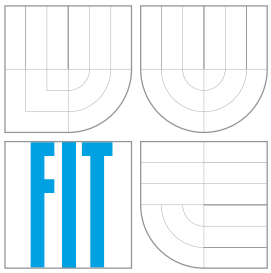
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE REAL-TIME OPERAČNÍHO SYSTÉMU μC/OS-II NA PLATFORMĚ ARM CORTEX-M4

IMPLEMENTATION OF μC/OS-II REAL-TIME OPERATING SYSTEM ON ARM CORTEX-M4 PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — MIKHAIL ANISIMOV
AUTHOR

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

BRNO 2016

Zadání diplomové práce/16090/2015/xanisi00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Anisimov Mikhail, Ing.**
Obor: Počítačové a vestavěné systémy
Téma: **Implementace real-time operačního systému uC/OS-II na platformě ARM Cortex-M4**
Implementation of uC/OS-II Real-Time Operating System on ARM Cortex-M4 Platform
Kategorie: Operační systémy

Pokyny:

1. Detailně se seznamte s architekturou ARM Cortex-M4 (dále jen "ARM") a tvorbou aplikací založených na ARM.
2. Seznamte se se základními pojmy z oblasti real-time (RT) operačních systémů (RTOS) a vytvořte přehled "open-source" RTOS volně použitelných pro nekomerční účely.
3. Klíčové rysy z bodů 1 a 2 přehledově shrňte.
4. Seznamte se s architekturou operačního systému uC/OS-II a analyzujte implementační nároky kladené na ARM. Vytvořte procesorově specifické hlavičkové/zdrojové soubory/kódy pro ARM, ověřte funkčnost jimi poskytovaných služeb a vytvořte k nim dokumentaci.
5. Přeneste jádro uC/OS-II na ARM. Funkčnost přeneseného jádra demonstrujte nejprve izolovaně a poté pomocí sady několika vhodně zvolených praktických řídicích RT aplikací běžících nad tímto jádrem.
6. Zhodnoťte dosažené výsledky.

Literatura:

- Cortex-M4 Processor [on-line]. c2013. [cit. 2013-09-04]. Dokument dostupný na http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php.
- Micrium - Empowering Embedded Systems [on-line]. c2012. [cit. 2012-09-10]. Dokument dostupný na http://www.micrium.com.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Strnadel Josef, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Tato diplomová práce se zabývá zprovozněním operačního systému reálného času μC/OS-II na platformě FITkit 3, jeho testováním a prokázáním jeho funkčnosti na jednoduchých příkladech. Popisuje příklad využití systému μC/OS-II pro aplikaci zobrazení obrázků na E-ink displeji a aplikaci metody ECCA pro zvýšení spolehlivosti systému.

Abstract

This Master's project deals with implementation of μC/OS-II real-time operating system on FITkit 3 platform, its testing and proving its functionality with simple examples. Describes an example of μC/OS-II application for displaying images on a E-ink display and application of ECCA method for increasing fault tolerance of the system.

Klíčová slova

μC/OS-II, Operační systémy, Vestavěné systémy, FITkit 3 Minerva, Systémy reálného času, ARM Cortex-M4, ECCA, E-ink displej, SD-karta

Keywords

μC/OS-II, Operating systems, Embedded systems, FITkit 3 Minerva, Real-time systems, ARM Cortex-M4, ECCA, E-ink display, SD-card

Citace

ANISIMOV, Mikhail. *Implementace real-time operačního systému μC/OS-II na platformě ARM Cortex-M4*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Strnadel Josef.

Implementace real-time operačního systému μC/OS-II na platformě ARM Cortex-M4

Prohlášení

Prohlašuji, že jsem tento diplomový projekt vypracoval samostatně pod vedením pana Ing. Josefa Strnadela, Ph.D. Uvedl jsem všechny zdroje, ze kterých jsem čerpal.

.....
Mikhail Anisimov
25. května 2016

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Josefu Strnadelovi, Ph.D., za poskytnutou pomoc během vypracování tohoto projektu

Obsah

Kapitola 1

Úvod

Všechny počítače pracují v reálném světě a řídí reálné procesy. Jedním s základních měřítek reálného světa je čas. Systém, který je navržen s ohledem na čas, se nazývá systém reálného času nebo RT systém (Real-Time system). Většinou se netriviální RT systém neimplementuje přímo na hardwarové platformě, to by vyžadovalo náročný návrh a následnou verifikaci. Proto se jako základní softwarová vrstva používají ověřené a certifikované operační systémy reálného času. Jedním s takovýchto operačních systému je systém μC/OS II, implementací kterého na platformě FITkit 3 se budeme zabývat v rámci této práce.

Ve druhé kapitole se seznámíme s základními pojmy a typy systémů reálného času. Také obsahuje přehled několika systému používaných v reálných aplikacích.

V třetí kapitole popíšeme hardwarové a softwarové prostředky, které byly použitý. Popíšeme systém μC/OS I, prostředí Kinetis Design Studio a platformu FITkit 3, podrobněji se zaměříme na mikrokontrolér MK60DN512VMD10 a procesor ARM Cortex-M4.

Ve čtvrté kapitole je popsána portace μC/OS II na kontrolér MK60DN512VMD10, jsou vysvětleny platformě závislé funkce systému a jejich implementace.

V páté kapitole se zaměříme na testování portu systému. Na příkladě několika jednoduchých demo aplikací prokážeme funkčnost systémových volání a ukážeme základy tvorby RT aplikací pro systém μC/OS II.

Šestá kapitola popisuje příkladovou aplikaci nad jádrem μC/OS II pro zobrazení bitmapy na E-ink displeji a příklad rozšiřující jádro systému o kontrolu toku řízení metodou Enhanced Control-Flow Checking Using Assertions pro zvýšení spolehlivosti.

Kapitola 2

Real-time systémy

V této kapitole uvedeme nejdůležitější pojmy z oblasti systémů reálného času. Terminologie a příklady jader byly převzaty z [11].

2.1 Základní pojmy

Pojem **systém** můžeme vyjádřit jako černou skříňku, která provádí mapování vstupních vektorů na výstupní. **Doba odezvy** systému je čas, který uplyne mezi výskytem podnětu (vstupní vektor) a odpovědí systému příslušným výstupem. **Systém reálného času** je takový systém, ve kterém nedodržení předem stanovených mezních dob odezev na vstupní vektory, stejně jako nekorektní hodnota na výstupu, je považován za selhání celého systému. Jednou z typických reprezentací real-time systému je model posloupnosti úloh.

Proces nebo **úloha** je abstrakce běžícího programu a jednotka plánování operačního systému. Proces je typicky reprezentován datovou strukturou, která obsahuje stav vykonávání, identifikaci, vlastnosti a používané zdroje. Jádro real-time systému musí poskytovat tři specifické funkce vzhledem k úlohám: plánování, přepnutí kontextu a komunikace/synchronizace. Plánovač zjišťuje která úloha poběží další a dispečer zajistí přepnutí kontextu.

Podle vážnosti následků, ke kterým může vést selhání systému, rozlišujeme hard, soft a firm real-time systémy. Příklady těchto systémů

- **Soft systém.** Automat na vydání pořadového lístku na poště. Nedodržování časových mezí nevede ke katastrofálním následkům.
- **Firm systém.** Automat řídící proces na výrobní lince. Pokud bude systém selhávat jen zřídka, vyřazení vadných výrobků zajistí kontrola kvality. Při částem selhávání systém degraduje a může způsobovat finanční ztráty, ale nenastane nic co by ohrožovalo zdraví a život zaměstnanců.
- **Hard systém.** Systém kontroly teploty reaktoru na jaderné elektrárně. Zpožděná odezva může vést k výbuchu a ztrátám na lidských životech.

Všechny události systému můžeme rozdělit na **synchronní**, jejichž výskyt můžeme předvídat, a **asynchronní** (nepředvídatelné). Každou z těchto skupin můžeme rozdělit na **periodické**, **aperiodické** (nevyskytují se pravidelně) a **sporadické** (aperiodické se známým nejkratším intervalem) události.

2.2 Typy jader

2.2.1 Vyzývací smyčka

Vyzývací smyčka (Polling loop) se využívá v případech potřeby rychlé odezvy na jeden podnět.

Předpokládejme například systém pro zpracování paketů dat od nějakého zařízení. V okamžiku kdy přijde nový paket, proměnná *packet_here* je nastavena na 1 řadičem DMA. Program testuje hodnotu této proměnné v nekonečné smyčce a začíná zpracovaní když je nastavena na 1. Daná úloha v jazyce C může vypadat takto:

```
for(;;) {
   if(packet_here)
   {
      process_data();
      packet_here = 0;
   }
}
```

Kladnými vlastnostmi takového systému je krátká doba odezvy a jednoduchost analýzy. Záporné vlastnosti jsou plýtvání cykly procesoru na dotazování a složitost návrhu pro složitější systémy s několika podněty.

2.2.2 Cyklické provádění

Systémy cyklického provádění (cyclic executives) jsou systémy neřízené přerušeními ale poskytující iluzi souběhu několika úloh. Například pro systém s N procesy:

```
for(;;) {
   process_1();
   process_2();
   process_3();
   ...
   process_N();
}
```

Pokud potřebujeme zmenšit periodu provádění některé z úloh, můžeme proložit její volání. Například pro úlohu process_3:

```
for(;;) {
   process_1();
   process_3();
   process_2();
   process_3();
}
```

Mezi problémy takového systému patří potřeba rozdělení na přibližně stejně trvající úlohy a dlouhá doba odezvy v nejhorším případě (úloha musí čekat až všechny předcházející úlohy dokončí svoji práci).

2.2.3 Stavově řízený kód

Stavově řízený kód je implementací konečného stavového automatu. Úloha je rozdělena na úseky kódu, které se vykonávají postupně. Příklad implementace pomocí příkazu *switch* jazyka C:

```
   for(;;) {
      switch(state)
      {
       case 1: task_1();break;
       ...
       default:
      }
   }
```

Výhodou takového systému je to, že teorie konečných stavových automatu je dobře prozkoumána a dají se použít formální matematické metody pro analýzu. Ne každá úloha však může být vhodně reprezentována pomocí KA.

2.2.4 Spolupracující úlohy

Model spolupracujících úloh (coroutines) je vylepšením stavově řízeného kódu. V tomto systému jsou úlohy rozděleny na fáze a po dokončení každé fáze je vyvolán plánovač, který uloží stav zastavené úlohy a vybere další běžící úlohu podle daného mechanismu. Příklad dvou úloh v systému coroutines:

```
void process_a() {
   for(;;) {
      switch(state_a)
      {
       case 1: phase_a1();break;
       case 2: phase_a2();break;
       ...
       default:
      }
   }
}
void process_b() {
   for(;;) {
      switch(state_b)
      {
       case 1: phase_a1();break;
       case 2: phase_a2();break;
       ...
       default:
      }
   }
}
```

Výhodou je, že tento systém zajišťuje souběh několika úloh (Multitasking). Problémem je samotný návrh úloh. Programátor musí vhodně rozdělit úlohy na fáze s ohledem na dobu zpracování každé fáze a požadovanou dobu odezvy.

2.2.5 Jádra řízená přerušeními

V systémech řízených přerušeními obsahuje hlavní program *main* inicializaci systému a prázdnou nekonečnou smyčku. Všechna práce je vykonávána ve funkcích obsluh přerušení. Priority vykonávání úloh je možné řídit nastavením priorit přerušení. Příklad systému řízeného přerušeními.

```
void main()
{
   init();
   while(TRUE);
}
void int1()
{
  save(context);
  task1();
  restore(context);
}
...
void intN()
{
  save(context);
  taskN();
  restore(context);
}
```

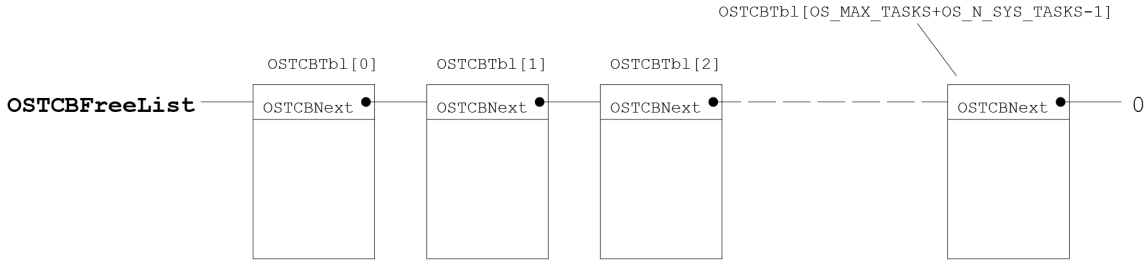
Kladnou vlastností takovéhoto systému je jednoduchost návrhu (přehlednost kódu úloh) a podpora uložení stavu přerušené úlohy v hardwaru. Záporné vlastnosti jsou plýtvání cykly procesoru v nekonečné smyčce a omezený počet úloh.

2.2.6 Systémy popředí/pozadí

Systémy s úlohami na popředí a pozadí (foreground/background) je vylepšením modelu řízeného přerušeními. Problém plýtvání výkonem procesoru v nekonečné smyčce je vyřešen tím, že funkce *main* na pozadí provádí užitečnou práci, která nemusí mít rychlou dobu odezvy.

2.2.7 Systémy založené na TCB

Model založený na TCB (Task Control Block) je používán ve většině dostupných real-time operačních systémů, včetně systému uC/OS II. V systémech založených na TCB je každá úloha reprezentována odpovídající datovou strukturu v jádře systému. Ta obsahuje stav vykonávání, identifikaci, vlastnosti a používané zdroje. Systém ukládá TCB úloh do jedné nebo několika datových struktur v jádře (typicky do vázaných seznamů), například seznamů všech úloh, seznamů úloh připravených k běhu atd.
Například některé položky TCB pro systém μC/OS II, představené jako struktura jazyka C, jsou:



Obrázek 2.1: Vázaný seznam volných TCB [10].

```
struct os_tcb {
    OS_STK *OSTCBStkPtr; // ukazatel na vrchol zásobníku
    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;
    INT16U OSTCBDly;
    INT8U OSTCBStat; // stav úlohy
    INT8U OSTCBStatPend;
    INT8U OSTCBPrio; // priorita
    ...
}
```

Pomocí ukazatelů *OSTCBNext a *OSTCBPrev všechny TCB existujících úloh tvoří obousměrně vázaný seznam. Průchod tímto seznamem například používá funkce OSTimeTick(), která dekrementuje hodnotu OSTCBDly. Tato hodnota udává počet cyklů SysTick Timeru, během kterých bude úloha uspána. Pole OSTCBTbl[] obsahuje TCB všech úloh a má velikost rovnou hodnotě OS_MAX_TASKS definovanou v souboru OS_CFG.H. Tuto hodnotu se dá nastavit na maximální počet úloh, který systém potřebuje, pro snížení paměťových nároku systému. Všechny nevyužité TCB jsou uchované v seznamu OSTCBFreeList 2.1.

2.3 Přehled operačních systémů reálného času

V současné době existuje velké množství systémů reálného času. V této podkapitole zmíníme jenom několik z nich. Seznam známých RT operačních systému můžete najít například na stránkách Wikipedie [1]. Všechny uvedené dále systémy mají otevřené zdrojové kódy.

FreeRTOS[2] je operační systém pro vestavěné zařízení. Existují porty tohoto systému pro více než 35 mikrokontrolérů. Byl navržen jako malý jednoduchý systém, binární obraz systému má velikost pouze 4-9 KB.

MenuetOS[3] je RT systém pro platformu x86 a je zajímavý tím, že celé jádro bylo napsáno kompletně v assembleru. Vyžaduje procesor Intel architektury P5 nebo vyšší, 32 MB RAM paměti, VESA 2.0 kompatibilní videoadaptér, COM nebo PS/2 myš , ROM obraz může být umístěn na 1.44 MB floppy-disk. Systém obsahuje stack protokolů TCP/IP a ovladače pro USB 2.0 zařízení. 32-bitová verze je zveřejněna s licenci GPL, 64-bitová verze však nemá volně dostupné zdrojové kódy.

μC/OS II a **μC/OS III**[4]. **μC/OS III** je rozšířením druhé verze systému. Vylepšení

[1] https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems
[2] http://www.freertos.org
[3] http://www.menuetos.net
[4] http://www.micrium.com

zahrnují například neomezený počet úloh a počet úloh na stejné prioritní hladině, systém může být konfigurován za běhu systému a jiné. Dále v teto práci se podrobněji zaměříme na systém μC/OS II.

Kapitola 3

Realizační prostředky

3.1 Platforma FITkit 3

Pro řešení projektu byla vybrána platforma FITkit 3 Minerva. Daná platforma obsahuje mikrokontrolér Kinetis MK60DN512VMD10 od firmy Freescale s procesorem ARM Cortex-M4, FPGA obvod z rodiny Spartan-6 od firmy Xilinx, periferii pro propojení platformy přes rozhraní USB, HDMI, Ethernet, 5 tlačítek a 4 diody LED, připojené na vstupy mikrokontroléru. Dále podrobněji popíšeme procesor ARM Cortex-M4 a mikrokontrolér, který byl použit v řešení projektu.

3.1.1 Procesor ARM Cortex-M4

ARM Cortex-M4 je 32-bitový RISC procesor. Má 3 stupňovou pipeline, a je založen na Harvardské architektuře, což dovoluje souběžné načtení instrukce a přístup k paměti. Pro programování se používá instrukční sada Thumb-2, která obsahuje 32 i 16-bitové instrukce pro zmenšení velikosti výsledného kódu bez ztráty rychlosti zpracování. Kromě samotného výpočetního jádra procesor také obsahuje řadič přerušení, který podrobněji popíšeme v jedné z dalších podkapitol, a systémový časovač pro vyvolávání periodických přerušení.

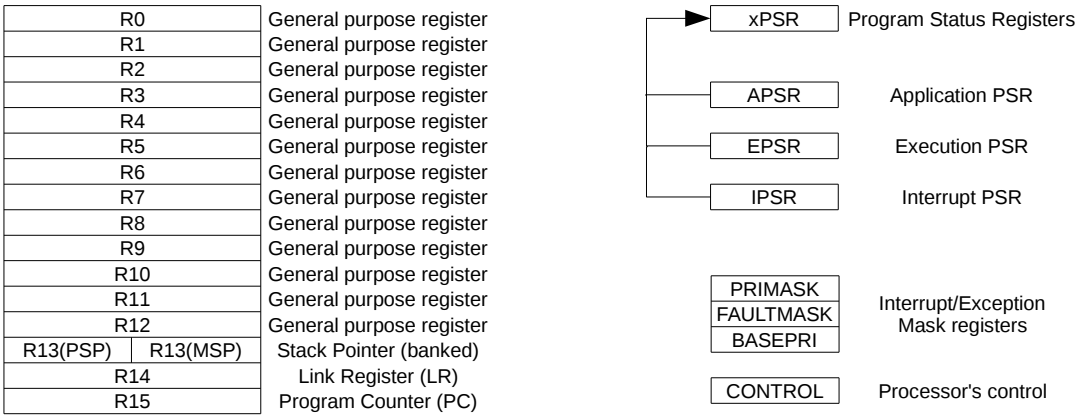
Programovací model

Během své činnosti se procesor může nacházet ve dvou stavech: stav *Ladění* a *Thumb* stav. Do stavu debugování může být převeden buď debuggerem nebo při dosažení breakpointu. V tomto stavu je zpracování instrukcí pozastaveno. Ve stavu *Thumb* procesor vykonává instrukce, tento stav je rozdělen na dva režimy: Handler mode a Thread mode. Režim Handler slouží pro zpracování přerušení a režim Thread pro vykonávání aplikačního kódu. V každém z těchto režimu může být použit jiný stack-pointer.

Registrová sada ARM Cortex-M4 obsahuje 12 obecných registrů R0-R12, ukazatel zásobníku (R13 nebo SP), registr návratové adresy (link register, R14 nebo LR), programový čítač (R15 nebo PC). Všechny registry jsou 32-bitové.

Kromě registrů v registrové sadě ještě existují speciální registry, které obsahují stav procesoru, definují operační režim a masku přerušení. Sada registrů je znázorněna na obrázku 3.1.

Registr xPSR se mapuje na registry APSR, EPSR, IPSR. Jeho struktura je na obrázku 3.2. N, Z, C, V a Q jsou příznaky procesoru (negative, zero, carry, overflow, sticky saturation).



Obrázek 3.1: Sada registrů a specialní registry ARM Cortex-M4.

31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
N	Z	C	V	Q	ICI/IT	T		GE	ICI/IT		Exception Number				

Obrázek 3.2: Registr xPSR.

Registr PRIMASK je jednobitový a obsahuje hodnotu 1, pokud jsou zakázány všechny výjimky, kromě nemaskovatelného přerušeni (NMI) a HardFault exception. Registr FAULTMASK se chová podobně s tím rozdílem, že při hodnotě 1 je navíc zakázána HardFault exception. Registr BASEPRI zakazuje výjimky se stejnou nebo menší prioritou než je jeho hodnota. Pokud obsahuje 0, nemá na výjimky žádný vliv.

Registr CONTROL definuje aktuálně používaný stack-pointer (MSP nebo PSP).

Procesor ARM Cortex-M4 má také sadu floating-point registrů, ale v tomto projektu nejsou využívány. Jejich popis můžete najít například v [13].

Systém přerušení

Pro řízení přerušení obsahuje procesor vestavěný řadič přerušení (Nested Vectored Interrupt Controller). Řadič má následující zdroje přerušení: periferní zařízení uvnitř mikrokontroléru, vnitřní systémový časovač, vnější zdroj přes vstup-výstupní porty a systémové výjimky samotného jádra procesoru. Proces obsluhy přerušení probíhá následujícím způsobem :

1. Zdroj přerušení posílá žádost řadiči NVIC.
2. NVIC podle nastavení vnitřních registru (BASEPRI, PRIMASK, priorita zdrojů atd.) a aktuálního stavu procesoru (priorita aktuálně obsluhovaného přerušení) rozhoduje zda přerušení bude ignorováno, akceptováno, nebo bude jeho obsluha odložena.
3. Procesor pozastavuje běžící program, zapisuje hodnoty registrů R0-R3, R12, LR a xPSR na zásobník a načítá do registru PC hodnotu z tabulky vektorů přerušení odpovídající adrese funkce obsluhy.
4. Po vykonání funkce obsluhy vrací původní stav registrů R0-R3, R12, LR a xPSR ze zásobníku a pokračuje vykonávání programu.

NVIC dovoluje zanořování úrovní přerušení, pokud během vykonávání funkce obsluhy přijde žádost o přerušení s vyšší prioritou. ARM Cortex-M4 podporuje efektivní řetězené zpracování přerušení (tail chaining). Tento mechanismus spočívá v tom, že když byl procesor v průběhu zpracování přerušení požádán o přerušení s nižší prioritou, nevrací se po dokončení zpracování do původního programu, ale hned začíná obsluhu druhého přerušení bez nutnosti vracení hodnot registrů ze zásobníku a jejich následného zápisu.

Při řešení tohoto projektu byly použité následující zdroje přerušení: vstup-výstupní port pro práci s tlačítky, systémový časovač pro zahájení periodického vykonávání kódu jádra operačního systému, výjimka procesoru PendSV pro provedení operace přepnutí kontextu. Pokud je nastavena nejnižší priorita spustí se její zpracování pouze po dokončení všech ostatních přerušení.

Výjimka PendSV (Pended Service Call) může být vyvolána softwarově, nastavením žádosti v registru NVIC:

```
.equ NVIC_INT_CTRL,   0xE000ED04   @ Interrupt control state register.
.equ NVIC_PENDSVSET,  0x10000000   @ Value to trigger PendSV exception.
LDR     R0, =NVIC_INT_CTRL         @ Trigger the PendSV exception.
LDR     R1, =NVIC_PENDSVSET
STR     R1, [R0]
```

3.1.2 Kontrolér Kinetis MK60DN512VMD10

Kinetis MK60DN512VMD10 je 32-bitový mikrokontrolér. Je implementován v pouzdře MAPBGA a obsahuje na čipu procesor ARM Cortex-M4 a velký počet periferních zařízení. V této podkapitole uvedeme jen nejdůležitější vlastnosti kontroléru a periferie použité v projektu:

- Nejvyšší frekvence hodin procesoru - 100 MHz.
- Velikost Flash paměti pro program - 512 KB.
- Velikost SRAM paměti - 128 KB.
- 5 Vstup/Výstupních portů GPIO.
- Secure Digital Host Controller (SHDC).
- 3 moduly SPI.
- 5 modulů sériové linky (UART).
- 2 ADC převodníky.
- 3 moduly FlexTimer, které mohou být použity pro generování PWM signálu.

Podrobný popis mikrokontroléru můžete najít v dokumentaci od firmy Freescale [5].

3.2 Operační systém μC/OS II

μC/OS II je přenositelný, škálovatelný, preemptivní a víceúlohový real-time operační systém, založený na TCB. Systém je portován na více než 40 různých platforem. μC/OS II

se používá například v leteckých zařízeních, kamerách, audio zařízeních, medicínských nástrojích, síťových adaptérech, bankomatech aj. Používá se pro výuku real-time systémů na mnoha školách světa a je volně dostupný na stránkách firmy Micrium[1]. Pro komerční použití je však nutná licence.

Následně popíšeme nejdůležitější prostředky, které poskytuje systém μC/OS II pro programátora aplikací.

3.2.1 Správa úloh

Úloha v μC/OS II je představena funkcí jazyka C s návratovou hodnotou typu *void* a s jedním parametrem - ukazatelem na typ *void*. Tento parametr ukazuje na data, která předá do úlohy uživatel při její vytváření funkcí OSTaskCreate() nebo OSTaskCreateExt()[2]. Díky tomu, že každá úloha má vlastní zásobník a tudíž lokální proměnné, uživatelský parametr může být použit pro identifikaci úloh, které sdílejí stejný kód. Každá úloha μC/OS II musí buď obsahovat nekonečnou smyčku nebo mazat sama sebe pomocí funkce OSTaskDel(OS_PRIO_SELF) před návratem. Příklady korektních úloh:

```
void Task1(void *pdata)
{
   while(1)
   {
      OSTimeDly(5);
      ProcessData(); /* User function */
   }
}
void Task2(void *pdata)
{
   ProcessData2();
   OSTaskDel(OS_PRIO_SELF);
}
```

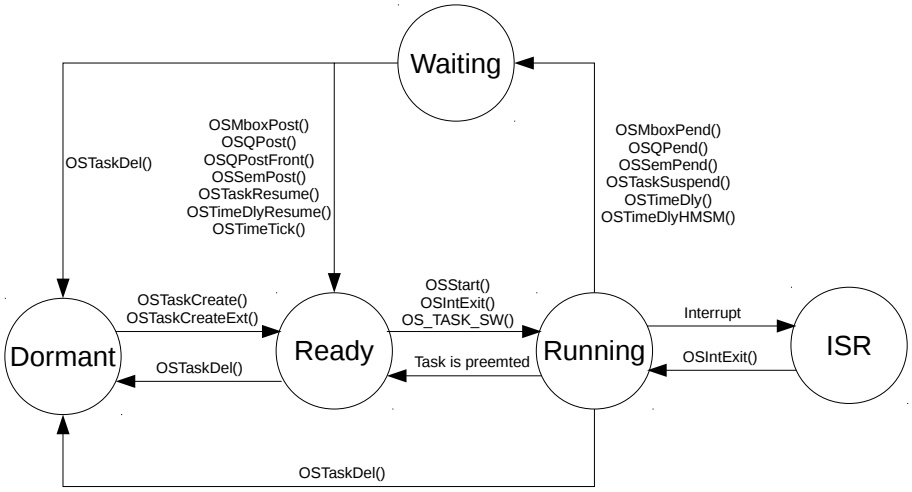
Úloha se může nacházet v jednom z těchto stavů:

- **Ready.** Úloha byla vytvořena, je připravena k běhu ale existuje jiná připravená úloha s vyšší prioritou, která momentálně běží.
- **Running.** Úloha má nejvyšší prioritu mezi připravenými úlohami a momentálně běží.
- **Waiting.** Úloha byla pozastavena a čeká na událost.
- **Dormant.** Úloha byla smazána funkcí OSTaskDel(). Kód úlohy zůstává v paměti ale operační systém přestal ji zpravovat a uvolnil všechny odpovídající struktury jádra (TCB, seznam připravených úloh atd.).
- **ISR running.** Běžící úloha byla pozastavena přerušením.

Stavový automat úlohy a funkce, které mohou převést k přechodu, jsou znázorněny na obrázku 3.3.

[1] http://www.micrium.com

[2] Všechny funkce a vnitřní proměnné systému μC/OS II mají prefix OS. Podrobný popis těchto a jiných funkcí zmíněných v tomto textu můžete najít v uživatelské příručce [10]



Obrázek 3.3: Stavy úloh v μC/OS II [10].

Idle a statistická úloha

μC/OS II má dvě vnitřní systémové úlohy: idle úloha existuje vždy od začátku běhu systému, přítomnost statistické úlohy může být nastavená uživatelem v souboru OS_CFG.H. Idle úloha má nejnižší prioritu a běží pouze pokud neexistuje žádná další připravená k běhu. Tato úloha obsahuje nekonečnou smyčku, ve které se inkrementuje proměnná OSIdleCtr. Idle úloha může být použita pro uspání procesoru pro snížení spotřeby energie nebo pro vykonávání práce na pozadí, která nemá omezení na dobu odezvy.

Statistická úloha se vykonává každou vteřinu a používá proměnnou OSIdleCtr pro výpočet zatížení procesoru. Globální proměnná OSCPUUsage udává zatížení procesoru v procentech.

3.2.2 Správa času

Pro svoji práci systém μC/OS II potřebuje zdroj periodického přerušení. Pro tento účel na architektuře ARM Cortex-M4 můžeme použít systémový časovač (SysTick Timer). Frekvence přerušení se obvykle volí v rozmezí 10-1000 Hz.

Operační systém nabízí několik systémových volaní pro práci s časem:

- OSTimeDly(), OSTimeDlyHMSM() - pozastavit úlohu na dobu, kterou můžeme zadat jako parametr. Počet přerušení nebo čas ve formátu HMSM (Hodina, Minuta, Sekunda, Mikrosekunda).
- OSTimeDlyResume() - pokračování pozastavené úlohy bez vypršení času nastaveného jednou z předchozích funkcí.
- OSTimeGet(), OSTimeSet() - čtení a nastavení systémového času. μC/OS II uchovává hodnotu systémového času v proměnné OSTime, která se inkrementuje s každým přerušením od systémového časovače. Při startu systému tato proměnná je vynulována.

3.2.3 Komunikační a synchronizační prostředky

V systému μC/OS II jsou implementovány standardní synchronizační prostředky - semafory a mutexy. Pro komunikaci mezi úlohami se používají schránky (Message Mailbox) a fronty

zpráv (Message Queue). Mutexy v μC/OS II mají kromě vlastností binárních semaforů zabudovaný mechanismus pro zabránění vzniku inverze priorit Priority Celling Priority (PCP). Funkčnost PCP spočívá v tom, že se priorita úlohy s nízkou prioritou, která vlastní mutex, zvyšuje na prioritu vyšší než priorita úlohy, která čeká na uvolnění mutexu.

3.2.4 Správa paměti

Systém μC/OS II nabízí systémové volání OSMemGet() pro dynamickou alokaci paměti, protože použití standardních funkcí ANSI C malloc() a free() není bezpečné z hlediska determinismu doby jejich vykonávání. Algoritmus používaný ve funkci malloc() může selhat kvůli fragmentaci. Mechanismus použitý v μC/OS II zabraňuje fragmentaci a spočívá v tom, že nejdřív si programátor připraví pomocí funkce OSMemCreate() souvislý oddíl paměti (partition), který je rozdělen na bloky stejné velikosti. V jazyce C základem oddílu může být dvourozměrné pole, ve kterém jedna dimenze definuje počet bloků a druhá dimenze jejich velikost. Pak funkcí OSMemGet() se bloky alokují a funkcí OSMemPut() se vracejí do seznamu volných bloků. Pokud aplikace potřebuje alokaci paměti různé velikosti, může být vytvořeno několik oddílů s různými velikostmi bloků.

3.2.5 Přerušení v μC/OS II

Pro korektní činnost systému μC/OS II v tělu funkce obsluhy přerušení musí být provedeny následující kroky:

1. Zápis všech registrů procesoru na zásobník.
2. Inkrementace proměnné OSIntNesting, která se používá ve mnoha funkcích systému na zjištění jestli funkce byla volaná z ISR nebo z kontextu úlohy.
3. Pokud přerušením byla pozastavena běžící úloha, zápis ukazatele zásobníku úlohy do jejího TCB.
4. Mazání bitu požadavku o přerušení.
5. Vykonávání uživatelského kódu pro zpracování přerušení.
6. Volání OSIntExit(), která dekrementuje OSIntNesting a pokud se rovná 0, zavolá plánovač a spustí se úloha s nejvyšší prioritou
7. Volání instrukce návratu z přerušení.

Kroky 1 a 3 vyžadují přímou manipulaci s registry a tudíž musí být implementovány v assembleru. Na architektuře ARM Cortex-M4 však funkce obsluhy přerušení může být napsána kompletně v jazyce C, protože některé registry jsou zapsány na zásobník automaticky při vyvolání přerušení (viz Kapitola 3.1.1), zachování hodnot zbylých registrů zajistí C kompilátor přidáním instrukcí pro zápis na zásobník registrů, které byly použité ve funkci podle standardu Procedure Call Standard for ARM Architecture [9]. Díky použití dvou různých zásobníků (PSP a MSP) pro kontext úlohy a kontext obsluhy přerušení, můžeme si být jistí že se hodnota PSP zachová do doby přepnutí kontextu na jinou úlohu a nebude přepsána v ISR. Tím pádem zápis ukazatele zásobníku můžeme provést až při přepnutí kontextu (viz Kapitola 4.3). Díky tomu, že při zahájení přerušení procesor ARM Cortex-M4 vkládá do registru návratové adresy LR speciální hodnotu EXC_RETURN, pro návrat z podprogramu obsluhy přerušení se používá obyčejná instrukce POP PC, která přepíše

hodnotu programového čítače na hodnotu návratové adresy. Tuto instrukci vkládá C kompilátor na konci každé funkce. Podle hodnoty EXC_RETURN procesor pozná, že jde o návrat z obsluhy přerušení.
Základní kostra podprogramu obsluhy přerušení pro procesor ARM Cortex-M4 vypadá následovně:

```
void PORTE_IRQHandler()
{
    OS_CPU_SR  cpu_sr;
    OS_ENTER_CRITICAL();
    OSIntNesting++;
    PORTE->ISFR = PORTE->ISFR; // mazání bitu požadavku o přerušení
    OS_EXIT_CRITICAL();
    // user_code
    OSIntExit();
}
```

3.2.6 Implementační nároky μC/OS II

Systém μC/OS II může být portován na danou platformu, pokud procesor splňuje následující požadavky:

- Pro procesor existuje C kompilátor, který generuje reentrantní kód.
- Podporuje přerušení a obsahuje zdroj periodického přerušení.
- Zásobníku může být použít na umístění do několika kilobytů dat.
- Instrukční sada má instrukce pro zápis a čtení obsahu registrů do/z paměti.

Systém je velice flexibilní z hlediska nároků na paměť. V souboru OS_CFG.H programátor může konfigurováním systému výrazně zmenšit velikost jeho ROM obrazu (například omezením počtu priorit úloh). Velikost požadované ROM paměti je v rozmezí 4 až 24 Kbytů. V závislosti na aplikaci velikost požadované RAM paměti je 1 Kbyte a více.

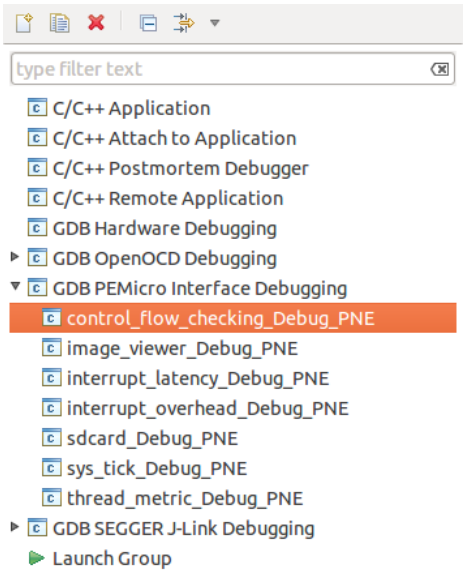
3.3 Softwarové prostředky

Pro kompilaci, ladění, nahrávání firmwaru do paměti, správu zdrojových souboru, interakci s vývojovou deskou přes USB, výpis výsledků na obrazovku byly použité některé aplikace pro PC. Popíšeme je v této podkapitole. Při práci na projektu byl použit PC s procesorem Intel Core 2 Duo 2.53 GHz s operačním systémem Ubuntu 14.04 LTS.

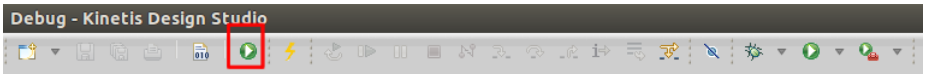
3.3.1 Kinetis Design Studio

Kinetis Design Studio (dále KDS) je volně dostupné vývojové prostředí od firmy NXP, založené na Eclipse a nástrojích GNU (GNU assembler, GCC, GDB) pro mikrokontroléry s jádrem ARM Cortex-M rodiny Kinetis. KDS obsahuje několik modulu, například generátor kódu pro periferie mikrokontroléru Processor Expert, jádro systému MQX a jiné. Podrobný popis nástroje a návod k instalaci můžete najít v uživatelské příručce [8].

Při vytváření projektu pro zjednodušení návrhu můžeme přidat do projektu několik důležitých zdrojových souboru. Je to startovací kód, který obsahuje inicializaci kontrolérů



Obrázek 3.4: Dialogové okno Debug Configurations.



Obrázek 3.5: Tlačítko Simple Run vyznačené červeně.

a tabulku vektorů přerušení, implementace knihovny CMSIS-core (Cortex Microcontroller Software Interface Standard) a hlavičkové soubory, ve kterých je popsán přístup k registrům periferií kontroléru v jazyce C pomocí ukazatelů na struktury.

Všechny příklady tohoto projektu jsou implementovány jako projekty prostředí KDS. Projekty jsou rozděleny do dvou Workspace, konkretní umístění a strukturu adresářů na CD můžete najít v příloze.

Pro vyzkoušení příkladu na vývojové desce FITkit 3 je třeba postupovat následně:

1. **Otevření Workspace.** Při spuštění KDS v dialogovém oknu Workspace Launcher vyberte adresář s Workspace.

2. **Překlad projektu.** V oknu Project Explorer vyberte potřebný projekt. V hlavním menu zvolte `Project->Build project`.

3. **Nahrávání do paměti mikrokontroléru.** V hlavním menu zvolte `Run->Debug Configurations`. V otevřeném dialogu napravo zvolte typ debugeru GDB PEMicro, název projektu (viz obrázek 3.4) a zmáčkněte tlačítko Debug.

4. **Spuštění.** Po přepnutí prostředí do stavu Debug zmáčkněte tlačítko Simple Run (viz obrázek 3.5).

3.3.2 Terminál

Komunikace mezi vývojovou deskou a PC, pro získávání výsledků a zadávání vstupů uživatelem, probíhá pomocí rozhraní USB. FITkit 3 obsahuje debugovácí obvod, který slouží

můstkem mezi SCI modulem mikrokontroléru a komunikační linkou USB. Pro snadný výpis formátovaného řetězce s využitím funkce printf() slouží modul firmwaru, který je popsán v souborech ve složce *terminal* každého projektu. Každý soubor, který používá printf() pro komunikaci přes sériovou linku musí obsahovat direktivu:

```
#include "io.h"
```

Na straně počítače může být použita libovolná aplikace, sloužící pro komunikaci přes sériovou linku. Například PuTTy pro Windows, nebo Minicom pro Linux.

Nastavení sériové linky jsou: 115200 bps, 8 bitů, bez parity, 1 stop bit (115200 8N1).

Kapitola 4

Port μC/OS II

Přesto, že systém μC/OS II je z větší části napsán v jazyce C a je maximálně přenositelný, pro portování systému na jinou platformu musí být doplněno několik platformě závislých funkcí. V tabulce 4.1 jsou uvedeny potřebné funkce, krátký popis jejich činnosti a soubor ve kterém jsou popsány.

Pro řešení tohoto projektu jako základ byl použít existující port operačního systému pro vývojovou desku TWR-K60N512 a sadu nástrojů IAR Embedded Workbench. Tento port můžete najít na stránkách firmy Micrium[1]. Následně zdrojové kódy byly opraveny pro použití s platformou FITkit 3 a vývojovým prostředím Kinetis Design Studio (dále KDS).

Dále podrobněji popíšeme jednotlivé platformě závisle části systému.

4.1 Kritická sekce

Port operačního systému nabízí tři metody implementace kritické sekce:

1. Zákaz přerušení při vstupu do kritické sekce a povolení při výstupu. Problém této metody spočívá v tom, že přerušení budou při výstupu povolena nezávisle na tom, jestli byla povolena před vstupem.

[1] http://micrium.com/download/micrium_twr-k60n512_ucos-ii/

Tabulka 4.1: Platformě závisle funkce

Název funkce nebo makra	Popis činnosti	Soubor
OS_CRITICAL_ENTER()	Vstup do kritické sekce	os_cpu.h
OS_CRITICAL_EXIT()	Výstup z kritické sekce	os_cpu.h
OSStartHighRdy()	Zahájení běhu první úlohy	os_cpu_a.S
OSCtxSw()	Zahájení přepnutí kontextu z úrovně úloh	os_cpu_a.S
OSIntCtxSw()	Zahájení přepnutí kontextu z úrovně obsluhy přerušení	os_cpu_a.S
PendSV_Handler()	Obsluha přerušení PendSV. Slouží pro přepnutí kontextu.	os_cpu_a.S
OSTaskStkInit()	Inicializace zásobníku při vytváření úlohy	os_cpu_c.c
OS_CPU_SysTickHandler()	Obsluha přerušení systémového časovače. Slouží pro volání funkce OSTimeTick()	os_cpu_c.c

2. Druhý způsob vylepšuje první tím, že při vstupu do KS se do zásobníku zapisuje současný stav registru PRIMASK, který je při výstupu obnoven. Problém nastává, když program potřebuje manipulovat se zásobníkem v kritické sekci.

3. Třetí metoda řeší tyto problémy tím, že zapisuje stav registru PRIMASK do lokální proměnné. Pro fungování této metody musí každá funkce, která používá kritickou sekci, obsahovat lokální proměnnou *cpu_sr* typu OS_CPU_SR.

4.2 Zahájení běhu první úlohy

Činnost funkce OSStartHighRdy() se skládá z následujících kroků:

1. Nastavení priority přerušení PendSV na nejnižší:
```
LDR R0, =NVIC_SYSPRI14 @ adresa příslušného registru NVIC
LDR R1, =NVIC_PENDSV_PRI @ hodnota nejnižší priority
STRB R1, [R0]
```

2. Nastavení hodnoty ukazatele zásobníku PSP na 0:
```
MOVS R0, #0
MSR PSP, R0
```

3. Nastavení hodnoty ukazatele zásobníku MSP:
```
LDR R0, =OS_CPU_ExceptStkBase
LDR R1, [R0]
MSR MSP, R1
```

4. Nastavení globální proměnné OSRunning na TRUE:
```
LDR R0, =OSRunning
MOVS R1, #1
STRB R1, [R0]
```

5. Zahájení přerušení PendSV:
```
LDR R0, =NVIC_INT_CTRL
LDR R1, =NVIC_PENDSVSET
STR R1, [R0]
```

6. Povolení všech přerušení:
```
CPSIE I
```

4.3 Přepnutí kontextu

Pro přepnutí kontextu musí být zahájeno přerušení PendSV. Díky tomu, že toto přerušení má nejnižší prioritu, můžeme použít stejný mechanismus pro přepnutí z funkce obsluhy přerušení a z režimu běhu úloh (víme že návrat z funkce bude vždy do režimu běhu úloh). To znamená, že pro funkce OSCtxSw a OSIntCtxSw můžeme použít stejný kód, který zahájí přerušení PendSV. Samotné přepnutí probíhá ve funkci obsluhy přerušení PendSV.

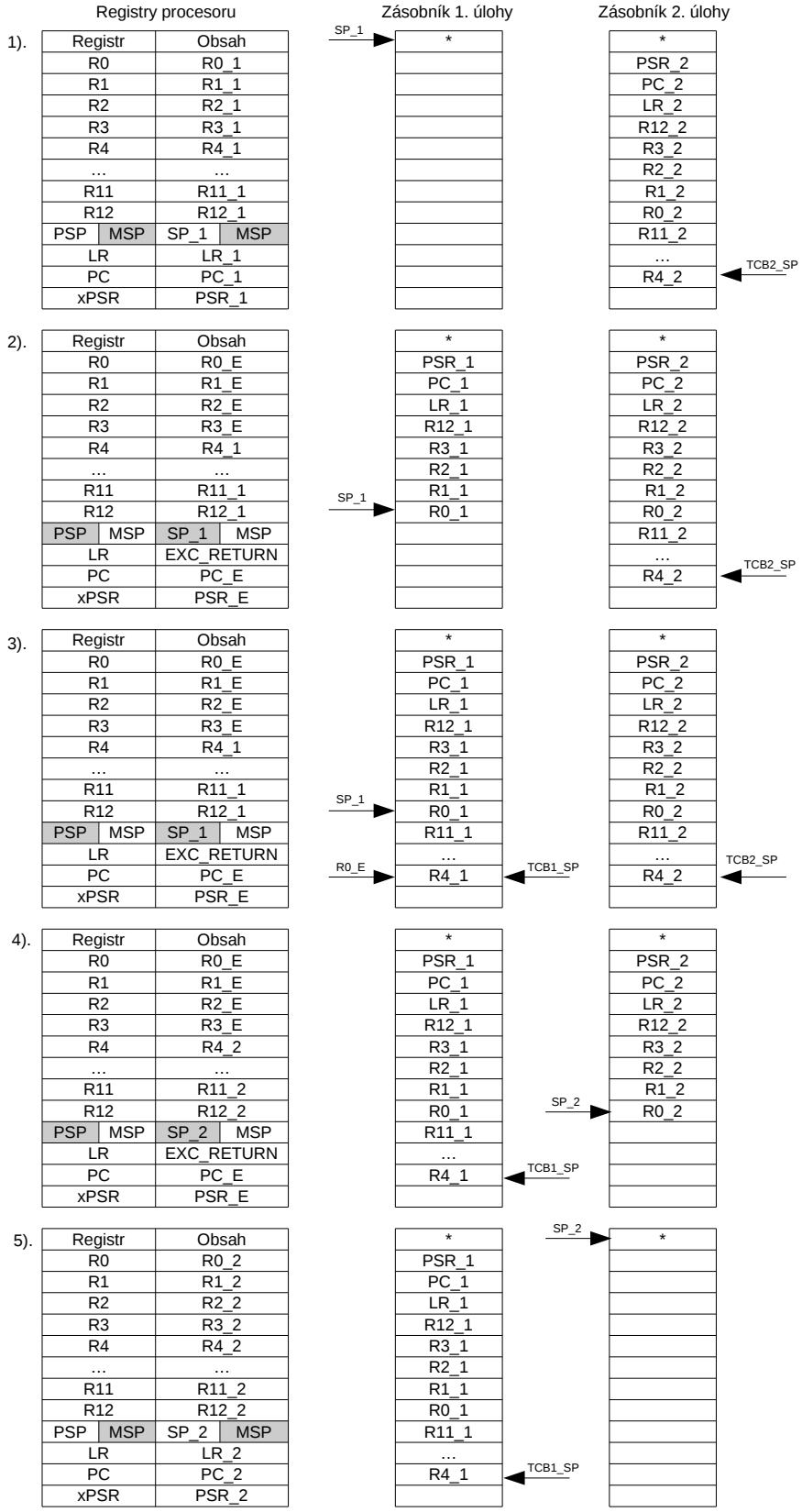
Při vstupu do funkce obsluhy přerušení PendSV_Handler() jsou registry xPSR, PC, LR, R12 a R0-R3 automaticky zapsány na zásobník, aktuálně používaný ukazatel zásobníku je přepnut z PSP na MSP, globální proměnná OSTCBCur ukazuje na TCB (Task Control Block) úlohy, která musí být pozastavena a proměnná OSTCBHighRdy ukazuje na TCB

úlohy, kterou vybral plánovač pro běh. Zdrojový kód funkce PendSV_Handler() vypadá následovně:

```
PendSV_Handler:
1.      CPSID   I
2.      MRS     R0, PSP
3.      CBZ     R0, OS_CPU_PendSVHandler_nosave
4.      SUBS    R0, R0, #0x20
5.      STM     R0, {R4-R11}
6.      LDR     R1, =OSTCBCur
7.      LDR     R1, [R1]
8.      STR     R0, [R1]
OS_CPU_PendSVHandler_nosave:
9.      PUSH    {R14}
10.     LDR     R0, =OSTaskSwHook
11.     BLX     R0
12.     POP     {R14}
13.     LDR     R0, =OSPrioCur
14.     LDR     R1, =OSPrioHighRdy
15.     LDRB    R2, [R1]
16.     STRB    R2, [R0]
17.     LDR     R0, =OSTCBCur
18.     LDR     R1, =OSTCBHighRdy
19.     LDR     R2, [R1]
20.     STR     R2, [R0]
21.     LDR     R0, [R2]
23.     LDM     R0, {R4-R11}
24.     ADDS    R0, R0, #0x20
25.     MSR     PSP, R0
26.     ORR     LR, LR, #0xF4
27.     CPSIE   I
28.     BX      LR
```

Činnost funkce PendSV_Handler() je znázorněná na obrázku 4.1. Šedou barvou je vyznačen aktuálně nepoužívaný ukazatel zásobníku. Jednotlivé kroky na obrázku jsou (v závorkách jsou uvedeny odpovídající řádky kódu z listingu výše):

1. Stav před zahájením přepnutí kontextu.
2. Stav po zahájení přerušení. Registry xPSR, PC, LR, R12, R3, R2, R1, R0 jsou zapsány na zásobník hardwarově.
3. Zápis registrů R4-R11 na zásobník úlohy (řádky 4-5). Zápis ukazatele zásobníku úlohy do jejího TCB (řádky 6-8).
4. Obnovení ukazatele zásobníku z TCB nové úlohy (řádek 21 a 25). Obnovení registrů R4-R11 ze zásobníku nové úlohy (řádky 23-24).
5. Stav po návratu z podprogramu obsluhy přerušení zápisem hodnoty EXC_RETURN do registru PC.



Obrázek 4.1: Obsah registrů procesoru a zásobníků úloh při přepnutí kontextu.

Obrázek 4.2: Obsah zásobníku po inicializaci.

Další řádky funkce PendSV_Handler() mají význam:

- **Řádky 2-3.** Pokud hodnota PSP se rovná 0 skočíme na řádek 9. Protože se provádí zahájení běhu první úlohy a proto nepotřebujeme ukládat registry předchozí úlohy a hodnota ukazatele OSTCBCur nemá význam.
- **Řádky 9-12.** Volaní funkce OSTaskSwHook(), kam uživatel může přidat svůj kód.
- **Řádky 13-20.** Nastavení proměnných OSPrioCur, OSTCBCur na hodnotu priority a adresu TCB nové úlohy.

4.4 Inicializace zásobníku úlohy

Před prvním spuštěním úlohy musí její zásobník vypadat jako po zápisu registrů při přepnutí kontextu. Inicializace se provádí ve funkci OSTaskStkInit(), která se volá při vytváření úlohy funkcí OSTaskCreate(), nebo OSTaskCreateExt(). Po návratu z této funkce musí být ukazatel zásobníku úlohy zarovnán na 8 bytů. Hodnota programového čítače PC na zásobníku úlohy musí být nastavená na adresu úlohy, návratový registr LR na adresu funkce OS_TaskReturn[2] a registr R0 se nastavuje na hodnotu parametru zadaného uživatelem při spouštění OSTaskCreate(). Uspořádání položek zásobníku úlohy po inicializaci je ilustrován na obrázku 4.2.

[2]Slouží pro mazání úlohy, která byla nesprávně napsána uživatelem (neobsahuje nekonečnou smyčku nebo nevolá OSTaskDel(OS_PRIO_SELF) na konci)

Kapitola 5

Ověření funkčnosti a ohodnocení vlastností portu

5.1 Testování portu

Autor μC/OS II J. Labrosse doporučuje následující kroky při testování portu:

1. Ověření že se kód úspěšně skopmpiloval.
2. Ověření funkcí OSTaskStkInit() a OSStartHighRdy().
3. Ověření funkce OSCtxSw().
4. Ověření funkcí OSIntCtxSw() a OSTickISR().

5.1.1 OSTaskStkInit() a OSStartHighRdy()

V daném testu nepotřebujeme nastavení přerušení od systémového časovače a žádnou jinou úlohu než Idle. Funkce main vypadá následovně:

```
void main(void)
{
   OSInit();
   OSStart();

}
```

Nastavíme breakpoint na řádek 1439 v souboru os_core.c kde je popsáno vytváření Idle úlohy:

```
(void)OSTaskCreateExt(OS_TaskIdle,
          (void *)0,
          &OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE - 1u],
          OS_TASK_IDLE_PRIO,
          OS_TASK_IDLE_ID,
          &OSTaskIdleStk[0],
          OS_TASK_IDLE_STK_SIZE,
          (void *)0,
          OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
```

Address	0 - 3	4 - 7	8 - B	C - F
200009D0	00000000	00000000	00000000	00000000
200009E0	00000000	00000000	04040404	05050505
200009F0	06060606	07070707	08080808	09090909
20000A00	10101010	11111111	00000000	01010101
20000A10	02020202	03030303	12121212	B11B0000
20000A20	B5100000	00000001	00000000	00000000

Obrázek 5.1: Zásobník Idle úlohy po inicializaci.

Pomocí debuggeru zjistíme adresu vrcholu zásobníku Idle úlohy a pomocí nástroje Memory Monitor v prostředí KDS pozorujeme jak se změní obsah zásobníku po vykonání funkce OSTaskCreateExt(). Výsledek je znázorněn na obrázku 5.1. Slova paměti která byla změněna jsou vyznačena červenou barvou a modrým trojúhelníkem v levém spodním rohu, obsah paměti je zapsán ve formátu Little-Endian.

Popíšeme jednotlivé položky zásobníku ode dna na adrese 0x20000A24:

- Počáteční obsah registru xPSR. Nastaven flag T, procesor je v Thumb stavu.
- Adresa funkce úlohy. Počáteční hodnota programového čítače.
- Adresa funkce OS_TaskReturn(). Počáteční hodnota návratového registru.
- Čtyři položky pro registry R12, R3, R2, R1. Mohou být nastaveny libovolně.
- Registr R0. Obsahuje hodnotu parametru při vytváření, pro Idle úlohu obsahuje nulu.
- Osm položek pro registry R11 až R4. Mohou být nastaveny libovolně.

Tím jsme prokázali správnost činnosti funkce OSTaskStkInit(). Pro ověření funkce OSStartHighRdy() stačí pokračovat ve vykonávání kódu a přesvědčit se, že jsme se dostali do nekonečné smyčky Idle úlohy. Což prokazuje korektnost činnosti OSStartHighRdy().

5.1.2 OSCtxSw()

V tomto testu přidáme jednu úlohu:

```
void main(void)
{
   OSInit();
   OSTaskCreate(TestTask, (void *)0, &TestTaskStk[99], 0);
   OSStart();

}
void TestTask (void *pdata)
{
   while(1)
      OSTimeDly();
}
```

Důkazem korektnosti OSCtxSw() může být ten fakt, že se po uspání úlohy TestTask provede přepnutí kontextu na Idle úlohu.

Tabulka 5.1: Příkazy pro Task Manager

Přijatý symbol	Akce
h	Výpis této tabulky do terminálu
i	Výpis informace o systému
g	Zapnout/vypnout Ganttův diagram
1	Spustit sadu úloh Semaphore
2	Spustit sadu úloh Mutex
3	Spustit sadu úloh Mutex Deadlock
4	Spustit sadu úloh Bomb
5	Spustit sadu úloh Memory

5.1.3 OSIntCtxSw() a OSTickISR()

Funkce OSIntCtxSw() v našem portu vypadá stejně jako OSCtxSw(). Pro ověření OSTickISR() v úloze TestTask přidáme inicializaci přerušení od systémového časovače a kód pro blikání LED diodou na desce FITkit 3:

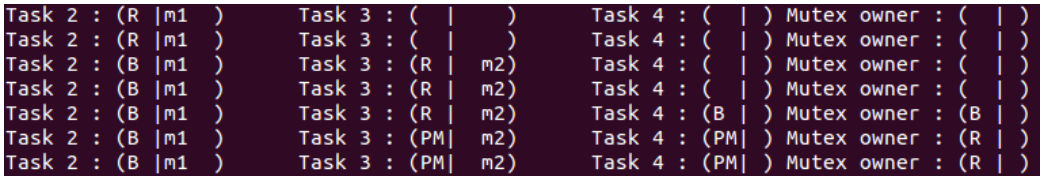
```
void TestTask (void *pdata)
{
  int freq=10; /* 10 Hz SysTick frequency, T = 100ms*/
  OS_CPU_SysTickInit(50000000u/freq);
  while(1)
  {
     OSTimeDly(10); /* 1s*/
     F3Led3Toogle();
  }
}
```

Při spouštění této úlohy můžeme pozorovat blikání diody D12 s periodou 1s, což prokazuje korektnost funkce OSTickISR().

5.2 Demo aplikace

Demo aplikace představuje sadu příkladů na prokázání funkčnosti různých prostředků operačního systému uC/OS II. Při spouštění systému běží pouze jedna úloha Task Manager, která slouží pro interakci s uživatelem a spouštění zvolené sady úloh. Interakce probíhá pomocí sériové linky mezi virtuálním COM-portem počítače a UART modulem mikrokontroléru přes rozhraní USB. Kód symbolu se předává do úlohy Task Manager funkcí obsluhy přerušení UART modulu pomocí prostředku operačního systému Mailbox. Akce Task Manageru v závislosti na přijatém symbolu jsou uvedeny v tabulce 5.1.

Při zapnutém režimu Ganttova diagramu se do terminálu v pravidelných intervalech vypisuje současný stav úloh na 4 používaných prioritách: Task 2 má prioritu 15, Task 3 prioritu 13, Task 4 prioritu 12. Sloupec Mutex Owner odpovídá prioritě mutexu a vypisuje stav úlohy běžící na prioritě 10 při spouštění mechanismu dědění priorit Priority Celling Priority. Příklad výpisu Ganttova diagramu je na obrázku 5.2. Na tomto diagramu je informace o úlohách v kulatých závorkách je rozdělena do dvou částí. První část zobrazuje stav úlohy, R - běžící úloha, B - úloha je připravena k běhu ale neběží, PM - čeká na uvolnění mutexu, mezera - úloha je uspána nebo neexistuje. Druhá část za svislou čárou



Obrázek 5.2: Ganttův diagram v demonstračním příkladu

udává informace o mutexech, které vlastní daná úloha. Výpis stavu úloh je implementován v souboru app_hooks.c funkcí App_TimeTickHook(), která je volána při každém přerušení systémového časovače.
Kostra hlavní smyčky úlohy Task Manager vypadá následovně:

```
while(1)
{
    msg = OSMboxPend(Mbox, 0, &err); // přijmout symbol, zadány uživatelem
    if(*(char*)msg == 'h')
    {
           ... // výpis nápovědy
    }else if(*(char*)msg == 'i')
    {
           ... // výpis systémových informací na terminál
    }else if(*(char*)msg == '1')
    {
           ... // vytvoření úloh pro 1. příklad
    }else
           ...
}
```

5.2.1 Sada úloh Semaphore

Tento příklad demonstruje činnost a použití semaforů v uC/OS II pro signalizování událostí mezi úlohami. Sada se skládá ze tří úloh a používá tři semafory. Na začátku všechny semafory jsou inicializovány hodnotou 0 a každá z úloh je uspána kvůli čekání na svém semaforu. Zmačknutím tlačítka SW6 na desce funkce obsluhy přerušení signalizuje úloze Task 2 že může pokračovat v běhu, v polovině své práce Task 2 pošle signál pro úlohu Task 3 a jelikož Task 3 má vyšší prioritu, úloha Task 2 bude pozastavena, v polovině práce Task 3 pošle signál úloze Task 4. Po dokončení úlohy Task 4 se procesor vrací postupně k úlohám Task 3 a Task 2. Vždy po dokončení práce každá úloha smaže sama sebe voláním funkce OSTaskDel().
Například tělo úlohy 3 se skládá z následujících příkazu:

```
OSSemPend(sem_2_3, 0u, &err); // čekání na semafor od předchozí úlohy
dummy_work(3);
OSSemPost(sem_3_4); // signalizace další úlohy
dummy_work(3);
OSTaskDel(OS_PRIO_SELF); // mazání sama sebe
```

Tabulka 5.2: Příkazy pro příklad Memory

Přijatý symbol	Akce
ESC	Ukončit příklad
Enter	Výpis obsahu fronty
Backspace	Smazat poslední element fronty
Space	Smazat první element fronty
Jiný symbol	Zařaď symbol do fronty podle ASCII kódu

5.2.2 Sada úloh Mutex

Příklad demonstruje použití prostředku mutex pro vzájemné vyloučení a činnost protokolu Priority Celling Priority pro zamezení problému inverze priorit při použití mutexů. Při spouštění příkladu jsou nastartovány tři úlohy. Task 2 začíná svůj běh jako první a vstupuje do kritické sekce chráněnou mutexem pak s časovým rozestupem začínají běžet úlohy Task 3 a Task 4. Po krátké době Task 4 chce vstoupit kritické sekce chráněnou mutexem, který vlastní Task 2, a jelikož Task 2 má nejnižší prioritu je zablokována, a tím pádem úloha Task 4 musí počkat až doběhne Task 3 a Task 2 uvolní mutex. Ale díky protokolu PCP inverze priorit nenastává, protože se priorita úlohy Task 2 zvyšuje na prioritu mutexu na dobu běhu v kritické sekci.

5.2.3 Sada úloh Mutex Deadlock

Protokol PCP však nezamezuje vzniku deadlocku při snáze dvou úloh vstoupit do dvou kritických sekcí v opačném pořadí, což ilustruje příklad Mutex Deadlock. Úlohy Task 2 a Task 3 se snaží vstoupit do kritických sekcí v opačném pořadí a jsou zablokovány, dokud nevyprší timeout čekání na mutex a úlohy pokračují v běhu a smažou se pro dokončení příkladu.

5.2.4 Sada úloh Bomb

Sada demonstruje synchronizaci úloh pomocí semaforů a použití periferií na desce FITkit 3, skládá se ze tří úloh:

- Task_Music - přehrává hudbu, zatímco uživatel může pomocí tlačítek SW2-SW5 na desce nastavit hodnotu čítače. Po zmačknutí tlačítka SW6, signalizuje start úlohy Counter
- Task_Counter - S každou vteřinou dekrementuje hodnotu čítače. Pokud hodnota čítače dojde do 0, signalizuje start úlohy Bomb
- Task_Bomb - imituje výbuch blikáním LED diod a zvukem nízké frekvence

5.2.5 Sada úloh Memory

Příklad ilustruje použití dynamické alokace paměti prostředky operačního systému. Pro demonstraci je implementována datová struktura prioritní fronta pomocí obousměrně vázaného seznamu. Interakce s uživatelem probíhá pomocí příkazů z tabulky 5.2. Velikost dynamické paměti je omezená na 16 prvků.

5.3 Benchmarking portu

5.3.1 Přímé testy

Přímé testy jsou zaměřeny na ohodnocení doby vykonávání nejdůležitějších událostí systému μC/OS II. Jednotkou měření v těchto testech je počet cyklů, které vykonal procesor. Modul DWT [3] (Data Watchpoint and Trace unit) procesoru ARM Cortex-M4 obsahuje 32-bitový čítač cyklů CYCCNT, ke kterému můžeme přistupovat pomocí struktury DWT, popsané v souboru *core_cm4.h*. Základní kostra přímých testů vypadá následovně:

```
DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk; // povolení čítače
uint32_t cycles = 0; // proměnná pro uchování výsledku
// Začátek testu, provede se EXPER_NUMBER-krát
for(i = 0; i < EXPER_NUMBER; i++)
{
    // Zákaz přerušení od systémového časovače, aby měření nebylo zkresleno
    SysTick->CTRL &= ~SysTick_CTRL_ENABLE_Msk;
    // Vynulování čítače cyklů
    DWT->CYCCNT = 0x00;
    // Tělo testu,
    // část kódu, dobu vykonávání kterého chceme změřit
    ...
    some_function();
    ...
    // Proměnná cycles strádá počet cyklů
    cycles += DWT->CYCCNT;
    // povolení přerušení od systémového časovače
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
}
// Výpis výsledků do terminálu
printf("Test 1 : %d\n", cycles/EXPER_NUMBER);
```

V některých testech v tělu bylo třeba zahájit přerušení. Pro tento účel byla použitá přerušení GPIO portů pomocí konstrukce, uvedené níže:

```
// Nastavení čekání obsluhy přerušení pro port E
NVIC_SetPendingIRQ(PORTE_IRQn);
__DSB(); // Ensure transfer is completed
__ISB(); // Ensure side effect of the write is visible
```

Vzhledem k tomu, že zahájení přerušení trvá nějakou dobu, příkazy ___DSB(), ___ISB() vytvoříme paměťovou bariéru a zajistíme, že instrukce po funkce nastavení pending bitu v NVIC nebudou vykonány, dokud se neprovede návrat z funkce obsluhy přerušení.

V rámci přímých testů byla provedena následující měření:

- Doba startu systému (funkce OSInit(), OSStart(), vytvoření první úlohy, přepnutí kontextu na tuto úlohu).
- Doba zahájení (včetně inkrementace proměnné OSIntNesting) a vykonání prázdné obsluhy přerušení (zahájení a návrat. Obsahuje pouze nezbytné pro systém μC/OS II konstrukce, viz Kapitola 3.2.5).

Tabulka 5.3: Výsledky přímých testů.

	Počet cyklů	Čas, μs
Start systému	74739	1494,78
Zahájení přerušení	83	1,66
Vykonání prázdné ISR	291	5,82
Zprává pomocí Message box	821	16,42
Signalizace semaforem	793	15,86
Režie obsluhy systémové časovače	487	9,74
Vytváření úlohy	937	18,74
Kritická sekce	20	0,4
Kritická sekce pomocí mutex	297	5,94

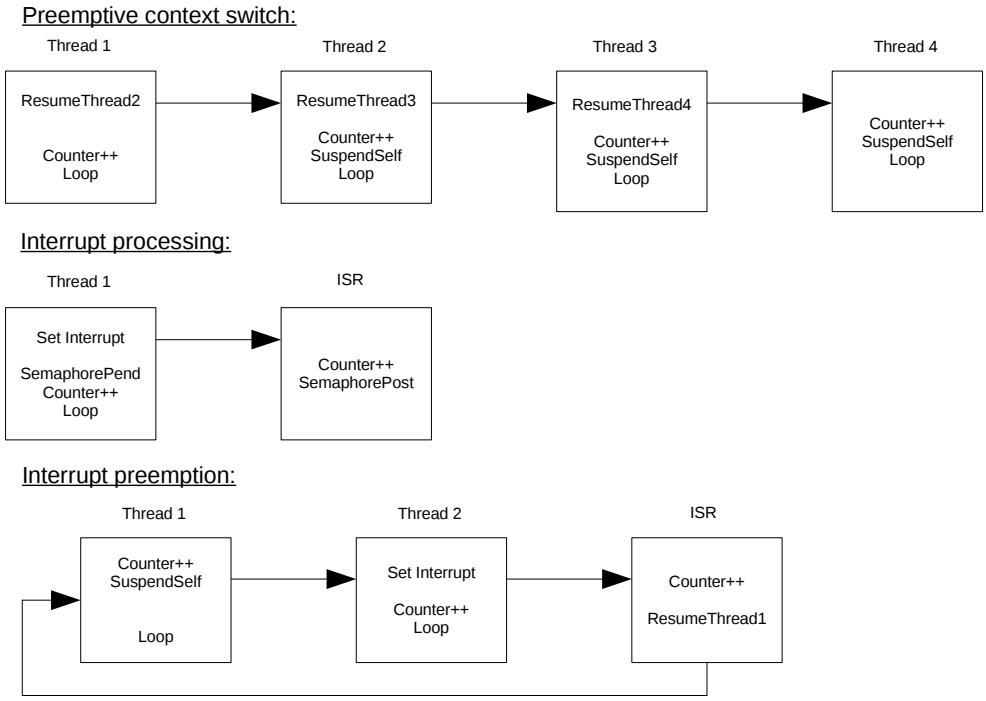
- Doba mezi zahájením přerušení pro zaslání úloze zprávy pomocí message boxu a časem přijetí této zprávy úlohou. Typickým použitím dané konstrukce je informování úlohy o nějaké vnější události, např. zmáčknutí tlačítka.
- Doba mezi zahájením přerušení pro zaslání úloze signálu pomocí semaforu a časem přijetí této zprávy úlohou.
- Doba provádění obsluhy přerušení od systémového časovače.
- Doba vytváření nové úlohy pomocí funkce OSTaskCreate().
- Doba vykonání prázdné kritické sekce (režie maker OS_ENTER_CRITICAL() a OS_EXIT_CRITICAL()).
- Režie vstupu a výstupu do kritické sekce, chráněnou mutexem.

Výsledky přímých testů jsou uvedeny v tabulce 5.3. Ve třetím sloupci je uveden čas v μs pro frekvenci systémových hodin 50 MHz.

5.3.2 Sada testů Thread-Metric

Thread-Metric ja sada testů operačních systémů reálného času navržená firmou Express Logic [2]. Testy jsou napsány v jazyce C. Pro použití s deskou FITkit 3 bylo potřeba upravit soubor *tm_porting_layer_ucosii.c*, ve kterém je popsáno mapování mezi funkcemi operačního systému μC/OS II a rozhraním pro jednotlivé testy. Průběh všech testů spočívá v počítání iterací, které vykoná systém za dobu 30 vteřin. Popíšeme jednotlivé testy, které byly provedeny v rámci tohoto projektu:

- **Basic test.** Ve smyčce provádí výpočty na poli velikosti 1024 položek. Používá se pro zjištění poměru výkonu na různých platformách. Pro porovnání dvou platforem, výsledky každého z dalších testu se dělí hodnotou z tohoto testu.
- **Preemtive context switching** Obsahuje 5 úloh na různých prioritách. Na začátku každá úloha probouzí úlohu na vyšší prioritě, to znamená, že může pokračovat svůj běh až se uspí všechny úlohy na vyšších prioritách, pak inkrementuje čítač a uspí se. Jedná iterace skončí, když se uspí úloha na nejnižší prioritě, potom začne novou iteraci probouzením úlohy s vyšší prioritou. Výsledkem testu je suma hodnot čítačů všech úloh.



Obrázek 5.3: Ilustrace k testům Thread-Metric.

- **Interrupt processing** Obsahuje úlohu, která na začátku iterace zahájí přerušení, ve funkci obsluhy přerušení se dekrementuje hodnota semaforu. Až se řízení vrátí, v úloze proběhne kontrola synchronizace semaforem a inkrementace čítačů události, a začne se nová iterace.
- **Interrupt preemtion** Obsahuje dvě úlohy na různých prioritách. Úloha s vyšší prioritou inkrementuje čítač a uspí se, kontext se přepne na druhou úlohu, která zahájí přerušení. Ve funkci obsluhy přerušení se provede probouzení první úlohy. Až se první úloha znovu uspí, doběhne druhá a tím iterace skončí.
- **Semaphore processing** Obsahuje jednu úlohu, která v každé iteraci postupně vezme a vrátí semafor.
- **Message processing** Obsahuje jednu úlohu, která v každé iteraci pošle a přijme zprávu přes message box.

Výsledky testů jsou v tabulce 5.4. Ilustraci k některým testům můžete najít na obrázku 5.3.

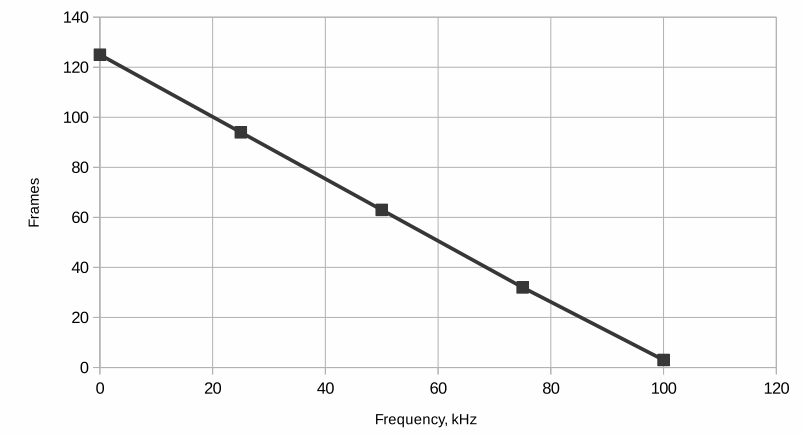
5.3.3 Vliv frekvencí přerušení a systémového časovače na výkon

Systémový časovač

Volba frekvence systémového časovače je důležitým rozhodnutím při návrhu real-time systému. Zvýšením frekvence můžeme zvýšit přesnost časově závislých funkcí μC/OS II (například funkce OSTimeDly(), timeouty při práci s semafory a mutexy atd.). Negativní stránkou takovéhoto zvýšení bude velká režie operačního systému, což přivede k poklesu

Tabulka 5.4: Výsledky Thread-Metric testů.

	Počet iterací
Basic test	39518
Preemtive context switching	2098250
Interrupt processing	2714800
Interrupt preemption	1374751
Semaphore processing	5429538
Message processing	2653510



Obrázek 5.4: Závislost výkonu na frekvenci systémového časovače

výkonu. V této podkapitole ukážeme závislost výkonu na volbě frekvence systémového časovače a zjistíme maximální frekvenci, na které port μC/OS II na FITkit 3 bude ještě schopen vykonávat užitečnou práci. Jako zátěž použijeme výpočet konvolučního filtru 3×3 pro zvýraznění hran, aplikovaný na obrázky velikosti 100×100 pixelů. Měřítkem výkonu aplikace bude počet snímku, zpracovaných za 10 vteřin.

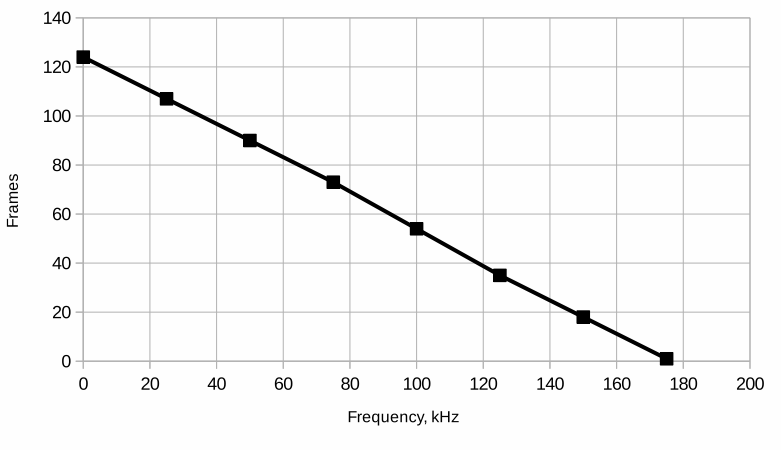
Měření bylo realizováno pomocí dvou úloh. Úloha $Task\_Worker$ má nižší prioritu, počítá konvoluci a inkrementuje globální proměnnou po každém zpracovaném snímku. Úloha $Task\_Start$ má vyšší prioritu. Na začátku vynuluje čítač snímku a uspí se na 10 vteřin, po vypršení tohoto času vypíše hodnotu čítače do terminálu. Pro každý krok testu je třeba ručně nastavit frekvenci systémového časovače do proměnné $freq$, zkompilovat projekt a nahrát ho do MCU. Graf s výsledky experimentu je zobrazen na obrázku 5.4.

V průběhu experimentu se zjistilo že při frekvenci 100 kHz úloha zpracovává 3 snímky za 10 vteřin, při frekvenci 105 kHz úloha pro výpis výsledku nikdy nedostane procesorový čas. Tato hodnota odpovídá době vykonávání obsluhy přerušení systémového časovače 9,74μs z přímých testů v tabulce 5.3:

$$f_{max} = \frac{1}{9,74 \cdot 10^{-6} s} = 102,7 \cdot 10^{3} Hz$$

Přerušení

V tomto experimentu byla zjištěna závislost výkonu systému na hodnotě frekvenci vnějšího přerušení. Jako zdroj přerušení byl použit laboratorní generátor signálů připojeny na pin



Obrázek 5.5: Závislost výkonu na frekvenci vnějších přerušení

PTA8 mikrokontroléru. Tvar signálu byl nastaven na obdélníkový, amplituda 4 V, frekvence se postupně měnila od 0 do 200 kHz.

Graf s výsledky experimentu je zobrazen na obrázku 5.5. Stejně jako v případě měření maximální frekvence systémového časovače byly použité úlohy $Task\_Worker$ a $Task\_Start$.

Při frekvenci 175 kHz systém zpracovává 2 snímky za 10 vteřin, při frekvenci 180 kHz přestává vykonávat užitečnou práci. Podle výsledku přímých testů doba vykonávání prázdné funkce obsluhy přerušení systému μC/OS II trvá 5,82μs. Spočítáme maximální frekvenci z těchto údajů:

$$f_{max} = \frac{1}{5,82 \cdot 10^{-6} s} = 171,8 \cdot 10^{3} Hz$$

Kapitola 6

Příklady aplikací pro systém μC/OS II

6.1 Zvýšení spolehlivosti systému

V této podkapitole se seznámíme s metodou Enhanced Control-Flow Checking Using Assertions pro zvýšení odolnosti proti poruchám a chybám v systému.

6.1.1 Enhanced Control-Flow Checking Using Assertions

Tato technika je založena na tom, že program se dělí na bloky neskokových instrukcí. Můžeme pohlížet na takový program jako na orientovaný graf $G = (V, E)$, kde V je množina všech bloků, E množina všech povolených přechodu mezi bloky. Každému bloku B se přiřazuje identifikátor b_{IDi}, který muže být libovolným prvočíslem kromě 2 a hodnota b_{NEXTi}, která se spočítá jako součin identifikátorů povolených následovníků daného bloku. Na začátku každého bloku se provádí operace SET, která zjišťuje zda program skočil do daného bloku z povoleného předchůdce, a nastavuje novou hodnotu proměnné G_{ID}. G_{ID} je globální proměnná, kterou potřebujeme deklarovat pro každý graf G. Tato operace se provádí pomocí následujícího vztahu:

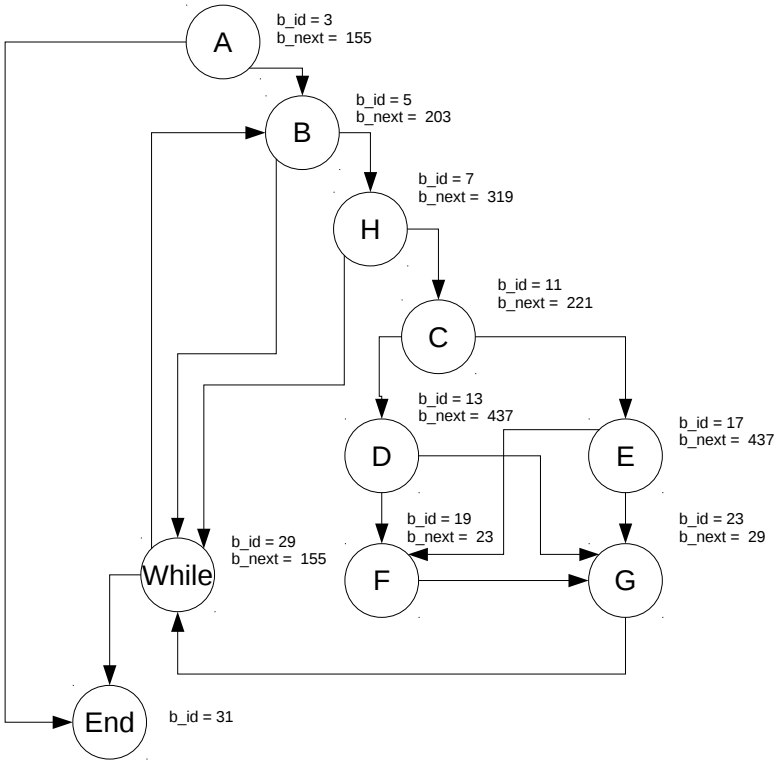
$$G_{ID} = \frac{b_{IDi}}{(\overline{G_{ID} \; mod \; b_{IDi}})(G_{ID} \; mod \; 2)}$$

Na konci každého bloku je vykonána operace TEST podle následujícího vztahu:

$$G_{ID} = b_{NEXTi} + diff(G_{ID} - b_{IDi})$$

Funkce diff() vrací 0, pokud argumenty mají stejnou hodnotu, jinak vrací 1.

Chyba je detekována během operace SET, která při nesprávném toku programu přivede na dělení nulou. Výraz $\overline{G_{ID} \; mod \; b_{IDi}}$ se rovna 0 pokud blok B_i není platným následníkem předcházejícího bloku. $G_{ID} \; mod \; 2$ se rovna 0, pokud G_{ID} je sudé číslo. Toto se může nastat, pokud v předcházející operaci TEST funkce diff() vrátila 1, což ukazuje na to, že tato operace TEST byla provedena v jiném bloku, než předcházející operace SET.



Obrázek 6.1: Graf programu funkce OSTimeTick().

Pro implementaci dané techniky na platformě FITkit 3 bylo třeba povolit generování výjimky při dělení nulou pomocí příkazu:

```
SCB->CCR |= SCB_CCR_DIV_0_TRP_Msk; // enable division by zero exception
```

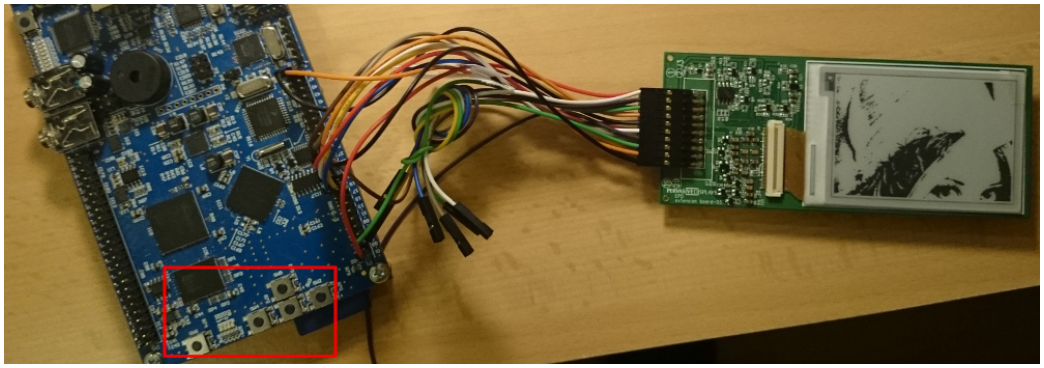
Po tomto povolení při výskytu chyby dělení nulou řízení bude předáno funkci HardFault_Handler(), kde můžeme popsat kroky pro zotavení nebo pro bezpečné zastavení systému.

V souborech controlflow.h a controlflow.c jsou popsány funkce ECCA_test, ECCA_set a množina prvočísel *primes*.

Vyzkoušíme použití této techniky na příkladě funkce OSTimeTick(), funkce obsluhy přerušení od systémového časovače v μC/OS II. Graf programu této funkce je na obrázku 6.1. Pomocí metody z kapitoly 5.3.1 byla změřena režie funkce obsluhy systémového časovače se změnami s této podkapitoly. Režie tvoří 1299 cyklů procesoru. Celkové zpomalení metodou ECCA: $\frac{1299}{487} = 2.67$ krát.

6.2 Aplikace řízení E-ink displeje

Jako příklad řídicí aplikace s využitím μC/OS II byl implementován systém pro zobrazení obrázku, načtených s SD-karty, na E-ink displeji a přehrávání melodie pomocí piezoměniče LS1 na desce FITkit 3.



Obrázek 6.2: Celkový pohled na implementovaný systém. Tlačítka pro ovládání jsou vyznačena červeně.

Tabulka 6.1: Propojení konektoru displeje s deskou FITkit 3

Číslo pinu konektoru displeje	Signál	Číslo pinu konektoru P1 na desce FITkit 3
1	VCC 5V	2
6	TEMPERATURE	5
7	SPI CLK	19
8	BUSY	23
9	PWM	28
10	RESET	25
11	PANEL_ON	26
12	DISCHARGE	24
13	BORDER_CONTROL	30
14	SPI MISO	22
15	SPI MOSI	20
18	FLASH_CS	49
19	EPD_CS	27
20	GND	50

6.2.1 Popis hardwarové části

Táto aplikace se skládá ze tří základních hardwarových částí: deska FITkit3 Minerva, E-ink displej EG020AS012-ND a SD-karta pro ukládání obrázků.
Popis desky FITkit byl uveden v kapitole 3.1.
EG020AS012-ND [4] je 2.7 palcový E-ink displej (nebo také Electronic paper display, dále EPD). Pro zjednodušení připojení a ovládání displeje byla použita pomocná deska EPD Extension Kit od výrobců [7]. Deska obsahuje kondenzátory pro zajištění napájení, potřebné pro funkčnost displeje, čidlo teploty s analogovým výstupem a 20-pinový konektor. Propojení tohoto konektoru s konektorem P1 na desce FITkit 3 bylo implementováno podle tabulky 6.1.
Pro připojení SD-karty deska FITkit 3 již obsahuje konektor J8, který je spojen s odpovídajícími výstupy modulu SDHC mikrokontroléru.
Fotografii systému můžete najít na obrázku 6.2.

6.2.2 Popis softwarové části

Ovladač displeje

Displej obsahuje vestavěný kontrolér (Chip On Glass, dále COG), který umožňuje komunikaci pomocí SPI. Ovládání displeje se provádí pomocí rozhraní SPI, signálu PWM s frekvencí 100-300 kHz, několika číslicových výstupu kontroléru (GPIO) a ADC převodníku pro zjištění aktuální teploty. Inicializace těchto komunikačních prostředků kontroléru Kinetis K60 je popsáno ve funkci F3DisplayInit() v souboru F3.c. Proces komunikace s kontrolérem COG můžeme rozdělit na následující kroky (každý z těchto kroku je implementován jako C funkce v souboru COG_driver.c ve složce Drivers archivu projektu):

1. Zapnutí napájení COG, popsáno ve funkci `COG_power_on()`.
2. Inicializace driveru COG, funkce `COG_init_driver()`.
3. Zápis dat, popsáno ve funkci `COG_write_image_from_array(uint8_t *image, uint8_t stage)`.
4. Vypnutí napájení COG, popsáno ve funkci `COG_power_off()`.

Podíváme se podrobněji na zápis dat v 3. kroku. Pro každý pixel můžeme nastavit jednu z tří hodnot: bílý, černý a nothing pixel. Kvůli fyzickým vlastnostem E-ink displejů pro odstranění viditelných stop po předchozích obrázcích zobrazení se provádí ve 4 fázích a trvání každé fáze závisí na okolní teplotě. Pro hodnotu okolní teploty 20 stupňů zobrazení trvá kolem 5 vteřin. Jednotlivé fáze jsou:

1. Negativ původního obrázku.
2. Prázdný obrázek (celý displej je bílý).
3. Negativ nového obrázku.
4. Nový obrázek v normálních barvách.

Podrobný popis komunikace, seznam příkazů přes SPI a časování všech potřebných signálu můžete najít v dokumentu od Pervasive Displays [6].
Z této metody zobrazení plyne, že v paměti kontroléru potřebujeme uchovávat původní i nový obrázek. V aplikaci to může být zohledněno použitím dvou bufferů pro obrázky a ukazatelů na ně, které se prohazují s každým novým zobrazením.
Celý proces zobrazení je zapouzdřen do funkce:
`show_image(uint8_t *old_image, uint8_t *new_image)` ze souboru COG_driver.c. Parametry této funkce jsou ukazatelé na obrázek, který je současně zobrazen na displeji a nový obrázek. Obrázky musí mít format bitmapy jako 1D pole pixelů od horního levého rohu obrázku, kde bit v 1 znamená bílý pixel, 0 - černý pixel.

Ovladač SDHC modulu

Každá SD-karta kromě samotné paměti obsahuje také vestavěný kontrolér pro řízení paměti. Význam kontaktů karty je popsán v tabulce 6.2.

Komunikace s kontrolérem SD-karty může probíhat buď pomocí protokolu SD nebo pomocí SPI. Na desce FITkit 3 komunikace přes SPI není možná kvůli zapojení pinů modulu

Tabulka 6.2: Význam kontaktů SD-karty v režimech SD a SPI

Číslo pinu	Název	Funkce SD	Funkce SPI
1	DAT3/CS	Data Line 3	Slave select SS
2	CMD/DI	Command Line	MOSI
3	VSS1	Ground	Ground
4	VDD	Supply Voltage	Supply Voltage
5	CLK	Clock	SCK
6	VSS2	Ground	Ground
7	DAT0/D0	Data Line 0	MISO
8	DAT1/IRQ	Data Line 1	Unused or IRQ
9	DAT2/NC	Data Line 2	Unused

SPI2: výstup MOSI je propojen s výstupem SD-karty DO, vstup MISO se vstupem DI. Komunikace na SD sběrnici je založena na řídicím a datovém toku. Řídicí tok používá kontakt CMD a skládá se z příkazu (Command) od mikrokontroléru ke kartě a odpovědi (Response) SD-karty. Datový tok probíhá po kontaktech DAT0-DAT3 v obou směrech. Kontrolér SD-karty se může nacházet ve dvou režimech: režim identifikace (Card Identification Mode), kdy nastavujeme parametry SDHC modulu a SD-karty a režim přenosu (Data Transfer Mode). Ovladač je popsán v souboru SHDC_driver.c ve složce Drivers. Fázi identifikace zajišťuje funkce `sd_card_init()`, fázi přenosu dat funkce `sd_read_image(uint8_t *buf, uint32_t image_number)`, parametry této funkce je ukazatel na buffer pro ukládání obrázku a číslo obrázku pro načtení s SD-karty.

Podrobný popis komunikačního protokolu, seznam příkazů a popis stavů kontroléru SD-karty můžete najít v dokumentaci od SanDisk [12].

Ovladač nepředpokládá, že SD-karta je formátována a obsahuje nějaký souborový systém. Obrázky jsou nahrávány za sebou do 512-bajtových sektorů, začínaje od 0. sektoru. Jeden obrázek zabírá 12 sektorů. Nahrát obrázek na kartu můžeme pomocí aplikace `dd`: `dd if=<obrázek> of=<SD karta>` , kde

- obrázek - binární soubor, vygenerovány aplikací image_converter, kterou popíšeme v jedné s dalších podkapitol. Pro nahrání několika obrázků je možno spojit tyto soubory příkazem cat.
- SD karta - speciální soubor SD karty ve složce /dev (například může být /dev/mmblck0).

Úlohy μC/OS II

Funkčnost aplikace je zajištěna třemi úlohami μC/OS II. Úlohy v klesajícím pořadí jejich priorit jsou:

- `Task_SW6_handler.` Spouští se po zmačknutí tlačítka SW6. Zjišťuje jak dlouho bylo zmáčknuto tlačítko, po rychlém zmáčknutí zapíná nebo vypíná přehrávání melodie. Po zmáčknutí na dobu víc než 1 vteřina přepíná frekvenci hodin SPI modulu pro komunikaci s displejem. Snížení frekvence komunikačního protokolu může být použito, pokud kvalita propojení nedovoluje použití frekvence 12 MHz. Toto přepnutí je chráněno mutexem SPI_mutex, aby nedošlo k přepnutí během přenosu dat.
- `Task_Music.` Načítá z pole Melody tón a délku noty. Nastavuje odpovídající frekvenci do PWM modulu, který řídí piezoměnič, a uspává se na dobu délky noty.

Tabulka 6.3: Popis ovládání aplikace

Tlačítko	Funkce
SW4	Zobrazit předchozí obrázek
SW2	Zobrazit následující obrázek
SW5	Obnovit aktuální obrázek
SW3	Zobrazit první obrázek
SW6	Zapnout/Vypnout přehrávání melodie
SW6 na dobu delší 1s	Přepnout frekvenci SPI modulu

- `Task_Start`. Na začátku vytváří dvě další úlohy, pak vstupuje do smyčky ve které se nejdřív uspí, probouzí se po zmáčknutí tlačítka SW2, SW3, SW4 nebo SW5. Po probouzení načítá obrázek s SD-karty a zobrazuje jej na displeji. Přenos dat je chráněn mutexem SPI_mutex.

6.2.3 Ovládání

Systém se ovládá pomocí tlačítek na desce FITkit 3, vyznačená červeným čtvercem na obrázku 6.2. Popis jejich funkcí je uveden v tabulce 6.3.

6.2.4 Pomocné aplikace pro PC

Vytvoření bitmapy z obrázku v běžném formátu

Pro generování bitmapy z obrázků v běžných formátech (všechny formáty, které podporuje knihovna OpenCV [1]. Například Windows bitmaps, JPEG, TIFF a jiné) byla napsána aplikace pro PC, zdrojové kódy které můžete najít ve složce image_converter. Aplikace používá knihovnu OpenCV a spouští se příkazem:
`./image_converter <název obrázku> <práh> <x> <y>` , kde

- název obrázku - cesta k obrázku pro zobrazení.
- práh - hodnota od 0 do 255 pro prahování podle jasu (pro rozhodnutí který pixel bude považován za bílý a který za černý).
- x a y - souřadnice levého horního rohu obrázku pro zobrazení v původním obrázku.

Oblasti, které přesahují hranice původního obrázku, budou vygenerovány jako bílé.
Výstupem této aplikace jsou tři soubory:

- result.png - přehled jak bude vypadat obrázek na displeji.
- <název obrázku>.h - obsahuje bitmapu jako pole jazyka C, které můžeme přidat do zdrojových souborů.
- <název obrázku>.bin - bitmapa v binárním formátu, kterou můžeme předat na kontrolér pomocí libovolného komunikačního rozhraní.

Vytvoření melodie

Aplikace Music slouží pro vytvoření pole jazyka C, které může být přehráno úlohou Task_Music ze souboru v textovém formátu, který je čitelný pro člověka. Příklad vstupného souboru:

```
H2 e F#3 e G3 e F#3 e H2 e F#3 e D3 e F#3 e
A2 e F#3 e G3 e F#3 e A2 e F#3 e D3 e F#3 e
```

Kde každá nota je popsána jejím tónem a délkou. Například `F#3 e`, popisuje osminovou notu F# třetí oktávy. Pro přehlednost jeden takt může odpovídat jednomu řádku. Možné délky not jsou: w - celá (whole), h - půlová (half), q - čtvrťová (quarter), e - osminová (eighth), s - šestnáctinová (sixteenth).
Aplikace se spouští příkazem:

```
./music <input_file> <temp>
```

Parametr temp zadává tempo melodie v počtu čtvrťových not za minutu. Výstupem aplikace je soubor out, který obsahuje odpovídající pole Melody jazyka C.

Kapitola 7

Závěr

Tato práce byla zaměřena na implementaci systému reálného času μC/OS II na platformě FITkit 3 s procesorem ARM Cortex-M4. Během vypracování projektu byl vytvořen port jádra systému v podobě Workspace se všemi potřebnými zdrojovými kódy a nastaveními pro vývojové prostředí Kinetis Design Studio. Nad tímto jádrem bylo napsáno několik jednoduchých aplikací na prokázání činnosti systémových volání μC/OS II. Dále bylo provedeno kvantitativní ohodnocení portu systému (benchmarking). Bylo implementováno několik metod pro zvýšení spolehlivosti systému v jádru μC/OS II. Práce byla dovršena implementací příkladu aplikace nad jádrem μC/OS II pro zobrazení obrázků načtených s SD-karty na E-ink displeji.

Jako pokračování této práce by mohlo být vyzkoušení připojení k platformě dalších periferních zařízení (např. LCD displeje, servomotory atd.) nebo přidání do jádra systému dalších mechanismu zvýšení spolehlivosti.

Literatura

[1] OpenCV 2.4.13.0 documentation.
URL http://docs.opencv.org/2.4/modules/highgui/doc/reading_and_writing_images_and_video.html#imread

[2] Thread-Metric Benchmark Suite.
URL http://rtos.com/PDFs/MeasuringRTOSPerformance.pdf

[3] ARM v7-M Architecture Reference Manual. 2010.
URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403e.b/index.html

[4] Product Specifications 2.7" TFT EPD Panel , 1P008-00. 2011.
URL http://www.seeedstudio.com/document/pdf/1P008-00_01_EM027AS011.pdf

[5] K60 Sub-Family Reference Manual, K60P144M100SF2V2RM. 2012.
URL
http://cache.nxp.com/files/32bit/doc/ref_manual/K60P144M100SF2V2RM.pdf

[6] E-paper Display COG Driver Interface Timing, 4P008-00. 2013.
URL http://www.pervasivedisplays.com/_literature_138408/E-paper_Display_COG_Driver_Interface_Timing_V110

[7] EPD Extention Kit User Manual. 2015.
URL http://www.pervasivedisplays.com/_literature_222703/EPD_Extension_Kit_User_manual_v03

[8] Kinetis Design Studio User's Guide, KDSUG. 2015.
URL
http://cache.nxp.com/files/microcontrollers/doc/user_guide/KDSUG.pdf

[9] Procedure Call Standard for ARM Architecture, ARM IHI 0042D. 2015.
URL http:
//infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf

[10] Labrosse, J. J.: *uC/OS II User's Manual*. Micrium Press, 2015.

[11] Laplante, P.: *Real-Time Systems Design and Analysis*. IEEE, 2004.

[12] SanDisk: *SanDisk Secure Digital Card, Product Manual, 80-13-00169*. SanDisk Corporation, 2003.

[13] Yiu, J.: *The definitive guide to the ARM Cortex-M3 and Cortex-M4 processors*. Oxford, UK, Waltham, MA: Newnes, 2014.

Přílohy

Seznam příloh

Příloha A

Obsah CD

Na přiloženém CD se nachází následující soubory a adresáře:

```
\Workspace_SP
   < KDS workspace s příkladem z kapitoly 5.2 >
\workspace
   +- control_flow_checking
      < implementace metody ECCA, kapitola 6.1 >
   +- image_viewer
      < příklad aplikace s E-ink displejem, kapitola 6.2 >
   +- interrupt_latency
      < přímé testy, kapitola 5.3.1 >
   +- interrupt_overhead
      < vliv frekvence přerušení, kapitola 5.3.3 >
   +- sys_tick
      < vliv frekvence systémového časovače, kapitola 5.3.3 >
   +- thread_metric
      < sada testů thread metric, kapitola 5.3.2 >
\PC_apps
   +- image_converter
      < aplikace pro převod obrázku na bitmapu >
   +- music
      < aplikace pro vytváření melodie >
\text
   +- src
      < LaTeX zdrojové kódy textu práce >
   xanisi00.pdf  // text práce ve formátu PDF
```