Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Návrh jádra OS pro procesor CC2431

Plzeň 2012 Petr Mayr

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 11. května 2012

Petr Mayr

Abstract

This work is an introduction to the concepts of small operating systems suited for embedded systems. It explains basic principles of preemptive scheduling and inter-process communication. The actual implementation of kernel is described and application examples are presented.

Obsah

1 Úvod

Cílem této práce je uvést čtenáře do problematiky vícevláknových aplikací a operačních systémů. Je zde popsán programátorský model mikrokontroléru CC2430, jeho adresní prostor a periferie. Čtenář by měl získat ucelenou představu o základních principech plánování procesů, časování událostí a synchronizace procesů

Dále je zde důkladně popsána implementace navrženého systému a použitý překladač. Jádro je navrženo na míru mikrokontroléru CC2430, který se od CC2431 liší jen minimálně v organizaci paměťového prostoru. Výsledek této práce by měl zásadním způsobem zjednodušit a urychlit vývoj aplikací pro daný systém a umožnit jejich paralelní zpracování.

2 Hardware

2.1 Mikrokontrolér CC2430

CC2430 je 8-bitový mikrokontrolér stavěný na míru IEEE 802.15.4 a Zigbee aplikacím. Díky jeho mnoha úsporným režimům je velice vhodný pro systémy, kde je vyžadována extrémně nízká spotřeba energie, jako např. senzorické sítě, systémy pro automatizaci domácnosti, dálková ovládání, zabezpečovací systémy apod. Výdrž baterií u těchto zařízení je často srovnatelná s jejich životností, případně mnohem delší.

Jádrem CC2430 je vylepšená verze procesoru 8051 s 256 byty interní operační paměti. Jako zdroj hodinového signálu lze použít 16 MHz nebo 32 MHz oscilátor.

2.1.1 Adresní prostor

Organizace adresního prostoru vychází z Harwardské architektury. Procesor má celkem pět adresních prostorů: programovou paměť, interní a externí pamět pro data, bitově adresovatelnou paměť a funkční registry.

- Programová paměť (dále jen CODE) obsahuje všechny instrukce, konstanty a řetězce. Je adresována 16-bitovým registrem program counter (dále jen PC), nebo několika instrukcemi použitím 16-bitového data pointeru (dále jen DPTR). Lze tedy v principu adresovat 64 KB paměti.

- Externí paměťový prostor (dále jen XDATA) obsahuje všechny proměnné, datové struktury, a buffery, které se nevejdou do interní paměti. Může být adresována pouze nepřímo, a to přes registr DPTR, nebo přes 8-bitové registry R0 nebo R1. Druhý způsob je rychlejší, ale vyžaduje vložení horní části adresy do registru MPAGE[1].

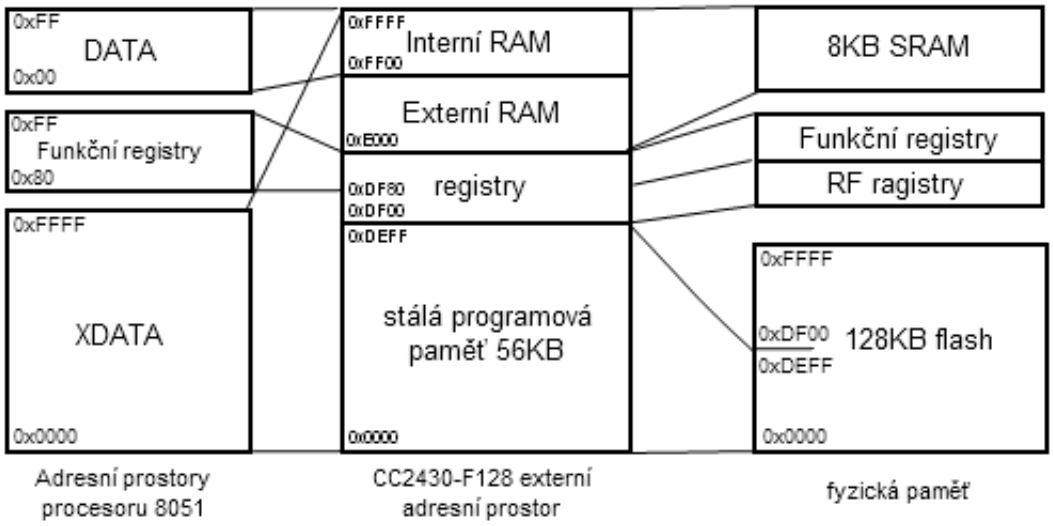
- V interním paměťovém prostoru jsou uloženy čtyři registrové banky,

[1] Část adresního prostoru zpřístupněná pomocí registru MPAGE překladač označuje jako PDATA.

bitově adresovatelná paměť a programový zásobník. Dolních 128 bytů lze adresovat přímo, zatímco horních 128 bytů pouze nepřímo.

- V prostoru funkčních registrů (dále jen SFR) se nachází všechny registry pro přístup k periferiím osazeným na čipu a některé registry procesoru, které musí být programově přístupné. Jde o akumulátor, program status word (dále jen PSW) a ukazatel na vrchol zásobníku (stack pointer, dále jen SP). SFR prostor překrývá horní část interního adresního prostoru a liší se tím, že může být adresován pouze přímo.
- Bitově adresovatelný prostor slouží k uložení příznakových bitů a bitových proměnných. Jeho maximální velikost je 256 bitů. 128 bitů překrývá dolní část interní paměti, zbylých 128 bitů představuje ty SFR registry, jehjichž adresa je dělitelná osmi. Může být adresován pouze přímo nebo použitím bitových instrukcí.

Na obr. 2.1 je ilustrován externí paměťový prostor mikrokontroléru CC2430-F128. Od standartního prostoru 8051 se liší ve dvou aspektech: interní pa-



Obrázek 2.1: externí adresní prostor CC2430-F128

měť, funkční registry a programová paměť jsou namapovány do XDATA, důsledkem čehož je zpřístupnění veškeré fyzické paměti DMA (direct memory access) řadiči, a lze použít dvě různá schémata pro mapování prostoru CODE. Použitím neunifikovaného schématu mapování dosáhneme toho, že do prostoru CODE je mamapována pouze paměť flash. Ta je dále rozdělena

na čtyři části o 32 KB. Do dolní části CODE je vždy namapováno spodních 32 KB flash paměti. Obsah horní části závisí na aktuálně zvoleném paměťovém banku, tedy lze do ní namapovat libovolnou ze čtyř částí flash paměti. Unifikované schéma do CODE mapuje dolních 54 KB flash paměti (z čehož 22 KB opět závisí na zvoleném banku) a do zbylé části registry a veškerou paměť RAM, a to stejným způsobem jako do XDATA. Výběr paměťového banku a schématu mapování se provádí zápisem příslušné hodnoty do registru MEMCTR.

Externí paměť RAM je namapována do XDATA od adresy 0xE000. Nutno podotknout, že v úsporných režimech PM2 a PM3 se uchovává pouze obsah paměti od adresy 0xF000 po 0xFD55, tedy pouze 3413 bytů. Od adresy 0xFF00 je znova namapovaná interní paměť procesoru.

2.1.2 Úsporné režimy

- PM0 - neboli aktivní režim. V tomto režimu je procesor, periferie, vysokofrekvenční oscilátory a napěťový regulátor v aktivním stavu. Spotřeba systému se pohybuje mezi 320μA a 550μA, závisí zjeména na aktuálním vytížení Zigbee transmitteru.

- PM1 - v tomto režimu jsou vysokofrekvenční oscilátory vypnuté. Napěťový regulátor a vybraný 32 KHz oscilátor jsou aktivní, spotřeba systému je 190μA. Tento režim je vhodné použít, pokud je očekávaná doba probuzení kratší než 3 ms. Přechod z PM1 do aktivního režimu trvá 4.1 μs.

- PM2 - pouze sleep timer, 32 KHz krystalový oscilátor a přerušovací systém jsou aktivní, všechny ostatní vnitřní obvody jsou vypnuté, celková spotřeba systému je 0.5μA. Z tohoto stavu může systém probrat pouze přerušení sleep timeru nebo externí přerušení. Přechod do PM0 trvá 120 μs.

- PM3 - všechny vnitřní obvody až na přerušovací systém jsou vypnuté, spotřeba je 0.3μA. Pouze externí přerušení může systém dostat do aktivního režimu, používá se tedy při čekání na externí událost. Přechod do PM0 trvá 120μs.

2.2 Periferní zařízení

2.2.1 Oscilátory a časovače

CC2430 má dva vysokogrefvenční oscilátory: 16 MHz RC oscilátor a 32 MHz krystalový oscilátor. Dále jsou zde 32,753 KHz RC oscilátor a 32,768 KHz krystalový oscilátor. Přesnost krystalových oscilátorů je 40 ppm. Vysokofrekvenční oscilátory jsou zdrojem hodinového signálu pro procesor, pro 16-bitový tříkanálový časovač, dva 8-bitové časovače a watch-dog timer. Dále je zde 24-bitový časovač (sleep timer), který má jako zdroj hodinového signálu jeden ze 32 KHz oscilátorů.

2.2.2 Řadič přerušení

Řadič přerušení obsluhuje celkem 18 možných zdrojů přerušení rozdělených do šesti skupin. Každé skupině může být přiřazena jedna ze čtyř priorit. Řadič je aktivní ve všech režimech napájení, přerušení je tedy obslouženo i v případě, když je systém zrovna v úsporném režimu. Každý zdroj přerušení lze individuálně povolit či zakázat. Když nastane přerušení, dojde k uložení obsahu programového čítače (program counter, dále jen PC) na zásobník[2], adresa příslušného přerušovacího vektoru je vložena do PC a procesor nastaví příznakový bit příslušného přerušení. Pro návrat z přerušovací rutiny je nutné použít instrukci `reti`.

2.2.3 Vstupně/výstupní porty

CC2430 má 21 digitálních vstupně/výstupních (I/O) pinů. Na všech pinech lze generovat přerušení, tedy v případě potřeby mohou externí zařízení komunikovat s procesorem pomocí přerušovacího systému. Piny lze nakonfigurovat jako běžné I/O piny nebo jako periferní I/O signály připojené na vstup zařízení na čipu, jako například A/D převodníku, časovače nebo komunikačního interface USART. Piny jsou organizovány jako dva 8-bitové porty P0, P1 a jeden pětibitový port P2. Porty jsou bitově adresovatelné, výběr funkce portu (pinu) ovládá registry PxSEL, nastavení směru, tedy zda jde o vstupní nebo

[2]Na zásobník se vždy ukládá adresa následující instrukce, tedy registr PC, a to nejprve jeho horní část.

výstupní port, ovládají registry PxDIR. Vstupní pin lze nakonfigurovat jako pull-up, pull-down nebo jako třístavový člen. Vstup do úsporných režimů PM2 a PM3 uchovává nastavení I/O funkce portů, a pokud to je možné i hodnotu, kterou měly před vstupem do daného režimu.

2.2.4 Další periferie

Na čipu CC2430 se dále nachází koprocesor pro Zigbee komunikaci a koprocesor pro AES šifrování, generátor náhodných čísel a CRC-16 polynomu. Dále dva nezávislé komunikační interface USART (universal synchronous/asynchronous reciever/transmitter), DMA řadič a analogově-digitální převodník. Tyto zařízení pro implementaci jádra nebyly využity.

3 Překladač

Při výběru překladače jsem se rozhodl pro SDCC (small device C compiler). Jde o kompilátor jazyka C určený pro 8-bitové mikroprocesory. Aktuální verze 3.1 je zaměřena na procesory Intel řady MCS51, Dallas DS\80C390, Freescale (dříve Motorola) HC08 a Zilog Z80. Zdrojový kód SDCC je šiřitelný pod GNU General Public License. Tato kapitola je zaměřena na popis paměťových modelů, některých klíčových slov a knihoven překladače.

3.1 Paměťové modely

Pro procesory řady MCS51 překladač nabízí výběr ze čtyř paměťových modelů: small, medium, large a huge. Modely medium, large a huge používají jako výchozí úložiště pro proměnné a parametry funkcí (vyjma reentrantních) do externí paměti. Model medium používá PDATA, modely large a huge používají XDATA. Model huge překládá všechny funkce jako banked, tedy rovnou je přiřazuje konkrétnímu paměťovému banku. Jinak je stejný jako model large. Model small jako výchozí úložiště používá interní paměť.

Na implementaci jádra jsem se rozhodl použít model large, protože je nutné efektivně využít jak interní tak externí operační paměť. Ideální by bylo použít model huge, ale SDCC ve verzi 3.1 ještě nemá dobrou podporu pro banking a pro tento model téměř chybí dokumentace.

3.2 Paramerty funkcí, lokální proměnné

První parametr funkce se předává přes registr DPTR (pokud se vejde) nebo přes kombinaci DPTR a R5, R6 a R7, a to pouze pokud je v těle funkce referencován do té doby, než se změní obsah použitých registrů. Jinak pro něj platí totéž, co pro další parametry. U reentrantních funkcí jsou parametry a lokální proměnné uložené na zásobníku. SDCC proto využívá globální proměnnou _bp, do které ukládá hodnotu SP v moment volání dané funkce. Chceme-li, aby se daný parametr nepředával přes zásobník, musí být deklarován jako static. Pokud v reentrantní funkci chceme mít proměnnou uloženou v konkrétní paměťové oblasti, musí být také deklarovaná jako static. Nereentrantní funkce mají parametry a lokální proměnné uložené (pokud

není explicitně určeno jinak) v interní pamětí, jejich pojmenování se řídí následující konvencí: parametry - `jméno-funkce_PARM_x`, lokální proměnné - `jméno-funkce_jméno-proměnné_x_y`, kde x a y značí číslo parametru nebo proměnné. Použitím přepínače –stack-auto nebo #pragma stack-auto překladač zkompiluje všechny funkce v daném souboru jako reentrant. Návratová hodnota funkce se předává opět přes DPTR (první byte přes DPL), nebo přes registry R5, R6 a R7.

3.3 Critical

Jako critical musí být deklarované takové úseky, kde za žádných okolností nesmí dojít k přerušení vykonávání kódu. Překladač proto využívá instrukce jbc (jump relative if bit operand is set and clear bit operand). Tato instrukce nastaví bitový operand na 0 a provede skok, pokud byla původní hodnota operandu 1. Nejprve uvedu příklad. Blok __critical {i++;} se zkompiluje následujícím způsobem:

```
        setb c
        jbc ea,00103$
        clr c
00103$:
        push psw
        mov dptr,#_i
        movx a,@dptr
        add a,#0x01
        movx @dptr,a
        pop psw
        mov ea,c
```

Nejprve se nastaví hodnota bitu C (carry) na 1. Pak se atomicky vynuluje registr EA (globální povolení přerušení) a za předpokladu, že jeho původní hodnota byla 1, se provede skok, tedy nedojde v vynulování bitu C. Jde o jediný způsob, jak zajistit vypnutí přerušovacího systému a zároveň uložit informace o tom, zda byl před tím zapnutý. Jako critical může být deklarován libovolný blok kódu. Critical lze též kombinovat s reentrant.

3.4 Naked

Deklarace funkce jako naked zabrání překladači generování jakéhokoliv prologu a epilogu k této funkci. Programátor je tedy zodpovědný za uložení registrů procesoru, jejichž obsah by se při volání funkce mohl změnít, a jejich následné obnovení, dále za správné uložení návratové hodnoty a za instrukci ret, tedy návrat z podprogramu, na konci funkce. Znamená to tedy, že tyto části kódu musí být napsané v assembleru. V implementaci jádra je to použito u přerušovacích rutin časovačů, které přepínají kontext z jednoho procesu na jiný, a u funkce, která zaručuje řádné ukončení procesu. U těchto rutin by takový prolog a epilog narušoval správné uložení a obnovu kontextu daného procesu. Naked nelze kombinovat s critical.

3.5 Knihovny překladače

3.5.1 Malloc, free, heap

SDCC pro práci s dynamickou pamětí nabízí knihovní funkci malloc. Mallocu se během startu systému přidělí část paměti dané velikosti (heap)[1]. Samotná alokace probíhá následovně: pokud je heap prázdný, uloží se na jeho začátek header, ve kterém je uvedena velikost alokovaného prostoru a ukazatel na následující volný segment paměti. Tento ukazatel se spočte jako součet adresy začátku heapu, velikosti headeru a velikosti alokované paměti. Při dalším volání mallocu již vzniká spojový seznam alokovaných segmentů. Uvolnění paměti probíhá tak, že se daný header označí jako volný. Pokud je i předchozí nebo následující segment také volný, spojí se do jednoho, na jehož začátku bude uvedena jeho velikost, informace o tom, že je volný a ukazatel na následující alokovaný segment. Alokace paměti využívá algoritmus first-fit, tedy seznam se prochází vždy od začátku a hledá se první volný segment, který má minimálně požadovanou velikost.

[1] Při kompilaci jádra je výchozí velikost heapu 2048 B, dá se změnit přepsáním hodnoty heap_size v makefile.

3.5.2 Násobení, dělení, modulo a jiné

Jak jsem již uvedl, procesor 8051 je 8-bitový, nemá tedy instrukce pro práci s proměnnými typu short (16 bitů) nebo long (32 bitů). Na operace násobení, dělení a modulo SDCC poskytuje knihovny. Tyto knihovny jsou předkompilované jako non-reentrant, proto je nutné je překompilovat s přepínačem –stack-auto. Dále je při kompilaci jádra nutné použít přepínač –int-long-reent.

Protože za běhu operačního systému může být proces kdykoliv přerušen, je nutné, aby všechny funkce které volá, byly reentrantní. Jinak by mohly nastat nepředvídatelné a těžko odhalitelné chyby v běhu procesu. Výstupem překladače je mimo jiné soubor hexfile.map, kde jsou uvedeny adresy všech funkcí a globálních proměnných. Neměly by se zde tedy být uvedeny žádné globální proměnné, o kterých si nejsme stoprocentně jisti, že se s nimi pracuje jako s kritickou sekcí.

4 Implementace jádra

V této kapitole se budu zabývat důkladným popisem jádra, plánování procesů, práce s časovači, synchronizací procesů a meziprocesovou komunikací. Protože procesor 8051 nemá prostředky (memory management unit) k tomu, aby měl každý proces svůj virtuální adresní prostor oddělený od ostatních, procesy sdílejí společný adresní prostor, a jde tedy spíše o vlákna než procesy. Z tohoto důvodu se nehodí na implementaci multi-uživatelského operačního systému.

4.1 Zásobník, přepnutí kontextu

Jedním ze zásadních nedostatků procesoru 8051 je hardwarová podpora zásobníku. Ukazatel na vrchol zásobníku je 8-bitový a pro zásobník lze využít pouze interní paměť, tedy maximálně 256 bytů. Aktuální stav vykonávání procesu je plně charakterizován obsahem registrů procesoru a obsahem zásobníku, na kterém jsou uloženy návratové adresy z volaných funkcí, případně i jejich parametry a lokální proměnné. Uložení kontextu procesu lze tedy realizovat pouze tak, že se uloží obsah registrů a zásobník na dané místo v externí paměti. Podobně vypadá obnovení kontextu - z externí paměti obnovíme zásobník a registry. Samotné přepnutí kontextu je tedy časově poměrně náročná operace. Konkrétně jde řádově o desetiny mikrosekundy.

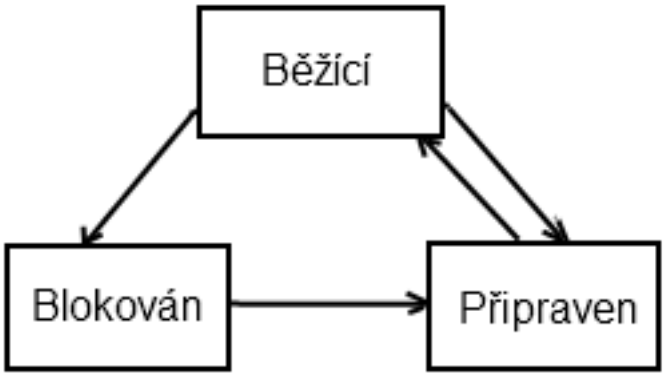
4.2 Proces

Soubor informací nutných pro řízení procesu, neboli process control block (dále jen PCB) má následující strukturu:

```
struct PCB{
       uint8_t pid;
       uint8_t state;
       int8_t priority_level;
       uint8_t *stack;
```

```
        uint8_t stack_actual_size;
        uint32_t wakeup_time;
        semaphore_t* waiting_processes;
}
```

Položka `pid`, neboli process ID je jednoznačný číselný identifikátor daného procesu. `State` udává stav procesu, `stack_actial_size` a `*stack` udávají aktuální velikost zásobníku a jeho umístění v paměti. `Waiting_processes` je ukazatel na semafor, na kterém jsou blokovány procesy čekající na jeho ukončení. O zbylých položkách se zmíním v dalších částech textu. Na obr. 4.1 je zachycen graf přechodů stavů procesu pro preemptivní plánování. Proces se může dostat ze stavu běžící do stavu připraven, tedy o tom, kdy dojde k přepnutí kontextu, rozhoduje operační systém. Plánování, kdy se čeká na proces,



Obrázek 4.1: graf přechodů mezi stavy

až se dostane do stavu blokován, nebo až sám předá řízení operačnímu systému, se označuje jako nepreemptivní [1]. Oba režimy mají své výhody. Jádro v preemptivním režimu má rychlejší odezvu a procesy sdílejí procesorový čas ”spravedlivě”. V nepreemptivním režimu nedochází k tak častému přepínání kontextu, tudíž režie systému je v tomto ohledu minimální. Využití vyrovnávacích (cache) pamětí je také mnohem efektivnější. Nepreemptivní plánování může dosahovat tedy lepší průchodnosti, jinými slovy více užitečné práce

[1] Jádro lze zkompilovat aby fungovalo jak v preemptivním režimu plánování, tak v nepreemptivním. Pro preemptivní plánování musí být v souboru kernel.h definována konstanta PREEMPTIVE_SCHEDULING

za méně času, ale nehodí se pro systémy, kde se očekává rychlá odezva a interakce s uživatelem.

4.2.1 Vytvoření procesu

Nový proces (vlákno) lze vytvořit pomocí funkce `process_init()`. Tato funkce má tři parametry. Prvním je ukazatel na funkci typu `int8_t fcn(void*arg)`. Druhým parametrem je ukazatel `void*arg`. Jde o adresu parametru, který chceme dané funkci předat. Poslední parametr je typu process_params_t*params, tedy ukazatel na strukturu, ve které jsou obsažený následující informace:

```
struct process_params{
       uint8_t stack_size;
       int8_t priority_level;
       uint8_t initial_state;
       uint32_t wakeup_time;
} process_params_t;
```

Tento parametr zahrnuje soubor informací, jakým způsobem má být daný proces inicializován. Položka `stack_size` udává velikost paměťového prostoru, který se vyhradí pro odkládání zásobníku. `Priority_level` udává prioritní úroveň, na které má být daný proces spuštěn, a `initial_state` výchozí stav procesu. Proces lze vytvořit buď ve stavu `READY`, nebo ve stavu `SLEEPING`. V druhém případě musí být ještě uvedena položka `wakeup_time`, která udává počet milisekund, za který má být nový proces probuzen. Funkce `process_init()` v případě úspěšné inicializace procesu vrací jeho *PID* (process ID). V případě chybných parametrů ke spuštění vrací konstantu `BAD_PARAMS`, v případě, nedostatku paměti vraci `MEMORY_ALLOCATION_ERROR`. Pro zjištění stavu procesu lze použít funkci `get_process_state()` a jako parametr jí předat *PID* procesu, jehož stav chceme zjistit.

4.2.2 Předání parametru procesu

Překladač pro reentrantní funkce předává první parametr přes registr DPTR, a to pouze v případě, že je referencován pouze do té doby, než se obsah DPTR změní. Jinak se předá přes zásobník. Nereentrantní procesy mají uložené parametry v globální proměnné. Protože je při vytváření nového pro-

cesu nemožné zjistit, jak daný proces s parametrem nakládá, nebo zda je reentrantní, rozhodl jsem se, že se parametr procesu vždy předá přes registr DPTR. U reentrantních procesů to má za důsledek to, že se parametr musí uložit do nějaké lokální proměnné dříve, než je použit registr DPTR a dále na něj neodkazovat. Pro nereentrantní procesy to znamená v podstatě to, že na začátku musí mít část kódu napsanou v assmebleru, ve které dojde k uložení DPTR do lokální proměnné, případně do té proměnné, do které by ji překladač normálně umístil.

4.2.3 Zásobník

Způsob inicializace zásobníku je odvozen od toho, jak probíhá uložení kontextu procesu. Před samotným uložením se už na zásobníku nachází adresa následující instrukce procesu, tedy registr PC. Během inicializace zásobníku se tedy na místo, kde by se nacházel obsah registru PC, uloží adresa funkce, kterou má daný proces vykonávat. Na místo, ze kterého se obnoví obsah DPTR, se uloží ukazatel na předávaný parametr. Aby mohl proces skončit, na místo v zásobníku, kde bude očekávána návratová adresa, se uloží adresa funkce, která zaručí jeho řádné ukončení a uvolnění prostředků nutných pro jeho řízení. Proces s takto inicializovaným zásobníkem je připraven ke spuštění.

Velikost zásobníku je nutné volit velmi obezřetně. Je třeba brát v potaz, že proces může být přerušen v nejméně vhodný okamžik. Velikost se tedy spočte jako `STACK_INIT_LENGTH` + dva byty za každé vnořené volání funkce + paměťová náročnost parametrů a lokálních proměnných daných funkcí v případě, že jsou deklarované jako reentrant. Je třeba uvážit i případná volání knihovních funkcí, jako např. násobení čísel typu long nebo `_gptrget`, které jsou vidět až z přeloženého kódu.

4.2.4 Čekání na ukončení procesu

Pokud potřebuje proces čekat na ukončení jiného procesu, lze použít funkci `wait_PID()`, která má jako parametr *PID* procesu, na který se má čekat, a vtací jeho exit kód. Na ukončení procesu může obecně čekat libovolný počet procesů. Pro tento účel je inicializován a použit semafor, na který odkazuje PCB daného procesu.

Během ukončení procesu systém uvolní paměť, která byla vyhrazena pro jeho zásobník, dále vymaže odkaz na jeho PCB z prioritních front plánovače.

Pokud na jeho ukončení čekají další procesy, nastaví se jejich stav na `READY` a na místo v jejich zásobníku, ze kterého se bude obnovovat registr DPL, se zapíše exit kód právě ukončeného procesu.

4.2.5 Násilné ukončení procesu

Násilné ukončení procesu lze realizovat pomocí funkce `kill_process()`, která má jako parametr *PID* procesu, který se má ukončit. Funkce probíhá stejně jako standartní ukončovací rutina s tím rozdílem, že pokud je proces ve stavu `SLEEPING`, je vybrán z fronty časově blokovaných procesů. Pokud na ukončení procesu čekají další procesy, je jim předána návratová hodnota `KILLED`.

Systém si nevede žádné údaje o tom, který proces je blokován na kterém semaforu. Pokud by tedy došlo k násilnému ukončení procesu blokovaného na semaforu, nebyl by vybrán z fronty procesů čekajících na daný semafor, zůstal by tam tedy odkaz na již neexistující proces, což by mohlo vést k nepředvídatelnému chování systému, případně jeho pádu. Při pokusu o ukončení takového procesu funkce `kill_process()` vrací hodnotu `REFUSED`.

4.2.6 Plánování procesů

Při inicializaci nového procesu je důležitým parametrem položka `priority_level`. Systém ukládá odkazy na řídící bloky spuštěných procesů do obousměrně zřetězených spojových seznamů, každý seznam představuje jednu prioritní úroveň[2]. Priorita je procesu dána při jeho inicializaci a nelze ji dále měnit. Na každé prioritní úrovni se ukládá informace o tom, který proces byl naposledy spuštěn. Rozhodnutí, který proces se bude vykonávat, provádí plánovač následujícím způsobem: postupně prochází seznamy v pořadí sestupně podle priority daného seznamu. Když narazí na neprázdný seznam, zjistí, zda existuje odkaz na naposledy spuštěný proces. Pokud ano, posune se na další položku seznamu, pokud ne, jako naposledy spuštšný se oznací první proces v seznamu. Dále testuje, zda je vybraný proces ve stavu `READY`. Pokud ano, na dané prioritní úrovní ho označí jako naposledy spuštěný, a danému procesu bude přidělen procesorový čas. Pokud ne, posune se na další položku seznamu. Výběr procesu na dané prioritní úrovni se tedy děje metodou round-robin.

[2]Změna počtu prioritních úrovní lze provést přepsáním hodnoty konstanty PRIORITY_LEVELS_NO v souboru kernel.h. Výchozí počet je nastaven na 8.

4.2.7 Systémový proces idle

Na nejnižší prioritní úrovní je při startu systému inicializován systémový proces idle. Tento proces je vždy ve stavu `READY`. Plánovač ho tedy vybere pouze v případě, že jsou ostatní procesy blokované. Idle se stará o p5echod do úsporných režimů procesoru. V případě neprázdné časové fronty vypočte čas, jak dlouho bude systém v režimu spánku. Pokud je tento čas delší než 3 ms, převede systém do režimu PM2. V případě kratšího spánku vybere PM1. Pokud je časová fronta prázdná, systém přejde do režimu PM3, tedy čekání na externí událost.

4.3 Časování událostí

4.3.1 Sleep timer

Sleep timer je 24-bitový časovač, který má jako zdroj hodinového signálu jeden ze 32 kHz oscilátorů. Tento časovač je spuštěn ihned po resetu a je aktivní ve všech režimech napájení, kromě PM3. Lze ho tedy použít pro časované probuzení systému z úsporných režimů PM1 a PM2. Aktuální hodnota čítače sleep timeru odpovídá hodnotám registrů ST2:ST1:ST0. V moment čtení registru ST0 je obsah registrů ST1 a ST2 hardwarově uložen do bufferu pro jejich následné čtení, čímž je zajištěna konzistence čtených dat. Zápis do komparátoru časovače se provádí přes tytéž registry. Samotný zápis ST2 a ST1 proběhne až v moment zápisu do ST0. Pokud je hodnota čítače rovna hodnotě zapsané do komparátoru, je nastaven příznak přerušení sleep timeru.

4.3.2 Systémový čas

Systémový čas je promenná typu `uint32_t`. Jde v podstatě o prodloužení registrů sleep timeru o jeden byte. Nejnižší byte systémového času odpovídá registru ST0, výchozí hodnota nejvyššího bytu je 0 a s každým (zjištěným) přetečením čítače sleep timeru se inkrementuje. Obnovení (update) systémového času probíhá následovně: nejprve se do dvou pomocných proměnných typu `uint32_t` `old_time` a `new_time` zapíše stará hodnota systémového času. Dále se první, druhý a třetí byte proměnné `new_time` nahradí hodnotami registrů ST0, ST1 a ST2. Pokud je potom hodnota proměnné `old_time` větší

než `new_time`, znamená to, že nastalo přtečení a inkrementuje se nejvyšší byte proměnné `new_time`, která v daný moment představuje systémový čas. Update času zajišťuje funkce `update_system_time()`. Tato funkce je deklarovaná jako *critical*, nemůže tedy dojít k nekonzistenci času.

Systémový čas tedy není čas jako takový, jeho update probíhá pouze pokud má systém naplánované časově vázané události. Při pravidelné obnově nastává přetečení času přibližně každých 36 hodin a 24 minut. Aktuální hodnotu systémového času vrací funkce `get_system_time()`.

4.3.3 Časová fronta, funkce sleep()

Jednou ze základních služeb, které jádro poskytuje, je funkce `sleep()`. Má jeden parametr typu `uint32_t`, který představuje počet milisekund, za který má být proces probuzen. `Sleep()` funguje následujícím způsobem: nejprve přepočte počet milisekund na počet tiků časovače a k výsledné hodnotě přičte aktuální systémový čas. Tímto je spočten absolutní čas probuzení procesu, který se zapíše do řídícího bloku daného procesu. Stav procesu je nastaven na `SLEEPING` a proces se zařadí do časové fronty. Časová fronta je prioritní fronta, ve které jsou události řazeny vzestupně podle času, kdy mají nastat. Pokud je událost probuzení daného procesu vložena na začátek této fronty, do registrů ST0 - ST2 se zapíšou spodní tři byty absolutního času této události a povolí se přerušení sleep timeru.

Obsluha přerušení sleep timeru se stará o provedení časově vázaných událostí a o jejich správné časování. Nejprve se uloží kontext právě vykonávaného procesu a obnoví se systémový čas. Dále se z časové fronty vyberou všechny procesy, které mají čas probuzení menší nebo roven systémovému času, jejich stav se nastavi na `READY`. Dále se zkontroluje stav časové fronty. Pokud je prázdná, zakáže se přerušení od sleep timeru. Pokud není, znovu se obnoví systémový čas a kontroluje se čas nadcházející události. Na tomto místě je důkladně ošetřeno, že čas nadcházející události je opravdu větší, než systémový čas[3]. Pokud je, proběhne nastavení sleep timeru stejným způsobem, jako ve funkci `sleep()`. Nakonec se zkontroluje systémový čas - pokud je jeho nejvyšší bit roven jedné, nastaví se na 0 stejně jako nejvyšší bity časů probuzení všech událostí v časové frontě, tedy od časů plánováných událostí a od systémového času se odečte (0x80000000). Následuje volání plánovače a obnova kontextu vybraného procesu.

Konstanta `LONGEST_SLEEP` udává maximální (bezpečný) počet milisekund,

[3]Funkce get_system_time() nefunguje nekonečně rychle a mohlo by se stát, že by se sleep timer nastavil na "minulost".

s jakým je možné volat funkci `sleep()`. Jde přibližně o 18 hodin. Pokud je dodržen tento limit, pak je úpravou časové základny a časů jednotlivých událostí zaručeno, že během výpočtu absolutního času události nedojde k přetečení, tedy vypočtený čas nebude nikdy menší než aktuální (systémový) čas. Při překročení je možné, že se čas vypočte špatně a proces bude hned zase probuzen.

V rámci obsluhy přerušení sleep timeru se mění čas nadcházející události a obsah časové fronty. Protože proces idle může být touto rutinou kdykoliv přerušen, obsluha se musí postarat o jeho reset, jinak by mohl být vybrán nevhodný úsporný režim. V nejhorším případě by to mohlo dopadnout následovně: v časové frontě by byla událost naplánovaná na dobu kratší než 3 ms. Idle by proto zvolil řežim PM1. Před samotným přechodem do tohoto režimu by byl přerušen obsluhou sleep timeru, v rámci které by se daný proces označil jako `READY`. Po skončení procesu by byl opět vybrán proces idle, který už by převedl systém do režimu PM1, a to i v případě, že by již nebyly naplánované žádné časově vázané události. Reset procesu idle zaručuje funkce `reset_idle()`.

4.4 Semafory

Jádro pro synchronizaci procesů poskytuje semafory. Semafor má následující strukturu:

```
struct semaphore{
      int32_t value;
      task_queue_t *waiting_list;
} semaphore_t;
```

Položka `value` je aktuální hodnota semaforu, `*waiting_list` je ukazatel na frontu procesů blokovaných nad tímto semaforem. Nejde o frontu typu FIFO (first in first out), ale o frontu, ve které jsou čekající procesy řazeny podle priorit. Tedy pokud se na semaforu nejprve zablokuje proces A s prioritou 2 a potom proces B s prioritou 3, při uvolnění semaforu se odblokuje nejprve proces B. Pro procesy na stejné prioritní úrovni se fronta chová jako standartní FIFO.

K inicializaci semaforu slouží funkce `semaphore_init()`. Prvním parametrem je ukazatel na strukturu typu `semaphore_t`, tedy na semafor, který má být inicializován. Druhým parametrem je hodnota, na kterou má být daný

semafor inicializován. Hodnota je typu `int32_t`. Inicializace semaforu obnáší alokaci prostředků pro frontu (konkrétně 5 bytů) a její následnou inicializaci. Pokud inicializace proběhne v pořádku, funkce vrací 0, v případě chyby alokace vrací -1. Funkce `semaphore_destroy()`, jejíž parametrem je ukazatel na semafor, zaručuje odblokování všech procesů čekajících na daný semafor a uvolnění prostředků pro frontu. Následuje popis operací, které lze nad semaforem provádět. Jde o funkce, jejichž jediným parametrem je ukazatel na na semafor, nad kterým má být daná operace provedena.

- `Semaphore_wait()`, neboli čekání na semafor, blokující. Pokud je hodnota semaforu menší než 0, proces je blokován a zařazen do fronty čekající na daný semafor. Hodnota semaforu se sníží o 1.
- `Semaphore_signal()` provede zvýšení hodnoty semaforu o 1. V případě, že fronta procesů blokovaných nad tímto semaforem není prázdná, dojde v vybrání jednoho procesu z fronty a k jeho odblokování.
- `Semaphore_trylock()`, neboli pokus o uzamčení semaforu, neblokující. Pokud je hodnota semaforu větší nebo rovna 0, hodnota se sníží o 1 a funkce vrátí `true`. Pokud je hodnota semaforu menší než 0, funkce vrátí `false`.
- `Semaphore_notify_all()` zvýší hodnotu semaforu o délku fronty procesů, které na daný semafor čekají, a tyto procesy odblokuje.

5 Závěr

Navržené jádro se podařilo úspěšně implementovat. Přestože je velice malé, jsou v něm obsaženy některé důležité funkce:

- preemptivní plánování, které minimalizuje rychlost odezvy systému,
- možnost vytvoření nebo ukončení procesu kdykoliv za běhu systému, možnost čekání na ukončení procesu nebo jeho násilného ukončení,
- semafory pro synchronizaci procesů a meziprocesovou komunikaci,
- efektivní implementaci časovačů,
- systémový proces idle, který zaručuje minimální spotřebu systému.

Snažil jsem se, aby jádro bylo snadno upravitelné a rozšiřitelné a maximálně modulární. Jeho přenositelnost nebyla opomenuta - je téměř celé naprogramované v jazyce C, proto by proces přenesení na jinou platformu znamenal úpravu ovladačů a částí kódu napsaných v assembleru. Dále je třeba zdůraznit, že zdrojové kódy lze překládat pomocí open-source překladače SDCC.

Stabilita systému byla důkladně testována, stejně jako všechny jeho funkce. Na testovacích aplikacích se systém zdá být stabilní a nevykazuje chyby. Před prohlášením jádra za stabilní by však bylo potřeba provést mnoho dalších testů a důkladnou analýzu přeložených kódů.

Literatura

A True System-on-Chip solution for 2.4 GHz IEEE 802.15.4 / ZigBee® Texas Instruments, 2012. Dostupné z http://www.ti.com/lit/gpn/cc2430

8051 Cross Assembler User's menual. MetaLink Corporation Chandler. Arizona 1996.
Dostupné z http://www.xess.com/manuals/asm51.pdf

SDCC Compiler User Guide.
Dostupné z http://sdcc.sourceforge.net/doc/sdccman.pdf