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

Cílem této práce je realizace podpory volně dostupného operačního systému reálného času s fixní prioritou úloh na mikropočítači Raspberry Pi 3B+. Jako vhodný systém je vybrán projekt FreeRTOS, pro který je v práci zrealizováno běhové prostředí a představena podpora pro tvorbu uživatelských aplikací s preemptivními úlohami. To je prezentováno pomocí dvou demonstračních aplikací, z nichž první využívá dvou periodických úloh a monitorování sériovou linkou, a v rámci druhé je vytvořena podpora pro sběrnici CAN, pomocí které je stav úloh s čítači reportován. Výsledkem práce je tedy funkční úprava systému FreeRTOS určená pro běh na mikropočítači Raspberry Pi 3B+ vhodná pro časově kritické aplikace.

Abstract

The main goal of this work is to create a support for an open-source real-time operating system on the computer Raspberry Pi 3B+. The project FreeRTOS is selected as a great candidate for further work. The runtime environment and support for user-space applications are presented. Two demonstration applications serve as proofs of support, the first one uses two periodic tasks and reports their state to the serial interface. The second demonstration application runds the same periodic tasks, but reporting the state is done using the CAN bus, for which is the driver realised. The result of this thesis is a working system FreeRTOS for Raspberry Pi 3B+ computer with support for time-critical usages.

Klíčová slova

operační systém reálného času, plánovač, preemce, FreeRTOS, Raspberry Pi, Arm, sběrnice CAN, MCP2515

Keywords

real-time operating system, scheduler, preemption, FreeRTOS, Raspberry Pi, Arm, CAN bus, MCP2515

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
18. května 2022

Poděkování

Rád bych tímto vyjádřil mé poděkování doc. Ing. Vladimíru Janouškovi, Ph.D., za odborné vedení, rady a konzultace při realizaci této práce. Ta nejvřelejší poděkování z hloubi duše posílám Ing. Zuzaně Greguškové za její nezištnou vytrvalou podporu, smysluplné komentáře a neutuchající elán při motivaci k dokončení této práce.

Obsah

1 Úvod **2**

2 Operační systémy reálného času **3**
2.1 Definice RTOS . . . 3
2.2 Specifika RTOS . . . 3
2.3 FreeRTOS . . . 9
2.4 ChibiOS . . . 9
2.5 RTEMS . . . 10
2.6 RT-Thread . . . 10
2.7 µC/OS . . . 11
2.8 OS Linux jako RTOS . . . 11

3 Raspberry Pi 3B+ **14**
3.1 Základní informace o zařízení . . . 14
3.2 Dostupný hardware . . . 14
3.3 Zavádění systému . . . 17

4 Demonstrační aplikace A **19**
4.1 Nastavení překladače a linkeru . . . 19
4.2 Podpora pro FreeRTOS . . . 22
4.3 Běhová a standardní knihovna jazyka C . . . 24
4.4 Tvorba samotné aplikace . . . 25
4.5 Sestavení a spuštění aplikace . . . 29
4.6 Vyhodnocení aplikace . . . 30

5 Demonstrační aplikace B **33**
5.1 Bezpečné sdílení dat mezi úlohami . . . 33
5.2 Realizace bezpečného sdílení dat v FreeRTOS . . . 33
5.3 Sběrnice CAN . . . 34
5.4 Ovladač pro modul sběrnice CAN . . . 36
5.5 Tvorba demonstrační aplikace . . . 37
5.6 Běh aplikace . . . 38

6 Použití pro distribuované řídící aplikace **39**

7 Závěr **40**

Literatura **41**

Kapitola 1

Úvod

Operační systémy reálného času, na rozdíl od systémů pro obecné použití, jsou provozovány v aplikacích kritických na správné časování běhu systému a jeho vstupně-výstupních operací. Díky definovaným časovým zárukám, které takové systémy nabízí, je jejich použití v oblastech průmyslové výroby, robotiky, automatizace či zdravotnictví na místě. V takových aplikacích by nedostatečná časová přesnost mohla způsobit nevratné škody.

Na druhou stranu, mikropočítač Raspberry Pi je komerční běžně dostupné zařízení mířící na široké spektrum využití od edukačních účelů, přes automatizaci domácností, až k síťovým aplikacím s velkým množstvím spolupracujících počítačů. Jeho úspěch tkví v souhře nízkých pořizovacích nákladů, početné nabídky zabudovaných rozhraní, vysokého výkonu integrovaného čipu, a silné komunity okolo tohoto počítače přezdívaného „malina“.

Kombinací operačního systému reálného času s počítačem Raspberry Pi získáváme možnost efektivního vývoje časově kritických aplikací na zařízení, které disponuje širokou škálou již připravených rozhraní. Tato kombinace je tedy vhodná mj. pro dostupné distribuované řídící aplikace.

V následujících kapitolách bude rozebrána problematika operačních systémů reálného času a představeni zástupci systémů s otevřeným zdrojovým kódem – pro systém FreeRTOS bude navrhnuta a zrealizována podpora běhu přímo na mikropočítači, což bude demonstrováno pomocí souběžně běžících úloh na zařízení. Jako druhá demonstrační úloha bude představena podpora pro sběrnici CAN a bezpečné sdílení dat mezi úlohami při komunikaci s ní. Na závěr pak bude diskutováno použití spojení Raspberry Pi s FreeRTOS pro distribuované řídící aplikace.

> 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

Tato kapitola se věnuje systémům reálného času z pohledu jejich definice v kapitole 2.1 a jejich specifickým vlastnostem oproti operačním systémům obecného použití v kapitole 2.2. Druhá část obsahuje informace a charakterizaci několika vybraných volně dostupných RTOS v kapitolách 2.3, 2.4, 2.5, 2.6, 2.7 a na závěr specifického případu, kterým je OS Linux při použití jakožto RTOS v kapitole 2.8.

2.1 Definice RTOS

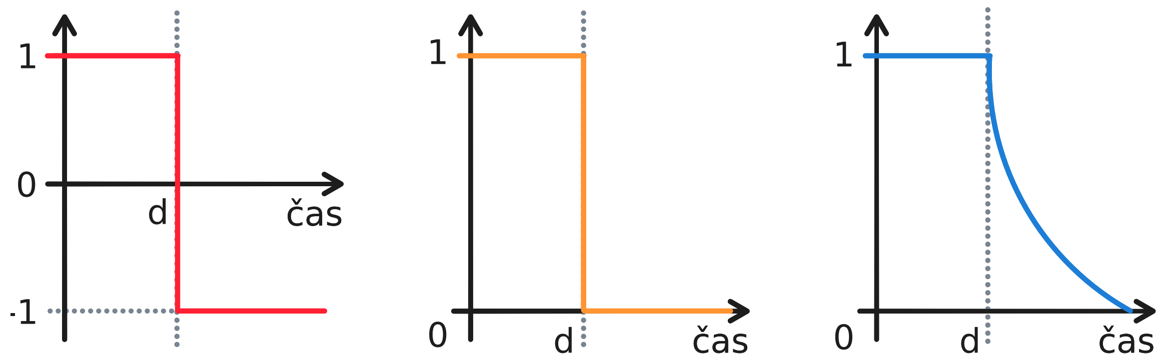
Operační systém reálného času (*Real-Time Operating System, RTOS*) je typ operačního systému, který poskytuje možnost reagovat na události v okolí počítače průběžně (tj. v reálném čase) [18].

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 2.2.1 [38].

Oproti operačním systémům reálného času, obecné operační systémy (*General Purpose OS, GPOS*) nezajišťují obecně deterministické plánování a úlohy nemají zaručený procesorový čas [27, 23]. V GPOS je k úlohám přistupováno férově a jejich priorita není na prvním místě důležitosti při plánování běhu. Zároveň nejsou jádra GPOS připravena k preempci, resp. můžou obsahovat dlouhé kritické sekce.

2.2 Specifika RTOS

Jedním z klíčových požadavků na operační systém reálného času je jeho schopnost preempce. Preempce je přerušení právě vykonávané úlohy bez přímé kooperace s jádrem systému. Díky preempci může systém přerušit právě běžící úlohu bez její spolupráce a nahradit ji jinou úlohou pomocí přepnutí kontextu.



Obrázek 2.1: Diagram tříd RTOS dle použitelnosti výsledku okolo časového limitu – vertikální osa značí použitelnost výsledku, kde hodnota 1 značí dosažení požadovaného výsledku, 0 nepoužitelný výsledek při nesplnění limitu a −1 nepoužitelnost výsledku a navíc škodu-způsobující následky. Zleva se nejprve jedná o třídu tvrdých RTOS, uprostřed stálé systémy a vpravo poté systémy v měkké třídě.

2.2.1 Třídy RTOS dle důležitosti plnění časového limitu

Podle toho, jak je splnění časového limitu kritické pro systém, dělíme operační systémy reálného času do tříd [29, 33, 9] (vyobrazených též na obrázku 2.1):

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 dopravních prostředků či stroje ve zdravotnictví.

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

2.2.2 Plánovač

Plánovač je pro RTOS kritická část zajišťující samotný deterministický běh systému v reálném čase. Je zodpovědný za deterministické plánování běhů úloh – to je obvykle prak-

[1] QoS – Quality of Service je vlastnost popisující kvalitu služby vzhledem k požadovanému výsledku – v kontextu měkké třídy operačních systémů reálného času je jedná o kvalitu požadovaného produktu úlohy konkrétního RTOS.

ticky zajištěno nastavením priorit pro jednotlivé úlohy. Plánovač poté garantuje spouštění úloh podle jejich priority. Samotné plánovače využívají preemptivnosti úloh a můžou implementovat jeden z několika možných algoritmů plánování, níže následuje výběr v praxi užívaných [18, 43]:

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 – podrobněji je předávání řízení popsáno v obrázku 2.2.

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.

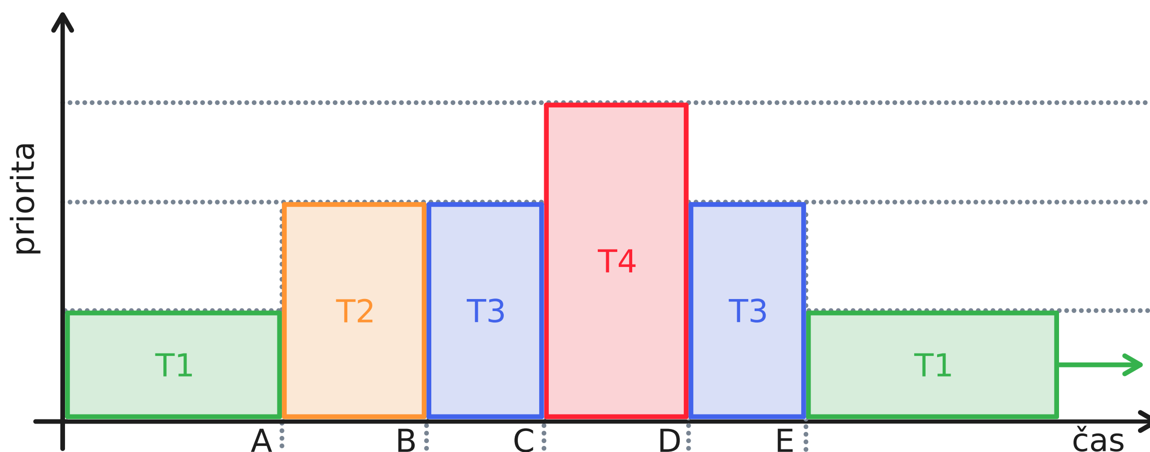
Princip round-robin plánování je zobrazen a popsán na obrázku 2.3.

Frekvenčně monotónní plánování (*Rate Monotonic Scheduling, RMS*)
: 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 [26], 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:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \qquad U \leq n(2^{1/n} - 1) \tag{2.1}$$

Liu a Layland [26] zároveň ukázali, že nelze-li plánovat množinu úloh algoritmem frekvenčně monotónního plánování, nelze ji naplánovat ani žádným jiným algoritmem s fixní prioritou úloh. Časový diagram pro toto plánování je zobrazen na obrázku 2.4.

[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.



Obrázek 2.2: Časový diagram pro prioritní plánování s vnitřním FCFS plánováním: při práci úlohy T1 s nejnižší prioritou dojde k preempci a úloha T2 s vyšší prioritou v čase A dostane řízení. V čase B odevzdá dobrovolně úloha T2 řízení, čímž se na řadu dostane úloha T3 díky vnitřnímu plánování FCFS, která přešla do stavu připravenosti na běh mezi časy A a B. Řízení je úloze T3 preemptivně odebráno v čase C, kdy je na běh připravena úloha T4 s nejvyšší prioritou. Ta poté v čase D odevzdá řízení T3, která následně v čase E vrátí řízení méně prioritní úloze T1.

Plánování dle nejbližšího termínu (*Earliest Deadline First, EDF*)
je plánování založené na dynamických prioritách úloh stanovenými dle jejich termínů, do kterých musí být úloha vykonána. Plánovač nastavuje nejvyšší prioritu úloze s nejbližším termínem dokončení – na rozdíl od frekvenčně monotónního plánování zde není podmínka na úlohy, aby prováděly periodické činnosti. Pro tento algoritmus taktéž existuje test naplánovatelnosti [26], u kterého pro stejné prostředí jako u frekvenčně monotónního plánování musí platit:

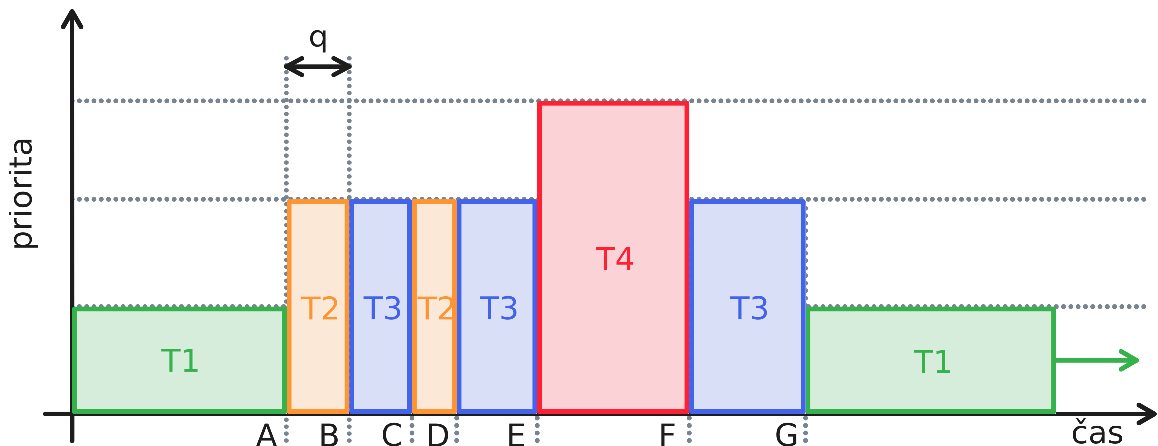
$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \qquad U \leq 1 \tag{2.2}$$

Plánování dle nejbližšího termínu je při preemptivních aplikacích [42] optimální plánovací algoritmus za podmínky, že je daná množina úloha naplánovatelná dle testu.

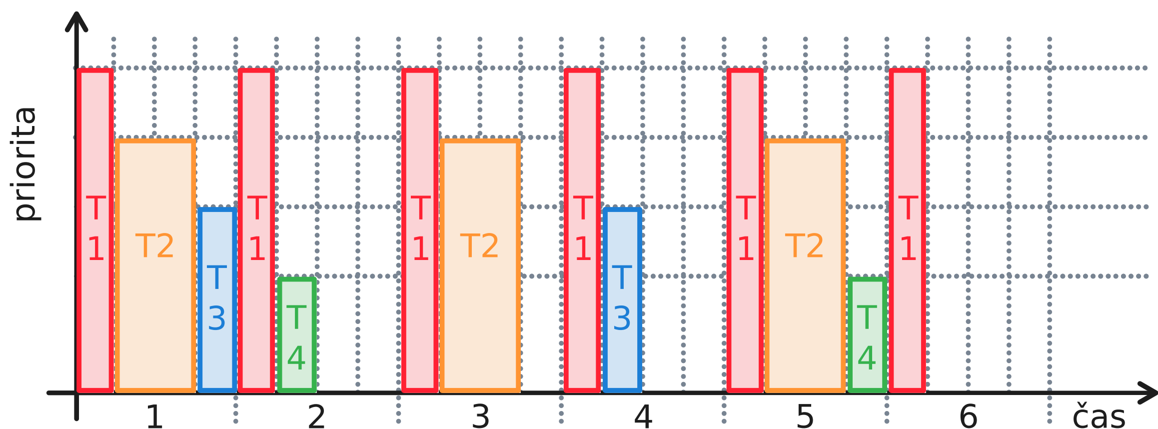
Plánování s konstantní šířkou pásma (*Constant Bandwidth Server, CBS*)
je plánování, u kterého dochází k dělení běhu na variabilní časové úseky, pro které je algoritmem zajištěno rozdělení úloh tak, že je procesor využíván rovnoměrně mezi všemi časovými úseky [1]. Algoritmus spoléhá na skutečnost, že úlohy definují délku svého běhu a tím je vhodný mj. pro periodická plánování. Jeho revidovaná verze [2] přidává podporu pro úlohy zablokované další stranou během jejich provádění, což plánování s konstantní šířkou pásma uzpůsobuje pro praktické použití v operačních systémech.

2.2.3 Synchronizace

Systémy s více úlohami implikují možnou existenci souběžného přístupu k prostředkům, u kterého může neřízený souběžný přístup vést na nespecifikované chování u těchto pro-



Obrázek 2.3: Časový diagram pro round-robin plánování: úloha T1 běží do té doby, než do stavu připravenosti v čase A přejde úloha T2. Ta běží do doby B, než vyčerpá přidělené časové kvantum q a následně je běh předán plánovačem úloze T3, které taktéž plně vyčerpá přidělené časové kvantum. Úloha T2 následně dokončí práci před vyčerpáním kvanta a tedy je řízení předáno T3, která běží dále bez ohledu na časové kvantum, vzhledem k tomu, že žádná úloha se stejnou nebo vyšší prioritou není připravena k běhu. To se stane v okamžiku E, kdy je úloha T3 přerušena úlohou T4 s vyšší prioritou, která následně běží dokud se nevzdá řízení.



Obrázek 2.4: Časový diagram pro frekvenčně monotónní plánování: úloha T1 obdrží nejvyšší prioritu vzhledem k nejvyšší požadované frekvenci plánování, analogicky poté nižší priorita pro úlohy T2, T3 a T4. Pro danou množinu úloh s definovanými frekvencemi běhů a jejich délek selže test naplánovatelnosti, což má za následek mj. nesplnění časového limitu pro úlohu T4 v časovém úseku 1.

středků. Mezi základní synchronizační primitiva zajišťující korektní souběžný přístup dvou úloh na zdroj patří:

Mutex
: *Mutually Exclusive* je primitivum nabízející rozhraní definující stavy zamčeno a odemčeno. V případě, že je zamčen, je vlastněn konkrétní úlohou a další úlohy musí počkat, než bude mutex uvolněn.

Semafor
: Semafor definuje na svém rozhraní nezáporný čítač, který je pro zamčení úlohou dekrementován. Úloha musí čekat, jestli čítač nelze dekrementovat.

2.2.4 Inverze priorit úloh

Existence synchronizačních primitiv pro souběžný přístup na sdílené zdroje v preemptivním systému vede k problému zvanému inverze priorit. Jedná se o situaci, u které uvažujeme dvě úlohy H a L s vysokou, resp. nízkou prioritou, a prostředek R. Problém nastane, když se úloha H pokusí uzamčít zámek přístupu na R potom, co jej uzamkla úloha L. Úloha H tedy čeká, než je zámek uvolněn, což má za následek běh úlohy L s nižší prioritou do uvolnění zámku. To i přestože úloha H s vyšší prioritou je připravena na běh. Ideální systém je tuto situaci schopen řešit okamžitým uvolněním sdíleného prostředku R po požadavku úlohy H o prostředek. I v ideálním systému ovšem může dojít k situaci tranzitivního blokování, kdy třetí úloha M se střední prioritou je připravena na běh během běhu úlohy L s vlastněným zámkem nad R. Preemptivně je pak úloha L přerušena a nahrazena úlohou M, přestože by dle priority měla běžet úloha H.

Inverze priorit může mít za následek zpoždění běhu úlohy s vyšší prioritou, lze ji tedy v případech RTOS ve stálé či měkké třídy za určitých podmínek ignorovat. Pro její řešení lze naopak aplikovat několik existujících přístupů [15, 36]:

Selektivní vypnutí přerušení
: Hlavní myšlenkou je selektivní vypnutí přerušení (a tedy preempce) v systému během kritické sekce přístupu na sdílený zdroj.

Dědičnost priority
: Po dobu, kdy úloha s nižší prioritou drží sdílený zdroj, který zároveň žádá úloha s vyšší prioritou, je priorita úlohy s nižší prioritou povýšena na úroveň úlohy s vyšší prioritou. Nižší priorita je úloze opět nastavena ve chvíli, kdy odevzdá sdílená zdroj, a úloha s vyšší s prioritou je tak schopna zdroj převzít.

Protokol s horní mezí (*Original priority ceiling protocol, OPCP*)
: Princip je založen na přiřazení efektivní priority úlohám, která je odvozena jako maximální priorita z úloh, od kterých konkrétní úloha zdědila priority [30]. Každý prostředek má následně stanovenou horní mez priority, která odpovídá nejvyšší prioritě úlohy, která k danému prostředku přistupuje. A konečně, úloha může uzamčít prostředek pouze v případě, pokud je její efektivní priorita vyšší než priorita kteréhokoliv zablokovaného prostředku v systému kromě těch, které již má samotná úloha uzamčené.

Protokol s bezprostřední horní mezí (*Immediate priority ceiling protocol, IPCP*)
Princip vychází z protokolu s horní mezí, avšak je zjednodušen [10]. Jediným rozdílem je odlišný výpočet dynamické priority pro úlohy. Tento rozdíl je odvozen pouze z prostředků, které má daná úloha zamčené. Výsledkem této úpravy je stav, kdy je úloha blokována pouze na začátku jejího běhu a následně již ne při něm. Ve chvíli samotného startu totiž budou všechny požadované prostředky zřejmě volné, jinak by nedošlo k samotnému spuštění úlohy. Důvodem je, že by některý z prostředků již vlastnila úloha se stejnou nebo vyšší prioritou.

2.3 FreeRTOS

FreeRTOS je multiplatformní RTOS pro vestavěná zařízení s oficiální podporou pro desítky různých architektur [5, 20]. 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 [17] je druhým nejpoužívanějším veřejně dostupným RTOS na trhu (po OS Linux pro vestavěná použití). Z technických vlastností:

Plánovač
Preemptivní jádro používá ve výchozím nastavení plánovač Round-robin, ovšem lze konfigurovat i pro prioritní či frekvenčně monotónní plánování.

Velikost
Jádro je naimplementováno ve třech modulech a na platformě obvykle zabírá 6–12 kB.

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.

Verifikované verze
Existuje několik příbuzných projektů RTOS, které jsou matematicky ověřené jako správné či mají komerční podporu.

Režim nízké spotřeby
Při odpovídající konfiguraci lze zapnout netikající (*tickless*) režim, kdy může být procesor zcela uspáván na dobu bez aktivní úlohy.

2.4 ChibiOS

ChibiOS je projekt 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 [37, 11]. Jeho vývoj začal v roce 2007, licencovaný je s GPL3 (s částí modulů proprietárně) a v dnešní době je udržován jedním hlavním vývojářem. Z technických vlastností ChibiOS/RT:

Plánovač
Preemptivní jádro používá plánování Round-robin, ovšem lze konfigurovat na prioritního plánování s FCFS při shodných prioritách.

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

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

2.5 RTEMS

RTEMS (*Real-Time Executive for Multiprocessor Systems*) je RTOS s otevřeným zdrojovým kódem, jehož vývoj započal již v osmdesátých letech 20. století [35, 28]. 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 frekvenčně monotónní plánování, až k algoritmu konstatní šířky pásma – pokud je to pro dané plánování validní požadavek, lze konfigurací vypnout preempci úloh v systému.

Hardware podpora
: Podporu nabízí od instrukční sady RISC-V, přes x86 až k ARM, ARM Thumb a AArch64, ovšem bez podpory pro Cortex-A53.

Vývoj aplikací
: Projekt má publikované obrazy virtuálních počítačů k vývoji nad sadou nástrojů RTEMS, taktéž je dostupná dokumentace pro běhu v simulátorech.

Ověřená verze
: Projekt nabízí certifikaci pro aplikace ve vesmíru při použití na specifických vývojových deskách[3].

2.6 RT-Thread

Projekt RT-Thread zastřešuje operační systém reálného času s otevřeným zdrojovým kódem, jehož vznik je datován v Číně do roku 2006 [40]. Je cílený především na ARM architekturu modelové řady procesorů Cortex, ovšem oficiální podpora pro Cortex A53 chybí, což významně ovlivňuje tuto práci.

Plánovač
: Kernel systému RT-Thread využívá plánovač založený na plně preemptivním přístupu k úlohám rozlišenými prioritou, při shodě priorit je pak na jejich úrovni plánováno dle konstantního kvanta procesorového času.

[3] https://rtems-qual.io.esa.int/

Nástrojová podpora
: Projekt z vývojářských nástrojů nabízí i vývojové prostředí *RT-Thread Studio* a konzolové utility pro konfiguraci systému.

Hardware podpora
: RT-Thread míří na široké spektrum architektur a procesorů od nízkoúrovňových až k vícejádrovým anebo vysokoúrovňovým.

Podpora pro aplikace
: Tvůrci deklarují podporu pro standardizované rozhraní POSIX, nejmenší verze kernelu zabírá 1,2 kB a systém taktéž podporuje režimy nízké spotřeba běhu.

2.7 µC/OS

Micro-Controller Operating Systems (též MicroC/OS či µC/OS) je operační systém z rodiny systémů reálného času vyvinutý v jazyce C s otevřeným zdrojovým kódem [41, 39]. Po první verzi z roku 1992 je aktuálně distribuován ve třetí verzi s posledním vydáním v roce 2016 – spravován společností Micrium míří tento OS na nízkoúrovňová vestavěná použití.

Plánovač
: µC/OS využívá Round-robin plánování s neomezeným počtem úloh a jejich priorit, každá z úloh má navíc implicitní frontu zpráv.

Synchronizace
: Systém nabízí vývojářům základní synchronizační primitiva: semafory a mutexy.

Paměť úloh
: Využito je fixní rozdělení paměti na bloky, z nichž každé úloze je staticky přiřazen celočíselné množství bloků.

Hardware podpora
: Vývojáři projektu deklarují více než 50 podporovaných architektur a vývojových desek. Vzhledem ke kontextu této práce je význačná podpora pro CPU Cortex řady A implementující standard architektury ARM-v8.

Za zmínění stojí projekt RealPi [16] implementující podporu pro Raspberry Pi v základních verzích Zero, 1, 2 a 3 – projekt se edukativně zabývá přípravou kompletní sady nástrojů pro hardwarovou podporu a zavedení operačního systému µC/OS na dotyčný mikropočítač.

2.8 OS Linux jako RTOS

Myšlenky za operačním systémem Linux, jakožto systémem pro obecné použití, nevedly při jeho vývoji primárně pro použití v časově kritických operací z pohledu operačních systémů reálného času. Přesto je cílem několika projektů docílit pro OS Linux chování přibližující se chování RTOS. Dva z těchto projektů budou v následujících kapitolách představeny.

2.8.1 Raspbian s PREEMPT_RT

Projekt PREEMPT_RT je patch[4] do operačního systému Linux, který z něj částečně dělá systém schopen preemptivní plánování s důrazem na běh v reálném čase – míra schopnosti je závislá na konkrétní použité konfiguraci [25, 34, 19]. Vzhledem k jeho rozšíření se jedná o *de facto* standard v oblasti využití systému Linux v časově kritických aplikacích. Princip a funkce úpravy PREEMPT_RT je založena na minimalizaci délky kritických sekcí v jádře systému, či případném vypnutí preempce pro specifické části, čímž je zajištěno dosažení požadované latence.

Nejrozšířenější distribucí systému Linux pro Raspberry Pi je projekt Raspbian jakožto derivát systému Debian. Vzhledem k tomu, že patch PREEMPT_RT je určen pouze a přímo pro jádro systému, je možné aplikovat úpravy přímo na zdrojové kódy projektu Raspbian. Nutná je shoda verzí, vzhledem k tomu, že patch je vydáván pouze pro každou desetinnovou verzi jádra s plánovanou dlouhodobou podporou. Takto upravené jádro operačního systému Linux s sebou nese řadu výzev – i přes snahu jeho vývojářů není v roce 2022 patch do jádra plně integrovaný a taktéž se nedá spoléhat na determinismus jádra po jeho aplikaci.

Hardware podpora

Samotný projekt Raspbian pokrývá všechny existující verze počítačů Raspberry Pi, jádro operačního systému Linux (potenciálně s patchem PREEMPT_RT) pak deklaruje podporu pro ARM architekturu od verze 8 pro 64bitové režimy procesorů, a od verze 7 v 32bitových režimech běhu – a to pouze v případech, že architektura poskytuje jednotku pro řízení paměti (*MMU, Memory Management Unit*). Přímou podporu pro tyto architektury pak deklarují distribuce Debian, Fedora, Ubuntu či Arch Linux. Procesory bez jednotky pro správu paměti pak podporuje vydání systému uClinux, pro kterou je patch taktéž aplikovatelný.

Plánovač

Originální OS Linux nabízí tři typy chování plánovače pro konkrétní úlohy:

`SHED_OTHER`
výchozí plánovač využívající rovnoměrné sdílení časových kvant,

`SCHED_BATCH`
založen na výchozí verzi, ale uzpůsoben pro aplikace kritické na procesorový čas, jeho standardní časové kvantum je 1,5 sekundy,

`SCHED_IDLE`
nastaví úloze nejnižší prioritu, tedy je spuštěna pouze v případě, že žádná jiná úloha s vyšší prioritou nežádá o procesorový čas.

Patch PREEMPT_RT pak následně přidává další tři typy chování plánovače pro úlohy:

`SCHED_FIFO`
implementuje chování *First Come, First Serve* v preemptivní verzi,

`SCHED_RR`
zavádí pro úlohy Round-robin plánování,

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

`SCHED_DEADLINE`
: implementuje revidovanou verzi plánování s konstantní šířkou pásma.

2.8.2 Projekt LITMUS^{RT}

Projekt LITMUS^{RT} [12, 8, 13] (*Linux Testbed for Multiprocessor Scheduling in Real-Time*) je experimentální rozšíření do operačního systému Linux zajišťující zlepšení latence jádra za účelem jeho použití v časově kritických aplikacích. Tento projekt se narozdíl od PREEMPT_RT zaměřuje na úpravu plánovače jádra systému, který zjednodušuje a zavádí rozhraní pro další rozšíření.

Poslední vydaná verze 2017.1 byla vydána v roce 2017 a podporuje jádro OS linux ve verzi 4.9.30 – vzhledem k experimentální povaze projektu sloužícího především jako *Proof of concept* není v plánu vývojářů jej přibližovat k trvalému připojení do hlavní vývojové verze jádra systému Linux.

Kapitola 3

Raspberry Pi 3B+

Následující kapitola představuje zařízení Raspberry Pi+ jakožto mikropočítač v kontextu této práce – kapitola 3.1 stručně shrnuje jeho historické pozadí, následně kapitola 3.2 popisuje dostupné prostředky na desce mikropočítače a závěrečná kapitola 3.3 princip zavádění systému po startu.

3.1 Základní informace o zařízení

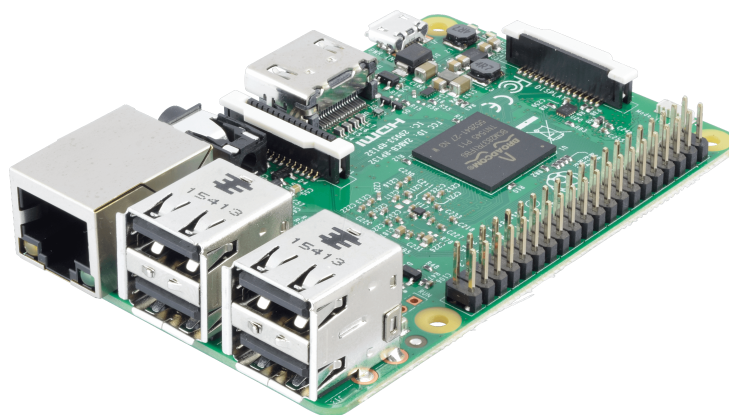
Raspberry Pi je malý nízkonákladový jednodeskový počítač mířící do širokého spektra použití, od projektů využívajících lokální vstupně-výstupní sběrnice, přes multimediální centra, až k masovým použitím ve výpočetních clusterech. Jeho celosvětový úspěch spočívá v realizaci velkého počtu portů a sběrnic na malém prostoru při zachování relativní nízké ceny – organizace Raspberry Pi Fundation oznámila ve výroční zprávě k roku 2020 celosvětově prodaných přes 37 miliónů mikropočítačů Raspberry Pi [31].

Od prvního vydání modelu Raspberry Pi Model B v roce 2012 vydala společnost další tři vývojové generace plnohodnotných desek: 2, 3 a 4 – k těm byly v posledních letech připojena generace Zero s redukovanou velikostí a hardwarovou výbavou, a Pico jakožto deska odlišné architektury s jedním mikrokontrolérem. Dále se práce bude věnovat tomuto mikropočítači v třetí generaci ve verzi Raspberry Pi 3B+ z roku 2018 vyobrazeném na obrázku 3.1. Jedná se totiž o poslední generaci, u které není dlouhodobý produkční nedostatek, který nastal v poslední době u čtvrté generace.

3.2 Dostupný hardware

Středobodem mikropočítače Raspberry Pi 3B+ je SoC[1] BCM28370 od společnosti Broadcom představený v roce 2016 založený na 40nm výrobní technologii. Je to SoC v evoluční řadě obsahující čtyři jádra procesoru Cortex-A53 vydaného v roce 2012 společností ARM Holdings – historicky se jedná o první procesor implementující instrukční sadu ARMv8-A, jeho součástí je taktéž jednotka pro vektorové zpracování čísel s plovoucí desetinnou čárkou.

[1] *System on Chip* (*SoC*) je integrovaný obvod obsahující veškeré periferie zabudované přímo v čipu. Tento princip se používá ve vestavěných systémech mj. díky jeho nízké spotřebě.



Obrázek 3.1: Počítač Raspberry Pi 3B+, převzato od společnosti Reichelt: https://www.reichelt.com/magazin/en/wp-content/uploads/2017/07/RASP_03_01.png

SoC taktéž obsahuje grafický procesor VideoCore 4 od společnosti VideoCore zajišťující pro mikropočítač Raspberry Pi vykreslování grafického výstupu. Nutné je poznamenat, že mikropočítač neobsahuje modul pro hodiny reálného času – vnitřní časování je tedy řízeno pouze vestavěným krystalem a mikropočítač si při vypnutí nedrží stav aktuálního času.

Následuje výběr z dostupného hardware na mikropočítači, jehož kompletní blokové schéma je umístěno v obrázku 3.2:

BCM2837B0
SoC jednotka integrující 64bitový procesor Cortex-A53 s provozní frekvencí 1,4 GHz

1 GB DDR2 RAM
nerozšířitelná operační paměť zabudovaná přímo na SoC

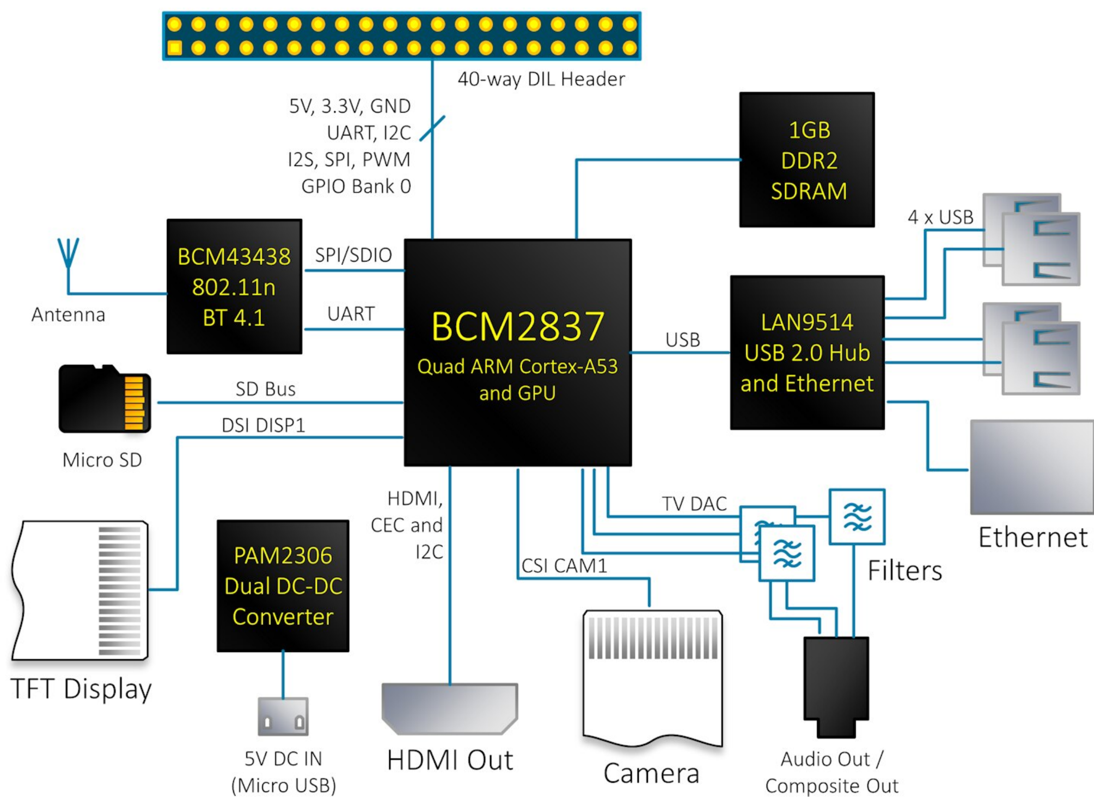
802.11.b/g/n/ac
SoC taktéž obsahuje modul pro podporu WiFi v uvedených standardech

Bluetooth 4.2 a Low Energy
zajištěno společně s WiFi modulem Cypress CYW43455

Ethernet
použit ve standardu 1 Gbit/s, efektivně lze komunikovat pouze na úrovni 300 MBit/s vzhledem k tomu, že modul je připojen k čipu přes USB ve verzi 2.0

HDMI
plnohodnotný konektor HDMI nabízí Full HD rozlišení pro 30 sn/s

GPIO rozhraní
počítač nabízí rozhraní s 40 piny, z nichž každý slouží různému účelu – od statických napěťového charakteru, přes volně řiditelného pomocí software, k pinům sloužícím pro rozrahní UART, SPI či I^2C



Obrázek 3.2: Blokové schéma počítače Raspberry Pi 3B+, převzato z fóra Element 14: https://community.element14.com/products/raspberry-pi/b/blog/posts/raspberry-pi-3-block-diagram

3.3 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 použity ve výchozích verzích a konfiguracích.

Kromě význačného souboru `kernel.img` se systémem a textového konfiguračního souboru `config.txt` jsou soubory zmíněné v této kapitole poskytnuty výrobcem výhradně v binární formě – vzhledem k jejich povaze a důležitosti je v zájmu výrobce uchovat průmyslové tajemství. Pro firmware grafického jádra přesto vzniklo několik komunitních projektů suplementující proprietární software otevřenou alternativou[2].

3.3.1 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[3] – hledaným souborem 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[4] požadavek na soubory `bootcode.bin` a `bootsig.bin`. První z nich z nich obsahuje zavaděč druhé úrovně, druhý je očekávaně dle specifikace nenalezen – očekávané selhání při požadavku na soubor `bootsig.bin` je popsáno v dokumentaci[5].

3.3.2 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 specifické dle aktivní režim procesoru aktivuje jádro ARM procesoru

[2] Komunitní projekty suplementující proprietární firmware grafického jádra: `https://github.com/itszor/vc4-toolchain`, `https://github.com/christinaa/rpi-open-firmware` nebo `https://github.com/christinaa/LLVM-VideoCore4`.

[3] Sběrnice *Multi Media Card* (*MMC*) je zodpovědná za připojení a komunikaci s paměťovým médiem, kterým je v případě Raspberry Pi SD karta.

[4] Trivial File Transfer Protocol (TFTP) je silně zjednodušená verze protokolu FTP určená pro specifická použití, kdy by byl protokol FTP příliš komplexní – v tomto případě použit při zavádění systému po počítačové síti.

[5] `https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#network-booting`

– režim je stanoven mj.[6] konfiguračním parametrem arm_64bit ze souboru config.txt. V případě, že nabývá hodnoty 1, startuje procesor v 64bitovém režimu a konfigurační parametr kernel_address udávající adresu paměti, od které je procesor při startu spuštěn, nabývá hodnoty 0x200000. Pro 32bitový režim je nutné příznak nastavit na hodnotu 0 a procesor v tu chvíli startuje od adresy 0x8000 – což je případ této práce, která využívá procesor ve 32bitovém módu.

[6]Procesor taktéž nastartuje v 64bitovém módu, jestliže je na médiu přítomen soubor kernel8.img a není přepsán konfigurační příznak kernel.

Kapitola 4

Demonstrační aplikace A

V této kapitole bude popsána tvorba vývojového prostředí a demonstrační aplikace s RTOS implementující tři oddělené úlohy – dva čítače po 10 Hz a 50 Hz, a také monitorovací úlohu odesílající 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 [17, 3], licenční politiku, veřejný zdrojový kód, aktivní vývoj (je vydáván na měsíční bázi) a širokou komunitní podporu skrz oficiální fórum. Očekávaným výsledkem práce na návrhu a řešení v této kapitole je připravené prostředí pro aplikační vývoj pro počítač Raspberry Pi 3B+ a samotná demonstrační aplikace reprezentující funkci operačního systému reálného času FreeRTOS na základě tří úloh, z nichž dvě zapisují na sdílené čítače a třetí reportuje jejich stav na sériovou linku. Očekávané je v prostředí systému též preempční chování, kdy samotné demonstrační úlohy nejsou na preemci nějak specificky připraveny.

Dále je práce rozdělena do kapitol dle jednotlivých kroků vývoje – prvně je v kapitole 4.1 popsáno instruování překladače a linkeru, následně zajištění podpory a konfigurace pro systém FreeRTOS v kapitole 4.2, rozebrán je též přístup ke knihovnám jazyka C v kapitole 4.3 a sekce je zakončena trojicí kapitol 4.5, 4.5 a 4.6 popisující tvorbu samotné aplikační části, její spuštění a na závěr její vyhodnocení.

4.1 Nastavení překladače a linkeru

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 posloupnosti.

Aplikace bude vzhledem k cílové platformě sestavena nad sadou nástrojů `arm-none-eabi` – sada pro architekturu `aarch32` implementuje ARM EABI[1]. Základní stavební kámen je soubor pravidel `Makefile` odpovídá typickému použití pro projekty v jazyce C. Kompilátor `arm-none-eabi-gcc` pro části projektu v jazyce C je instruován s následujícími argumenty:

[1]EABI (*Embendedd Application Binary Interface*) je standard definující binární rozhraní programů pro vzájemnou kooperaci – v tomto případě při použití ve vestavěných systémech.

-mcpu=cortex-a53+crypto
: Nastavuje cílový procesor (a tedy i architekturu) – překladač `gcc` je schopen optimalizovat přímo na cílový procesor, jinak by postačovalo i použití obecného `-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 (*Vector Floating Point*) jednotku ve specifikaci `VFPv4-D16`, je vhodné ji tedy použít. Konkrétně je vhodné ji použít 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 [4], pro práci s jednotkou je díky `-mfloat-abi=hard` kompilátor instruován k jejímu přímému použití (bez emulace případnými instrukcemi pro procesor) při aritmetice s plovoucí řadovou čárkou.

-ffreestanding
: Instruuje kompilátor 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 automaticky nalinkována standardní knihovna `libc`, ani běhová.

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

Tabulka 4.1: Mapa paměti pro linker, podle které budou umístěny jednotlivé sekce programu do výsledného birnáního souboru – v projektu je její definice uložena v souboru `raspberry.ld`.

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 a následně volá plnohodnotnou funkci jazyka C `start`, která se postará o další konfiguraci procesoru.

```c
// symbol defined by linker table
extern uint32_t __stack_end;

// put it in binary section .init (defined by linker table)
// naked since it's not called by HW, just jumped into
void __attribute__((section(".init"), naked)) _start() {
    // set SP into stack space
    set_sp((uint32_t) (__stack_end));
    // call the true entrypoint
    start();
    // park CPU into endless loop
    while (1) {};
}
```

Zdrojový kód 4.1: Úvodní rutina volaná jádrem procesoru, která není funkcí z pohledu jazyka C – proto pouze nastaví ukazatel na zásobník na odpovídající místo v paměti a předá řízení do inicializační funkce `start`.

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 režim jádra supervizor – to zajistí pomocí nastavení odpovídajících bitů do registru `CPSR` (*Current Processor State Register*). Nutno je taktéž zabezpečit, že se procesor (např. po resetu) nenachází 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 rutin 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. Vzhledem k tomu, že ukazatel na vrchol zásobníku je jeden z tzv. bankovaných registrů[2], je na místě pro jednotlivé exekuční režimy procesoru nastavit jeho hodnoty. Funkce `setup_stacks` pro každý možný mód nastaví adresu samostatného prostoru pro zásobník.
7. A konečně, prostředí je nakonfigurováno dostatečně tak, že lze spustit samotnou aplikaci – pomocí vstupní funkce `void main(void)`.

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

4.2 Podpora pro FreeRTOS

Tato kapitola popisuje zavedení systému FreeRTOS do vytvořeného prostředí – ten bude zaveden ve verzi `202111.00` vydané v listopadu 2021. Pro jeho běh na zařízení je důležitých několik částí, které budou postupně popsány v následujících kapitolách.

4.2.1 Konfigurace frameworku

Soubor `FreeRTOSConfig.h` definuje základní konfiguraci pro demonstrační aplikace, následuje výběr symbolů z něj:

`configCPU_CLOCK_HZ = 100000000UL`
: základní frekvence procesoru, tedy 100 MHz

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

`configUSE_PREEMPTION = 1`
: parametr, který aktivuje preemptivní mód, tedy přepínání kontextu bez přímé spolupráce od právě běžící úlohy

`configUSE_TIME_SLICING = 0`
: deaktivuje přepínání úloh stejné priority na základě vyčerpání časového kvanta (tedy v praxi zapne prioritní plánování pro plánovač namísto Round-robin plánovaní)

`configUSE_IDLE_HOOK = 0`
: deaktivuje použití funkce `void vApplicationIdleHook(void)`, která by byla volána periodicky ve chvílích, kdy neexistuje jiná úloha připravená k běhu a plánovač tedy nemá v tu chvíli spustitelnou úlohu

`configUSE_TICK_HOOK = 0`
: deaktivuje použití funkce `void vApplicationTickHook(void)`, která by byla volána periodicky se samotným tikem plánovače

`configUSE_TICKLESS_IDLE = 0`
: deaktivuje použití módu, při kterém je procesor uspán do režimu nízké spotřeby během doby, kdy pro procesor neexistuje úloha a je tak nečinný. Toto chování nastává na základě očekávané doby spánku a probíhá pouze na platformách, které nabízí podporu pro režim nízké spotřeby.

4.2.2 Implementace pro platformní podporu jádra OS

Pro podporu je nutné naimplementovat a definovat sadu funkcí a konstant sloužících frameworku pro práci s hardware systému. Následuje výčet těch nejdůležitějších:

`void prvSetupTimerInterrupt(void)`
: Funkce, která je 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.

`portENABLE_INTERRUPTS, portDISABLE_INTERRUPTS`
Dvojice maker, kterou používá systém FreeRTOS pro zapnutí, resp. vypnutí přerušení `IRQ`[3] a `FIQ`[4] (typicky při kritických sekcích nebo práci jádra OS). Implementovány jsou pomocí modifikace řídícího registru `CPSR`.

`vPortHandleISR`
Důležitá rutina, která zpracovává příchozí přerušení, a jejíž adresa je nastavená v tabulce vektorů přerušení. Rutina implementuje přepnutí kontextu, tedy ten aktuální 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 systém FreeRTOS ukazatel na vrchol zásobníku aktivní úlohy – ten je při uložení kontextu uložen do této proměnné, při obnovení naopak z ní získán a obnoven.

`vPortYieldProcessor`
Jedná se o rutinu 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, portEXIT_CRITICAL`
Dvojice maker, která implementuje vstup do kritické sekce a výstup z ní. Využívají odpovídající makra pro zapnutí a vypnutí přerušení, a navíc pomocí čítače implementují rekurzivní zanořování kritických sekcí.

4.2.3 Implementace ovladačů periférií

Pro vývoj základních aplikací je záhodno implementovat i práci s periferiemi – pro základní vývojářský komfort postačí 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.

[3] *Interrupt Request* (*IRQ*) je požadavek na přerušení s normální prioritou.
[4] *Fast Interrupt Request* (*FIQ*) je požadavek na přerušení s vyšší prioritou uzpůsobený pro zpracování událostí s co nejnižší latencí.

Ovladač pro Mailbox

Díky požadavkům ve formě zpráv na GPU lze mj. ovládat druhou signalizační LED diodu[5], pro účely vývoje je tedy implementován pouze tento typ zpráv.

4.3 Běhová a standardní knihovna jazyka C

Vzhledem ke kompilaci bez přiložení běhové i standardní knihovny jazyka C je v případě, kdy chceme funkce z těchto knihoven používat, nutné suplovat jejich implementaci.

Běhová knihovna (*runtime library*) zajišťuje pro standardní knihovnu rozhraní, které je obecné bez závislosti na konkrétní použité architektuře a její instrukční sadě – standardizuje přístup k procesoru, dodatečně programově implementuje instrukce, které daná architektura (a tedy instrukční sada) nenabízí přímo a zajišťuje jednotné rozhraní pro software na vyšších úrovních. Jako běhová knihovna v této práci bude použita knihovna z projektu Compiler-RT[6], který implementuje běhovou knihovnu pro architekturu ARMv8-32. Pravidla pro sestavení knihovny jsou též součástí projektu a po jejím sestavení vznikne soubor `librt.a` s běhovou knihovnou, která bude staticky připojena k výslednému programu.

Standardní knihovna jazyka C (*standard library*) bude využita z projektu Newlib[7], který poskytuje standardní knihovnu v implementaci funkční na vestavěných systémech, respektive na architektuře s instrukční sadou ARMv8-32. Projekt Newlib nabízí implementaci ve formě zdrojových kódů, je ji tedy nutné zkompilovat pro konkrétní architekturu – výsledkem jsou soubory standardní knihovny `lib.c` a knihovny matematických funkcí `lib.a` určené pro statické sestavení s výsledným programem. Vedlejším produktem překladu standardní knihovny jsou též odpovídající hlavičkové soubory určené pro proces překladu celé aplikace využívající standardní knihovnu.

K takto poskytnuté standardní knihovně je nutno doplnit implementaci pro systémová volání konkrétních standardních funkcí, které bude aplikace používat. Pro vývoj na vestavěné platformě bude implementována práce s dynamickou pamětí a standardním výstupem – obě v modulu `syscalls.c`, který je přímo linkován do výsledného programu.

Dynamická paměť

V programovacím 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 (dle rozdělení popsáného v tabulce 4.1).

Standardní výstup

Volání standardních funkcí z rodiny `printf` vedou mj. 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

[5] 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.

[6] Compiler-RT je projekt obsahující mj. implementaci *runtime* knihovny pro jazyk C dostupný na https://compiler-rt.llvm.org/.

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

vyprázdněn buffer UART rozhraní pomocí `uart_flush`. Součástí musí být též implementace pro systémová volání `_close`, `_fstat`, `_isatty` a `_lseek` – jejich implementace na systémech bez podpory pro souborové systémy nedává smysl, jsou tedy prázdné a návratovou hodnotou je výchozí hodnota odpovídající jejich návratovému typu.

4.4 Tvorba samotné aplikace

Demonstrace funkce bude provedena pomocí tří periodických úloh z definice zadání 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 implementací úloh je nekončená smyčka, před kterou se nachází inicializační část. Frekvence spouštění úloh je odvozena od hodnoty symbolu `portTICK_PERIOD_MS` – to ze systému FreeRTOS definuje počet tiků operačního systému za jednu milisekundu běhu procesoru.

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 úloh v systému FreeRTOS mají generický typ `void *`). Ukázka 4.2 zobrazuje implementaci 50Hz čítače.

Monitorovací úloha zobrazená v ukázce 4.3 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).

```c
void task50Hz(void* pParam) {
    // recast to proper data type
    unsigned long* counter = (unsigned long*) pParam;
    // computed desired period in ticks
    const TickType_t tickFreq = (1000 / 50) / portTICK_PERIOD_MS;
    // reset the counter
    *counter = 0;

    TickType_t lastWakeTime = xTaskGetTickCount();
    while (1) {
        // use critical section during accessing the shared memory
        taskENTER_CRITICAL();
        (*counter)++;
        taskEXIT_CRITICAL();

        // go to sleep to maintain the wanted frequency
        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.

```c
void taskPrinter(void* pParam) {
    unsigned long* counters = pParam;

    // once per 3 seconds
    const TickType_t tickFreq = 3000 / portTICK_PERIOD_MS;
    TickType_t lastWakeTime;

    lastWakeTime = xTaskGetTickCount();
    while (1) {
        taskENTER_CRITICAL();
        // report states of both counters
        printf(
            "%10lu;%10lu;%10lu\n",
            lastWakeTime, counters[0], counters[1]
        );
        taskEXIT_CRITICAL();

        // go sleep to maintain frequency
        xTaskDelayUntil(&lastWakeTime, tickFreq);
    }
}
```

Zdrojový kód 4.3: Kód úlohy reportující stav čítačů. V inicializační sekci úlohy je pro použití upraven typ příchozího generické parametru na jeho reálný typ ukazatele na numerické čítače a následně ve smyčce co tři vteřiny reportován stav na sériovou linku – přímé použití `printf` pro zápis na sériovou linku je možný díky implementaci popsané v 4.3. Zajištění požadované frekvence zařizuje plánovač s pomocí uspávání úlohy.

```c
void main(void) {
    // start uard with baudrate 115200 bd/s
    uart_init(115200);
    // local stack used as shared memory,
    // access MUST BE mutually exclusive
    unsigned long counters[2];
    // disabled buffering on stdout
    setvbuf(stdout, NULL, _IONBF, 0);

    // counter tasks
    xTaskCreate(
        task10Hz,    "task10Hz",    256, counters + 0,
        tskIDLE_PRIORITY + 1, NULL
    );
    xTaskCreate(
        task50Hz,    "task50Hz",    256, counters + 1,
        tskIDLE_PRIORITY + 1, NULL
    );

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

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

Zdrojový kód 4.4: Vstupní funkce aplikace `main`. Ta ve svém těle nejprve deklaruje paměťový prostor pro interní čítače, následně naplánuje spuštění tří úloh pro plánovač a ten poté spustí. Přestože se neočekává, že funkce `vTaskStartScheduler` skončí, je program zakončen nekonečnou smyčkou pro tzv. zaparkování procesoru[8].

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 ze systému FreeRTOS `xTaskCreate` s následujícími parametry:

`TaskFunction_t pvTaskCode` ukazatel na kód úlohy

`const char * const pcName` název úlohy pro vývojáře, samotný název není plánovačem nikde efektivně použit

`configSTACK_DEPTH_TYPE usStackDepth` velikost zásobníku pro úlohu

`void *pvParameters` ukazatel na generický ukazatel na data pro úlohu

[8]Zaparkováním procesoru se rozumí spuštění nekonečné smyčky, v jejímž těle nedochází k provádění žádné efektivní práce, což chrání procesor před během kódu z paměťového prostoru, který pro uložení programu není určen a nachází se za částí paměti s programem.

`UBaseType_t uxPriority` číselná priorita úlohy

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

Z ukázky 4.4 je tedy zřejmé, že úlohy k čítačům přistupují na úroveň lokálního paměťového prostoru funkce `main`. Priority úloh jsou číselně stanoveny nad úroveň bázové priority z FreeRTOS.

4.5 Sestavení a spuštění aplikace

Pro běh demonstrační úlohy je nutné ji po zkompilování a sestavení umístit vhodně na zaváděcí médium. Tím bude v tomto případě paměťová karta, postup systavení a zavedení aplikace je následující:

Překlad programu
Projekt obsahuje soubor s pravidly pro nástroj `make`, který ve svém hlavním cíli `all` provede s pomocí překladače `arm-none-eabi-gcc` překlad zdrojových kódů systému FreeRTOS a demonstrační aplikace na objektové soubory. Na objektové soubory jsou též přeloženy zdrojové kódy z jazyka symbolických instrukcí. Všechny objektové soubory jsou poté pomocí linkeru `arm-none-eabi-ld` sestaveny do jednoho výsledného spustitelného souboru, do kterého jsou linkerem přidány i ELF hlavičky.

Extrakce instrukcí
Ze sestaveného binárního souboru jsou pomocí nástroje `arm-none-eabi-objcopy` vyextrahovány pouze část bez hlavičky (ta v případě spouštění na přímo na procesoru nedává smysl, program je zavedený přímo do paměti bez dodatečného mezikroku od operačního systému).

Ladící informace
Z výsledného přeloženého programu je pomocí utility `arm-none-eabi-readelf` zaznamenána vnitřní struktura programu – jedná se o seznam přítomných symbolů a další ladící detaily o výsledném programu. S pomocí utility `arm-none-eabi-objdump` je taktéž vyextrahován samotný obsah binárního výsledku v detailu na jednotlivé instrukce i s jejich interpretací do jazyka symbolických instrukcí.

V tuto chvíli jsou tedy vyprodukovány následující soubory:

`kernel.elf` – prvotní přeložený a sestavený program celé aplikace

`kernel.img` – pouze přeložená aplikace jakožto bez hlavičky ELF, který bude dále použit pro tvorbu obrazu

`kernel.map` – soubor s ladícími informacemi popisující datovou strukturu vytvořeného programu, který pro každý symbol popisuje, odkud pochází

`kernel.dmp` – ladící informace k jednotlivých datovým sekcím ve výsledném programu

`kernel.list` – ladící informace popisující obsah jednotlivých symbolů i s jeho interpretací zpět do jazyka symbolických instrukcí

Příprava obrazu karty

Pro kartu je nejprve připravena tabulka oddílů s jedním oddílem souborového systému typu FAT o velikosti 15 MiB, což je cíleně naddimenzovaná hodnota oproti reálné velikosti přeložené aplikace. Aplikační oddíl je posunutý od začátku obrazu, aby vytvořil prostor pro samotnou tabulku. Následně jsou na oddíl zapsány soubory nutné pro zavedení systému na Raspberry Pi: `bootcode.bin`, `start.elf` a `fixup.dat`, jejichž důležitost byla zmíněna v kapitole 3.3. Posledním souborem přímo zapsaným na oddíl je `config.txt` konfigurující zavedení systému a názvy souborů, podle kterých hledá zavaděč jednotlivé úrovně. Na závěr je na oddíl připraven příznačný soubor `kernel.img` obsahující samotný uživatelský program z předchozího kroku.

Zápis obrazu karty

Na připojené paměťové médium je vytvořený obraz karty zapsán binárně přesně.

Spuštění aplikace

Samotné spuštění nevyžaduje na počítači Raspberry Pi žádné další specifické kroky kromě zavedení média s firmware a vynucení resetu zařízení.

4.6 Vyhodnocení aplikace

Po spuštění počítače Raspberry Pi 3B+ se zavedeným médiem s připravenou aplikací dojde bezprostředně k jeho spouštění, spuštění zavaděčů a následně i samotného systému (jak popisuje kapitola 3.3). Ten, jak bylo ukázáno v 4.4, inicializuje komunikační rozhraní a úlohy, a následně spustí interní plánovač z FreeRTOS. Díky komunikačnímu rozhraní UART je pomocí tiskové úlohy reportován stav po sériové lince, na jejíž druhé straně lze běh úloh pozorovat – zkrácený výstup z linky lze pozorovat v ukázce 4.5.

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

Zdrojový kód 4.5: 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 na sériovou linku.

Měření na druhé straně linky probíhalo pomocí skriptu v jazyce Python, který se na sériovou linku připojil a následně měřil zpoždění oproti očekávané periodě – skript je zobrazen v ukázce 4.6, jeho výstup poté v ukázce 4.7.

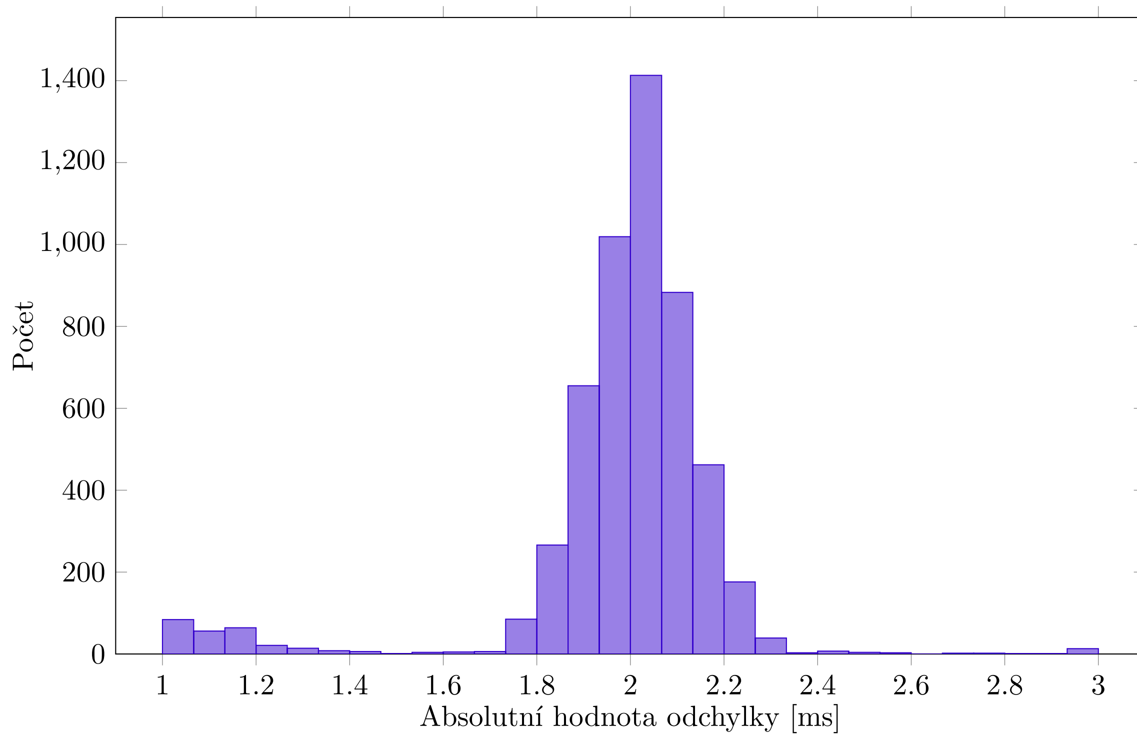
Pro měřenou vlastnost absolutního zpoždění oproti předpokládané periodě X nad změřenými $N = 5303$ vzorky lze určit střední dobu zpoždění $E(X) = 1,9813$ ms i 95. percentil zpoždění, u kterého platí $Q_{0.95} = 2,1962$ – tedy že 95 % vzorků dosahuje zpoždění nižšího než 2,1962 ms. Histogram vzorků s pro měření lze vidět v obrázku 4.1. Vzhledem k požadované periodě 3000 ms lze konstatovat, že se relativní chyba pohybuje na úrovni jednotek setin procent.

```python
import serial
import sys
from time import perf_counter

PORT = sys.argv[1]
EXPECTED_PERIOD_MS = 3
BAUDRATE = 115200

last = None
# connect the serial port based on port, baudrate and timeout
with serial.Serial(PORT, BAUDRATE, timeout=EXPECTED_PERIOD_MS * 2) as s:
    while True:
        line = s.readline().decode().strip()
        # compute the delay versus last value
        if not last:
            last = perf_counter()
            continue

        now = perf_counter()
        real_period = now - last
        jitter_ms = abs((EXPECTED_PERIOD_MS - real_period) * 1000)
        last = now

        print(f'{jitter_ms};{line}')
```

Zdrojový kód 4.6: Měřící skript v jazyce Python: po připojení na sériovou linku dle zadaného portu je ve smyčce příjmána hodnota z linky a měřena je nežádoucí odchylka (*jitter*) oproti očekávané periodě.

```
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 4.7: Měření z monitorovací úlohy na druhém konci sériové linky – reportován je absolutní hodnota jitteru s milisekundách oproti očekávané periodě 3 sekund a samotný výstup z aplikačních čítačů včetně vnitřního čítače tiků systému FreeRTOS.



Obrázek 4.1: Histogram změřeného zpoždění oproti požadované periodě reportování.

Kapitola 5

Demonstrační aplikace B

V této kapitole bude popsána tvorba druhé demonstrační aplikace nad sběrnicí *CAN bus*, pomocí které bude úloha reportovat stav vnitřních čítačů. Kapitola nejprve uvádí problematiku sdílení dat více procesy v kapitole 5.1 a jejího řešení v systému FreeRTOS kapitole 5.2. Následně je rozebrán princip sběrnice CAN a jejích rámců v kapitole 5.3, rozhraní a použití ovladače pro tuto sběrnici v kontextu mikropočítače Raspberry Pi 3B+ v kapitole 5.4 a na závěr použití sběrnice pro realizaci demonstrační aplikace v kapitole 5.5.

5.1 Bezpečné sdílení dat mezi úlohami

Za bezpečné sdílení dat mezi procesy lze označit takovou práci se sdílenými daty v programu, při které je zaručeno, že jednotlivé procesy nepřistoupí na hodnotu dat v nežádaném stavu – přístup na sdílený prostředek, kterým jsou v tomto případě konkrétní sdílená data, je tedy řízen synchronizačním primitivem.

5.2 Realizace bezpečného sdílení dat v FreeRTOS

Pokud to podporuje samotná platforma, systém FreeRTOS nabízí podporu pro kritické části. Jak bylo zmíněno a implementováno v kapitole 4.2.2, platformní implementace nabízí vývojáři makra `portENABLE_INTERRUPTS` a `portDISABLE_INTERRUPTS` pracující s obsluhami přerušení.

Systém pak definuje symboly, které jsou veřejným rozhraním pro použití kritické sekce – jsou nimi makra `portENTER_CRITICAL` a `portEXIT_CRITICAL`. Při jejich použití jako globálního zámku při přístupu na sdílené zdroje je lze považovat za způsob bezpečného sdílení dat – ukázka 5.1 prezentuje funkci pracující takto s kritickou sekcí.

```
extern void * shared_data;

void function() {
    // code possibly preempted by kernel

    portENTER_CRITICAL();

    // in critical section, the resource is locked
    // code section cannot be preempted
    *shared_data *= 42;

    portEXIT_CRITICAL();

    // code possibly preempted by kernel
}
```

Zdrojový kód 5.1: Funkce využívající bezpečného sdílení dat – proměnná `shared_data` je ukazatel na sdílená data více úlohami a tedy přístup na ni musí být synchronizován. To je zajištěno dvojicí maker ze systému FreeRTOS vytvářející kritickou sekci, během které nemůže být úloha preempčně přerušena.

5.3 Sběrnice CAN

Controller Area Network (*CAN*) je aplikační sběrnice z rodiny standardu IEC 61158 o průmyslových sběrnicích, která bude v popsána ve verzi `CAN2.0A` [22, 7, 21]. Využívá tři úrovně, z nichž nejnižší fyzická implementuje fyzickou vrstvu modelu ISO/OSI, a horní objektová a přenosová implementují linkovou vrstvu téhož modelu. Fyzická vrstva definuje přenosové médium, kterým je typicky dvojice diferenciálních vodičů či optická linka. Díky návrhu zajištuje komunikaci v reálném čase s důrazem na vysokou úroveň zabezpečení. Data organizuje v rámcích, z nichž každý obsahuje identifikátor a až 8 datových bajtů – identifikátor specifikuje obsah a prioritu přenášené zprávy.

Samotný přenos dat je definován nad recesivními a dominantními bity – fyzická vrstva definuje implementaci dominantního bitu vzhledem k logickým úrovním a hodnotám bitů. Protokol rozlišuje čtyři typy rámců – datové, chybové, rámce vzdáleného volání (též rozšířené či *extended* rámce) a rámce signalizující přetížení. Specifikem sběrnice CAN je absence cílové adresy u zpráv, z čehož plyne výhoda možnosti příjmu jedné zprávy na více koncových zařízení.

Standard sběrnice CAN, vzhledem k tomu, že se jedná o asynchroní sběrnici, definuje povinný princip prokládání bitů (*bit stuffing*). V případě, že by na sběrnici bylo odesláno pět po sobě jdoucích bitů stejné hodnoty, je bitový proud proložen jedním bitem opačné hodnoty. Tomuto chování podléhají všechny části rámců s dynamickými hodnotami – vyloučeny jsou tedy části s potvrzovacími 2 bity, ukončujícími bity cyklického kontrolního kódu a přízakem konce rámce.

Obrázek 5.1: Obecná struktura datového a rozšířeného rámce sběrnice CAN: obsahuje začátek rámce S, část arbitrážní pro stanovení odesílatele a smyslu zprávy ARB, řídící část popisující délku dat a případný příznak rozšíření CTRL, datovou část značenou DATA, kontrolní cyklický kód v části CRC, potvrzovací bity pro stav přijetí ACK, signalizovaný konec rámce E a nakonec mezirámcový prostor IFS.

5.3.1 Struktura rámců

Datový rámec je základním stavebním kamenem komunikace po sběrnici CAN, jehož hrubá struktura je zobrazena v diagramu 5.1, dále následuje podrobný seznam jeho součástí:

Start rámce
je zahájen dominantním bitem, který slouží k synchronizaci stanic.

Identifikátor
je 11bitový identifikátor určující obsah a prioritu zprávy.

Bit požadavku na data
Remote Transmission Request (*RTR*) značí při dominantní hodnotě aktivní požadavek odesílatele zprávy na zpětné zaslání dat – konkrétní význam definuje identifikátor zprávy.

Příznak rozšíření
značí, zda je rámec přenášen s či bez rozšíření.

Rezervované 2 bity
jsou vyhrazeny pro budoucí použití přenášené jako dominantní.

Délka datové části
na 4 bitech označuje délku následující datové části v bytech.

Datová část
obsahuje specifikované množství užitečných dat.

Cyklický kontrolní kód
je držen v 16 bitech jako kontrola datové části.

Potvrzovací 2 bity
jsou odesílány jako recesivní a příjmač má možnost nastavit první z nich na dominantní, jestliže detekoval chybu přenosu.

Konec rámce
je označen 7bitovou hodnotu.

Mezirámcový prostor
je opět 7bitový prostor poskytující zařízením čas zpracovat ukončenou zprávu.

Obrázek 5.2: Obecná struktura chybového rámce a rámce signalizujícího přetížení: první část FLAG značí typ rámce a chyby, druhá část DEL označuje jeho konec.

5.3.2 Struktura rozšířeného rámce

Struktura rozšířeného rámce se od standardního datového rámce liší v arbitrážní části – to konkrétně od RTR až k prvním rezervovaným bitům:

Bit požadavku na data
Substitute Remote Request (*SRR*) nahrazuje RTR bit ze základního rámce a označuje rozšířený rámec.

Příznak rozšíření
v recesivním stavu označuje, že bude následovat další identifikátor zprávy.

Identifkátor požadavku
označuje na 18 bitech typ zprávy, o kterou je žádano.

5.3.3 Chybový rámec a rámec signalizující přetížení

Oba rámce mají podobnou strukturu, která je ovšem zcela odlišná od struktury datového a rozšířeného rámce. Obecná struktura obou rámců rámců je vyobrazena na obrázku 5.2, níže pak popis jednotlivých částí:

Příznak chyby
je 6bitová část signalizující aktivní či pasivní[1] chybu přenosu. Rozlišení mezi chybovým rámcem a rámcem signalizující přetížení přichází se zahrnutím bitového prokládání – v případě chybového rámce nedochází ke prokládání, v případě rámce přetížení ano.

Ukončení rámce
je 8bitová recesivní hodnota ukončující rámec.

5.4 Ovladač pro modul sběrnice CAN

Jako modul pro sběrnici CAN bude použita vývojová deska s mikrokontrolérem MCP2515 a řadičem TJA1050[2]. Prvně jmenovaný je na desce hlavním řídícím prvkem, který vystavuje

[1] Odeslání aktivní chyby na sběrnici způsobí chybový stav u všech zařízení na sběrnici, vzhledem k tomu, že je aktivní chybový stav cíleně odeslán bez bitového prokládání dominantními bity. Pasivní chyba je signalizována recesivními bity, které chybu u dalších zařízení nevyvolají – to protože je recesivní stav výchozím stavem na sběrnici.
[2] https://www.laskakit.cz/mcp2515-can-bus-modul-tja1050--8mhz--spi/

rozhraní SPI[3] pro komunikaci a řídí druhý mikrokontrolér, který je rozhraním pro CAN sběrnici. Jedná se o stejnou konfiguraci mikrokontrolérů, jaká se nachází na známějším, avšak komerčně nedostupném, modulu Canberry.

Mikropočítač se systémem FreeRTOS je pomocí svého prvního modulu sběrnice SPI připojen k této desce a sběrnicí SPI je i následně řízen modul pro komunikaci na sběrnici CAN. Samotný ovladač pro sběrnici SPI pro Raspberry Pi 3B+ je založen na projektu `bcm2835` ve verzi `1.71` vývojáře Mike McCauleyiho[4], který implementuje ovladač pro starší verzi Raspberry Pi. V rámci této práce byla knihovna upravena pro novější verzi Raspberry Pi 3B+ se SoC BCM2837.

Samotný ovladač pro sběrnici CAN definuje ve veřejném rozhraní několik symbolů, z nichž nejdůležitější jsou `struct CanFrame`, `void can_receive_frame(CanFrame* frame)` a `void can_send_frame(CanFrame* out_frame)`. Jejich konkrétní použití lze vidět v ukázce 5.2.

```
void main(void) {
    // prepare frame with identifier=1
    // and data as string "foo bar" (without null delimiter)
    CanFrame frame = {1, {'f','o','o', ' ', 'b', 'a', 'r'}, 7};

    // sends prepared frame to bus
    can_send_frame(&frame);

    // wait for frame and store it into frame variable
    can_receive_frame(&frame);

    // received data are stored in frame.data
}
```

Zdrojový kód 5.2: Práce s ovladačem pro sběrnici CAN: nejprve je připraven statický rámec pro odeslání na sběrnici, ten je následně odeslán a na závěr je na stejný paměťový prostor přijat příchozí rámec ze sběrnice.

5.5 Tvorba demonstrační aplikace

Po integraci ovladačů pro sběrnice SPI a potažmo CAN bylo možné v kontextu této práce přistoupit k samotné implementaci aplikace používající modul sběrnice CAN pro komunikaci vnitřního stavu. Oproti první demonstrační aplikaci z kapitoly 4 se aplikace liší mírně ve vstupní funkci `void main(void)` tím, že je přidána inicializace modulu přes funkci ovladače `void can_init(void)` a odebrána registrace úlohy reportující stav čítačů na sériovou linku. Podstatně změněná je úloha `void task10Hz(void * param)`, která nyní obsahuje reportování stavu čítačů na sběrnici CAN – její nová podoba je zobrazena v ukázce 5.3.

[3] *Serial Peripheral Interface* (*SPI*) je sériové rozhraní pro komunikaci s periferiemi. Je synchronní vůči hodinovému signálu a v komunikaci používá schéma *master-slave*, kdy se na jedné sběrnici může nacházet více podřízených zařízení.

[4] https://www.airspayce.com/mikem/bcm2835/index.html

```
void task10Hz(void* pParam) {
    // pointer to both counters
    unsigned long* counters = pParam;

    const TickType_t tickFreq = (1000 / 10) / portTICK_PERIOD_MS;
    TickType_t lastWakeTime;
    // prepare CAN frame with identifier and expected length:
    // data length is based on total length of both counters
    CanFrame frame;
    frame.id = 1;
    frame.length = 2 * sizeof(unsigned long);

    // reset counted related to 10 Hz task
    *counters = 0;
    lastWakeTime = xTaskGetTickCount();
    while (1) {
        // enter the critical section
        taskENTER_CRITICAL();
        // increment the 10 Hz counter
        (*counters)++;

        // write data from counters into prepared CAN frame
        *((unsigned long*) &frame.data + 0) = *(counters + 0);
        *((unsigned long*) &frame.data + 1) = *(counters + 1);

        // send the frame to CAN bus
        can_send_frame(&frame);

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

Zdrojový kód 5.3: Úloha z druhé demonstrační aplikace běžící na frekvenci 10 Hz: oproti situaci z první demonstrační aplikace je připraven rámec sběrnice CAN při inicializaci úlohy – identifkátor zprávy je staticky nastaven na hodnotu 1, délka dat zprávy pak podle celkové délky obou čítačů. Do tohoto rámce jsou v rámci smyčky reportovány samotné stavy čítačů a s pomocí rozhraní ovladače je rámec odeslán na sběrnici.

5.6 Běh aplikace

Jako u první demonstrační úlohy, mikropočítač po zavedení paměťové karty s obrazem aplikace a resetu zařízení nastartuje a bezprostředně začne vykonávat definované a zaregistrované úlohy. Při připojení modulu na logický analyzátor lze následně sledovat provoz na sběrnici CAN, respektive reportované stavy vnitřních čítačů obou úloh.

Kapitola 6

Použití pro distribuované řídící aplikace

Distribuované řídící aplikace jsou typicky průmyslové aplikace, při kterých je jejich součástí vyšší množství autonomních senzorů, akčních členů a obecně prvků, z nichž ovšem žádný není v pozici hlavního řídícího prvku [32]. Mikropočítač Raspberry 3B+ je vzhledem ke své variabilitě použití vhodným kandidátem pro aplikace v distribuovaných systémech – především díky jeho široké nabídce periférií a komunikačních rozhraní. Použitá architektura Arm též poskytuje několik režimů běhu procesoru v módech snížené spotřeby, což mikropočítač mj. předurčuje pro použití s periodickými úlohami s dlouhou periodou.

Průmyslová sběrnice CAN pochází z automotivního průmyslu, ovšem díky své jednoduchosti a univerzálnosti se rozšířila do oblasti řídících a měřících systémů [6, 14, 24]. Spojení zmíněného mikropočítače, operačního systému FreeRTOS a sběrnice CAN představuje silnou kombinaci vhodnou pro použití v aplikacích, kam se přímo nehodí komerční průmyslová řešení – tedy do vývojových, experimentálních, či edukativních použití, u kterých ve vhodné rychlé prototypování výsledného zařízení. Projekt FreeRTOS navíc s širokou komunitní podporou a dokumentací představuje spolehlivý operační systém z rodiny volně dostupných systémů, a tedy je vhodný do výše zmíněných aplikací.

Kapitola 7

Závěr

Práce se věnuje problematice operačních systémů reálného času s fixní prioriou úloh v kontextu jejich nasazení v praxi. Tyto systémy jsou nejprve rozebrány z pohledu definice, jejich tříd dle plnění časových limitů a následně jsou popsány typy jejich kritických komponent, kterými jsou plánovače. U těch je představeno a vizualizováno chování těch v praxi nejčastěji používaných a následně práce navazuje přehledem volně dostupných operačních systémů reálného času a představením mikropočítače Raspberry Pi 3B+ a jeho základních principů.

Z dostupných systémů byl vybrán systém FreeRTOS, pro nějž se v rámci práce podařilo vytvořit podporu běhu na mikropočítači Raspberry Pi 3B+ – počínaje konfigurací překladače a linkeru, přes podporu systémových volání a vytvoření prostředí pro vývoj aplikací, až ke zpracování přerušení a podpoře pro aplikační úlohy v rámci RTOS. První demonstrační úloha je realizována s třemi úlohami, z nichž dvě periodicky inkrementují své čítače ve sdílené paměti, a třetí reportuje jejich stav na sériovou linku. Následně je vytvořeno prostředí pro komunikaci s modulem komunikujícího na sběrnici CAN. Toho je využito pro druhou demonstrační aplikaci, v rámci které je stav interních čítačů reportován pomocí zpráv odesílaných na tuto sběrnici. Na závěr je diskutováno použití projektu FreeRTOS na zmíněném mikropočítači pro distribuované řídící aplikace – to je označeno za vhodné především díky širokému záběru vestavěných periférií a architektuře Arm.

Výsledkem práce je zrealizovaná podpora pro volně dostupný operační systém reálného času FreeRTOS pro mikropočítač Raspberry Pi 3B+ – výsledný systém podporuje systémová volání, základní periferie mikropočítače a preempci systému, která je důležitá pro splnění podmínky jeho běhu v reálném čase.

Literatura

[1] Abeni, L.; Buttazzo, G.; Superiore, S.; aj.: Integrating Multimedia Applications in Hard Real-Time Systems. 07 2000.

[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] Amos, B.: *Hands-On RTOS with Microcontrollers*. Birmingham, England: Packt Publishing, 5 2020.

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

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

[6] Bo, L.; Tao, J.: *The design of monitoring system based on CAN bus*. IEEE, 05 2012, ISBN 978-1-4577-1604-1, 137-140 s., doi:10.1109/MIC.2012.6273241.

[7] Boland, H.; Burgett, M.; Etienne, A.; aj.: *An Overview of CAN-BUS Development, Utilization, and Future Potential in Serial Network Messaging for Off-Road Mobile Equipment*. Bahauddin Zakariya University, 07 2021, ISBN 978-1-83881-921-7, doi:10.5772/intechopen.98444.

[8] Brandenburg, B.; Anderson, J.; Baruah, S.; aj.: BJÖRN B. BRANDENBURG: Scheduling and Locking in Multiprocessor Real-Time Operating Systems. 05 2012.

[9] Brandt, T. K. C. L. S.: Firm Real-Time Processing in an Integrated Real-Time System. 2006.

[10] Burns, A.; Wellings, A.: *Real-time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. International computer science series, Addison-Wesley, 2009, ISBN 9780321417459.
URL https://books.google.cz/books?id=X09dPgAACAAJ

[11] Cabrera-Gámez, J.; de Miguel, A. R.; Domínguez-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.

[12] Calandrino, J. M.; Leontyev, H.; Block, A.; aj.: LITMUSRT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, 2006, s. 111–126, doi:10.1109/RTSS.2006.27.

[13] Cerqueira, F.; Brandenburg, B. B.: A Comparison of Scheduling Latency in Linux, PREEMPT-RT, and LITMUS RT. 2013.

[14] Chen, L.; Guo, Y.: Design of the Distributed Control System Based on CAN Bus. *Computer and Information Science*, ročník 4, 06 2011: s. 83–89, doi:10.5539/cis.v4n4p83.

[15] Davari, S.; Sha, L.: Sources of Unbounded Priority Inversions in Real-time Systems and a Comparative Study of possible Solution. *ACM SIGOPS Operating Systems Review*, ročník 26, 06 1992, doi:10.1145/142111.142126.

[16] 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.

[17] 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

[18] Fan, X.: *Real-time embedded systems*. London, England: Newnes, 1 2015.

[19] Fayyad, H.; Perneel, L.; Timmerman, M.: Linux PREEMPT-RT v2.6.33 versus v3.6.6: better or worse for real-time applications? *ACM SIGBED Review*, ročník 11, 02 2014: s. 26–31, doi:10.1145/2597457.2597460.

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

[21] Gay, W.: *Beginning STM32*. Apress Berkeley, CA, 06 2018, ISBN 978-1-4842-3623-9, s. 317–331, doi:10.1007/978-1-4842-3624-6_18.

[22] Ismail, K.; Muharam, A.; Pratama, M.: Design of CAN Bus for Research Applications Purpose Hybrid Electric Vehicle Using ARM Microcontroller. *Energy Procedia*, ročník 68, 2015: s. 288–296, ISSN 1876-6102, doi:https://doi.org/10.1016/j.egypro.2015.03.258, 2nd International Conference on Sustainable Energy Engineering and Application (ICSEEA) 2014 Sustainable Energy for Green Mobility.
URL https://www.sciencedirect.com/science/article/pii/S1876610215005640

[23] Kim, J.; Kim, S.-W.; Kim, D.-Y.; aj.: Soft real-time scheduling in a general purpose operating system: Scheduling daemon approach. *Comput. Syst. Sci. Eng.*, ročník 17, 01 2002: s. 3–11.

[24] a kol., J. K.: *Distribuované systémy řízení*. Vysoká škola báňská - Technická univerzita Ostrava, ISBN 978-80-248-2599-1.

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

[26] 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

[27] Murikipudi, A.; Venugopal, P.; Vigneswaran, T.: Performance Analysis of Real Time Operating System with General Purpose Operating System for Mobile Robotic System. *Indian Journal of Science and Technology*, ročník 8, 08 2015, doi:10.17485/ijst/2015/v8i19/77017.

[28] 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.

[29] 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.

[30] Oshana, R.: 8 - Real-Time Operating Systems for DSP. In *DSP Software Development Techniques for Embedded and Real-Time Systems*, editace R. Oshana, Embedded Technology, Burlington: Newnes, 2006, ISBN 978-0-7506-7759-2, s. 261–320, doi:https://doi.org/10.1016/B978-075067759-2/50010-7.
URL
https://www.sciencedirect.com/science/article/pii/B9780750677592500107

[31] Ossont, S.; Basford, P.; Perkins, C.; aj.: Commodity single board computer clusters and their applications. *Future Generation Computer Systems*, ročník 89, 06 2018, doi:10.1016/j.future.2018.06.048.

[32] Otčenášek, M.: Distribuované řídící systémy a jejich využití v praxi. 2008.
URL https://dspace.vutbr.cz/handle/11012/17775

[33] Queudet, A.; Chetto, M.: *Quality of Service Scheduling in the Firm Real-Time Systems*. University of Nantes, 04 2012, ISBN 978-953-51-0510-7, doi:10.5772/37318.

[34] 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.

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

[36] Sha, L.; Rajkumar, R.; Lehoczky, J.: Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, ročník 39, č. 9, 1990: s. 1175–1185, doi:10.1109/12.57058.

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

[38] STRNADEL, J.: Studijní opora k předmětu ROS. 2006.
URL https://www.vut.cz/studenti/predmety/detail/174572?apid=174572

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

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

[41] *uC/OS-III The Real-Time Kernel User's Manual*. Micrium Press, 2010, ISBN 978-0-9823375-9-2.

[42] Xu, J.; Parnas, D.: Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, ročník 16, č. 3, 1990: s. 360–369, doi:10.1109/32.48943.

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