj ostatních aplikací, jelikož umožňuje zvolit toolchain (kompilátor, linker, debugger...) pro konkrétní projekt. Díky tomu byly všechny aplikace, které jsou součástí Minerva frameworku, vytvořeny v KDS a s jejich zdrojovými kódy lze přehledně pracovat v rámci jednoho vývojového prostředí.

Importování zdrojových kódů do KDS

Zdrojové kódy jsou rozděleny do jednotlivých projektů vytvořených v prostředí KDS a v rámci jedné pracovní plochy (workspace) představují kompletní softwarové řešení frameworku. Přehled projektů v prostředí KDS je znázorněn na obrázku A.1. Importování projektu se provede v menu `File`→`Import...` a následně specifikováním importovaného objektu jako `General`→,,`Existing Project Into Workspace`‘‘.

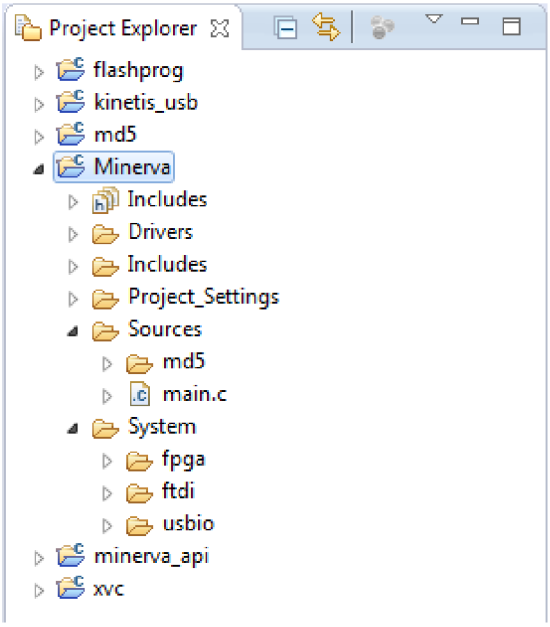


Figure A.1: Jednotlivé projekty v prostředí KDS

Veškeré potřebné závislosti pro provedení kompilace jsou již nastaveny jako součásti projektů. Pro vývoj byl použit kompilátor GNU GCC, resp. MinGW a nastavení projektů by tak mělo fungovat v prostředí Windows i Linux bez dalších modifikací. Veškeré nastavení překladu pro vybraný projekt lze provést v menu `Project`→`Properties` a následně v záložce `C/C++ Build`→`Settings`.

Organizace SW částí frameworku

Význam jednotlivých částí implementace je shrnut v tabulce B.1. Hlavní částí je projekt s názvem **Minerva**, který obsahuje zdrojové kódy aplikace mikrokontroléru. Výstup tohoto projektu (`Minerva.elf`) je pak určen k programování MCU. Složka `Sources` (obrázek A.1) je určena pro zdrojové kódy uživatelské aplikace a již obsahuje demonstrační implementaci MD5 součtu, která byla popsána v kapitole 9. Dalším důležitým adresářem v projektu Minerva je `System`, ve kterém jsou umístěny zdrojové kódy systému pro komunikaci s PC, FPGA a emulaci FTDI obvodu. USB ovladač mikrokontroléru byl umístěn do separátního projektu **kinetis_usb**, jehož výstupem je statická knihovna, která je linkována k hlavnímu projektu Minerva.

Projekt	Strana	Výstup	Popis
`Minerva`	`MCU`	`elf`	Implementace SW MCU
`kinetis_usb`	`MCU`	`lib`	USB ovladač MCU
`minerva_api`	`PC`	`lib`	API k USB kanálu PC↔MCU
`flashprog`	`PC`	`exe`	Aplikace k programování FPGA flash
`xvc`	`PC`	`exe`	XVC server pro iMPACT
`md5`	`PC`	`exe`	PC strana pro MD5 demo

Table A.1: Popis jednotlivých částí SW implementace frameworku

Projekt **minerva_api** obsahuje aplikační rozhraní pro PC aplikace, které pomocí něj komunikují s MCU. Výstupem projektu je statická knihovna, která je linkována k PC aplikacím **flashprog**, **xvc** a **md5**.

Nastavení programování MCU

Nejsnazší cestou k naprogramování mikrokontroléru je přes akci `Debug`, která zároveň spustí ladění kódu. Nastavení programátoru je třeba provést v menu `Run`→`Debug Configurations`, kde se poklepáním na položku ,,`GDB PEMicro Interface Debugging`‘‘ vytvoří nová konfigurace. Pokud byl již projekt **Minerva** úspěšně přeložen, mělo by dojít k automatickému vybrání souboru `Minerva.elf` v `Main`→`C/C++ Application`. V nově vytvořené konfiguraci je třeba provést nastavení podle obrázku A.2. Ostatní parametry nastavení mohou zůstat ve své implicitní hodnotě.

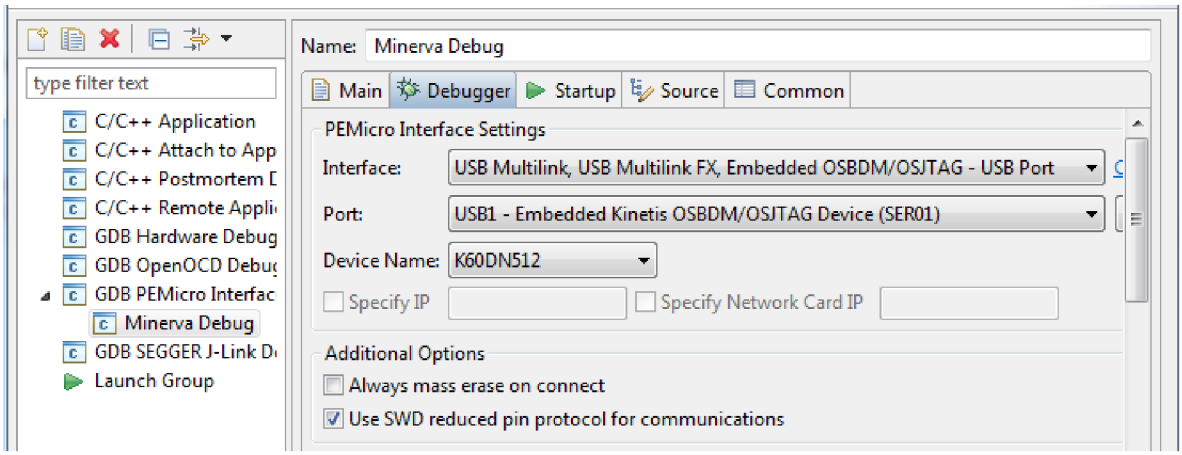


Figure A.2: Konfigurace programátoru pro MCU

Appendix B

Obsah CD

`src_sw`	Složka se zdrojovými kódy SW implementace
`src_hw`	Složka se zdrojovými kódy HW pro FPGA
`utils`	Aplikace pro programování FPGA přeložené pro Win32
`fpga_conf`	Složka s konfiguračním bitstreamem FPGA
`dp_xbucht15.pdf`	Text diplomové práce

Table B.1: Obsah přiloženého CD