KIV/OS - cvičení č. 1

Martin Úbl

10. srpna 2021

1 Orientační plán cvičení

Cvičení budou probíhat prezenčně v laboratoři UC-326. Studentům bude k dispozici potřebný hardware do skupin po 2-4 lidech (dle dostupnosti a obsazenosti cvičení). Plán cvičení:

1. úvodní cvičení
   - seznámení s hardware, nastavení vývojového prostředí, základní pojmy
   - rozblikání LED na desce
2. struktura projektu, bootloader, nahrávání kernelu z PC
   - základní struktura projektu OS
   - miniUART a orientace v manuálu k BCM2835
   - UART bootloader
3. AUX koprocesor, UART
   - AUX koprocesor, komunikace přes UART
   - driver pro UART (miniUART)
4. přerušení
   - operační módy procesoru ARM1176
   - tabulka vektorů přerušení, IRQ
   - driver pro ARM timer, využití IRQ
5. paměť a procesy
   - alokátor paměti (bitmapový), kernel heap
   - procesy (tasky) a jejich implementace
   - time-slicing plánovač

6. filesystem a systémová volání
    - filesystem manager, filesystem drivery
    - základní drivery pro FS (GPIO, UART)
    - systémová volání (open, read, write, close, ioctl), RTL
7. ovladače pro periferie desky KIV-DPP-01
    - posuvný registr a 7-segmentový displej
    - generátor náhodných čísel (TRNG)
    - I2C a OLED displej
8. stránkování, uživatelský režim tasků
    - stránkování, TLB, správa tabulek stránek
    - data abort, prefetch abort
    - vedlejší: RTL support pro OLED displej
9. real-time
    - GPIO přerušení
    - periodické a aperiodické tasky
    - EDF plánovač
    - semafory, mutexy, podmínkové proměnné
    - základy power managementu (WFI, WFE)
10. eMMC a SD karta
    - převzatý eMMC driver a driver pro SD kartu
    - čtení bloků z karty, vlastní jednoduchý filesystém
11. co zbylo
    - WiFi, USB
    - co se jinam nevešlo

2 Hardware

V rámci cvičení budeme využívat vývojovou desku Raspberry Pi Zero WH (dále jen RPi Zero). Ta je osazena mikrokontrolérem BCM2835 s procesorem ARM1176JZFS, grafickým procesorem a dalšími periferiemi, které probereme později.

Dále bude pro potřeby cvičení používána rozšiřující deska KIV-DPP-01, která obsahuje řadu periferií, pro které budeme v rámci cvičení psát ovladače. Tato deska obsahuje:

- senzor náklonu (binární)
- 7-segmentový displej
- OLED displej (na rozhraní I2C)
- tlačítko
- 2x dvou-polohový přepínač
- 1x LED
- pasivní bzučák (k použití s PWM)

Budeme tedy určitě potřebovat dokumentaci:

- BCM2835: ¡TODO:LINK¿
- ARM1176JZFS: ¡TODO:LINK¿
- příručku ARM6 assembly: ¡TODO:LINK¿
- RPi Zero pinout: ¡TODO:LINK¿
- KIV-DPP-01: ¡TODO:LINK¿

3 Seznámení s hardware

RPi Zero obsahuje již zmíněný mikrokontrolér BCM2835. Jedná se o tzv. System on a Chip (SoC), tedy integraci více komponent do jednoho pouzdra, které dohromady tvoří funkční programovatelný systém. Jádrem mikrokontroléru je procesor ARM1176JZFS, tedy procesor ARM řady 7 s taktovací frekvencí 800 Mhz. Tento procesor budeme programovat částečně v assembly a částečně v C/C++.

3.1 Registry procesoru

Procesor ARM1176JZFS obsahuje celkem 15 obecných registrů značených `r0 - r15`. Pak obsahuje dva řídicí registry (`cpsr` a `spsr`).

Některé z registrů mají speciální význam:

- `r13` (`sp`) - stack pointer, ukazatel na vrchol zásobníku
- `r14` (`lr`) - link register, registr obsahující návratovou adresu (pro návrat z volání podprogramu)
- `r15` (`pc`) - program counter, ukazatel na instrukci k načtení
- `cpsr` - current program state register - obsahuje příznakové bity (pro vykonání podmínek), stav procesoru, režim procesoru, apod.

- `spsr` - saved program state register - obsahuje uložený stav `cpsr` před vznikem výjimky (resp. před přechodem do jiného režimu procesoru, viz dále)

Tento procesor může operovat v následujících režimech:

- system/user - systémový (výchozí) a uživatelský režim
- FIQ - fast IRQ režim pro rychlé zpracování HW přerušení
- supervisor - privilegovaný režim
- abort - režim pro zpracování abort výjimek
- IRQ - IRQ režim pro zpracování neprioritních HW přerušení
- undefined - mód pro zpracování výjimky „neznámá instrukce“
- secure monitor - mód pro TrustZone Secure Monitor

Registry mohou být tzv. *banked* („uschované“). Každý režim má vždy svou „verzi“ registrů `r13` (stack pointer) a `r14` (link register), které jsou nahrány vždy při přechodu do daného režimu a uschovány při odchodu. Navíc režim FIQ uschovává registry `r8 - r12`. Uschované verze registrů jsou z privilegovaného režimu dostupné užitím prefixu (např. `r8_fiq`).

3.2 Assembly

Procesor ARM je schopen pracovat v několika režimech zpracování instrukcí. Mezi základní dva patří tzv. ARM režim a Thumb režim. Instrukce v ARM formátu mají fixní délku 32 bitů (4 byty) a jejich zarovnání v paměti tomu musí odpovídat - musí začínat vždy na adrese, která je násobkem 4. Instrukce formátu Thumb mají variabilní délku - buď 16 nebo 32 bitů a procesor musí být přepnutý do daného režimu, aby je uměl číst a vykonávat. Zarovnání musí v tomto případě být na násobky 2.

Pro moderní vývoj v assembly byl definován standard zápisu UAL (Unified Assembly Language), který dovoluje psát instrukce v unifikované podobě, a až kompilátor v době překladu rozhodne, která instrukční sada se použije. Vybrané prvky jazyka ale zůstávají stejné.

Následují vybrané konstrukce, které se mohou při programování pro ARM hodit.

Můžeme například definovat sekce direktivou `.section`, do kterých se má následující kus kódu/dat uložit. To se nám bude hodit v momentě, kdy budeme chtít uložit konkrétní věci na konkrétní místa. Typickým příkladem je právě bootovací sekvence.

```
.section .text
```

Můžeme definovat návěští pouhým jménem a dvojtečkou - návěští je pouze pojmenovaná adresa. To se bude hodit často - zejména pro definici rutin, které budeme volat z jazyka C (protože něco prostě v C zapsat neumíme).

```
mojefunkce:
```

Rovněž budeme chtít návěští exportovat tak, aby jej viděl linker (a potažmo ostatní moduly) direktivou `.global`.

```
.global mojefunkce
```

Komentáře se standardně v námi použitém assembleru značí středníkem a zavináčem.

```
;@ Toto je komentar
```

Konstanty lze definovat direktivou `.equ` - jde vlastně o obdobu maker v C.

```
.equ KONSTANTA, 42
```

Pak už můžeme pracovat s registry, pamětí, daty a jinými. Více pochopitelně v referenčním manuálu nebo v dodané „quick reference card“. Můžeme například chtít:

```
;@ presunout cislo do registru r1
mov r1, #99

;@ secist obsah dvou registru a vysledek dat do registru r2
add r2, r1, r3

;@ presunout provadeni programu jinam
b mojefunkce

;@ regulerne zavolat funkci (nastavuje LR)
bl mojefunkce

;@ vratit se z volani funkce
bx lr
```

Více až když to budeme potřebovat, resp. v daném referenčním manuálu.

3.3 Paměť a memory-mapped IO

Procesor má k dispozici adresní prostor, který lze adresovat čísly od 0 do `0xFFFFFFFF`. Jde tedy o 32-bitový adresní prostor. Z něj ale pouze část představuje fyzická paměť. Obvykle bývá fyzická paměť mapována na dolní část rozsahu, aby adresa na sběrnici odpovídala adrese ve fyzické paměti.

Zmínili jsme pojem „sběrnice“ a v souvislosti s tímto pojmem nějakou adresu. Když hovoříme o adrese, můžeme tím myslet adresu fyzickou, sběrnicovou, virtuální nebo jinou. V každém případě jde o číslo, které je často v procesu adresace různě překládáno a mapováno do adresního prostoru, ve kterém se právě pohybujeme.

Tohoto mapování využívají i některé periferie, které jsou mapovány do tohoto adresního rozsahu na předem známé adrese. Pro BCM2835 a tento procesor jde o rozsah fyzických adres `0x20000000 - 0x20FFFFFF`, které jsou ovšem mapovány na sběrnicový rozsah adres `0x7E000000 - 0xFEFFFFFF`. Procesor tedy pracuje s adresami z rozsahu sběrnicového, interně je ale tento rozsah mapován na rozsah fyzický. V dokumentaci pro BCM2835 nalezneme adresy sběrnicové.

Poté můžeme používat registry periferií pouhým vybráním příslušné buňky v dokumentaci a překladem do adresního prostoru. Například zápis na pin GPIO řadiče (aby se např. rozsvítila LED) můžeme provést zápisem bitu na specifickou adresu „set“ registru dané sady pinů. Příkladem může být následující kód, který zapíše 1 na pin 0 - adresu `0x7E20001C` nalezneme v BCM2835 manuálu jako adresu registru `GPSET0`:

```
(*(volatile int*)0x7E20001C) = (1 << 0);
```

Pochopitelně je potřeba provést ještě několik dalších věcí (nastavit režim GPIO, ...), ale o tom až jindy.

K drtivé většině periferií budeme přistupovat takto. K části se pak přistupuje pomocí principu tzv. *mailboxů*, o kterých si povíme až v pozdějších cvičeních.

3.4 Bootloader

Bootloader je ve zkratce program, který se spustí jako první po spuštění počítače. Je součástí operačního systému, který jej obvykle dodává ve svém instalačním balíku.

Bootovací proces u RPi Zero (a vlastně u spousty dalších embedded zařízení) je poněkud složitější. Po zapnutí desky je přečten z SD karty seznam oddílů, vybere se první, který je naformátovaný na souborový systém FAT32 a s tím se dále pracuje. Tam je nalezen soubor `bootcode.bin`, který představuje vlastní bootloader (stage 1). Ten přečte dále obsah souboru `config.txt` a jiných, nastaví co je třeba a do grafického jádra načte obsah souboru `start.elf`. Grafické jádro pak v bootování pokračuje tím, že procesor resetuje do pracovního stavu a načte do paměti z SD karty obsah souboru `kernel.img`. To je soubor, který

obsahuje samotný obsah paměti (RAM) a funguje tedy jako jakýsi „stage 2“ bootovacího procesu. Poté nastaví programový čítač na předem sjednanou hodnotu (`0x8000`) a předá řízení hlavnímu procesoru.

My bychom chtěli ve fázi vývoje neustále nahrávat nový kód do našeho zařízení. To můžeme například soustavným vyndaváním SD karty a přehráváním souboru `kernel.img`. To sice není úplně praktické, ale pro začátek to stačí.

Lepší nápad bude mít právě jeden `kernel.img`, který bude sloužit jen jako již zmíněná „stage 2“ bootovacího procesu. Tento kód bude čekat, až mu po nějakém komunikačním rozhraní skutečný kernel odešleme z našeho vývojového počítače, „stage 2“ ho pak nahraje do paměti a spustí. O tom ale až ve cvičení číslo 2.

3.5 Volací konvence

Ještě je třeba zmínit, jak probíhá volání funkcí od momentu předávání parametrů až po návrat včetně předání návratové hodnoty. Volání se týká všech registrů následovně:

r0-r3 - předávání parametrů

r0-r1 - předávání návratové hodnoty

r4-r11 - callee-saved registry – pokud tyto registry chce volaná procedura použít, je povinna jejich obsah předtím uložit na zásobník a před návratem tento stav obnovit

r11 - frame pointer – registr může být použit jako ukazatel na rámec volání podprogramu

r12 - temporary, scratch registr – registr, jehož obsah se může po volání procedury změnit

r13 (SP) , resp. zásobník jako takový - předávání parametrů které se nevejdou do registru, ukládání registrů pro přemazání; jeho hodnota by měla po návratu být stejná, jako před voláním podprogramu

r14 (LR) - obsahuje adresu, na kterou se má program vrátit po dokončení provádění podprogramu – volající ukládá hodnotu na zásobník při volání dalšího podprogramu

r15 (PC) - nastaven na cílovou adresu po přípravě kontextu, pro návrat je přemazán hodnotou `LR`

Nemá cenu zde probírat celou volací konvenci, jen je dobré zmínit pár důležitých bodů:

- `r13` (SP) musí být v momentě volání podprogramu zarovnaný na násobek 8

- přechody mezi režimy procesoru (ARM a Thumb) jsou možné jak při volání (instrukce `blx` a `bx`, např. `blx r11`), tak při návratu (`bx lr`) – režim je určen posledním bitem hodnoty registru
- volaná funkce nemusí okamžitě ukládat všechny registry – jen ty, které plánuje použít a klidně až před momentem prvního použití
- registry `r0 - r3` mohou plnit funkci scratch registrů, pokud nenesou parametry volání funkce

Více o volací konvenci ARM zde: `https://developer.arm.com/documentation/ihi0042/j/`

4 Nastavení prostředí pro vývoj

Jediné, co pro vývoj explicitně budeme potřebovat je překladač jazyka C/C++ a ARM assembly a sadu nástrojů pro převod výstupu do přenositelné podoby. Na to je k dispozici na různých distribucích Linuxu sada `gcc-arm-none-eabi`, která vlastně obsahuje vše, co budeme potřebovat.

Pro Debian-based (apt-based) distribuce stačí tedy použít příkaz (volitelně prefixovaný `sudo`):

```
apt install gcc-arm-none-eabi
```

Uživatelé systému Windows mohou použít od verze operačního systému Windows 10 i vestavěný WSL (Windows Subsystem for Linux) a nějakou instalovatelnou distribuci (např. Debian, Ubuntu, ...). Instalace je pak totožná. Odvážnější mohou zkusit zprovoznit balík `gcc-arm-none-eabi` pro Windows přímo z oficiálních distribucí (TODO:LINK) bez nutnosti instalovat WSL.

Samozřejmě bude dobré mít i nějaké IDE, které nám trochu pomůže s vývojem. Mně se osvědčilo MS Visual Studio Code s pluginy pro jazyk C++ a assembly.

5 Testovací kód

Pro začátek si zkusíme jen něco jednoduchého - rozblikáme LED na desce (tzv. ACT LED, activity LED) bez nutnosti mít jakkoliv použitelný operační systém.

Nejprve je potřeba napsat kus assembly, jelikož nelze nijak snadno a univerzálně v C kódu povědět, že chceme mít funkci, která vlastně není funkce, a že je zarovnána na přesnou adresu. Tedy - ne že by to nešlo, jen to dělat nechceme a musíme si trochu zvyknout na to, že budeme mixovat jazyky C a C++ s assembly.

Nejprve nakopírujme obsahy souborů, níže jejich obsah trochu rozebereme.

5.1 Kód

Definujme soubor `start.s`:

```
.global _start
.global dummy

;@ vstupni bod do kernelu
_start:
  mov sp,#0x8000
  bl blinker_main
hang:
  b hang

;@ dummy funkce (nevyoptimalizuje se; fixni pocet taktu procesoru
    )
dummy:
  bx lr
```

Rovněž si definujme soubor `blinker.c`:

```
#define GPFSEL3 0x2020000C
#define GPFSEL4 0x20200010
#define GPSET1  0x20200020
#define GPCLR1  0x2020002C

extern void dummy(unsigned int);

// zapise 32bitovou hodnotu na danou adresu
void write32(unsigned int addr, unsigned int value)
{
  *((volatile unsigned int*)addr) = value;
}

// precte 32bitovou hodnotu ze zadane adresy
unsigned int read32(unsigned int addr)
{
  return *((volatile unsigned int*)addr);
}

// aktivni "spanek" - spali nekolik taktu procesoru naprazdno
void active_sleep(unsigned int ticks)
{
  volatile unsigned int ra;
  for (ra = 0; ra < ticks; ra++)
    dummy(ra);
```

```
}

int blinker_main(void)
{
  unsigned int ra;

  // nastavime pin ACT LEDky na vystupni
  ra = read32(GPFSEL4);
  ra &= ~(7 << 21);
  ra |= 1 << 21;
  write32(GPFSEL4,ra);

  // nekonecna smycka
  while (1)
  {
    // ACT LED je zapojena "proti intuici", tzn. zapisem 0 se
    // rozsviti, zapisem 1 zhasne

    // 0 --> ACT LED (rozsvitit)
    write32(GPCLR1,1<<(47-32));
    // spalit 0x80000 cyklu ("pocka" par milisekund)
    active_sleep(0x80000);

    // 1 --> ACT LED (zhasnout)
    write32(GPSET1,1<<(47-32));
    // ...
    active_sleep(0x80000);

    write32(GPCLR1,1<<(47-32));
    active_sleep(0x80000);

    write32(GPSET1,1<<(47-32));
    active_sleep(0x300000);
  }

  return 0;
}
```

To budeme potřebovat namapovat nějak do paměti (linker nám s tím pomůže) pomocí souboru `memmap`:

```
MEMORY
{
  ram : ORIGIN = 0x8000, LENGTH = 0x10000
}
```

```
SECTIONS
{
  .text : { *(.text*) } > ram
  .bss : { *(.bss*) } > ram
}
```

A to všechno sestavíme dle návodu v `Makefile`:

```
ARMGNU ?= arm-none-eabi

AOPS = --warn --fatal-warnings
COPS = -Wall -Werror -O2 -nostdlib -nostartfiles -ffreestanding

all: kernel.img

clean:
  rm -f *.o
  rm -f *.bin
  rm -f *.hex
  rm -f *.srec
  rm -f *.elf
  rm -f *.list
  rm -f *.img

start.o: start.s
  $(ARMGNU)-as $(AOPS) start.s -o start.o

blinker.o: blinker.c
  $(ARMGNU)-gcc $(COPS) -c blinker.c -o blinker.o

blinker.elf: link.ld start.o blinker.o
  $(ARMGNU)-ld start.o blinker.o -T link.ld -o blinker.elf
  $(ARMGNU)-objdump -D blinker.elf > blinker.list

kernel.img: blinker.elf
  $(ARMGNU)-objcopy blinker.elf -O binary kernel.img
```

5.2 Význam konstrukcí

V souboru `start.s` vidíme 3 návěští:

- `_start` - vstupní bod do jádra, sem skočí bootloader

- `hang` - jen nekonečná smyčka, nemáme kam se „vrátit“ z kernelu, tak se zacyklíme
- `dummy` - dummy funkce, která se nevyoptimalizuje a má fixní počet taktů na zpracování, budeme ji volat kvůli aktivnímu zpoždění

Jediný kód, který vlastně něco dělá, se nachází v `_start` - nastaví ukazatel na vrchol zásobníku na adresu `0x8000` a zavolá C funkci `blinker_main`. Vrchol zásobníku nastavujeme na `0x8000`, a jelikož zásobník roste na opačnou stranu, tak vkládáním bude tato adresa klesat. Nepřepíšeme si tedy žádný z kódů, který je naopak situován do oblasti nad adresou `0x8000`.

Soubor `blinker.c` již obsahuje o něco zajímavější konstrukce. Zejména si všimněte, že je často používáno klíčového slova `volatile`. To proto, že pokud by kompilátor měl optimalizovat a narazil na zápis do místa, ze kterého se nikde v programu nečte, mohl by tento zápis odstranit. My ale potřebujeme, aby se zápis provedl, protože tím ovládáme vybrané periferie (které „čtou“ vždy to, co zapíšeme).

Dále jsou zde 4 funkce:

- `write32` - provede nevyoptimalizovaný zápis na danou adresu
- `read32` - přečte (necachovaně) z dané adresy
- `active_sleep` - aktivní „spánek“ („pálíme“ cykly procesoru na neužitečné práci)
- `blinker_main` - vlastní program pro blikání s ACT LED

První zajímavou funkcí pro nás je `active_sleep`, která pochopitelně nepředstavuje něco, co bychom měli použít v produkčním kódu. Jde o aktivní spánek, tedy vlastně jen sekvenci instrukcí, které se budou provádět bez většího užitku. My to využijeme pro časování blikání LED. Později si ukážeme, jak aktivní spánek nahradit nařízením časovače a využitím přerušení.

Následuje funkce `blinker_main`, která obsahuje vlastní logiku. Nejprve je nutné podotknout, že chceme blikat LED, ale fakticky jde o nastavování nějakého výstupu na „zapnuto“ (vysoká úroveň napětí, HIGH) a „vypnuto“ (nízká úroveň napětí, LOW). K tomu je potřeba ovládat tzv. GPIO (General-Purpose Input Output) řadič a jeho piny.

Tento řadič se dá ovládat zápisy do dedikovaných registrů, které jsou namapovány do paměťového prostoru. Pro úspěšné blikání s ACT LED potřebujeme nejprve nastavit její vyhrazený GPIO pin na výstupní (tedy my budeme ovládat výstup) a pak až pin nastavovat na žádanou úroveň napětí.

GPIO pin může operovat v několika režimech:

- input (vstupní)
- output (výstupní)
- alternate function 0-5 (speciální funkce pro nějaké periferie; více o tomto jindy)

Ten se nastavuje do příslušného registru zvaného `GPFSEL` (*function select*). Jakmile zapíšeme mód do tohoto registru, můžeme zápisem do registru `GPSET`, resp. `GPCLR` výstup ovládat.

ACT LED je připojena na GPIO pin 47. V manuálu se dočteme, že musíme nastavit registr `GPFSEL4` a pro ovládání registry `GPSET1` a `GPCLR1`. V kódu lze vidět čtení `GPFSEL4`, odmaskování 3 bitů na pozici 21, 20 a 19, které opět dle manuálu přísluší námi vybranému pinu, zápisu bitu na pozici 21 a zápis upraveného čísla zpět do registru. Tímto jsme nastavili pin 47 na výstupní.

Dále se dozvíme v manuálu, že pro ovládání výstupu existují registry `GPSET0`, `GPSET1` a jejich ekvivalenty `GPCLR0` a `GPCLR1`. Každý bit v každém z těchto 32-bitových registrů odpovídá jednomu pinu. `GPSET0` a `GPCLR1` obstarávají dolní piny 0-31, `GPSET1` a `GPCLR1` pak obstarávají piny 32-63 (teoreticky, některé nejsou použité).

Nyní můžeme v jedné velké smyčce nastavit ACT LED, aktivně čekat pár taktů a zase ji vypnout. V příkladu LED problikne dvakrát za sebou a pak následuje větší časové okno.

6 Úkol za body

Na rozšiřující desce KIV-DPP-01 je obsažena LED, která je připojena na pin ¡TODO¿. Rozblikejte ji tak, aby vysílala SOS signál (tři krátká, tři dlouhá a tři krátká probliknutí následovaná pauzou). Věnujte pozornost jak dokumentaci mikrokontroléru BCM2835, tak schématu přídavné desky.