Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Technologický experiment na pikosatelitu

Plzeň 2019 Petr Mayr

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 15. května 2019

Petr Mayr

Abstract

This work is a documentation of experiment, that was developed to verify behavior of low-cost processor MSP430 with FRAM memory on Low Earth orbit on board of picosatellite. Designed hardware is well described and each software module including used algorithms is documented in detail.

Abstrakt

Práce popisuje návrh experimentu, který má za cíl ověřit chování nízkonákladového procesoru MSP430 s pamětí typu FRAM v podmínkách na oběžné dráze na palubě pikosatelitu. Obsahem práce je z velké části popis softwarového vybavení pro experiment, dále je zde popsám použitý hardware a schéma zapojení.

Obsah

1 Úvod **7**

2 Procesor MSP430 **8**
2.1 Zdroje hodinového signálu 8
2.2 Adresní prostor 9
2.3 FRAM, Memory Protection Unit 10
2.4 Periferní zařízení 11
2.4.1 Řadič přerušení 11
2.4.2 Časovače 12
2.4.3 Vstupně-výstupní porty 12
2.4.4 Sériová komunikační rozhraní (eUSCI) 12
2.4.5 DMA 13
2.4.6 Watchdog 13
2.5 Bootstrap loader (BSL) 14
2.6 Použité nástroje, překladač 15

3 Širší perspektiva projektu **17**
3.1 Hardware a schéma zapojení 17
3.2 Struktura projektu 19
3.3 Organizace paměťového prostoru 20
3.4 Zavaděč systému 21
3.5 Architektura systému a služby 22
3.5.1 Kontrola integrity paměti 23
3.5.2 Logování a statistiky 24
3.5.3 Přepínání režimu (role) 26
3.5.4 Obnova paměti při výpadku 27
3.5.5 Spouštění testů 29
3.5.6 Vzdálený upgrade firmware 30

4 Softwarové vybavení **32**
4.1 Knihovna ovladačů pro MSP430 32
4.1.1 Konvence a datové struktury 34
4.1.2 WDT a přerušovací systém 37
4.1.3 Zásobník 37
4.1.4 Přerušovací vektory 38
4.1.5 Časovače 40

4.1.6 CRC 42
4.1.7 Vstupně-výstupní porty 43
4.1.8 Komunikační rozhraní 44
4.2 Operační systém 45
4.2.1 Konvence a správa zdrojů 46
4.2.2 Fronty a řazení 47
4.2.3 Akce, fronta akcí 49
4.2.4 Proces 51
4.2.5 Plánovač 52
4.2.6 Signál 55
4.2.7 Událost 56
4.2.8 Časování 57
4.2.9 Spuštění OS 60
4.2.10 Výkonnostní parametry 61
4.3 Komunikační rozhraní 62
4.3.1 Linková vrstva rozhraní UART 63
4.3.2 Transportní vrstva rozhraní UART 65

5 Závěr **68**

Přílohy:

A Schéma zapojení **69**

B Konfigurace portů **70**

Literatura **71**

1 Úvod

Cílem této práce je návrh zařízení s vhodným softwarovým vybavením, které by mělo umožnit vzdálené spouštění testů na procesoru MSP430 na palubě pikosatelitu Pilsen Cube II na oběžné dráze ve vzdálenosti do 2000 km od povrchu Země. Procesor bude vybaven operační pamětí typu FRAM (Ferroelectric Random Access Memory) a použitý software by měl být odolný vůči chybám, které v prostředí se zvýšenou radiací zcela běžně nastávají, jako je například náhodné přepisování obsahu paměťových buněk. Experiment by měl v první řadě ověřit, zda použitý hardware v tomto prostředí vůbec bude schopen pracovat. Pokud ano, tak by mělo být možné procesor vzdáleně programovat a spouštět na něm libovolné testy a informace o průběhu testů z procesoru zpětně vyčíst a ověřit tak chování například některých periferních zařízení v daném prostředí. Dále by v procesoru měly být důkladné statistiky o chybovosti paměti a chybovosti softwarového vybavení a mělo by být možné případné chyby dodatečně opravit.

Experiment by tedy měl být velice odolný vůči chybám a náhodným poruchám a zároveň by měl poskytnout infrastrukturu pro vzdálené testování procesoru MSP430 a jeho periferií na palubě pikosatelitu. Z tohoto důvodu bylo rozhodnuto, že se procesory použijou dva a v jeden moment bude první v pasivím režimu a druhý v aktivním. Na procesoru v aktivním režimu se budou sekvenčně spouštět nahrané testy, které budou moci libovolně zapisovat do logu událostí, a procesor v pasivním režimu bude jen kontrolovat druhý procesor a v případě poruchy se ho pokusí oživit. Z procesoru v pasivním režimu bude možné kdykoliv vyčíst log událostí. Systém by měl být odolný i vůči případným chybám na procesoru v pasivním režimu a proti náhodným výpadkům napájení.

Oba procesory by měly mimo jiné provádět pravidelnou kontrolu integrity vlastní paměti a být do určité míry schopné chyby v paměti opravit. Pokud oprava nebude možná, tak by chybu měl být schopen opravit druhý procesor tím, že procesor s chybami v paměti přeprogramuje vlastním kódem, takže ve výsledku na obou procesorech bude stejný kód jako ve výchozím stavu. Na procesoru v aktivním režimu by tato konrola integrity měla být transparentní a nenarušovat průběh spouštěných testů. V neposlední řadě by softwarové vybavení mělo umožňovat testovat všechny možnosti procesoru MSP430 včetně úsporných režimů a v případě potřeby do libovolného testu zahrnout i druhý procesor.

2 Procesor MSP430

Procesory řady MSP430 od firmy Texas Instruments patří již minimálně deset let mezi nejpoužívanější šesnáctibitové procesory na trhu. Vzdledem ke svým vlastnostem jako je například poměrně vysoký výkon, nízká spotřeba energie a relativně nízká cena, má široké spektrum využití v automatizaci, zabezpečovací technice, senzorických sítích nebo v medicínské informatice. Jednotlivé modelové řady se liší osazenými periferiemi, typem paměti (flash nebo FRAM) a velikostí adresního prostoru. Procesorové jádro MSP430X má dvacetibitové registry a je rozšířeno o instrukce, kterými lze adresovat až 1MB paměti. Procesory řady MSP430Fxx mají interní pamět typu flash, procesory MSP430FRxx mají paměť typu FRAM, obě řady mají 2KB - 8KB volatilní paměti RAM, jejíž obsah se po vypnutí napájení ztrácí. Pro tento experiment byly vybrány procesory MSP430FR5994 a obsahem této kapitoly je stručný popis tohoto procesoru, periferií použitých pro experiment, adresního prostoru, možností programování interní paměti a nástrojů, které Texas Instruments poskytují pro vývoj softwarového vybavení. Jsou zde pouze informace nutné k pochopení navazujícího textu, další informace lze najít v datasheetu[1] nebo v uživatelské dokumentaci[2], oba dokumenty jsou volně ke stažení na webu výrobce.

2.1 Zdroje hodinového signálu

Procesor MSP430FR5994 má v základu tři zdroje hodinového signálu: DCO (digitally-controlled oscillator) o nastavitelné frekvenci v rozsahu 1MHz - 24MHz, MODCLK (nízkonapěťový oscilátor) na frekvenci 5MHz a VLO (very-low-power oscillator) na frekvenci zhruba 10kHz. Dále je možné připojit externí vysokofrekvenční krystalový oscilátor (HFXT - high-frequency crystal oscillator) a nízkofrekvenční krystalový oscilátor (LFXT - low-frequency crystal oscillator). DCO poskytuje relativně stabilní a teplotně nezávislý hodinový signál, nicméně má poněkud vyšší spotřebu energie a po zapnutí trvá delší dobu, než se jeho signál stabilizuje, VLO naopak má velice nízkou spotřebu a teplotní závislost typicky 0.2% / °C a MODOSC je kompromis mezi oběma možnostmi. Jako zdroj hodin pro procesorové jádro (MCLK - master clock) může sloužit kterýkoliv z těchto oscilátorů, maximální frekvence procesorového jádra je 24MHz. Rozvod hodinového signálu pro periferie zajišťuje nezávisle na sobě SMCLK (submodule clock) a ACLK (auxiliary clock). Zdrojem hodin pro SMCLK a ACLK může být opět kterýkoliv osci-

látor, oba zdroje jsou na sobě zcela nezávislé a lze jim softwarově nastavit děličku frekvence v rozsahu /1 - /32.

Na paluě pikosatelitu bude zajištěno stabilní napájení a není důvod šetřit energií, jako zdroj hodin pro MCLK byl tedy zvolen DCO. Vzhledem k tomu, že maximální frekvence paměti FRAM je 8MHz, byla také frekvence DCO zvolena 8MHz, aby nebylo nutné při čtení paměti vkládat čekací cykly. Jako zdroj SMCLK byl také zvolen DCO, tato konfigurace je staticky nastavená během startu systému a nepředpokládá se, že by bylo nutné ji v budoucnu měnit. Rozvod ACLK je vyplnutý a je zcela k dispozici pro testy. Externí krystalové oscilátory nejsou připojené.

2.2 Adresní prostor

Adresní prostor procesoru MSP430FR5994 znázorňuje tabulka 2.1. Tabulka přerušovacích vektorů je na rozsahu 0xFFB4 - 0xFFFF a obsahuje celkem 38 šesnáctibitových přerušovacích vektorů, což mimo jiné znamená, že obsluhy přerušení musí ležet v paměti adresovatelné šesnácti bity. V případě potřeby je možné tabulku přemapovat na horní konec paměti RAM (tedy na rozsah 0x3BB4 - 0x3BFF) nastavením bitu SYSRIVECT v registru SYSCTL. Tiny RAM je nevolatilní paměť, která uchovává svůj obsah ve všech úsporných režimech, SFR jsou funkční registry procesoru a periferií, BSL je paměť bootstrap loaderu (viz 2.5) a paměťi info a device descriptor[1] uchovávají informace o procesoru a periferiích. Horní 4KB paměti RAM jsou sdílené s LEA (low-energy accelerator pro zpracování signálů).

paměť	velikost	adresní rozsah
Tiny RAM	22B	0x0A - 0x1F
SFR	4KB	0x0020 - 0x0FFF
BSL memory (ROM)	2KB	0x1000 - 0x17FF
Info memory (FRAM)	512B	0x1800 - 0x19FF
Device descriptor (FRAM)	256B	0x1A00 - 0x1AFF
RAM	8KB	0x1C00 - 0x3BFF
FRAM	256KB	0x4000 - 0x43FFF

Tabulka 2.1: Adresní prostor procesoru MSP430FR5994

[1] obsahem device descriptoru je mimo jiné i 128-bitová inicializační hodnota pro deterministický generátor náhodných čísel, tzv. *randseed*

2.3 FRAM, Memory Protection Unit

Paměť FRAM (Ferroelectric RAM) je konstrukčně podobná paměti DRAM, paměťové buňky jsou založené na feroelektrické vrstvě narozdíl od DRAM, kde se pro uchování dat využívají kondenzátory. Technologie FRAM je na trhu již zhruba od roku 2000 a mezi hlavní výrobce patří Texas Instruments, Cypress Semiconductor a Fujitsu. FRAM je nevolatilní paměť, tedy paměť, která při výpadku napájení neztrácí obsah. Narozdíl od flash pamětí FRAM vydrží 10^{10}[3] - 10^{14}[4] zápisových cyklů, má nižší spotřebu a zápisy do FRAM jsou podstatně rychlejší. V porovnání s DRAM má technologie FRAM nižší hustotu paměťových buněk a tím pádem vyšší cenu, vybavovací doba je také delší a pohybuje se okolo 35 ns[5] narozdíl od 2 ns u DRAM. Čtecí cyklus je destruktivní a do přečtených paměťových buněk musí být původní hodnota opět zapsána.

Procesor MSP430FR5994 má paměť FRAM namapovanou přímo do adresního prostoru a z programátorského hlediska se chová stejně jako RAM. Přístup do paměti je osmibitový nebo šesnáctibitový. Je schopna pracovat až na frekvenci 8 MHz a pokud procesor pracuje na vyšší frekvenci, tak se mezi přístupy do paměti musí vkládat čekací cykly. Paměť má dvě vyrovnávací paměti o velikosti 64 B, bohužel bližší informace k fungování cache se nepodařilo dohledat. V neposlední řadě má paměť zabudovaný modul pro ochranu proti chybám (ECC - error correction code), který je schopen opravit jeden chybný bit a detekovat chybu ve dvou a více bitech. Během přístupu do paměti detekované chyby ve výchozím stavu způsobí reset procesoru, nicméně je možné řadič paměti nastavit tak, že detekované chyby způsobí SYSNMI (nemaskovatelné přerušení), a tyto chyby pak lze počítat a reagovat na ně vlastním způsobem.

Pro ochranu paměti je možné využít MPU (memory protection unit) a rozdělit paměťový prostor až na tři segmenty, kterým lze nastavit práva pro čtení, zápis a vykonávání kódu. MPU opět při nepovoleném přístupu ve výchozím stavu způsobí reset procesoru a na tyto události je opět možné reagovat v rámci SYSNMI. Pokud situace vyžaduje například zápis do segmentu, který má právo pouze pro čtení, pak je MPU modul nutné před touto operací vypnout. Řadič MPU lze nastavit tak, že MPU nebude možné vypnout až do resetu procesoru (bit MPULOCK v registru MPUCTL0). Tabulka přerušovacich vektorů na adresním rozsahu 0xFFB4 - 0xFFFF nelze chránit pomocí MPU a je na ní vždy právo pro čtení i pro zápis. MPU je v tomto projektu nastaveno tak, že chrání paměťové segmenty obsahující kód proti zápisu, nepovolené přístupy se zapisují do logu událostí a způsobují reset procesoru.

2.4 Periferní zařízení

Na čipu MSP430FR5994 je k dispozici řada standartních periferií a zařízení, jako například časovače nebo sériová komunikační rozhraní. Zde je pouze stručný přehled a popis periferií, které využívá softwarové vybavení tohoto projektu. Detailní informace lze nalézt v uživatelské příručce procesoru[2].

2.4.1 Řadič přerušení

Řadič přerušovacího systému obsluhuje celkem 36 maskovatelných přerušení a dvě nemaskovatelná přerušení, které nelze zakázat a jejichž obsluha se spustí i v případě, že přerušovací systém je vypnutý. Přerušení lze jinak globálně povolit nebo zakázat manipulací bitu GIE (global interrupt enable) v regitru SR (status register) procesoru. maskovatelná přerušení je možné individuálně povolit nebo zakázat a každé přerušení má vlastní (neměnnou) prioritu. Obsluha přerušení nejprve vypne přerušovací systém, uloží návratovou adresu na zásobník a obsah příslušného přerušovacího vektoru vloží do registru PC (program counter). Na zásobník se kromě návratové adresy uloží i obsah SR, který se instrukcí RETI obnoví. Přerušovací vektory jsou šesnáctibitové, obsluhy přerušení musí tedy ležet v adresním prostoru adresovatelném šesnáctibitovou adresou. Během jedné obsluhy přerušení není možné spustit jinou obsluhu přerušení, pokud to během první obluhy není explicitně povoleno nastavením GIE bitu.

Většina přerušovacích vektorů je vyhrazena pro více událostí a má vlastní IV registr (interrupt vector generator). Do tohoto registru jsou zavedeny všechny indikátory přerušení (IFG - interrupt flag) z řídících registrů a jeho čtení automaticky nuluje IFG s nejvyšší prioritou. Pro IFG s nejvyšší prioritou se čtením příslušného registru dostane hodnota 0x02 a pro každý následující IFG s nižší prioritou se vždy čte hodnota pro ten s o 1 vyšší pioritou +0x02. Obsah IV registru lze tedy přičíst k registru PC, kde už se může rovnou skočit na obsluhu pro daný IFG. Tímto způsobem je možné se vyhnout použití switche, nicméně tato metoda zneplatňuje instrukční pipeline procesoru a obsluha musí alespoň z části být napsaná v assembleru, což je velice nepraktické. Reset vektor má také vlastní IV registr (SYSRSTIV), jehož obsahem je identifikátor události, která způsobila reset procesoru, například již dříve uvedené chyby čtení FRAM paměti a nepovolené přístupy do paměti, nebo přístup na adresu mimo paměťový prostor, probuzení z úsporného režimu, výpadek napájení (*brownout*) apod.

2.4.2 Časovače

Na čipu MSP430FR5994 jsou dva tříkanálove a tři dvoukanálové časovače (Timer A) a jeden sedmikanálový časovač (Timer B). Všechny časovače mají přidělené dva přerušovací vektory, kde první je vyhrazen pouze pro obsluhu prvního kanálu (CCR0) a druhý je sdílený pro ostatní kanály a obsluhu přetečení. Zdrojem hodinového signálu může být buď SMCLK, ACLK, nebo externí signál a lze nastavit děličku frekvence na /(n * m), kde $n \in \{1...8\}$ a $m \in \{1, 2, 4, 8\}$. Časovače mají šesnáctibitové čítače, délka čítače u časovače B je nastavitelná na 8, 10, 12 nebo 16 bitů. Každý kanál lze nastavit do režimu *compare*, kde se obsluha přerušení spouští, když hodnota vnitřního čítače je rovna obsahu příslušného CCRn (capture/compare register) registru, nebo do režimu *capture*, kdy se aktuální hodnota vnitřního čítače uloží do příslušného CCRn registru na externí událost. Časovače A se liší počtem kanálů a také tím, že TA2 a TA3 mají pouze vnitřní vstupy a výstupy (nejsou připojené na vstupně-výstupní porty).

2.4.3 Vstupně-výstupní porty

Procesor MSP430FR5994 má celkem osm GPIO (general-purpose input / output) portů po osmi pinech a každý port má přidělen jeden přerušovací vektor pro obsluhu přerušení. Registry portů jsou adresovatelné osmibitově nebo šesnáctibitově, čímž lze přistupovat k registrům dvou sousedních portů najednou. Každý pin lze individuálně nastavit do režimu vstup, výstup nebo jako signál připojený na vstup jiného zařízení na čipu, například A / D převodníku, časovače nebo sériového komunikačního rozhraní. Vstupní pin lze nakonfigurovat s pull-up nebo pull-down rezistorem a přerušení lze generovat buď na náběžné nebo na sestupné hraně vstupního signálu. Po resetu procesoru jsou všechny piny ve stavu vysoké impedance a jsou nastavené jako vstupní, je tedy nejprve nutné porty odemknout (bit LOCKLPM5 v registru PM5CTL0). Procesor je možné probudit ze všech úsporných režimů signálem na kterémkoliv vstupním pinu.

2.4.4 Sériová komunikační rozhraní (eUSCI)

eUSCI (enhanced universal serial communication interface) je rozhraní pro komunikaci pomocí UART, SPI nebo I^2C. Na čipu MSP430FR5994 jsou tři rozhraní eUSCI A, které lze použít jako UART nebo SPI a tři rozhraní eUSCI B, které lze použít jako SPI nebo I^2C. Každé rozhraní má přidělen jeden přerušovací vektor. UART má mnoho možností konfigurace, lze použít jako standartní UART, jako dekodér IrDA signálu, je možné použít

automatickou detekci přenosové rychlosti (baud rate) nebo lze nastavit do režimu komunikace s procesory na sdílené lince pro multiprocesorové systémy. UART má duplexní asynchnonní přenos dat, pro přenos lze nastavit paritní bit a jeden nebo dva stop bity. Rozhraní SPI funguje buď v režimu master nebo slave, v režimu slave nezávisí na zdroji hodinového signálu a je schopno pracovat úsporném režimu LPM4. Rozhraní I^2C není v rámci projektu využito.

2.4.5 DMA

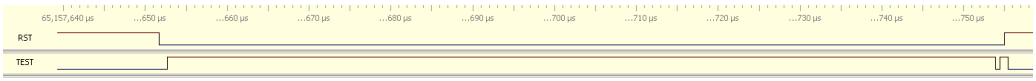
DMA (direct memory access) řadič procesoru MSP430FR5994 má šest nezávislých kanálů s podporou osmibitového nebo šesnáctibitového přenosu. Datové přesuny se dějí bez zásahu procesorového jádra, jeden přesun trvá dva strojové cykly a řadič funguje v úsporných režimech LPM1 - LPM4. Každý kanál má individuálně nastavitelnou zdrojovou a cílovou adresu, zda inkrementovat / dekrementovat adresy po přesunu jednoho byte / wordu a velikost bloku až $2^{16} - 1$ bytů (64KB). Každý kanál je na externí událost schopen přesunout jeden byte / word nebo celý blok paměti. Použití DMA kanálu je jediná cesta pro blokové přenosy dat přes SPI při rychlostech od 1MHz a vyšších, kde přenos jednoho byte trvá 8μs a je tedy nemyslitelné, aby se pro každý přenesený byte volala obsluha přerušení.

2.4.6 Watchdog

Časovač watchdog (WDT - watchdog timer) je standartní periferie, kterou lze nalézt na drtivé většině mikrokontrolérů. WDT slouží k automatickému resetu procesoru v případě chyby v softwaru, například při zacyklení nebo pokud se nějaký kód vykonává déle, než by v nejhorším případě měl. Jako zdroj hodinového signálu pro WDT může sloužit SMCLK nebo ACLK a má osm nastavitelných intervalů do kdy musí být WDT vynulován od 64 cyklů až po dvě miliardy cyklů. WDT je také možné použít jako standartní časovač (nastavením bitu WDTTMSEL v registru WDTCTL), nicméně není možné nijak číst obsah vnitřního čítače. V tomto experimentu má interní WDT individuální nastavení pro každý spuštěný proces a je zodpovědnost daného procesu watchdog nulovat. WDT se automaticky nuluje při každém přepnutí kontextu, kdy se také obnovuje uložené nastavení WDT pro daný proces. Během vytváření nového procesu se do jeho řídící struktury ukládá aktuální obsah kontrolního registru WDT. Idea je taková, že by interní WDT měl vždy běžet a být nastaven na pokud možno co nejnižší hodnotu a mělo by být možné ho vypnout v procesu, v jehož kontextu se spouští testy.

2.5 Bootstrap loader (BSL)

Bootstrap loader na procesorech MSP430 umožňuje číst a zapisovat jejich vnitřní paměť přes sériové rozhraní, v případě MSP430FR5994 je to přes UART. Procesor je nejprve nutné přepnout do BSL režimu resetovací sekvencí, jejíž průběh je zachycen na obr. 2.1. Přepnutí do BSL režimu je možné pouze tehdy, pokud procesor není ovládán pomocí JTAG rozhraní (pokud



Obrázek 2.1: Resetovací sekvence BSL

není např. na debuggeru) a pokud není funkce resetovacího pinu nastavena na NMI (non-maskable interrupt). BSL je také možné vypnout zápisem definované hodnoty do BSL signatury na adrese 0xFF84. Kód BSL je na adresním rozsahu 0x1000 - 0x17FF v nepřepisovatelné paměti. V BSL režimu procesor čeká na požadavek a po jeho přijetí a zpracování odesílá odpověď.

BSL požadavek i odpověď se přenáší formou BSL packetu, který má následující strukturu:

- `záhlaví packetu` 0x80, hodnota kterou musí začínat každý packet
- `délka packetu` (2 byty), délka zahrnuje pouze obsah packetu (přkaz, adresu a data)
- `příkaz` (command, 1 byte) identifikátor BSL příkazu
- `adresa` (3 byty) adresu obsahují jen některé příkazy
- `data` datový buffer packetu, opět ne všechny příkazy ho obsahují
- `CRC otisk` (2 byty) CRC_CCITT otisk obsahu packetu

Délka, adresa a otisk se přenáší jako Little Endian, tedy nejprve nižší byte. Maximální délka obsahu packetu je 260 bytů, tedy 1 byte příkaz, 3 byty adresa a 256 bytů datový buffer. Na chyby přenosu, například neplatné záhlaví packetu, nulová délka nebo neplatné CRC, odpovídá BSL hned, jednoduše pošle jeden byte s chybovým kódem. Pokud je požadavek přijat v pořádku, pak BSL odešle jeden byte ACK (0x00), po čemž se začne příkaz provádět a může následovat odpověď typu message nebo odpověď typu data. Packet odpovědi má stejný formát jako packet požadavku, příkaz je vždy buď 0x2A (data) nebo 0x2B (message). Message packet má vždy jeden byte s chybovým kódem a obsahuje výsledek vyžádané operace, data packet obsahuje datový buffer dlouhý až 256 bytů. BSL tedy na některé příkazy pošle jen ACK byte, na jiné příkazy odpovídá message packetem nebo data packetem a na příkaz Mass Erase (kompletní vymazání paměti) a Load PC (nastavení programového čítače) neodpoví ani ACK bytem. BSL definuje minimální dobu jak

dlouho se nesmí začít vysílat po přijetí znaku na 1,2ms, což odpovídá době, za kterou se přenese jeden znak na baud rate 9600. Přenos dat tedy probíha v poloduplexním[1] režimu a každý odeslaný packet se potvrzuje.

Pamět je chráněna heslem, které odpovídá obsahu přerušovacích vektorů, konkrétně jde o paměťový rozsah 0xFFE0 - 0xFFFF. Po přepnutí procesoru do BSL režimu je tedy nejprve nutné odeslat heslo, aby bylo možné používat příkazy, které zapisují nebo čtou paměť. V případě, že se odešle nesprávné heslo, tak se celá vnitřní paměť maže a je nutné znovu přepnout procesor do BSL režimu a odeslat výchozí heslo (32 x 0xFF). Zápisem definované hodnoty na signaturu BSL lze BSL nastavit tak, že při zadání špatného hesla se paměť nesmaže, ale procesor odpoví message packetem s chybovým kódem *BSL_PASSWORD_ERROR*, nicméně hádání hesla nepřipadá v úvahu, protože na packet obsahující heslo BSL odpovídá se zpožděním minimálně 45ms.

BSL tedy lze velmi dobře použít pro obnovu paměti procesoru, který z nějakého důvodu přestane odpovídat. Vzhledem k tomu, že v projektu jsou dva procesory s identickým kódem, je možné paměť nefunkčního procesoru přepsat pamětí funkčního procesoru, a pokud šlo o vadu paměti, pak by se druhý procesor měl opět probrat. Pro detailní popis procesu obnovy paměti viz 3.5.6. Nicméně BSL je možné využít i v jiných situacich, například lze cílit upgrade softwarového vybavení na konkrétní procesor, viz 3.5.6. BSL protokol mimo jiné poskytuje poměrně robustní řešení pro spolehlivý a potvrzovaný přenos dat mezi procesory, bylo tedy rozhodnuto, že se použije jako protokol pro hlavní komunikační kanál mezi procesory, což by stejně vyžadovalo návrh protokolu s podobnými parametry, a není tedy nutné implementovat protokoly dva. Pro více o komunikačním kanálu mezi procesory viz 4.3. Detailní informace o všech BSL příkazech lze nalézt v uživatelské příručce[6].

2.6 Použité nástroje, překladač

Softwarové vybavení experimentu je napsáno v programovacím jazyce C a některé manipulace se zásobníkem jsou napsány v assembleru. Texas Instruments pro sestavení kódu poskytují dva překladače, a to MSP430-GCC a TI Compiler (TI překladač). Oba překladače se stále vyvíjí, vývoj MSP430-GCC do roku 2016 zajišťoval Red Hat a od té doby ho vyvíjí Somnium[7]. Oba překladače jsou součástí vývojového prostředí Code Composer Studio (CCS) a jsou do jisté míry kompatibilní, například TI překladač

[1] v poloduplexním režimu komunikace probíhá vždy pouze jedním směrem

podporuje velkou část GCC rozšíření[8] a některé mohou být použity jako alternativa k direktivám TI překladače (#pragma), na druhou stranu GCC nepodporuje direktivy TI překladače. Některé části projektu, konkrétně jádro operačního systému (OS), knihovna ovladačů pro MSP430 a projekt s funkčními testy OS, lze sestavit oběma překladači a bylo experimentálně zjištěno, že TI překladač generuje kompaktnější strojový kód a výsledný kód je lépe optimalizovaný. Další nevýhodou MSP430-GCC je to, že ladění programu přeloženého tímto překladačem je velmi problematické, prostředí CCS často neumí namapovat některé globální symboly na konkrétní adresy a do některých modulů například nelze nastavit breakpoint, a to i v případě, že optimalizace je nastavena na *-Og*, tedy optimalizace pro ladící prostředí. Nicméně v tomto případě může jít také o chybu přímo v CCS, jehož použití je samo o sobě velice problematické. Jako překladač byl tedy zvolen TI compiler verze 18.12.1LTS.

Vývoj software probíhal v IDE (integrated development environment) CLion, který poskytuje velice efektivní prostředí pro práci se zdrojovým kódem, a CCS byl použit jen pro nahrávání sestaveného kódu do procesoru a pro účely ladění. Původní myšlenka byla taková, že by se nejprve připravil *toolchain* pro oba překladače pro CMake[1] a vývoj, překlad i ladění by tím pádem mohly probíhat přímo v prostředí CLionu. Pár podobných (dokonce úspěšných) pokusů lze na internetu nalézt[9, 10], bohužel zprovoznit tyto *toolchainy* už nebylo v mých časových možnostech, ale rozhodně by se vyplatilo do toho ještě nějaký čas investovat. Vývoj by pak nezávisel na CCS a bylo by možné jednoduše vygenerovat projekt např. pro Visual Studio nebo pro CodeBlocks. CCS je založený na dnes již zastaralém prostředí Eclipse, kde práce se zdrojovým kódem je krajně neefektivní, a navíc je CCS velice nestabilní. Projekt tedy obsahuje soubory CMakeLists.txt, které jsou zde zatím pouze z toho důvodu, aby šel projekt otevřít v CLionu.

Jako verzovací systém byl zvolen GIT, zdrojové kódy operačního systému a knihovny ovladačů jsou zveřejněny na mém githubu pod BSD-3 licencí, lze je tedy volně distribuovat a použít pro vývoj proprietárních systémů bez nutnosti zveřejnění zdrojových kódů. Softwarové vybavení experimentu využívá tyto dva projekty jako submoduly GITu a celý projekt lze bez problémů otevřít v prostředí CCS. V neposlední řadě je třeba zmínit, že ladění softwarového vybavení probíhalo také za pomoci logického analyzátoru Sigma 2 od české firmy ASIX, a průběhy signálů prezentované v této práci jsou zachycené pomocí tohoto zařízení. Pro simulaci sběrnice na palubě satelitu, na kterou bude experiment připojen, byl použit MSP-exp430f5438[11].

[1] CMake je moderní generátor build systémů, sám o sobě však build systém není.

3 Širší perspektiva projektu

Postup realizace experimentu lze rozdělit do několiká částí, od návrhu hardware a schématu zapojení přes návrh infrastruktury softwarového vybavení až po realizaci chování celého systému. Tato kapitola popisuje všechny tyto části bez přílišného zacházení do detailů, takže by po jejím přečtení měl čtenář získat ucelenou představu o fungování navrženého systému a o architektuře softwarového vybavení. Detailní informace týkající se software, jeho implementace a testování, lze nalézt v navazujících kapitolách.

3.1 Hardware a schéma zapojení

Plošný spoj s oběma procesory bude na palubě satelitu připojen přes čtyřvodičové SPI rozhraní v režimu *slave*. SPI pin *slave transmit enable* (STE) je tedy pouze jeden a z hlediska komunikačního rozhraní satelitu se procesory jeví jako jedno zařízení. Rozhodnutí o tom, který procesor zrovna bude komunikovat s rozhraním satelitu, závisí pouze na tom, jak se procesory mezi sebou domluví. Procesory na desce mají propojené dvě komunikační rozhraní eUSCI A0 a A1, které mohou pracovat pouze jako UART, dále jedno rozhraní eUSCI B1, které může být použito jako SPI nebo I^2C, a ještě čtyři další piny, které mohou být nastaveny pouze do GPIO (general-purpose IO) režimu a jeden z nich lze zavést na vstup časovače B. Rozhraní eUSCI A0 a A1 nemohou být použita v SPI režimu, protože jejich hodinový signál (CLK) není propojen. Možnosti vzájemné komunikace jsou tedy velice široké, nicméně snaha je minimalizovat moduly procesoru, na kterých bude software záviset. Důvody jsou dva, v první řadě je nutné použít jen nezbytně nutný hardware pro snížení pravděpodobnosti selhání systému, například pokud bude software zajišťující základní funkce systému záviset na násobičce, která v daném prostředí přestane pracovat, pak přestane pracovat celý systém. Druhým důvodem je ten, že je nutné uvolnit všechny možné prostředky pro spouštěné testy.

Komunikační rozhraní A0 je zároveň rozhraní, které používá bootstrap loader (BSL), a tím pádem musí být využito pro případné účely obnovy paměti procesoru, který přestal pracovat. Bylo tedy rozhodnuto, že veškerá komunikace mezi procesory nutná pro chod systému bude probíhat po tomto rozhraní. Použití komunikace typu *master-slave*, kterou poskytuje SPI a I^2C, je pro tento účel nevhodná, protože neposkytuje duplexní komunikační ka-

nál, což by celou situaci zbytečně komplikovalo. Přepnutí procesoru do BSL režimu musí předcházet BSL resetovací sekvence, a z toho důvodu musí být piny TEST a RST jednoho procesoru připojené na GPIO piny druhého procesoru a opačně. Jeden procesor tedy může kdykoliv resetovat ten druhý, a aby se zabránilo tomu, že se systém dostane do stavu, kdy jeden procesor pouze resetuje ten druhý a jinak je například zacyklený a nedělá nic, tak byly na desku přidány dva externí WDT (watchdog timer). Tyto WDT tedy zabraňují tomu, aby procesor, který WDT v případě poruchy pravidelně neresetuje, měl přístup k pinům TEST a RST druhého procesoru. WDT také v případě poruchy omezuje komunikaci s rozhraním satelitu tím, že signál STE se nedostane na SPI rozhraní vadného procesoru (je vždy v logické nule). Oba externí WDT také generují signál *ALIVE*, který je připojen na procesor, jehož zodpovědností není ten konkrétní WDT nulovat, takže se sestupnou hranou na signálu *ALIVE* dozví, že došlo k timeoutu daného WDT. Popsané schéma lze najít v příloze A.1.

Vzhledem k tomu, že software zajišťující běh systému je na obou procesorech identický, je nutné, aby bylo možné nějakým způsobem procesory od sebe odlišit. Toho je docíleno tím, že jeden procesor má na pinu *MCU_ID* zavedenou logickou 1 a druhý logickou 0. Tím pádem lze vstup na tomto pinu přečíst a software může rozpoznat, na kterém z procesorů běží. K oběma procesorům je připojená sada rezistorů takovým způsobem, že bude možné testovat chování A/D (analog to digital) převodníků a komparátoru napětí. Procesory mají propojené některé sousední piny pro případné použití v rámci testů. Konfigurace portů a to, který signál je připojen na který pin, lze nalézt v přílohách B.1 a B.2

Pro vývoj softwarového vybavení byly využity dvě desky MSP430FR5994 LaunchPad Development Kit[12], na kterých je sonda umožňující ladění software, jsou zde dvě tlačítka, tlačítko na reset procesoru, dvě LED diody a GPIO piny vyvedené na standartní externí konektory. Dále byl vyroben plošný spoj, který obsahuje veškerou výše popsanou logiku a na který lze tyto desky připojit, takže propojení procesorů odpovídá výše popsanému schématu. Na simulaci SPI rozhraní satelitu je použita deska MSP-exp430f5438[11], kterou lze k testovacímu plošnému spoji opět připojit standartním způsobem. Průběh signálů na jednotlivých pinech lze sledovat logickým analyzátorem. Zapojení na testovací desce je tedy shodné se zapojením na finální desce, která se bude instalovat do satelitu, a zároveň umožňuje vývoj a ladění software. Jediný rozdíl spočívá v tom, že timeout externích WDT na finální desce je nastaven na 10 minut a na testovací desce je to 15 vteřin.

3.2 Struktura projektu

Software projektu má modulární architekturu a je založen na operačním systému (OS) a knihovně ovladačů. Oba tyto moduly jsou samostatné projekty, ovladače nijak nezávisí na OS a OS nezávisí na této konkrétní knihovně ovladačů, a v podstatě nezávisí ani na hardware a použitém překladači. Většina ovladačů by měla fungovat na všech procesorech řady MSP430 a lze je použít zcela samostatně. Tyto dva moduly jsou v projektu použity jako externí knihovny, z hlediska verzovacího systému jde o submoduly. Knihovna ovladačů poskytuje abstraktní vrstvu nad hardware a obsahuje mimo jiné ovladače pro snadnou manipulaci s přerušovacími vektory, dynamické nastavování obsluhy přerušení, ovladače pro práci s komunikačními rozhraními a DMA řadičem, s časovači, IO porty nebo pro manipulace se zásobníkem. Operační systém poskytuje infrastrukturu pro spouštění procesů, časování, asynchronní zpracování událostí, blokující volání a synchronizaci procesů, předávání signálů, správu zdrojů a plánovač reálného času. Pro důkladné odladění veškeré funkcionality těchto dvou knihoven vznikl projekt, který obsahuje sadu systémových testů, které je možné sekvenčně spouštět na procesoru MSP430FR5994. Tento projekt lze najít na githubu[1] a lze ho použít také jako sadu příkladů použití těchto dvou knihoven, nicméně jeho obsah zde dále nebudu rozvádět. Obě knihovny jsou v adresáři *module*, která se nachází v kořenovém adresáři projektu.

Zdrojové kódy software, který zajišťuje základní funkcionalitu, jako například komunikaci s rozhraním satelitu, spouštění testů nebo kontrolu integrity paměti, lze nalézt v adresáři *system* v kořenovém adresáři projektu. Je zde také mimo jiné zavaděč systému, konfigurace portů a kódy komunikačního rozhraní mezi procesory. V adresáři *application* jsou pak všechny testy, které se na daném procesoru v aktivním režimu spouštějí. Veškerý kód, data i výchozí zásobník jsou umístěny do paměti FRAM, paměť RAM zůstává nevyužita. Pro organizaci paměti není využit žádný konkrétní paměťový model, naopak vše je jasně definováno v konfiguračním souboru pro linker (*lnk_msp430fr5994.cmd*). Kód, data i zásobník mohou obecně ležet kdekoliv v paměti, z toho důvodu je při překladu kódu nastaven data a code model *large*, tím pádem všechny pointery na data i na funkce mají délku 32 bitů. Pro start (boot) systému není využita žádná externí knihovna (RTS - runtime support library) a systém má vlastní zavaděč.

[1] https://github.com/mutant-industries/PrimerOS-test-project

3.3 Organizace paměťového prostoru

Vzhledem k tomu, že na palubě satelitu může dojít kdykoliv k výpadku napájení, je nutné, aby byl systém zcela nezávislý na obsahu paměťových segmentů, kde jsou globální a statické proměnné, tedy aby byl z tohoto pohledu systém bezestavový. Zároveň je zde požadavek na to, aby byl systém schopen se probrat z úsporných režimů LPMx.5, kdy se procesor resetuje a opět se provádí kód zavaděče. Z tohoto důvodu byl datový segment rozdělen na tři sekce: *noinit*, *persistent* a *zerofill*. Sekci *noinit* zavaděč nikdy neinicializuje, obsahuje tedy data, která by z nějakého důvodu měla vydržet mezi resety, například aktuální velikost logu událostí. Sekci *zerofill* zavaděč nuluje při každém startu a sekci *persistent* nuluje v případě, pokud reset nezpůsobilo probuzení z úsporného režimu LPMx.5. Sekce persistent by tedy měla obsahovat všechny datové struktury, se kterými pracuje operační systém, aby se mohl obnovit jeho stav před přepnutím do LPMx.5. Datová paměť vyhrazená pro testy je oddělená od datové paměti systému a zavaděč ji nikdy neinicializuje, lze ji tedy chápat jako *noinit*. Dále je zde speciální sekce vyhrazená pro statistiky a log událostí, který si spravuje sám modul, který zajišťuje logování.

Kód systému leží na začátku paměti FRAM a je rozdělen do menších sekcí o velikosti do 3KB. Každá tato sekce je zarovnaná na hranici 64 byte (0x40) a obsahuje 64 bytů rezervy pro případné pozdější úpravy. První a poslední sekce obsahují hlavní a záložní kód zavaděče, druhá sekce obsahuje nutné minimum pro spuštění kontroly integrity paměti a logování a dalších 35 sekcí obsahuje ovladače, operační systém a veškerý kód nutný k běhu systému. Pro každou sekci linker generuje CRC záznam a CRC tabulky jsou umístěny za záložním zavaděčem. Kódová paměť systému včetně rezerv a CRC tabulek má celkem 37KB[1] a chrání ji jednotka MPU proti zápisu. Každá sekce zpravidla obsahuje jeden modul (jeden .obj) a jako první má vždy namapované globální funkce (která je zpravidla jedna), což umožňuje pozdější úpravy v lokálních (*static*) funkcích (případně v celém modulu, pokud je globální funkce jen jedna) bez nutnosti přeprogramovat celý firmware (za předpokladu, že se úpravy ještě vejdou do rezervy vyhrazené pro daný modul). Nicméně kompletní update firmware systém také umožňuje, viz 3.5.6. Celý tento paměťový segment se zrcadlí do záložního paměťového prostoru na konci FRAM, ze kterého ho zle obnovit, více informací v 3.5.1.

Kód testů leží na konci paměti FRAM, kterou opět chrání jednotka MPU proti zápisu. Pro testy je vyhrazeno 32KB paměti a CRC tabulka

[1] 37KB bez optimalizací překladače (-O0), 35KB má finální verze (-O3 a žádný ladící kód)

s šesnácti záznamy (sloty). Kód jednoho testu může ležet kdekoliv v rámci tohoto segmentu a nesmí přesahovat 8KB. Slot obsahuje spustitelný test v případě, že jeho CRC záznam je platný, vstupní bod (entry point) testu je vždy začátek sekce, ve které leží (tedy adresa z jeho CRC záznamu). Kód testů je opět zrcadlen do záložního segmentu, ze kterého lze obnovit. Na samotném konci paměti FRAM je vyhrazeno 64KB paměti pro vzdálený upgrade firmware a nahrávání testů. Zjednodušený přehled organizace paměťového prostoru zachycuje tabulka 3.1, detailní informace lze najít v souboru *XX_diplomka.map*, který generuje linker při sestavení programu.

obsah	velikost	adresní rozsah
kód systému	37KB	0x4000 - 0xD1FF
data systému	11KB	0xd200 - 0xFDFF
výchozí zásobník	380B	0xFE00 - 0xFF7B
log událostí, statistiky	21KB	0x10000 - 0x15553
data testů	21KB	0x15560 - 0x1A9FF
kód testů	32KB	0x1aa00 - 0x229FF
záloha kódu testů	32KB	0x22a00 - 0x2A9FF
záloha kódu systému	37KB	0x2ab00 - 0x33CFF
upgrade buffer	64KB	0x33d00 - 0x43CFF

Tabulka 3.1: Organizace paměťového prostoru, podbarvené segmenty chrání jednotka MPU proti zápisu

3.4 Zavaděč systému

Po resetu procesoru se nastavuje programový čítač (PC, program counter) na adresu nastavenou v tabulce přerušovacích vektorů, konkrétně na *reset vector*, který ve výchozím stavu obsahuje adresu hlavního zavaděče. V tento moment není známá integrita kódu zavaděče, proto jako první instrukce je nastavení *reset vectoru* na adresu záložního zavaděče. Kód zavaděče je jedna monolitická funkce, jejíž součástí je kód pro výpočet CRC jí samotné. Nejprve se tedy provede kontrola integrity kódu zavaděče, která pouze používá registry procesoru a čte paměť, žádné zápisy ani uložení návratových adres na zásobník v tento moment nepřipadají v úvahu. Při neúspěšné kontrole se čeká na reset WDT (watchdog timeru) a po resetu se spouští záložní zavaděč. Když kontrola proběhne úspěšně, tedy pokud vypočtené CRC odpovídá záznamu pro daný kód zavaděče, pak je patrné nejen to, že aktuálně spouš-

těný kód je platný, ale že je k dispozici ověřená funkce pro výpočet CRC. V tento moment se *reset vector* nastaví zpět na adresu aktuálního zavaděče, nastaví se ukazatel na vrchol zásobníku (SP - stack pointer) a nastaví se zdroj hodinového signálu pro MCLK (master clock) a SMCLK (submodule clock) na 8MHz. Dále pomocí ověřeného kódu pro výpočet CRC proběhne kontrola integrity (a případná obnova ze záložního segmentu) sekce, která obsahuje standartní (optimalizovaný) kód pro ověřování integrity paměti. Po neúspěšné kontrole se opět čeká na reset WDT a v tomto případě bude muset kód obnovit druhý procesor pomocí BSL. Dále se nastaví a zapne MPU (memory protection unit) jednotka a provede se inicializace datového segmentu systému. Nakonec se již ověřeným (optimalizovaným) kódem pro kontrolu integrity paměti (a případnou obnovu ze záložního segmentu) provede kompletní kontrola kódu systému. Po neúspěšné kontrole se opět čeká na reset WDT a v tomto případě opět bude muset kód obnovit druhý procesor pomocí BSL. Pokud je kontrola úspěšná, tak se nenávratně skočí na vstupní bod (entry point) systému.

Celý kód zavaděče je složen z několika funkcí deklarovaných jako *inline* a překladač tuto direktivu zcela ignoruje v případě, že jsou vyplé optimalizace kódu (*-Ooff*). Vzhledem k tomu, že se SP nastavuje až potom, co se ověřila integrita kódu zavaděče, je nutné kód překládat alespoň s úrovní optimalizace *-O0*. Pro ladění tohoto kódu není možné využít softwarové breakpointy, protože mění obsah paměti a vypočtené CRC pak nesouhlasí s CRC záznamem, který generuje linker.

3.5 Architektura systému a služby

Systém během startu spouští dva procesy, v jejichž kontextu probíhá asynchronní obsluha událostí, jako je například příjem packetu od druhého procesoru, příjem požadavku ze sběrnice satelitu, skončení procesu, v jehož kontextu probíhal test, apod. Skutečnost, že nějaká událost do dané doby (timeoutu) měla nastat a nenastala, se zde označuje také jako událost. Vyjma spouštěných testů je tedy celý systém zcela asynchronní, každá událost má svojí prioritu, kterou dědí proces, v jehož kontextu se obsluha této události provádí. V moment nečinnosti běží proces (idle), který dokola provádí kontrolu integrity paměti, případně je schopen systém přepnout do úsporného režimu. Jeden ze spuštěných procesů je vyhrazen pro události s velmi rychlou obsluhou (řádově 100μs) a druhý pro události s nižší prioritou, během kterých je nutné například provést výpočet CRC pro paměťový segment před spuštěním testu. V kontextu jednoho procesu se obsluha události vždy do-

končí před spuštěním obsluhy další události (aktuálně s nejvyšší prioritou), případně se proces přestane plánovat, pokud nenastaly žádné další události, jejichž obsluha probíhá v jeho kontextu. Pomineme-li fakt, že proces dědí prioritu událostí, pak tento přístup odpovídá návrhovému vzoru *Active Object pattern*. Během zpracování obsluhy události je možné, že se přepne kontext z jednoho procesu na druhý, pokud se v jeho kontextu má obsloužit událost s vyšší prioritou, než je nejvyšší priorita ve frontě událostí prvního procesu. Proces, v jehož kontextu se spouští test, má nižší prioritu, než systémové události, nečinný proces (idle) má prioritu 0. Přehled priorit všech událostí a procesů lze nalézt v souboru *priority.h*.

Systém po startu tedy nastavuje obsluhy (handlery) na všechny očekávané události. Skupina handlerů, které dohromady tvoří nějaký logický celek, se v tomto textu označuje jako *služba*. Zdrojový kód všech systémových služeb je v adresáři *system/service* a následuje jejich stručný popis.

3.5.1 Kontrola integrity paměti

Integrita a případná oprava paměti kódu systému a jeho zálohy probíhá v procesu s prioritou 0, který se tedy plánuje pouze v případě, že zrovna není nic jiného na práci. V paměti je každá kódová sekce a její CRC záznam dvakrát a kontrola probíhá tak, že se najde alespoň jedna platná kombinace paměťové sekce a CRC záznamu a druhá (hlavní nebo záložní) sekce a CRC záznam se následně porovnají s touto platnou dvojicí a opraví se případné rozdíly. Ve výsledku oba CRC záznamy a obě paměťové sekce obsahují identická data, takže každá kombinace CRC záznamu a paměťové sekce je platná. Po vymazání obsahu paměti a nahrání kódu je segment se zálohou paměti vyplněn hodnotou 0xFF, takže při prvním průběhu kontroly integrity se celý kód a jeho CRC záznamy zrcadlí do záložního segmentu. Kompletní kontrola obsahu systémové paměti a její zálohy trvá 500ms, kontrola včetně vytvoření zálohy trvá 800ms.

Systémový kód je rozdělen celkem na 38 sekcí do velikosti 3KB, a to hned ze dvou důvodů. Prvním je ten, že při vysokém výskytu chyb bude efektivnější opravovat paměť po malých částech a sníží se pravděpodobnost, že se objeví chyba zároveň v hlavním i záložním segmentu, čímž se paměť stává neopravitelnou. Druhým důvodem je ten, že CRC záznam při náhodném přepsání obsahu může obsahovat náhodná data, takže i délka segmentu může být až 64KB. Omezíme-li maximální délku na nějakou mezní hodnotu, pak se všechny CRC záznamy s délkou segmentu větší, než je tato hodnota, automaticky označí za neplatné, a není nutné provádět výpočet CRC, což by bylo zbytečné plýtvání procesorovým časem. Mimo to je pak možné určit

maximální počet strojových cyklů nutných pro kontrolu sekce omezenou na určitou maximální velikost.

Případ s nejdelší výpočetní dobou je ten, když CRC záznam a paměťová sekce jsou platné pouze v záložním segmentu a když CRC záznam v kódovém segmentu má nastavenou délku sekce na maximální povolenou velikost. V tomto případě je platná pouze kombinace záložní CRC záznam - záložní sekce a je nutné provést výpočet CRC pro zbylé tři kombinace a nakonec ještě obnovit původní sekci a CRC záznam. Bylo experimentálně zjištěno, že pro velikost 8KB tento nejhorší případ trvá přibližně 6400K strojových cyklů. Vzhledem k tomu, že intervaly, na které se dá nastavit interní WDT, jsou 512K cyklů a další je až 8192K cyklů, bylo rozhodnuto, že se maximální velikost sekce omezí na 8KB. Proces tedy může nastavit WDT na tento počet cyklů před kontrolou integrity každé sekce. Z toho důvodu také kód jednoho testu nemůže přesahovat limit 8KB.

V případě, že chyby v paměti mají takový rozsah, že nelze obnovit její integritu, tak se procesor resetuje a čeká na to, až jeho paměť přepíše služba obnovy paměti druhého procesoru (viz 3.5.4). Situace, kdy kód pouze jednoho zavaděče je platný, se nepovažuje za důvod k resetu. Pokud paměť na obou procesorech nelze obnovit, pak systém přestane reagovat a nebude schopen se z tohoto stavu zotavit. Nicméně vzdledem s tomu, že na procesoru v pasivním režimu se kromě kontroly a oprav chyb v paměti neděje nic, je tato situace velmi nepravděpodobná. Vedlejším efektem kontroly integrity paměti je to, že během vývoje nelze používat softwarové breakpointy, které instrukci, na kterou má být breakpoint přidán, přepisují instrukcí skoku. Na MSP430FR5994 jsou k dispozici pouze tři hardwarové breakpointy, což proces ladění softwarového vybavení poměrně dost omezuje, proto je často vhodné kontrolu integrity na ladícím prostředí vypnout.

3.5.2 Logování a statistiky

Logování, neboli zápisy do logu událostí, a čítače událostí ověřené CRC záznamem a vyčítání těchto informací, umožňují získání přehledu o tom, co se na daném procesoru ve skutečnosti děje. V systému je celkem 14 čítačů chybových událostí, jako například již zmíněné opravitelné a neopravitelné chyby při čtení FRAM paměti, pokusy o zápis do paměťového segmentu chráněného proti zápisu jednotkou MPU, chyby na komunikačním rozhraní mezi procesory nebo přístupy na adresu mimo adresní prostor. Dále jsou zde čítače spuštění jednotlivých testů a jejich ukončení s nenulovou návratovou hodnotou. Všechny čítače mají délku 16 bitů a svůj vlastní CRC otisk. Otisk se kontroluje před inkrementací a přepočítává se po inkrementaci, případně

po vynulování. Pokud CRC otisk nesouhlasí s hodnotou čítače, tak se to zapíše do logu událostí a čítač se nuluje. Každé přetečení čítače se zapisuje do logu událostí. Pro tyto čítače je vyhrazená paměť o velikosti 256 bytů, na kterou navazuje paměť vyhrazená pro log událostí.

Log událostí má celkovou velikost 21KB a záznam v logu má následující strukturu:

- `záhlaví` 0xAA, hodnota kterou musí začínat každý záznam
- `délka záznamu` (2 byty), délka zahrnuje pouze obsah záznamu
- `identifikátor procesoru` (1 byte) podle pinu MCU_ID
- `aktuální čas v` μ`s` (4 byty) pokud je časování během vytváření záznamu vypnuté, tak je hodnota času 0
- `typ záznamu` (1 byte) info, error, warning apod.
- `identifikátor modulu, který loguje` (1 byte)
- `typ logované události` (2 byte)
- `buffer` (délka až 21KB), nepovinný
- `CRC otisk záznamu` (2 byte)

CRC otisk se počítá pro každý záznam zvlášť, protože je výhodnější mít pár neplatných záznamů než celý log událostí. Zápis do logu je atomická operace, probíhá s vypnutým přerušovacím systémem a až po tom, co je záznam zapsán do logu, se upravuje délka logu událostí. Je možné nastavit globální proměnnou *log_level* a filtrovat tak všechny záznamy s důležitostí nižší, než je tato nastavená úroveň. Zápis záznamu typu debug nebo trace na finální (release) verzi programu nemá žádný efekt (makra, která jinak volají logovací funkci, jsou prázdná).

Požadavek na vyčtení logů uzamyká log událostí a nastavuje adresu a délku bufferu, který se má odeslat po komunikační sběrnici satelitu. Adresa a délka se nastavuje v závislosti na tom, zda od předchozího vyčtení logů došlo ke změně v čítačích nebo v logu událostí. Paměťové segmenty čítačů a logů na sebe navazují, takže je možné je oba odeslat jako jeden souvislý buffer, případně odeslat jen jeden z nich. Tuto adresu a délku pak obsluha komunikačního rozhraní se satelitem použije k jednorázovému nastaveni DMA (direct menory access) řadiče a v ten moment je obsah logů připraven k odeslání. I během odesílání, když je log událostí uzamčený, je možné logovat, všechny záznamy zapsané za tuto dobu se po odeslání přesunou na začátek logu událostí a je možné je vyčíst při následujícím čtení. Pokud je log událostí plný, tak do něj nelze zapisovat a každý pokus o takový zápis inkrementuje chybový čítač přetečení logu.

3.5.3 Přepínání režimu (role)

Procesor je vždy buď v aktivním režimu a spouštějí se na něm testy, nebo je v pasivním režimu a funguje jako podpora pro druhý procesor a komunikuje s rozhraním satelitu. Procesory nikdy nejsou oba v aktivním režimu a zpoždění mezi tím, co se první přepne do pasivního režimu a druhý do aktivního, je menší, než 1ms. Po resetu procesoru je jeho role nedefinovaná a procesor odesílá packet, který zahájí rozhodování o tom, který procesor bude mít kterou roli. Tento packet je možné odeslat kdykoliv a je za všech okolností povinnost na něj odpovědět do 2500 ms (4 * doba, za kterou procesor po resetu začíná komunikovat), jinak se spouští služba obnovy paměti druhého procesoru. Pokud na procesor v pasivním režimu nepřijde požadavek na přepnutí rolí do definované doby, tak také spouští službu obnovy paměti[1]. Dále je definována minimální doba, za kterou lze přepnout role. Pokud procesor v aktivním režimu odešle požadavek na přepnutí role dříve, tak je zamítnut, a procesor v pasivním režimu sám odešle tento požadavek po uplynutí této minimální doby. V tomto případě procesor v aktivním režimu také nastavuje časovač na maximální dobu pro přepnutí rolí pro případ, když by se procesor v pasivním režimu dostal do nezotavitelného stavu. Služba přepínání rolí tedy reaguje na události, kdy přepnutí rolí nenastalo do definované doby a kdy druhý procesor přestal odpovídat, a funguje tedy jako ochrana proti výpadku druhého procesoru. Na procesoru v pasivním režimu se zpravidla neděje nic, co by mohlo ohrozit jeho chod, a je to v podstatě WDT (watchdog timer) pro procesor v aktivním režimu.

Přepnutí nebo vyhodnocení rolí po resetu obou procesorů vždy začíná odesláním packetu s udevedým požadavkem, tento požadavek a všechny další packety až do vyhodnocení rolí mají obsah s následující strukturou:

```
typedef struct Role_arbiter_payload {
    // set by higher prio MCU (lower prio does not reply if set)
    //  - if not set then other side must always reply
    bool commit;
    // applies when commit set, request must match response
    MCU_role source_MCU_requested_role;
    MCU_role target_MCU_requested_role;
    // source (local) priority and expected target priority
    uint16_t source_MCU_priority;
    uint16_t target_MCU_priority;
} Role_arbiter_payload_t;
```

[1] v případě, že tato služba paměť není schopna z nějakého důvodu obnovit, tak to procesor zkouší dokola s periodou maximální doby pro přepnutí rolí

Vyhodnocení role má dvě fáze. Nejprve si procesory vymění vlastní prioritu a poté procesor s vyšší prioritou rozhodne, který z procesorů bude mít kterou roli. Finální rozhodnutí potvrzuje nastaveným bitem *commit* a na tento packet procesor s nižší prioritou už neodpovídá. Procesor v aktivním režimu (nebo s nedefinovanou rolí) má vždy svou výchozí prioritu[1], procesor v pasivním režimu má vždy vyšší prioritu, než je nejvyšší možná výchozí priorita.

Během přepnutí role se nuluje externí WDT a na procesoru v pasivním režimu se spouští periodické nulování externího WDT. Pokud v pasivním režimu procesor od satelitu přijal datový buffer s upgrade firmware, tak tento upgrade zahájí po přepnutí do aktivního režimu. Maximální doba, do kdy se musí přepnout role, je nastavena na 25 minut, je tedy možné spouštět velice sofistikované testy, případně testovat funkcionalitu externích WDT, které mají timeout 10 minut, testovat úsporné režimy apod. Minimální doba, od kdy je možné přepnout role, je nastavena na jednu minutu. Tato hodnota vychází z odhadu nejhoršího případu, kdy spuštění všech testů bude mít za důsledek zapsání 1KB do logu událostí, a k naplnění logu tedy dojde až za 40 minut od posledního vyčtení logů. Dá se očekávat, že k vyčtení logů dojde jednou za 30 minut, a tímto způsobem se zabrání stavu, kdy už do logu událostí nebude možné zapisovat, protože bude plný.

3.5.4 Obnova paměti při výpadku

Služba obnovy paměti se spouští v případech uvedených v 3.5.3, kdy druhý procesor přestane komunikovat po hlavním komunikačním rozhraní a je nutné ho z tohoto stavu nějakým způsobem probrat. V případě, že procesor v aktivním režimu zjistí, že ten druhý přestal reagovat, tak se přepíná do pasivního režimu, služba spouštění testů se pozastavuje a aktivuje se komunikační rozhraní připojené na sběrnici satelitu. Dokud se druhý procesor nepodaří oživit, tak ten první zůstává v pasivním režimu a je pouze možné z něho číst log událostí, kam se detailně zapisuje průběh obnovy druhého procesoru. Zde je myšlenka taková, že pokud jeden procesor běží a komunikuje se sběrnicí satelitu, tak logicky musí být obsah jeho paměti v naprostém pořádku. Pokud tedy touto svojí pamětí přes BSL přeprogramuje druhý procesor a ten stále nefunguje, pak na něm musela nastat neopravitelná hardwarová chyba, případně samotný kód BSL je poškozený. V tomto případě se tedy alespoň dozvíme, že experiment je u konce.

Proces oživení druhého procesoru má dvě fáze. V první řadě si procesor

[1] výchozí priorita vzniká složením 1 byte z *randseed* (nižší byte) a 1 byte podle pinu MCU_ID (vyšší byte) pro případ selhání jednoho z těchto zdrojů

resetuje vlastní komunikační rozhraní, aby se nestalo to, že je chyba na jeho straně, a zkusí druhý procesor zresetovat. Následně se čeká 2000ms (3 * doba, za kterou procesor po resetu začíná komunikovat) zda začne komunikovat. Oba procesory po resetu odesílají packet na adresu portu služby obnovy paměti, který je za normálních okolností zahozen. Nicméně na tento packet v tento moment služba čeká, a pokud ho druhý procesor do dané doby odešle, tak se služba obnovy pozastavuje a proces oživení je u konce.

V opačném případě začíná druhá fáze oživení, která probíhá pomocí BSL. V tento moment se komunikační rozhraní znova resetuje a nastavuje se na něj filtr, který zaručuje, že se odešlou pouze packety služby obnovy paměti, a vypne se potvrzování příchozích packetů, na které BSL reaguje chybovým kódem BSL_HEADER_INCORRECT (0x51). Provede se BSL resetovací sekvence (viz 2.5) a odešle se heslo, které by pravděpodobně mohlo být správné, tedy obsah lokální tabulky přerušovacích vektorů. Pokud je heslo špatné, tak dojde ke smazání obsahu paměti druhého procesoru, pokud druhý procesor má náhodou vypnuté mazání paměti při odeslání špatného BSL hesla a odpoví zprávou s obsahem BSL_PASSWORD_ERROR, tak je stejně nutné obsah jeho paměti smazat. Po smazání paměti je tedy nutné ještě jednou zadat výchozí heslo a pak už lze druhý procesor začít programovat. Do druhého procesoru se kopíruje veškerá kódová paměť systému a CRC tabulky, kódová paměť testů a výchozí obsah tabulky přerušovacích vektorů. Nakonec se resetuje stav komunikačního rozhraní do výchozího stavu a provede se reset druhého procesoru, který by se už měl probrat.

V tento moment se opět čeká na packet adresovaný službě obnovy paměti a pokud ani tentokrát nedorazí, tak se služba pozastavuje a čeká na dalsí spuštění. Pokud během procesu obnovy paměti dojde k nějaké nečekané chybě, například druhý procesor se nepřepne do BSL režimu nebo přestane potvrzovat příchozí packety, tak se také služba pozastavuje a čeká na další spuštění. Linková vrstva komunikačního rozhraní je implementována tak, že s velkým počtem chyb během přenosu dat je schopna si poradit sama. Jde hlavně o chyby typu timeout potvrzení nebo neplatné CRC odeslaného packetu a v takových případech zkusí packet odeslat ještě několikrát před tím, než skončí s chybovým kódem (více v 4.3.1). Proces obnovy paměti lze považovat za velice spolehlivý, zajišťuje mimo jiné také stabilitu systému při výpadku napájení během upgrade firmware nebo pokud nějaký spuštěný test dostane paměť procesoru do nezotavitelného stavu. Nevýhoda spočívá pouze v tom, že v případě odeslání špatného hesla se paměť kompletně maže a log událostí a systémové statistiky jsou nenávratně ztraceny.

3.5.5 Spouštění testů

Služba spouštění testů reaguje na událost přepnutí role procesoru. Nejprve se ukončí právě spuštěný test, pokud tedy zrovna nějaký běží, a v aktivním režimu se začnou sekvenčně spouštět jednotlivé nahrané testy. V pasivním režimu služba pouze poskytuje možnost spustit konkrétní test na vyžádání procesoru v aktivním režimu, sama od sebe však nedělá nic.

V aktivním režimu se prochází CRC tabulka testů a hledají se platné záznamy[1]. Platný CRC záznam znamená spustitelný test, jehož vstupní bod (entry point) musí vždy ležet na začátku paměťové sekce, na kterou záznam odkazuje. Tento entry point se použije jako adresa, od které proces, ve kterém test bude spuštěn, začne vykonánat kód. Testy by neměly spoléhat na obsah paměťové sekce vyhrazené pro data testů, tato sekce se sice nikdy neinicializuje, nicméně pokud test vyžaduje mít trvale uložená data, která přežijou mezi jednotlivými spuštěními testu, tak by měl implementovat detekci, zda od minulého spuštění nedošlo k přemazání paměti druhým procesorem (viz 3.5.4). Pokud došlo ke smazání paměti, pak je celá paměť vyplněná hodnotou 0xFF. Prvním parametrem při spuštění testu je ukazatel na 32-bitovou hodnotu (tzv. *persistent_context*), která je vyhrazena pro daný test a jejíž obsah se uchovává mezi jednotlivými spuštěními testu. Pro tuto proměnnou se detekce smazání paměti děje automaticky a po přemazání paměti je vynulována. Dále pokud test vyžaduje trvale uložená data, tak by do paměti, kde tyto data leží, neměl zapisovat žádný jiný test.

Každý test se spouští v procesu s nižší prioritou, než je priorita událostí, na které musí systém reagovat, a na obsah testu (tedy na to, co konkrétně bude test dělat) se nekladou žádné nároky. Test může libovolně využívat služby poskytované operačním systémem a ovladače, spouštět nové procesy, případně spustit sám sebe na druhém procesoru, zaslat mu parametr spuštění a dále s ním komunikovat přes hlavní komunikační rozhraní, nicméně spuštění všech testů by celkem nemělo trvat déle, než je maximální doba pro přepnutí rolí. Operační systém má vlastní správu zdrojů, takže veškeré systémové zdroje není třeba v rámci testu uvolňovat, děje se to automaticky po ukončení procesu s testem. Tato funkcionalita je zde z toho důvodu, že pokud je nutné se přepnout do pasivního režimu v moment, kdy je spuštěn nějaký test, tak je test ukončen a všechny jeho zdroje jsou uvolněny automaticky, Kód testu pak nemá žádný dopad na stav systému a není nutné kvůli tomu resetovat systém, více v 4.2.1.

Na každém procesoru mohou být nahrané jiné testy, případně ty samé testy v různém pořadí. Spuštění aktuálního testu na procesoru v pasivním

[1] platný záznam musí mít velikost sekce alespoň 3 byte a menší než 8KB

režimu tedy probíhá tak, že se odešle pouze CRC kódu testu a parametr pro jeho spuštění. Procesor v pasivním režimu pak projde CRC tabulku testů a pokud najde test s uvedeným CRC, tak ho spustí stejným způsobem, jako by ho spustil v aktivním režimu, a předá mu zmíněný parametr. Na procesoru v pasivním režimu by se neměl spouštět žádný kód, který by mohl ohrozit jeho bezpečný chod. Test po spuštění může číst globální proměnnou *role_current* a podle jejího obsahu se zachovat. Tímto způsobem tedy lze testovat například další nevyužitá komunikační rozhraní nebo úsporné režimy, kdy se lze nechat probudit procesorem v pasivním režimu po nějaké konkrétní době apod.

V aktivním režimu se po ukončení každého testu nuluje externí WDT a do logu událostí se zapisuje návratová hodnota testu. Systém dále pro každý test udržuje statistiky kolikrát byl spuštěn, kolikrát skončil s nenulovou návratovou hodnotou a kolikrát došlo k resetu procesoru po jeho spuštění (*crash_count*). Pokud čítač *crash_count* dosáhne hodnoty 10 a ze statistiky bude patrné, že jeho spuštění způsobí reset procesoru alespoň v 50% případů, pak ho systém odmítne znovu spustit až do vynulování statistiky daného testu (během upgrade firmware). Sekce paměti, která obsahuje tyto statistiky, se odesílá v rámci vyčítání logů na komunikační rozhraní satelitu, viz 3.5.2.

3.5.6 Vzdálený upgrade firmware

Služba upgrade firmware se spouští po tom, co se do procesoru v pasivním režimu přes komunikační rozhraní satelitu nahraje kód, který upgrade provede. Konkrétně se služba spouští v moment, kdy se režim procesoru mění na aktivní, tedy před tím, než se na něm začnou spouštět testy. V tomto stavu jsou ideální podmínky pro provedení upgrade firmware, protože procesor nemusí odpovídat na požadavky od rozhraní satelitu, kódovou paměť má zaručeně v konzistentním stavu a má celých 25 minut na to, aby odeslal požadavek na přepnutí rolí. O tom, že se na procesoru provádí upgrade firmware, druhý procesor neví.

Spustitelný kód firmware upgrade se přes rozhraní satelitu ukládá na konec paměti FRAM a jeho maximální velikost je 64KB. Satelit nejprve pošle jeho délku, a poté následuje buffer s kódem firmware upgrade, na jehož konci je CRC otisk kódu. Po ukončení přenosu je jak délka tak celý buffer chráněn proti zápisu pomocí jednotky MPU. Služba nejprve pozastaví interní WDT a ověří platnost CRC kódu firmware upgrade. Pokud je otisk platný, tak se nuluje uložená délka kódu firmware upgrade, není tedy možné ho spustit dvakrát. Poté se skočí na adresu, kam je kód firmware upgrade

uložen, kde se tedy očekává jeho vstupní bod.

Kód firmware upgrade má maximální velikost 64KB, je tedy možné kompletně přepsat obsah jak kódové paměti systému tak testů, případně do procesoru nahrát zcela nový systém. Kód by neměl obsahovat reference na žádné globální funkce, které sám přepisuje, tedy neměl by například používat funkci na nulování paměti (*zerofill()*), pokud její obsah zrovna přepisuje nebo pokud ji přesunul na jinou adresu. Kód dále může do paměti nahrát nové testy a příslušné CRC záznamy, nulovat systémové statistiky, případně využít BSL a testy nahrát na druhý procesor. Firmware upgrade je spuštěn v obsluze události s nejvyšší prioritou a před spuštěním neběží žádné časovače, takže provádění kódu nebude přerušeno žádnou interní událostí. Pokud by mělo dojít k přepsání kódu komunikačního rozhraní, tak je nutné toto rozhraní nejprve uvolnit, a pokud se bude přepisovat kód ovladače IO portů, tak je nutné tyto ovladače portů uvolnit. Změny v kódu systému je vhodné zakončit resetem procesoru, protože návratová adresa (návrat z kódu firmware upgrade) už nemusí být platná, případně CRC tabulky, které se používají ke kontrole integrity paměti v době nečinnosti systému, už mohou ležet na jiné adrese apod.

Komunikační rozhraní satelitu nerozlišuje to, se kterým procesorem právě komunikuje, oba procesory se pro něj jeví jako jedno zařízení, takže se kód firmware upgrade nahraje do náhodného procesoru. Kód může číst hodnotu na pinu *MCU_ID* a podle toho rozhodnout, zda změny provede na aktuálním procesoru, nebo na druhém procesoru pomocí BSL. Takto lze tedy jednotlivé testy adresovat na konkrétní procesor. Je nutné počítat s tím, že na palubě satelitu může kdykoliv dojít k výpadku napájení, proto upgrade systémového kódu na obou procesorech najednou bych považoval za poměrně riskantní záležitost. Pokud se pouze jeden procesor během firmware upgrade dostane do nezotavitenlého stavu, pak ho druhý procesor oživí pomocí BSL, viz 3.5.4.

4 Softwarové vybavení

Software projektu má modulární vícevrstvou architekturu, kterou lze rozdělit na infrastrukturu a aplikační logiku. Aplikační logika systému využívá služby, které infrastruktura poskytuje, a je z velké části popsána v předchozí kapitole. Infrastrukturu lze dále dělit na knihovnu ovladačů pro MSP430, na operační systém (OS) a na komunikační rozhraní. Knihovna ovladačů a OS jsou dva naprosto nezávislé projekty, ovladače lze použít zcela samostatně a OS nezávisí na této konkrétní knihovně ovladačů. Komunikační rozhraní lze dělit na linkovou a transportní vrstvu. Linková vrstva komunikačního rozhraní závisí pouze na ovladačích pro UART a časovač, poskytuje služby pro spolehlivý a potvrzovaný přenos paměťových bloků a je kompatibilní s BSL protokolem. Transportní vrstva závisí na OS a na (nějaké) linkové vrstvě. Transportní vrstva poskytuje služby pro blokující a asynchronní odesílání a příjem packetů, registrování portů a adresování packetů na číslo portu a zajišťuje fragmentaci odesílaných packetů a defragmentaci přijímaných packetů. Software projektu má dále modul pro komunikaci s rozhraním satelitu, který využívá ovladače pro SPI rozhraní, DMA řadič a časovač a také některé služby OS. Tento mudul je poměrně specifický a jeho popis a popis komunikačního protokolu lze najít v dokumentu, který není součástí této práce.

4.1 Knihovna ovladačů pro MSP430

Procesory řady MSP430 mají na čipu standartní periferie (viz 2) a přístup k nim je do jisté míry unifikovaný, některé periferie a rozvržení jejich řídících registrů v paměti se za celou dobu vývoje nových modelových řad nezměnily téměř vůbec, například časovače. Každá periferie má vždy definovaný tzv. *base*, tedy adresu, na které lze zpravidla najít hlavní řídící registr periferie, a *offsety* dalších řídících registrů, tedy pouze rozdíly skutečné adresy a *base*. Periferie stejného typu mají tedy vlastní *base* adresu a společné *offsety*. *Base* adresy pro každou periferii jsou definované v příslušném *msp430xx.h* souboru, který je standartní výbavou překladače. *Offsety* jsou definované pouze pro novější modelové řady (konkrétně MSP430F5xx, MSP430F6xx a MSP430FRxx).

Tabulka přerušovacích vektorů je na všech modelech mapovaná na konec adresního prostoru adresovatelného 16-ti bity, její velikost závisí na počtu přerušovacích vektorů a resetovací vektor, tedy adresa, na kterou se skočí po

resetu procesoru, je vždy na adrese 0xFFFE. Tabulku lze na novějších modelech s flash pamětí (konkrétně MSP430F5xx a MSP430F6xx) a na všech modelech s FRAM pamětí (MSP430FRxx) přemapovat na horní konec paměti RAM (nastavením bitu SYSRIVECT v registru SYSCTL).

Firma Texas Instruments pro periferie všech procesorů MSP430 poskytuje vlastní knihovny, nicméně jejich použití je často velice krkolomné, jejich kód je poměrně neefektivní a poskytuje jen omezenou množinu operací s danými periferiemi. Dále nenabízí možnost dynamicky měnit obsah tabulky přerušovacích vektorů a je lepší se jí zcela vyhnout a pracovat rovnou s funkčními registry procesoru. Na internetu lze nalézt pár dalších pokusů o implementaci knihovny ovladačů pro MSP430, bohužel žádný z těchto projektů neodpovídá požadavkům na software pro tento experiment, neexistujou na ně žádné automatické testy a ani se nevyplatí je použít jako základ pro vývoj knihovny vlastní.

Mezi hlavní požadavky na knihovnu ovladačů patří odolnost proti souběžnému přístupu (*thread safety*) a dynamické registrování obsluhy přerušení. V tomto projektu nelze předpovídat, které přerušovací vektory budou použity pro který účel a je velice pravděpodobné, že dva různé testy budou potřebovat zaregistrovat vlastní obsluhu pro ten samý přerušovací vektor. Dále je zde požadavek na to, aby tato obsluha mohla ležet kdekoliv v paměti (ne pouze v adresním prostoru adresovatelném 16-ti bitovou adresou) a aby bylo možné jí volat s přednastavenými parametry. Tento přístup je velice výhodný v moment, kdy stejný driver ovládá dvě periferie, obsluha přerušení pro oba případy volá tu samou funkci, ale pokaždé s jiným parametrem, ve kterém je například ukazatel na řídící strukturu té konkrétní periferie. Mezi další požadavky patří zapouzdření logiky ovladače a poskytnutí jen nezbytně nutných globálních funkcí, aby se minimalizoval počet chybových stavů, dále stavovost (*statefulness*) řídících struktur ovladačů a možnost je kdykoliv uvolnit (*dispose*) a tím uvést periferní zařízení nebo přerušovací vektor do původního stavu. V neposlední řadě se požaduje, aby používání této knihovny bylo intuitivní a nebylo náchylné k chybám a aby knihovna uživatele nijak neomezovala a podporovala všechny způsoby použití dané periferie. Všechny ovladače by měly fungovat na pokud možno co největším počtu modelových řad procesoru MSP430, minimálně na MSP430F5xx a MSP430FRxx, které jsou použité v tomto projektu. Tato kapitola popisuje implementaci jednotlivých ovladačů a způsoby použití, automatické testy a příklady použití lze nalézt na githubu[1]. Detailní informace lze nalézt přímo ve zdrojových kódech knihovny.

[1] https://github.com/mutant-industries/PrimerOS-test-project

4.1.1 Konvence a datové struktury

Všechny ovladače, které nemají formu *header-only* knihovny, mají hierarchické datové struktury a technika programování velice připomíná objektově orientovaný přístup (OOP). Hlavní myšlenka spočívá v tom, že pokud je k dispozici ukazatel (pointer) na srukturu, tak je to zároveň pointer na jeho první atribut. Lze tedy pointer na tuto strukturu vždy přetypovat na pointer na typ jejího prvního atributu. Například máme-li definovanou strukturu

```
typedef struct Vector_slot {
    // enable dispose(Vector_slot_t *)
    Disposable_t _disposable;
    ...
} Vector_slot_t;
```

pak lze pointer na strukturu typu *Vector_slot_t* vždy přetypovat na pointer na strukturu typu *Disposable_t*. Pomineme-li nutné přetypování, tak lze tímto způsobem dosáhnout dědičnosti známé z objektově orientovaných jazyků, tedy toho, že struktura *Vector_slot_t* dědí všechny atributy struktury *Disposable_t*. Dalším pojmem známým z OOP je instanční metoda.

```
typedef struct Vector_handle Vector_handle_t;

struct Vector_handle {
   ...
    // -------- public --------
    // trigger interrupt (set corresponding IFG)
    uint8_t (*trigger)(Vector_handle_t *_this);
    ...
};
```

Tento příklad uvádí funkční pointer na struktuře, a pokud první parametr volání je pointer na strukturu samotnou, pak to lze označit jako volání metody. Lze zavést konvenci, že funkční pointery, jejichž název začíná podtržítkem, jsou *private* a všechny ostatní jsou *public*. Takové volání metody je poměrně nepraktické, nicméně lze pro něj definovat makro:

```
#define _vector_handle_(_handle)   ((Vector_handle_t *) (_handle))

#define vector_trigger(_handle)                                    \
          (_vector_handle_(_handle)->trigger(_vector_handle_(_handle)))
```

Takto lze tedy volat metodu *trigger* na libovolné struktuře, která dědí strukturu *Vector_handle_t*. Volání funkce, jejíž adresa je ve funkčním pointeru

na struktuře, má v porovnání s voláním globální funkce minimální výkonnostní dopad, kontrétně jde pouze o jednu instrukci navíc, protože pointer na danou strukturu před voláním už musí být v registru procesoru, přes který se předává první parametr. Dalšími pojmy jsou konstruktor, destruktor a zapouzdření. Všechny ovladače poskytují pouze jednu globální funkci pro registrování ovladače na dané struktuře, tuto funkci lze označit jako konstruktor. Tato funkce má vždy jasně definované povinné parametry, které potřebuje pro nastavení atributů daného ovladače, jako je například již zmíněná *base* adresa registrů dané periferie nebo číslo přerušovacího vektoru. Konstruktor pouze provede nastavení atributů a funkčních pointerů na dané struktuře a vzdledem k tomu, že vždy jde o pointery na funkce definované jako *static*, tak lze hovořit o zapouzdření (ukrytí) rozhraní modulu. Konstruktor neprovádí dynamickou alokaci paměti a vždy očekává pointer na strukturu ovladače, kterou má inicializovat, jako první parametr.

```
void UART_driver_register(UART_driver_t *driver, uint16_t base,
    uint8_t vector_no);
```

Samotný modul ovladače lze tedy chápat jako třídu a ukazatel na strukturu ovladače jako referenci na objekt. Všechny datové struktury ovladačů jsou oddděné od třídy *Dispose_hook_t*, která má jedinou metodu.

```
typedef struct Dispose_hook Dispose_hook_t;
typedef void (*(*dispose_function_t)(Dispose_hook_t *))(void);

struct Dispose_hook {
    dispose_function_t _dispose_hook;
};
```

Typ *dispose_function_t* je pouze zjednodušený, protože jazyk C nedovoluje rekurzivní definici typu, nicméně jde o pointer na funkci s jedním parametrem (typu pointer na strukturu typu *Dispose_hook_t*), která vrací pointer na funkci toho samého typu. Fakt, že všechny struktury ovladačů dědí *Dispose_hook_t* znamená, že pointery na tyto struktury lze vždy přetypovat na pointery na funkci typu *dispose_function_t*. Konstruktor struktury každého ovladače nastavuje tento pointer na funkci, která daný ovladač uvolní a nastaví danou periferii do výchozího stavu, lze tedy označit jako destruktor. Ovladač, který dědí jiný ovladač, vždy volá rodičovský konstruktor a předává mu pointer na jeho destruktor. Je zodpovědností rodičovského ovladače vrátit pointer na tento destruktor po skončení jeho vlastního destruktoru. Všechny ovladače lze uvolnit použitím makra *dispose()* a následuje jeho definice a kód, který volá samotnou hierarchii destruktorů.

```
#define dispose(_handle) \
    __do_dispose((Dispose_hook_t *) (_handle));

void __do_dispose(Dispose_hook_t *handle) {

    if ( ! handle || ! handle->_dispose_hook) {
        return;
    }

    interrupt_suspend();

    dispose_function_t dispose_hook = handle->_dispose_hook;
    // dispose() thread-safety
    handle->_dispose_hook = NULL;

    interrupt_restore();

    while (dispose_hook) {
        dispose_hook = (dispose_function_t) (*dispose_hook)(handle);
    }
}
```

Kód je optimalizovaný z hlediska využití zásobníku a z uvedeného kódu je patrné, že nezáleží na délce hierarchie destruktorů. Třídu ovladače, která nemá v konstruktoru parametr typu *dispose_function_t*, lze podle konvence označit jako *final*, tedy jako třídu, která nelze dále dědit. Dále lze zavést konvence pro *private* atribut jako atribut, jehož název začíná podtržítkem, a makra, u kterých lze podle názvu rozeznat, zda jde o *getter* nebo *setter*, tedy je vždy patrné, zda je dovoleno hodnotu atributu měnit.

Vzhledem k tomu, že všechny ovladače nějakým způsobem dědí vlastnosti ovladače pro manipulaci přerušovacího vektoru, se tato technika ukázala být velice výdodná a dobře se s ní pracuje. Zapouzdření a uvedené konvence omezují množství chybových stavů na minimum a náchylnost k chybnému použití je také minimální, nelze například použít driver bez předchozího volání jeho konstruktoru, které samo o sobě vynucuje zadání všech povinných parametrů a zaručuje konzistentní stav. Dále z návratové hodnoty metod ovladače lze poznat, zda byl ovladač jíž uvolněn nebo zda byly zadány korektní parametry. V neposlední řadě makra, která volají metody ovladačů, definují standartní rozhraní, a pokud například operační systém toto rozhraní používá, pak ho lze použít s jinou knihovnou ovladačů, která pouze implementuje toto rozhraní a není nucena používat uvedenou techniku.

4.1.2 WDT a přerušovací systém

Ovladače pro interní WDT (watchdog timer) a stav přerušovacího systému mají formu *header-only* knihoven, obsahují tedy pouze makra, která přímo manipulují příslušné řídící registry. Oba ovladače dále poskytují funkci uložení aktuálního stavu[1] řídícího registru do lokální proměnné a jeho následné obnovení, což umožňuje vnořené použití. Přerušovací systém je možné vypnout, pozastavit a posléze obnovit, případně pozastavit a zároveň nastavit WDT na požadovaný interval a posléze obnovit jak stav WDT tak stav přerušovacího systému. WDT je možné použít v režimu *interval_mode*, tedy jako standartní časovač, je ale nutné předtím registrovat obsluhu příslušného přerušovacího vektoru a povolit na něm přerušení. Tento přístup je velice výhodný při ladění systému, protože WDT timeout spouští pouze obsluhu přerušení a nedochází k resetu procesoru.

4.1.3 Zásobník

Ovladač zásobníku je poněkud specifický tím, že umožňuje pouze manipulaci obsahu registru SP (stack pointer), ulkádání obsahu registrů procesoru (kontextu) na zásobník, jejich obnovu a také inicializaci kontextu včetně přednastavení návratových adres na libovolném místě v datové paměti. Jde tedy o funkce, které za normálních okolností nemají žádné využití, nicméně jsou klíčové pro operační systém. Knihovna má opět formu *header-only* a poskytuje tedy abstraktní vrstvu pro všechny operace, které jsou nutné pro implementaci operačního systému na daný procesor. Je nutno podotknout, že návratová adresa se na procesorech MSP430 na zásobník ukládá zcela odlišným způsobem během standartního volání podprogramu (instrukce *CALL* a *CALLA*) a během obsluhy přerušení. První a druhá část (2 byte) návratové adresy se v těchto případech ukládjí v opačném pořadí, přerušení k vyšším dvěma bytům adresy přidává aktuální obsah SR (status register). Instrukce *CALL* ukládá návratovou adresu dlouhou pouze 2 byty. Z toho mimo jiné plyne, že instrukce *RET*, *RETA* a *RETI* pro návrat z podprogramu a z obsluhy přerušení nejsou v žádném případě kompatibilní.

Pro manipulaci obsahu SP překladače poskytují vlastní (*intrinsic*) funkce, lze s nimi však zapisovat a číst pouze 16 bitů registru SP. Pravděpodobně se při rozšíření procesoru MSP430 o CPUX instrukce na tento detail zapomnělo a pokud situace vyžaduje manipulaci celého SP registru (20 bitů), tak v případě TI překladače nezbývá než hodnotu nejprve zapsat do globální proměnné, kterou pak v assembleru další instrukcí zapsat do registru SP.

[1] makra *interrupt_suspend_xx()* a *WDT_backup_xx()*

4.1.4 Přerušovací vektory

Procesory MSP430 mají šesnáctibitové přerušovací vektory, obsluha přerušení z principu nemá žádný parametr a musí být definovaná jako *__interrupt*, což má za důsledek to, že funkci linker mapuje do paměti adresovatelné 16-ti bity a překladač pro ni generuje odpovídající *prolog* a *epilog*, tedy uložení všech tzv. *scratch* registrů na zásobník, a pro návrat z obsluhy instrukci *RETI*. Aby se předešlo těmto komplikacím a zaručila se dostatečná flexibilita pro registrování obsluhy přerušení, tak tato knihovna definuje slot:

```
typedef struct Vector_slot {
    // enable dispose(Vector_slot_t *)
    Disposable_t _disposable;
    // vector interrupt service handler arguments
    void *_handler_arg_1;
    void *_handler_arg_2;
    // number of interrupt vector
    uint8_t _vector_no;
    // original vector handler, restored on dispose
    uint16_t _vector_original_content;
    // vector interrupt service handler
    vector_slot_handler_t _handler;
} Vector_slot_t;
```

Kromě slotu definuje šablonu pro generování obsluh přerušení:

```
#define __interrupt_handler_generator(no)                          \
__naked __interrupt void __interrupt_handler_name_gen(no) () {  \
    __asm__("  "__pushm__" #5, R15");                              \
    Vector_slot_t *slot = &_vector_slot_array[no];                 \
    slot->_handler(slot->_handler_arg_1, slot->_handler_arg_2); \
    __asm__("  "__popm__" #5, R15");                              \
    reti;                                                          \
}
```

Během překladu se definuje požadovaný počet slotů (*__vector_slot_array*) a pro každý slot se vygeneruje příslušná obsluha přerušení, jejíž adresa ve výchozím stavu není zapsána do žádného přerušovacího vektoru. Dynamické registrování obsluhy přerušení pro vektor tedy probíhá tak, že se nejprve najde prázdný slot, nastaví se na něm pointer na funkci, kterou má obsluha přerušení volat, a nastaví se na něm dva (volitelné) parametry, které se této funkci při volání předají. Následné se do přerušovacího vektoru s požadova-

ným číslem[1] zapíše adresa příslušné obsluhy přerušení, která pracuje s tímto slotem. Tímto způsobem lze tedy se zanedbatelným výkonnostním dopadem dynamicky nastavit obsluhu přerušení pro libovolný vektor na funkci, která leží kdekoliv v paměti a kterou není nutné definovat jako *__interrupt*, a pokud nastane přerušení tohoto konkrétního vektoru, tak se této funkci předají dva nastavitelné parametry typu datový pointer. Při uvolnění slotu se obnovuje původní obsah přerušovacího vektoru. Kód obsluhy se do assembleru na TI překladači i na MSP430-gcc přeloží následovně:

```
     _vector_slot_0():
144F              PUSHM.A #5,R15
002C D216         MOVA    &0x0d216,R12
002D D21A         MOVA    &0x0d21a,R13
1380 D222         CALLA   &0x0d222
164B              POPM.A  #5,R15
1300              RETI
```

Je tedy patrné, že z hlediska výkonu jde navíc o dvě instrukce naplnění registrů procesoru konstantou, instrukci volání podprogramu a instrukci návrat z podprogramu. Pro případ překladače MSP430-gcc je použití slotu z hlediska výkonu srovnatelné se standartní obsluhou přerušení, protože při překladu pomocí MSP430-gcc obsluhy přerušení na zásobník vždy ukládají obsah všech registrů procesoru, což je často zbytečné. To je nicméně také důvod, proč je obsluha definovaná *__naked* a proč je explicitně uvedeno, že se na zásobník mají uložit pouze *scratch* registry procesoru.

Počet slotů se definuje při kompilaci a není možné ho později měnit. Na základě požadovaného počtu slotů preprocesor generuje daný počet obsluh přerušení. Samo generování je založeno na tom, že preprocesor do jisté míry zvládá rekurzi, a pro tento účel je využita knihovna Chaos Preprocessor. V tomto projektu je počet slotů nastaven na 35, což odpovídá celkovému počtu přerušovacích vektorů na procesoru MSP430FR5994.

Na procesorech MSP430Fxx nelze tabulku přerušovacích vektorů dynamicky přepisovat, protože je namapovaná do flash paměti. Je tedy nutné tabulku přemapovat do paměti RAM a ovladač tuto funkcionalitu podporuje[2] také pro všechny modely MSP430FRxx, tedy procesory s pamětí FRAM. Software tohoto projektu má tabulku přerušovacích vektorů přemapovanou do RAM z toho důvodu, že tabulka v paměti FRAM funguje zároveň jako heslo BSL. Pokud se tedy tabulka ve FRAM nějakým nestandartním způso-

[1] z čísla vektoru lze snadno dopočítat adresu, na které daný vektor leží

[2] informace o tom, jaké symboly musí být během sestavení definovány pro dosažení této funkcionality, lze nalézt v souboru *driver/config.h*

bem nepřepíše, tak se situace ohledně hádání BSL hesla velmi zjednodušuje. Heslo tedy nabývá pouze dvou hodnot a závisí pouze na tom, zda je resetovací vektor nastaven na hlavní nebo na záložní zavaděč.

Na funkcionalitě tohoto ovladače jsou založeny všechny ostatní ovladače, které knihovna poskytuje. V každém případě lze ovladač použít samostatně a při jeho registrování je možné uvést nepovinné parametry, konkrétně adresy příslušného IE (*interrupt enable*) a IFG (*interrupt flag*) registru a příslušné masky. Pokud jsou tyto hodnoty nastavené a daný vektor to podporuje, pak je možné softwarově vyvolat přerušení, čehož využívá například operační systém při přepínání kontextu, dále je možné povolovat a zakazovat přerušení na daném vektoru a nulovat bit IFG (*interrupt flag*). Ovladač lze použít na všech procesorech řady MSP430F5xx, MSP430F6xx a MSP430FRxx.

4.1.5 Časovače

Časovače na procesorech MSP430 mají jeden řídící registr, dále registr, který obsahuje aktuální hodnotu vnitřního čítače, a pro každý kanál řídící registry *CCTLn* (capture / compare control) a *CCRn* (capture / compare), maximální počet kanálů jednoho časovače je 7. Každý časovač má přidělené dva přerušovací vektory, jeden je vyhrazen pouze pro kanál *CCR0* a druhý je vyhrazen pro ostatní kanály a signalizaci přetečení. Registrace ovladače má mimo jiné povinný parametr s konfigurací děliček frekvence časovače, dálé je nutné uvést čísla obou přidělených vektorů, celkový počet kanálů a adresu *base*. Na tomto ovladači je poté možné registrovat řídící struktury (*handle*) pro každý kanál a signalizaci přetečení, které dědí vlastnosti ovladače přerušovacího vektoru. Je vždy nutné uvést typ kanálu a podle návratové hodnoty lze poznat, zda z nějakého důvodu *handle* pro daný kanál nebylo možné registrovat. Tento *handle* lze použít pro manipulaci příslušných *CCTLn* a *CCRn* registrů, lze tedy nastavovat kanál do požadovaného režimu (capture nebo compare), konfigurovat výstup kanálu, dotazovat se na jeho stav apod.

Ovladač časovače ukládá reference na řídící struktury všech kanálů, které jsou na něm registrované. Během registrace kanálů, které sdílí jeden přerušovací vektor, se na jejich rodičovské struktuře *vector* přepisuje pointer na funkci, která registruje obsluhu přerušení. Je nutné, aby se pro tento vektor alokoval slot pouze jednou a aby obsluha přerušení tohoto vektoru volala interní obsluhu časovače a předala mu jako parametr pointer na daný ovladač časovače. V rámci této obsluhy se z příslušného IV (*interrupt vector generator*) registru zjistí, na kterém kanálu přerušení nastalo, dohledá se reference na řídící strukturu pro daný kanál a poté se volá obsluha registrovaná pro daný kanál s přednastavenými parametry. Zdrojový kód této obsluhy vypadá

následnvně:

```
static void _shared_vector_handler(Timer_driver_t *driver) {
    uint8_t interrupt_channel_index;
    uint16_t interrupt_source;
    Timer_channel_handle_t *handle;

    if ( ! (interrupt_source = hw_reg_16(driver->_IV_register))) {
        return;
    }

    // IV -> channel number (0x02 - TxCCR1.CCIFG interrupt...)
    interrupt_channel_index = (uint8_t) (interrupt_source / 2 - 1);

    handle = ((Timer_channel_handle_t **)
        &driver->_CCR1_handle)[interrupt_channel_index];

    // execute handler with given handler arguments
    handle->_handler(handle->_arg_1, handle->_arg_2);
}
```

Takto je tedy možné se vyhnout použití *switche*, přeložením dostaneme:

```
        _shared_vector_handler():
   4C1F 0018         MOV.W   0x0018(R12),R15
   4F2F              MOV.W   @R15,R15
   930F              TST.W   R15
   240F              JEQ     ($C$L12)
   035F              RRUM.W  #1,R15
   835F              DEC.B   R15
   065F              RLAM.W  #2,R15
   0E4F              RLAM.A  #4,R15
   0D4F              RRAM.A  #4,R15
   0CEF              ADDA    R12,R15
   0F3F 0020         MOVA    0x0020(R15),R15
   0F3C 004E         MOVA    0x004e(R15),R12
   0F3D 0052         MOVA    0x0052(R15),R13
   00AF 004A         ADDA    #0x0004a,R15
   0F00              BRA     @R15
        $C$L12:
   0110              RETA
```

Vidíme tedy, že kromě čtení registru *IV* a testování jeho nenulové hodnoty je zde pouze šest instrukcí, které počítají adresu řídící struktury pro pří-

slušný kanál, a optimalizovaný skok na obsluhu přerušení daného kanálu (není nutné ukládat návratovou adresu). V porovnání s doporučovaným postupem, tedy použití konstrukce *switch (__even_in_range(IV, ...))*, je tato technika co se týče výkonu srovnatelná, nicméně je podstatně flexibilnější. Po uvolnění řídící struktury se vypíná přerušení pro daný kanál a po uvolnění celeho ovladače se uvolňují řídící struktury všech kánálů a obsah přerušovacích vektorů se nastavuje do původního stavu.

Přes řídící struktury kanálů lze mimo jiné kanál zapnout, vypnout, číst hodnotu vnitřního čítače a nulovat jeho obsah. Chování zapnutí a vypnutí záleží na tom, zda je kanál v režimu *capture* nebo *compare*, a obě tyto funkce manipulují hodnotu počítadla spuštěných kanálů, která je na ovladači příslušného časovače. Při vypnutí posledního kanálu se vypíná funkce celého časovače, protože v daný moment není důvod, aby časovač běžel a zbytečně spotřebovával energii. Nulování vnitřního čítače je možné pouze tehdy, když nejsou aktivní žádné jiné kanály, které by tím pádem na tomto čítači závisely. Čtení hodnoty vnitřního čítače probíhá metodou hlasování, kdy rozdíl dvou po sobě jdoucích čtení musí být menší než nějaký maximální práh (*threshold*). Čtení je takto implementováno z toho důvodu, že zdroj hodinového signálu pro časovač může být jiný než zdroj MCLK a při čtení registru může v krajních případech docházet k chybám. Na řídící struktuře kanálu časovače v režimu *compare* lze libovolně vyvolávat softwarový interrupt a nezáleží na tom, zda časovač je nebo není zapnutý.

4.1.6 CRC

MSP430FR5994 má dva hardwarové moduly pro výpočet CRC_CCITT, jeden pro vstup délky 16 bitů a druhý pro vstup délky 32 bitů. Ovladač pro výpočty CRC otisku podporuje pouze vstup délky 16 bitů a navíc poskytuje softwarovou podporu výpočtu (*fallback*) pro případ poruchy hardwarového modulu. Softwarový *fallback* má tu výhodu, že lze používat i na zařízeních, které hardwarový modul nemají, a pokud každý spuštěný proces bude pro výpočty CRC používat tento *fallback*, pak nehrozí problémy vícenásobného přístupu ke sdílenému zdroji (není nutné používat mutex).

Základní softwarové vybavení projektu hardwarový modul nepoužívá právě z toho důvodu, že by před každým výpočtem proces musel uzamknout mutex, je tedy přenechán k dispozici pro případné použití během testů, a systém pro všechny ovladače využívá softwarový *fallback*. Co se týče výkonu, tak použití hardwarového modulu je zhruba dvakrát rychlejší, než výpočet pomocí software. Vzhledem k případům použití tento rozdíl nehraje roli, systém zpravidla počítá CRC pouze pro velice malé bloky paměti (packet, záznam

v logu událostí apod.) a během kontroly integrity paměti na rychlosti výpočtu také nezáleží. Inicializační hodnota CRC ovladače (*seed*) pro paměťové bloky je 0x0000 (CRC tabulky, které generuje linker) a pro komunikační rozhraní 0xFFFF (*seed* definuje BSL protokol).

4.1.7 Vstupně-výstupní porty

Potry na procesorech MSP430 mají opět standartní rorvržení řídících registrů, každý port má definovanou *base* adresu a *offsety* zbylých registrů jsou konstantní. K registrům lze přistupovat instrukcemi pro čtení a zápis jednoho byte (X.B) případně lze ovládat zároveň dva sousední registry pomocí instrukcí čtení a zápis dvou bytů (X.W). Knihovna poskytuje makra na přímou manipulaci řídících registrů a ovladače pro snadnou obsluhu přerušení na vstupních pinech. Každý port má přiřazen jeden přerušovací vektor[1]. Registrace obsluhy přerušení probíhá stejně jako je tomu u ovladače pro časovač (4.1.5). Opět je nejprve nutné registrovat ovladač pro daný port a na něm tzv. *handle* pro požadovanou masku pinů. Na tomto *handlu* lze opět využít zděděné funkce od ovladače pro přerušovací vektor, tedy registrovat obsluhu přerušení, povolit nebo zakázat přerušení na daných pinech, vyvolat softwarové přerušení apod. Vzhledem k tomu, že jeden *handle* může obsluhovat přerušení na více pinech, tak druhým parametrem registrované obsluhy je vždy číslo pinu, na kterém přerušení nastalo. Port má alokovaný pouze jeden slot a uvolnění ovladače portu nebo samotného *handlu* uvede registrovanou masku pinů do výchozích stavu, tedy režim IO vstup, a zakáže se přerušení na dané masce pinů.

Úsporné režimy LPMx.5 procesorů MSP430, kdy se celková spotřeba pohybuje v řádu desítek až stovek nA, umožňují, aby výdrž baterií na zařízeních s těmito procesory násobně přesáhla životnost zařízení samotného. Z režimu LPM4.5 je možné probuzení pouze změnou hodnoty signálu na vstupním pinu a tento ovladač umožňuje obsluhovat přerušení po probuzení z těchto režimů. Tato funkce je podporována pouze na procesorech s pamětí typu FRAM, protože zakládá na tom, že datové struktury v paměti FRAM mohou přečkat reset procesoru. Reference na registrované ovladače se uchovávají v poli deklarovaném jako *___persistent*, tedy je uloženo v paměťové sekci, která se během startu procesoru neinicializuje. Je třeba, aby všechny registrované ovladače a *handly* ležely také v nějaké sekci, která se po startu neinicializuje. Dále je vhodné, aby se těsně před přepnutím do úsporného režimu LPMx.5 (v moment, kdy už je přerušovací systém vypnutý) zavolala funkce *IO_low_power_mode_prepare()*, která uvolní registrované sloty IO

[1] na starších modelových řadách lze generovat přerušení pouze na portech 1 a 2

portů. Volání této funkce nutné není, nicméně pokud se tak nestane a není zapnuté mapování přerušovacích vektorů do RAM, tak jejich původní obsah bude ztracen.

Po probuzení stačí zavolat funkci *IO_wakeup_reinit()*, která obnoví datové struktury do stavu před přepnutím do úsporného režimu a odemkne IO porty (bit LOCKLPM5 v registru PM5CTL0), obsluha přerušení pak proběhne standartním způsobem. Tato funkcionalita se testuje v rámci *standalone* testů OS a jednoho systémového testu v tomto projektu, jehož zdrojový kód lze nalézt v *application/LPM.c*. Tento test sám sebe spustí na procesoru v pasivním režimu, na procesoru v aktivním režimu zaregistruje obsluhu přerušení na jednom z pinů, které mají procesory propojené, a přepne se do LPM4.5. Procesor v pasivním režimu ho po nějaké době probudí změnou signálu na tomto pinu, čímž se mimo jiné testuje, zda je celý operační systém schopen uchovat svůj stav v paměti FRAM a po probuzení ho opět obnovit.

4.1.8 Komunikační rozhraní

Knihovna poskytuje ovladače pro komunikační rozhraní UART a SPI a je zde k dispozici ovladač na DMA řadič. DMA řadič má na procesoru MSP430FR5994 celkem šest kanálů a jeho ovladač se svou strukturou příliš neliší od ovladačů pro IO porty nebo časovače. Opět je nutné registrovat ovladač, na kterém lze registrovat *handle*, který poskytuje abstraktní rozhraní pro přístup k jeho řídícím registrům, a je možné na něm registrovat obsluhu přerušení standartním způsobem. DMA řadič má mnohé další způsoby využití, než v kombinaci s nějakým komunikačním rozhraním, které jeho ovladač sice podporuje, ale nebudu je zde dále rozvádět.

Oba ovladače pro UART a SPI vychází ze společné struktury generického ovladače eUSCI. V rámci tohoto nadřazeného modulu jsou vyřešeny problémy s mapováním řídících registrů a lze ho taktéž použít pro vývoj ovladače pro I^2C rozhraní. Každé z eUSCI rozhraní má přiřazen jeden přerušovací vektor, pro jehož obsluhu se vyhradí slot během registrování ovladače. Na ovladačí lze poté pro všechny události nastavit *callbacky*, tedy funkce, které se volají v moment dané události, a přednastavit jim parametry. eUSCI zařízení v režimu UART má celkem čtyři zdroje přerušení, konkrétně příjen znaku, příjem start bitu, vyprázdnění odesílacího bufferu a odeslání znaku. V režimu SPI jsou zde pouze dva zdroje přerušení - příjen znaku a vyprázdnění odesílacího bufferu. Na tyto události lze *callbacky* nastavit a poté je nutné přerušení generované danou událostí povolit. Ovladače poskytují rozhraní pro manipulaci řídících registrů, například nastavení režimu nebo přenosové rychlosti, a lze je snadno kombinovat s ovladačem pro DMA.

4.2 Operační systém

Požadavky na softwarové vybavení projektu v principu nepřipouští jinou možnost, než použití systému, který umožňuje spouštět kód v nezávislých kontextech, obsluhovat asynchronní události a podle potřeby přepínat kontext. Dále je nutné mít k dispozici prostředky, které umožní spolehlivé časování událostí a odezvu na události do nějaké přípustné doby. Tyto a mnohé další požadavky na software vedly k rozhodnutí, že součástí softwarového vybavení projektu musí být operační systém.

Operační systémy pro jednočipové procesory poskytují služby pro souběžné spouštění procesů, jejich synchronizaci a často mají plánovač, který umožňuje odezvu na externí události v reálném čase (*RTOS*). Mezi hlavní vlastnosti *RTOS* patří deterministické chování a možnost predikce chování OS za všech myslitelných okolností. Operační systémy mají zpravidla modulární architekturu a je možné je nějakým způsobem naportovat na hardware, který splňuje požadavky na jeho provoz, například nároky na paměť. Mezi hlavní představitele *RTOS* patří Embedded Linux, FreeRTOS, TI RTOS a dnes již poněkud zastaralý Contiki. Dále je k dispozici velké množství menších OS, které poskytují jen omezené možnosti, například TinyOS, scmRTOS nebo Blu.OS a lze nalézt spousty pokusů, které umožňují pouze přepínání kontextu nebo kooperativní plánování procesů. Systémy lze kategorizovat podle velkého množství parametrů, jako je například rychlost odezvy, paměťová náročnost, množství a typy hardware, na které je systém naportovaný a v neposlední řadě podle licence, která omezuje jejich použití, zda se OS stále aktivně vyvíjí nebo zda má podrobnou dokumentaci. Parametry jednoho OS mohou být pro některé případy užití velice výhodné a pro jiné případy nepřípustné, nelze tedy tvrdit, že některý konkrétní OS je nejlepší.

Přestože lze použít některé stávající řešení existuje mnoho důvodů pro vývoj nových operačních systémů. Častým důvodem je ten, že žádné z nabízených řešení nevyhovuje požadavkům pro daný projekt, případně že by úsilí upravit stávající řešení tak, aby požadavkům vyhovovalo, bylo srovnatelné s úsilím nutným k vynaložení na vývoj nového OS. Požadavky na OS pro tento projekt jsou poměrně specifické. Na jednu stranu je nutné reagovat na externí události v reálném čase, tedy je zde požadavek na asynchronní obsluhu událostí s nastavitelnými prioritami, na stranu druhou je třeba spouštět procesy (testy) a poskytnout možnost blokujícího volání při čekání na událost, tedy synchronní zpracování, a je nutné mít k dispozici spolehlivou infrastrukturu pro časování událostí. Systém poběží na procesoru MSP430 s moderní pamětí typu FRAM, která poskytuje možnost perzistentního uložení dat a OS by měl být schopen svůj stav v této paměti trvale uchovat. Dále musí být možné

využít všechny možnosti procesoru MSP430 a sám by neměl dělat žádná rozhodnutí co se týče například frekvence hodinového signálu nebo použitých časovačů a žádným způsobem jeho uživatele neomezovat. OS musí záviset na pokud možno co nejmenším množství periferních zařízení, aby se snížila pravděpodobnost jeho selhání v podmínkách na oběžné dráze. OS dále musí poskytovat velice specifické komunikační rozhraní postavené na UARTu a kompatibilní s BSL protokolem a zároveň nesmí obsahovat žádnou zbytečnou funkcionalitu navíc, případně nějakou složitou rozhodovací logiku, která může narušovat stabilitu systému. OS včetně ovladačů na hardware nesmí zabírat více než 50KB paměti pro kód i data. Uvedeným požadavkům žádné hotové řešení, které je volně k dispozici, nevyhovuje, proto bylo rozhodnuto, že pro tento projekt vznikne operační systém nový.

4.2.1 Konvence a správa zdrojů

Kód operačního systému má v základu stejné konvence jako knihovna ovladačů a jsou popsané v kapitole 4.1.1. Uvolňování ovladačů (obecně zdrojů) a popsaná hirarchie destruktorů je zde rozšířena o správu zdrojů. Jedním z požadavků na systém je ten, aby bylo možné jakýkoliv proces v náhodný moment ukončit a aby se tím uvolnily všechny zdroje, které do té doby proces stihl alokovat, jako například ovladače, obsluhy událostí nebo procesy, které z jeho kontextu byly spuštěny. Myšlenka pro implementaci správy zdrojů zakládá na tom, že ukazatele na všechny datové struktury, které budou označené jako zdroj (*resource*), bude možné přetypovat na ukazatele na struktury typu *Disposable_t*, tedy podle terminologie OOP budou mít všechny stejného předka. Prakticky to znamená, že všechny zdroje OS budou mít stejného předka jako všechny struktury ovladačů, které lze uvolnit pomocí makra *dispose()*. Pak lze strukturu *Disposable_t* rozšířit následovně:

```
struct Disposable {
    // 'resource <-> process' reference cleanup hook
    Dispose_hook_t _resource_dispose_hook;
    // owner process control block
    Process_control_block_t *_owner;
    // child hook responsible for freeing up the resource
    dispose_function_t _dispose_hook;
    // bidirectional chaining
    struct Disposable *_next;
    struct Disposable *_prev;
};
```

Zdroje lze pak řetězit, uchovávat referenci na vlastníka a zároveň mít vlastní destruktor, který zdroj uvolní ze spojového seznamu zdrojů a který po skončení vrátí ukazatel na destruktor, který si sám zdroj nastavil v konstruktoru. Obousměrné zřetězení je zde pouze z důvodu optimalizace, pro uvolnění struktury ze seznamu není nutné procházet celý seznam a hledat předchozí strukturu, jen aby se na ní mohla upravit reference *_next*.

Implementace je tedy poměrně jednoduchá a OS lze přeložit s podporou správy zdrojů i bez ní, ve výchozím stavu je funkce vypnutá. Pokud je správa zdrojů vypnutá, tak je použita původní definice struktury *Disposable_t*, zdroje je možné uvolňovat pomocí makra *dispose()*, nicméně po skončení procesu automaticky uvolněny nejsou. Důvod, proč funkci nezapínat, může být to, že nemá v daném projektu žádné opodstatnění a všechny zdroje pouze zabírají více paměti[1].

Pro volání funkčních pointerů na datových strukturách, které v předchozím textu bylo označeno jako volání metod, jsou vždy definované makra, která provádějí přetypování a tato volání výrazně zjednodušují. Častým případem je volání takové metody dejme tomu se dvěma parametry, které mají v drtivé většině případů použití stejnou hodnotu, a je proto poměrně nepraktické, že se musí pokaždé všechny uvádět. Jazyk C neumožňuje definovat výchozí hodnotu vstupních parametrů funkce pro případ, že parametry nejsou zadané, a z toho důvodu i tento problém je řešen přes makra, kterým lze zadat volitelný počet parametrů, a pro nezadané parametry jsou definované výchozí hodnoty.

4.2.2 Fronty a řazení

Všechny datové struktury, se kterými OS v základu pracuje, je možné řetězit a řadit podle priority, logika vytváření obousmerně zřetězených seznamů je v modulu *deque* (double-ended queue) a je založena na následující struktuře:

```
typedef struct Deque_item Deque_item_t;

struct Deque_item {
    // resource, enable dispose(Deque_item_t *)
    Disposable_t _disposable;
    // bidirectional chaining
    Deque_item_t *_next;
    Deque_item_t *_prev;
    Deque_item_t **_container;
};
```

[1] každý zdroj má čtyři pointery navíc, použitá paměť záleží na datovém modelu

Spojové seznamy jsou tedy obousměrně zřetězené, lze s nimi pracovat jako se zdrojem a mají odkaz na ukazatel, který odkazuje na první prvek seznamu a který lze chápat jako například ukazatel na seznam samotný nebo obecně na tzv. *container* založený na spojovém seznamu. Tento ukazatel je v paměti vždy jen jednou a všechny prvky seznamu na něj uchovávají ukazatel. Seznamy jsou vždy obousměrně zřetězené, poslední prvek odkazuje na první a první na poslední. Tato struktura je výhodná z několika důvodů. V první řadě se u každého prvku vždy ví, do kterého seznamu je zařazen (případně že není zařazen v žádném seznamu) a lze ho z tohoto seznamu vždy odebrat s konstantní složitostí. Pokud je ze seznamu odebrán jeho první prvek, pak se upravuje ukazatel na první prvek seznamu (**_container*). Dále lze vždy přistupovat na první a na poslední prvek seznamu s konstantní složitostí.

Struktura *Deque_item_t* je rozšířena o prioritu následovně:

```
typedef uint16_t priority_t;

typedef struct Sorted_set_item {
    // resource, chainable
    Deque_item_t _chainable;
    // item priority the set is sorted by
    priority_t _priority;
} Sorted_set_item_t;
```

Prvky seznamu lze nyní řadit podle priority. Algoritmus řazení je jeden a zda je seznam seřazen podle priority sestupně nebo vzestupně záleží na tom, ze kterého konce k seznamu přistupujeme. Uvažujeme-li sestupné řazení, pak algoritmus má tu vlastnost, že prvek s prioritou n je vždy do seznamu umístěn na pozici za poslední prvek s prioritou $>= n$. V praxi to znamená, že prvek s prioritou menší nebo rovnou prioritě prvku, který má v daném seznamu nejmenší prioritu, je vždy umístěn na konec seznamu, a logicky prvek s prioritou 0 je za všech okolností umístěn na konec seznamu, a tato operace má konstantní složitost. Na prvku seřazeného seznamu lze měnit prioritu, jeho pozice v seznamu je pak odpovídajícím způsobem upravena. Je nutno podotknout, že tato funkce je optimalizována pro všechny případy změn priority, a po odebrání a zpětném přidání prvku do seznamu se seznam nikdy neprochází celý. 'Změna' priority prvku na tu samou prioritu, kterou daný prvek má, má stejný dopad, jako kdyby se prvek ze seznamu nejprve odebral a pak do něj zase vložil, tedy je vždy do seznamu umístěn na pozici za poslední prvek s prioritou $>= n$. Funkce pro prácí se seznamy samy o sobě nejsou odolné proti souběžnénu přístupu (*thread-safe*).

4.2.3 Akce, fronta akcí

Všechny datové struktury OS jsou založené na strukruře typu *Action_t*, tedy ukazatel na danou strukturu lze vždy přetypovat na ukazatel na strukturu typu *Action_t*. Jedinou vyjímkou je fronta akcí (*Action_queue_t*), která sama o sobě není zdroj. Struktura *Action_t* má následující definici:

```
typedef struct Action Action_t;

typedef void * signal_t;
typedef signal_t action_arg_t;

typedef void (*action_trigger_t)(Action_t *_this, signal_t signal);
typedef bool (*action_handler_t)(action_arg_t, action_arg_t);
typedef void (*action_released_hook_t)(Action_t *_this,
    Action_queue_t *origin);

struct Action {
    // resource, sorted set item
    Sorted_set_item_t _sortable;
    dispose_function_t _dispose_hook;

    action_arg_t arg_1, arg_2;

    // -- abstract public --
    action_trigger_t trigger;
    action_handler_t handler;
    action_released_hook_t on_released;
};
```

Jde tedy o generickou strukturu, která má dva datové a dva funkční pointery. Metodu trigger lze vyvolat použitím makra *action_trigger(*, signal_t)*, a budeme-li na všechny struktury OS pohlížet jako na akce, pak lze takto dosáhnout *polymorfismu*, tedy toho, že chování triggeru závisí na tom o jaký typ akce jde. Například trigger na akci typu proces způsobí, že se proces vloží do fronty připravených procesů apod. Atributy *arg_1* a *arg_2* a funkční pointer *handler* lze v rámci triggeru využít libovolným způsobem a v kontextech různých typů akcí mohou mít zcela jiný význam a použití, případně není nutné je využívat vůbec. Modul *action* neklade na konkrétní použití žádné nároky a pouze poskytuje rozhraní pro vytváření akcí, základní operace jako je změna priority nebo odebrání z fronty akcí.

Fronta akcí (*Action_queue_t*) je datová struktura, která poskutuje rozšířené rozhraní spojového seznamu a veškeré operace s ní jsou odolné vůči sou-

běžnému přístupu (*thread-safe*). Rozhraním fronty je opět množina funkčních pointerů (metod), které se nastavují při vytvoření fronty v závislosti na vstupních parametrech. V základu lze vytvořit buď prioritní frontu, tedy spojový seznam, jehož prvky jsou seřazené podle priority, nebo frontu *FIFO* (first in, first out). Vnitřní implementace prioritní fronty vždy udržuje informaci o prioritě, kterou má tzv. *queue head*, tedy akce s nejvyšší prioritou. Na frontu lze nastavit *callback*, který se zavolá vždy v moment, kdy dojde ke změně této priority. *Callback* se volá vždy s jedním parametrem typu datový pointer, který je možné na frontě nastavit na libovolnou hodnotu (vlastník fronty), a druhým parametrem je pak tato nová priorita. Deklarace funkce pro změnu priority akce vypadá následovně:

```
bool action_set_priority(Action_t *action, priority_t priority);
```

Jednoduše lze pak dosáhnout toho, že priorita jedné akce bude vždy stejná jako priorita akce s nejvyšší prioritou v nějaké frontě akcí. Vlastník fronty se nastaví na pointer na tuto akci a uvedený *callback* na pointer na funkci pro změnu priority. Lze pak hovořit o dědičnosti priority, tedy že akce dědí prioritu dané fronty. Je možné tuto dědičnost libovolně řetězit, akce může dědit prioritu fronty a sama být v jiné frontě, jejíž prioritu dědí jiná akce. Implementace této řetězené dědičnosti je optimalizovaná vzhledem k použití zásobníku a při libovolně dlouhém řetězení velikost zásobníku nenarůstá. Tato optimalizace přináší jedno omezení, kdy změna priority jedné akce může zapříčinit změnu priority nejvýše jedné další akce. Toto omezení však znemožňuje pouze velice nestandartní způsoby použití a není nutné se nad ním dále pozastavovat. Pro celý řetězec akcí je priorita změněna atomicky a probíhá s vypnutým přerušovacím systémem.

Odebrání akce z fronty je možné provést použitím makra *action_release(*)* a není nutné jako parametr uvádět ukazatel na frontu, ve které daná akce je, protože tento ukazatel je vždy nastaven na samotné akci (*_container* na nadřazené struktuře *Deque_item_t*). Akci lze z fronty odebrat kdykoliv, z fronty se může odebrat i sama akce v rámci jejího triggeru a při přidávání akce do nějaké fronty je akce automaticky nejprve odebrána ze stávající fronty (pokud v nějaké frontě je). Vzhledem k tomu, že některé typy akcí potřebují reagovat na událost, kdy byly odebrány z fronty, lze na nich registrovat *callback*, který se volá po odebrání z fronty, a pointery na tuto frontu a na akci samotnou se do *callbacku* předají jako parametry. Pokud je akce v nějaké frontě, pak se při uvolnění (*dispose*) z fronty automaticky odebere.

Trigger na akci lze snadno kombinovat s obsluhou přerušení, tedy je možné, aby přerušení vyvolalo trigger na zvolené akci. Tato funkcionalita zakládá na tom, že mezi samotným přerušením a voláním triggeru na akci je

jeden mezikrok, který zavolá konkrétní trigger předá do něj pointer na tuto akci a případně ještě jeden libovolný parametr. Tento mezikrok poskytuje knihovna ovladačů a konkrétně by použití mohlo vypadat následovně:

```
vector_register_handler(DMA_handle,
   action(TX_channel_action)->trigger, TX_channel_action, NULL);
```

V tomto případě se trigger na akci TX_channel_action spustí v kontextu přerušení příslušného DMA kanálu. Trigger lze volat z libovolného množství kontextů a je možné využít parametr triggeru (signal) například k tomu, aby daný trigger poznal, ze kterého kontextu je spuštěn.

Fronta akcí dále poskytuje funkci *trigger_all()*, tedy spuštění triggeru na všech akcích, které jsou v dané frontě. Tato funkce sekvenčně od nejvyšší priority prochází všechny akce v dané frontě, spouští na nich trigger a je odolná vůči všem manipulacím s danou frontou, které během *trigger_all()* mohou nastat v náhodný okamžik (například z kontextu obsluhy přerušení). Toho je dosaženo tím způsobem, že fronta vždy udržuje referenci na příští akci, na které se má zavolat trigger, zatímco probíhá trigger na akci předchozí (*iterator*). Pokud se v nějaký okamžik z fronty odebere právě ta akce, na kterou odkazuje *iterator*, tak se *iterator* pouze posune na následující akci. Stejné pravidlo platí pro případ, kdy se mění priorita akce, na kterou odkazuje *iterator*, a její pozice ve spojovém seznamu se může změnit, aby se zachovalo řazení fronty. Nicméně tento případ je komplikovanější, může se stát, že během jednoho volání *trigger_all()* se na jedné akci s dynamickou prioritou spustí trigger dvakrát nebo případně ani jednou. Pokud by toto chování mělo z nějakého důvodu vadit, pak lze frontu nastavit do režimu, kdy během *trigger_all()* sice na všech akcích je možné prioritu měnit, ale jejich pozice ve spojovém seznamu zůstane zachovaná. Takto nastavená fronta pak zaručuje, že *trigger_all()* provede trigger na všech akcích právě jednou, ale zároveň hrozí, že se tím naruší řazení fronty. Oba způsoby použití mají uplatnění, detailní dokumentaci lze nalézt v souboru *action/queue.h*.

4.2.4 Proces

Řídící struktura procesu, tzv. Process Control Block (PCB), je odvozena od struktury *Action_t* a uchovává informace o stavu procesu, reference na jeho alokované zdroje, aktuální prioritu, frontu neobsloužených signálů (viz 4.2.6), frontu akcí čekajících na ukončení procesu, ukazatel na zásobník procesu, lokální úložiště (*process local*) apod. Je to tedy opět zdroj, jehož vlastníkem je proces, v jehož kontextu byl spuštěn, a je možné ho standartním způsobem uvolnit (*dispose()*), nicméně zde může nastat situace, že uvol-

nění procesu probíhá již v jiném kontextu (například v kontextu procesu samotného) a je tedy nutné čekat, až se uvolnění dokončí. Proto namísto *dispose()* je vhodné používat *process_kill()*. Proces může spustit libovolé množství dalších procesů, jediné omezení pro maximální počet je velikost paměťového prostoru pro uložení PCB a zásobníků jednotlivých procesů. Z hlediska správy zdrojů pak jednotlivé PCB tvoří stromovou strukturu a ukončení jednoho procesu nejprve ukončí všechny procesy, které byly spuštěny z jeho kontextu, takže dochází u uvolnění celého podstromu procesů a všech zdrojů alokovaných těmito procesy. Uvolňování zdrojů se vždy děje v opačném pořadí, než v jakém byly alokovány.

Řídící struktura obsahuje dále konfiguraci pro spuštění procesu, kde je vstupní bod (*entry point*) procesu, parametry, které se mu po startu předají, výchozí prioritu a adresu a velikost paměťového bloku vyhrazeného pro zásobník procesu. Během vytvoření procesu nejprve proběhne inicializace zásobníku. Na zásobník se uloží návratová adresa na obsluhu ukončení procesu a dále se inicializuje kontext, ze kterého se při prvním spuštění procesu naplní registry procesoru. Přepínání kontextu se děje po vyvolání softwarového přerušení a standartně probíhá tak, že obsah registrů procesoru (kontext procesu) se uloží na zásobník, obsah registru SP (*stack pointer*) se uloží do PCB běžícího procesu, dále se registr SP nastaví na vrchol zásobníku jiného procesu a obnoví se jeho kontext, takže po návratu z obsluhy přerušení se pokračuje ve vykonávání kódu tohoto procesu. Popsaný způsob na procesoru MSP430 obnoví také obsah registru SR (*status register*). Manipulací obsahu SR lze procesor uvést do úsporných režimů, implementace nečinného (idle) procesu, který bude procesor pouze přepínat do úsporných režimů, je tedy na MSP430 za použití tohoto OS zcela triviální. Samotné procesy o přepínání kontextu neví, a pokud na tyto události potřebují z nějakého důvodu reagovat, tak je možné pro ně zaregistrovat odpovídající *callbacky*[1]. Jedním z případů užití těchto *callbacků* je právě ten, když proces před naplánováním potřebuje resetovat paměťový prostor, ze kterého se obnoví SR, například když nečinný proces před přepnutím do úsporného režimu vždy musí něco udělat.

4.2.5 Plánovač

Plánovače pro systémy reálného času narozdíl od plánovačů systémů, kde se vyžaduje spíše optimální využití zdrojů než minimální doba odezvy na události, jako například Linux nebo Windows, mají tu výhodu, že nemusí obsahovat téměř žádnou rozhodovací logiku, protože se očekává, že za všech

[1] *pre_schedule_hook* a *post_suspend_hook*

okolností poběží proces s nejvyšší prioritou, který je ve stavu připraven (*READY*). Zařízení, která vyžadují striktní dodržení zpracování obsluhy událostí do definované doby (*deadline*), jako například řídící jednotka auta, musí být založené na systému, jehož plánovač to podporuje (*hard RTOS*). Návrhu těchto zařízení většinou předchází analýza plánovatelnosti a systém se v klíčových situacích musí chovat naprosto deterministicky. V době rozvoje IoT a automatizace je toto často diskutované téma a systémy založené na *RTOS* lze již najít například ve většině kávovarů, v kartáčku na zuby nebo v žárovce s podporou IPv6. Na *safety-critical* systémy, kde nedodržení *deadline* má zpravidla fatální následky, se kladou mnohé další požadavky, jako je statická alokace paměti, pravidelné spouštění self testů, kdy software nějakým způsobem testuje správnou funkčnost hardware, a mnohé další. Plánovač OS tohoto projektu byl navržen tak, aby bylo možné systém klasifikovat jako *hard RTOS*. Umožňuje plánování událostí metodami SCS (*Static Cyclic Scheduling*) RMS (*Rate Monotonic Scheduling*) a DMS (*Deadline Monotonic Scheduling*) a nastavení priority pro obsluhy externích událostí. Plánování metodou EFD (*Earliest Deadline First*) není k dispozici.

Výchozí priorita procesu se nastavuje během jeho vytvoření. Od momentu vytvoření procesu až po jeho ukončení jeho priorita nikdy neklesá pod tuto výchozí hodnotu. Na struktuře PCB jsou dvě fronty, jedna pro události čekající na zpracování (*signal*) a druhá pro akce čekající na ukončení procesu. Proces dědí prioritu těchto dvou front způsobem podobným tomu, který je popsán v 4.2.3, tedy priorita procesu nikdy není nižší, než priorita akce s nejvyšší prioritou v těchto dvou frontách. Prioritu lze dále dynamicky zvyšovat při každém blokujícím volání, kdy lze nastavit prioritu, jakou proces bude mít po opětovném probuzení (například při blokujícím čekání na událost). Tato priorita je pak platná od momentu daného volání až po další blokující volání, nebo do momentu, kdy proces zavolá *yield()*, priorita se pak resetuje na nejnižší přípustnou hodnotu.

Blokující volání funguje tak, že se struktura PCB odebere z fronty připravených procesů a přepne se kontext na jiný proces, blokovaný (*suspended*) proces pak čeká na to, až se z nějakého jiného kontextu naplánuje. Strukturu PCB lze chápat jako akci typu proces, a po vyvolání triggeru na PCB se proces opět vloží do fronty připravených procesů. Parametrem každého blokujícího volání je již zmíněná dynamická priorita a *timeout*. Podle návratové hodnoty volání lze zjistit, zda došlo k *timeoutu*, návratová hodnota vždy obsahuje druhý parametr volání *action_trigger(PCB, signal_t)*. Význam této hodnoty je v každém kontextu jiný, například při blokujícím čekání na ukončení procesu se takto předává návratová hodnota daného procesu apod. Se strukturou PCB tedy lze pracovat jako se standartní struk-

turou typu *Action_t*, je možné jí vložit do libovolné fronty akcí a vlastník této fronty opět může nějakým způsobem dědit prioritu této fronty, tedy i tohoto procesu. Například jí lze vložit do fronty akce typu událost (*event*), která dědí prioriu této fronty (a tím pádem i daného procesu). Obsluha události se pak provádí v kontextu nějakého procesu, který dědí prioritu této události, jak již bylo popsáno dříve. Tato obsluha provede *trigger_all()* na frontě akcí čekajících na tuto událost. Trigger na akci typu proces (PCB) způsobí to, že se odebere z této fronty a přidá se do fronty procesů připravených ke spuštění, což může být příčinou pro pokles priority procesu, v jehož kontextu se provádí obsluha události, takže se rovnou přepíná kontext na proces, který na událost čekal. Na podobném principu je založena akce typu semafor, nicméně dědičnost priorit lze využít i jinak. Akce typu mutex se po uzamčení přidává do fronty akcí čekajících na ukončení procesu, ten pak dědí jeho prioritu. Sám mutex dědí prioritu fronty procesů blokovaných na daném mutexu, čímž je vyřešen problém inverze priorit (*priority inversion*).

Procesy se ve frontě připravených procesů řadí sestupně podle priority způsobem, který popisuje kapitola 4.2.2, tedy proces je po vložení do této fronty vždy zařazen za procesy s vyšší nebo stejnou prioritou. V praxi to znamená, že pokud proces vyvolá trigger na událost se stejnou prioritou, pak se kontext hned nepřepíná na obsluhu dané události. Dále funkce *yield()* způsobí reset priority procesu na nejnižší přijatelnou hodnotu v daný moment (zneplatní se případná dynamická priorita nastavená při blokujícím volání) a pozice PCB se ve frontě připravených procesů posune za všechny procesy s vyšší nebo stejnou prioritou. Přepnutí kontextu se vyvolá vždy, když *head* fronty připravených procesů je jiný než právě běžící proces. Pomocí *yield()* lze tedy realizovat kooperativní multitasking.

Plánovač mimo jiné také umožňuje přepínat nastavení interního WDT (watchdog timer) v závislosti na tom, který proces právě běží[1]. Během přepnutí kontextu se pak WDT vynuluje a obnoví se stav jeho řídícího registru, který je uložen na PCB. Při vytvoření procesu se do PCB ukládá aktuální stav řídícího registru WDT, takže lze opět hovořit o dědičnosti nastavení WDT. Pokud je tedy proces spuštěn z nějakého kontextu, kde běží WDT, pak i po jeho spuštění běží WDT s těmi samými parametry. Pokud je pak nutné tyto parametry změnit nebo WDT vypnout, tak se to musí stát až v kontextu nově spuštěného procesu.

[1]tuto funkci je nutné zapnout v době sestavení programu definováním symbolu *___PROCESS_LOCAL_WDT___*

4.2.6 Signál

Akce typu signál (*Action_signal_t*) je prostředek pro oddělení požadavku na volání funkce (*invocation*) od samotného volání funkce (*execution*) do dvou nezávislých kontextů. Každý signál má referenci na proces, v jehož kontextu má dané volání proběhnout, umožňuje tedy propagování události (*trigger*) z libovolného kontextu do cílového kontextu, ve kterém proběhne její obsluha (*handler*). Druhý parametr volání *action_trigger(*, signal_t)* se uchovává a předává se do příslušného *handleru* opět jako druhý parametr.

Na signálu lze teoreticky vyvolávat trigger rychleji, než je v daný moment možné volat obsluhu, například pokud, je zrovna spuštěn proces s vysokou prioritou a za dobu jeho běhu se dvakrát vyvolá trigger na signálu, jehož obsluha se provádí v procesu s nízkou prioritou. Pro tento případ se akce typu signál chová (ve výchozím stavu) tak, že po každém triggeru inkrementuje vnitřní čítač, který po ukončení obsluhy opět dekrementuje a jeho obsluha se provádí do té doby, dokud hodnota tohoto čítače nedosáhne nuly. Na zmíněný parametr volání *action_trigger(*, signal_t)* zde není žádný buffer a do *handleru* se předává vždy poslední hodnota, se kterou se trigger vyvolal.

Pro obsluhu signálů musí být proces ve stavu čekání na signál (*waiting*). Čekání na signál je blokující volání, které má opět jako parametry dynamickou prioritu a timeout. Dynamická priorita udává minimální prioritu, kterou má proces mít po celou dobu, dokud čeká na signály a zpracovává jejich obsluhy. Tuto dynamickou prioritu lze zrušit v rámci obsluhy jakéhokoliv signálu voláním *yield()* nebo standartním způsobem po přepnutí zpět do normálního stavu. Funkční typ obsluhy signálu je definován následovně:

```
typedef bool (*signal_handler_t)(void *owner, signal_t signal);
```

Prvním parametrem obsluhy je vlastník, kterého lze libovolně na signálu nastavit, a návratová hodnota je typu *bool*. Opustit stav čekání na signál lze tak, že obsluha signálu vrátí *false*, případně vyvoláním triggeru přímo na daném PCB, což se například děje v případě timeoutu čekání. Proces ve stavu čekání na signál lze probudit vyvoláním triggeru na akci typu signál, která má kontext obsluhy nastaven na daný proces. V tomto stavu tedy pouze sekvenčně obsluhuje signály, a pokud je fronta nezpracovaných signálů prázdná, pak se proces odebere z fronty připravených procesů a plánovač ho dále neplánuje. Plánovač opět poskytuje možnost nulování WDT po skončení obsluhy každého signálu.

Během obsluhy signálu se může priorita tohoto signálu z nějakého důvodu libovolně měnit, případně v rámci samotné obsluhy lze daný signál vyjmout z aktuální fronty nezpracovaných signálů (a například ho vložit do jiné).

Tyto operace mohou mít přímý dopad na prioritu procesu, v jehož kontextu obsluha právě probíhá. Toto chování pro obsluhu akce typu událost (*event*) je velice výhodné, nicméně v jiných případech použití může situace vyžadovat, aby priorita procesu nikdy neklesla pod prioritu, kterou signál měl před tím, než se spustila jeho obsluha, po celou dobu vykonávání kódu dané obsluhy. Toto chování lze nastavit individuálně pro každou akci typu signál (a všechny z něj odvozené).

OS po startu spouští jeden proces, jehož kontext je vyhrazený pouze pro obsluhu signálů (*signal procesor*[1]). Tento proces má výchozí prioritu 0, v jeho kontextu se mimo jiné vyvolává trigger na časovaných signálech a je to výchozí kontext pro oblsluhu akcí typu událost. Uvážíme-li celou situaci z širšího kontextu, pak je patrné, že obsluha jednoho signálu nikdy není přerušena obsluhou jiného signálu se stejnou prioritou, a nezáleží na tom, zda se obsluhy provádějí ve stejném kontextu. Zároveň pro obsluhy ve stejném kontextu platí, že se jedna obsluha vždy dokončí před tím, než se spustí obsluha další, a to i v případě, že nemají stejnou prioritu. Pokud pak množina signálů, jejichž obsluhy se navzájem nemohou přerušit, pracuje s globálními proměnnými, ke kterým mají přístup pouze tyto obsluhy, pak lze z tohoto pohledu obsluhu považovat za atomickou operaci. Vhodným návrhem se tedy často lze vyhnout použití mutexu.

4.2.7 Událost

Akce typu událost (*event*) je odvozena od akce typu signál. Struktura akce typu událost obsahuje frontu akcí, jejíž prioritu událost dědí. Pro každou událost je možné nastavit minimální prioritu, pod kterou priorita události nikdy neklesne. Po vyvolání triggeru na události se v rámci její obsluhy v nastaveném kontextu sekvenčně volá trigger na všech akcích ve frontě dané události. Trigger na událost lze vyvolat z libovolného kontextu a lze jí chápat jako hranici mezi nezávislými kontexty a jako *proxy* vstupu na množinu akcí. Dále je k dispozici akce typu *proxy*, která v rámci svého triggeru vyvolává trigger na jednu jinou akci a má nastavitelný filtr (*interceptor*) na vstup. Akce typu *subscription* je pak akce typu *proxy* na akci typu signál.

Vzhledem k tomu, že všechny datové struktury OS nemohou být zároveň ve dvou a více frontách, vzniká problém, když vyvolání triggeru na některé akci vždy způsobí vložení této akce do nějaké fronty, logicky tedy i odebrání z fronty, ve které zrovna je. Typickým příkladem je akce typu signál, která se po triggeru vždy přidá do fronty neobsloužených signálů na příslušném kontextu. Problém pak spočívá v tom, když se nějaká obsluha má spustit po

[1]pro tento proces lze přednastavit výchozí interval WDT během sestavení programu

každém triggeru na nějaké události. V takovém případě musí být daný signál vždy ve frontě dané události, aby se po každém spuštění obsluhy události inkrementoval vnitřní čítač na daném signálu. Z toho důvodu vznikla akce *proxy*, která může být v libovolné frontě a šířit trigger na jinou akci, na které se tedy vyvolá trigger vždy, a nezávisí na tom, ve které frontě zrovna je.

Tyto datové struktury poskytují široké možnosti pro obsluhu událostí. Je možné vyvolávat trigger na signál z několika různých zdrojů a je možné šířit událost z jednoho zdroje do více signálů. Trigger z více zdrojů lze směrovat na jeden signál přes akce typu *proxy*, které mohou mít společný filtr (*interceptor*), lze tedy v daný moment obsluhovat jen vybrané události nebo čekat na více událostí najednou. Na tyto struktury lze pak pohlížet jako na uzly orientovaného grafu, jehož hrany odpovídají směru šíření události. Uzlem v tomto grafu pak může být akce libovolného typu a zdrojem událostí mohou být senzory zařízení, čas (periodický trigger nebo timeout) případně nějaký proces, a výstupem může být například manipulace řících registrů zařízení apod. Tento model pak lze označit jako neuronovou síť.

4.2.8 Časování

Modul časování, který systém poskytuje, má na rozdíl od ostatních modulů pouze omezené množství případu užití (*use-case*), ale jeho implementace je relativně složitá. Modul poskytuje službu vyvolání triggeru na akci typu signál po nějaké definované době, periodické vyvolávání triggeru s konstantní periodou, funkci pro konverzi časových jednotek a pro zjištění aktuálního času. Hlavní požadavky jsou přesné časování[1], minimální závislost na počtu a typu systémových zdrojů a minimální dopad na spotřebu systému. V neposlední řadě se požaduje dostatečná efektivita kódu a minimální režie a také minimalizace doby, po kterou je nutné vypínat přerušovací systém.

Časovač je standartní periferie, kterou lze nalézt na každém procesoru. Pro implementaci přenositelného systému je nutné uvážit, že vnitřní čítače mají rozsah 8 - 32 bitů a že frekvence jejich vstupního signálu nemusí být (řádově) stejná jako frekvence oscilátoru procesorového jádra, může být výrazně nížší, ale v krajních případech i výrazně vyšší. Dále je nutné stanovit jednotku času (*time unit*), se kterou bude systém pracovat. Tento OS je vhodný pro jednojádrové procesory, jejichž parametry nedovolují provoz OS Linux, tedy 8bitové, 16tibitové a některé 32bitové procesory s frekvencí do několika stovek MHz. Granularitu 1μs tedy zle považovat za dostatečnou a časová jednotka byla definována následovně:

[1] přesné v rámci možností použitého oscilátoru

```
struct Time_unit {
    // number of microseconds, never exceeds 3600 * 1000 * 1000
    uint32_t usecs;
    // number of hours
    uint16_t hrs;
};
```

Součástí modulu je převod času (hodin, vteřin, ms a μs) do této časové jednotky. Pokud na procesoru není k dispozici instrukce násobení (ani externí násobička) nebo systém vyžaduje vyhnout se použití této násobičky, pak je k dispozici násobení pomocí bitových posunů a sčítání optimalizované pro šesnáctibitové instrukce. Maximální hodnota, kterou struktura *Time_unit* může nabývat, je přibližně sedum let.

Modul časování pracuje s řídící strukturou kanálu časovače. Pokud systém v daný moment nevyžaduje časování, tedy pokud nejsou naplánované žádné časové události, pak se kanál vždy nechává vypnutý z důvodu snížení spotřeby energie. Konkrétně na MSP430 je to také z toho důvodu, že po vypnutí časovače pak zaniká *clock request* a neblokuje se tak přepínání do úsporných režimů. Řídící struktura kanálu je rozšířena následovně:

```
typedef struct Timing_handle {
    // resource, timer handle
    Timer_channel_handle_t timer_handle;
    ...
    uint32_t (*ticks_to_usecs)(uint32_t ticks);
    uint32_t (*usecs_to_ticks)(uint32_t us);
    // counter bit width, accepted range 8 - 32
    uint8_t timer_counter_bit_width;
} Timing_handle_t;
```

Převodní funkce (počet tiků na μs a opačně) a rozsah vnitřního čítače použitého časovače je nutné nastavit před inicializací modulu časování. Pro daný rozsah se pak určí inkrement, který představuje počet tiků časovače, po kterém se přepočítává přesná (stabilní) hodnota aktuálního času. S časovačem se v principu pracuje tak, že je vždy známá poslední hodnota *compare* registru (CCR), pro kterou je dopočten stabilní čas (stabilní CCR). Dále je možné nastavovat hodnotu CCR na libovolnou hodnotu podle toho, zda je do příštího inkrementu naplánovaná nějaká časová událost, nikdy však na vyšší (pozdější) hodnotu, než stabilní CCR + uvedený (maximální) inkrement. V moment, kdy hodnota vnitřního čítače dosáhne hodnoty stabilní CCR + maximální inkrement, se stabilní CCR nastaví na stabilní CCR + maximální inkrement a opět se pro stabilní CCR dopočte stabilní čas. Jinými slovy in-

krement udává nejdelší časový interval (periodu), během kterého nemusí nastat přerušení časovače pokud to není nutné, po uplynutí každé periody se ke stabilnímu času přičítá inkrement (přepočtený na μs) a aktuální čas se vždy počítá jako stabilní čas + počet tiků časovače od poslední periody (přepočtený na μs).

Hodnota inkrementu musí splňovat dva požadavky. V první řadě rozdíl maximální hodnoty vnitřního čítače a inkrementu musí představovat časový interval (podstatně) větší než doba, do které se za všech okolností obslouží přerušení časovače. Výchozí intervaly se přednastavují na 75 - 90% maximální hodnoty, konkrétně pro rozsah 16 bitů se interval nastaví na 0xE100. Prakticky pak na časovači se vstupní frekvencí 1MHz se přerušení musí obsloužit do 8ms, což pro procesorové jádro na frekvenci 8MHz představuje 64K strojových cyklů. Dále musí jít o univerzální hodnotu, kterou lze převést na 32 bitovou hodnotu μs v ideálním případě beze zbytku po dělení. Z těchto důvodů byla zvolena hodnota 0xE100 ($5^2 * 3^2 * 2^8$). Tento stabilní inkrement a inkrement převedený na počet μs se počítají pouze jednou při inicializaci modulu časování. Jde o optimalizaci, kdy se při čekání na vzdálenou událost při každém přerušení časovače k obsahu CCR pouze přičítá předvypočtený inkrement a ke stabilnímu času předvypočtený inkrement převedený na čas. Výhodou tohoto řešení je, že v libovolný moment se lze dotazovat na aktuální čas, který se dopočítá ze stabilní hodnoty času, ale stabilní hodnota se neupravuje. Kdyby tomu tak nebylo, tak by pro rozdíl aktuální hodnoty vnitřního čítače a poslední známé, pro kterou je známý čas, nemusel převod na μs vycházet beze zbytku po dělení, a se vzrůstajícím počtem dotazů na aktuální čas by se zvyšovala chyba posledního známého času. Další výhoda je ta, že není nutné obsluhovat indikaci přetečení (*overflow*) vnitřního čítače, což by znamenalo během jednoho přetečení CCR o jednu obsluhu přerušení navíc a pravděpodobně by to ani nevedlo k řešení problému.

Nastavení hodnoty *compare* registru (CCR) na požadovaný počet tiků v principu spočívá v přečtení vnitřní hodnoty čítače, výpočtu hodnoty, na kterou se CCR má nastavit, a samotném nastavení. Celý tento proces trvá nezanedbatelný počet tiků časovače, a z toho důvodu vzniká problém, že v moment samotného zápisu vypočtené hodnoty do CCR už je hodnota čítače vyšší než zapsané CCR, tedy vypočtená hodnota je v moment zápisu již minulost. Tento problém je jednoduše vyřešen tak, že se popsaný proces děje v rámci jedné funkce a během inicializace modulu časování se zjistí přesný počet tiků časovače, který uběhne za dobu volání této funkce. Tento počet tiků se pak použije jako minimální práh (*threshold*), kde hodnotu CCR nikdy nelze nastavit na bližší hodnotu, než je aktuální (přečtená) hodnota vnitřního čítače + *threshold*.

Během obsluhy přerušení časovače se vždy aktuální čas porovnává s časem nejbližší časované události, a pokud je větší, tak se vyvolá trigger na signálu s relativně vysokou prioritou, který obsluhuje frontu nadcházejícíci událostí. Během obsluhy přerušení časovače se tedy děje minimum operací. V rámci obsluhy tohoto signálu se vyvolá trigger na všech událostech, jejichž *trigger time* je menší nebo roven aktuálnímu času, a nastaví se hodnota CCR na čas následující události nebo na následující stabilní CCR, pokud nastane dříve. Události jsou tedy ve frontě nadcházejících událostí řazeny vzestupně podle jejich *trigger time*. Během požadavku na naplánování časovaného signálu (akce typu časovaný signál) se nejprve tento signál vloží do vstupní fronty seřazené podle priority a vyvolá se trigger na signál, který dědí prioritu této fronty a v rámci jehož handleru se provede zařazení signálu do fronty seřazené podle času. Výpočet aktuálního času a řazení fronty jsou operace poměrně náročné na procesorový čas, na MSP430 s frekvencí 8MHz modul časování vypíná přerušení minimálně na 150μs (v závislosti na počtu časovaných událostí), tedy zhruba 1200 strojových cyklů. Celkový overhead cyklu naplánování časovaného signálu a pozdějšího vyvolání triggeru je zhruba 600 μs. V daném projektu je dělička frekvence použitého časovače nastavena na /8, takže jeden tik časovače je roven jedné μs. Detailní informace týkající se časování lze dále najít v souboru *time.h*.

4.2.9 Spuštění OS

Ke startu OS a plánovače slouží funkce *kernel_start()*. Funkce má jako povinné parametry PCB *init* procesu a jeho prioritu, driver na vyvolání softwarového přerušení a ukazatel na inicializační funkci. Během startu se inicializuje plánovač, předá se mu daný driver a z aktuálního kontextu se vytvoří proces (*init*) s danou prioritou. Pokud je dále uveden nepovinný parametr s řídící strukturou kanálu časovače, pak se inicializuje modul časování. Tento modul je možné inicializovat až pozdějí po startu OS případně pak dynamicky měnit kanál časovače, na kterém modul závisí. Dále se volá inicializační funkce a vlastníkem všech zdrojů vytvořených v rámci této funkce je *init*. Nakonec se zavolá funkce *yield()* a povolí se přerušení. *Init* je tedy plánovatelný stejným způsobem jako jakýkoliv jiný proces, nicméně nesmí nastat situace, že žádný proces není plánovatelný. V kontextu *init* procesu by tedy neměly probíhat žádná blokující volání ani *sleep()* a je vhodné mu nastavit prioritu 0 a využít ho na přepínání do úsporných režimů. Z jeho kontextu je možné kdykoliv vyvolat ukončení OS (*kernel halt*) uvolněním PCB init procesu, čímž se zároveň uvolní celý strom alokovaných zdrojů a spuštěných procesů.

Posledním parametrem spuštění OS je *wakeup*, který má smysl pouze pro zařízení s perzistentní datovou pamětí, jako například FRAM. Pokud je nastaven, pak systém nejprve provede základní kontrolu platnosti struktury PCB *init* procesu. Pokud se *init* jeví jako plánovatelný, pak se pouze inicializuje plánovač (a případně modul časování), vyvolá se trigger na události probuzení systému, a nakonec se pouze vyvolá přepnutí kontextu. Systém pak implicitně předpokládá, že veškeré datové struktury, se kterými pracuje, jsou uložené v perzistentní datové paměti, a je tedy možné obnovit jeho stav bez nutnosti vše inicializovat znova. V rámci obsluhy odálosti probuzení systému lze obnovit stav ovladačů systému a ovladače vstupně-výstupních portů, takže je možné rovnou reagovat na externí událost, která vedla k probuzení systému.

4.2.10 Výkonnostní parametry

Mezi hlavní parametry, které charakterizují *RTOS* (operační systém reálného času), patří latence výsledného systému, tedy doba odezvy systému na externí událost. Latence se odvíjí od doby, kterou systém potřebuje pro přepnutí kontextu, a od nejdelší doby, na kterou systém vypíná přerušovací systém. Na procesuru MSP430 s frekvencí MCLK 8MHz přepnutí kontextu proběhne za 45 - 50μs, maximální doba, po kterou systém vypíná přerušovací systém, je pak 150 - 200μs v závislosti na maximálním počtu časovaných událostí. Latence od samotného přerušení až po spuštění obsluhy dané události je 150μs, v nejhorším případě (pokud je přerušovací systém v moment přerušení vypnutý) tedy 350μs.

Nároky na kódovou paměť systému záleží na datovém modelu (velikosti pointeru), dále na tom, které moduly jsou v době sestavení do OS přilinkované a také na překladačí. Velikost minimálního systému, tedy plánovače a obsluhy událostí, se pohybuje mezi 4 - 5KB. Dodatečný modul časování má velikost 2 - 3KB, mutex zhruba 500B a semafor taktéž 500B. Velikost PCB se odvíjí opět od datového modelu, dále záleží na tom, zda je systém sestaven s podporou správy zdrojů a zda je přilinkován modul časování. Maximální velikost je pak 290 bytů, pro datový model *small* bez použití správy zdrojů je to pak 146 bytů (velikost *Action_t* je 22 byte).

Porovnání parametrů s ostatními *RTOS* není předmětem tohoto textu, nicméně jen pro představu systém scmRTOS, který si zakládá na minimální době odezvy a minimálním nároku na paměťový prostor, pro MSP430 na 5MHz udává dobu přepnutí kontextu na 45-50 us a nároky na datovou paměť od 512B. Výsledek práce lze tedy považovat za poměrně zdařilý.

4.3 Komunikační rozhraní

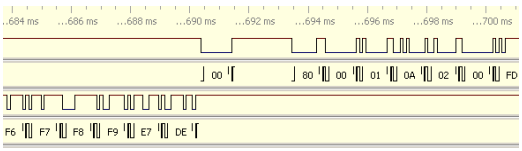
Operační systémy standartně poskytují podporu pro různá komunikační rozhraní a protokolový zásobník, zpravidla jde o vrstvenou architekturu podobnou modelu ISO/OSI a implementaci mnoha protokolů na každé vrstvě. Nejnižší (fyzická) vrstva poskytuje prostředky pro přístup na dané komunikační médium a samotnou komunikaci zpravidla zajišťuje hardware vyhrazený pro tento konkrétní účel, např. síťová karta, eUSCI, bluetooth aj. Abstraktní vrstvu nad tímto hardware (HAL - hardware abstraction layer) poskytuje ovladač daného zařízení. Ovladač používá nadřazená vrstva, která typicky zajišťuje přístup na dané médium, řeší konflikty přístupu na médium a poskytuje prostředky pro blokové přenosy dat. Každá další vrstva přináší novou úroveň abstrakce a definuje rozhraní, které poskytuje vyšší vrstvě, a závislost na rozhraní nižší (*underlying*) vrstvy. Protokoly na stejné vrstvě závisí na stejném rozhraní nižší vrstvy. Tato modulární architektura umožňuje oddělit aplikaci od konkrétní implementace protokolového zásobníku nebo použitého média a má mnohé další výhody.

Protokolové zásobníky pro vestavěná (*embedded*) zařízení musí odpovídat možnostem a omezením daného zařízení, kterými jsou nízká výpočetní kapacita, omezený paměťový prostor a omezené možnosti napájení. Mezi nejznámější patří *uIP* TCP/IP stack systému *Contiki*, který poskytuje protokolové zásobníky pro IPv4 a uIPv6 za použití pouze 30KB paměti pro kód a 10KB pro data. *Contiki* má vlastní protokolový zásobník *Rime* jako sadu protokolů pro případy, kde overhead IPv4 je nepřijatelný a pro které jsou standartní internetové protokoly nevhodné, například pro senzorické sítě.

Mnohé zařízení mají vlastní protokolové zásobníky pro specifické účely a jejich konkrétní způsob implementace plyne z požadavků plynoucích z případů užití. Software tohoto projektu poskytuje dvě rozhraní a obě mají velice specifické požadavky. Rozhraní pro komunikaci se satelitem je postavené na SPI v režimu *slave*, kde přenosy zahajuje řídící jednotka satelitu. Přes toto rozhraní dochází k vyčtení logu událostí a statistik a přenosu bloku pro upgrade firmware procesoru. Rozhraní využívá pouze systém, není k dispozici pro testy. Dále je zde rozhraní pro komunikaci mezi procesory pro všeobecné využití a pro účely obnovy paměti při výpadku přes BSL. Toto rozhraní je postavené na UART, pro komunikaci na linkové vrstvě se využívá rozšířený BSL protokol, a poskytuje spolehlivou a potvrzovanou službu pro blokující a asynchronní datové přenosy mezi procesory. Tato kapitola dále popisuje implementaci obou rozhraní a jejich jednotlivé vrstvy.

4.3.1 Linková vrstva rozhraní UART

Linková vrstva komunikačního rozhraní poskytuje služby pro blokové přenosy paměťových bloků po rozhraní UART. Přenosy probíhají formou packetů, každý packet začíná charakteristickým znakem (*header*), následuje délka (2 byte), adresní byte, obsah (*payload*) a na konci každého packetu je vždy CRC otisk jeho obsahu. Platnost obsahu lze tedy po přenesení ověřit kontrolou CRC. Po každém přenosu druhá strana tedy zkontroluje CRC a odpovídá jedním byte s potvrzením (*ACK*) nebo s chybovým kódem. Po takovém přenosu je na straně odesílatele vždy známé, zda byl packet druhou stranou přijat a zda byl přijat v takovém stavu, v jakém byl odeslán, přenos tedy lze označit jako spolehlivý. Během přenosu může druhá strana kdykoliv poslat byte s chybovým kódem, pokud je nějaký důvod packet odmítnout, nebo pokud například na začátku přenosu nebyl *header*, příadně pokud byl *header* během přenosu ztracen apod. Protokol jinak definuje minimální dobu, za kterou je možné začít vysílat po přijetí znaku (*TX mindelay*), na $3ms$, což na baud rate 9600 odpovídá přibližně době přenesení dvou znaků. Je tedy patrné, že přenosy probíhají v poloduplexním režimu. Ze skutečnosti, že po přijetí byte je nutné čekat *TX mindelay* před zahájením odesílání a že je po příjeti packetu nutné odeslat *ACK*, plyne, že pokud mají obě strany neprázdnou frontu packetů k odeslání, pak po odeslání packetu jedním směrem vždy následuje odeslání packetu druhým směrem. Průběh v takové situaci je zachycen na obr. 4.1. Přijetí dalšího znaku se tedy očekává nejdéle do *TX mindelay* a pak je možné začít vysílat, stav přijímacího automatu se resetuje. Protokol dále definuje *ACK* timeout (*ACK maxdelay*) na 3/2 *TX*



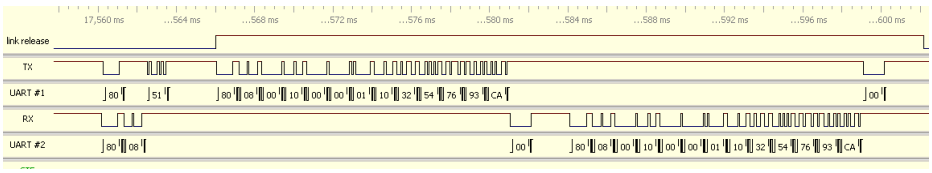
Obrázek 4.1: Průběh na lince po přijetí ACK

mindelay, tedy dobu, do které je nutné odeslat *ACK*. Pokud do této doby *ACK* nedorazí, tak je packet nepotvrzený a je opět možné začít vysílat.

Linková vrstva sama o sobě omezuje délku packetu na 64KB, nicméně přenos dlouhého bloku v rámci jednoho přenosu není optimální. Pokud během přenosu dlouhého packetu dojde k chybě v jednom bytu, pak je nutné celý packet odeslat znova, a vzhledem k poloduplexní povaze linkové vrstvy není optimální, aby se přenosové médium na delší dobu zablokovalo komunikací v jednom směru. Uvedený protokol a forma packetu je kompatibilní s protokolem BSL (viz 2.5) s tím rozdílem, že BSL definuje maximální délku

obsahu packetu na 260 bytů. Tento problém je vyřešen na transportní vrstvě, která zajišťuje fragmentaci packetů, viz 4.3.2. Parametry pro přenesení jednoho znaku jsou převzaté z protokolu BSL, přenos vždy začíná start bitem, následuje osm datových bitů, jeden stop bit a sudá parita. Přenosový signál má 9600 baud rate, což opět odpovídá stavu po přepnutí do BSL režimu resetovací sekvencí. BSL protokol sice umožňuje zvýšení baud rate, nicméně bylo rozhodnuto, že na hlavním komunikačním kanálu se baud rate měnit nebude, aby se zamezilo stavu, že UART na obou procesorech bude nastaven na jinou baud rate.

Linková vrstva dále zajišťuje přístup na komunikační médium (MAC - media access control). Je povoleno začít vysílat kdykoliv, když zrovna neprobíhá příjem. Je tedy možné, že se obě strany rozhodnou vysílat ve stejný okamžik. Po detekování takového stavu, tedy po přijetí *headeru* v moment, kdy probíhá odesílání packetu, se na každém procesoru vypočítá doba, po kterou se odmlčí před tím, než zkusí vysílání opakovat (*backoff delay*). Tato doba (počet ms) se vypočítá z *randseed* (rozmezí 2048 - 4095) a pokud má procesor nastaven pin MCU_ID, pak se k ní přičte 4096. Typický průběh takové kolize je zachycen na obr. 4.2.



Obrázek 4.2: Průběh kolize na lince

Rozhraní linkové vrstvy poskytuje funkci pro odeslání adresního bytu a až dvou paměťových bloků v rámci jednoho packetu. Součet délek obou bloků nesmí přesáhnout 64KB a vždy se odesílají v definovaném pořadí. Význam adresního byte je záležitost nadřazeného protokolu, pro BSL jde o kód příkazu *command*, transportní vrstva ho využívá k adresování *portu*. Požadavek na odeslání je přijat pouze pokud se předchozí odeslání dokončilo, o dokončení a jeho stavu se nadřazená vrstva vždy dozví z příslušného *callbacku*. Na některé chyby přenosu, jako je timeout potvrzení a chyba kontrolního součtu (CRC), linková vrstva reaguje tak, že přenos zkusí ještě dvakrát zopakovat před tím, než vyvolá *callback* s chybovým kódem. *Callback* se vždy volá z kontextu obsluhy přerušení, celková režie (*overhead*) odeslání jednoho byte je zhruba 240 strojových cyklů, tedy 30μs na frekvenci 8MHz.

Přijem packetu probíhá tak, že se po přijetí délky a adresního byte volá příslušný *callback*, v rámci kterého packet může nadřazená vrstva odmítnout s chybovým kódem, který se pak odesílá a tím se přenos daného packetu

ukončuje. Na straně odesílatele pak končí odeslání tohoto packetu s daným chybovým kódem. Pokud je packet přijat a neobsahuje pouze adresní byte, pak musí nadřazená vrstva poskytnout buffer a jeho délku, kam se bude zapisovat obsah packetu. V moment, kdy je buffer plný, se opět volá nadřazená vrstva, která opět packet může odmítnout, nebo musí vrátit další buffer. Po prvním *callbacku* vždy nakonec následuje *callback* s informací o stavu přijetí packetu, může jít o chybu CRC případně o timeout. V principu je tedy po tom, kdy je nadžazená vrstva informována o zahájení příjmu packetu, vždy informována o jeho výsledku. *Callbacky* se volají z kontextu obsluhy přerušení, *overhead* přijetí jednoho byte je zhruba 320 MC (40μs na 8MHz).

4.3.2 Transportní vrstva rozhraní UART

Transportní vrstva pracuje s generickým typem *Serial_link_t*, tedy s vrstvou, která poskytuje rozhraní a *callbacky* popsané v předchozí kapitole. Detailní popis rozhraní této vrstvy lze nalézt v souboru *serial/link.h*. Vrstva využívá také služeb operačního systému jako je blokující volání, zpracování asynchronní události a časování. Poskytuje služby pro registrování *portu* s definovaným číslem, několik režimů přijímaní dat na daném portu a blokující a asynchronní obsluhu přijatého packetu nebo timeoutu. Dále poskytuje blokující a neblokující odesílání packetu s nastavitelným timeoutem a adresování packetu na číslo portu. Součástí transportní vrstvy je fragmentace packetů na nastavitelnou délku, možnost asynchronního nastavení *headeru* fragmentu a asynchronní obsluhu přijetí *headeru*.

Datové struktury port a packet jsou odvozené od akce typu signál, lze tedy hovořit o akci typu port a akci typu packet. Port se stejným číslem lze na rozhraní transportní vrstvy registrovat pouze jednou a na port lze přijmout datový buffer pouze v případě, že je na daném rozhraní registrován. Na portu je nutné po vytvoření nastavit přijímací buffer, jeho délku a režim, který nastavuje chování portu po přijetí packetu a po skončení obsluhy přijetí packetu. Konkrétně velikost příchozího packetu (fragmentu) může být menší, než je velikost přijímacího bufferu, pak lze nastavit, zda vyvolat obsluhu daného portu po přijetí každého fragmentu, nebo až po přijetí datového bloku o stejné velikosti, jakou má přijímací buffer. Dále lze nastavit, zda po provedení obsluhy opět registrovat port na daném rozhraní s původními parametry, nebo povolit přijetí pouze jednoho datového bloku.

Odeslání packetu na rozhraní dané vrstvy způsobí buď přidání packetu do fronty packetů čekajících na odeslání, nebo se jeho první fragment rovnou předá linkové vrstvě, pokud je fronta prázdná. Fronta packetů je standartní fronta akcí seřazená podle priority a packety s vyšší prioritou mají tedy vždy

přednost. Pokud mají všechny packety stejnou prioritu, pak se fronta chová jako standartní FIFO (first in, first out). *Callbacky* linkové vrstvy ukončení přijímání nebo odesílání se vždy volají z kontextu obsluhy přerušení a vyvolávají trigger na příslušné akci typu port nebo akci typu packet, do kterého jako druhý parametr předávají výsledný status. Zde je tedy hranice mezi kontextem přerušení a kontextem obslužného procesu. Ze skutečnosti, že po přijetí packetu se na portu vyvolá trigger, mimo jiné plyne, že není možné přijmout dva packety na stejný port v době, kdy se ještě nedokončila obsluha daného portu po přijetí prvního packetu, tedy dokud je port ve frontě nezpracovaných signálů příslušného procesu. Struktura akce typu port je z hlediska infrastruktury OS velice podobná akci typu packet, obslužnou funkci lze nastavit přímo na dané akci a výchozí obslužná funkce pouze směruje trigger na jinou akci, kterou může být například akce typu proces v případě blokujícího volání. Obě tyto struktury obsahují vlastní akci typu časovaný signál z toho důvodu, aby bylo možné, realizovat asynchronní volání s timeoutem, nicméně samy nemohou být akcemi typu časovaný signál, protože akce nemohou být ve dvou frontách najednou.

Obsahem packetu je vždy adresní byte, kterým se adresuje port na transportní vrstvě na přijímací straně, packet dále může obsahovat *header* o maximální délce odpovídající maximální délce fragmentu daného rozhraní - 1 a datový buffer o maximální délce 64KB. Pokud je *header* nastaven, pak se pro každý fragment posílá ten samý blok paměti, kde *header* leží. Na packetu je dále možné nastavit *callback*, který se zavolá před odesláním každého fragmentu a během kterého lze obsah *headeru* upravovat. Do tohoto callbacku se jako parametry posílají ukazatel na daný packet, ukazatel na datový blok *headeru* a číslo fragmentu. Na přijímací straně je situace taková, že délka *headeru* se individuálně nastavuje pro každý port během jeho vytvoření. Pokud má port nastavenou nenulovou délku *headeru*, pak se tento počet bytů ukládá do statického bufferu dané transportní vrstvy. Jde vždy o byty následující adresní byte. Jedinou možností, jak k tomuto headeru přistupovat, je pak v rámci *callbacku on_header*, který lze registrovat na daném portu. Tento callback se volá v moment přijetí *headeru*, tedy před tím, než se začne přijímat zbytek packetu a CRC. V rámci tohoto *callbacku* je možné packet odmítnout, případně manipulovat adresu a velikost přijímacího bufferu apod. Je tedy patrné, že nastavení *headeru* na odchozím packetu ještě neznamená, že přijímací port tuto část obsahu packetu bude automaticky považovat za *header*.

Dále je nutno podotknout, že je možné odeslání prázdného packetu, tedy packetu, který neobsahuje ani *header* ani datový buffer. Tento packet může vyvolat trigger na akci typu port na přijímací straně, tím pádem zde lze

vyvolat trigger na libovolné akci. Pokud chápeme akce jako uzly orientovaného grafu (kap. 4.2.7), pak lze takto jednoduše sestrojit hranu mezi dvěma procesory. Událost se pak přes graf, který popisuje topologii šíření události na druhém procesoru, může teoreticky dostat na libovolný další procesor. Chápeme-li číslo portu jako identifikátor události, je tímto způsobem možné vyvolat na přijímací strané konkrétní událost. Na takovou událost lze reagovat stejným způsobem jako na přerušení, tedy jako na lokální událost.

Akci typu port a akci typu packet lze kdykoliv odebrat z fronty ve které se zrovna nachází. Pokud je packet ve frontě neodeslaných packetů a právě se odesílá, nebo pokud je port ve frontě registrovaných portů a zrovna přijímá packet, pak dochází k resetu linkové vrstvy. V případě resetu rozhraní se atomicky vyvolá trigger na všech packetech ve frontě (s parametrem RESET) a resetuje se linková vrstva. Funkci rozhraní lze pozastavit, čímž se linková vrstva zastaví (*halt*), ale stále je možné zařazovat packety do fronty neodeslaných.

Co se týče použití s BSL, pak je situace velice jednoduchá. BSL je jen jedno specifické použití rohoto rozhraní, kdy adresní byte se označuje jako číslo příkazu (*command*), *header* jako adresa, kterou když packet obsahuje, tak má vždy délku 3, a velikost fragmentu je vždy maximálně 260 bytů. Je zde pouze rozdíl ten, že příchozí packety se nijak nepotvrzují, což je možné nastavit na linkové vrstvě. Toto rozhraní je tedy velice dobře použitelné pro implementaci vlastního *bootloaderu* s rozšířenou funkcionalitou například o různé úrovně zabezpečení a šifrování přenosu.

5 Závěr

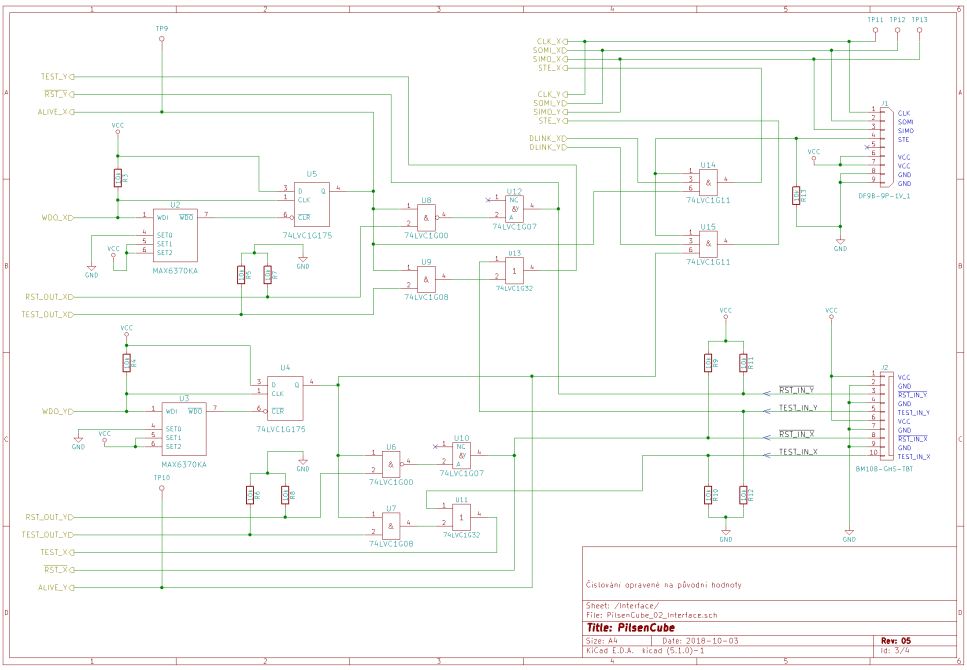
Výstupem této práce je zařízení, které umožňuje vzdálené spouštění libovolných aplikací na mikrokontroléru MSP430 s operační pamětí typu FRAM. Zařízení pro tyto aplikace poskytuje velice stabilní infrastrukturu co se týče hardware i softwarového vybavení a neklade na ně žádná omezení. Softwarové vybavení v základu poskytuje služby operačního systému, jako například *RTOS* plánovač, časování, asynchronní obsluhu událostí nebo logování. Logy lze ze zařízení zpětně vyčítat a mít tedy přehled o výsledném chování daných aplikací. Je zde mimo jiné k dispozici sada ovladačů pro řadu periferií na čipu mikrokontroléru.

Navržený hardware obsahuje dva procesory MSP430FR5994, na jednom se vždy sekvenčně spouští nahrané aplikace a druhý funguje jako podpora pro případ, že se první procesor dostane do nezotavitelného stavu. Aplikaci lze spustit na obou zařízení zároveň, pokud si to sama vyžádá a z nějakého důvodu ke svému běhu potřebuje podporu druhého procesoru. Pro komunikaci s druhým procesorem lze pak využít rozhraní, které poskytuje služby pro spolehlivý přenos datových bloků. Role procesorů se po určité době prohodí a celý proces se opakuje. Oba procesory mohou mít ve své paměti jiné aplikace a nahrání nové lze cílit na konkrétní procesor. V případě potřeby je možné vzdáleně přepsat celý firmware obou procesorů.

Na obou procesorech v době nečinnosti běží kontrola integrity paměti a procesor sám má k dispozici kopii kódové paměti, ze které je schopen případné chyby opravit. Pokud narazí na chybu, ze které se není schopen sám zotavit, tak obsah jeho paměti do určité doby obnoví druhý procesor obsahem paměti vlastní. Systém je odolný vůči náhodným výpadkům napájení.

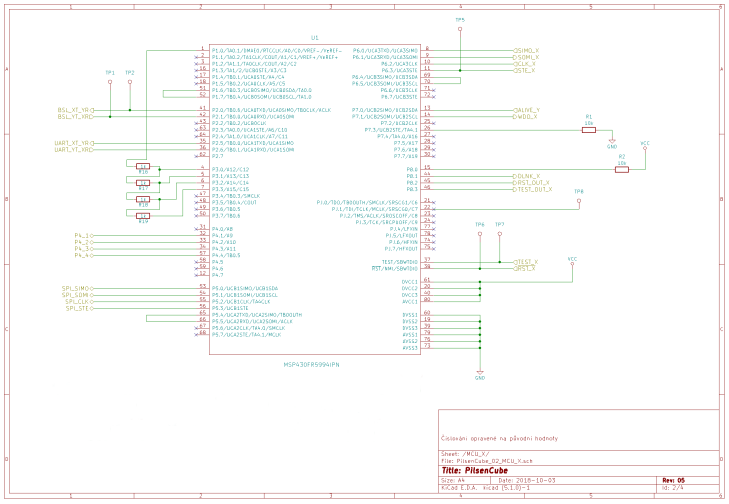
Zařízení bylo navrženo tak, že bude možné ho připojit do pikosatelitu PilsenCube II. Experiment v první řadě otestuje, zda daný hardware bude vůbec schopen pracovat v prostředí se zvýšeným vlivem radiace na oběžné dráze. Pokud ano, pak bude k dispozici systém, na kterém bude možné testovat vlastnosti mikrokontroléru MSP430FR5994 a všech jeho periferních zařízení v prostředí na palubě pikosatelitu. V neposlední řadě budou k dispozici statistiky chybovosti paměti typu FRAM v tomto prostředí.

A Schéma zapojení

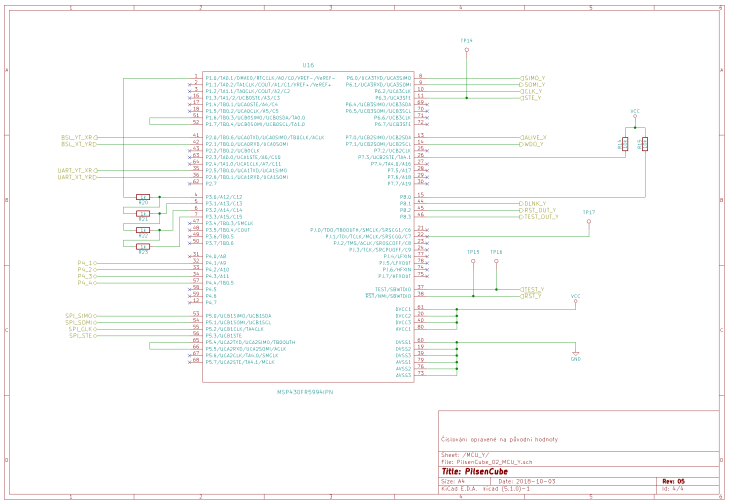


Obrázek A.1: Schéma zapojení logiky na plošném spoji

B Konfigurace portů



Obrázek B.1: Vstupně-výstupní signály připojené na procesor X



Obrázek B.2: Vstupně-výstupní signály připojené na procesor Y

Literatura

[1] Texas Instruments. MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers. https://www.ti.com/lit/ds/symlink/msp430fr5994.pdf, 2018.

[2] Texas Instruments. MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide. https://www.ti.com/lit/ug/slau367o/slau367o.pdf, 2017.

[3] Fujitsu Semiconductor. Memory FRAM MB85R4001A. https://www.fujitsu.com/us/Images/MB85R4001A-DS501-00005-3v0-E.pdf.

[4] Cypress. 4-Mbit (256K × 16) F-RAM. https://www.cypress.com/file/136476/download.

[5] Lee, Dong-Jae; Seok, Yong-Sik; Choi, Do-Chan; Lee, Jae-Hyeong; Kim, Young-Rae; Kim, Hyeun-Su; Jun, Dong-Soo; Kwon, Oh-Hyun. A 35 ns 64 Mb DRAM using on-chip boosted power supply. http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel2/662/5918/00229238.pdf?tp=&arnumber=229238&isnumber=5918, 1992.

[6] Texas Instruments. MSP430™ FRAM Devices Bootloader (BSL). https://www.ti.com/lit/ug/slau550t/slau550t.pdf, 2019.

[7] GCC - Open Source Compiler for MSP Microcontrollers. https://www.ti.com/tool/MSP430-GCC-OPENSOURCE. 6. května 2019.

[8] GCC Extensions in TI Compilers. http://processors.wiki.ti.com/index.php/GCC_Extensions_in_TI_Compilers. 6. května 2019.

[9] CMake + MSP430. https://drvlas-embedder.blogspot.com/2017/12/cmake-msp430.html. 6. května 2019.

[10] Using CLion for AVR (ATmega, Arduino) development. https://medium.com/@scireum_aha/using-clion-for-avr-atmega-arduino-development-f8d3e351f3a0. 6. května 2019.

[11] Texas Instruments. MSP-EXP430F5438 Experimenter Board. https://www.ti.com/lit/ug/slau263i/slau263i.pdf, 2013.

[12] Texas Instruments. MSP430FR5994 LaunchPad Development Kit. https://www.ti.com/tool/MSP-EXP430FR5994, 2016.