VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA JADER REAL-TIME OPERAČNÍCH SYSTÉMŮ BĚŽÍCÍCH NA PLATFORMĚ FITKIT

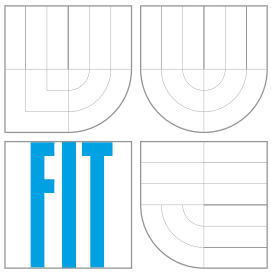
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETER RAJNOHA

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA JADER REAL-TIME OPERAČNÍCH SYSTÉMŮ BĚŽÍCÍCH NA PLATFORMĚ FITKIT

ANALYSIS OF REAL-TIME OPERATING SYSTEM KERNELS RUNNING ON FITKIT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETER RAJNOHA
AUTHOR

VEDOUCÍ PRÁCE — Ing. JOSEF STRNADEL, Ph.D.
SUPERVISOR

BRNO 2009

Abstrakt

Táto práca sa zaoberá problematikou výstavby RT operačných systémov pre použitie vo vstavaných zariadeniach. Práca sa zameriava hlavne na možné využitie RT systémov pre platformu FITkit a popisuje jednotlivé problémy a ich možné riešenia. Jedným z takýchto problémov je získanie časových závislostí pre úlohy, ak chceme zabezpečiť ich RT vlastnosti. Pre tento účel bol rozšírený existujúci simulátor daného mikrokontroléru, ktorý sa nachádza aj v platforme FITkit. Simulátor je potom možné využiť pre detailné sledovanie behu jednotlivých úloh v systéme na základe dynamickej analýzy, zbierať časové štatistiky pre časti programu, prípadne ho rozšíriť o ďalšie moduly. S využitím znalosti funkcionality konkrétneho operačného systému a získaných časových závislostí bol ako ukážka integrovaný plánovací mechanizmus RM do systému FreeRTOS.

Kľúčové slová

FITkit, real-time, operačný systém, mikrokontrolér, plánovanie, úlohy, RM, odozva, simulátor, časová analýza, FreeRTOS, MSPsim

Abstract

The project is dedicated to the identification of the problems found while building RT operating systems for use in embedded devices. The project's main topic is the possibility of using RT system in the FITkit platform and it also discusses individual problems and their possible solutions. One of the problems is the way of acquiring the timing information for tasks to ensure their RT properties. We have extended existing simulator for given microcontroller that is also part of the FITkit. The simulator can be used for detailed monitoring of the execution of individual tasks in the system based on dynamic analysis, collecting timing statistics for given blocks of the program or it can be extended by new modules. The RM scheduling mechanism has been integrated into FreeRTOS systems as an example by considering the knowledge of the concrete operating system and acquired timing information.

Keywords

FITkit, real-time, operating system, microcontroller, scheduling, tasks, RM, response time, simulator, timing analysis, FreeRTOS, MSPsim

Citácia

Peter Rajnoha: Analýza jader real-time operačních systémů běžících na platformě FITkit, diplomová práce, Brno, FIT VUT v Brně, 2009

Analýza jader real-time operačních systémů běžících na platformě FITkit

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Josefa Strnadela, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Peter Rajnoha
20. mája 2009

Poďakovanie

Týmto by som chcel poďakovať pánovi Ing. Josefovi Strnadelovi, Ph.D., vedúcemu mojej diplomovej práce, za jeho cenné rady, pripomienky a pomoc pri vypracovávaní tejto práce.

Obsah

Kapitola 1

Úvod

V praktickom živote sa denne stretávame s veľkým množstvom *vstavaných zariadení*, ktoré vykonávajú rozmanité činnosti. Takéto zariadenia sa vyrábajú pre riešenia konkrétnych problémov, a preto ich väčšinou považujeme za jednoúčelové a pevne viazané na oblasť, v ktorej daný problém riešia. S postupným vývojom, zvyšujúcim sa výkonom, klesajúcou cenou súčastí potrebných na výrobu takýchto zariadení a taktiež aj mierou ich dostupnosti na trhu, neustále narastá aj komplexnosť jednotlivých riešení.

Nárast v zložitosti produkovaných systémov je spôsobený zvyšujúcimi sa požiadavkami, ktoré sú na zariadenia kladené. Ide predovšetkým o rozšírenia v rámci miery zabezpečovanej funkcionality, taktiež aj vyššie požiadavky kladené na flexibilitu. Riešenie je tým flexibilnejšie, čím viac povoľuje dodatočné úpravy a doplnenia funkcionality na základe budúcich požiadavok používateľov bez toho, aby bolo nutné fyzicky meniť hardvérovú štruktúru. Z tohto pohľadu sa určitá časť návrhu a realizácie presúva do softvérovej oblasti, ktorá ponúka otvorenejší prístup vývoja s možnosťou uvažovania prípadných budúcich zmien, resp. možnosti výroby dostatočne všeobecného hardvérového riešenia, ktoré je možné ľahko prispôsobiť konkrétnej situácií na základe dodania príslušného špecializovaného softvéru.

Zloženie súčasných vstavaných zariadení pozostáva vo väčšine prípadov z výpočtovej jednotky (CPU), pamäte (RAM, ROM, FLASH) a vstupno-výstupných súčastí, ktoré tvoria rozhranie s okolitým prostredím. Štandardne sa používajú mikrokontroléry (μC), ktoré integrujú práve niekoľko takýchto súčastí systému v rámci jedného fyzického púzdra. Vstavané zariadenie ako celok spracováva hodnoty získané zo vstupného rozhrania a na ich základe spolu s presne definovaným postupom uloženým v pamäti, produkuje výstupné hodnoty. Príkladom môže byť riadiaci systém, ktorý pozostáva zo vstupných senzorov, rozhodovacieho podsystému a akčných členov, ktoré zabezpečujú konkrétnu realizáciu funkcionality. Ak existuje v systéme viacero takýchto vstupných a výstupných bodov, pričom používame spoločné prostriedky pre realizáciu výpočtu alebo rozhodovania, je nutné vhodným spôsobom koordinovať ich využitie tak, aby bolo efektívne a zároveň spĺňalo zadané podmienky.

Funkcionalitu komplexného systému rozdeľujeme na logické časti. Ak je táto časť realizovaná tak, že využíva CPU daného vstavaného systému (to znamená, že je reprezentovaná určitým programom, ktorý dokáže CPU interpretovať), tak o jednotlivých inštanciách programu potom hovoríme ako o *úlohách*, resp. *procesoch* systému. Jednou z definovaných podmienok pre jednotlivé úlohy môže byť správnosť a *včasnosť vykonania výstupnej akcie na vstupný podnet*. V tomto prípade ide o zabezpečenie odpovedi systému na podnet do určitého definovaného *časového limitu*. Ak systém dokáže zabezpečiť splnenie týchto podmienok, hovoríme o tzv. *real-time (RT) systéme*[11]. Požiadavky kladené na presnosť a včasnosť odozvy systému na podnety môžeme nachádzať práve vo vstavaných systémoch.

Na základe toho, do akej miery je splnenie úlohy do určitého definovaného časového okamihu kritické, môžeme rozlišovať tri základné druhy RT systémov [11, 14]:

- Soft RT systémy – u týchto systémov je síce definovaný časový limit pre jednotlivé úlohy, ale ich nedodržanie nespôsobí zlyhanie systému ako celku a iné vážnejšie problémy. Ak bude ale dochádzať k stálemu nedodržiavaniu týchto limitov, môže byť funkcionalita a kvalita služieb systému degradovaná z pohľadu používateľa.
- Firm RT systémy – občasné nedodržanie časových limitov úloh nespôsobí fatálne zlyhanie systému. Ak sa ale prekročí určitá miera nedodržiavania limitov, môže sa systém stať nepoužiteľným a môže spôsobovať škody.
- Hard RT systémy – systémy tohto typu sa vyznačujú pevnou podmienkou pre dodržanie časových limitov úloh (resp. zabezpečením odozvy systému na podnet). Ak sa časový limit prekročí, bude to mať katastrofálne následky, alebo to povedie k úplnému zlyhaniu systému.

Je zrejmé, že základom pre RT systémy je práve uvažovanie času a vhodné *naplánovanie* využitia prostriedkov tak, aby boli časové podmienky (limity) danej úlohy splnené. Ak takýto systém pozostáva z viacerých úloh, ktoré využívajú a zdieľajú hardvérové prostriedky, komunikujú medzi sebou, sú v určitých momentoch synchronizované a ich beh je možné prelínať a vytvoriť pseudoparalelné spracovanie (ak je počet úloh väčší ako počet dostupných CPU), tak je vhodné uvažovať o vytvorení a použití *operačného systému* aj v prostredí vstavaných systémov. Jednou z najdôležitejších súčastí jadra RT operačného systému (RTOS) je práve *plánovač úloh*. Ten tvorí hlavný bod pri rozhodovaní a prideľovaní CPU ako prostriedku a tým aj určovaní úspešnosti vykonania danej úlohy pri splnení časových podmienok.

Zavedenie operačného systému nám teda umožní jednak vytvoriť abstraktnú vrstvu nad prostriedkami daného zariadenia, ale aj vytvoriť centrálny rozhodovací bod pri plánovaní úloh a vytvoriť požadovanú flexibilitu. To znamená, že spôsob, akým pracuje zariadenie nie je pevne dané, ale je možné ho meniť na základe zavádzania rôznych úloh do systému. V prípade, že je to systém s *dynamickým zavádzaním úloh*, tak bude možné tieto úlohy za behu systému spúšťať a ukončovať. Naopak, pri *statickom zavádzaní úloh*, sú tieto pevne dané od počiatku a nemenné počas celého behu RT systému (budú zavedené v systéme neustále). Keďže služby operačného systému zabezpečia základnú prácu s hardvérovými prostriedkami, je možné sa sústrediť už len na konkrétne riešenia problémov a prerozdelenie funkčnosti systému na jednotlivé procesy.

Cieľom práce je vytvorenie alebo využitie a vhodné upravenie existujúceho jednoduchého jadra RTOS tak, aby pracovalo na platforme *FITkit*. Zároveň bude našou úlohou poskytnúť prostriedok pre vykonávanie časových analýz a celkového behu systému. FITkit pozostáva z niekoľkých základných stavebných blokov, ktoré sa používajú aj v reálnych vstavaných zariadeniach a slúži predovšetkým na študijné účely v prostredí FIT VUT. V súčasnosti ale nie je priamo dostupný nástroj na sledovanie behu systému a zároveň sledovanie jednotlivých dôležitých udalostí spolu s meraním času vykonávania častí programu tohto systému. Preto sa v našej práci zameriame aj na úpravu existujúceho simulátoru, čím umožníme používateľovi získať prehľad o tom, čo sa v systéme deje a prípadne môže toto slúžiť aj ako pomôcka pri hľadaní problémov a ladení programov. My využijeme túto funkcionalitu na získanie prehľadu o časových závislostiach v systéme.

Práca je rozdelená do niekoľkých logický častí. V rámci druhej kapitoly sa zameriame na stručný popis platformy FITkit, hlavne na obsiahnutý mikrokontrolér. Bude taktiež

poukázané na spôsob vytvárania jednotlivých aplikácií pre túto platformu. Ďalej, v tretej kapitole, bude uvedená problematika RT systémov, vývoja RTOS, možné implementácie, model a mechanizmy plánovania RT úloh. Nasledujúca časť v rámci štvrtej kapitoly práce bude obsahovať úvod do problematiky časových analýz a simulácie systémov, kde budú popísané jednotlivé problémy, s ktorými sa môžeme stretnúť. V piatej kapitole sa zameriame na popis rozšírenia funkcionality existujúceho simulačného nástroja pre účely vykonávania meraní a získavania časových závislostí. V rámci siedmej kapitoly vyberieme existujúci referenčný RTOS, oboznámime sa s jeho internou štruktúrou a funkcionalitou. Tento systém prípadne vhodne upravíme tak, aby spĺňal naše dodatočné požiadavky na real-time vlastnosti. Záverečná časť práce obsiahnutá v siedmej kapitole bude popisovať spôsob, akým je možné využiť časové štatistiky pre zabezpečenie RT vlastností systému. Ako ukážka bude slúžiť integrácia algoritmu RM do systému FreeRTOS. Na záver tejto práce poukážeme na ďalšie možné využitia daného simulátora spolu s realizovaným rozšírením a zhrnieme výsledky práce.

Kapitola 2

Platforma FITkit

FITkit je platforma, ktorá primárne slúži ako pomôcka pre zoznamovanie sa s problematikou navrhovania vstavaných systémov a vytváranie príslušných aplikácií. Základnými súčasťami platformy je μC a programovateľné hradlové pole FPGA. Tým je možné modifikovať funkcionalitu platformy ako celku zo softvérového, ale aj z hardvérového hľadiska. Medzi ďalšie súčasti patria aj jednotlivé periférie, ktoré sú prístupné a je možné ich neobmedzene využívať. Platforma je koncipovaná ako open-source pre softvérovú a open-core pre hardvérovú časť [15]. FITkit pozostáva z týchto konkrétnych hardvérových súčastí a rozhraní:

- programovateľné hradlové pole FPGA Xilinx Spartan 3 XC3S50-4PQ208C,
- μC Texas Instruments MSP430F168,
- USB-UART prevodník FTDI FT2232C,
- stereo audio IN/OUT zosilňovač Texas Instruments TPA6111A2,
- dva PS2 konektory,
- jeden VGA konektor,
- jeden RS232 konektor,
- DRAM 8x8Mbit,
- serial FLASH 2Mbit,
- 16-tlačidlová klávesnica (4x4),
- jednoriadkový LCD displej so šírkou 16 symbolov.

2.1 Súčasný spôsob vytvárania aplikácií

V súčasnosti existuje pre FITkit niekoľko ukážkových aplikácií. Aplikácia pozostáva zo softvérovej a hardvérovej časti popisu riešenia. Pre softvérovú časť aplikácie existuje už vytvorená knižnica s názvom `libfitkit`, ktorá ponúka základné inicializačné a konfiguračné operácie potrebné pre beh aplikácie na danom μC. Taktiež sú (v čase písania tohto textu) k dispozícii knižničné funkcie pre prácu s LCD displejom, klávesnicou, funkcie pre komunikáciu s FPGA a pre prácu so senzorom teploty.

Softvérová časť je popisovaná v *jazyku C*, kde pre konkrétny použitý μC (MSP430F168) existuje príslušný prekladač (napríklad `mspgcc`). Hardvérová časť popisu riešenia sa opiera o využitie FPGA, ktorej funkcionalitu je možné meniť na základe výslednej konfigurácie.

Podobne, ako u softvérovej časti, aj tu existuje už vytvorená základná štruktúra, nad ktorou sa vybudovávajú konkrétne hardvérové riešenia popisované v *jazyku VHDL*. Časti, ktoré sú popisované jazykom VHDL sú v konečnom kroku syntetizované použitím vhodného vývojového nástroja, pričom výsledkom je práve spomínaná konfigurácia, ktorá mení spôsob práce FPGA ako hardvérovej súčasti systému. V nasledujúcom texte sa budeme zaoberať už len časťou, ktorá priamo súvisí s vývojom a behom cieľového RTOS spolu s RT aplikáciami, teda μC (resp. jeho CPU), na ktorom je kód danej aplikácie spúšťaný.

Aplikácie určené pre spustenie na CPU v rámci platformy FITkit sú v súčasnej dobe väčšinou vytvárané ako nekonečné cykly, v tele ktorých sa nachádza obslužný kód. Ten obsahuje jednak volania funkcií vykonávajúcich konkrétnu činnosť typickú pre danú aplikáciu a taktiež aj volanie knižničnej funkcie pre obsluhu pripojeného terminálu `TERMINAL_Idle()`, ktorá realizuje aj jednoduchý príkazový riadok dostupný v termináli. Keďže je nastavený aj watchdog, je nutné jeho explicitné resetovanie volaním knižničnej funkcie `SW_WDG()` na začiatku každej iterácie vykonávaného nekonečného cyklu, alebo aj v momente, keď je to potrebné v rámci aplikačných funkcií, ak sú tieto výpočtovo náročné (cyklus watchdogu je štandardne nastavený na hodnotu 8.9 ms). Pred samotným nekonečným cyklom sa však ešte musí inicializovať μC. Pre túto inicializačnú činnosť je už pripravená funkcia `INIT_HW_MCU()`, ktorá je tiež súčasťou knižnice `libfitkit`.

Nasledujúci úsek kódu v jazyku C reprezentuje príklad základnej štruktúru programu aplikácie s využitím knižnice `libfitkit`, ktorá je určená pre μC v rámci platformy FITkit (pri použití kompilátoru `mspgcc`).

```
#include <msp430x16x.h>
#include <fitkitlib.h>

unsigned char USER_CMD_Decode(char *UString, char *String) {
  if (strcmp5(UString, "HELLO")) {
    /* ...kód realizujúci príkaz "HELLO"... */
  }
  else if (strcmp5(UString, "WORLD")) {
    /* ...kód realizujúci príkaz "WORLD"... */
  }
  else {
    return (CMD_UNKNOWN);
  }
  return USER_COMMAND
}

void Application_Function() {
  /* ...kód aplikácie... */
}

int main(void) {
  INIT_HW_MCU();            /* inicializácia mikrokontroléru */
  for( ; ; ) {
    SW_WDG();               /* resetovanie watchdogu */
    Application_Function(); /* naša aplikačná funkcia */
    Terminal_Idle();        /* obsluha terminálu, dekódovanie príkazov */
  }
}
```

Funkcia `INIT_HW_MCU` obsahuje niekoľko inicializačných krokov. V prvom rade obsahuje čakaciu dobu na stabilizáciu napájacieho napätia, následne inicializuje hodiny pre μC a FPGA, inicializuje reset pre FPGA, inicializuje softvérový watchdog, povolí prerušenia, spustí obsluhu terminálu a nakoniec naprogramuje FPGA z konfiguračných informácií zapísaných v (externej) pamäti FLASH. Úplne poslednou akciou je zaslanie znaku `>` na terminál, čím sa indikuje pripravenosť μC na zadávanie príkazov.

V uvedenom príklade si okrem popisovaných funkcií môžeme všimnúť aj použitie funkcie `USER_CMD_Decode(char *UString, char *String)`. Táto je zodpovedná za obsluhu dodatočných príkazov, ktoré je možné rozpoznať v rámci vstupu z terminálu. Takýmto spôsobom je možné realizovať vlastné aplikačné príkazy, ktoré môžu slúžiť, napríklad, na riadenie našej aplikácie. Samotná knižnica `libfitkit` poskytuje momentálne realizáciu týchto základných príkazov (výpis príkazu `help` v termináli pripojenom na FITkit):

```
PROG FPGA..programovani z pc (XModem)
PROG FPGA FLASH..programovani FPGA z flash

RESET MCU...software reset MCU
RESET FPGA...software reset FPGA
CLS...reset terminalu

FLASH W C chr adr...zapis znaku 'chr' dat do flash na adresu 'adr'
FLASH W X adr...zapis dat do flash na adresu 'adr' (XModem)
FLASH W FPGA..zapis FPGA dat z PC do flash (XModem)
FLASH R A adr....vypis 64Bytu flash z adresy 'adr'
FLASH R P page...vypis stranky c. 'page' flash
FLASH R B block..vypis bloku c. 'block' flash
FLASH R S........vypis status registru flash
FLASH E FPGA..odstraneni FPGA dat z flash
FLASH E A adr....smazani bytu ve flash na adrese 'adr'
FLASH E P page...smazani stranky 'page' flash
FLASH E B block..smazani bloku 'block' flash
FLASH E ALL......smazani cele flash

RAM D....dump RAM Memory
```

Ak sme definovali v rámci našej aplikácie vlastné aplikačné príkazy, budú tieto zobrazené na konci uvedeného zoznamu príkazov.

Použitím pripojeného terminálu a vyhradeného príkazu (`PROG FPGA` a `PROG FPGA FLASH`) je možné nahrať do FPGA konfiguráciu buď priamo, alebo z externej FLASH pamäte, ktorá je súčasťou platformy FITkit. Samotný program pre μC sa nahráva pomocou špecializovaného programu msp430-bsl spolu s využitím boot-loaderu, ktorý je obsiahnutý v mikroprocesore. Presný postup je možné nájsť v [15].

2.2 Mikrokontrolér MSP430F168

Mikrokontrolér MSP430F168, ktorý je súčasťou platformy FITkit obsahuje nasledovné integrované súčasti (podľa [6], [7]):

- 16-bitový procesor s architektúrou RISC a 16 registrami,

- 2KB RAM,
- 48KB FLASH (+256B informačnej pamäte),
- watchdog,
- dva 16-bitové časovače (označované ako `TIMER_A3` a `TIMER_B7`),
- jeden komparátor (označovaný ako `A`),
- DMA radič (3 kanály),
- jeden 12-bitový ADC (8 kanálov) a dvojitý 12-bitový DAC prevodník (2 kanály),
- komunikačné sériové rozhranie (označované ako `USART0`), ktoré môže fungovať ako asynchrónny UART, synchrónne SPI alebo I^2C,
- komunikačné sériové rozhranie (označované ako `USART1`), ktoré môže fungovať ako asynchrónny UART alebo synchrónne SPI,
- šesť 8-bitových `I/O` portov (označených ako `P1`–`P6`),
- JTAG rozhranie,
- kontrola napätia (SVS), generátor signálu reset pri dodaní napätia (POR) a detektor podnapätia (Brownout).

Pre realizáciu operačného systému, ktorý bude spustiteľný na danom μC, je potrebné zoznámiť sa predovšetkým s možnosťami integrovaného procesoru, jeho registrami, organizáciou pamäte, časovačom, systémom prerušení a celkovo procesom inicializácie mikrokontroléru.

2.2.1 Procesor a registre

Procesor, ktorý je súčasťou μC, je 16-bitový s architektúrou RISC, little endian. Inštrukčná sada obsahuje spolu 27 inštrukcií a procesor podporuje 7 rôznych adresných módov. Celkový počet registrov CPU je 16 [7].

Prvým z nich je *programový čítač* (register s označením `R0`, resp. `PC` – Program Counter). Jeho význam je štandardný, to znamená, že obsahuje adresu nasledujúcej inštrukcie, ktorá sa bude vykonávať. Adresa uložená v tomto registri je vždy párna.

Ďalším registrom je *zásobníkový ukazovateľ* (register `R1`, resp. `SP` – Stack Pointer). Využíva sa pre ukladanie návratových adries v rámci volaní funkcií (procedúr) a pri volaní obslužných rutín prerušení. Adresa uložená v registri je tiež vždy párna a na počiatku je nastavená na určitú adresu v adresovom priestore RAM pamäte. Register môže byť použitý so všetkými dostupnými inštrukciami a adresnými módmi.

Stavový register (`R2`, resp. `SR` – Status Register) má dvojitú funkciu. Pri použití registrového adresného módu obsahuje základné stavové informácie. Prístup v rámci iných adresných módov je vyhradený pre generovanie a používanie rôznych konštantných hodnôt. V tomto prípade sa register nazýva *prvý register pre generovanie konštánt* (`CG1` – Constant Generator 1).

Druhý register pre generovanie konštánt (`R3`, resp. `CG2` – Constant Generator 2) je register, ktorého úlohou je len generovanie konštantných hodnôt.

Ďalších dvanásť registrov je určených pre *všeobecné použitie* (`R4` – `R15`, General Purpose). Môžu byť použité ako registre pre dátové, adresné a indexovacie hodnoty.

Okrem uvedených registrov, existujú aj také, ktoré nie sú priamo súčasťou CPU, ale sú mapované do 64KB adresného priestoru hneď na jeho počiatku (viď tabuľka 2.1). Ide o tzv. *registre so špeciálnou funkciou* (`SFR` – Special Function Registers). Existuje celkovo šesť SFR registrov – dva registre pre povoľovanie prerušení (`IE1`, `IE2` – Interrupt Enable 1, 2), dva registre príznakov pre prerušenia (`IFG1`, `IFG2` – Interrupt Flag 1, 2) a dva registre pre povoľovanie jednotlivých modulov μC (`ME1`, `ME2` – Module Enable 1, 2).

2.2.2 Organizácia pamäte

Procesor mikrokontroléru je 16-bitový, máme teda k dispozícii *64KB adresovateľného pamäťového priestoru*. Tento adresovateľný priestor je rozdelený na niekoľko častí. Ak postupujeme od najnižšej adresy (`00h`) po najvyššiu (`0FFFFh`), tak máme adresovateľný priestor pre špeciálne registre, priestor pre mapovanie periférií, RAM, pamäť so štartovacími informáciami, informačnú pamäť, pamäť pre kód a dáta a na vrchole vektor prerušení. Tabuľka 2.1 prehľadovo zobrazuje organizáciu pamäťového priestoru [6].

Účel	**Typ pamäte**	**Veľkosť**	**Rozsah (hexadecimálne)**
Vektor prerušení	FLASH	32B	`0FFE0h - 0FFFFh`
Kód/dáta	FLASH	49152B	`04000h - 0FFFFh`
Informácie	FLASH	256B	`01000h - 010FFh`
Štartovacie informácie	ROM	1024B	`0C00h - 0FFFh`
RAM	RAM	2048B	`0200h - 09FFh`
Periférie (16-bit.)	-	256B	`0100h - 01FFh`
Periférie (8-bit.)	-	240B	`010h - 0FFh`
Register ME2	-	1B	`05h`
Register ME1	-	1B	`04h`
Register IFG2	-	1B	`03h`
Register IFG1	-	1B	`02h`
Register IE2	-	1B	`01h`
Register IE1	-	1B	`0h`

Tabuľka 2.1: Organizácia pamäte μC MSP430F168

Detailný popis mapovania jednotlivých periférií, resp. modulov a ich príslušných registrov prístupných práve v rámci vyhradenej oblasti pamäťového priestoru je možné nájsť v [7].

2.2.3 Prerušenia

Priority prerušení sú pevné a dané aj usporiadaním jednotlivých modulov, ktoré sú ich zdrojom. V prostredí používaného μC sa uvažujú tri základné druhy prerušení [7]:

- systémový reset (*system reset*),
- nemaskovateľné prerušenia (*non-maskable interrupts – NMI*),
- maskovateľné prerušenia (*maskable interrupts*).

Systémový reset má najväčšiu prioritu zo všetkých dostupných prerušení (priorita 15). K výskytu tohto prerušenia dochádza v čase spustenia μC, externého resetu, resetu od modulu watchdog alebo resetu na základe nesprávne zadaného hesla pre prístup do pamäte

FLASH. V rámci vektoru prerušení sa nachádza na najvyššej adrese v rámci adresového priestoru procesoru (`0FFFEh`).

Nemaskovateľné prerušenie nie je možné ignorovať globálnym nastavením príslušného bitu v rámci stavového registra (bit `GIE`, register `SR`). Je však možné tieto prerušenia povoliť alebo zakázať, ak je to možné, v rámci registrov pre povoľovanie prerušení (`IE1` alebo `IE2`), ktoré sú prístupné cez adresovateľný priestor μC (viď podsekcia 2.2.2). V tomto momente je dôležité brať do úvahy fakt, že pri výskyte nemaskovateľného prerušenia sa tieto registre (`IE1`, `IE2`) resetujú, a preto je nutné ich priamo softvérovo povoliť, ak chceme aj naďalej prijímať a obsluhovať nemaskovateľné prerušenia. Nemaskovateľné prerušenia majú po systémovom resete druhú najvyššiu prioritu (priorita 14, adresa `0FFFCh` v rámci adresového priestoru). Zdrojom nemaskovateľných prerušení môže byť [7]:

- Detekovaná hrana na $\overline{\texttt{RST}}$`/NMI` fyzickom vývode μC (nastaví sa zároveň bit `NMIIFG` v registri príznakov prerušení `IFG1`).
- Chyba oscilátoru (nastaví sa zároveň bit `OFIFG` v registri príznakov prerušení `IFG1`).
- Ochrana prístupu do FLASH pamäte (nastaví bit `ACCVIFG` v registri `FCTL3`, ktorý je tiež mapovaný do adresového priestoru v rámci periférií).

Maskovateľné prerušenia môžu byť zamedzené nastavením príslušného bitu, ktorý prislúcha priamo danému zdroju prerušenia. Taktiež môže byť tento typ prerušení zamedzený ako celok na globálnej úrovni, a to nastavením bitu `GIE`, ktorý sa nachádza v stavovom registri `SR` procesoru μC. Tento druh prerušení ma menšiu prioritu ako nemaskovateľné prerušenia a v rámci vektoru prerušení sa nachádzajú na adresách `0FFE0h` (najmenšia možná priorita s hodnotou 0) až `0FFFAh` (najväčšia priorita maskovateľných prerušení s hodnotou 13).

Pri výskyte prerušenia sa vždy dokončí aktuálne vykonávaná inštrukcia. Následne je na zásobník uložená aktuálna hodnota registra `PC` a stavového registra `SR`. Tieto dva registre sú jediné, ktoré sú automaticky ukladané ako kontext danej vykonávanej úlohy, ktorá bola prerušená. V prípade, že existuje viacero prerušení v jednom momente, budú sa tieto spracovávať v poradí ich priority. Po uložení kontextu sa resetuje obsah registra `SR` a obsah `PC` je nastavený na hodnotu, ktorá je zapísaná na príslušnom mieste vo vektore prerušení. Ďalšie (maskovateľné) prerušenia sú zakázané automatickým nulovaním bitu `GIE`. Následne sa bude vykonávať obsluha daného prerušenia. Ak chceme povoliť prerušenia aj v rámci aktuálne vykonávanej obsluhy prerušenia (tzv. *vnorovanie prerušení*), tak stačí, ak ich v rámci obsluhy explicitne povolíme nastavením bitu `GIE`. Pri návrate z prerušenia bude kontext prerušenej úlohy automaticky obnovený. Detailnejšie informácie o prerušeniach je možné nájsť v [6, 7].

2.2.4 Časovač

Mikrokontrolér, ktorý sa používa v platforme FITkit, obsahuje *dva nezávislé 16-bitové časovače* označované ako `TIMER_A3` a `TIMER_B7`. Tieto časovače môžu, v závislosti na spôsobe fungovania sčítača a jeho resetovania, pracovať v troch rôznych režimoch:

- sčítavanie po určitú hodnotu (*up mode*),
- kontinuálne sčítavanie (*continuous mode*),
- sčítavanie po určitú hodnotu a následné odčítavanie (*up/down mode*).

V každom z týchto módov je možné nastaviť hodnoty, s ktorými bude aktuálna hodnota sčítača porovnávaná. V prípade, že sa budú hodnoty zhodovať, bude časovač generovať prerušenie. Pri *sčítavaní po určitú hodnotu* sa po dosiahnutí určenej hodnoty sčítač resetuje na nulovú hodnotu. V prípade *kontinuálneho sčítavania* bude pokračovať až do maximálnej hodnoty, až potom bude táto nulovaná. V treťom možnom móde sa *sčítava po zadanú hodnotu, potom sa začne odčítavať* až po nulovú hodnotu. Nastavenie časovačov sa vykonáva, podobne ako pri iných moduloch, nastavením príslušných registrov, ktoré sú mapované do 64KB adresového priestoru.

Časovač `TIMER_A3` má tú istú funkcionalitu ako `TIMER_B7`. Rozdielom je, že u `TIMER_B7` je možné nastaviť veľkosť sčítača na 8, 10, 12 alebo 16 bitov. Taktiež tu existuje buffer pre porovnávané hodnoty, do ktorých sa vykonáva samotné sčítanie (umožňuje lepšie načasovanie zmeny porovnávanej hodnoty). U `TIMER_A3` je možné nastaviť tri nezávislé porovnávané hodnoty, u `TIMER_B7` je to až sedem nezávislých hodnôt. Informácie o ďalších odlišnostiach je možné nájsť v [7].

Úlohu jednoduchého časovača môže zastávať aj samotný modul watchdog. Tento modul dokáže pracovať v dvoch režimoch. Prvým režimom je práve klasický watchdog, kde po vypršaní určitého časového limitu určeného pre jeho znovunastavenie dochádza k resetovaniu μC. Druhým podporovaným režimom je intervalový časovač, ktorý, podobne ako v prípade časovačov `TIMER_A3` a `TIMER_B7`, dokáže generovať pravidelné prerušenia. V tomto prípade sa ale nenastavuje priamo hodnota 16-bitového sčítača, ale interval sa určí na základe konfigurácie v rámci jedného z watchdog registrov (register `WDTCTL` mapovaný v adresovom priestore v rámci periférií). Intervaly, ktoré budú dostupné, závisia od nastavenia zdroja hodín pre watchdog modul. Dostupné intervaly potom môžeme získať delením hodinového zdroja. Delitele sú 32768, 8192, 512 a 64.

2.2.5 Inicializácia mikrokontroléru

U μC použitého v platforme FITkit rozlišujeme tri príčiny, ktoré môžu spôsobiť jeho reinicializáciu (reset) [7]:

- POR (*Power-on Reset*) – reset pri spustení
- PUC (*Power-up Clear*) – watchdog reset a reset na základe ochrany prístupu
- BOR (*Brownout Reset*) – reset pri podnapätí

POR vzniká v prípade dodania elektrického napätia pre μC, signálom na $\overline{\texttt{RST}}$`/NMI` fyzickom vývode a taktiež aj prípadným detekovaným riadiacim signálom od modulu SVS. *PUC* vzniká vždy v prípade vzniku POR, naviac však aj v prípadoch vypršania intervalu pre obnovenie watchdogu a bezpečnostných kontrol pre prístup k watchdogu a FLASH pamäte. *BOR* vzniká v prípade detekovaného podnapätia.

Ako uvádza zdroj [7], v situácií resetu μC označovaného ako POR, dochádza k nasledovným automatickým inicializačným akciám:

- $\overline{\texttt{RST}}$`/NMI` vývod je nakonfigurovaný do módu reset,
- `I/O` vývody sú nastavené do vstupného režimu,
- jednotlivé moduly a ich príslušné registre sú nastavené na základné hodnoty,
- obsah stavového registra `SR` je resetovaný,
- watchdog modul je nastavený do režimu klasického watchdogu (teda nie časovač),
- obsah registra `PC` je nastavený na hodnotu, ktorá je umiestnená na adrese `0FFFEh`.

Následne po nastavení registra `PC` sa spustí vykonávanie inštrukcií programu s počiatkom na príslušnej adrese. Prvé softvérové kroky vedú k ďalšej inicializácií. V tomto prípade ide o tieto nastavenia [7]:

- inicializácia registra vrcholu zásobníka `SP` (väčšinou na najvyššiu adresu RAM v rámci adresového priestoru),
- inicializácia watchdog modulu do potrebného režimu,
- konfigurácia a inicializácia používaných modulov (periférií).

Po uvedenej inicializácií už môže byť spustený aplikačný kód, ktorý vykonáva a riadi požadovanú funkcionalitu mikrokontroléru.

2.2.6 Postinicializačné akcie

V rámci platformy FITkit sa nachádza aj programovateľné hradlové pole FPGA, ktoré je tiež nutné inicializovať a spustiť. Preto prvé akcie vykonávané po inicializácií samotného μC by mali zahŕňať práve nastavenie a spustenie tejto komponenty (napríklad nakonfigurovanie komunikačného kanálu medzi μC a FPGA, naprogramovanie FPGA z externej FLASH pamäte).

Kapitola 3

Real-time systémy

Ako už bolo spomínané v úvode, real-time (RT) systémy sa odlišujú od bežných systémov hlavne v požiadavke kladenej na *správnosť* a *včasnosť* odozvy na vstupné podnety. Podľa toho, do akej miery je splnenie týchto podmienok záväzné, odlišujeme *soft* (najmenšia záväznosť), *firm* (stredná záväznosť) a *hard* (najväčšia záväznosť) RT systémy. Okrem spomínaných podmienok býva u RT systémov kladený dôraz aj na *determinizmus*. Systém môžeme považovať za deterministický, ak pre každý možný stav a každý možný vstup existuje príslušná zmena stavu a výstup [11].

Bežný *model fyzickej realizácie* RT systému pozostáva z riadiaceho a riadeného systému a jeho okolia. Riadený systém poskytuje pre RT jadro (súčasť riadiaceho systému) jednotlivé parametrické hodnoty získané zo senzorov. Jadro (s uloženým vnútorným stavom) potom vykonáva rozhodovanie, na základe ktorého sa určia jednotlivé pokyny pre akčné členy riadeného systému, prípadne sa zmení vnútorný stav rozhodovacieho mechanizmu. Tento model sa opiera o riešenie sústavy rovníc nad vstupnými a príslušnými výstupnými hodnotami, ktoré sú vytvárané na základe skúseností a pozorovania reálneho prostredia [14].

Praktické modelovanie RT systémov sa však operia o *procesný pohľad* na problém (tzv. *logický model*) [14]. To znamená, že funkcionalita komplexného systému je rozložená na úlohy (procesy), ktoré sú vstupom pre spracovanie a naplánovanie samotným jadrom RT systému. Jadro potom na základe definovaných pravidiel, či už sa jedná o prioritné pravidlá, pravidlá pre splnenie časových limitov alebo využitie zdrojov, produkuje plán úloh. Plánom sú určené postupnosti vykonávania úloh. Tie sú viazané na špecifické podnety z okolia, resp. udalosti, ktoré následne spracovávajú.

Systém môže získavať jednotlivé podnety z prostredia dvoma základnými spôsobmi. Prvým z nich je tzv. *cyklický generátor*, ktorý v určitých časových okamihoch vzorkuje svoje prostredie a takto získava vstupné hodnoty. Druhý spôsob je založený na opačnom princípe (systém je v tomto prípade pasívny). Okolie, ktoré je zdrojom podnetov, dáva o svojej zmene vedieť *prerušením systému* [2]. Systém potom môže na základe výsledku rozhodovania buď okamžite spracovať danú zmenu, zaregistrovať ju pre neskoršie spracovanie, alebo úplne ignorovať. V prípade využitia prvého spôsobu môžeme hovoriť aj o tzv. *synchrónnych udalostiach*, pretože ich zisk je riadený samotným systémom. V druhom prípade môžeme hovoriť o *asynchrónnych udalostiach*, pretože tieto prichádzajú priamo z prostredia (to znamená externe vzhľadom na RT systém).

Na základe toho, aké sú časové vzťahy medzi jednotlivými výskytami udalostí, rozlišujeme *periodické* (časové okamihy výskytu udalostí sú pravidelné a periodické), *aperiodické* (časové okamihy, ktoré môžu byť pravidelné, ale nie periodické) a *sporadické* (nie sú ani pravidelné, ani periodické, väčšinou sú to výnimky a náhodné udalosti).

3.1 Real-time operačné systémy

Operačné systémy tvoria *abstraktnú vrstvu* nad hardvérovými prostriedkami. Sú zodpovedné za ich prideľovanie jednotlivým úloham, ktoré v systéme existujú. Medzi tieto prostriedky patrí hlavne CPU (abstrakcia na úrovni úloh, procesov, vlákien so súčasným vytvorením viacúlohového prostredia s prepínaním kontextu úloh), primárna pamäť (vytvorenie transparentného virtuálneho adresového priestoru pre jednotlivé úlohy), sekundárna pamäť (abstrakcia úložiska na úrovni súborového systému) a periférie (rôzne fronty pre prístup a podobne). Taktiež ponúkajú vysokoúrovňové softvérové prostriedky pre synchronizáciu úloh a ich vzájomnú komunikáciu (semafory, správy, signály, zdieľaná pamäť, sockety atď.).

Hlavný rozdiel medzi štandardným a real-time operačným systémom (RTOS) spočíva v spôsobe prideľovania jednotlivých prostriedkov. RTOS používa *špecializované prístupy* a algoritmy pre zabezpečenie splnenia definovaných *časových limitov*. Taktiež je RTOS z architektonického hľadiska navrhovaný tak, aby jednotlivé časové oneskorenia boli *deterministické* a v rámci možností minimalizované. Z hľadiska spracovania jednotlivých prerušení sú RTOS *reaktívnejšie* než klasické operačné systémy. Naopak, klasický operačný systém sa zameriava predovšetkým na zabezpečenie vlastností ako napríklad celková priepustnosť systému, interaktivita a optimálne využitie pamäte. Determinizmus a časové limity nie sú v tomto prípade brané do úvahy tak, ako v prípade RTOS a dôležitá je hlavne optimalizácia systému z pohľadu použiteľnosti pre koncového zákazníka, nie z pohľadu riadenia kritických úloh (úloh s pevnými podmienkami pre dodržanie limitov, najmä časových).

U jednoduchších systémov sa môže *prideľovanie prostriedkov vykonať staticky*, napríklad hneď po jeho spustení. Vzniká tým presne určený plán využitia prostriedkov. Tento prípad nastáva, ak je dopredu presne známa miera a momenty použitia jednotlivých prostriedkov. Nie je preto potrebné vykonávať dynamické kontroly a používať iné komplexné kontrolné mechanizmy, ktoré overujú a zabezpečujú prístup k prostriedkom počas behu systému. Tým sa celkový návrh zjednoduší, čo je dôležité hlavne pre vstavané RT systémy, pretože práve tieto majú oveľa obmedzenejšie prostriedky ako bežné systémy.

Vo svojej najzákladnejšej podobe by real-time operačný systém mal realizovať aspoň tieto tri mechanizmy [11]:

- plánovanie úloh,
- zavádzanie úloh a priraďovanie vstupných udalostí na spracovanie úlohami,
- komunikácia a synchronizácia úloh.

Centrálnou súčasťou RTOS je plánovač úloh. Dôvodom je fakt, že plánovač je zodpovedný za prideľovanie CPU ako prostriedku pre jednotlivé úlohy, čím vlastne aj priamo rozhoduje o úspešnosti alebo neúspešnosti dodržania časového limitu pre dokončenie danej úlohy. Pre plánovanie úloh existuje v prostredí RT systémov zavedený *model úloh*, ktorý definuje časové okamihy a intervaly (viď sekcia 3.2). Ich hodnoty tvoria podstatnú časť vstupných parametrov pre konkrétne plánovacie algoritmy.

Požiadavka na komunikačné a synchronizačné nástroje poskytované RT operačným systémom plynie z uvažovania viacúlohového systému s paralelným (resp. pseudoparalelným) behom jednotlivých úloh v RT prostredí.

3.1.1 Základné prístupy pre výstavbu jadra RTOS

Plánovacie jadro zabezpečuje viacúlohovosť systému a presne určuje, ktorá úloha bude spustená v určitom momente. Pre zabezpečenie tejto funkcionality existujú už vybudované základné prístupy [11, 14]:

- pseudojadrá,
- jadrá využívajúce prerušovací systém,
- jadrá realizujúce tzv. FBS (Foreground Background System) model,
- jadrá realizujúce tzv. TCB (Task Control Block) model.

Prvé tri uvedené prístupy sú určené hlavne pre systémy, u ktorých je dopredu známe zloženie z jednotlivých úloh (statické zavádzanie úloh). V prípade, že sa v čase behu zavádzajú nové úlohy do systému, resp. sú ukončované, tak je vhodné využiť štvrtý uvedený prístup, a to jadrá s TCB modelom úloh. Tento prístup umožňuje realizáciu dynamického zavádzania úloh.

Pseudojadrá

Pseudojadrá majú najjednoduchšiu architektúru. Ich názov odzrkadľuje fakt, že tieto jadrá nie sú plnohodnotné v zmysle definície základných poskytovaných mechanizmov a taktiež pracujú bez využitia prerušovacieho podsystému. Výhodou tohto spôsobu je jednoduchosť analýzy a celkovo nízke nároky na prostriedky. Pseudojadrá môžu mať niekoľko podôb, a to napríklad vyzývací cyklus, synchronizovaný vyzývací cyklus, cyklické vykonávanie, stavovo riadený kód a spolupracujúce úlohy (tzv. korutiny) [11, 14].

Vyzývací cyklus je cyklus, v rámci ktorého sa na základe podmienky spustí obslužná úloha. Podmienkou môže byť napríklad nastavenie príslušného bitu v určitom registri, ktorý sa testuje. Tento spôsob je výhodný v prípade, že procesor vykonáva buď jednoúčelovú úlohu, alebo kde nedochádza k prekrývaniu jednotlivých úloh, ak aj sú v systéme prítomné vo väčšom počte.

Synchronizovaný vyzyvací cyklus sa odlišuje od štandardného vyzývacieho cyklu zavedením čakacej doby medzi výskytom udalosti (resp. nastavením príslušného bitu) a obsluhou. Dôvodom zavedenia čakacej doby je čas potrebný na ustálenie prechodného stavu.

Ak máme v systéme niekoľko úloh, ktoré majú približne rovnaké doby vykonávania, tak ich môžeme cyklicky spúšťať jednu za druhou. Periódu volania určitej úlohy potom zvýšime jednoducho tak, že je v rámci jedného cyklu zavolaná viac krát. Pseudojadrá realizované s využitím tohto mechanizmu sa nazývajú pseudojadrá s *cyklickým vykonávaním*.

V prípade, že je možné úlohy logicky oddeliť na časti tak, že každú z nich je možné oddelene spúšťať na základe podmienok, tak hovoríme o *stavovo riadenom kóde*. Takýto kód je možné reprezentovať aj konečným automatom. Konečné automaty majú už solídne vypracovaný teoretický základ, takže takýto systém je možné potom aj ľahšie optimalizovať a verifikovať.

Spolupracujúce úlohy využívajú stavovo riadený kód jednotlivých úloh, pričom po ukončení každej časti je volaný plánovač, ktorý potom centrálne rozhoduje o spustení ďalšej časti konkrétnej úlohy. Plánovač je volaný explicitne na konci každej logickej časti úloh.

Jadrá využívajúce mechanizmus prerušení

Pre funkcionalitu prerušovacieho systému je potrebná hardvérová podpora, ktorá umožňuje takéto prerušenia generovať a prijímať. Zdrojom prerušení sú buď hardvérové súčasti (väčšinou periférie) alebo samotný program (softvérové prerušenie na základe použitia špeciálne vyhradenej inštrukcie procesoru). Pri výskyte prerušenia je aktuálne spustená úloha prerušená, pričom jej kontext je automaticky uložený pre budúce obnovenie.

Pre plánovanie procesov má prerušovací systém osobitný význam, pretože umožňuje realizovať *preemptívne plánovanie* úloh. V tomto prípade môže byť aktuálne spustenej úlohe odobraté CPU ako prostriedok a byť potom priradené inej úlohe, ktorá je pripravená, je v spustiteľnom stave a má vyššiu prioritu než aktuálna úloha. Dôležitým rozdielom oproti kooperatívnemu plánovaniu je ten, že úloha sa nemusí explicitne vzdávať CPU, aby mohla byť spustená iná úloha. Toto rozhodnutie už je možné vykonávať transparentne, bez pričinenia samotnej úlohy (samozrejme, aj v preemtívnom systéme môže ale stále existovať spôsob, akým sa úloha dobrovoľne vzdá CPU).

Pri prerušení sa nemusí automaticky ukladať plný kontext úlohy tak, ako je niekedy požadované. Preto je vždy nutné zoznámiť sa s cieľovou architektúrou a prípadné ďalšie položky kontextu uložiť dodatočne. Každá konkrétna architektúra je v tomto ohľade osobitná (viď napríklad automatické ukladanie kontextu pri prerušení v μC MSP430F168, podsekcia 2.2.3).

Tento spôsob práce s úlohami v systéme, vo svojej najzákladnejšej podobe, predpokladá, že existuje nekonečný cyklus, ktorý je prerušovaný prichádzajúcimi prerušeniami. Tie sú následne obsluhované. Obsluhu prerušenia tvorí priamo konkrétna úloha. Priorita úloh je daná prioritou jednotlivých prerušení (tá obyčajne býva priamo realizovaná hardvérovo). Takýto systém potom tvorí základ pre ďalšie odvodené preemptívne systémy.

Ak sa vyskytne opätovné prerušenie počas spracovávania predchádzajúceho prerušenia toho istého druhu, tak je možné prerušiť aktuálnu obsluhu, len ak je kód prerušenia reentrantný (vlastnosť zabezpečujúca možnosť simultánneho spustenia toho istého kódu vo viacerých kontextoch). Ak je táto vlastnosť splnená, môžeme uvažovať aj vnáranie jednotlivých prerušení.

Jadrá realizujúce FBS model

Systémy s úlohami v popredí a pozadí (FBS – Foreground Background System) sú odvodené od základného modelu systému založeného na prerušeniach. Rozdielom je využitie hlavného cyklu na vykonávanie užitočnej práce, ktoré nazývame pozadím. V pozadí sú spúšťané väčšinou nízkoprioritné úlohy, ktoré budú spustené v prípade, že žiadne prerušenie nie je nutné práve obsluhovať.

Jadrá realizujúce TCB model

Podstatou TCB modelu je priradenie bloku informácií ku každému procesu. Takýto blok obsahuje záznamy, ktoré reprezentujú danú úlohu. Je to jednak kontext, ktorý môže byť do TCB bloku uložený a ďalšie dôležité a individuálne informácie pre danú úlohu (identifikátor, priorita, stav a podobne). V systéme potom existuje zoznam TCB záznamov pre všetky úlohy, z ktorého sa potom vyberá vždy jedna, ktorá bude spustená (bude jej priradené CPU). Toto rozhodovanie vykonáva práve plánovač.

V systéme môže existovať niekoľko takýchto zoznamov pre rôzne stavy úloh. Štandardne je to zoznam úloh pripravených k spusteniu a zoznam úloh, ktoré sú pozastavené, alebo čakajú na určitú udalosť (napríklad príchod prerušenia od periférneho zariadenia, odblokovanie prístupu do kritickej sekcie pri používaní synchronizačných mechanizmov, uplynutie nastavenej hodnoty časovača a podobne). Plánovač je potom zodpovedný aj za premiestňovanie úloh z jednotlivých zoznamov na základe splnených podmienok pre prechod. Tým sa tento typ jadier odlišuje od klasického modelu využívajúceho prerušovací mechanizmus. Plánovač v tomto prípade tvorí centrálnu rozhodovaciu jednotku. Taktiež, uchovávanie

informácií o jednotlivých úlohách (kontext), ich prioritizácia a celkovo plánovací mechanizmus je plne v softvérovej kompetencii. Tento model sa používa aj v klasických operačných systémoch a v prostredí RTOS ponúka najväčšiu mieru flexibility.

Plánovacie rozhodnutia sa môžu vykonávať v periodických časových okamihoch, napríklad použitím časovača, po ktorého uplynutí je generované prerušenie. Taktiež môžu byť tieto rozhodnutia vykonané po zmene stavu jednotlivých úloh.

3.1.2 Komunikačné a synchronizačné nástroje

V prostredí systému s viacerými úlohami a ich paralelným (resp. pseudoparalelným) spracovaním, kde jednotlivé úlohy môžu pristupovať k jednej dátovej položke, je potrebné riešiť aj zachovanie konzistencie dát. Ďalším prípadom môže byť situácia, kde určitá úloha by mala byť synchronizovaná s behom inej úlohy. Taktiež môže byť kladená požiadavka na možnosť priamej komunikácie medzi jednotlivými procesmi a možnosť zasielania správ alebo krátkych dátových položiek. Ide o tzv. synchronizačné a komunikačné nástroje, ktoré sú realizované priamo operačným systémom, pretože ten tvorí prostredie, v ktorom sú jednotlivé úlohy spúšťané. V prípade RT operačných systémov môžeme nachádzať podobné požiadavky na vybavenie.

Globálne premenné

V rámci komunikácie medzi jednotlivými úlohami môžeme uvažovať niekoľko prístupov, ktoré zabezpečujú zasielanie a príjem dát (resp. správ). Najjednoduchším spôsobom je použitie *globálnych premenných*. Tento spôsob sa môže využívať v najjednoduchších RT systémoch, hlavne kvôli rýchlosti a jednoduchosti realizácie [11]. Nevýhodou však je, že môže existovať úloha, ktorá bude dáta ponúkať nepravidelným spôsobom tak, že prijímajúca úloha nebude stíhať čítať obsah a tak môže dochádzať k stratám.

Vyrovnávacie pamäte

Robustnejší systém pre zasielanie správ a dát medzi úlohami je založený na *vyrovnávacích pamätiach* (buffer). Tento prístup rieši problém v rozdieloch medzi rýchlosťami zasielania dát a ich následným príjmom (resp. vyrovnáva nepravidelnosti v rýchlosti dodávania dát prijímajúcej úlohe). Existuje niekoľko prístupov pre realizáciu vyrovnávacích pamätí, ako napríklad klasická vyrovnávacia pamäť s definovaným maximálnym počtom položiek (buffer), dvojitá vyrovnávacia pamäť (double buffer) a kruhová vyrovnávacia pamäť (ring buffer) [11, 14].

Poštové schránky

Pre zasielanie správ existujú tzv. *poštové schránky* (mailbox). Tento komunikačný mechanizmus je založený na definovanom spoločnom priestore, cez ktorý si jednotlivé úlohy vymieňajú správy. Pri jeho použití nemusí úloha aktívne čakať (čakanie na prichádzajúcu hodnotu s neustálym testovaním v cykle). Ak sa v schránke nenachádza správa, je úloha automaticky uvedená do čakajúceho (neaktívneho) stavu a jej beh sa obnoví až po príchode novej správy. Ak implementácia schránok povoľuje spracovanie viacerých možných požiadavok, tak potom hovoríme o tzv. *frontách* (queues) [11].

Semafory, deaktivácia prerušení

Pre synchronizačné účely sa na úrovni operačného systému realizuje štandardný synchronizačný nástroj *semafor* v rôznych podobách, ako *binárny semafor*, *mutex* a *počítací semafor*.

V krajnom prípade, výhradný prístup do kritickej sekcie môžeme zaručiť aj *deaktivovaním prerušení* na určitý krátky čas. Tým zamedzíme preempcii aktuálnej úlohy, takže nebude dochádzať k zmene citlivých dát a ich následnej nekonzistencii. V prostredí RT systémov ale musíme vždy zvážiť, či takýto krok nepovedie ku chybám funkcionality systému ako celku, pretože môžu byť stratené prerušenia a tým aj vykonanie dôležitých vysokoprioritných úloh.

3.2 Model RT úloh

Model RT úloh určuje konkrétne parametre, ktoré definujú ich časové vlastnosti a limity, zároveň sa používajú ako vstupné parametre v rámci RT plánovacích algoritmov. Tieto parametre môžeme rozdeliť do dvoch skupín primárne a dynamické parametre [2, 14].

Primárne parametre sú nemenné počas behu úlohy, majú teda statický charakter. Medzi tieto parametre patrí:

- čas požiadavky na spustenie úlohy (r),
- maximálna doba behu úlohy (C),
- relatívny časový limit úlohy (D),
- absolútny časový limit úlohy ($d = r + D$),
- pre periodické úlohy aj perióda jej volania (T).

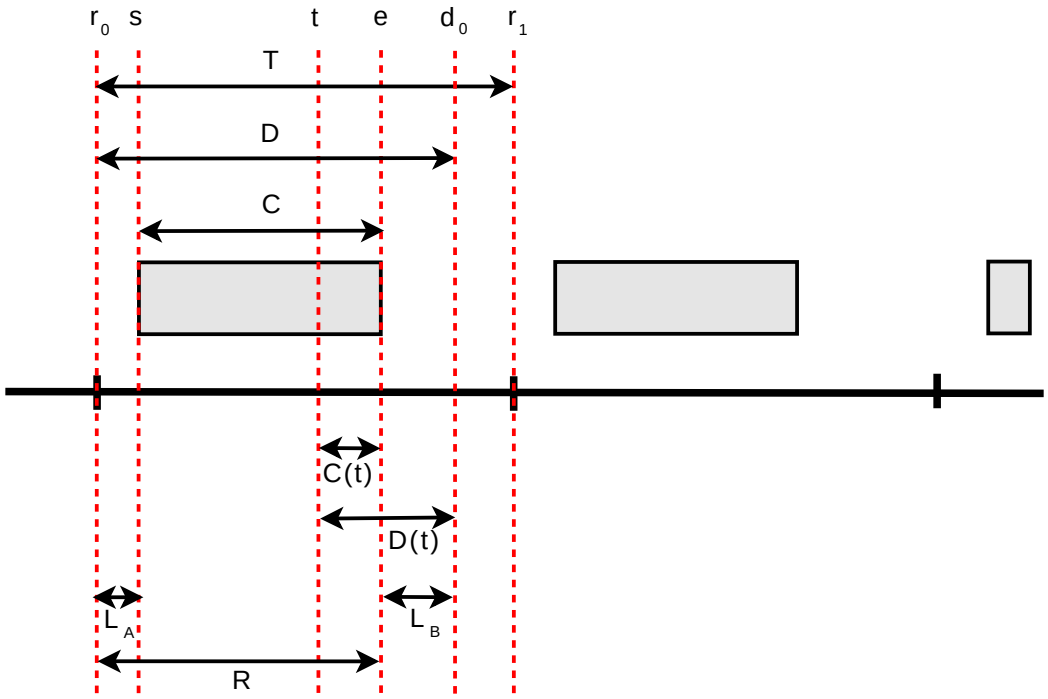
V prípade, že uvažujeme periodické úlohy, tak potom r_0 je čas prvej požiadavky na spustenie úlohy a $r_k = r_0 + kT$ čas k. požiadavky na spustenie úlohy. Podobne, d_0 je absolútny časový limit pre prvý beh úlohy a $d_k = d_0 + kT$ absolútny časový limit pre k. beh úlohy. Pre správne formulované primárne parametre úlohy musí platiť $0 < C \leq D \leq T$.

Z uvedených primárnych parametrov je možné odvodiť hodnotu činiteľa využitia CPU úlohou u a činiteľa zaťaženia CPU úlohou ch:

- činiteľ využitia CPU úlohou $u = \frac{C}{T}$,
- činiteľ zaťaženia CPU úlohou $ch = \frac{C}{D}$.

Hodnoty dynamických parametrov sa menia počas behu úlohy a reprezentujú stav behu úlohy. Medzi tieto parametre patrí (viď taktiež obr. 3.1):

- čas počiatku vykonávania úlohy, s,
- čas ukončenia vykonávania úlohy, e,
- zostávajúci relatívny časový limit úlohy v čase t, $D(t) = d - t$, pričom $0 \leq D(t) \leq D$,
- zostávajúci čas do konca vykonávania úlohy v čase t, $C(t)$, pričom $0 \leq C(t) \leq C$,
- relatívna doba voľnosti úlohy, $L = D - C$,
- zostávajúca relatívna doba voľnosti úlohy v čase t, $L(t) = D(t) - C(t)$,
- doba odozvy úlohy, $R = e - r$,
- zostávajúce zaťaženie, $CH(t) = \frac{C(t)}{D(t)}$, pričom $0 \leq CH(t) \leq (\frac{C}{T})$.



Obr. 3.1: Model RT úloh

3.3 Plánovanie RT úloh

Pre plánovanie úloh v prostredí s RT požiadavkami boli vypracované špecializované algoritmy, ktoré sa snažia brať do úvahy definované časové limity úloh. RT plánovacie mechanizmy môžeme rozdeliť na základe niekoľkých podmienok. Jednou z nich je aj to, kedy dochádza k samotnému plánovaniu úloh. Na základe tejto podmienky odlišujeme *off-line plánovanie* (plánovanie je vykonávané len jeden raz, a to pred spustením úloh) a *on-line plánovanie* (plánovanie je vykonávané počas behu úloh) [14].

Systém ako celok nemusí pozostávať len z jedného uzlu. Ak chceme v takomto prostredí vykonávať plánovanie, tak odlišujeme *centralizované plánovanie* a *distribuované plánovanie*. V prípade centralizovaného plánovania je za vytváranie plánu zodpovedný jeden centrálny uzol. V prípade distribuovaného ide o súčinnosť viacerých lokálnych plánovacích mechanizmov, ktoré medzi sebou komunikujú (resp. vymieňajú si úlohy) a vytvárajú tým globálny plán. V takomto prípade odlišujeme lokálny plánovač (plánovač, ktorý pracuje len v rozmedzí jedného uzlu) a globálny plánovač (vykonáva plánovanie pre systém uzlov na globálnej úrovni).

Ďalšie delenie plánovacích mechanizmov sa týka možnosti prerušenia aktuálne vykonávanej úlohy a spustenie inej úlohy: *preemptívne plánovanie* a *nepreemptívne plánovanie*. V prípade, že pre dané úlohy existujú priority, tak hovoríme o *prioritnom plánovaní*, v opačnom prípade o *neprioritnom plánovaní*. Jednotlivé priority v rámci prioritného plánovania môžu byť buď *statické* alebo *dynamické*. Statické priority sú odvodené od statických parametrov, ktoré sú známe ešte pre spustením úlohy. Naopak, dynamická priorita v sebe zahŕňa aj dynamické vlastnosti úlohy a táto priorita je počas behu menená na základe použitého plánovacieho algoritmu.

V kontexte s plánovaním úloh v RT prostredí je používané označenie pre tzv. *prípustný*

plán. Plán ma túto vlastnosť, ak zabezpečuje dodržanie časovým limitov, ktoré sú určené pre RT úlohy. Ak takýto plán existuje, tak daná množina úloh je potom *plánovateľná*. Ak používaný plánovací algoritmus nájde prípustný plán pre ľubovoľnú množinu úloh, tak je *optimálny*. *Testom plánovateľnosti* sa potom určuje, či daná množina úloh je vôbec plánovateľná. Ak v systéme môžu byť spustené kedykoľvek nové úlohy, tak určenie, či pridanie tejto úlohy do aktuálnej skupiny úloh budú stále splnené časové limit, nazývame *testom prijatia* [14].

Existujú dve základné realizácie plánovačov – *volebná tabuľka* a *prioritne usporiadaný zoznam* [14]. Volebná tabuľka sa používa v prípade, že plán je pevne zadaný. To znamená, že tento prístup sa používa v prípade off-line plánovania. Volebná tabuľka je statická štruktúra, ktorá obsahuje informácie o tom, ktorá úloha bude spustená v určitom okamihu. Prioritne usporiadaný zoznam je určený hlavne pre prípad on-line plánovacích mechanizmov. Zoznam umožňuje dynamicky meniť usporiadanie jeho prvkov, resp. zloženie, čo zodpovedá dynamickým zmenám priorít, resp. spustením nových úloh v systéme.

3.3.1 Základné plánovacie mechanizmy

V súčasnosti sa najviac využívajú prioritné on-line plánovacie mechanizmy. Tieto sú určené hlavne pre periodické úlohy, ktoré sú v prostredí RT systémov bežné. Existujú však vypracované algoritmy aj pre plánovanie hybridných úloh, ktoré pozostávajú jednak z periodických, ale aj aperiodických úloh. V nasledujúcom texte bude uvedený len stručný prehľad základných algoritmov. Detailnejší popis a analýza vlastností bude už súčasťou naväzujúcej práce.

Plánovanie množiny periodických úloh

Základným a najznámejším algoritmom pre plánovanie množiny periodických úloh je *algoritmus RM* (Rate Monotonic). Tento je založený na statických prioritách, ktoré sú priradené úlohám od počiatku na základe známej hodnoty periódy ich volania. Čím je volanie danej úlohy častejšie (perióda je kratšia), tým je úlohe priradená aj väčšia priorita. Tento algoritmus je určený pre úlohy, ktorých relatívne časové limity sa rovnajú ich periódam, teda $D = T$ podľa modelu RT úloh uvedeného v sekcii 3.2 (je optimálny pre tento uvažovaný prípad).

Algoritmus DM (Deadline Monotonic) tiež pracuje so statickými prioritami. Tie sú, na rozdiel od predchádzajúceho algoritmu, priradené na základe časového limitu danej úlohy. Čím je relatívny časový limit úlohy kratší, tým je priorita úlohy väčšia. Algoritmus je optimálny aj pre prípad, že relatívne časové limity úloh sú menšie než ich periódy, teda $D < T$.

Ak uvažujeme zmeny priorít aj počas behu úloh, tak potom v tejto triede RT mechanizmov plánovania môžeme nájsť *algoritmus EDF* (Earliest Deadline First). Algoritmus priraďuje priority úlohám na základe hodnôt ich zostávajúcich relatívnych časových limitov $D(t)$. Úloha s najmenším relatívnym časovým limitom bude mať najväčšiu prioritu. Prepočítavanie (priraďovanie) priorít jednotlivých úloh sa vykonáva v čase volania novej úlohy.

Ďalším algoritmom pracujúcim s dynamickými prioritami je *algoritmus LLF* (Least Laxity First). V tomto prípade sa však uvažuje hodnota zostávajúcej relatívnej doby voľnosti úlohy $L(t)$. Čím je táto doba menšia, tak bude úlohe priradená väčšia priorita. Prepočítavanie priorít sa vykonáva, podobne ako v predchádzajúcom prípade, v čase volania novej úlohy.

Plánovanie hybridnej množiny úloh

Medzi elementárne algoritmy pre plánovanie hybridných množín úloh patrí *plánovanie v pozadí*, *plánovanie pomocou využitia serverov*, *plánovanie s využitím voľnosti cudzích úloh* (slack stealing) a *spoločné EDF plánovanie periodických a aperiodických úloh.*

Plánovanie v pozadí je založené na jednoduchom princípe, kde aperiodické úlohy sú spustené len v prípade, že neexistuje v danom momente periodická úloha, ktorá by mala byť vykonávaná. Periodické úlohy sú plánované niektorým z algoritmov určených pre tento typ úloh.

Plánovanie využitím serverov je založené na myšlienke, pri ktorej je pre spúšťanie aperiodických úloh vyhradená určitá kapacita času CPU. Existuje niekoľko variant tohto typu plánovania aperiodických úloh:

- výzývací server
- odkladací server
- sporadický server
- server s výmenou priorít

Kapitola 4

Metódy časových analýz

Jedna z uvažovaných zložiek v rámci analýz real-time systémov a použitých plánovacích algoritmov je aj *celkový čas CPU potrebný pre vykonanie danej úlohy.* Tento celkový čas vykonávania úlohy (resp. jej časti) ale nemusí byť v reálnych podmienkach konštantný, pretože môže závisieť od typu prostredia, v ktorom je systém umiestnený a od rôznych podmienok, na základe ktorých sa vykonáva výpočet a riadenie v rámci úlohy. Z tohto pohľadu môžeme hovoriť o *toku riadenia v programe na základe hodnôt vstupných dát.* Vstupné dáta môžu byť práve hodnoty získané externými senzormi. V tomto prípade je potom nutné uvažovať najhoršiu možnú situáciu (trajektóriu vo vykonávaní kódu programu), ktorú označujeme ako *WCET (Worst Case Execution Time).* Hodnotu WCET následne používame v miestach výpočtov, kde je požadovaná informácia o celkovej časovej náročnosti. Potom pri použití známych a overených real-time plánovacích algoritmov je možné zabezpečiť, že úloha bude spĺňať dané časové kritériá aj v tých najkrajnejších prípadoch. Primárnym cieľom je teda horné ohraničenie časovej náročnosti systému alebo jeho časti pri uvažovaní všetkých možných vstupných podmienok, ktoré majú vplyv na tok programu.

Problematika výpočtu WCET je pomerne rozsiahla a tvorí jednu z podstatných častí analýz real-time systémov. Existujú *dva základné prístupy pre získanie hodnôt WCET*, prípadne ich aproximácií. Prvým z nich je využitie tzv. *statickej analýzy.* Tá je založená na formálnych metódach pre analýzu programu bez jeho reálneho spustenia. Analýza sa vykonáva na úrovni kódu a to buď pri uvažovaní *zdrojového kódu* zapísaného vo vyššom programovacom jazyku, alebo na úrovni *strojového kódu* s inštrukciami, tzv. objektového kódu (menej obvyklý spôsob). Zdrojový kód v niektorých prípadoch môže obsahovať aj špecializované *anotácie.* Ich účelom je dopomôcť pri vykonávaní a celkovom postupe automatizovaných statických analýz v častiach kódu, ktoré by mohli spôsobovať problémy vo výpočtoch, prípadne v častiach, ktoré by mohli byť zdrojom skreslených výsledkov bez ich uvedenia a uvažovania v rámci postupu analýzy. Primárnym účelom anotácií je teda explicitné uvedenie známych parametrov a limitov v konkrétnych bodoch daného programu. Anotácie sú zadávané manuálne programátorom.

Druhým prístupom je tzv. *dynamická analýza* kódu, nazývaná aj *analýza na základe meraní.* V tomto prípade ide priamo o skúmanie časovej náročnosti spusteného systému. Využívané metódy sú založené na *trasovaní* reálneho systému vykonaním vyhradených funkcií pre zaznamenávanie postupu, alebo sú založené na *simulácií* systému. V oboch prístupoch je dôležitou súčasťou zaznamenávanie časových značiek výskytu udalostí, ktoré sú spúšťané vykonaním určitej časti kódu, prípadne konkrétnym prístupom do určenej oblasti pamäte. Je taktiež možné využiť aj priame postupy s využitím *logických analyzátorov* alebo *osciloskopov* a priamo takto sledovať výskyt signálov, ktoré zodpovedajú vykonaniu konkrétnej časti kódu v rámci programu.

4.1 Možné problémy časových analýz

Pri vykonávaní analýzy, či už ide o statický alebo dynamický prístup, sa môžeme stretnúť s viacerými problémami. Zdroj [10] uvádza tieto základné komplikácie, ktoré sa objavujú pri vykonávaní analýz kódu:

- problém nájdenia cesty,
- problém stavov,
- problém prekladu.

V rámci využitia statických analýz sa snažíme v počiatočných krokoch celkového postupu nájsť všetky možné cesty, z ktorých potom neskôr následne vyberáme tú najdlhšiu možnú. Tá nám bude určovať hornú hranicu vykonávania kódu – WCET. V tomto prípade ale môžeme naraziť na problém veľkého počtu ciest, ktoré môžu narastať exponenciálne spolu s veľkosťou programu [10].

Podobne je to aj v prípade dynamickej analýzy, kde ale tento problém má podobu zložitosti nájdenia vhodných vstupných testovacích dát, ktoré by spustili najdlhšiu možnú cestu počas vykonávania programu.

Ďalší problém, tzv. problém stavov, súvisí s počítaním a následným sčítavaním času vykonávania jednotlivých častí kódu. Tu sa môžeme stretnúť s komplikáciami, ktorých zdrojom je komplexnosť použitých hardvérových prostriedkov, predovšetkým CPU, ale aj rôznych uvažovaných periférnych zariadení. V tomto prípade je nutné brať do úvahy aj možnosť závislosti času vykonávania kódu na histórií. Ide vlastne o uvažovanie kontextu, v ktorom sa daný analyzovaný úsek nachádza. Dôvodom a zdrojom takejto závislosti na kontexte a histórií danej časti kódu sú hlavne optimalizačné hardvérové techniky, ako napríklad použitie rôznych vyrovnávacích pamätí, využitie techník predpovedania vetvení, technika pipelines, špekulatívne vykonávanie kódu a podobne.

Zdroj [10] v tomto prípade uvádza tzv. časovo závislý stav procesoru – TRPS (timing relevant processor state), pričom stav procesoru, označovaný skrátene PS (processor state), je definovaný ako stav všetkých relevantných pamäťových prvkov. TRPS obsahuje zoznam častí PS, ktoré majú vplyv na vykonávanie aspoň jednej z uvažovaných inštrukcií aktuálneho kontextu. V prípade, že použitý procesor sa nevyznačuje takouto časovou závislosťou, je analýza jednoduchšia, pretože vo výpočtoch v rámci statickej analýzy sa použijú priame nemenné hodnoty časov vykonávania inštrukcií (ktoré v tomto prípade je obyčajne možné nájsť aj v manuále daného CPU). Podobne je to aj v prípade dynamickej analýzy, kde ekvivalentom situácie je také vykonávanie programu na základe testovacích dát, že nebude záležať na poradí, v akom sa budú tieto vstupné podnety objavovať a konečné časy vykonávania budú vždy rovnaké. V opačnom prípade musí analýza s takýmito vlastnosťami počítať. Tým sa ale sa proces značne komplikuje, pretože počas analýzy kódu sa musí uchovávať príslušný kontext, aby bolo možné vypočítať čas vykonávania budúcich inštrukcií (a tým vlastne simulovať TRPS). V prípade dynamickej analýzy by bolo nutné odmerať všetky možné kombinácie vstupných podnetov, ktoré majú vplyv na tok programu.

Predchádzajúce poznatky môžeme zhrnúť do podoby, že ak súčasný stav závisí od predchádzajúceho stavu (resp. stavov), môžeme sa dostať do oblasti časových anomálií. Tie porušujú intuitívny predpoklad, že lokálne vypočítané časy WCET pre jednotlivé bloky programov oddelene povedú po ich sčítaní ku globálnemu času WCET [18].

Problém prekladu súvisí s rozdielnym výsledkom kompilácií zdrojových kódov pri použití rôznych kompilátorov. Dôvodom je fakt, že každý kompilátor môže obsahovať rozdielne

optimalizačné techniky, ktoré majú vplyv na výslednú kvalitu kódu z časového hľadiska. To znamená, že získať úplne presné časové informácie analýzou len na základe príslušného zdrojového kódu je nemožné bez vedomosti o použitom kompilátore. Situáciu je možné v tomto prípade zjednodušiť vypnutím optimalizácií, ale tým sa dobrovoľne vzdáme výhod plynúcich z ich použitia.

Dynamické analýzy sú z tohto pohľadu výhodnejšie, pretože analýza sa realizuje priamo vykonávaním konkrétneho strojového kódu, ktorý už je výsledkom kompilácie. Samozrejme, je to len v prípade, ak by sa spustenie alebo simulácia vykonávala priamo, teda nie na zdrojovom, ale strojovom kóde. V opačnej situácií sa stretávame s podobným problémom, ako pri statickej analýze. Dynamickú analýzu na zdrojovom kóde si môžeme predstaviť ako interpretáciu toku programu priamo zo zdrojového kódu na základe vstupných dát. Takýto prístup už ale hraničí s počiatkami statickej analýzy a je otázne uvažovať o dynamickej analýze v pravom slova zmysle.

4.2 Statická analýza

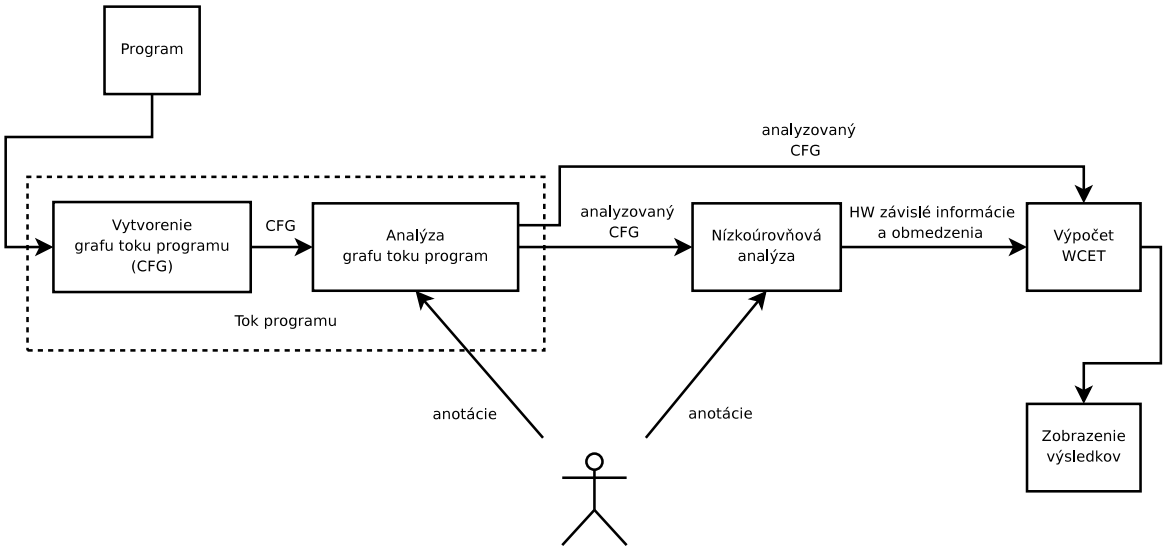
Statická analýza sa zameriava na výpočet horného ohraničenia doby výpočtu programu (WCET) s využitím matematických modelov. Toto ohraničenie zavisí predovšetkým od vstupných dát, celkovej logiky programu, časových vlastností a odoziev použitých hardvérových prostriedkov [9]. Všetky tieto prvky, ktoré majú nezanedbateľný vplyv na výsledný čas WCET, je potrebné zahrnúť v analýze. Tá pozostáva štandardne z týchto krokov [9, 13, 18] (viď taktiež obr. 4.1):

- vytvorenie grafu toku programu (CFG – control flow graph),
- analýza toku programu,
- nízkoúrovňová analýza,
- konečný výpočet (aproximácie) hraničného času WCET,
- prípadná vizualizácia výsledkov, zobrazenie štatistík.

Graf toku programu sa vytvára zo samotného kódu daného programu (táto prvotná časť analýzy sa v angl. literatúre označuje aj ako *frontend*). Vytvorený graf pozostáva zo *základných stavebných blokov programu* (v angl. literatúre označované ako *basic blocks*). Tie obsahujú priame kódové sekvencie bez opakovania a vetvenia. Graf taktiež zachytáva *body vetvenia*. Základné bloky sú potom reprezentované uzlami v grafe a vetvenie jednotlivými spojnicami medzi týmito blokmi. Graf obyčajne obsahuje aj doplnkové informácie o maximálnom počte iterácií a možné intervaly vstupných hodnôt, ktoré sú získané z dodaných anotácií v rámci zdrojového kódu, alebo sú tieto informácie vpisované priamo a manuálne do grafu počas jeho vytvárania.

Vytvorený graf CFG je potom vstupom pre následnú *analýzu toku programu* (CFA – Control-Flow Analysis). Jej účelom je určiť možné trajektórie v programe a odstrániť tie, ku ktorým nemôže za žiadnych podmienok dôjsť (vzájomne vylučujúce sa podmienky).

Množina ciest v rámci toku programu je vždy konečná, pretože musí byť garantované ukončenie. Všeobecne sa ale úplná množina ciest nedá určiť, ale každá nadmnožina týchto ciest bude vždy určite bezpečnou aproximáciou. Čím menšia bude táto nadmnožina ciest, tým lepšie. Čas vykonávania každej cesty, ktorá bola bezpečne eliminovaná, môžeme ignorovať v rámci výpočtov WCET, pretože nebude mať vplyv na celkovú výslednú hodnotu



Obr. 4.1: Základné kroky statickej analýzy

času vykonávania. Výsledok analýzy toku programu môžeme chápať aj ako zistené obmedzenia pre dynamiku úlohy. To v sebe zahŕňa informácie o tom, ktoré funkcie môžu byť volané, aké sú závislosti medzi podmienkami v kóde a informácie o realizovateľnosti, resp. nerealizovateľnosti (alebo vhodnosti) danej cesty [18]. Všetky tieto informácie sú dôležitou súčasťou pre následujúce kroky analýzy. Účelom je vytvoriť potrebné zjednodušenia tak, aby bol vytvorený určitý model toku programu a neanalyzovali sa zbytočné cesty, ku ktorým v reálnych podmienkach nikdy nedochádza.

Nízkoúrovňová analýza je časť statickej analýzy zameranej na vlastnosti použitých hardvérových prostriedkov, predovšetkým CPU, ale aj časy prístupu do pamäte, prípadne uvažovanie vyrovnávacích pamätí, u zložitejších procesorov aj predpovedania vetvení (branch predictions) a tzv. pipelines. Z tohto pohľadu sa môže analýza značne skomplikovať, pretože pri zložitejších hardvérových optimalizačných technikách určených všeobecne pre zvýšenie výkonu sa analýza stáva závislou na kontexte a histórii predtým vykonaných častí programu. V takýchto prípadoch by bolo veľmi obtiažne používať model v rámci automatizovaných analýz, ktorý by plne zodpovedal reálnemu hardvéru. Preto sa vo väčšine podobných situácií využívajú rôzne aproximácie a abstraktné modely.

Ak sú k dispozícií informácie z predošlých krokov statickej analýzy, môžeme pristúpiť k samotnému *výpočtu hraničného času WCET*. Vstupné informácie tvorí vlastne množina *faktov o toku daného programu* (ohraničenia pre cykly, nerealizovateľné alebo nevyhovujúce cesty v rámci CFG, závislosti pre beh programu a podobne) a celkový efekt hardvérových súčastí na čas vykonávania programu, ako napríklad efekt vyrovnávacích pamätí, pipeline a iných hardvérových čŕt [4]. Existujú tri základné metódy, ktoré sa v súčasnosti pre účel výpočtu WCET využívajú [4, 13]:

- metóda založená na *syntaktických stromoch* (angl. tree-based), taktiež metóda založená na *štruktúrach*,
- metóda založená na *cestách* (angl. path-based),
- metóda založená na *implicitnom počítaní ciest* (IPET, z angl. implicit path enumeration technique).

4.3 Dynamická analýza

Základne body dynamickej analýzy môžeme charakterizovať aj pomocou porovnania so statickou analýzou. Dynamická analýza tvorí totižto komplement ku statickej analýze, ak uvažujeme tieto body [1]:

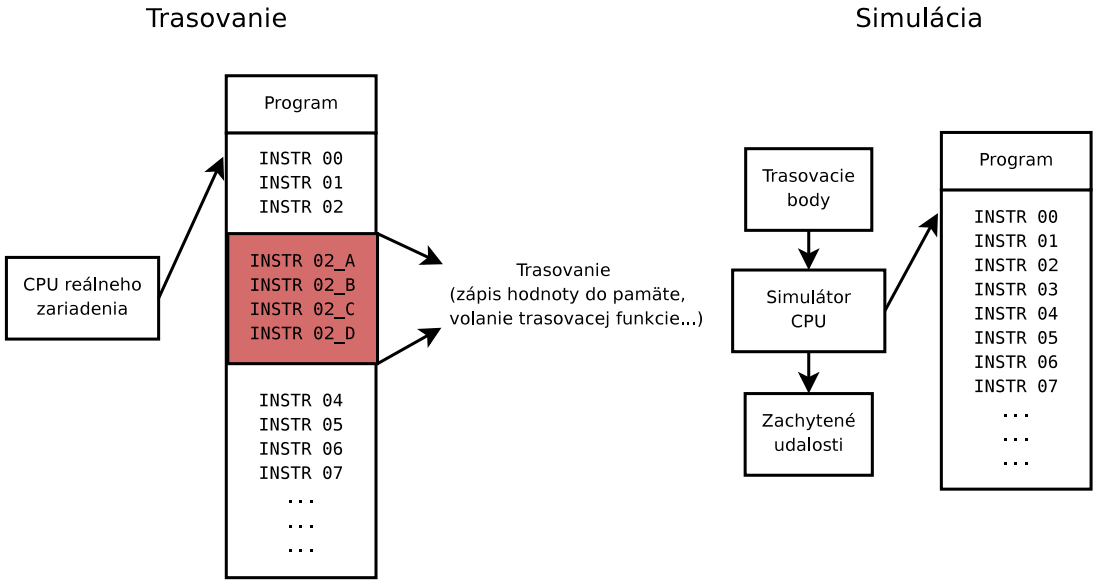
- úplnosť,
- rozsah,
- presnosť.

Úplnosť súvisí s uvažovaním jednotlivých ciest v toku daného programu. V rámci statickej analýzy sa uvažujú všetky možné cesty, pričom v dynamickej analýze je vlastnosť úplnosti obmedzená, pretože závisí od rôznorodosti súboru vstupných testovacích dát. V prípade, že tento súbor nebude dostatočný, je možné, že bude dochádzať k nepresnostiam vo výpočtoch. Môže nastať situácia, že segment cesty, ktorý je súčasťou najdlhšieho toku, nebude započítaný (výsledkom potom budú príliš optimistické časové ohraničenia). Naopak, statická analýza môže ponúkať chybné výsledky práve na základe uvažovania veľkého množstva ciest, pričom niektoré z nich môžu byť nerealizovateľné v reálnych podmienkach. Takáto chyba sa bude samozrejme propagovať ďalej vo výpočtoch v rámci analýz, ak naším cieľom je práve výpočet WCET (bude dochádzať k príliš pesimistickým časovým ohraničeniam).

Rozsah určuje dĺžku v rámci cesty, ktorú je daná metóda schopná analyzovať. Statická analýza je obmedzená iba na rozsah programu, ktorú je metóda schopná analyzovať efektívne v reálnom čase a môže mať problémy pri uvažovaní dlhšej trajektórie v kóde [1]. Keďže dynamická analýza vykonáva priamo kód programu, je v tomto prípade situácia jednoduchšia, pretože môžeme zaznamenať trajektóriu po celej dĺžke.

Presnosť sa vzťahuje k miere abstraktnosti metódy. Analýza vykonávaná spúšťaním reálneho kódu, na rozdiel od metód statických analýz, v sebe neobsahuje žiadnu abstrakciu, čím dostávame výsledky, ktoré sú v podstate presné v maximálnej možnej miere. Problémom v tomto prípade je, ako už bolo popisované, nájsť takú množinu vstupov, ktorá by vhodne otestovala tie najpodstatnejšie cesty, ktoré majú vplyv na určitý sledovaný parameter (akým je práve napríklad aj WCET). Je možné využiť aj kombináciu so statickou analýzou, ktorá nám dopomôže k nájdeniu vhodných testovacích vstupných dát. Prípadne je možné takýto súbor vstupných dát vytvoriť manuálne na základe samostatnej prehliadky zdrojového kódu.

Dynamickú analýzu môžeme vykonávať viacerými spôsobmi. Najpoužívanejšími sú *trasovanie kódu* a *simulácia kódu* s využitím simulátorov spolu s integrovaným trasovaním (obr. 4.2). Pri využití trasovania sa v určitých sledovaných miestach v programe pridávajú trasovacie funkcie, ktorých účelom je zachytiť sledovaný stav systému, resp. jeho časti, v momente, keď sa tok programu dostane do miesta tejto funkcie. Výhodou popisovaného prístupu je jednoduchosť, pretože ide v podstate len o samotné pridanie trasovacích funkcií do kódu. Nevýhodou je predovšetkým fakt, že aj samotná trasovacia funkcia je súčasťou programu (na obr. 4.2 vyznačená červenou farbou, dodatočné inštrukcie `INSTR 02_A` až `INSTR 02_B`). Ak sa snažíme získať časové údaje, ktoré sa využívajú ako vstup pre následné analýzy, môžu byť tieto nesprávne, pretože vložené funkcie majú vplyv na výsledné časy. Taktiež, v reálnom nasadení sú tieto trasovacie funkcie odstránené. Môže dochádzať k odchýlkam v správaní systému ako celku, ak trasovanie pozmení časovanie systému natoľko, že bude spôsobovať nezanedbateľné zmeny v rámci globálneho stavu systému. Analýzy sa budú vykonávať na základe nesprávnych časových informácií, ktoré v reálnom prostredí a nasadení môžu byť úplne iné spolu s príslušným správaním sa systému.



Obr. 4.2: Dva spôsoby vykonávania dynamickej analýzy

Alternatívou je simulácia kódu s využitím simulátorov spolu s integrovaným trasovaním, prípadne s podporou pre záchytné trasovacie body. V tomto prípade sa program nespúšťa na reálnych hardvérových prostriedkoch, ale simuluje hlavne s využitím softvérových súčastí. Výhodou je väčšia kontrola nad procesom dynamickej analýzy. Je možné pozastaviť proces simulácie na určitú dobu, meniť parametre za behu simulácie, trasovať tok programu, resp. zapisovať do pamäťových miest nové hodnoty a automaticky pozastaviť program pri výskyte danej podmienky, prípadne synchronizovať vykonávanie programu s príslušnými riadkami zdrojového kódu (podobne ako je to pri ladení programov). Tým dostávame jasnejšiu predstavu o behu systému. Nezáleží tu na reálnom čase vykonávania, ale do úvahy sa berie logický čas (čas, ktorý simulátor počíta interne na základe výskytu inštrukcií v programe, prípadne na základe prístupu k periférnym zariadeniam s dopredu známymi prístupovými časmi a časmi spracovania požiadavok). Pri simulácií s podporou trasovacích bodov sa tieto načítajú zo vstupného súboru a pri výskyte danej udalosti (napríklad prečítanie danej bunky pamäte a podobne) sa táto zapíše s rôznymi informáciami do výstupného súboru (napríklad informácie o stave procesoru, čase výskytu danej udalosti a podobne).

Hlavnou nevýhodou takéhoto prístupu je jeho aplikovateľnosť len na jednoduchšie hardvérové prostriedky, ktoré sú ľahšie simulovateľné (napríklad procesory bez predikcie vetvení, bez vyrovnávacích pamätí, pipelines a podobne, taktiež hardvérové prostriedky s pevnými časmi spracovania inštrukcií). V opačnom prípade bude simulátor do určitej miery abstrakciou a vhodnosť jeho použitia bude záležať od detailnosti simulátoru a od povahy cieľových informácií, ktoré chceme získať. Ak naším cieľom bude získať len informácie o toku daného programu, je možné brať do úvahy aj simulátor, ktorý je aproximáciou a dokáže na vstupy reagovať tak isto ako reálny systém, hoci sa neberú do úvahy časové vlastnosti. Ak však ide o časovú analýzu, bude pre nás určite dôležité, aby takýto simulátor obsahoval časové závislosti daného konkrétneho simulovaného zariadenia, ktoré sú takmer presnou kópiou reálneho systému. Na to je však potrebné, aby model systému obsahoval detailné informácie o časových špecifikáciách použitých hardvérových prostriedkov a časoch celkovo.

Kapitola 5

Simulátor mikrokontroléru MSP430

Mikrokontrolér MSP430 v sebe zhŕňa jednoduchý RISC procesor. Ten sa vyznačuje *konštantnými časmi spracovania inštrukcií*. To znamená, že čas spracovania nie je závislý na kontexte, v ktorom sa daný blok programu vykonáva a nie je závislý ani na histórií behu predchádzajúcich častí programu (neobsahuje vyrovnávaciu pamäť cache, ani žiadnu z pokročilých metód pre akceleráciu a optimalizáciu práce CPU). Informácie o počte cyklov potrebných pre vykonanie jednotlivých inštrukcií sú uvedené priamo v manuále daného mikrokontroléru. Časy spracovania inštrukcií sú variabilné len na základe metódy prístupu k pamäti. Počet metód pre prístup do pamäte je ale konečný, a tak je konečný aj počet možných rôznych časových variácií spracovania jednotlivých uvažovaných inštrukcií. Popisovaná vlastnosť je veľmi dôležitá pre konštrukciu simulátoru. Umožňuje to vytvorenie modelu, ktorý je svojimi časovými vlastnosťami zhodný s reálnym systémom.

5.1 Simulátor MSPsim

Jedným zo simulátorov mikrokontroléru MSP430 je aj MSPsim[1]. Je napísaný v programovacom jazyku Java a primárne určený pre simulácie senzorových sietí založených na danom mikrokontrolére. Zdrojové kódy MSPsim sú voľne dostupné[2]. Licencia je typu BSD, čo nám umožňuje prípadnú modifikáciu zdrojových kódov a doplňanie funkcionality a následné šírenie bez akýchkoľvek reštrikcií. Simulátor je pomerne novým produktom, ktorý je ešte stále vo vývoji, pričom zatiaľ nebola vydaná plne stabilná verzia. V čase písania tohto textu je aktuálna verzia s označením 0.95.

MSPsim je *simulátorom na úrovní inštrukcií* s podporou presnej časovej simulácie, pričom spúšťa nemodifikovaný firmvér danej cieľovej platformy. Je rozšíriteľný o rôzne periférne zariadenia, čím je umožnená simulácia rôznych platforiem založených práve na mikrokontrolére MSP430 [3]. Vo svojej základnej podobe, po získaní zdrojových kódov simulátoru, bude k dispozícií pre používateľa podpora pre platformu ESB[3] a Sky[4].

V súčasnosti podporuje simulátor tieto hardvérové prostriedky:

[1] MSPsim je vyvíjaný na Swedish Institute of Computer Science, prezentáciu projektu je možné nájsť na adrese http://www.sics.se/project/mspsim.

[2] Zdrojové kódy sú voľne k dispozícií na adrese http://sourceforge.net/projects/mspsim.

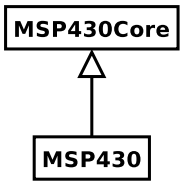
[3] Embedded Sensor Board, prototyp zariadenia pre bezdrôtové senzorové siete, vyvíjaný na Freie Universität Berlin

[4] Tmote Sky, senzorové zariadenie od spoločnosti Sentilla (pôvodne spoločnosť Moteiv)

- CPU (simulácia na inštrukčnej úrovni),
- BCM (basic clock module), časovač A, časovač B,
- USART,
- digitálny I/O,
- jednotka pre násobenie,
- A/D podsystém,
- watchdog.

5.1.1 Základné funkčné prvky

Základ simulátoru tvorí *simulačné jadro CPU* (trieda `MSP430Core`, resp. jeho reprezentácia na vyššej úrovni abstrakcie v podobe triedy `MSP430`). Jeho primárnou úlohou je vytvorenie spojiva medzi jednotlivými základnými súčasťami mikrokontroléru MSP430 (pamäť, moduly a periférie), taktiež interpretácia inštrukcií a počítanie logického času (počtu cyklov od spustenia). Simulácia sa môže vykonávať buď kontinuálne, alebo krokovým spôsobom po jednotlivých inštrukciách, prípadne po niekoľkých inštrukciách naraz. Rýchlosť simulácie je možné nastaviť pomocou parametra faktoru rýchlosti, taktiež je možné proces simulácie úplne pozastaviť.



Obr. 5.1: Triedy `MSP430Core` a `MSP430` reprezentujúce simulačné jadro

Okrem popisovanej základnej functionality, poskytuje simulátor aj ďalšie prídavné prostriedky, ktorých účelom je dopomôcť pri ladení a trasovaní programov. Sem zaraďujeme nasledovné implementované súčasti:

- načítavanie symbolov a ladiacich informácií z binárneho súboru vo formáte ELF,
- zobrazenie obsahu registrov,
- výpis a nastavovanie obsahu pamäťových buniek,
- podpora pre záchytné body v rámci ladenia (*breakpoints*),
- podpora pre sledovanie zápisu a čítania do/z určenej adresy (resp. symbolu), podpora pre sledovanie zápisu do určeného registra (*watchpoints*),
- zobrazenie adresy vrcholu zásobníka a zobrazenie zásobníka volaní (*stack trace*),
- zobrazenie profilovacích informácií (*profiler*),
- zobrazovanie stavu po každej odsimulovanej inštrukcii, tzv. ladiaci mód (*debug mode*).

5.1.2 Ovládacie rozhranie

Rozhranie simulátoru pozostáva z grafickej a negrafickej (terminálovej) časti. *Grafickú časť* tvorí hlavné okno s programovými inštrukciami a základnými ovládacími prvkami, grafické znázornenie simulovanej platformy, monitor pre zásobník a monitor pre USART. Súčasťou je taktiež okno so zobrazením príslušného zdrojového kódu a aktuálneho riadku (riadok v rámci zdrojových kódov sa aktualizuje len v prípade krokovania, nie v prípade kontinuálnej simulácie z výkonnostných dôvodov).

Terminálová časť, tzv. CLI (command line interface), obsahuje prístup ku vlastnému príkazovému riadku simulátoru. Tu je umožnený prístup k funkciám, ktoré nemajú svoju grafickú reprezentáciu, prípadne je tu možné pristupovať k alternatívnej negrafickej reprezentácií jednotlivých funkcií simulátoru a taktiež aj k ich výstupným hodnotám.

Obe rozhrania je možné jednoduchým spôsobom rozširovať o nové zobrazenia v grafickom móde, prípadne rozširovať o nové terminálové príkazy. Rozšírenia v tomto smere sú vykonávané na základe dedičnosti existujúcich predpripravených tried a rozhraní, ktoré sú súčasťou základného systému simulátoru MSPsim.

5.2 Rozšírenie simulátoru MSPsim

Ako bolo uvedené v úvode (sekcia 5.1), pôvodný simulátor podporuje len dve platformy – ESB a Sky. Naše rozšírenie spočíva práve v pridaní podpory pre platformu FITkit. Ďalšie rozšírenia budú súvisieť s plánovaným spúšťaním operačných systémov na tejto platforme a možnosti simulácie a sledovania takejto konfigurácie.

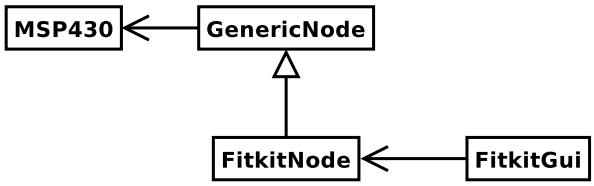
V základnej podobe podporuje simulátor len jednoduché simulácie jednoúlohových systémov. Ak ale uvažujeme operačný systém s viacerými úlohami a preemptívnym prepínaním úloh, je nutné vykonať rozšírenie, ktoré by dokázalo takúto konfiguráciu spracovávať v reálnom čase v rámci simulácie a poskytovať správne informácie. Taktiež, ak je naším cieľom časová analýza, je potrebné mať k dispozícií prostriedky pre efektívne získavanie časových značiek o výskyte nami určených udalostí. Obvykle sa zaujímame o udalosti, ako napríklad vstup do určitej časti kódu, vstup do funkcie (a parametre, s akými bola daná funkcia volaná), výstup z funkcie, zápis hodnoty na pamäťové miesta, prípadne iné udalosti týkajúce sa napríklad výskytov prerušení v systéme.

Cieľom je teda vytvorenie podpory pre viacúlohové systémy v existujúcom základe simulátoru. Tá nám umožní sledovanie behu použitého operačného systému a zároveň sledovanie zadaných udalostí. Koncept pozostáva zo všeobecného modelu behu operačného systému v rámci simulátoru, ktorý bude možné rozšíriť o špecifické prvky konkrétneho použitého systému. Takýmto spôsobom nebude model viazaný len na jeden systém, ale bude umožnené pridávať podporu aj pre iné v prípade potreby. Na základe sledovaných udalostí, ktoré sú významné z hľadiska zmien stavu systému, bude príslušne menený aj takto uvažovaný model.

V konečnej fáze bude možné stav a jednotlivé prvky, parametre a vlastnosti modelu reprezentovať graficky v podobe grafov, tabuliek s hodnotami určených premenných, prípadne iných prehľadných výpisov, ktoré nám budú poskytovať informácie o stave a behu systému v reálnom čase. Takto získané informácie bude potom možné využiť pre následnú analýzu.

5.2.1 Podpora pre platformu FITkit

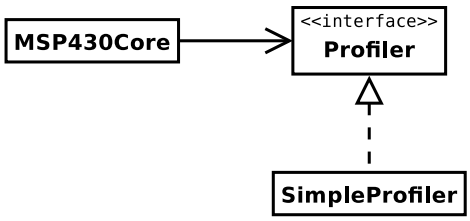
Architektúra simulátoru dovoľuje pomerne jednoduchým spôsobom pridávať podporu pre nové platformy. K tomuto účelu je vyhradená abstraktná trieda `GenericNode`. Každá platforma potom rozširuje túto triedu a pridáva svoje vlastné špecifické prvky. Takýmto spôsobom bola vytvorená aj podpora pre FITkit v podobe triedy `FitkitNode`. Tá tvorí logickú časť platformy, jej grafická časť je reprezentovaná triedou `FitkitGui` (obr. 5.2).



Obr. 5.2: Vzťah medzi triedami `GenericNode`, `FitkitNode`, `FitkitGui` a `MSP430`

5.2.2 Trasovanie a zaznamenávanie udalostí

Simulátor vo svojej pôvodnej podobe obsahuje jednoduchý profilovací podsystém, ktorého súčasťou je aj podpora pre zachytávanie udalostí týkajúcich sa volaní jednotlivých funkcií, príslušných návratov z týchto funkcií, výpis profilu a jednoúlohový výpis zásobníka volania funkcií (trieda `SimpleProfiler` implementujúca rozhranie `Profiler`, obr. 5.3). Naším cieľom je ale podpora pre viacúlohové systémy a taktiež rozšírená podpora pre zaznamenávanie udalostí spolu s časovými značkami. Pre tento účel nevyhovujú súčasne dostupné prostriedky v rámci simulátoru a je nutné realizovať vhodné rozšírenia.

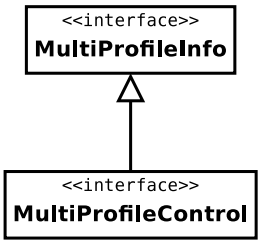


Obr. 5.3: Trieda `SimpleProfiler`

Rozhranie `MultiProfileControl` a `MultiProfileInfo`

Keďže naším cieľom je podpora pre viacúlohové systémy, je nutné zaručiť, aby nová realizácia profilovacieho podsystému dokázala získavať informácie o vzniku novej úlohy, zániku určitej úlohy, informáciu o aktuálnej úlohe a informácie o prepínaní aktivity týchto úloh. Nová trieda realizujúca dané správanie bude ale implementovať pôvodné rozhranie `Profiler` (je súčasťou pôvodných zdrojových kódov simulátoru), čím zaručíme bezproblémovú integráciu do existujúceho kódu simulačného jadra. Rozšírenie o podporu viacerých profilov je dané novým rozhraním `MultiProfileControl`. Tabuľka 5.1 popisuje základné funkcie tohto rozhrania.

Z dôvodov, ktoré budú popisované ďalej v texte, je nutné oddeliť riadiacu a informatívnu časť rozhrania. Z tohto dôvodu bolo vytvorené rozhranie `MultiProfileInfo` (tab. 5.2). Vzťah medzi týmito rozhraniami zobrazuje obr. 5.4.



Obr. 5.4: Rozhranie `MultiProfileInfo` a `MultiProfileControl`

Funkcia rozhrania	**Popis**
`createProfile(int id)`	Vytvorenie nového profilu s identifikátorom `id` (vytvorenie novej úlohy s daným identifikátorom).
`removeProfile(int id)`	Zrušenie profilu s identifikátorom `id` (zrušenie úlohy s daným identifikátorom).
`setCurrentProfile(int id)`	Nastavenie aktuálneho profilu na profil s identifikátorom `id` (nastavenie aktuálne bežiacej úlohy na úlohu s daným identifikátorom).

Tabuľka 5.1: Základ rozhrania `MultiProfileControl`

Funkcia rozhrania	**Popis**
`getCurrentProfileId()`	Získanie identifikátoru pre aktuálny profil (úlohu).
`getCurrentProfileName()`	Získanie názvu aktuálneho profilu (štandardne názov úlohy).
`getProfileName(int id)`	Získanie názvu profilu pre úlohu s konkrétnym identifikátorom.

Tabuľka 5.2: Rozhranie `MultiProfileInfo`

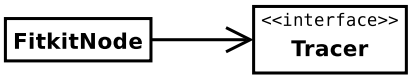
Rozhranie `Tracer`, `TracerControl` a `TracerControlFeedback`

Samotné sledovanie udalostí vstupu a výstupu z určitých funkcií a príslušné profilovacie informácie nie sú dostatočné pre časovú analýzu. Z tohto dôvodu rozšírenie obsahuje aj podporu pre generovanie, presmerovávanie a následné zachytávanie udalostí, ktoré súvisia s aktuálnou adresou programového čítača, čítaním a zápisom hodnôt v rámci pamäťových buniek, sledovaním aktivácie a deaktivácie prerušení, taktiež aj udalosti týkajúce sa počiatku a konca spracovávania prerušení. Takáto realizácia nám poskytne dostatočnú granularitu poskytovaných informácií pre vykonávanie časových analýz.

Z pohľadu funkcionality ide vlastne o realizáciu podobnú záchytným bodom (breakpoints), resp. bodom sledovania (watchpoints), taktiež zápisu do určitej adresy pamäťovej bunky (pre sledovanie zmien obsahu pamäte) alebo hodnoty registra programového čítača (pre sledovanie behu programu). Podobne je to aj v prípade sledovania prerušení a ich obslužných rutín. Tu sa ale zameriavame na sledovanie výskytu určitých konkrétnych inštrukcií (inštrukcia pre zamedzenie spracovávania prerušení `DINT`, pre ich opätovnú aktiváciu `EINT` a inštrukcia pre návrat z obsluhy prerušenia `RETI`).

Existujúce riešenie v podobe breakpoints a watchpoints nie je ale robustné pre nami uvažované ciele, hoci v základe ide o veľmi podobnú myšlienku. Je taktiež žiadúce oddeliť

pôvodnú funkcionalitu sledovania behu programu od pridávanej funkcionality rozšíreného trasovania, ktorá má úplne iný účel. Tou je práve získavanie časových značiek výskytov daných udalostí. Vytvorením oddelenej realizácie podpory trasovania týchto udalostí budeme môcť aj naďalej využívať funkcionalitu breakpoints a watchpoints, bez akejkoľvek vzájomnej interferencie. Zároveň bude možné pridávať nové trasovacie možnosti, ktoré nie sú v kompetencii breakpoints, resp. watchpoints.



Obr. 5.5: Rozhranie `Tracer`

Tabuľka 5.3 obsahuje podobu a popis jednotlivých základných funkcií navrhnutého rozhrania s názvom `Tracer`. Toto rozhranie obsahuje funkcie pre registráciu prijímateľov trasovacích informácií. Takýmto spôsobom registruje základná trieda `FitkitNode` všetky trasovacie moduly (obr. 5.5). Je nutné upozorniť, že v súčasnej realizácií tohto rozhrania je možné registrovať niekoľko štandardných prijímateľov s pasívnou spätnou väzbou, ale len jedného prijímateľa s aktívnou spätnou väzbou[5]. Spätná väzba v tomto prípade znamená, že po prijatí a interpretácií trasovacích informácií, môže takýto prijímateľ zisťovať stav (pasívny prístup) alebo naviac aj meniť stav profilovacieho podsystému (aktívny prístup). Každý prijímateľ musí realizovať presne definované rozhranie `TracerListener`. Z tohto pohľadu môžeme potom hovoriť o prijímateľoch ako o moduloch trasovacieho systému.

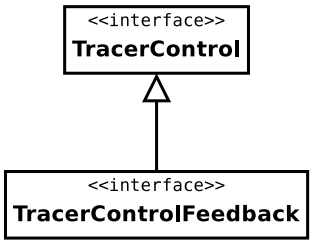
Funkcia rozhrania	**Popis**
`setCPUBind (CPUTraceControl ctc)`	Nastavenie prepojenia s CPU `ctc` (oddeľujúce rozhranie `CPUTraceCotrol`).
`addTracerListener (TracerListener listener)`	Registrácia prijímateľa `listener` preposielaných udalostí (pasívna spätná väzba).
`removeTracerListener (TracerListener listener)`	Zrušenie registrácie prijímateľa `listener` preposielaných udalostí (pasívna spätná väzba).
`addTracerListenerWithFeedback (TracerListener listener)`	Registrácia prijímateľa `listener` preposielaných udalostí (aktívna spätná väzba).
`removeTracerListenerWithFeedback (TracerListener listener)`	Zrušenie registrácie prijímateľa `listener` preposielaných udalostí (aktívna spätná väzba).

Tabuľka 5.3: Rozhranie `Tracer`

Každý modul má pre prístup v rámci spätnej väzby presne definované a centralizované rozhranie[6]. Moduly s pasívnou spätnou väzbou využívajú rozhranie `TracerControl` (tab. 5.4) a moduly s aktívnou spätnou väzbou využívajú rozhranie `TracerControlFeedback` (tab. 5.5), ktoré je rozšírením `TracerControl`.

[5]Toto obmedzenie je vykonané z dôvodu zachovania konzistencie trasovacieho a profilovacieho podsystému. V prípade, že by sme povolili viacero prijímateľov s aktívnou spätnou väzbou, je nutné realizovať aj striktnú synchronizáciu nad takto aktívnym prístupom.

[6]Tým je umožnená kontrola aj nad volaním jednotlivých funkcií trasovacieho podsystému z jednotlivých modulov – je možné realizovať identifikáciu pôvodcu danej požiadavky na trasovací podsystém. Informáciu môžeme využiť na riadenie prístupu a spôsob práce samotných funkcií, ktoré sú volané ako spätná väzba z modulov. Ďalej v texte sa oboznámime, ako je možné tento mechanizmus využiť aj pre optimalizáciu výkonu.



Obr. 5.6: Rozhranie `TracerControl` a `TracerControlFeedback`

Funkcia rozhrania	**Popis**
`setTracePoint (int adr, CPUMonitor mon)`	Nastavenie trasovania pre adresu `adr` s monitorom `mon` (platí pre všetky adresy – interne simulátor neodlišuje programovú pamäť a pamäť pre dáta).
`setInterruptTracePoint (int vector, CPUMonitor mon)`	Nastavenie trasovania pre prerušenie `vector` s monitorom `mon` (na základe tohto nastavenia sa budú generovať udalosti pre počiatok spracovávania prerušenia, ale aj pre udalosť ukončenia spracovávania prerušenia).
`removeTracePoint (int adr)`	Zrušenie trasovania pre adresu `adr`.
`removeInterruptTracePoint (int vector)`	Zrušenie trasovania pre prerušenie `vector`.
`getElf()`	Získanie informácií o binárnom súbore s programom (spolu s ladiacimi informáciami).
`getCPUReader()`	Získanie prístupu k reprezentácií stavu CPU a pamätí (pre čítanie).
`getMultiProfileInfo()`	Získanie prístupu k informáciám týkajúce sa profilovacieho podsystému.

Tabuľka 5.4: Rozhranie `TracerControl`

V konkrétnej realizácií rozhraní `Tracer`, `TracerControl` a `TracerControlFeedback` existuje jednoduchý spôsob priraďovania jednoznačných identifikátorov jednotlivým modulom. Pre každý registrovaný modul sa vytvorí inštancia realizácie `TracerControl`, resp. `TracerControlFeedback` a v nej sa potom nachádza identifikátor. Pri inicializácií modulu sa táto inštancia rozhrania predá ako parameter. Následne modul využíva túto inštanciu pre centralizovaný prístup k funkcionalite trasovacieho podsystému. To znamená, že pri volaní metód tohto rozhrania presne vieme, ktorý modul volá danú funkcionalitu.

Pre generovanie identifikátorov existuje v trasovacom podsystéme globálna premenná typu `int`. Každý bit tejto premennej potom reprezentuje jeden modul. Pri registrácii nájdeme voľný bit a jeho hodnotu tvorí potom priamo identifikátor pre modul. Po odregistrovaní je tento bit uvoľnený a môže byť použitý ako identifikátor pre iný modul. Takýmto spôsobom síce vytvoríme limit pre počet možných modulov (v prípade typu `int` je tento počet 32), ale netvorí nejaké podstatné obmedzenie. V opačnom prípade môžeme zmeniť spôsob realizácie priraďovania identifikátorov, ak to bude potrebné.

Princíp identifikácie modulov sa využíva v procese registrácie trasovacích bodov jednotlivými modulmi. V súčasnej realizácií (kvôli udržaniu výkonnosti) trasovacieho podsystému existuje len jeden monitor pre jednu konkrétnu adresu, resp. jednu konkrétnu sledovanú

Funkcia rozhrania	Popis
`getMultiProfileControl()`	Získanie prístupu k informáciám a riadeniu profilovacieho podsystému.
`getMultiStackControl()`	Získanie prístupu k riadeniu monitorovania zásobníka.

Tabuľka 5.5: Rozhranie `TracerControlFeedback`

udalosť (princíp je popisovaný ďalej v texte – monitory typu `CPUMonitor` registrované v simulačnom jadre). To znamená, že v jednoduchej realizácií by jedna adresa (udalosť) mohla byť sledovaná len jedným modulom, alebo, naopak, by museli všetky moduly získavať aj udalosti, ktoré si tento modul nezaregistroval. Takto by mohlo dochádzať k vzájomnému rušeniu modulov. Toto nie je prípustné, preto potrebujeme spôsob, akým budeme robiť selekciu nad prichádzajúcimi udalosťami a podľa toho potom presmerujeme informáciu len modulom, ktoré si tento monitor zaregistrovali.

Realizovaný trasovací podsystém riadi udržiavanie informácie o tom, ktorý modul má zaregistrovaný daný monitor v simulačnom jadre. Každý takýto monitor obsahuje premennú typu `int`, kde sa uchovávajú jednotlivé bity modulov, ktoré si zaregistrovali tento monitor. Pri výskyte udalosti sa presmerovávanie riadi na základe porovnania tejto premennej a vymaskovania bitu s bitom, ktorý reprezentuje identifikátor modulu. Takýmto spôsobom môžeme registrovať niekoľko modulov pre monitorovanie jednej udalosti a zároveň zabezpečíme, že tieto udalosti rýchlo presmerujeme. Zároveň oddelíme moduly, takže ak si sledovanie konkrétnej udalosti zaregistruje len jeden modul, bude informácie dostávať len tento jeden modul, nebude dochádzať k vzájomnému rušeniu.

Rozhranie `TracerListener`

Rozhranie `Tracer`, ktoré je centrálnou súčasťou trasovacieho podsystému, obsahuje aj funkcie pre prácu s registráciou jednotlivýh modulov (viď tab. 5.3). Všetky udalosti, ktoré sú prijímané týmto podsystémom zo simulačného jadra, sú následne preposielané ďalej registrovaným modulom.

Z tohto pohľadu môžeme hovoriť o konkrétnej triede s rozhraním `Tracer` ako o určitom smerovači jednotlivých udalostí. Tvorí totižto centrálne miesto, odkiaľ sú tieto udalosti preposielané. Takýmto spôsobom je možné kontrolovať tok preposielaných informácií a taktiež umožňuje priamo za behu systému pridávať nové moduly. Každý modul potom môže iným spôsobom spracovávať a interpretovať prijaté informácie o udalostiach zo simulačného jadra. Takéto riešenie ponúka priestor pre budúce vylepšenia v podobe výmeny existujúcich a pridávania nových modulov.

Každý modul je pred svojím použitím inicializovaný (funkcia `initialize`) a po použití finalizovaný (funkcia `shutdown`). Rozhranie obsahuje aj body pre prípad trasovania volania funkcií a príslušných návratov z týchto funkcií (`traceFunctionCall`, `traceFunctionReturn`). Parameter typu `MapEntry` je objektovou reprezentáciou daného funkčného symbolu (pochádza z pôvodného základu simulátoru, kde sa využíva aj interne). Typ `TraceFnParams` bol pridaný v rámci nášeho rozšírenia a slúži pre objektovú reprezentáciu parametrov danej trasovanej funkcie (interne obsahuje potom zoznam parametrov spolu s priradenými typmi). Modul dostáva aj informácie o vzniku, zmene a zániku úloh (kontextu, profilu). Parameter `timestamp`, ktorý sa objavuje ako parameter v jednotlivých trasovacích funkciách, obsahuje

informáciu o tom, kedy daná udalosť nastala. Tieto časové údaje môžeme potom využiť pre následné analýzy. Rozhranie je rozdelené podľa funkcionality do štyroch kategórií (tab. 5.6, 5.7, 5.9, 5.8).

Funkcia rozhrania	**Popis**
`getTracerListenerType()`	získanie typu modulu
`initialize()`	inicializácia modulu
`getStringID()`	získanie názvu modulu
`simStart()`	udalosť spustenia simulácie
`simEnd()`	udalosť ukončenia simulácie
`shutdown`	finalizácia modulu

Tabuľka 5.6: Rozhranie `TracerListener` – všeobecná funkcionalita

Funkcia rozhrania	**Popis**
`contextCreate(int context, long timestamp)`	udalosť vytvorenia nového kontextu (úlohy, profilu)
`contextCreatePost(int context)`	udalosť vytvorenia nového kontextu (volané v prípade, že modul bol registrovaný neskôr, ako vznikol kontext)
`contextChange(int context, long timestamp)`	udalosť zmeny kontextu
`contextChangePost(int context, long timestamp)`	udalosť zmeny kontextu (volané v prípade, že modul bol registrovaný neskôr, ako bol zmenený kontext)
`contextRemove(int context, long timestamp)`	udalosť odstránenia kontextu

Tabuľka 5.7: Rozhranie `TracerListener` – kontext úlohy

Funkcia rozhrania	**Popis**
`traceFunctionCall (int context, MapEntry entry, TraceFnParams params, long timestamp)`	udalosť volania funkcie, informácie o funkcii `entry`, parametre `params`
`traceFunctionCallPost (int context, MapEntry entry, TraceFnParams params, long timestamp)`	udalosť volania funkcie, informácie o funkcii `entry`, parametre `params`
`traceFunctionReturn (int context, MapEntry entry, TraceFnParams params, long timestamp)`	udalosť ukončenia funkcie, informácie o funkcii `entry`, parametre `params`

Tabuľka 5.8: Rozhranie `TracerListener` – volanie funkcií a návrat z funkcií

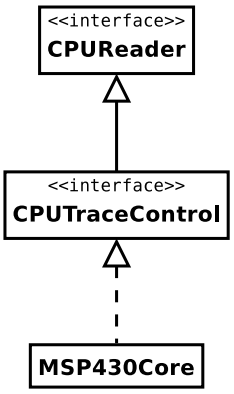
Funkcia rozhrania	Popis
`traceMemoryRead (int context, int adr, int data, long timestamp)`	udalosť čítania z pamäte, adresa `adr`, hodnota `data`
`traceMemoryWrite (int context, int adr, int data, long timestamp)`	udalosť zápisu do pamäte, adresa `adr`, hodnota `data`
`traceProgram (int context, int adr, long timestamp)`	udalosť dosiahnutia určitej hodnoty programového čítača (pred vykonaním danej inštrukcie), adresa programovej pamäte `adr`
`traceProgramPost (int context, int adr, long timestamp)`	udalosť dosiahnutia určitej hodnoty programového čítača (po vykonaní danej inštrukcie), adresa programovej pamäte `adr`
`traceInterruptsEnable (int context, long timestamp)`	udalosť povolenia prerušení
`traceInterruptsDisable (int context, long timestamp)`	udalosť zákazu prerušení
`traceInterruptBegin (int context, int adr, int vector, long timestamp)`	udalosť počiatku obsluhy prerušenia, aktuálna adresa programovej pamäte `adr`, prerušenie `vector`
`traceInterruptEnd (int context, int adr, int vector, long timestamp)`	udalosť ukončenia obsluhy prerušenia, aktuálna adresa programovej pamäte `adr`, prerušeie `vector`
`traceInterruptEndOnly (int context, long timestamp)`	udalosť volania návratu z prerušenia (bez obsluhy príslušnej prerušenia)

Tabuľka 5.9: Rozhranie `TracerListener` – pamäť, program a prerušenia

Rozhranie `CPUReader` a rozhranie `CPUTraceControl`

Simulačné jadro bolo rozšírené o možnosť registrácie nových typov monitorov určených pre trasovacie účely. Tieto sú potom umiestnené v príslušných častiach kódu, v ktorých sa vykonáva simulácia danej sledovanej akcie. Z dôvodu oddelenia funkcionality simulačného jadra ako celku a samotných funkcií týkajúcich sa nastavení trasovacích bodov (monitorov) bolo vytvorené rozhranie s názvom `CPUTraceControl` (tab. 5.11). Jadro simulátoru implementuje práve toto rozhranie (obr. 5.7).

`CPUTraceControl` je rozšírením základného rozhrania `CPUReader`, ktoré obsahuje základné operácie pre načítanie bloku pamäte (štandardného bloku pamäte aj bloku pamäte zo zásobníka) a načítanie obsahov registrov (tab. 5.10). Táto funkcionalita je dôležitá pre podporu načítania stavu z vyšších vrstiev, ktoré sú umiestnené v hierarchii nad simulačným jadrom (význam bude zrejmý pri konkrétnom využití v rámci modelov operačných systémov, ktoré budú popisované neskôr v texte).



Obr. 5.7: Rozhranie `CPUReader` a `CPUTraceControl`

Funkcia rozhrania	**Popis**
`readMemN (int adr, int[] value)`	načítanie bloku pamäte s počiatkom na adrese `adr` do poľa `value`, počet načítaných pamäťových buniek sa rovná počtu alokovaných položiek poľa
`readStackMemN (int adr, int[] value)`	načítanie bloku pamäte zo zásobníka s počiatkom na adrese `adr` do poľa `value`, počet načítaných pamäťoých buniek sa rovná počtu alokovaných položiek poľa `adr`
`readRegN (int reg, int[] value)`	načítanie obsahu registrov s počiatočným registrom `reg` do poľa `value`, počet načítaných registrov sa rovná počtu alokovaných položiek poľa `value` (poradie registrov je dané štandardným číselným označením, ktoré je možné nájsť v manuále pre mikrokontrolér MSP430)
`getInstructionOp(int adr)`	získa opcode inštrukcie z adresy `adr`
`interruptsEnabled()`	získa stav prerušení
`getCurrentInterrupt()`	získa číslo aktuálne spracovávaného prerušenia

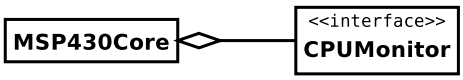
Tabuľka 5.10: Rozhranie `CPUReader`

Funkcia rozhrania	Popis
`setTracePoint (int adr, CPUMonitor mon)`	nastavenie monitoru `mon` pre prístup v rámci pamäte pre adresu `adr` (platí pre dátovú aj programovú pamäť)
`clearTracePoint (int adr)`	odstránenie monitoru pre prístup v rámci pamäte pre adresu `adr`
`hasTracePoint (int adr)`	test existencie monitoru pre pamäťovú adresu `adr`
`getTracePoint (int adr)`	získa registrovaný monitor pre adresu `adr`
`setInterruptTracePoint (int vector, CPUMonitor mon)`	nastavenie monitoru `mon` pre prerušenie `vector` (počiatok aj ukončenie obslužnej rutiny)
`clearInterruptTracePoint (int vector)`	odstránenie monitoru pre prerušenie `vector`
`hasInterruptTracePoint (int vector)`	test existencie monitoru pre prerušenie `vector`
`getInterruptTracePoint (int vector)`	získa monitor pre prerušenie `vector`

Tabuľka 5.11: Rozhranie `CPUTraceControl`

Rozhranie `CPUMonitor`

V simulačnom jadre sa jednotlivé záchytné trasovacie body realizujú využitím rozhrania `CPUMonitor` (obr. 5.8). Pôvodne bolo rozhranie vytvorené pre podporu udalostí čítania a zápisu do pamäťových buniek, čítania a zápisu do registrov a udalostí typu breakpoint. Na základe dostatočnej všeobecnosti je ale možné ho bez problémov rozšíriť pre splnenie našich účelov pridaním nových identifikátorov pre udalosti trasovania (tabuľka 5.12).



Obr. 5.8: Rozhranie `CPUMonitor`

Identifikátor typu udalosti	Popis udalosti
`MEMORY_READ`	Čítanie hodnoty z pamäťovej bunky.
`MEMORY_WRITE`	Zápis hodnoty do pamäťovej bunky.
`REGISTER_READ`	Čítane hodnoty z registra.
`REGISTER_WRITE`	Zápis hodnoty do registra.
`BREAK`	Breakpoint.
`TRACE_MEMORY_READ`	Trasovanie, čítanie hodnoty z pamäťovej bunky.
`TRACE_MEMORY_WRITE`	Trasovanie, zápis hodnoty do pamäťovej bunky.
`TRACE_PROGRAM`	Trasovanie, programový čítač dosiahol určitú hodnotu (pred vykonaním danej inštrukcie).
`TRACE_PROGRAM_POST`	Trasovanie, programový čítač dosiahol určitú hodnotu (po vykonaní danej inštrukcie).
`TRACE_CALL_POST`	Trasovanie, vykonaná inštrukcia CALL
`TRACE_INTERRUPTS_ENABLE`	Trasovanie, povolenie prerušení.
`TRACE_INTERRUPTS_DISABLE`	Trasovanie, zakázanie prerušení
`TRACE_INTERRUPT_BEGIN`	Trasovanie, vstup do obslužnej rutiny sledovaného prerušenia.
`TRACE_INTERRUPT_END`	Trasovanie, výstup z obslužnej rutiny sledovaného prerušenia.

Tabuľka 5.12: Identifikátory udalostí v rozhraní `CPUMonitor`

V realizácií v podobe triedy s daným rozhraním `CPUMonitor` sa potom nachádza obsluha určitej udalosti, ktorá je definovaná v tele funkcie `cpuAction(int type, int adr, int data, long timestamp)` (tá je súčasťou rozhrania `CPUMonitor`). Pre každú jednu udalosť, ktorú chceme zachytávať, existuje práve jeden príslušný monitor. V jadre simulátoru, kde sú jednotlivé monitory zaregistrované, sa potom vo vhodných okamihoch volá uvedená generická funkcia s vhodným nastavením parametrov podľa typu udalosti (tabuľka 5.13). Pre každý typ má posledný parameter s názvom `timestamp` rovnaký význam a tým je časová značka výskytu danej udalosti (v odkazovanej tabuľke sa tento parameter už z tohto dôvodu nenachádza).

Simulačné jadro potom obsahuje polia s inštanciami realizácií `CPUMonitor` a ak dosiahne simulácia patričný bod, ktorý monitorujeme, tak sa vyvolá zaregistrovaná funkcia `cpuAction`.

type	adr	data
MEMORY_READ	adresa dátovej pamäťovej bunky	prečítaná hodnota
MEMORY_WRITE	adresa dátovej pamäťovej bunky	zapísaná hodnota
REGISTER_READ	register	prečítaná hodnota
REGISTER_WRITE	register	zapísaná hodnota
BREAK	adresa aktuálnej programovej pamäťovej bunky	—
TRACE_MEMORY_READ	adresa dátovej pamäťovej bunky	prečítaná hodnota
TRACE_MEMORY_WRITE	adresa dátovej pamäťovej bunky	zapísaná hodnota
TRACE_PROGRAM	adresa aktuálnej programovej pamäťovej bunky	—
TRACE_INTERRUPTS_ENABLE	—	—
TRACE_INTERRUPTS_DISABLE	—	—
TRACE_INTERRUPT_BEGIN	adresa aktuálnej programovej pamäťovej bunky	vektor prerušenia
TRACE_INTERRUPT_END	—	vektor prerušenia, prípadne -1 (detekovaná samostatná inštrukcia RETI bez obsluhy prerušenia)

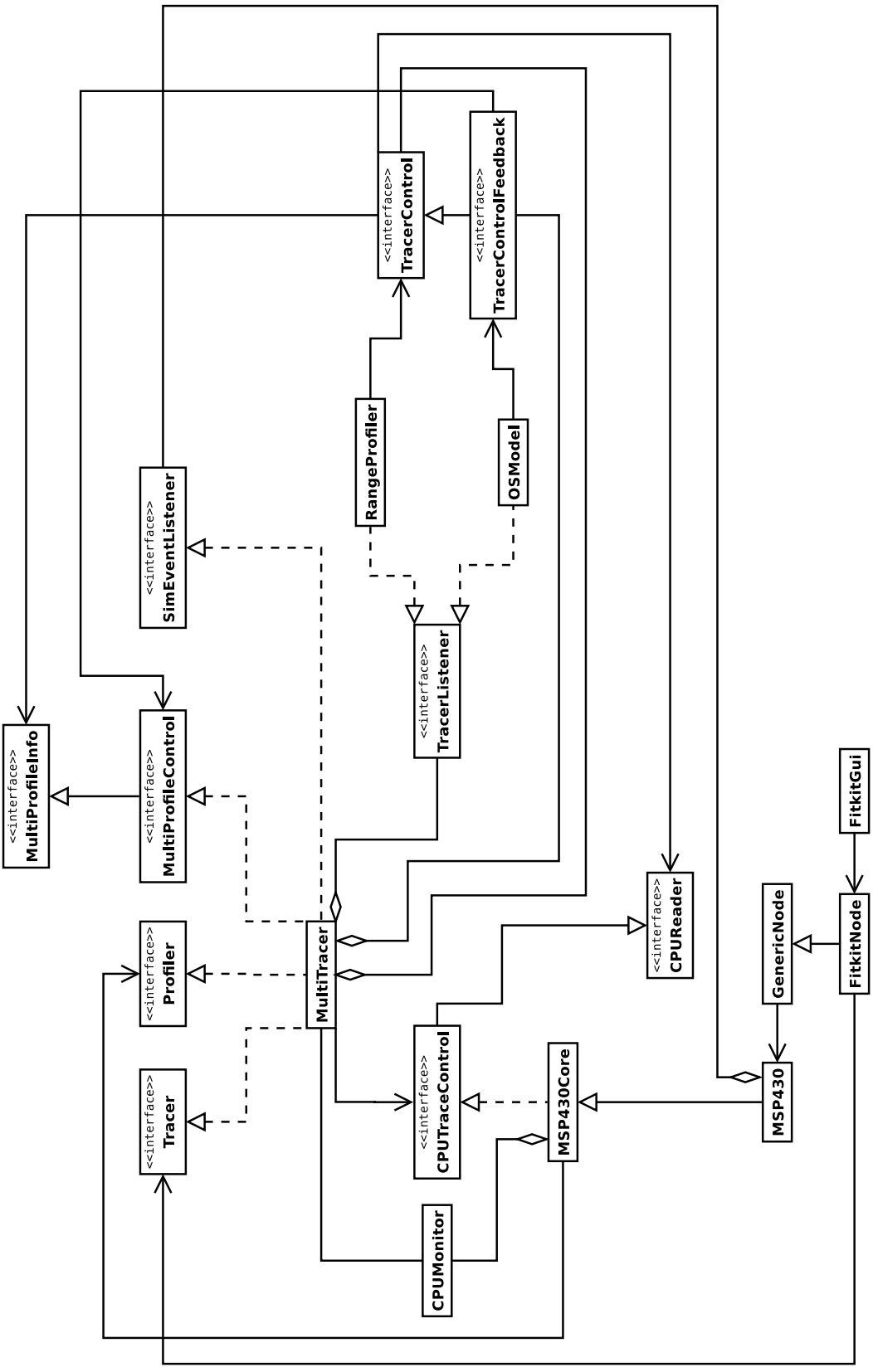
Tabuľka 5.13: Parametre cpuAction rozhrania CPUMonitor na základe typu udalosti

5.2.3 Konečná podoba realizovaného trasovacieho podsystému

Vyššie v texte boli popísané základné rozhrania, ktoré majú význam pre realizáciu trasovacieho podsystému v podobe triedy MultiTracer. Tá implementuje práve rozhrania Tracer, Profiler a MultiProfileControl, taktiež aj SimEventListener (to je súčasťou základného systému a slúži len na príjem informácií o spustení a ukončení simulácie).

Daná trieda obsahuje spätnú väzbu na simulačné jadro cez rozhranie CPUTraceControl. Tým je umožnená registrácia nových trasovacích bodov v rámci jadra MSP430Core (pripomeňme, že MSP430Core realizuje práve rozhranie CPUTraceControl). Taktiež obsahuje spätnú väzbu na simulačné jadro s využitím rozhrania CPUReader, ktoré potom sprístupňuje jednotlivým modulom čítanie stavových informácií a čítanie aktuálneho obsahu pamäte.

Multitracer registruje moduly realizujúce rozhranie TracerListener, ktorým potom presmerováva zachytené trasovacie informácie. My sme realizovali modul reprezentujúci model behu operačného systému (OSModel) a modul pre profilovanie častí kódu a získavanie štatistických informácií (RangeProfiler). Model operačného systému je modulom s aktívnou spätnou väzbou, keďže tento vytvára a mení na základe daného modelu a postupu simulácie nové úlohy, prepína medzi nimi a ruší ich. Profilovací modul je štandardný modul s pasívnou spätnou väzbou. Tieto moduly budu popisované v nasledujúcom texte. Celkovú podobu je možné vidieť na obr. 5.9.



Obr. 5.9: Rozšírenie simulátoru MSPsim – trasovací podsystém

5.2.4 Model operačného systému (modul)

Ak máme vytvorenú základnú infraštruktúru pre trasovanie kódu, môžeme pristúpiť ku konkrétnemu využitiu týchto informácií pre vybudovanie modelu operačného systému. Tento model bude upravovaný v reálnom čase tak, aby svojimi sledovanými udalosťami zodpovedal reálnemu stavu systému. Budeme sa pri tom zameriavať na sledovanie parametrov a vlastností, ktoré majú význam z pohľadu časových štatistík. Tie neskôr môžeme využiť pre vybudovanie RT podpory pre daný systém.

Abstraktná trieda `OSModel`

Na všeobecnej úrovni pozostáva model operačného systému predovšetkým z jednotlivých úloh, ktoré sú ním spravované. Každá úloha má potom priradený jednoznačný identifikátor. Taktiež, je známe, že na všeobecnej úrovni existujú tri spôsoby priraďovania výpočtových prostriedkov pre danú úlohu (plánovanie):

- kooperatívne plánovanie,
- preemptívne plánovanie,
- hybridné plánovanie (kombinácia koop. a preempt. plánovania).

Keďže z pohľadu vykonávania časových štatistík a analýz nás zaujímajú primárne úlohy a ich plánovanie v rámci real-time systémov, bude abstraktný model pozostávať práve z týchto dvoch základných prvkov – zoznamu úloh a typu určenia všeobecnej politiky ich plánovania. Ďalšie špecifické črty konkrétnych systémov už budeme modelovať odvodením od tejto všeobecnej reprezentácie operačného systému, ktorý je daný abstraktnou triedou `OSMOdel`.

Okrem iného obsahuje tento všeobecný model aj prístup k funkcionalite menovitého označenia systému (pre ľahšiu identifikáciu ako celku) a získania informácie o jeho príslušnej verzii.

V prípade, že v budúcnosti bude potrebné pridať novú funkcionalitu, ktorá by bola spoločná pre všetky realizované modely operačných systémov, bude táto abstraktná trieda práve tým pravým miestom pre vsadenie spoločnej logiky modelov. Hoci v súčasnosti nebolo potrebné realizovať takúto spoločnú logickú časť, je ale abstraktná reprezentácia vhodná pre uvažovanie budúcich rozšírení. Jej význam spočíva hlavne v jednotnej manipulácií a reprezentácií modelu na najvyššej úrovni abstrakcie.

Trieda `OSTask` a rozhranie `OSCommand`

Ako bolo popisované vyššie, model operačného systému v sebe zahŕňa zoznam úloh. Každá úloha je reprezentovaná inštanciou triedy `OSTask`. Táto trieda už ale nie je, na rozdiel od triedy `OSModel`, abstraktná, pretože obsahuje všetky primárne prvky potrebné pre jej zaradenie do konkrétneho modelu operačného systému. Na základnej úrovni je teda možné ju bez problémov priamo využiť, ak nebudeme požadovať rozšírenú funkcionalitu.

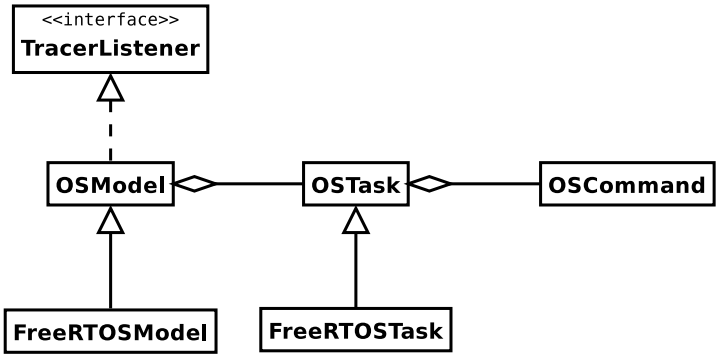
Počas simulácie je vždy aktívna práve jedna úloha (vo vytváraných modeloch sa zatiaľ neuvažujú viacprocesorové systémy). Z jadra simulátoru prichádzajú udalosti, ktoré sú potom na úrovni modelu operačného systému presmerované danej aktívnej úlohe. Tá môže potom vhodne reagovať na daný podnet a upraviť tak svoj stav, prípadne sa môže upraviť stav modelu operačného systému ako celku.

Pre definovanie reakcie na prichádzajúcu udalosť je vyhradené rozhranie `OSCommand` s jedinou funkciou `executeCommand (int event, int adr, int data, TraceFnParams params, long timestamp)`. Parametrom `event` je definovaný typ udalosti, parametre `adr` a `data` majú rovnaký význam ako v prípade funkcie `cpuAction` (viď tab. 5.13). Parameter `params` je nastavený v prípade trasovania funčných volaní – obsahuje nastavené parametre, `timestamp` obsahuje informáciu o čase výskytu danej udalosti.

Základná trieda `OSTask` poskytuje prostriedky pre identifikáciu úlohy a registráciu inštancií tried s rozhraním `OSCommand`. Každá takáto inštancia je viazaná práve na jednu konkrétnu adresu v pamäti (programovej alebo dátovej).

Tok informácií o udalostiach z jadra simulátoru k modelu úlohy

Pri práci s danou bunkou pamäte v jadre simulátoru sa vygeneruje príslušná udalosť pomocou rozhrania `CPUMonitor`. Tá je presmerovaná cez rozhranie `Tracer` k realizácií rozhrania `TracerLister`, ktorou je práve aj trieda s konkrétnym modelom operačného systému (rozširuje abstraktnú triedu `OSModel`). Odtiaľ je vybraná aktuálna úloha (trieda `OSTask`) a tej sa nakoniec presmeruje informácia o udalosti pomocou rozhrania `OSCommand` (obr. 5.10).



Obr. 5.10: Modul pre model operačného systému

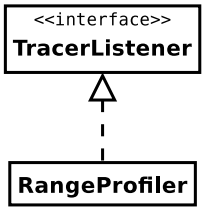
Nasleduje reakcia na udalosť súvisiaciu s danou bunkou pamäte a sú vykonané príslušné úpravy modelu OS na základe takto poskytnutých informácií. Definícia reakcie je práve v tele vyššie popisovanej funkcie `executeCommand`.

5.2.5 Profilovanie definovaných rozsahov v programe (modul)

Pre získanie informácií o časovej náročnosti vykonávania daného kódu bol vytvorený jednoduchý štatistický modul (trieda `RangeProfiler`). Tento modul načíta nastavenie trasovacích bodov zo súboru. Definícia rozsahu obsahuje počiatočnú adresu a konečnú adresu časti programu, ktorú chceme profilovať. Výhodnejšie a prehľadnejšie je ale využiť podporu pre ladiace informácie a zadávať priamo názov súboru a číslo riadku počiatočného a konečného bodu. Modul v tomto prípade automaticky preloží dané údaje na adresu v programovej pamäti. Prípadne môžeme zadať aj názov funkcie. Údaje, ktoré sa automaticky zbierajú pre dané vymedzené časti kódu, sú nasledovné:

- počet spustení danej časti kódu,
- posledná, priemerná a maximálna hodnota absolútneho času vykonávania (teda aj s prípadnými prepnutiami kontextu a behu iných úloh),

- posledná, priemerná a maximálna hodnota času vykonávania (odpočítaný čas strávený v iných úlohách, ak došlo k prepnutiu kontextu),
- posledná, priemerná a maximálna hodnota počtu prerušení v danej časti kódu,
- posledná, priemerná a maximálna hodnota času s vypnutými prerušeniami.



Obr. 5.11: Modul pre profilovanie rozsahov v programe

Profily sa môžu prelínať a taktiež je možné definovať viacero profilov s rovnakým počiatkom, resp. koncom. Každý rozsahový profil je viazaný na kontext (teda konkrétnu úlohu), ktorú určíme zadaním jej mena[7]. Definície rozsahových profilov sú umiestnené v externom súbore. Tie sa načítajú počas inicializácie modulu. Keďže v tomto čase ešte nie je spustený systém, sú tieto profily neinicializované. Tie sa inicializujú až keď sa daná úloha v systéme vytvorí (z tohto pohľadu ide o realizáciu určitej formy pozdnej väzby).

Rozsahový profil sa aktivuje v momente, keď programový čítač nadobudne hodnotu počiatočného bodu daného profilu a v tomto momente sa inicializujú aj štatistické počítadlá. Ak tok programu nadobudne koncový bod rozsahového profilu, tento sa deaktivuje, pričom sa prepočítajú štatistické údaje súvisiace s daným profilom. Je možné definovať aj počiatok a koniec v tom istom bode. V tomto prípade sa profil aktivuje a deaktivuje až pri ďalšom priechode daným bodom[8].

Za určitých podmienok môže nastať aj situácia, že sa sa dosiahne aktivačný bod profilu bez toho, aby bol predchádzajúci beh profilu ukončený. V tomto prípade sa bude predchádzajúci beh ignorovať (táto situácia môže nastať napríklad v prípade, že sa v predchádzajúcom behu preskočil bod v programe, kde sa nachádza koncový bod profilu).

5.2.6 Sledovanie stavu zásobníkov úloh vo viacúlohovom systéme

Základná verzia simulátoru obsahuje prostriedky pre sledovanie využitia zásobníka v jednoúlohovom systéme. Pre tento cieľ sa využíva rozhranie `CPUMonitor` (viď podsekcia 5.2.2), pomocou ktorého sa v jadre simulátoru zaregistruje sledovanie zmien registra vrcholu zásobníka (register `SP`).

V prostredí viacúlohového systému potrebujeme získavať naviac aj informácie o vzniku, zániku a prepínaní úloh. Rozšírenie je založené na rovnakom princípe ako rozšírenie profilovacieho podsystému pre toto viacúlohové prostredie (viď podsekcia 5.2.2, rozhranie `MultiProfileControl`). Pre tento účel bolo vytvorené rozhranie `MultiStackControl` (tab. 5.14).

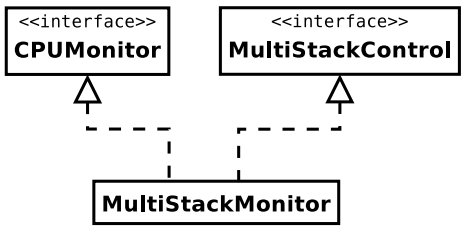
[7] Reťazec reprezentujúci meno úlohy používame preto, lebo dopredu nevieme, kde v pamäti sa bude daná úloha nachádzať, resp. nevieme, aký identifikátor systém priradí úlohe. Vo väčšine systémov sa pre úlohu definuje práve takéto meno. V opačnom prípade môžeme využiť akýkoľvek iný identifikátor, ktorý bude dopredu známy a jednoznačne rozpoznateľný počas behu systému tak, aby bolo možné daný rozsahový profil inicializovať v momente vytvorenia úlohy.

[8] Táto vlastnosť je vhodná predovšetkým pre profilovanie cyklov.

Funkcia rozhrania	Popis
`createStack (int id, int stackStartAddress, int stackDepth)`	vytvorenie modelu zásobníka pre úlohu s identifikátorom `id`, počiatočnou adresou `stackStartAddress` a veľkosťou `stackDepth`
`removeStack (int id)`	odstránenie modelu zásobníka pre úlohu s identifikátorom `id`
`setCurrentStack (int id)`	nastavenie aktuálneho zásoníka na zásobník patriaci úlohe s identifikátorom `id`

Tabuľka 5.14: Rozhranie `MultiStackControl`

Trieda `MultiStackMonitor` je potom konkrétnou realizáciou rozhraní `CPUMonitor` a `MultiStackControl` (obr. 5.12). Tým je umožnená registrácia inštancie tejto triedy v jadre systému pre sledovanie zmien registra `SP` a zároveň trieda zabezpečuje udržovanie informácií o zásobníku pre každú úlohu zvlášť. Udržujú sa informácie o počiatočnej adrese zásobníka pre danú úlohu, jeho veľkosť, aktuálna, minimálna a maximálna dosiahnutá výška zásobníka.



Obr. 5.12: Sledovanie stavu zásobníkov

Informácie o minime a maxime sú dôležité pre sledovanie podtečenia a pretečenia. Na ich základe je potom možné varovať vývojára systému, ktorý môže takto odhaliť napríklad chyby týkajúce sa nesprávne nastavenej veľkosti zásobníka, prípadne iné chyby súvisiace so zásobníkom.

Kapitola 6

Operačný systém FreeRTOS

FreeRTOS je jednoduchý a voľne dostupný operačný systém[1], napísaný prevažne v jazyku C, pričom hardvérovo závislé časti sú napísane v assembleri. Je primárne určený pre vstavané zariadenia a medzi jeho základné vlastnosti, ktoré ho predurčujú pre toto využitie, patria:

- jednoduchosť,
- minimalistické nároky na pamäťové a výpočtové prostriedky,
- prenositeľnosť kódu pre spúšťanie na rôznych architektúrach a platformách,
- dostupnosť zdrojových kódov a možnosť vlastných modifikácií (licencia GPL),
- konfigurovateľnosť na základe podmieneného kódu pre minimalizáciu využitia programovej a dátovej pamäte,
- solídny a odladený základ (aktuálna majoritná verzia s číslom 5, niekoľkoročný aktívny vývoj, využitie aj v komerčnej sfére).

FreeRTOS je voľne dostupnou verziou. Okrem nej existuje aj verzia SafeRTOS, ktorá je certifikovaná pre použitie v kritických aplikáciách. Ďalšou verziou je OpenRTOS, v rámci ktorej sa dodáva komerčná podpora, portovanie, licencovanie a iné doplnkové služby. Tieto produkty sú však už platené, preto sa budeme v ďalšom texte zaoberať len základnou a pre nás dostupnou verziou FreeRTOS.

Ak chceme získať informácie o časovej náročnosti spusteného systému spolu s úlohami, je nutné, aby sme sa zoznámili s realizáciou tohto systému. Po pochopení jeho základných štruktúr, architektúry a práce môžeme pristúpiť k výberu dôležitých a kritických súčastí, u ktorých má význam časovú analýzu vykonávať. V nasledujúcom texte sa preto zameriame na krátky popis základných štruktúr a fungovania daného systému.

6.1 Základné vlastnosti a prvky systému

FreeRTOS tvorí solídny základ pre využitie v rôznych konfiguráciách. Systém má tieto vlastnosti a prvky, ktoré sú k dispozícii pre využitie a prípadne ich následné rozšírenie [16]:

- podpora pre preemptívne, kooperatívne a hybridné plánovanie,

[1] Prezentáciu projektu, dokumentáciu a zdrojové kódy operačného systému FreeRTOS je možné nájsť na adrese http://www.freertos.org.

- podpora pre úlohy a korutiny[2],
- podpora pre vkladanie trasovacích prvkov do určitých častí kódu,
- podpora pre detekciu pretečenia zásobníka,
- počet úloh nie je softvérovo obmedzený,
- počet priorít pre úlohy nie je softvérovo obmedzený,
- podpora pre fronty, binárne semafory, počítacie semafory a rekurzívne mutexy,
- mutexy realizované s algoritmom dedenia priorít,
- na výber tri jednoduché realizácie dynamickej správy pamäte (`malloc`, `free`).

6.1.1 Súbory s implementáciou jednotlivých častí

Samotný operačný systém pozostáva len z troch (resp. štyroch) súborov umiestnených priamo v koreňovom adresári `Source`:

- `list.c` – realizácia zoznamu pre všeobecné použitie,
- `queue.c` – realizácia všeobecných front (synchronizačný a komunikačný prostriedok),
- `task.c` – realizácia úloh operačného systému a pridružených funkcií pre správu,
- `croutine.c` – realizácia spolupracujúcich úloh – korutín (voliteľné).

Implementácia častí systému v uvedených súboroch je spoločná pre všetky platformy. Okrem toho existujú časti systému, ktorých realizácia závisí od konkrétnych použitých hardvérových prostriedkov a nástrojov pre kompiláciu (vo väčšine prípadov je to hlavne časť kódu zodpovedná za prepínanie kontextu úloh). Tie sa nachádzajú v adresári `portable`. Jednou z poskytnutých realizácii je aj podpora pre kompilátor GCC a mikrokontrolér MSP430F449. Keďže rozdiely medzi mikrokontrolérmi MSP430F168 a MSP430F449 sú minimálne[3], využijeme túto pripravenú podporu pre naše ciele – použitie operačného systému v rámci platformy FITkit.

6.1.2 Globálne nastavenia

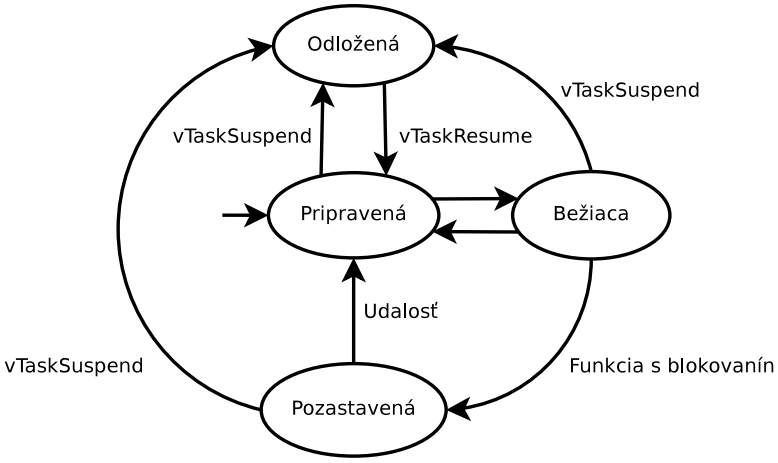
Súbor `FreeRTOSCOnfig.h`, nachádzajúci sa v koreňovom adresári so zdrojovými súbormi pre konkrétny typ platformy, obsahuje globálne nastavenia systému (viď tab. 6.1). Na základe tohto nastavenia sa bude vykonávať podmienený preklad. Tým je možné minimalizovať pamäťové nároky celého operačného systému v prípade, že niektorú funkcionalitu nepotrebujeme v rámci našej konfigurácie pre konkrétnu platformu.

6.2 Realizácia a správa plánovateľných entít

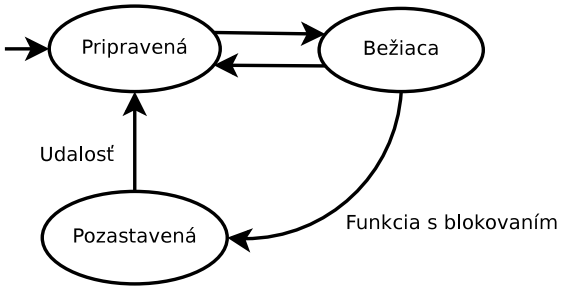
FreeRTOS má podporu pre dva rôzne typy plánovateľných entít – úlohy a korutiny. Jednotlivé stavy a možné prechody medzi nimi sú zobrazené na obr. 6.1 a 6.2 (prebraté z www stránky projektu FreeRTOS, viď [16]).

[2] V tomto prípade korutiny vo význame špeciálnych prípadov odľahčených úloh s menšou pamäťovou spotrebou, ale s určitými obmedzeniami.

[3] Z pohľadu operačného systému, akým je práve aj FreeRTOS, je v podstate rozdiel len vo veľkosti dostupnej programovej pamäte. MSP430F168 má k dispozícii 48kB, MSP430F449 má 60kB



Obr. 6.1: Stavy a prechody pre úlohy



Obr. 6.2: Stavy a prechody pre korutiny

Každá úloha má priradený tzv. kontrolný blok (TCB – task control block), kde sú uchované dôležité informácie pre spravovanie danej úlohy:

```
typedef struct tskTaskControlBlock
{
    /* ukazovateľ na vrchol zásobníka */
    volatile portSTACK_TYPE *pxTopOfStack;

    /* prvok pre vkladanie do zoznamu pripravených a blokovaných úloh */
    xListItem xGenericListItem;

    /* prvok pre vkladanie do zoznamu úloh čakajúcich na udalosť */
    xListItem xEventListItem;

    /* priorita úlohy */
    unsigned portBASE_TYPE uxPriority;

    /* ukazovateľ na počiatok zásobníka */
    portSTACK_TYPE *pxStack;

    /* meno úlohy */
    signed portCHAR pcTaskName[ configMAX_TASK_NAME_LEN ];
```

```
    /* počítadlo zanorení kritických blokov úlohy */
    #if ( portCRITICAL_NESTING_IN_TCB == 1 )
        unsigned portBASE_TYPE uxCriticalNesting;
    #endif

    /* pomocný identifikátor TCB pre zjednodušenie trasovania */
    #if ( configUSE_TRACE_FACILITY == 1 )
        unsigned portBASE_TYPE uxTCBNumber;
    #endif

    /* bázová priorita pre použitie v rámci algoritmu dedenia priorít */
    #if ( configUSE_MUTEXES == 1 )
        unsigned portBASE_TYPE uxBasePriority;
    #endif

    /* */
    #if ( configUSE_APPLICATION_TASK_TAG == 1 )
        pdTASK_HOOK_CODE pxTaskTag;
    #endif
} tskTCB;
```

Podobne, aj korutiny majú svoj kontrolný blok, ktorý je vlastne zjednodušenou podobou tohto bloku pre úlohy:

```
typedef struct corCoRoutineControlBlock
{
    /* funkcia reprezentujúca korutinu */
    crCOROUTINE_CODE pxCoRoutineFunction;

    /* prvok pre vkladanie do zoznamu pripravených a blokovaných korutín */
    xListItem xGenericListItem;

    /* prvok pre vkladanie do zoznamu korutín čakajúcich na udalosť */
    xListItem xEventListItem;

    /* priorita korutiny */
    unsigned portBASE_TYPE uxPriority;

    /* rozlišovací index pri použití rovnakej funkcie pre korutinu */
    unsigned portBASE_TYPE uxIndex;

    /* stav korutiny (interné použitie) */
    unsigned portSHORT uxState;
} corCRCB;
```

Z uvedených štruktúr pre kontrolné bloky úloh a korutín si môžeme všimnúť jeden z hlavných rozdielov. Tým je predovšetkým absencia podpory pre ukladanie informácií o

zásobníku v prípade korutín. Dôvodom je, že korutiny majú spoločný zásobník, čím dochádza k šetreniu spotreby RAM pamäte. Plánovanie medzi korutinami je kooperatívne, čím sa zjednodušuje problematika reentrantnosti daného kódu.

Nevýhodou použitia korutín je možná strata hodnôt lokálnych premenných v rámci funkcie danej korutiny v prípade, že bude blokovaná (čím vlastne iná korutina začne využívať tú istú oblasť zásobníka a s najväčšou pravdepodobnosťou prepíše existujúce hodnoty). V tomto prípade je nutné používať statické lokálne premenné, ak chceme hodnotu uchovať aj počas blokovania korutiny a následného neskoršieho návratu a používania danej premennej. Taktiež, je neprípustné blokovať úlohu v inej funkcii než je tá, ktorá je priamo korutinou (kvôli možnosti prepisu návratových hodnôt volaných funkcií z korutiny, čím by došlo k nedeterministickému skoku v rámci programu namiesto návratu do volajúcej funkcie).

Ďalšou z nevýhod korutín je prioritizácia len na úrovni ich samotných. Ak sa budú nachádzať v hybridnom prostredí spolu so štandardnými úlohami, budú úlohy mať vždy prioritu nad korutinami.

6.2.1 Zoznamy pre úlohy a korutiny

Ďalšou zo základných štruktúr v rámci systému FreeRTOS je zoznam (definovaný typ `xList`). Do týchto zoznamov sa vkladajú príslušné položky TCB úloh a korutín (položky `xGenericListItem` a `xEventListItem`, viď úseky kódov štruktúr TCB uvedené vyššie). Celkovo existuje šesť zoznamov, pričom dva z nich sú voliteľné a ich použitie závisí od nastavenia globálnych vlastností systému:

- `pxReadyTaskLists` – pole zoznamov úloh pripravených k spusteniu, pole zoradené podľa priority (pre každú prioritu existuje práve jeden zoznam, aby bolo možné definovať úlohy na rovnakej úrovni priority),
- `xDelayedTaskList1` – prvý zoznam pozastavených úloh (čakajúcich na udalosť),
- `xDelayedTaskList2` – druhý zoznam pozastavených úloh (čakajúcich na udalosť),
- `xPendingReadyList` – zoznam úloh, ktoré sa stali pripravenými na spustenie počas deaktivovaného plánovača úloh,
- `xTasksWaitingTermination` – zoznam odstránených úloh, u ktorých nebola ešte uvoľnená pamäť (voliteľné, závisí od nastavenia podpory pre odstraňovanie úloh zo systému – príslušná funkcia `vTaskDelete`),
- `xSuspendedTaskList` – zoznam odložených úloh (voliteľné, závisí od nastavenia podpory pre odkladanie úloh – príslušná funkcia `vTaskSuspend`).

Podobne je to aj v prípade korutín, ktoré majú vlastné zoznamy (s rovnakými menami ako zoznamy pre úlohy, ale namiesto `Task` je v mene zoznamu `CoRoutine`). Neobsahujú však voliteľné zoznamy odstránených a odložených úloh.

Uvedené zoznamy tvoria hlavnú štruktúru jadra systému (spolu s popisovanými štruktúrami pre TCB úloh a korutín v úvode tejto sekcie 6.2), žiadne iné základné štruktúry už operačný systém FreeRTOS neobsahuje (ak neberieme do úvahy realizáciu front a s nimi súvisice synchronizačné a komunikačné nástroje). Všetky operácie, ktoré manipulujú so stavom úloh a ich následným plánovaním, pracujú práve so zoznamami uvedenými vyššie (kód umiestnený v súbore `tasks.c` a `croutine.c` v koreňovom adresári `Source`).

6.2.2 Plánovač úloh

Plánovač úloh v systéme FreeRTOS potrebuje pre svoju správnu funkcionalitu v preemptívnom spôsobe plánovania zdroj periodických prerušení. V prípade mikrokontroléru, ktorý je súčasťou platformy FITkit, využijeme pre tento účel časovač A (`Timer_A`). Kód pre nastavenie časovača spolu s realizáciou prepnutia kontextu úloh je umiestnený v súbore `port.c` a `portmacro.h`, ktoré môžeme nájsť v adresári `portable/GCC/MSP430F168`. Tik plánovača je vo svojej základnej konfigurácií nastavený na 100 Hz.

V originálnej verzii FreeRTOS sa nachádza implementácia len jedného typu plánovacieho algortimu. Tým je prioritné plánovanie (vybraný je vždy proces s najväčšou prioritou) a v rámci jednej úrovne priority sa používa jednoduché plánovanie typu round-robin. Vyberá sa vždy úloha zo zoznamu úloh pripravených k spusteniu (`pxReadyTaskLists`).

Zdroj periodických prerušení od časovača sa používa interne aj na pozastavenie úloh na určitý čas. Pre tento účel sú vyhradené zoznamy `xDelayedTaskList1` a `xDelayedTaskList2`. Pri každom tiku časovača sa skontroluje, či v zozname nie je úloha, ktorá má byť spustená v danom čase. Ak áno, presunie sa do zoznamu úloh pripravených k spusteniu. Dva zoznamy pre pozastavené úlohy sa používajú z dôvodu vyriešenia prípadu pretečenia hodnoty určujúcej dobu, po ktorú má byť daná úloha pozastavená. Jeden zoznam obsahuje úlohy, u ktorých vypočítaný čas pre opätovnú aktiváciu nepretiekol svoju maximálnu hodnotu a druhý obsahuje úlohy, u ktorých došlo k pretečeniu. Ak prvý zoznam už bude spracovaný, význam zoznamov sa vymení (teda prvý bude plniť funkciu časových prípadov s pretečením, druhý bude aktuálne používaný).

Počet úloh a korutín je obmedzený len pamäťovými prostriedkami použitých hardvérových prostriedkov, práve kvoli využitiu dynamických štruktúr (zoznamy popisované vyššie) a možnosti vytvárania úloh za behu systému.

Nastavenie	Popis
configUSE_PREEMPTION	použitie preempcie (inak len kooperatívny OS)
configUSE_IDLE_HOOK	použitie funkcie volanej počas vykonávania úlohy IDLE
configUSE_TICK_HOOK	použitie funkcie volanej počas obsluhy prerušenia od časovača (tik plánovača)
configCPU_CLOCK_HZ	frekvencia použitého CPU
configTICK_RATE_HZ	frekvencia tiku plánovača
configMAX_PRIORITIES	maximálny počet priorít pre úlohy
configMINIMAL_STACK_SIZE	minimálna veľkosť zásobníka (pre jednu úlohu)
configTOTAL_HEAP_SIZE	celková veľkosť pamäte typu heap
configMAX_TASK_NAME_LEN	maximána dĺžka pre meno úlohy
configUSE_TRACE_FACILITY	použitie trasovania
configUSE_16_BIT_TICKS	použitie 16-bitovej reprezentácie tiku plánovača (inak 32-bitová)
configIDLE_SHOULD_YIELD	nastavenie spôsobu prideľovania CPU pre úlohy s rovnakou prioritou ako úloha IDLE (nastavenie 1 – úlohy s rovnakou prioritou ako IDLE budú uprednostnené okamžite, inak bude aplikovaný algoritmus round-robin)
configUSE_CO_ROUTINES	použitie korutín
configMAX_CO_ROUTINE_PRIORITIES	maximálny počet priorít pre korutiny
INCLUDE_vTaskPrioritySet	podpora pre dynamické nastavenie priority
INCLUDE_uxTaskPriorityGet	podpora pre získanie hodnoty priority
INCLUDE_vTaskDelete	podpora pre rušenie úloh
INCLUDE_vTaskCleanUpResources	podpora pre návrat využitých prostriedkov, ktoré boli pridelené úlohe
INCLUDE_vTaskSuspend	podpora pre odloženie úlohy
INCLUDE_vTaskDelayUntil	podpora pre pozastavenie úlohy
INCLUDE_vTaskDelay	podpora pre pozastavenie úlohy

Tabuľka 6.1: Globálne nastavenia FreeRTOS

Kapitola 7

RT plánovací mechanizmus v systéme FreeRTOS

Ako už bolo popisované v časti 6.2.2, systém FreeRTOS obsahuje vo svojej originálnej podobe len jeden plánovací mechanizmus, a to jednoduché prioritné plánovanie. Na jednej úrovni priority sú úlohy plánované cyklicky (round-robin). Ak chceme pridať podporu pre niektorý z existujúcich RT plánovacích algoritmov (viď sekcia 3.3), je výhodné vykonať najprv zmeny vo FreeRTOS tak, aby sme mohli tieto mechanizmy bez problémov do systému vsadiť.

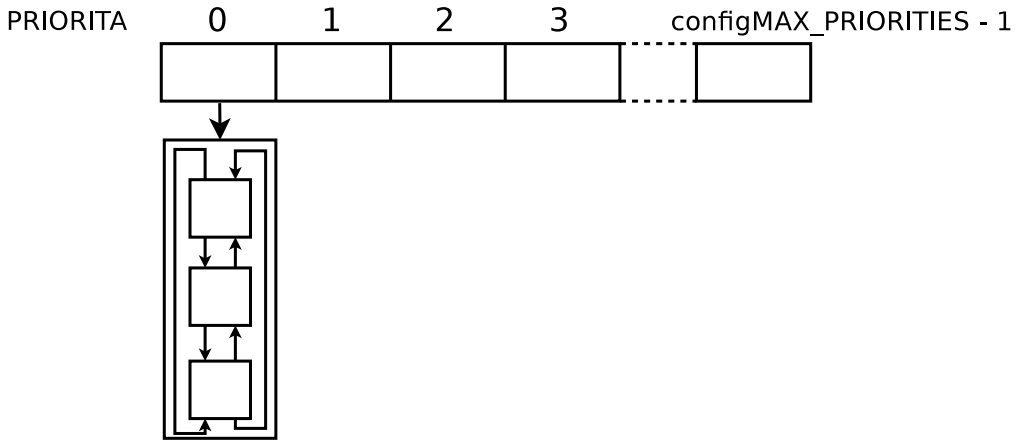
Po vykonaní potrebných zmien získame časy vykonávania jednotlivých základných operácií spolu s časovou náročnosťou konkrétnych úloh. Získané informácie potom použijeme ako vstup pre konkrétny RT plánovací mechanizmus.

7.1 Úprava štruktúry pre zoznam spustiteľných úloh

Pôvodná podoba zoznamu spustiteľných úloh v systéme FreeRTOS je jednoduchá a tvorená statickým poľom s dĺžkou rovnajúcou sa maximálnemu počtu priorít v systéme. Na každej úrovni priority je potom zoznam pre úlohy (obr. 7.1). V prípade, že priority úloh sú statické, alebo nie sú často menené, je táto štruktúra vyhovujúca. Ak ale budeme chcieť doplniť do systému plánovací algoritmus, ktorý bude často meniť tieto priority, bude pre nás výhodnejšie zmeniť štruktúru na dynamickú. Dynamická štruktúra bude výhodnejšia aj z pohľadu hodnôt použitých pre zoradenie úloh nachádzajúcich sa v tomto zozname. V prípade statickej štruktúry, teda poľa s presne ohraničenou najvyššou hodnotou a zároveň s presne definovanými hodnotami pre zoradenie, nemáme k dispozícii potrebnú voľnosť pri priraďovaní priorít.

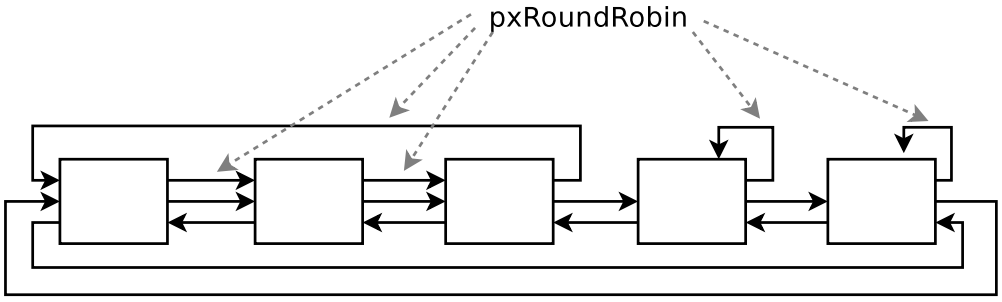
Ako príklad môžeme uviesť zoradenie úloh v zozname podľa hodnoty periódy (tak, ako je to v prípade plánovacieho mechanizmu RM). Tieto periódy môžu nadobúdať hodnoty s rozsahom, ktorý je v podstate ohraničený len použitými typmi pre ich uloženie. V prípade použitia 16-bitovej hodnoty môžeme počítať s rozsahom 65536 hodnôt. Ak by sme chceli použiť existujúcu štruktúru, buď by sme museli vytvoriť pole s danou veľkosťou, alebo by sme museli jednotlivé hodnoty rozsahovo kategorizovať (teda napríklad hodnota od 0 do 10 by patrila do kategórie 0, hodnota od 10 do 20 do kategórie 1 atď.). Prípadne by sme museli vykonať prepočet týchto hodnôt na poradie danej úlohy, čo by našu situáciu komplikovalo (museli by sme vždy brať do úvahy porovnanie s ostatnými úlohami v systéme). Na podobný problém by sme mohli naraziť ak plánujeme použiť aj iné veličiny pre zoradenie, napríklad zostávajúci čas do limitu danej úlohy, najmenšia voľnosť úlohy a podobne.

Aby sme sa vyhli príliš veľkým zmenám, môžeme pre tento účel použiť implementáciu dynamického zoznamu, ktorý je priamo súčasťou FreeRTOS. Zmeníme teda statické pole s úlohami na dynamický zoznam úloh.



Obr. 7.1: Pôvodná realizácia zoznamu spustiteľných úloh

Pre tieto účely bol upravený aj kód realizujúci zoznam – bol pridaný ukazovateľ s názvom `pxRoundRobin` (obr. 7.2). Ten umožňuje prepojiť prvky, ktoré majú rovnakú hodnotu priority. Prioritný zoznam je celkovo usporiadaný podľa hodnoty priority úlohy. Vyberá sa prvok vždy z konca zoznamu, čím sa zaručí výber úlohy s najvyššou prioritou v konštantnom čase, pretože v tomto prípade nie je už nutný priechod poľom, ako je to v prípade pôvodnej implementácie zoznamu spustiteľných úloh. Ak sa na najvyššej úrovni priority bude nachádzať prepojenie na iné úlohy s rovnakou prioritou (práve pomocou pridaného ukazovateľa `pxRoundRobin`), bude tento lokálny zoznam rotovaný. Tým sa dostane na koniec zoznamu ďalšia úloha s najvyššou prioritou a bude dochádzať k ich cyklickému plánovaniu.



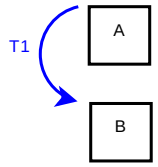
Obr. 7.2: Upravená realizácia zoznamu spustiteľných úloh

Pôvodný kód využívajúci tento zoznam bol vhodne upravený tak, aby bolo možné spustiť systém FreeRTOS s takto upravenou štuktúrou pre zoznam spustiteľných úloh s pôvodným plánovacím algoritmom. Zároveň ale úprava poskytuje väčšiu voľnosť pri realizácií iných plánovacích algoritmov. Tie môžu ale vyžadovať určenie maximálneho času pre prácu s týmito štruktúrami. V prípade dynamickej štruktúry pre zoznam spustiteľných úloh bude preto nutné vytvoriť priamo limit určením maximálneho počtu úloh v systéme. Tým bude potom možné aj časovo ohraničiť maximálne doby pri operáciách s danou štruktúrou.

7.2 Časová náročnosť základných operácií

Pre získavanie časových údajov týkajúcich sa vykonávania kódu využijeme vytvorený a popisovaný modul `RangeProfiler` (viď podsekcia 5.2.5). Pri označovaní kódu počiatočnými a koncovými značkami pre rozsahovú profiláciu budeme postupovať nasledovne (pre danú sledovanú časť kódu):

- V danom sledovanom bloku programu určíme základné bloky[1] a vytvoríme nový rozsahový profil pre tento blok. Koniec bloku bude reprezentovať riadok, resp. adresa v programovej pamäti, ktorá nasleduje hneď za sledovaným blokom. Konečný bod už nebudeme tým pádom započítavať do sledovaného času a profilácia sa ukončí pri vstupe na daný riadok kódu (resp. adresy v programovej pamäti). Tým získame dobu vykonávania daného základného bloku (obr. 7.3, blok `A` je sledovaný blok, blok `B` je blok, resp. časť programu nachádzajúca sa za sledovaným blokom).

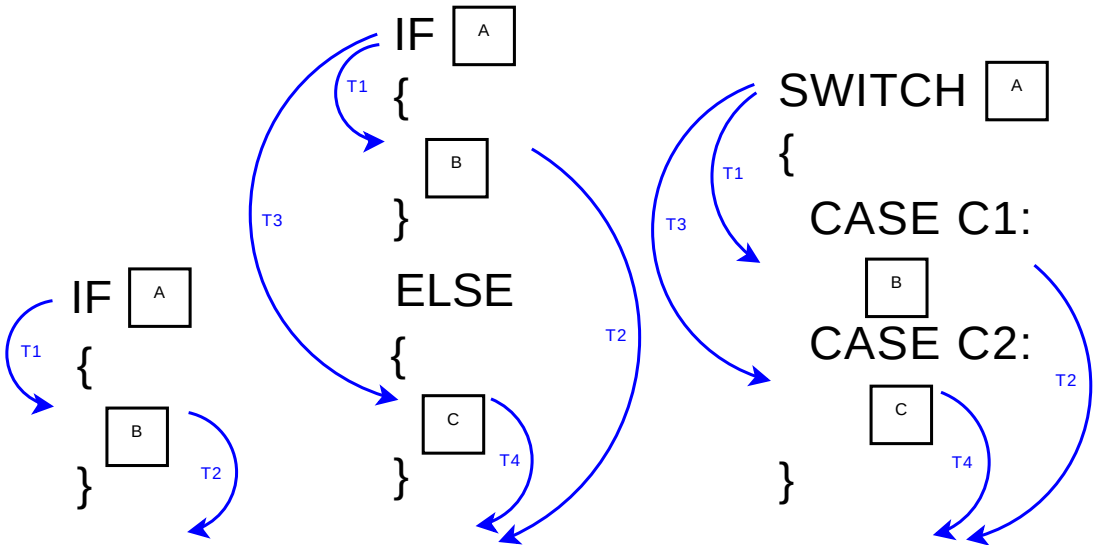


Obr. 7.3: Vytváranie rozsahových profilov pre základné bloky programu

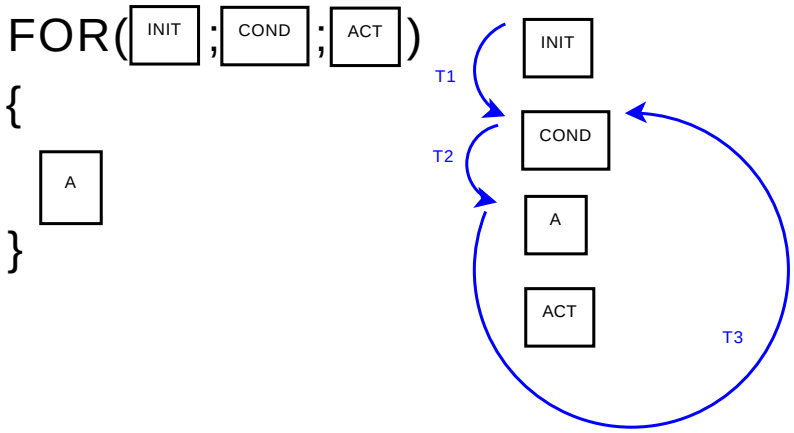
- Pre každé vetvenie typu `if` vytvoríme dva rozsahové profily tak, aby sme mohli odmerať časovú náročnosť vykonávania samotnej podmienky, ale aj tela danej vetvy. Prvý rozsahový profil bude mať svoj počiatok na riadku, kde sa nachádza samotná podmienka a koniec na prvom riadku daného tela vetvy. Druhý rozsahový profil bude mať svoj počiatok na prvom riadku tela vetvy a koniec na prvom riadku (s kódom), ktorý sa nachádza hneď za týmto telom. Pre telo vetvy postupujeme pri získavaní času vykonávania rekurzívne (teda aplikujeme presne tie isté postupy, ktoré sú tu popísané vo viacerých bodoch pre jednotlivé programové konštrukcie). Tým získame dobu vykonávania bloku danej vetvy programu spolu s príslušnou podmienkou. Podobne postupujeme u vetvenia typu `if...else`, prípadne `case` (obr. 7.4).
- Cyklus typu `for` je vzhľadom na svoju funkcionalitu zložitejší pre analýzu na základe využitia riadkových ladiacich informácií. Dôvod je ten, že cyklus pozostáva z inicializačnej časti, podmienkovej časti a časti pre nastavenie iteračnej premennej v rámci každého cyklu. Na základe ladiacich informácií môžeme získať adresu pre počiatok inicializačnej časti. Tá sa ale vykoná len na počiatku a nebude sa už ďalej vykonávať rámci každého cyklu (obr. 7.5).

  Ak chceme mať presné časové údaje, musíme v tomto prípade uvažovať reprezentáciu kódu na nižšej abstraktnej úrovni, teda assembler. Kompilátor GCC, ktorý využívame v rámci vývoja podporuje vytvorenie takejto reprezentácie do definovaného výstupného súboru. Využijeme teda túto funkcionalitu a následne budeme identifikovať jednotlivé časti v programe. Pomocou simulátoru a príkazu `lineadr` získame priamo adresu v programovej pamäti, kde sa nachádza počiatok príkazu `for`. Cyklus má na tejto úrovni tvar (príklad konkrétneho cyklu `for` vo funkcii `vListInsert`):

[1] Časti kódu bez vetvenia a cyklov, teda časti zložené z čistej sekvencie príkazov.



Obr. 7.4: Vytváranie rozsahových profilov pre `if`, `if...else` a `switch`

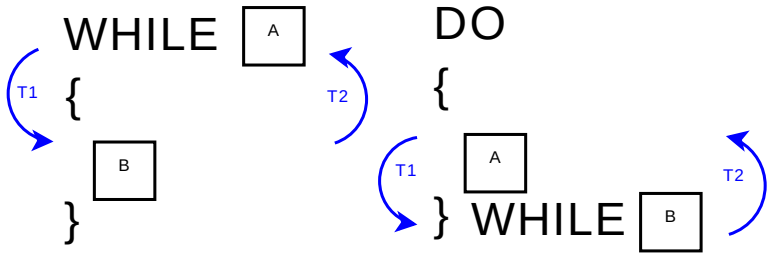


Obr. 7.5: Vytváranie rozsahových profilov pre `for`

```
;init
54be: 94 44 02 00  mov     2(r4), 4(r4)   ;0x0002(r4), 0x0004(r4)
54c2: 04 00
54c4: 1f 44 04 00  mov     4(r4), r15     ;0x0004(r4)
54c8: 1f 4f 02 00  mov     2(r15), r15    ;0x0002(r15)
;cond
54cc: a4 9f 06 00  cmp     @r15, 6(r4)    ;0x0006(r4)
54d0: 01 24        jz      $+4            ;abs 0x54d4
54d2: 06 3c        jmp     $+14           ;abs 0x54e0
;body/iter
54d4: 1f 44 04 00  mov     4(r4), r15     ;0x0004(r4)
54d8: 94 4f 02 00  mov     2(r15), 4(r4)  ;0x0002(r15), 0x0004(r4)
54dc: 04 00
54de: f2 3f        jmp     $-26           ;abs 0x54c4
```

Tento vzor sa opakuje pre všetky cykly typu `for`. Prvá časť je inicializačna (adresu počiatku `54be` sme získali pomocou príkazu simulátoru `lineadr`). Podmienku cyklu rozpoznáme podľa použitej inštrukcie pre porovnanie operandov (v tomto prípade `cmp`) a následné skoky – prvý skok do tela cyklu (v ukážke `jz $+4`) a ďalší skok pre ukončenie cyklu (v ukážke `jmp $+14`). Tretia časť je telo cyklu, ktoré bolo v tomto prípade ale prázdne a nastavenie iteračnej premennej, ktoré nasleduje. Telo cyklu spolu s nastavením iteračnej premennej je ukončené skokom spať na podmienku cyklu (v ukážke `jmp $-26`)[2].

- U cyklu typu `while` vytvoríme profil tak, že umiestnime počiatok na riadok s podmienkou a koncový bod na prvý riadok tela cyklu (získame čas vykonávania podmienky). Pre telo cyklu nastavíme počiatočný bod profilu na prvom riadku tela cyklu a koncový bod na riadok s podmienkou cyklu. Podobne budeme postupovať aj v prípade cyklu `do while` (obr. 7.6).



Obr. 7.6: Vytváranie rozsahových profilov pre `while` a `do...while`

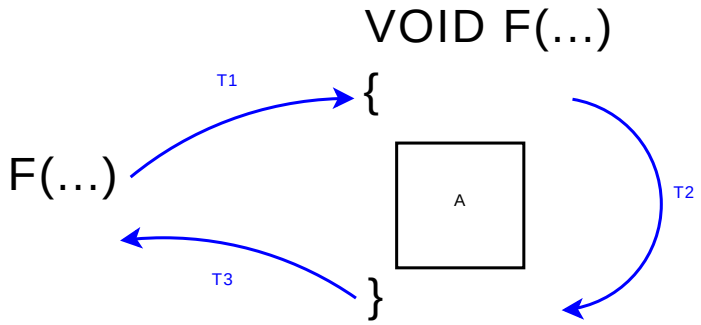
- Ďalšou dôležitou programovou konštrukciou sú funkcie. Funkcie štandardne pozostávajú z prológu, tela funkcie a epilógu. Pri volaní funkcie s parametrami je dôležité brať do úvahy aj čas použitý pre nastavenie daných parametrov pred samotným volaním funkcie. Ak využijeme ladiace informácie, tak pri označení riadku s volaním danej funkcie započítame aj prípadné nastavovanie parametrov. Pre získanie časovej náročnosti pri volaní a vykonávaní danej funkcie vytvoríme niekoľko rozsahových profilov.

  Prvý rozsahový profil bude mať svoj počiatok na riadku s volaním danej funkcie a koniec na prvom riadku tela funkcie – teda riadku, kde sa nachádza otváracia zložená zátvorka danej funkcie (tým získame čas využitý pre nastavenie parametrov a samotné volanie funkcie).

  Časovú náročnosť prológu funkcie získame použitím rozsahového profilu s počiatkom na riadku s otváracou zloženou zátvorkou danej funkcie a koncom na riadku s prvým príkazom tela danej funkcie.

  Pre samotné telo funkcie postupujeme presne podľa bodov pre vytváranie rozsahových profilov pre rôzne programové konštrukcie, ktoré sú tu popisované. Časovú náročnosť epilógu funkcie je možné odmerať vytvorením rozsahového profilu s počiatkom na zloženej zátvorke, ktorá uzatvára danú funkciu a s počiatkom na riadku s kódom nachádzajúcim sa hneď za volaním danej funkcie. V prípade, že daná funkcia má

[2]Takýto manuálny spôsob vytvárania rozsahových profilov priamo s použitím kódu v assembleri nie je príliš praktický, ale je nutný pre vykonanie presných profilov. Možným rozšírením pre modul rozsahových profilov by bolo práve automatické rozpoznávanie takýchto základných programových konštrukcií.



Obr. 7.7: Vytváranie rozsahových profilov pre funkcie

návratovú hodnotu a zároveň je táto priradená k hodnota nejakej premennej, bude započítaný aj čas tohto priradenia (viď obr. 7.7).

- Ak sa v kóde programu nachádzajú makrá, nebude možné pre ne využiť ladiace informácie súvisiace s riadkami zdrojového kódu. Pre makrá nie sú tieto informácie dostupné. V tomto prípade máme tri možnosti – uspokojíme sa s definovaním rozsahového profilu pre makro (v mieste volania) ako pre celok, alebo získame adresu v programovej pamäti, kde sa dané makro rozvinulo a vytvoríme rozsahové profily priamo pre konkrétne adresy, prípadne dočasne umiestnime telo makra na miesto jeho volania a tam definujeme rozsahové profily.

Tieto pravidlá pre vytváranie rozsahových profilov môžeme aplikovať na ľubovoľnú časť kódu. Jednotlivé časové údaje sčítame, pričom v prípade vetvenia budeme brať do úvahy najdlhší čas. Pre cykly je ale nutné určiť maximálny možný počet iterácií. Ten bude potom násobený súčtom vykonávania podmienky cyklu a samotného tela cyklu.

7.2.1 Práca so zoznamami

Ako už bolo popisované v časti 6.2.1, jedným zo základných prvkov v systéme FreeRTOS je práve zoznam. Operácie so zoznamom patria medzi najfrekventovanejšie, pretože jadro systému a celého plánovania úloh ich využíva ako základnú štruktúru pre uchovávanie a kategorizáciu entít (úloh, udalostí atď.). Pre prácu s týmito zoznamami existujú v uvažovanom systéme nasledujúce operácie:

- `vListInitialise` – inicializácia zoznamu,
- `vListInitialiseItem` – inicializácia prvku zoznamu,
- `vListInsertEnd` – vloženie prvku na koniec zoznamu,
- `vListInsert` – vloženie prvku do zoznamu (usporiadanie podľa hodnoty prvku),
- `vListRemove` – odstránenie prvy zo zoznamu.

Prvé tri operácie, `vListInitialise`, `vListInitialiseItem` a `vListInsertEnd`, sú z pohľadu analýzy časovej náročnosti jednoduché, pretože pozostávajú len z lineárneho kódu bez vetvenia a bez cyklov. Pre tieto stačí vytvoriť jednoduchý rozsahový profil, ktorého rozsah bude siahať od počiatku danej funkcie až po jej koniec. Takto sme získali nasledujúce informácie o časovej náročnosti daných operácií (tab. 7.1).

Funkcia	Maximálny čas vykonávania bez prerušenia (cykly)
`vListInitialise`	71
`vListInitialiseItem`	36
`vListInsertEnd`	103

Tabuľka 7.1: Časová náročnosť operácií pre zoznamy (časť 1)

Operácie `vListInsert` a `vListRemove` sú už zložitejšie a obsahujú ako aj vetvenia, tak aj cykly. Tieto cykly sú navyše závislé od maximálneho počtu úloh, ktoré sa v systéme môžu nachádzať. Pri získavaní časových údajov sme preto postupovali podľa pravidiel popisovaných v úvode, viď sekcia 7.2. Výsledné informácie o časovej náročnosti vykonávania týchto operácií sú uvedené v tab. 7.2.

Funkcia	Maximálny čas vykonávania bez prerušenia (cykly)
`vListInsert`	$49n + 139$
`vListRemove`	$21n + 102$

Tabuľka 7.2: Časová náročnosť operácií pre zoznamy (časť 2)

Obr. 7.8 znázorňuje postup, ktorý sme použili (pre ďalšie CFG grafy viď dodatok A). Vytvorili sme graf toku programu (CFG), v ktorom je možné vidieť body vetvenia a cyklov. V grafoch sme využili skratky pre pomenovanie jednotlivých blokov[3]:

- `START` – počiatok grafu,
- `BB` – základný blok (*basic block*),
- `COND` – jednoduché vetvenie, vyhodnotenie podmienky (*condition*),
- `FOR INIT` – inicializácia cyklu typu FOR (*initialization*),
- `FOR COND`, `WHILE COND`, `DOWHILE COND` – vyhodnotenie podmienky pre cyklus typu `FOR`, `WHILE` a `DO...WHILE`,
- `END` – koniec grafu.

Každý blok obsahuje v zátvorke údaj o jeho časovej náročnosti v cykloch (tento údaj sme získali práve použitím modulu `RangeProfiler`, podsekcia 5.2.5, zároveň postupovaním podľa krokov uvedených v sekcii 7.2). Ak tento blok obsahuje v sebe odkaz na iný graf (napríklad volanie inej funkcie), je tento fakt zaznamenaný v zátvorke pre časový údaj. Z neho potom získame maximálny čas vykonávania, ktorý použijeme ako súčasť výpočtu časovej náročnosti práve analyzovaného bloku. To znamená, že nasledujeme odkazy pre jednotlivé stromy a časy sčítavame.

Na obr. 7.8 môžeme vidieť aj ukážku príslušnosti jednotlivých blokov pre určitý kód programu a ich rozsah. Na základe znalosti CFG grafu a jednotlivých časov môžeme pristúpiť k nájdeniu najdlhšej cesty. Túto označujeme červenou farbou a plnými linkami, pričom

[3]Pre odkazovanie a rýchlejšiu orientáciu je každý blok pomenovaný – meno sa nachádza pred názvom typu bloku.

cesty, ktoré sa nezapočítavajú do doby najdlhšieho možného vykonávania sú označené šedou farbou a prerušovanými linkami. Vo všetkých prípadoch sme postupovali manuálne, pretože analyzované grafy sú jednoduché. V opačnom prípade by bolo nutné využiť externý automatizovaný nástroj pre nachádzanie najdlhšej možnej cesty v danom grafe.

7.2.2 Práca s úlohami

Pre prácu s úlohami existuje v systéme FreeRTOS viacero funkcií, z ktorých sme vybrali len tie najdôležitejšie a následne získali ich časovú náročnosť v najhoršej možnej situácií. Sú to nasledovné funkcie, prípadne makrá (príslušné CFG grafy je možné nájsť v dodatku A):

- `vTaskDelay` – pozastavenie úlohy na definovaný počet tikov časovača OS od volania tejto funkcie (relatívny čas),
- `vTaskDelayUntil` – pozastavenie úlohy do určeného času (absolútny čas),
- `prvCheckDelayedTasks` (makro) – kontrola zoznamu pozastavených úloh a aktivácia tých, ktorých doba pozastavenia skončila,
- `vTaskSuspend` – pozastavenie úlohy až do jej aktivácie,
- `vTaskResume` – aktivácia pozastavenej úlohy,
- `vTaskSuspendAll` – pozastavenie všetkých úloh v systéme,
- `xTaskResumeAll` – aktivácia všetkých úloh v systéme,
- `xTaskIsTaskSuspended` – kontrola stavu pozastavenia danej úlohy,
- `vTaskIncrementTick` – funkcia volaná v rámci tiku OS,
- `vTaskSwitchContext` – prepnutie kontextu aktuálnej úlohy na úlohu s najvyššou prioritou,
- `prvAddTaskToReadyQueue` (makro) – zaradenie úlohy do zoznamu úloh pripravených na spustenie,
- `vPortYield` – samotné prepnutie kontextu na nastavenú úlohu s najvyššou prioritou,
- `portENTER_CRITICAL` (makro) – vstup do kritickej sekcie,
- `portEXIT_CRITICAL` (makro) – výstup z kritickej sekcie.

Vo výpočtoch teoretického maximálneho času vykonávania sme počítali s dvoma premennými m a n. Premenná m reprezentuje dobu danú v počtoch tikov časovača pre plánovač, počas ktorej boli všetky úlohy pozastavené (to znamená doba medzi volaniami `vTaskSuspendAll` a `vTaskResumeAll`). Premenná n určuje maximálny počet úloh v systéme[4]. [5]

[4] Vzhľadom na fakt, že upravená verzia štruktúry zoznamu pre úlohy, ktoré sú pripravené k spusteniu, je dynamická, neexistuje priamy limit pre počet úloh v systéme. Pre účely výpočtov maximálnej doby vykonávania úloh je ale nutné takýto limit vytvoriť a potom na základe tejto informácie získať konečné hodnoty daných časov dosadením za daný parameter n.

Funkcia	Maximálny čas vykonávania bez prerušenia (cykly)
`vTaskDelay`	$91mn^2 + 91n^2 + 425mn + 99m + 486n + 921$
`vTaskDelayUntil`	$91mn^2 + 91n^2 + 425mn + 99m + 486n + 976$
`prvCheckDelayedTasks`	$91n^2 + 425n + 30$

Tabuľka 7.3: Časová náročnosť vybraných operácií pre úlohy (pozastavenie úloh na určitý čas a ich aktivácia)

Funkcia	Maximálny čas vykonávania bez prerušenia (cykly)
`vTaskSuspend`	$63n + 789$
`vTaskResume`	$70n + 662$
`vTaskSuspendAll`	30
`xTaskResumeAll`	$91mn^2 + 91n^2 + 425mn + 99m + 416n + 325$
`xTaskIsTaskSuspended`	100

Tabuľka 7.4: Časová náročnosť vybraných operácií pre úlohy (pozastavenie úloh na neurčitý čas a ich aktivácia)

7.3 Integrácia RT plánovacích mechanizmov

Jeden zo základných RT plánovacích algoritmov pre plánovanie množiny periodických úloh je práve algoritmus RM – Rate Monotonic (viď sekcia 3.3.1). Tento sa pokúsime integrovať do systému FreeRTOS ako ukážku. Iné algoritmy môžeme potom doplniť podobným spôsobom.

Pre integrovanie konkrétneho plánovacieho algoritmu do systému FreeRTOS budeme vychádzať predovšetkým z časových hodnôt vykonávania jednotlivých základných funkcií systému, ktoré boli popisované v predošlom texte. Taktiež využijeme predovšetkým informácie poskytnuté v [17]. Tento text analyzuje jednotlivé podrobnosti a praktické problémy týkajúce sa integrácie RT algoritmov do prostredia preemptívneho systému. Taktiež popisuje aj odklon od teoretického popisu týchto algoritmov a nutné rozšírenia pre ich správnu funkcionalitu v reálnom prostredí.

7.3.1 Algoritmus RM

Algoritmus RM priraďuje úlohám priority staticky a nepriamo úmerne hodnotám ich periód (alebo priamo úmerne ich kmitočtu [14]). Test plánovateľnosti pre pôvodný algoritmus RM[5], ktorého originálnu verziu môžeme nájsť v [12], má tvar (n je počet úloh, význam ďalších premenných je možné nájsť v časti 3.2):

[5] V literatúre sa väčšinou táto pôvodná analýza algoritmu RM spája s menami autorov – Liu a Layland.

Funkcia	Maximálny čas vykonávania bez prerušenia (cykly)
`vTaskIncrementTick`	$91n^2 + 425n + 82$
`vTaskSwitchContext`	131
`prvAddTaskToReadyQueue`	$49n + 151$
`vPortYield`	230

Tabuľka 7.5: Časová náročnosť vybraných operácií pre úlohy (tik plánovača a prepípanie kontextu úloh)

Funkcia	Maximálny čas vykonávania bez prerušenia (cykly)
`portENTER_CRITICAL`	5
`portEXIT_CRITICAL`	15

Tabuľka 7.6: Časová náročnosť vybraných operácií pre úlohy (kritická sekcia)

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

Uvedená podmienka je postačujúca, avšak nie je nutnou podmienkou. Hodnota na ľavej strane nerovnice je vlastne hodnota, ktorá určuje využitie procesora danou skupinou úloh. Hodnota na pravej strane určuje limit pre toto využitie tak, aby bola daná skupina úloh stále ešte plánovateľnou s využitím algoritmu RM. Pre zvyšujúce n sa blížime k limitnej hodnote $ln2 \approx 0.693$ [17].

Pre správnu funkcionalitu tohto algoritmu boli taktiež definované nasledujúce obmedzujúce požiadavky [12]:

- každá úloha, u ktorej uvažujeme hard real-time vlastnosti, je periodická a s konštantnými intervalmi medzi jednotlivými požiadavkami na spustenie,
- každá úloha musí byť ukončená do nasledujúcej požiadavky na spustenie ($T = D$),
- úlohy sú nezávislé,
- čas vykonávania úlohy je konštantný a nemení sa s časom, pričom čas vykonávania znamená vykonávanie úlohy na danom procesore bez prerušenia,
- každá neperiodická úloha v systéme je špeciálnym prípadom, nahrádza periodické úlohy a sama nemá hard real-time požiadavky.

V [17] môžeme nájsť konkrétnu praktickú ukážku, v ktorej určitá skupina úloh je plánovateľná využitím algoritmu RM, pritom ale nespĺňa podmienku plánovateľnosti určenú nerovnicou, ktorá je uvedená vyššie (to plynie aj z vlastnosti danej podmienky, ktorá je len postačujúcou). Taktiež tento text poukazuje na fakt, že daná úloha nemusí byť striktne periodická, ale podmienku pre periodicitu je možné upraviť tak, že skupina úloh bude plánovateľná aj vtedy, ak úloha prichádza najviac jeden krát v rámci každej svojej teoretickej periódy.

Model úloh bol neskôr vylepšený a uvedený v článku [8][6]. V tomto prípade sa do modelu pridalo niekoľko nových parametrov, ktoré umožnili lepšiu následnú analýzu RT systému. Hlavným parametrom, ktorý bol uvedený, je parameter určujúci najhoršiu možnú odozvu pre konkrétnu úlohu i, označovaný ako R_i. Platí [17]:

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j$$

Je to rekurentný vzťah platný iba v prípade $R_i \leq T_i$. Pre počiatok výpočtu je dôležité, aby $R_i^0 \leq R_i^n$, kde R_i^n je konečná hodnota. Výpočet ukončíme ak $R_i n + 1 = R_i^n$, alebo ak prekročíme určitú hodnotu (napríklad D_i) [17]. Parameter $hp(i)$ vo vzťahu reprezentuje množinu úloh s prioritou väčšou ako aktuálna úloha. Ak chceme určiť túto množinu, použijeme priradenie priorít na základe uvažovaného algoritmu, v našom prípade to bude algoritmus RM. Ostatné parametre boli už popisované v sekcii 3.2. Tento výpočet najhoršej odozvy vykonáme pre každú úlohu v systéme. Test plánovateľnosti potom pozostáva z jednoduchého porovnania pre všetky úlohy i:

$$R_i \leq D_i$$

Uvedená podmienka nám umožní prijať aj skupiny úloh, ktoré by neboli prijaté pôvodným testom plánovateľnosti pre algoritmus RM. Tento test je teda menej reštriktívny.

Ďalším parametrom, ktorý bol do modelu pridaný, je parameter B_i, ktorý popisuje mieru blokovania úlohy i inými úlohami. V tomto prípade ide o blokovanie úlohy s vyššou prioritou úlohami s nižšími prioritami. Dôvodom tohto blokovania je práve vzájomná synchronizácia medzi jednotlivými úlohami (napríklad použitím semaforov a podobne). Tým sa vlastne snažíme odstrániť ďalšiu z reštriktívnych podmienok pre pôvodný RM algoritmus tak, aby sa dal použiť aj v prípade reálnych podmienok, kde sa práve takéto situácie so synchronizáciou môžu vyskytnúť. Takže rovnica bude mať teraz tvar [17]:

$$R_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j$$

$$B_i = max_{\forall k \in lp(i) \forall s \in uses(k) | ceil(s) \geq pri(i)}(cs_{k,s})$$

Vo vyjadrení B_i je $lp(i)$ množina všetkých úloh s prioritou menšou ako aktuálna úloha i. Ak uvažujeme napríklad semafory ako použitý synchronizačný prostriedok, tak množina $uses(k)$ je množina semaforov, ktoré používa úloha k. Z tejto množiny potom vyberieme len tie semafory, ktorých priorita (pri uvažovaní techniky stropovania) je väčšia alebo rovná ako priorita aktuálnej úlohy. Z týchto potom vyberieme maximálny čas blokovania, teda maximálny čas trvania pre vstup úlohy k do kritickej sekcie zamknutej semaforom s, teda $max(cs_{k,s})$. A to je hľadaný parameter blokovania B_i.

Ďalšou reálnou komplikáciou, s ktorou sa môžeme stretnúť je opozdenie medzi momentom, v ktorom je úloha vyvolaná (v angl. literatúre označované ako *invocation*) a časom, kedy je systémom zaradená do zoznamu úloh pripravených na svoj beh (*release*). Táto hodnota sa môže meniť a čas medzi najskorším a najneskorším zaradením označujeme ako odchýlka zaradenia úlohy do zoznamu úloh pripravených k spusteniu (*release jitter*). Potom vzťah pre najhoršiu možnú odozvu úlohy bude mať tvar (odvodenie je možné nájsť v [17]):

[6] Táto verzia sa v literatúre spája s menami autorov Joseph a Pandya.

$$R_i = J_i + w_i$$

$$w_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j$$

Keďže sa ale člen w_i nachádza na oboch stranách rovnice, tak pre konkrétny výpočet použijeme rekurentný vzťah (podobne, ako to bolo v prípade R_i):

$$w_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n + J_j}{T_j} \right\rceil C_j$$

V prostredí preemptívneho systému je taktiež nutné uvažovať réžiu spojenú s prepínaním kontextu úloh, ktorá nie je priamo obsiahnutá v pôvodnom plánovacom algoritme RM. V tomto prípade uvažujeme možnosť prerušenia aktuálnej úlohy úlohou s vyššou prioritou. Započítavajú sa dve prepnutia kontextu – najprv je prepnutý aktuálny kontext na kontext úlohy s vyššou prioritou a po vykonaní tejto úlohy sa prepne kontext späť na aktuálny. To znamená, že pre každé C danej úlohy pripočítame dvakrát čas potrebný na prepnutie kontextu úloh, teda $2C_{sw}$, celkovo [17]:

$$w_i = C_i + 2C_{sw} + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil (C_j + 2C_{sw})$$

Preemptívne systémy sú založené na využití časovačov. Časovač generuje prerušenia v presne definovaných intervaloch. Takéto prerušenia s presne definovanou periódou využívame aj ako tik pre plánovač úloh. Na základe tiku plánovača môže byť aktivovaná úloha zo zoznamu pozastavených úloh. Čas potrebný pre vykonanie operácie presunu medzi jednotlivými zoznamami sa tiež započítava. Toto je možné vykonať tak, že pre každú úlohu v systéme definujeme virtuálnu úlohu, ktorá bude mať periódu a čas spracovávania rovnaký ako konkrétna úloha. Vypočítame počet možných presunov do zoznamu spustiteľných úloh v rámci periódy pre každú úlohu v systéme a túto hodnotu vynásobíme časom potrebným pre presun medzi zoznamami úloh. Naviac, ak máme prerušenia časovača iba v presne definovaných intervaloch, znamená to, že časové rozlíšenie je viazané práve na tieto intervaly. Ak chceme napríklad aktivovať pozastavenú úlohu v určitom čase, bude tento čas zaokrúhlený na časové rozlíšenie tiku plánovača. Preto musíme započítať aj hodnotu periódy tohto tiku do rovnice pre najhoršiu možnú odozvu úlohy (táto hodnota je vlastne pripočítaná k odchýlke pre zaradenie úlohy do zoznamu spustiteľných úloh). Taktiež je nutné započítať aj čas pre spracovanie samotnej obsluhy prerušenia, teda tiku plánovača. Celkovo [17]:

$$w_i = C_i + 2C_{sw} + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j + T_{tick}}{T_j} \right\rceil (C_j + 2C_{sw}) +$$

$$\sum_{\forall j \in alltasks} \left\lceil \frac{w_i + J_j + T_{tick}}{T_j} \right\rceil C_{queue} + \left\lceil \frac{w_i}{T_{tick}} \right\rceil C_{tick}$$

V uvedenom vzťahu je T_{tick} perióda tiku plánovača, $alltasks$ je množina všetkých úloh, C_{queue} čas potrebný pre aktivovanie pozastavenej úlohy (presun zo zoznamu pozastavených úloh do zoznamu úloh pripravených na spustenie) a C_{tick} je čas potrebný pre spracovanie obsluhy tiku plánovača (prerušenie). Jednotlivé zložky sú zobrazené na obrázku 7.9.

V preemptívnom systéme založenom na prerušeniach od časovača sa v určitých častiach kódu využíva zákaz prerušení po určitú dobu. Takýmto spôsobom sa chráni kritická časť kódu, pri ktorej vykonávaní nechceme, aby došlo k prepnutiu kontextu (hlavne z dôvodov zachovania konzistencie stavu systému). Čas, v ktorom sú prerušenia zakázané, je tiež potrebný započítať do vyššie uvedených vzťahov. Je to práve z dôvodu, že počas tejto doby zákazu prerušení nemôže byť žiadna iná úloha, hoci aj s vyššou prioritou, aktivovaná a spustená. Tento čas započítavame v rámci parametru B_i, ktorý obsahuje všetky možné blokácie úlohy i s vyššou prioritou úlohami s nižšími prioritami (v našej situácií je úloha s nižšou prioritou a zakázanými prerušeniami práve jeden z takýchto prípadov blokácie).

7.3.2 Realizácia pre systém FreeRTOS

Ak budeme uvažovať model a vzťah pre výpočet doby odozvy vyjadrený vyššie v texte, tak pre prostredie systému FreeRTOS môžeme vykonať niekoľko úprav. Tie sa budú týkať spôsobu prepínania kontextu a tiku plánovača. V systéme FreeRTOS existujú dva spôsoby, ktorým môže dôjsť k prepnutiu kontextu úloh:

- na základe tiku plánovača

```
interrupt (TIMERA0_VECTOR) prvTickISR( void )
{
  portSAVE_CONTEXT();
  vTaskIncrementTick();
  vTaskSwitchContext();
  portRESTORE_CONTEXT();
}
```

- na základe priameho volania funkcie pre prepnutie kontextu

```
void vPortYield( void )
{
  asm volatile ( "push r2" );
  _DINT();
  portSAVE_CONTEXT();
  vTaskSwitchContext();
  portRESTORE_CONTEXT();
}
```

Z uvedených príslušných častí zdrojového kódu vidíme, že tieto funkcie sú veľmi podobné. Taktiež si môžeme všimnúť, že v prípade obsluhy prerušenia od časovača dochádza k prepínaniu kontextu úloh vždy, aj v prípade, že kontext bude prepnutý na tú istú úlohu. Rozdiely sú minimálne – obsluha prerušenia obsahuje naviac zvýšenie hodnoty tiku plánovača (volanie `vTaskIncrementTick()`) a funkcia `vPortYield` obsahuje volanie pre vloženie obsahu registra `R2` (stavový register, viď 2.2.1) na vrchol zásobníka a následný zákaz prerušení (týmto spôsobom sa simuluje prerušenie, čím sa dostávame do toho istého stavu ako v prípade obsluhy reálneho prerušenia od časovača). Inak je realizácia oboch funkcií totožná.

Táto informácia je dôležitá pre posúdenie parametrov C_{sw} a C_{tick}, ktoré sa vyskytujú vo vzťahu pre výpočet maximálnej doby odozvy úlohy R_i v systéme (vzťah je odvodený a uvedený vyššie v texte). Tam sa vyskytujú členy v tvare $2C_{sw}$, čo zachytáva práve situáciu, kde je kontext prepnutý na určitú úlohu a neskôr, po ukončení danej úlohy (v rámci svojej periódy), je tento kontext prepnutý na inú úlohu. V tomto prípade sa ale nepočítalo so stavom, ktorý je napríklad v systéme FreeRTOS. Tu dochádza k prepínaniu kontextu na základe každého prerušenia. Čas potrebný pre obsluhu prerušenia od časovača je v danom vťahu reprezentovaná členom C_{tick}. V našom prípade budú teda členy C_{sw} zahrnuté práve v C_{tick}, preto ich môžeme ostrániť. Vzťahy budú mať tvar:

$$w_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j + T_{tick}}{T_j} \right\rceil C_j + \sum_{\forall j \in alltasks} \left\lceil \frac{w_i + J_j + T_{tick}}{T_j} \right\rceil C_{queue} + \left\lceil \frac{w_i}{T_{tick}} \right\rceil C_{tick}$$

V prípade, že úlohy nebudú mať žiadne odchýlky pre zaradenie do zoznamu úloh pripravených k spusteniu, môžeme člen J vynechať. Taktiež môžeme vynechať aj člen T_{tick} v nachádzajúci sa v súčte $w_i + T_{tick}$, ak budú všetky periódy násobkami T_{tick} (pretože v tom prípade budú úlohy aktivované presne na moment tiku plánovača a nemôže dôjsť k odchýlke). To môžeme vykonať tak, že paramtre periód pre dané úlohy budeme zadávať v počte tikov plánovača. Funkcie pre pozastavenie úlohy (`vTaskDelay` a `vTaskDelayUntil`), ktoré sa v systéme FreeRTOS nachádzajú, už majú svoj parameter zadávaný v tikoch. Týmto sa zjednoduší uvedený vzťah a tým aj výpočet doby odozvy. Konečný vzťah, ktorý použijeme pre výpočet bude mať preto tvar (v jeho finálnej rekurentnej podobe):

$$w_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j + \sum_{\forall j \in alltasks} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_{queue} + \left\lceil \frac{w_i^n}{T_{tick}} \right\rceil C_{tick}$$

Prvé a posledné prepnutie kontextu pre úlohu v rámci jej periódy je ale nutné pripočítať, takže vzťah pre výpočet celkovej hodnoty odozvy danej úlohy bude mať tvar:

$$R_i = w_i + C_{tick} + C_{yield}$$

C_{tick} je pre prepnutie kontextu na danú úlohu pre počiatok a C_{yield} je pre prepnutie kontextu na inú úlohu pri ukončení aktuálnej úlohy (v rámci jej periódy). Tento stav zobrazuje názorne obrázok 7.10. V tomto obrázku označenie X reprezentuje časť vzťahu (beh úloh s vyššou prioritou):

$$\sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

Označenie Y reprezentuje časť vzťahu (tik plánovača s prepínaním kontextu a s prípadným presunom úlohy zo zoznamu pozastavených úloh do zoznamu úloh pripravených na spustenie):

$$\sum_{\forall j \in alltasks} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_{queue} + \left\lceil \frac{w_i^n}{T_{tick}} \right\rceil C_{tick}$$

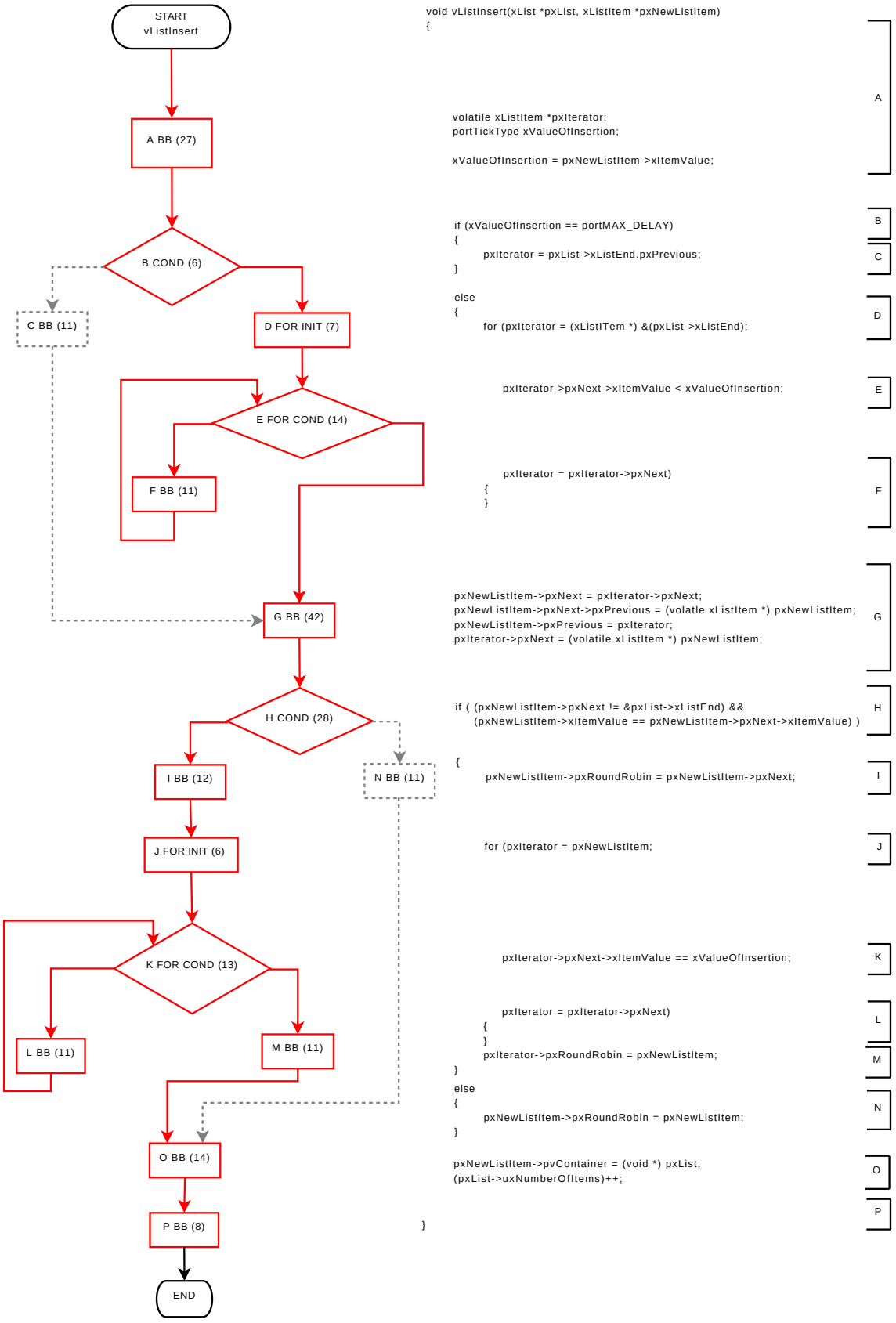
Označenie Z reprezentuje beh iných úloh, ktoré už ale nemajú priamy vplyv na sledovanú úlohu, pretože tá je už pozastavená.

Ak máme k dispozícií vzťah pre výpočet odozvy na systéme FreeRTOS, môžeme pristúpiť k určeniu jednotlivých zložiek:

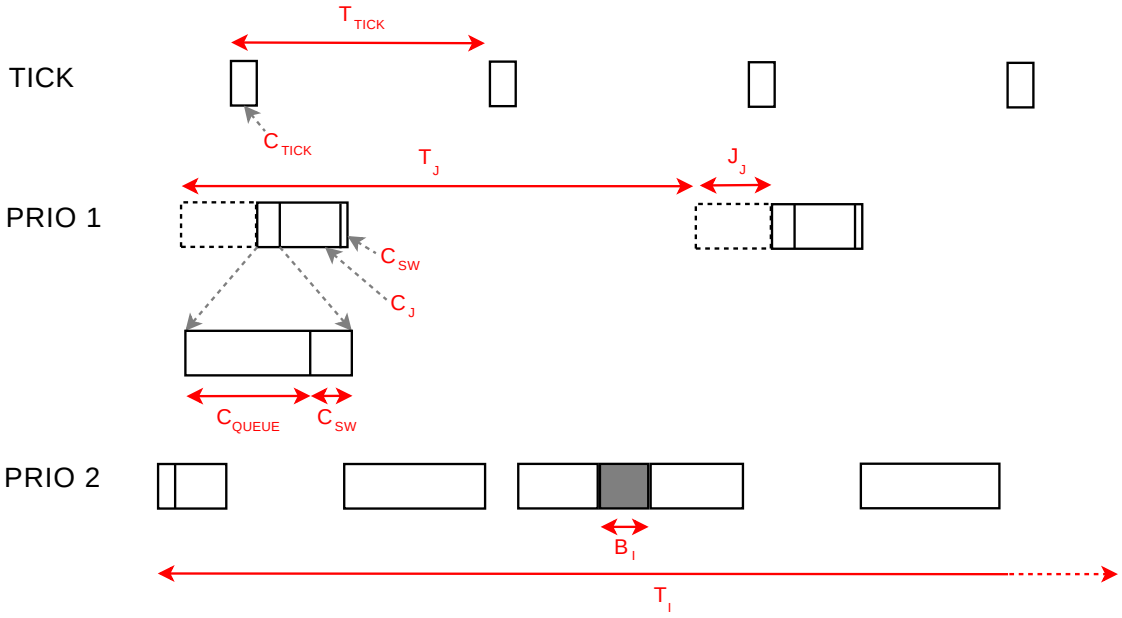
- C_i, C_j, T_j – určuje sa pre každú úlohu zvlášť, je to parameter, ktorý bude poskytnutý pri vytváraní danej úlohy,
- B_i – pozostáva z dvoch častí – časť, ktorá sa poskytne pri vytváraní úlohy (pre blokovania alebo zákaz prerušení priamo v rámci kódu úlohy samotnej) a časť pre blokovanú časť jadra bežiacej v kontexte danej úlohy,
- C_{queue}, T_{tick}, C_{tick}, C_{yield} – je špecifikované ako časová vlastnosť jadra systému.

Tým sme určili všetky potrebné časové vlastnosti pre model RT plánovacieho mechanizmu v takej podobe, v akej je popisovaný vyššie v texte. Do TCB pre úlohy v rámci FreeRTOS sme pridali štruktúru, v ktorej bude možné uchovávať jednotlivé RT parametre (štruktúra `xRTParamsType`). Taktiež pre výpočty testov plánovateľnosti je vhodné, aby sme mali k dispozícií prístup ku všetkým úlohám v systéme pre ich započítanie. Toto môže byť komplikované v prípade, že sa niektoré úlohy v momente pridávania novej úlohy nachádzajú v iných zoznamoch než je zoznam spustiteľných úloh. Z tohto dôvodu boli do TCB taktiež pridané ukazovatele, ktoré jednotlivé úlohy prepájajú bez ohľadu na ich príslušnosť do určitého zoznamu v rámci systému (ukazovatele `pxNext` a `pxPrevious`).

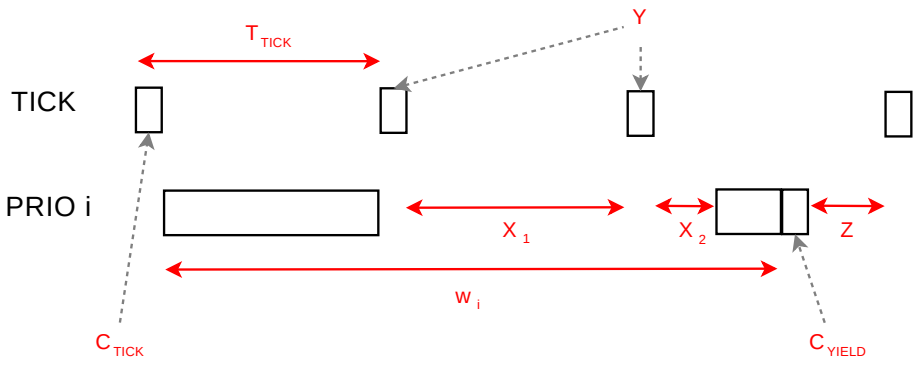
Konkrétnu realizáciu so všetkými potrebnými úpravami a s ukážkami je možné nájsť na priloženom CD, ktoré je súčasťou prílohy. Taktiež je tu možné nájsť aj rozšírenú verziu simulátora MSPsim, v ktorom sme vykonávali simulácie. Využili sme pri tom realizovaný trasovací podsystém a moduly pre model operačného systému (v tomto prípade FreeRTOS) a jednotlivé trasovacie body pre rozsahovú profiláciu v rámci modulu `RangeProfiler`. CD taktiež obsahuje aj krátku nápovedu pre prácu s týmto rozšírením.



Obr. 7.8: CFG s uvedenou dobou vykonávania pre bloky programu vo `vListInsert`



Obr. 7.9: Výpočet odozvy úlohy



Obr. 7.10: Výpočet odozvy úlohy v prostredí FreeRTOS

Kapitola 8

Záver

Práca sa vo svojej úvodnej časti zameriavala na uvedenie tematiky RT systémov. Cieľom bolo vytvorenie RT prostredia s podporou pre spúšťanie viacerých úloh na platforme FITkit. Pre vytváranie, prípadne úpravu existujúcich systémov, je nutné sa podrobne zoznámiť s cieľovou architektúrou a jej funkcionalitou. Úvodná časť preto obsahovala aj popis platformy FITkit, konkrétne popis použitého mikrokontroléru vzhľadom na požiadavky pre nasadenie preemptívneho operačného systému.

Ak chceme zabezpečiť RT vlastnosti systému, je potrebné získať detailné časové údaje pre jednotlivé úlohy, ktoré v systéme existujú. Sú dva základné prístupy k získaniu týchto informácií, a to buď využitím statickej analýzy alebo dynamickej analýzy. Práca preto obsahuje aj krátky popis jednotlivých metód, porovnáva ich vzhľadom na ich výhody, nevýhody a vhodnosť použitia pre konkrétne situácie. Spôsob, ktorý bol vybraný, je dynamická analýza s využitím simulátora a trasovaním behu programu.

V súčasnej dobe je pomerne ťažké nájsť voľne dostupný simulátor mikrokontroléru, ktorý je aj súčasťou platformy FITkit. Keďže sa FITkit používa predovšetkým pre študijné účely, je práve takýto nástroj vhodný a žiadaný. Projekt MSPsim, ktorý je ešte stále vo vývoji, predstavuje existujúci a voľne dostupný simulátor. Jeho nevýhodou však je, že podporuje len sledovanie behu jednej úlohy. Práca sa preto zameriavala na jeho rozšírenie tak, aby bolo možné sledovať beh operačných systémov s viacerými úlohami, pričom by bolo možné sledovať aj prepínanie kontextu úloh. Na to, aby sme mohli sledovať aj časové závislosti v systéme pri jeho behu, sme zovšeobecnili tento cieľ na vytvorenie trasovacieho podsystému, ktorý môže byť využitý aj na iné účely než len na sledovanie behu operačného systému.

Pri návrhu sa kládol dôraz na modulárnosť riešenia práve kvôli rôznorodosti možného použitia funkcionality trasovania. Vytvorili sme dva ukážkové moduly – prvý reprezentujúci model operačného systému a druhý, ktorého úlohou je zbieranie štatistických údajov o behu ľubovoľného rozsahu v programe a jeho spotrebe času CPU pre vykonanie. Na základe využitia týchto modulov a pri dodržiavaní určitého postupu sme získali detailné informácie o časovej náročnosti jednotlivých častí programu. Tieto údaje slúžili ako vstup pre integrovanie RT mechanizmu do systému FreeRTOS.

Záverečná časť práce sa zameriava na uvedenie ukážky integrácie RT plánovacieho mechanizmu RM do systému FreeRTOS. Boli uvedené vzťahy, ktoré sú platné pre tento plánovací mechanizmus, pričom sme sa zameriavali na uvažovanie reálnych podmienok. Medzi takéto patrí aj fakt, že réžia spojená so samotným plánovaním, a teda behom operačného systému, má nezanedbateľný vplyv na výsledné časové závislosti. Existujúce vzťahy sme mierne upravili, aby boli aplikovateľné na špecifické prostredie konkrétneho operačného systému, v našom prípade práve FreeRTOS. Tieto poznatky, spolu so získanými časovými

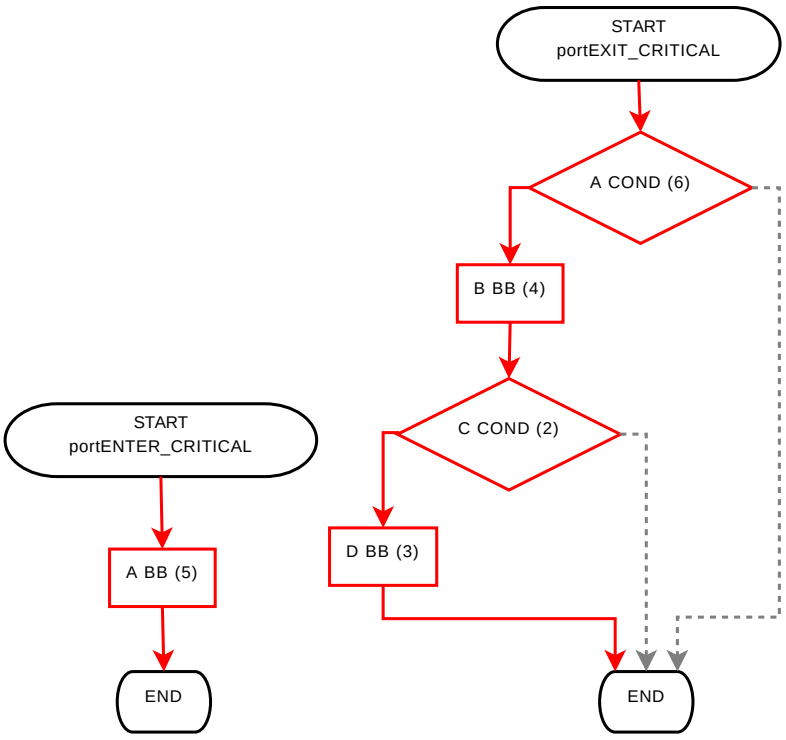
vlastnosťami systému, sme nakoniec využili pre úpravu systému tak, aby bolo možné zabezpečiť RT vlastnosti na základe použitia konkrétneho RT algoritmu.

Trasovací podsystém bol vytváraný tak, aby poskytoval čo najväčšiu možnosť pri jeho dopĺňaní o novú funkcionalitu, prípadne modifikáciu. Dôraz sa taktiež kládol na jednoduchosť pri vytváraní nových modulov a taktiež aj aplikovateľnosť pre rôzne operačné systémy, ktoré sa môžu týmto spôsobom analyzovať. Toto poskytuje možnosti pre budúce rozšírenia. Okrem rozšírení týkajúcich sa nových modulov pre trasovanie je možné prácu rozšíriť aj o vytvorenie prepracovaného grafického užívateľského rozhrania, ktoré by uľahčilo prácu používateľom.

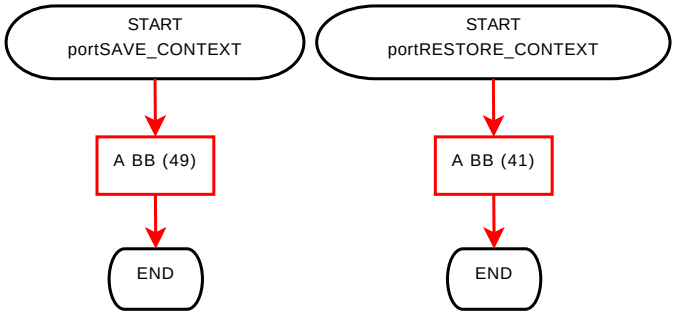
Zoznam príloh

Dodatok A

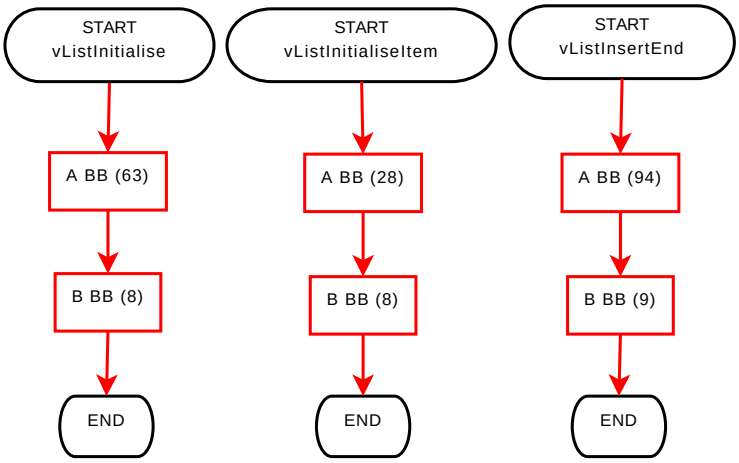
Základné funkcie v systéme FreeRTOS – CFG grafy a doba vykonávania základných blokov



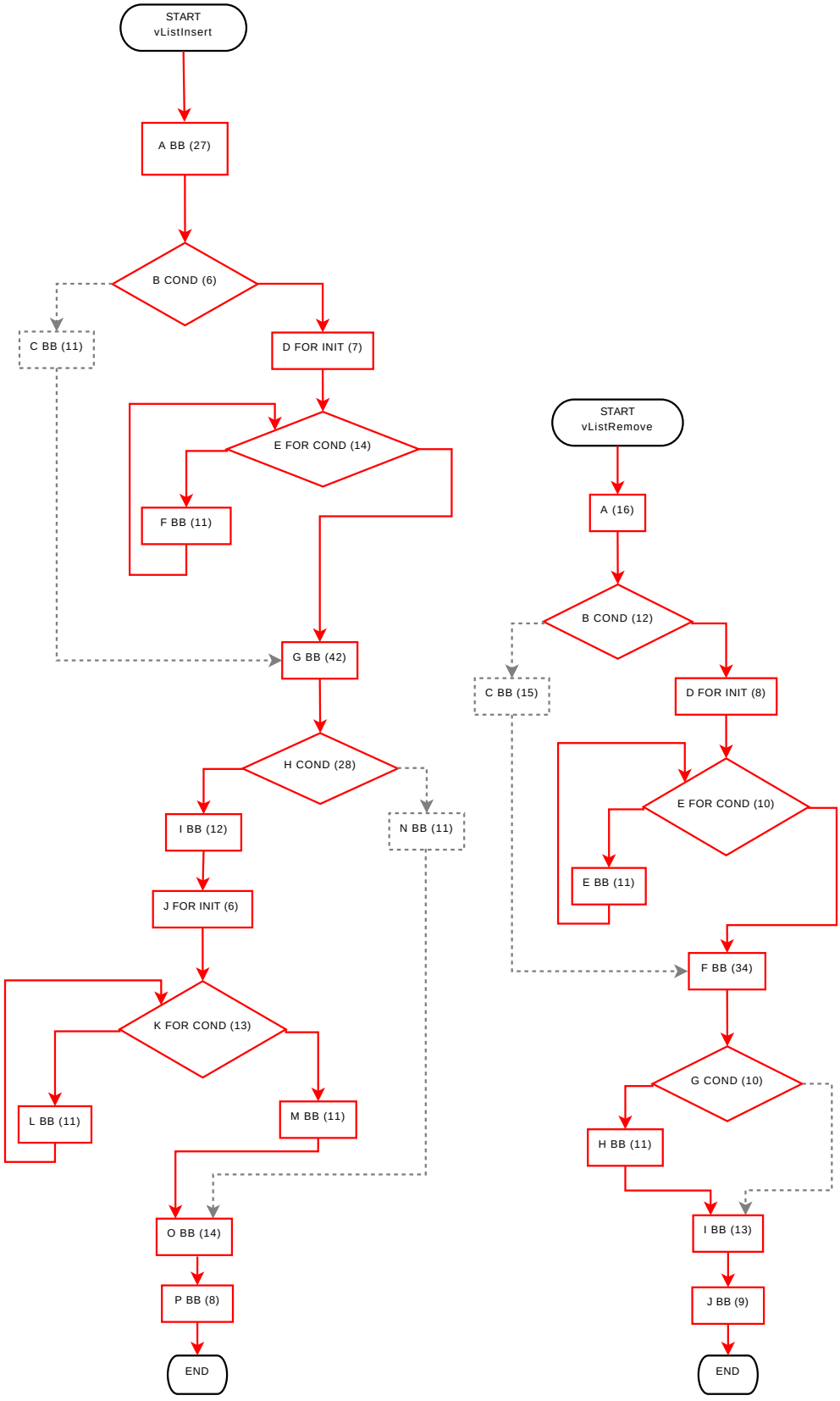
Obr. A.1: CFG pre `portENTER_CRITICAL` a `portEXIT_CRITICAL`



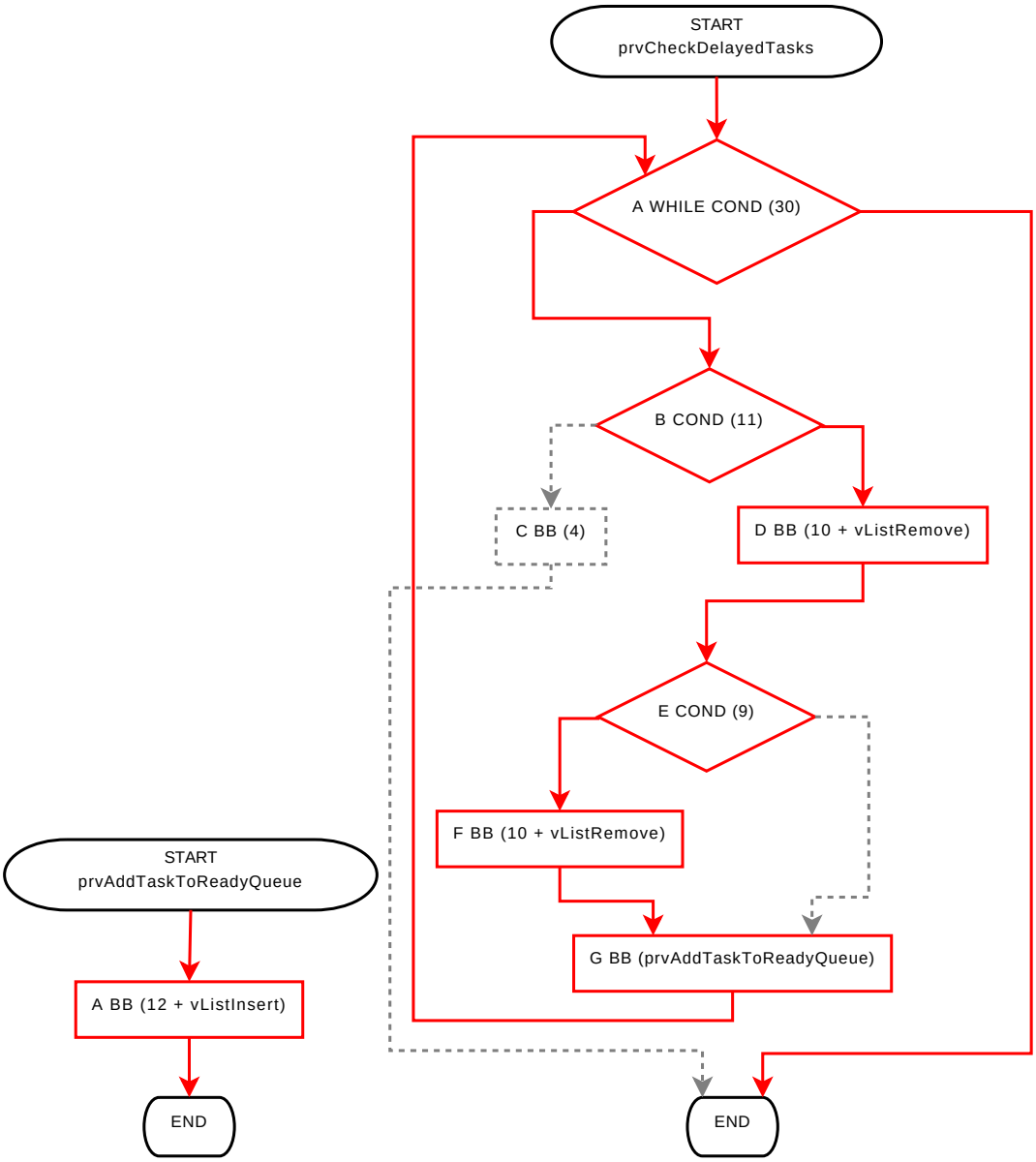
Obr. A.2: CFG pre `portSAVE_CONTEXT` a `portRESTORE_CONTEXT`



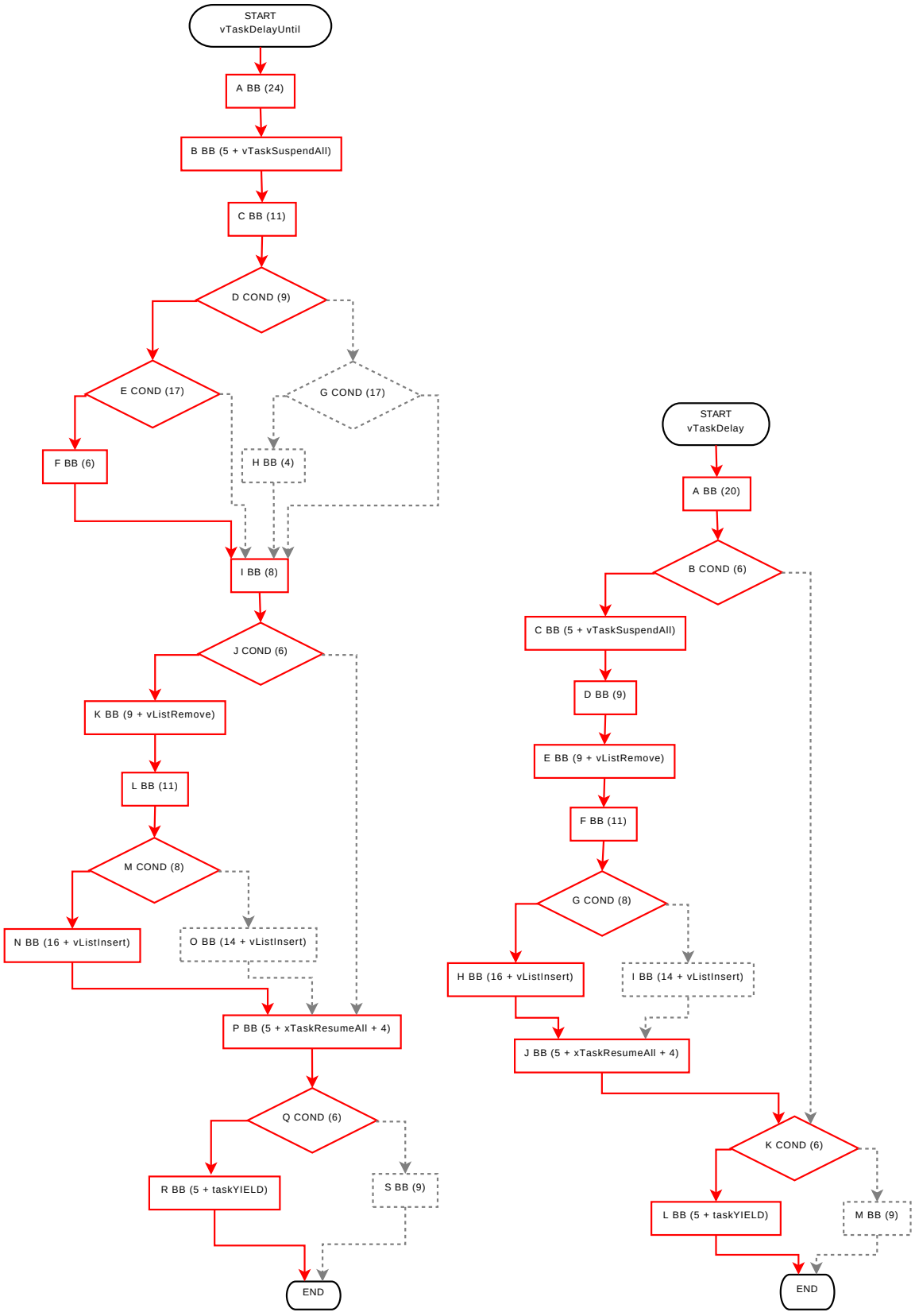
Obr. A.3: CFG pre `vListInitialise`, `vListInitialiseItem` a `vListInsertEnd`



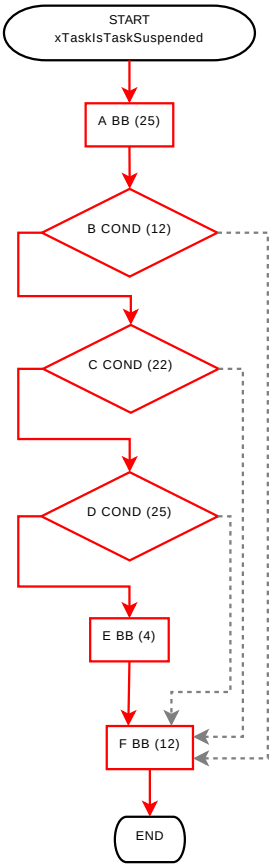
Obr. A.4: CFG pre `vListInsert` a `vListRemove`



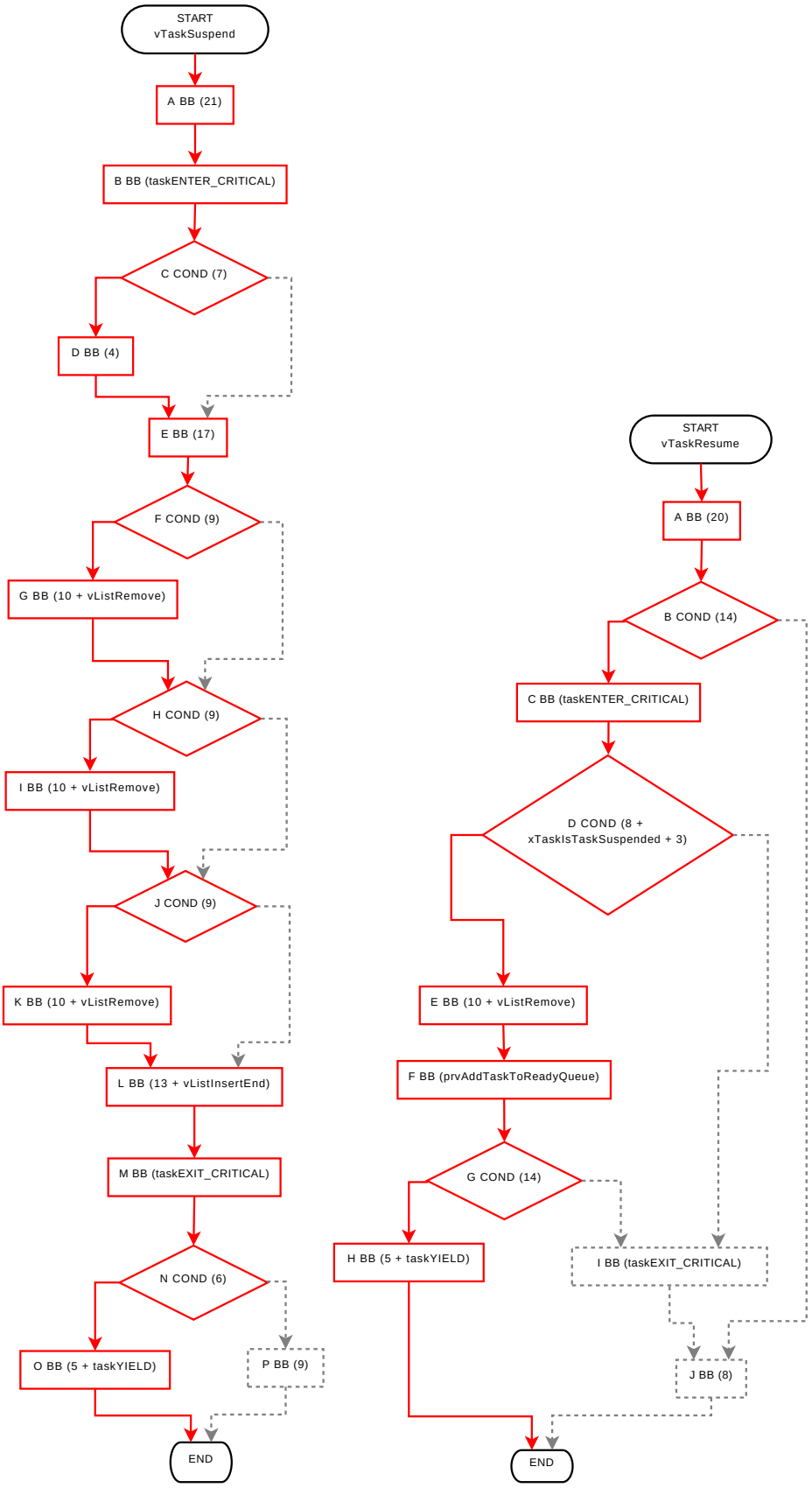
Obr. A.5: CFG pre `prvAddTaskToReadyQueue` a `prvCheckDelayedTasks`



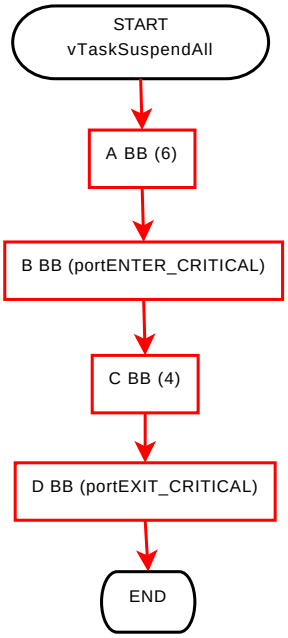
Obr. A.6: CFG pre `vTaskDelayUntil` a `vTaskDelay`



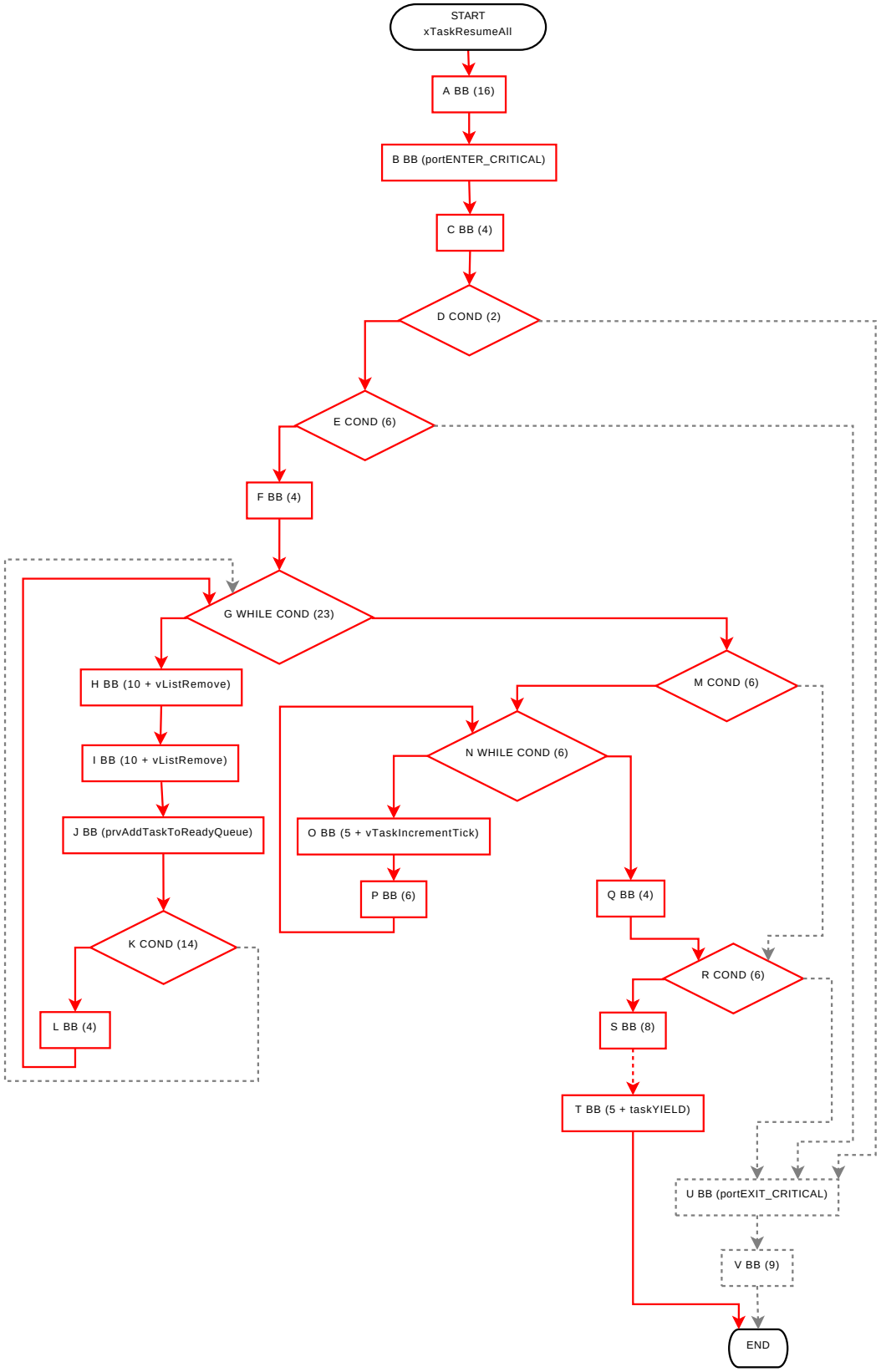
Obr. A.7: CFG pre `xTaskIsTaskSuspended`



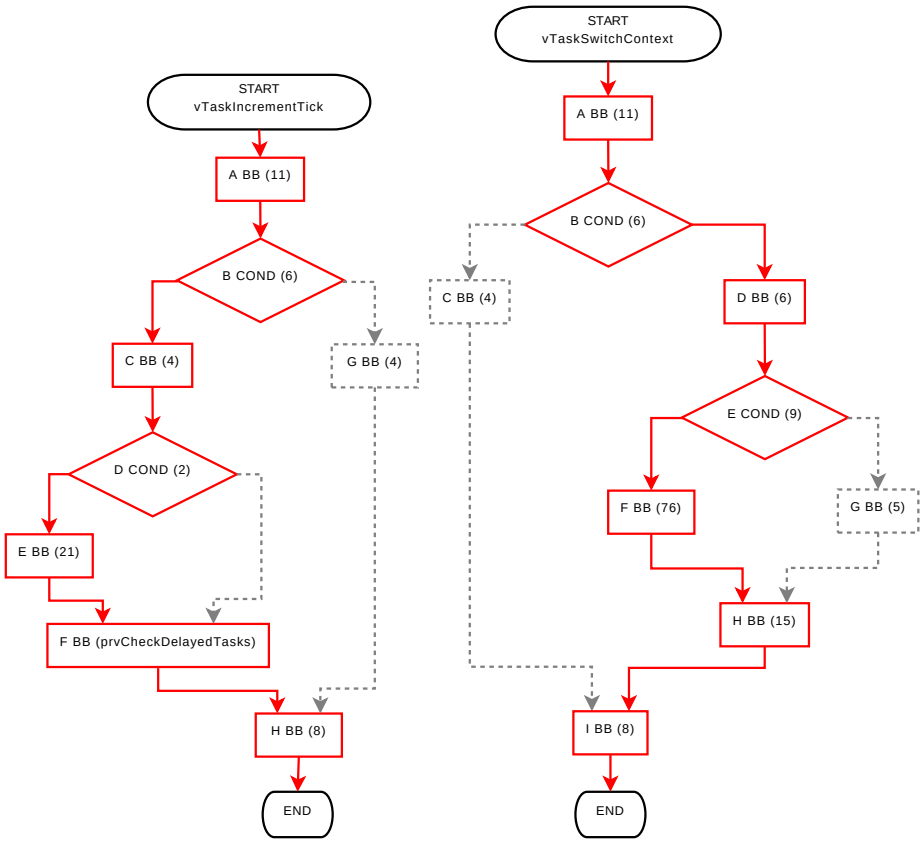
Obr. A.8: CFG pre `vTaskSuspend` a `vTaskResume`



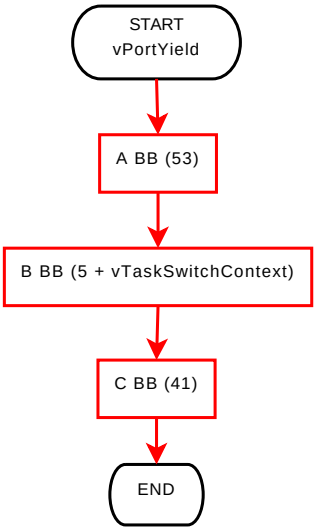
Obr. A.9: CFG pre `vTaskSuspendAll`



Obr. A.10: CFG pre `xTaskResumeAll`



Obr. A.11: CFG pre `vTaskIncrementTick` a `vTaskSwitchContext`



Obr. A.12: CFG pre `vPortYield`

Literatúra

[1] T. Bell. The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference*, pages 216–234, 1999.

[2] F. Cottet, J. Delacroix, C. Kaiser, and et al. *Scheduling in Real-Time Systems.* John Wiley & Sons, 2002.

[3] J. Eriksson, A. Dunkels, N. Finne, and et al. Poster abstract: Mspsim – an extensible simulator for msp430-equipped sensor boards. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, 2007.

[4] A. Ermedahl, F. Stappert, and J. Engblom. Clustered worst-case execution-time calculation. *IEEE Transactions on Computers*, 54(9):1104–1122, 2005.

[5] C. J. Fidge. Real-time schedulability tests for preemptive multitasking. *Real-Time Systems*, 14(1):61–93, 1998.

[6] Texas Instruments. Msp430x15x, msp430x16x, msp430x161x mixed signal microcontroller datasheet. `http://focus.ti.com/lit/ds/symlink/msp430f168.pdf`, 2007.

[7] Texas Instruments. Msp430x1xx family user's guide. `http://focus.ti.com/lit/ug/slau049f/slau049f.pdf`, 2007.

[8] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[9] R. Kirner and P. Puschner. Classification of wcet analysis techniques. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2005.

[10] R. Kirner and P. Puschner. Obstacles in worst-case execution time analysis. In *Proceedings of the 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 333–339, 2008.

[11] P. A. Laplante. *Real-Time Systems Design and Analysis.* John Wiley & Sons, 2004.

[12] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[13] D. Sandell. Evaluating static worst-case execution-time analysis for a commercial real-time operating system. Master's thesis, Mälardalen University, Sweden, July 2004.

[14] J. Strnadel. *Studijní opora k předmětu Real-Time operační systémy.* FIT VUT v Brně, 2006.

[15] WWW stránky. Fitkit. `http://merlin.fit.vutbr.cz/FITkit/`, 2007.

[16] WWW stránky. Freertos. `http://www.freertos.org`, 2008.

[17] K. Tindell and H. Hansson. Real time systems and fixed priority scheduling. Department of Computer Systems, Uppsala University, 1995.

[18] R. Wilhelm, J. Engblom, A. Ermedahl, and et al. The worst-case execution time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 5(3):1–53, 2008.