VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

OPERAČNÍ SYSTÉM REÁLNÉHO ČASU S FIXNÍ PRIORITOU ÚLOH PRO RASPBERRY PI

REAL-TIME OPERATING SYSTEM WITH FIXED TASK PRIORITY FOR RASPBERRY PI

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JOSEF KOLÁŘ**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Kolář Josef, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Počítačové sítě
Název: **Operační systém reálného času s fixní prioritou úloh pro Raspberry Pi**
Real-Time Operating System with Fixed Task Priority for Raspberry Pi
Kategorie: Operační systémy
Zadání:

1. Prostudujte problematiku operačních systémů reálného času (RTOS). Prostudujte volně šiřitelné RTOS pro vestavěné systémy a vyberte nejvhodnější k implementaci na platformě Raspberry Pi.
2. Vhodně vybraný RTOS použijte pro implementaci minimálně dvou oddělených procesů běžících na rozdílných frekvencích (např. 50 Hz a 10 Hz). Pro ověření vlastností systému v každém procesu naprogramujte čítač inkrementující s periodou procesu, hodnoty obou čítačů vypisujte každé 3 sekundy na sériový port. Vyhodnoťte vlastnosti této demonstrační aplikace a zdokumentujte postup tvorby aplikací s využitím vybraného RTOS.
3. Navrhněte a implementujte bezpečné sdílení společných dat mezi procesy. Navrhněte a implementujte kód pro připojení modulu Canberry k procesu s nižším kmitočtem (viz předchozí bod). Ověřte komunikaci po CAN sběrnici zobrazováním hodnot čítačů jednotlivých procesů, které budete periodicky vysílat prostřednictvím modulu Canberry. Postup tvorby této demonstrační aplikace zdokumentujte a vyhodnoťte její vlastnosti.
4. Na základě znalostí získaných z bodů 2 a 3 diskutujte možnosti využití vámi vybraného RTOS na platformě Raspberry Pi s modulem Canberry pro tvorbu distribuovaných řídících aplikací.

Literatura:

- AMOS, Brian. Hands-On RTOS with Microcontrollers. Packt Publishing Limited, 2020. ISBN 978-1-83882-673-4
- FAN, Xiaocong. Real-Time Embedded Systems. Elsevier Books, 2015. ISBN 978-0-12-801507-0

Při obhajobě semestrální části projektu je požadováno:

- První 2 body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Janoušek Vladimír, doc. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 3. listopadu 2021

Abstrakt

Abstract

Klíčová slova

operační systémy reálného času, plánovač, preemce, Raspberry Pi, Arm, Canberry, sběrnice CAN

Keywords

Citace

KOLÁŘ, Josef. *Operační systém reálného času s fixní prioritou úloh pro Raspberry Pi.* Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vladimír Janoušek, Ph.D.

Operační systém reálného času s fixní prioritou úloh pro Raspberry Pi

Prohlášení

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

.....
Josef Kolář
12. ledna 2022

Poděkování

Rád bych tímto vyjádřil mé poděkování doc. Ing. Vladimíru Janouškovi, Ph.D. za odborné vedení, rady a konzultace při realizaci této práce.

Obsah

1 Úvod **2**

2 Operační systémy reálného času **3**
2.1 Specifika RTOS 3
2.2 FreeRTOS 6
2.3 ChibiOS 6
2.4 RTEMS 6
2.5 RT-Thread 7
2.6 uCOS-II 7
2.7 Raspbian s PREEMPT_RT 7

3 Raspberry Pi 3B+ **8**
3.1 Dostupný hardware 8
3.2 Zavádění systému 8
3.3 Architektura ARMv8-A 10

4 Demonstrační aplikace A **11**
4.1 Prostředí pro programy v jazyce C 11
4.2 Podpora pro FreeRTOS 13
4.3 Standardní knihovna jazyka C 14
4.4 Tvorba samotné aplikace 15
4.5 Vyhodnocení funkce 17

5 Demonstrační aplikace B **18**
5.1 Bezpečné sdílení dat mezi úlohami 18
5.2 Ovladač pro modul Canberry 18
5.3 Tvorba demonstrační aplikace 18

6 Použití pro distribuované aplikace **19**

7 Závěr **20**

Literatura **21**

Kapitola 1

Úvod

> Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with your using PREEMPT_RT .
>
> *Linus Torvalds, 2004*

Kapitola 2

Operační systémy reálného času

Operační systém reálného času (anglicky real-time operating system, RTOS) je v informatice typ operačního systému, který poskytuje možnost reagovat na události v okolí počítače průběžně (tj. v reálném čase).

RTOS je používán například ve vestavěných systémech, robotice, automatizaci, elektronických měřeních nebo v telekomunikacích. Problém reálného času je specifický svou nutností splnění v určitém čase, tedy že problém musí být dokončen před určitým časovým limitem. Jeho nedokončení ve stanoveném čase je poté považováno za chybu – její charakter je určen třídou důležitosti časování, více v kapitole 2.1.1. [17]

2.1 Specifika RTOS

Jedním z klíčových požadavků na operační systém reálného času je schopnost preempce. Preempce je schopnost systému přerušit právě běžící úlohu bez její spolupráce a nahradit jinou úlohou pomocí přepnutí kontextu.

2.1.1 Třídy RTOS dle důležitosti plnění časového limitu

Podle toho, jak je splnění časového limitu kritické pro systém, dělíme operační systémy reálného času na třídy [12]:

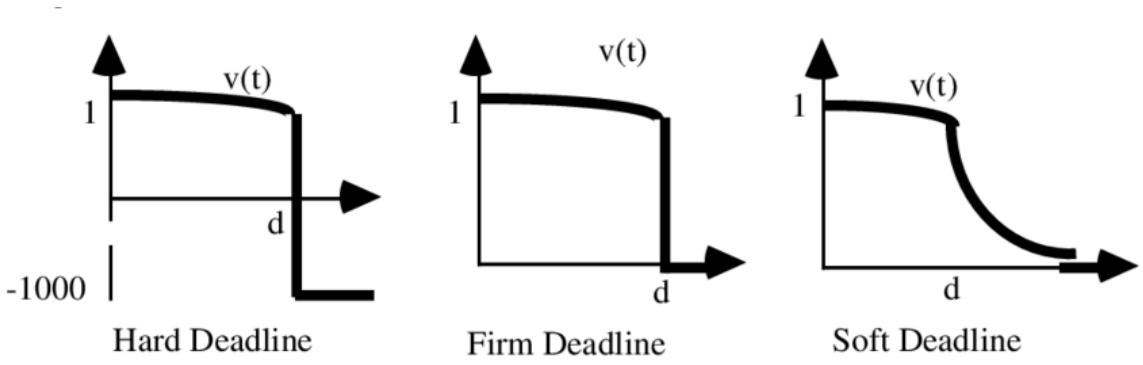
Tvrdá (*hard*) je nejpřísnější třída, při které je nesplnění časového limitu úlohy považováno za kompletní selhání systému – může vést k destruktivním následkům. Typické použití této třídy je systémech kritických na bezpečnost – např. řídící systémy vozidel.

Stálá (*firm*) je třída, při které nesplnění časového limitu nevede k totálnímu selhání systému, ale ke kompletnímu znehodnocení výsledku. Použití například při zpracování živého videa, zpožděné zpracování jednoho snímku nevede k selhání systému, ale pouze k nepoužitelnosti konkrétního snímku.

Měkká (*soft*) je třída, jejíž cílem je dodržení co nejvyšší úrovně QoS[1] – nesplnění časového limitu nevede ihned na znehodnocení výsledku, ale pouze na snížení úrovně kvality

[1] QoS – Quality of Service je

služeb. Použití této třídy je typické např. v systémech vykreslování, nesplnění limitu při jednom snímku pouze sníží FPS (a tedy QoS), nedojde k selhání systému.



Obrázek 2.1: Třídy RTOS dle použitelnosti výsledku

2.1.2 Plánovač

Plánovač je v RTOS zodpovědný za deterministické plánování běhů úloh – to je obvykle prakticky zajištěno nastavením priorit pro jednotlivé úlohy. Plánovač poté garantuje spouštění úloh podle jejich priority. Samotné plánovače využívají preemptivnosti úloh a může implementovat jeden z několika možných algoritmů plánování, níže výběr v praxi užívaných [19], jejichž časové diagramy jsou vyobrazeny v diagramu 2.2.

Prioritní plánování je plánování, při kterém se přímo využívá preempce a úlohy jsou vybírány podle jejich priority. Úloha tedy drží procesorový čas, dokud:

- Sama neukončí svůj běh.
- Úloha s vyšší prioritou nepřejde do stavu připravenosti na běh.
- Úloha se sama nevzdá procesorového času při čekání na sdílení zdroj či I/O operaci.

Situaci, kdy dvě úlohy sdílí stejnou prioritu i připravenost pro běh, řeší algoritmus typicky buď vnitřním Round-robin plánováním, nebo FCFS[2] plánováním.

Round-robin plánování je plánování s konstatním časovým kvantem, který je pravidelně úlohám přidělován. Při různých prioritách úloh je procesorový čas přidělován skupinám úloh se stejnou prioritou. Úloha tedy drží procesorový čas, dokud:

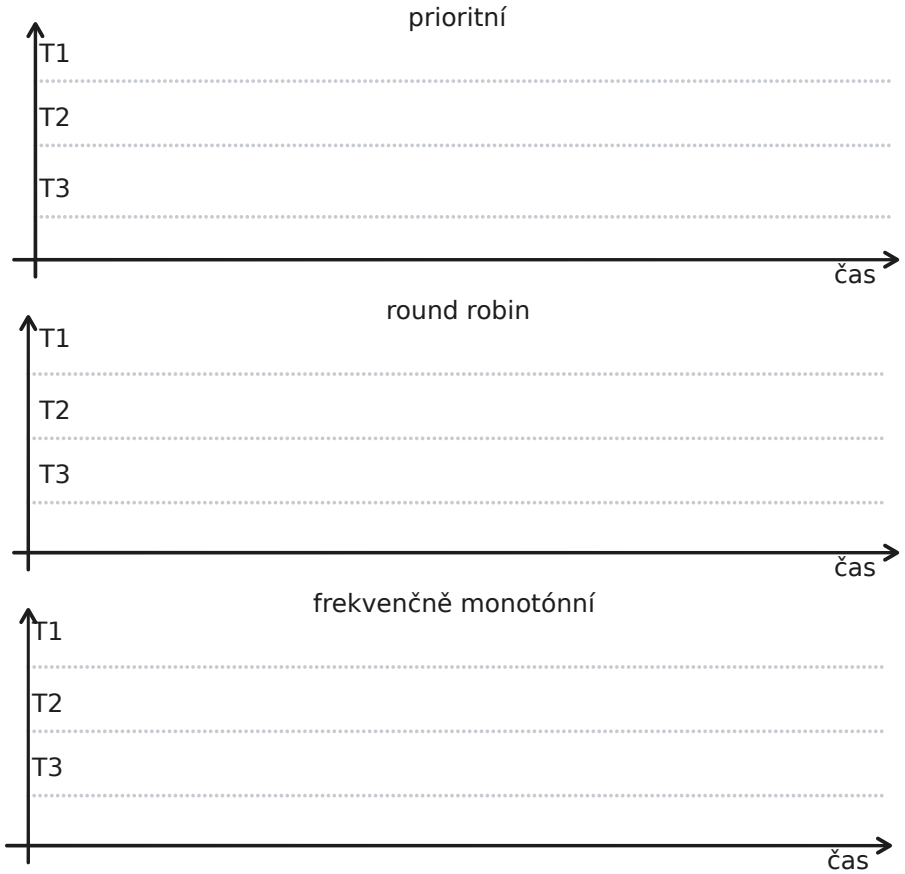
- Sama neukončí svůj běh.
- Úloha s vyšší prioritou nepřejde do stavu připravenosti na běh.
- Úloha se sama nevzdá procesorového času při čekání na sdílení zdroj či I/O operaci.
- Úloze nedojde přidělené časové kvantum.

[2]FCFS (*First Come, First Serve*) je ne-preemptivní plánování, které nebere v potaz priority a plánuje úlohy podle pořadí požadavků na procesorový čas – úloha, která zažádá jako první, dostane procesorový čas jako první.

Frekvenčně monotónní plánování (*Rate Monotonic Scheduling*) je plánování užívané u systémů s periodickými úlohami. Priority jsou úlohám přiřazeny podle jejich požadované frekvence spouštění – úlohy s vyšší frekvencí mají vyšší prioritu. Pro ideální prostředí lze testem ukázat [10], zda je u množiny n úloh Γ možné plánování na procesor, jestliže platí následující (C_i značí dobu provádění úlohy i, T_i časovou periodu spouštění úlohy i, a U využití procesoru) nerovnost:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \qquad U \leq n(2^{1/n} - 1)$$

Liu and Layland zároveň ukázali [10], že nelze-li plánovat množinu úloh algoritmem Frekvenčně monotóního plánování, nelze ji naplánovat ani žádným jiným algoritmem s fixní prioritou úloh.



Obrázek 2.2: Časové diagramy pro vybrané plánovací algoritmy: prioritní plánování, Round Robin plánování a frekvenčně monoťoní plánování.

2.1.3 Priorita úloh

2.2 FreeRTOS

FreeRTOS [5, 1] je multiplatformní RTOS pro vestavěná zařízení s oficiální podporou pro desítky různých architektur. Má otevřený zdrojový kód pod licencí MIT, širokou komunitní podporu v oficiální fóru, s více než 18 lety vývoje je jeho vlastníkem společnost Amazon Web Services a dle 2019 Embedded Markets Study [8] je druhým nejpoužívanějším veřejně dostupným RTOS na trhu (po OS Linux pro vestavěná použití). Z technických vlastností:

Velikost – jádro je naimplementováno ve třech modulech a na platformě obvykle okupuje 6–12 KB

Plánovač – preemtpivní jádro používá ve výchozím nastavení plánovač Round Robin, ovšem lze konfigurovat i pro priotirní či plánování RMS

Synchronizace – jeho rozhraní nabízí řadu synchronizačních primitiv: semafory (binární i čítací), fronty, mutexy včetně rekurzivní varianty, a další struktury pro mezi-úlohovou komunikaci: buffer pro proud zpráv či notifikace událostí pro úlohy

Ověřené verze – existuje několik příbuzných RTOS, které jsou matematicky ověřené jako správné či mají komerční podporu

Režim nízké spotřeby – při odpovídající konfiguraci lze zapnout *tickless* režim, kdy může být procesor uspáván po dobu bez aktivního úlohy

2.3 ChibiOS

ChibiOS je projekt [15, 6] několika knihoven zaměřených na RTOS, z nichž relevantní pro tuto práci jsou ChibiOS/RT a ChibiOS/NI – první jmenovaný je optimalizovaný na výkon, druhý na velikost. Jeho vývoj začal v roce 2007, licencovaný je s GPL3 (část modulů proprietárně) a v dnešní době je udržován jedním hlavním vývojářem. Z technických vlastností ChibiOS/RT:

Velikost – jádro bez ChibiOS/HAL (*Hardware Abstraction Layer*) přeložené zabírá 5–10 KB dle cílové platformy

Plánovač – preemptivní jádro používá plánování Round Robin, ovšem lze konfigurovat na prioritního plánování s FCFS při shodných prioritách

Synchronizace – samotné jádro nabízí práci s událostmi, mutexy a čítací semafory, ChibiOS/OSLIB poté přidává podporu pro binární semafory, proudy dat,

2.4 RTEMS

RTEMS (*Real-Time Executive for Multiprocessor Systems*) je RTOS [14, 11] s otevřeným zdrojovým kódem, jehož vývoj započal již v osmdesátých letech minulého století. Projekt

RTEMS částečně implementuje standard POSIX 1003.1b (v oblastech, kde to na vestavěných systémech dává smysl) a v dnešní době je vyvíjen společností OAR Corporation. Z jeho technických vlastností:

Plánovač – jádro RTEMS podporuje několik plánovacích algoritmů, od prioritního plánování, přes plánování Round Robin a RMS, až k algoritmu CBS[3] – v každém z typů plánování lze zapnout či vypnout preempci

2.5 RT-Thread

[18]

2.6 uCOS-II

[7, 16]

2.7 Raspbian s PREEMPT_RT

Projekt PREEMPT_RT je patch[4] do operačního systému Linux, který z něj částečně dělá systém schopen preemptivní plánování – míra schopnosti je závislá na použité konfiguraci. [9, 13]

[3] CBS (*Constant Bandwidth Server*) [2] je plánování, ...

[4] Patch soubor úprava zdrojových kódů publikovaná jako soubor úprav pro utilitu `patch`.

Kapitola 3

Raspberry Pi 3B+

Raspberry Pi je malý jednodeskový počítač.

3.1 Dostupný hardware

BCM28370

Mini Uart

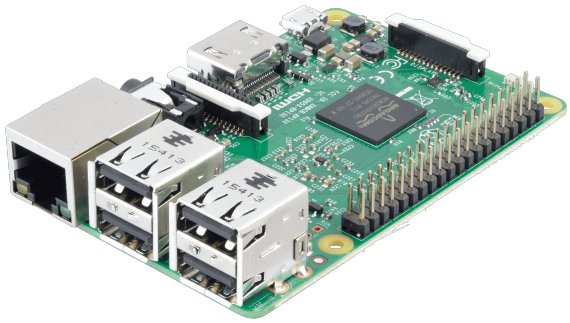
GPIO

Cortex-A53 r0p4 + VFP [4]

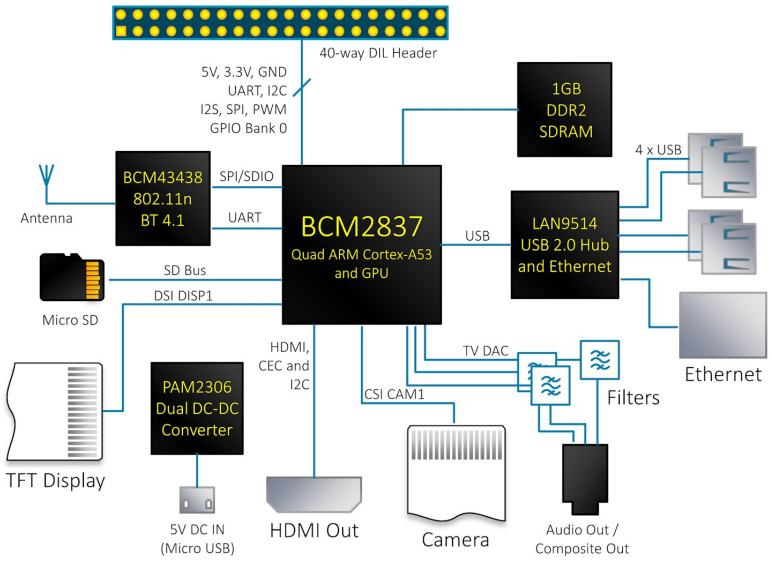
Broadcom Videocore-IV

3.2 Zavádění systému

V této kapitole bude popsán proces zavedení systému na vybraném Raspberry Pi 3B+ pro výchozí nastavení – tedy je použit dodávaný firmware pro čip BCM2387 a všechny úrovně zavaděče jsou použitý ve výchozích verzích a konfiguracích. Všechny soubory v této kapitole kromě `kernel.img` jsou výrobcem dodány pouze v binární formě.



Obrázek 3.1: Raspberry Pi 3B+, dostupné z https://www.reichelt.com/magazin/en/wp-content/uploads/2017/07/RASP_03_01.png



Obrázek 3.2: Blokové schéma raspberry Pi 3B+, dostupné z https://community.element14.com/products/raspberry-pi/b/blog/posts/raspberry-pi-3-block-diagram

3.2.1 Nalezení zavaděče druhé úrovně

Proces startu tohoto mikropočítače začíná startem GPU jednotky, přičemž ARM jádro a SDRAM jsou vypnuty, a jednotka využívá vestavěnou ROM pro čtení programu zavaděče první úrovně.

GPU jednotka v první kroku určí mód zavedení. Tím je ve výchozím nastavení start z SD karty, kterou se pokusí najít a připojit její první FAT oddíl na sběrnici MMC – hledaným je následně zavaděč druhé úrovně v souboru `bootcode.bin`.

Pokud proces připojení selže, jednotka se dále pokusí o zavedení z lokální sítě – po požadavku na DHCP server provede TFTP[1] požadavek na soubory `bootcode.bin` a `bootsig.bin`. První z nich z nich obsahuje zavaděč druhé úrovně, druhý je očekávaně dle specifikace nenalezen.

3.2.2 Běh zavaděče druhé úrovně

Kód zavaděče druhé úrovně je GPU jednotkou načten do L2 cache a následně spuštěn. Zavaděč druhé úrovně zavede soubor `start.elf` obsahující firmware pro GPU jednotku, stejně tak jako soubor `config.txt` obsahující konfiguraci pro kernel zavedený v dalším kroku. Firmware na jednotce GPU nastaví časování a jeho zdroje pro ARM jádro, zapne SDRAM a provede konfiguraci samotné GPU jednotky v oblasti akcelerované zpracování grafiky.

Zavaděč první úrovně svou práci ukončí zavedením souboru `kernel.img`, který nahraje na SDRAM a od adresy `0x8000` aktivuje jádro ARM procesoru.

[1] popsat TFTP

3.3 Architektura ARMv8-A

Kapitola 4

Demonstrační aplikace A

V této kapitole bude popsána tvorba demonstrační aplikace s RTOS implementující tři oddělené úlohy – dva čítače po 10 Hz a 50 Hz, a také monitorovací úlohu reportující stav čítačů na sériovou linku. Jako RTOS byl vybrán FreeRTOS díky jeho nízkým platformním požadavkům, přirozené podpoře pro prioritní plánování a možnosti kompletní statické alokace úloh. V potaz je také nutno vzít jeho popularitu, licenční politiku a veřejný zdrojový kód, aktivní vývoj (je vydáván na měsíční bázi) a širokou komunitní podporu skrz oficiální fórum.

4.1 Prostředí pro programy v jazyce C

Cílem této kapitoly je popsat vytvoření běhového prostředí pro programy v jazyce C na mikropočítači Raspberry Pi 3B+ . Běhové prostředí a jeho tvorba budou popsány z pohledu časové postupnosti.

Aplikace bude sestavena nad sadou nástrojů `arm-none-eabi` – tedy pro architekturu `aarch32` implementující Arm EABI. Základní stavební kámen je soubor pravidel `Makefile` odpovídá typickému použití pro projekty v jazyce C. Je vhodno zmínit některé argumenty pro kompilátor `arm-none-eabi-gcc` překládající část projektu v jazyce C:

`-mcpu=cortex-a53+crypto`
: Nastavuje cílový procesor (a tedy i architekturu) – `gcc` je schopno optimalizací přímo na cílový procesor, jinak by postačovali i použití `-mcpu=generic-arch`. Modifikátor `+crypto` zapíná podporu pro kryptografické rozšíření instrukční sady (vestavěná podpora pro algoritmy AES, SHA nebo ECC).

`-mfpu=neon-fp-armv8 -mfloat-abi=hard`
: Cortex A53 obsahuje VFP jednotkou ve specifikaci `VFPv4-D16`, je na místě ji použít – kompilátor je takto instruován používat přímo VFP jednotku pro aritmetiku s plovoucí řádovou čárkou. Konkrétné ji použije ve specifikaci `neon-fp-armv8`, což značí použití NEON standardu (extenze pro základní specifikaci ARM VFP) – tento standard je od ARMv8 kompatibilní s IEEE 754. Jak definuje Arm EABI[3], pro práci s jednotkou je díky `-mfloat-abi=hard` kompilátor instruován k jejímu přímému použití (bez emulace případnými CPU instrukcemi).

`-ffreestanding`
: Kompilátor instruuje nepředpokládat, že bude přítomna standardní definice knihovny C a vstupní symbol programu bude funkce s podpisem `int main(void);`

`-nostartfiles`
: Vypíná automatické linkování standardní knihovny – v tomto případě nebude nalinkována knihovna `libc`.

O slinkování se stará `arm-none-eabi-ld` ve výchozím nastavení, pro jeho instruování je použit skript `raspberry.ld` který definuje rozmistění jednotlivých objektů do výsledného binárního souboru – konkrétní rozmístění je popsáno v tabulce 4.1.

Tabulka 4.1: Mapa mapěti pro linker

začátek	sekce	obsah	velikost
0x00000		rezervováno	32 KB
0x08000	init	vstupní bod	32 KB
0x10000	text	samotný kód a RO data	128 MB
0x2000000000	bss	statické symboly	dynamická
po předchozím	heap	halda	dynamická
po předchozím	stack	zásobník	dynamická

Na adresu 0x8000, od které je ARM jádro spuštěno, je zavedena funkce `_start` z ukázky 4.1, jejíž implementace není plnohodnotnou funkcí ve smyslu jazyka C – je definovaná s atributem `naked` a tedy kompilátor pro ni nevygeruje kód zajišťující prostor na zásobníku a návrat z ní. Tato funkce je umístěna do sekce `.init`, což zaručí, že ji linker umístí na odpovídající místo. Její minimalistická implementace pouze nastaví ukazatel na zásobník na správné místo a následně volá plnohodnotnou C funkci `start`, která se postará o další konfiguraci procesoru.

```
extern uint32_t __stack_end;
void __attribute__((section(".init"), naked)) _start() {
    // set SP into stack space
    set_sp((uint32_t) (__stack_end));
    // call the true entrypoint
    start();

    while (1) {};
}
```

Zdrojový kód 4.1: Prvotní rutina volaná jádrem procesoru, která není funkcí z pohledu jazyka C – proto pouze nastaví ukazatel na zásobník na správné místo.

Procesor následně pokračuje funkcí `start`, která se postará o další konfiguraci:

1. Pomocí `zero_bss_section` vynuluje sekci statických symbolů (vzhledem k tomu, že to není zajištěno architekturou).

2. Spustí supervizor režim jádra – to zajistí pomocí nastavení odpovídajících bitů do registru `CPSR` (Current Processor State Register). Nutno je taktéž zabezpečit, že se procesor (např. po resetu) nenechází v hypervizor módu.

3. Zapne L1 cache pro dostupné instrukce a *branch predictor* – využije registr SCTLR (System Control Register).

4. Povolí použití jednotky VFP pro všechny režimy běhu procesoru – tedy jak ne/zabezpečené, tak ne/privilegované.

5. Nastaví tabulku vektorů pro zpracování přerušení – implementace jednotlivých rutiny pro zpracování bude popsána v sekci 4.2. Tabulka je u této architektury uložena ihned na začátku paměti, adresy jednotlivých rutin jsou nastaveny na začátek paměti.

6. Vhledem k tomu, že stack pointer je jeden z tzv. bankovaných registrů[1], je vhodné pro jednotlivé exekuční režimy procesoru nastavit jeho hodnoty. Funkce `setup_stacks` pro každý uvažovatelný mód nastaví adresu samotnatného prostoru pro zásobník.

7. A konečně, prostředí je nakonfigurováno dostatečně tak, že lze spustit samotnou aplikaci – to pomocí vstupní funkce `void main(void)`.

4.2 Podpora pro FreeRTOS

Tato kapitola popíše zavedení FreeRTOS do vytvořeného prostředí – ten bude zaveden ve verzi `202111.00` z listopadu 2021. Pro jeho běh na zařízení je důležitých několik částí:

Konfigurace frameworku
Soubor `FreeRTOSConfig.h` definuje základní konfiguraci pro aplikaci, níže výběr důležitých maker z něj:

`configCPU_CLOCK_HZ = 100000000UL` – základní CPU frekvence, tedy 100 MHz

`configTICK_RATE_HZ = 1000` – frekvence tiku operačního systému v Hz

`configUSE_PREEMPTION = 1` – zapne preemptivní mód, tedy připínání kontextu bez přímé spolupráce od právě běžící úlohy

`configUSE_TIME_SLICING = 0` – vypne přepínání úloh stejné priority na základě vyčerpání časového kvanta (tedy v praxi zapne prioritní plánování pro plánovač namísto Round Robin plánovaní)

Implementace pro platformní podporu jádra OS
Pro podporu je nutné naimplementovat a definovat sadu funkcí a konstant sloužících frameworku pro práci s hardware systému. Dále výčet těch nejdůležitějších:

`void prvSetupTimerInterrupt(void)` – funkce zodpovědná za prvotní zapnutí časovače zajišťující tik pro operační systém. V tomto případě je prvně aktivován ARM časovač s řídícími registry od paměťové adresy `0x7E00B000`, jehož odčítaná hodnota a předdělička jsou odvozeny z požadované frekvence systémového tiku, a následně je čítač spuštěn.

[1] Pracovní registry jsou na této architektuře rozděleny do bank, jejichž základní adresa je různá pro každý z exekučních režimů procesoru.

`portENABLE_INTERRUPTS a portDISABLE_INTERRUPTS` – dvojice maker používaná frameworkem pro zapnutí, resp. vypnutí přerušení `IRQ` a `FIQ` (typicky při kritických sekcích nebo práci jádra OS). Implementovány jsou pomocí modifikace řídícího registru `CPSR`.

`vPortHandleISR` – důležitá rutina zpracovávající příchozí přerušení, jejíž adresa je nastavená v tabulce vektorů přerušení. Rutina implementuje přepnutí kontextu, tedy ten akuální uloží na zásobník aktuální úlohy, voláním `vTickISR` oznámí operačnímu systému přerušení (ten v tu chvíli může dle situace přepnout úlohu) a následně obnoví ze zásobníku kontext pro potenciálně jinou úlohu.

Za zmínění stojí přístup na globální symbol `pxCurrentTCB`, do kterého uloží FreeRTOS ukazatel na vrchol zásobníku aktivní úlohy – ten je při uložení kontextu uložen do této proměnné, při obnocení naopak získán.

`vPortYieldProcessor` – rutina zpracovávající programově vyvolané přerušení, které nastává v případě dobrovolného uspání úlohy při jejím provádění. Její adresa je taktéž uložena v tabulce vektorů přerušení a při spuštění uloží kontext, nabídne operačnímu systému přepnutí úlohy a následně kontext obnoví.

`portENTER_CRITICAL a portEXIT_CRITICAL` – dvojice maker implementující vstup a výstup z kritické sekce. Využívají odpovídající makra pro zapnutí a vypnutí přerušení, a navíc pomocí čítače implementují rekurzivní zanořování kritických sekcí.

Implementace ovladačů periférií

Pro vývoj základních aplikací je záhodno implementovat i práci s periferiemi – do základu stačí ovládání GPIO na desce, ovladač pro Mini UART a Mailbox.

Ovládání GPIO – do modulu `gpio.c` je naimplementována podpora pro ovládání GPIO. Řídící registry pro tyto piny začínají na paměťové adrese `0x3F200000` a sada funkcí implementovaná v tomto modulu je schopna nastavování funkcí pro jednotlivé piny.

Mini UART – modul implementuje základní ovládání odpovídajícího rozhraní, tedy inicializaci, výpis a čtení po znaku, a taktéž vyprázdnění odesílací fronty. Jeho funkce je důležitá pro implementaci základní knihovny jazyka C v 4.3.

Ovladač pro Mailbox – díky požadavkům ve formě zpráv na GPU lze mj. ovládat druhou signalizační LED diodu[2], pro účely vývoje je tedy implementován pouze tento typ zpráv.

4.3 Standardní knihovna jazyka C

Vzhledem ke kompilaci bez přiložení standardní knihovny jazyka C je v případě, kdy chceme funkce z této knihovny používat, nutné nahradit její implementaci. Pro ni bude užito implementace z projektu Newlib[3], který poskytuje standardní knihovnu v implementaci plně

[2] Druhá LED dioda není připojena na GPIO a její ovládání je dostupné pouze z GPU – po zaslání zprávy s tagem `0x00038041` je nastavit její status: https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface#set-onboard-led-status.

[3] https://sourceware.org/newlib/

funkční na vestavěných systémech. Newlib poskytuje implementaci ve formě zdrojových kódů, je ji tedy nutné zkompilovat pro konkrétní architekuturu.

K takto použité standardní knihovně je nutno doiplementovat systémová volání konkrétních standardních funkcí, které bude aplikace používat. Pro usnadnění vývoje na platformě bude implementována práce s dynamickou pamětí a standardním výstupem – obě v modulu `syscalls.c`, který je přímo linkován do výsledku.

Dynamická paměť – v jazyce C se jedná o volání funkcí z rodiny `malloc`. To vede na systémové volání symbolu `void* _sbrk(intptr_t increment)`. Jako volná paměť se využívá prostor určený pro haldu – tedy vzestupně od symbolu `__heap_start` definovaného linker skriptem (rozdělení popsáno v tabulce 4.1).

Standardní výstup – volání standardních funkcí z rodiny `printf` vedou na systémová volání funkce `ssize_t _write(int fd, const void* buf, size_t count)`. Ta předpokládá inicializované rozhraní Mini UART a obsah, který je zapisován na standardní výstup, zasílá pomocí funkce `uart_send` na konzoli. Pro každé systémové volání je taktéž vyprázdněn buffer UART rozhraní pomocí `uart_flush`. Taktéž je třeba poskytnout implementaci pro systémové volání `_close`, `_fstat`, `_isatty` a `_lseek` – jejich implementace na OS s podporou pro souborové systémy nedává smysl, jsou tedy prázdné a vrací výchozí hodnotu odpovídající jejich návratovému typu.

4.4 Tvorba samotné aplikace

Demonstrace funkce bude provedena pomocí tří periodických úloh z definice této práce. Konkrétně se jedná o dvojici `task50Hz` a `task10Hz` implementující čítače o odpovídající frekvenci, a monitorovací úlohu `taskPrinter`, která implementuje periodický výpis na sériovou linku. Ústřední konstrukcí úloh je nekončená smyčka, před kterou se nachází inicializační část. Frekvence spouštění úloh je odvozena od hodnoty makra `portTICK_PERIOD_MS` – to z FreeRTOS definuje počet tiků operačního systému za jednu milisekundu.

Ve smyčce poté obsahují úlohy kritickou sekci a volání `xTaskDelayUntil`, které je schopno na základě stavu čítače tiků z předchozího obnovení úlohy a požadované frekvence úlohy uspat úlohu na odpovídající čas – úloha je naplánovaná na opětovné spuštění a pomocí programového přerušení je vyvoláno přepnutí kontextu. Kritická sekce obsahuje v případě čítačových úloh inkrementaci příslušných čítačů – ukazatel na čítač obdrží úloha jako parametr (je nutné konkretizovat jeho typ, vzhledem k tomu, že parametry FreeRTOS úlohy mají generický typ `void *`). Ukázka 4.2 zobrazuje implementaci 50Hz čítače.

Monitorovací úloha v kritické sekci přistoupí na oba čítače a společně s aktuální hodnotou systémového čítače tiků je vytiskne na standardní výstup (tedy na sériovou linku díky implementaci standardní knihovny z 4.3).

```
void task50Hz(void* pParam) {
    unsigned long* counter = (unsigned long*) pParam;

    const TickType_t tickFreq = (1000 / 50) / portTICK_PERIOD_MS;

    *counter = 0;

    TickType_t lastWakeTime = xTaskGetTickCount();
    while (1) {
        taskENTER_CRITICAL();
        (*counter)++;
        taskEXIT_CRITICAL();

        xTaskDelayUntil(&lastWakeTime, tickFreq);
    }
}
```

Zdrojový kód 4.2: Kód úlohy implementující 50 Hz čítač. V inicializační sekci před smyčkou je nastavena požadovaná frekvence a pomocí ukazatele je čítač vynulován. Ve smyčce je v kritické sekci inkrementován čítač a úloha je uspána na dobu potřebnou k dosažení frekvence 50 Hz.

O samotné spuštění plánovače a úloh se následně stará vstupní bod aplikace funkce `void main(void)`. Ten zapne Mini UART, definuje paměťové místo pro oba čítače, naplánuje úlohy a spustí plánovač. Plánování úloh využívá funkci z FreeRTOS `xTaskCreate` s následujícími parametry:

`TaskFunction_t pvTaskCode` ukazatel na kód úlohy

`const char * const pcName` název úlohy

`configSTACK_DEPTH_TYPE usStackDepth` velikost zásobníku pro úlohu

`void *pvParameters` ukazatel na generický parametr pro úlohu

`UBaseType_t uxPriority` priorita úlohy

`TaskHandle_t *pxCreatedTask` volitelně návratová hodnota pro uložení ukazatele na vytvořenou úlohu

Z ukázky 4.3 je tedy zřejmé, že úlohy k čítačům přistupují na úroveň lokální zásobníků funkce `main`. Priority úloh jsou zvolena nad úroven normální priority z OS.

```
void main(void) {
    uart_init();
    unsigned long counters[2];

    xTaskCreate(task10Hz,    "task10Hz",    256, counters + 0,
                tskIDLE_PRIORITY + 1, NULL);

    xTaskCreate(task50Hz,    "task50Hz",    256, counters + 1,
                tskIDLE_PRIORITY + 1, NULL);

    xTaskCreate(taskPrinter, "taskPrinter", 256, counters,
                tskIDLE_PRIORITY + 2, NULL);

    vTaskStartScheduler();
    while (1);
}
```

Zdrojový kód 4.3: Vstupní funkce aplikace `main`. Ta ve svém těle nejprve deklaruje prostor pro interní čítače, následně naplánuje spuštení tří úloh pro plánovač a ten poté spustí.

4.5 Vyhodnocení funkce

```
0;0;0
3000;30;150
6000;60;300
9000;90;450
12000;120;600
 ...
```

Zdrojový kód 4.4: Provoz na sériové lince po zapnutí aplikace na Raspberry Pi 3B+, monitorovací úloha tiskne stav aktuálního čítače tiků samotného systému, a stav obou interních čítačů.

```
0.15273;3000;30;150
0.397637;6000;60;300
-0.477973;9000;90;450
0.82201;12000;120;600
-0.52693;15000;150;750
-0.103775;18000;180;900
0.269487;21000;210;1050
-0.6214;24000;240;1200
0.7383;27000;270;1350
-0.455258;30000;300;1500
 ...
```

Zdrojový kód 4.5: Měření na monitorovací úlohy druhém konci sériové linky – reportován je jitter (ms) oproti očekávané periodě 3 sekund.

Kapitola 5

Demonstrační aplikace B

Navrhněte a implementujte bezpečné sdílení společných dat mezi procesy.

Navrhněte a implementujte kód pro připojení modulu Canberry k procesu s nižším kmitočtem (viz předchozí bod). Ověřte komunikaci po CAN sběrnici zobrazováním hodnot čítačů jednotlivých procesů, které budete periodicky vysílat prostřednictvím modulu Canberry.

Postup tvorby této demonstrační aplikace zdokumentujte a vyhodnoťte její vlastnosti.

5.1 Bezpečné sdílení dat mezi úlohami

5.1.1 Dostupné prostředky z FreeRTOS

5.1.2 Implementace bezpečného sdílení

5.2 Ovladač pro modul Canberry

5.2.1 Specifikace sběrnice CAN

5.2.2 Implementace ovladače

5.3 Tvorba demonstrační aplikace

Kapitola 6

Použití pro distribuované aplikace

Na základě znalostí získaných z bodů 2 a 3 diskutujte možnosti využití vámi vybraného RTOS na platformě Raspberry Pi s modulem Canberry pro tvorbu distribuovaných řidících aplikací.

Kapitola 7

Závěr

Literatura

[1] FreeRTOS™ Real-time operating system for microcontrollers. [cit. 2021-12-06].
URL https://www.freertos.org/

[2] ABENI, L.; LIPARI, G.; LELLI, J.: Constant Bandwidth Server Revisited. *SIGBED Rev.*, ročník 11, č. 4, jan 2015: str. 19–24, doi:10.1145/2724942.2724945.
URL https://doi.org/10.1145/2724942.2724945

[3] Arm: *Application Binary Interface for the Arm® Architecture - The Base Standard*. [cit. 2021-11-18].
URL https://developer.arm.com/documentation/ihi0036/latest

[4] Arm: *Arm® Cortex®-A53 MPCore Processor*. 08 2013.

[5] BARRY, R.: *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009.

[6] Cabrera-Gamez, J.; de Miguel, A. R.; Dominguez-Brito, A. C.; aj.: A Real-Time Sailboat Controller Based on ChibiOS. In *Robotic Sailing 2014*, editace F. Morgan; D. Tynan, Cham: Springer International Publishing, 2015, ISBN 978-3-319-10076-0, s. 77–85.

[7] DELANEY, S.; EGBERT, D. D.; HARRIS, F. C.: RealPi - A Real Time Operating System on the Raspberry Pi. In *Proceedings of 34th International Conference on Computers and Their Applications*, ročník 58, 2018, s. 8–16.

[8] 2019 Embedded Markets Study. 03 2019, [cit. 2022-01-10].
URL https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[9] The Real Time Linux collaborative project.
URL https://wiki.linuxfoundation.org/realtime/start

[10] LIU, C. L.; LAYLAND, J.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, ročník 20, č. 1, jan 1973: str. 46–61, ISSN 0004-5411, doi:10.1145/321738.321743.
URL https://doi.org/10.1145/321738.321743

[11] NICODEMOS, F.; SAOTOME, O.; LIMA, G.: RTEMS Core Analysis for Space Applications. In *2013 III Brazilian Symposium on Computing Systems Engineering*, 2013, s. 125–130, doi:10.1109/SBESC.2013.22.

[12] O'NEIL, P. E.; RAMAMRITHAM, K.; PU, C.: A Two-Phase Approach to Predictably Scheduling Real-Time Transactions. In *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, 1996, s. 42–42.

[13] REGHENZANI, F.; MASSARI, G.; FORNACIARI, W.: The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM Comput. Surv.*, ročník 52, č. 1, feb 2019, ISSN 0360-0300, doi:10.1145/3297714.

[14] Real-Time Executive for Multiprocessor Systems. [cit. 2021-11-23].
URL https://devel.rtems.org/

[15] SIRIO, G. D.: ChibiOS.
URL
https://www.chibios.org/dokuwiki/doku.php?id=chibios:documentation:start

[16] STRNADEL, J.: Návrh časově kritických systémů IV: realizace prostředky RTOS. *Automa*, ročník 2011, č. 4, 2011: s. 58–60, ISSN 1210-9592.

[17] STRNADEL, J.: Real-Time Systems. 2021.
URL TODO

[18] Team, B. X. . R.-T.: RT-Thread.
URL https://devel.rtems.org/

[19] ČERNOHORSKÝ, J.; SROVNAL, V.: Systémy reálného času (3). *AT&P journal*, 08 2005.