Praktická cvičení k programování mikropočítačů

Obsah

Úvod do programování mikropočítačů

Mikropočítač nebo také mikrokontrolér (anglicky microcontroller, zkratka MCU) je v podstatě miniaturní počítač integrovaný do jednoho čipu. Je to integrovaný obvod, který obsahuje procesor, paměť a další obvody jako jsou časovače, komunikační rozhraní, analogově-digitální převodník, atd.

Mikropočítače se v současnosti používají téměř ve všech elektronických přístrojích. Jednou z nejjednodušších aplikací mikropočítače může být výstražné světlo na kolo, které se ovládá jedním tlačítkem a má několik režimů blikání a svícení. Další příklady jsou různé systémy v automobilech (ABS, vstřikování, ovládání oken, centrální zamykání, alarm), radiopřijímače, přehrávače, digitální fotoaparáty, atd.

Pro programování mikropočítačů je možno využít různých programovacích jazyků. Na rozdíl od dřívějších dob už se jazyk asembler používá jen výjimečně. Průmyslovým standardem je jazyk C, případně C++.

Program se vytváří na běžném osobním počítači (stolním, notebooku) v tzv. integrovaném vývojovém prostředí (IDE). Toto prostředí slučuje editor pro psaní zdrojového kódu programu, nástroje pro překlad programu a nástroje pro ladění programu.

Postup tvorby programu pro mikropočítač:

- Zapíšeme program v textovém editoru
- Přeložíme program, vznikne binární soubor s příponou např. .elf, .bin, .s19
- Tento soubor (program) se nahraje do paměti mikropočítače. K tomu je potřeba buď malý zavaděč (bootloader) v mikropočítači, nebo programovací obvod, který může být buď ve formě samostatného zařízení (programátoru) nebo může být součástí desky plošného spoje s mikropočítačem. Jak je to běžné u vývojových kitů.
- Program se v mikropočítači spustí, případně je možno jej ladit

Tvorba programů pro mikropočítače má sice ve srovnání s tvorbou programů pro běžné počítače (PC) určitá specifika, ale v současnosti už je jich mnohem méně než v minulosti. S rozvojem výkonu i paměťových možností mikropočítačů se přešlo od jazyka asembler na vyšší jazyky (C, C++) a programátor není tolik omezen nedostatkem výkonu CPU a dalších zdrojů. I pro mikropočítače tedy platí stejná doporučení jako pro programování „větších“ počítačů - cestou k úspěchu je soustředit se na tvorbu srozumitelného, dobře čitelného kódu.

Část 1 - Programování mikropočítače NXP KL25Z v prostředí Kinetis Design Studio, váš první embedded program.

Úvod do „softwarové architektury“ pro mikropočítače – nekonečná smyčka

Minimální program pro mikropočítač (MCU) v jazyku C:

```
void main(void)
{
      X_init();    /* priprav se na ulohu X */

      while(1)  {
         X();      /* Proved ulohu X */
      }
}
```

Jeden ze základních rozdílů mezi program vytvořenými pro embedded systémy (mikropočítače) a programy vytvořenými pro jinou počítačovou platformu je, že embedded programy téměř vždy obsahují **nekonečnou smyčku**.

Nekonečná smyčka je nutná, protože práce embedded programu nikdy nekončí. Předpokládá se, že poběží, dokud nenastane konec světa nebo dokud není mikropočítač restartován – podle toho, co nastane dříve.

Na většině embedded systémů běží pouze jeden program. I když hardware je také důležitý, bez software je k ničemu. Pokud software mobilní telefonu, digitálních hodin nebo třeba mikrovlnné trouby přestane pracovat, hardware se stává zbytečný. Proto je část embedded programu, která zajišťuje funkčnost zařízení téměř vždy vložena do nekonečné smyčky, která zajistí, že poběží navždy.

Výhody a nevýhody systémů s nekonečnou smyčkou

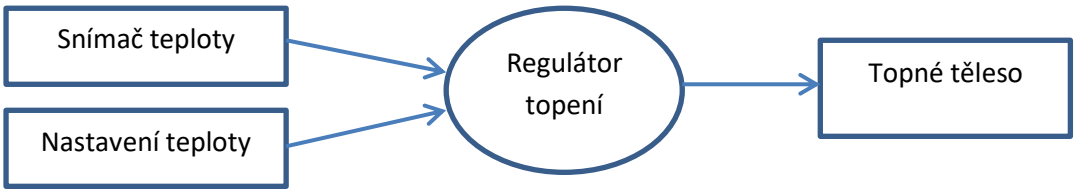
Výhody:

- Jsou jednoduché
- Jsou efektivní, mají malé požadavky na hardware, např. MCU nemusí mít modul časovače.
- Jsou snadno přenositelné – budou pracovat na jiném MCU s minimální potřebou změn v kódu.

Nevýhody:

- Nejsou přesné – pokud je potřeba měřit data každých 5 ms, tento typ systému neposkytne dostatečnou přesnost ani pružnost.
- V základní verzi pracuje systém stále „na plný výkon“. To znamená, že má velkou spotřebu energie.

Příklad vestavěného (embedded) systému – regulátor topení




```
void main(void)
{
  // Inicializace systemu
  TOPENI_Inicializuj();

  while(1) {   // nekonečná smyčka

    // Zjistíme, jakou teplotu požaduje uživatel – z uživatelského rozhraní
    TOPENI_Cti_Nastavenou_Teplotu();

    // Změříme skutečnou teplotu ze snímače teploty
    TOPENI_Cti_Aktualni_Teplotu();

    // Nastavíme výkon topeni podle potřeby
    TOPENI_Nastav_Topne_Teleso();
  }
}
```

Práce s diskrétními vstupy a výstupy

Vstupy a výstupy MCU se ovládají pomocí registrů. Prozatím budeme využívat připravené funkce, které pracují pouze s určitými piny (na kterých jsou na kitu připojeny LED a tlačítka). Později se dozvíme, jak tyto funkce pracují uvnitř a jak ovládat libovolný pin.

Dostupné funkce:

gpio_initialize –inicializuje ovladač (sadu funkcí) pro práci s vstupně/výstupními piny. Musí být voláno před používáním ostatních funkcí ovladače.

pinMode – nastavuje pin jako vstup nebo výstup.

pinWrite – nastavuje na výstupním pinu log. 1 nebo log. 0.

pinRead – vrací hodnotu HIGH nebo LOW, podle toho, jaká je napětí na vstupním pinu.

Podrobná dokumentace funkcí je uvedena v hlavičkovém souboru ovladače: drv_gpio.h. Tento ovladač je již vložen v ukázkovém projektu, najdete jej v IDE ve složce Sources.

Softwarové zpoždění (delay)

Jednoduché zpoždění, které nevyžaduje žádný speciální hardware (časovač):

```
void delay(void)
{
   long cnt;
   for ( cnt=0; cnt < 30000; cnt++)
      ;
}
```

Výhody a nevýhody softwarových zpoždění

Výhody:

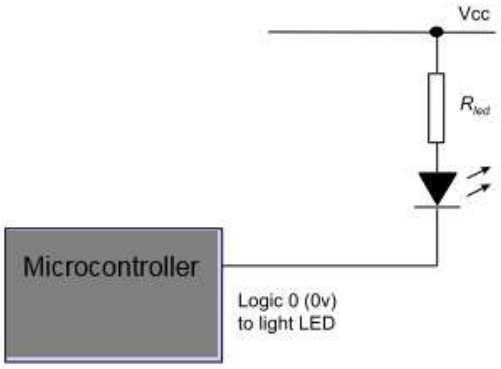
- Mohou vytvořit velmi krátká zpoždění.
- Nevyžadují žádné hardwarové časovače.
- Budou fungovat na libovolném mikropočítači.

Nevýhody:

- Je obtížné vytvořit zpoždění s přesnou dobou trvání.
- Smyčka (smyčky) musí být znovu „naladěny“ pokud se rozhodneme použít jiný mikropočítač, změníme nastavení hodinové frekvence procesoru nebo jen změníme nastavení optimalizací překladače.

Připojení LED k mikropočítači

LED je možno připojit přímo k pinu MCU, ale pro omezení proudu je nutno použít rezistor.



Příklad:

Napájecí napětí: Vcc = 5 V

Napětí LED v propustném směru (forward voltage): Vled = 2 V

Požadovaný proud diodou: Iled = 10 mA

Potřebná hodnota rezistoru: Rled = (Vcc – Vled) / Iled =(5 – 2) / 0,01 = 300 Ohm

Rezistory se vyrábějí jen v určitých řadách hodnot. Pokud vypočtená hodnota není dostupná, použije se nejbližší **vyšší** hodnota. Např. pro vypočtenou hodnotu 300 Ohm můžeme použít běžně dostupný rezistor 330 Ohm (330R).

LED na školním vývojovém kitu

Jsou k disppozici 3 led na hlavní desce:

- LD1 – červená, pin PTB8
- LD2 – žlutá, pin PTBB9
- LD3 – zelená, pin PTB10

A další 3 LED (přesněji trojbarevná RGB LED) přímo na desce s mikropočítačem FRDM-KL25Z:

- Červená, pin PTB18
- Zelená, pin PTB19
- Modrá, pin PTD1

Všechny LED jsou zapojeny tak, že svítí, pokud je na pinu log. 0 a nesvítí při log. 1.

Příklad 1: Blikání LED

Projekt **gpio_blink** obsahuje ukázkový program blikání LED s využitím předpřipravených funkcí (ovladač gpio).

Návod jak tento projekt spustit najdete v dokumentu o práci s IDE Kinetis Design Studio (KDS). Postupujte v těchto krocích:

- Importovat ukázkový projekt do vašeho workspace v KDS – kapitola Import existujícího projektu.
- Přeložit program – kapitola Překlad programu.
- Spustit program – kapitola Spuštění programu.

Příklad 2: Ladění programu

Projekt **gpio_buggy** obsahuje program blikání LED, který ale obsahuje chyby.

Vaším úkolem je tyto chyby najít a opravit.

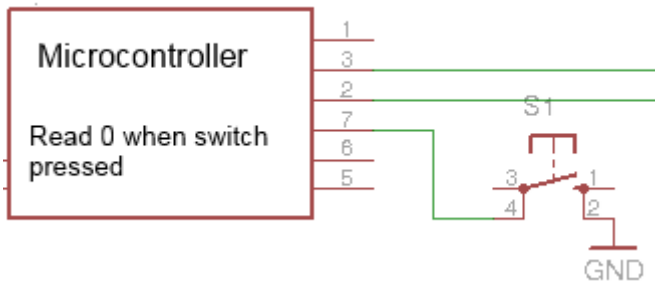
Návod jak ladit programy najdete v dokumentu o práci s IDE Kinetis Design Studio (KDS) v kapitole Ladění programu.

Tip: Na začátku zdrojového kódu gpio_buggy.c si můžete zvolit obtížnost ladění, na kterou si troufáte pomocí makra OBTIZNOST:

```
// Definujte si obtiznost:
// 0 = normal
// 1 = nightmare (nocni mura :) )
#define OBTIZNOST       0
```

Připojení tlačítka k mikropočítači

Tlačítka jsou běžnou součástí uživatelského rozhraní embedded systémů. Spolehlivé vyhodnocování tlačítek je potřebné téměř v každém embedded zařízení, od jednoduchého dálkového ovládače pro televizi až po autopilota v letadle.



- Když není tlačítko stisknuto (kontakt rozpojen), neovlivňuje nijak pin mikropočítače. Interní rezistor napětí na pinu „vytahuje nahoru“ (**pull up**) na úroveň napájecího napětí. Pokud pin přečteme, dostaneme hodnotu log. 1.
- Když je tlačítko stisknuto (kontakt spojen), je pin mikropočítače přímo připojen na zem (GND) a napětí na pinu bude 0 V. Pokud pin přečteme, dostaneme hodnotu log. 0.

Proč je potřebný pull-up rezistor?

Protože bez pull-up rezistoru by byl pin mikropočítače nezapojen („ve vzduchu“), pokud není tlačítko stisknuto. Proto by na pinu nebylo žádné napětí. Logická úroveň na pinu by byla nedefinovaná. Z pinu bychom pravděpodobně přečetli log. 0, tedy stejnou hodnotu, jako při stisknutém tlačítku.

Většina mikropočítačů má pull-up rezistory přímo integrované na čipu. Některé MCU mají i pull-down rezistory.

Tlačítka na školním vývojovém kitu

K dispozici jsou 4 tlačítka označená SW1 až SW4.

- SW1 – pin PTA4
- SW2 – pin PTA5
- SW3 – pin PTA16
- SW4 – pin PTA17

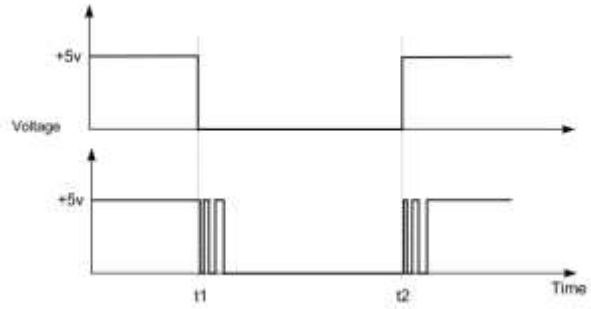
Pokud je tlačítko stisknuto, čteme z příslušného pinu hodnotu log. 0, pokud není stisknuto, log. 1.

Zákmity tlačítek

Všechny mechanické kontakty mají zákmity. Zákmity znamenají, že kontakt se opakovaně sepne a rozepne během krátkého času.

Anglické označení zákmitů tlačítka je switch bounce. Technika pro zvládnutí zákmitů se pak nazývá debouncing.

Zákmit tlačítka je znázorněn na obrázku:



Na vodorovné ose je čas, na svislé ose je napětí na pinu MCU.

Horní průběh je ideální tlačítko bez zákmitů. Po spojení kontaktů je na pinu logická nula. Po rozpojení kontaktu přechází na log. 1 a dále je už logická 1.

Spodní průběh je skutečné tlačítko se zákmity. Po stisku tlačítka se na pinu objeví log. 0, ale vzápětí následuje v rychlém sledu několik přechodů mezi log. 0 a log. 1. Doba zákmitů se liší pro různé tlačítka, ale obecně se pohybuje v milisekundách a většinou nepřesáhne 20 ms.

Pro program v mikropočítači je každý zákmit jako jedno stisknutí a uvolnění tlačítka. To může způsobovat problémy jako:

- Místo „A“ přečte program z klávesnice „AAAA“.
- Je obtížné počítat, kolikrát bylo tlačítko stisknuto.
- Pokud je tlačítko stisknuto a po nějaké době uvolněno, kvůli zákmitům se to může v programu jevit, jakoby bylo stisknuto znovu (v okamžiku, kdy je ve skutečnosti uvolněno).

Jak ošetřit zákmity tlačítek

Ošetřit zákmity lze buď hardwarově, vhodnou konstrukcí obvodu mezi tlačítkem a MCU. Nebo softwarově, např. takto:

- Přečíst hodnotu na pinu.
- Pokud se zdá, že bylo tlačítko stisknuto, počkat asi 20 ms a pak znovu přečíst pin.
- Pokud druhé čtení potvrzuje výsledek prvního (tlačítko stále stisknuto), předpokládáme, že tlačítko bylo opravdu stisknuto.

Příklad 3: Čtení vstupu z tlačítka

Kód je v projektu **gpio_switch**. LED indikuje stav tlačítka. Pokud je tlačítko stisknuto, LED svítí, pokud není stisknuto, LED nesvítí.

Takový kód může být použit pro operace jako:

- Rozsvícení světla po dobu, kdy je stisknuto tlačítko
- Zapnutí motoru (nebo např. čerpadla) dokud je tlačítko stisknuto

Stejná funkčnost by mohla být realizována s použitím obyčejného elektrického spínače, bez použití mikropočítače. Mikropočítač ale umožňuje realizovat komplexnější chování, např.:

- Světlo svítí při stisknutém tlačítku, ale požadavek je ignorován, pokud je dost denního světla

- Při stisku tlačítka zapnout čerpadlo, ale pokud je v nádrži méně než 100 litrů vody, nezapínat čerpadlo v hlavní nádrži, ale namísto toho zapnout čerpadlo v rezervní nádrži a napumpovat vodu z rezervní nádrže.

Pozor ovšem na problémy, které mohou nastat při použití tohoto kódu. Uživatel obvykle stiskne tlačítko na dobu kolem 500 ms a více. Pokud kód programu vyhodnocuje stisk tlačítka plnou rychlostí (jako je to v tomto příkladu), pak:

- Pokud bychom chtěli počítat, kolikrát bylo tlačítko stisknuto, program by „detekoval“ mnoho stisknutí místo jednoho.
- Pokud bychom takový kód použili pro přepínání stavu světla, motoru apod., stav by byl přepnut mnohokrát po každém stisku a výsledek by byl náhodný.

V takových případech bychom mohli čekat na uvolnění tlačítka předtím, než provedeme příslušnou akci. Nebo můžeme provést příslušnou akci hned (aby uživatel dostal zpětnou vazbu, že jeho příkaz je proveden) a pak čekat, až bude tlačítko uvolněno předtím, než budeme čekat na nový stisk tlačítka.

Doporučený postup práce a náměty k procvičení

1. Importujte projekt **gpio_blink**, přeložte jej a nahrajte do MCU. Vyzkoušejte úpravy tohoto programu, např.
    - Upravte program tak, aby bylo možno snadno změnit používanou LED a tlačítka bez nutnosti přepisovat kód na několika místech.
    - Rozblikejte více LED (mohou blikat současně nebo střídavě,…)
    - Změňte rychlost blikání LED
    - Vytvořte světelný efekt z LED, např. běžícího světla
2. Zjistěte, jak dlouho trvá zpoždění realizované funkcí delay() v ukázkovém programu. Vyzkoušejte různá zpoždění a vytvořte vlastní funkci nebo funkce pro pevné zpoždění zvolené délky, např. 100 ms. Funkce vhodně nazvěte, např. delay_100ms().
3. V ukázkovém projektu **gpio_buggy** odladťe chyby, tak aby fungoval podle zadání.
4. Předveďte vložení breakpointu na určené místo programu a krokování programu příkazem Step Over.
5. V ukázkovém programu **gpio_switch** nastudujte vyhodnocení stisku tlačítek na kitu. Vyzkoušejte úpravy programu, např. použití jiného tlačítka.
6. Vytvořte program, který bude využívat LED a tlačítka na kitu podle vlastního zadání. Např. můžete spouštět světelné efekty LED pomocí tlačítek.

Kontrolní otázky

1. Proč je v programech u mikropočítačů obvykle použita nekonečná smyčka.
2. Jaké výhody a nevýhody má použití nekonečné smyčky.
3. Jak se realizuje softwarové zpoždění? Jaké má výhody a nevýhody.
4. Jak se liší diskrétní a analogové vstupy?
5. Uveďte příklady diskrétních vstupů a výstupů.
6. Nakreslete schéma připojení LED k mikropočítači.
7. Nakreslete schéma připojení tlačítka k mikropočítači.
8. Co jsou to zákmity tlačítka? Jaké problémy mohou způsobit?

Část 2 – Odezva programu

V této části se budeme zabývat tím, jak tvořit programy pro mikropočítač tak, aby prováděly (zdánlivě) několik činností najednou (tzv. concurrent execution).

Programy pro mikropočítače většinou musí řešit několik činností současně. I jednoduchý program jako je blikačka na kolo musí zajistit blikání LED a současně vyhodnocovat stav tlačítka – aby uživatel mohl blikání vypnout a zapnout.

Toho je dosaženo sdílením procesoru mezi jednotlivými aktivitami.

Jako **odezvu programu (responsiveness)** budeme označovat to, jak rychle dokáže systém reagovat na událost na vstupu. Pokud má program dobrou odezvu (je responzivní), myslíme tím, že včas reaguje na nějakou událost, např. na stisk tlačítka.

Pokud uživatel musí např. držet tlačítko stisknuté dlouhou dobu, než program zareaguje, může si myslet, že systém nefunguje nebo že stisk tlačítka ignoroval.

Poznámka: Pojem concurrent execution se může plést s paralelním prováděním úloh (parallel execution), protože obojí se dá popsat jako „provádění několika úloh současně“. „Concurrent“ znamená, že úlohy se vykonávají zdánlivě současně, ve skutečnosti jedna za druhou, na procesoru se střídají. Lze to tedy realizovat na jedné procesní jednotce. Paralelní provádění znamená, že se úlohy skutečně provádějí ve stejném čase, současně, tj. je potřeba více procesních jednotek (procesorů/jader).

Ukázkový program – blikač

Problémy s odezvou programu si ukážeme na jednoduchém programu pro blikání LED. Program najdete v balíku ukázek v projektu **response**.

Požadované chování programu:

1. Po startu nesvítí žádná LED.
2. Po stisku tlačítka se budou postupně rozsvěcovat LED1 až LED3 – vždy jedna LED bude svítit, ostatní budou zhasnuté, tj. nejdřív se rozsvítí LED1, pak LED2, pak LED3, pak zase LED1, LED2, atd. Takže vznikne efekt „běžícího světla“.
3. Při uvolnění tlačítka má světelný efekt skončit.

Blikač verze 1

Toto je hlavní část kódu programu podle zadání výše. V ukázkovém projektu **response** jej aktivujete nastavením symbolu VERSION na 1 (#define VERSION 1).

```
// Stavy programu
#define ST_EFFECT   1
#define ST_OFF      2

int state = ST_OFF;
```

```
int main(void) {
	// inicializace ovladace pinu a delay
	init();

	while (1) {
		if (switch1_read() == SWITCH_PRESSED)
			state = ST_EFFECT;
		else
			state = ST_OFF;

		switch (state) {
		case ST_OFF:
			LED_control(false, false, false);
			break;

		case ST_EFFECT:
			LED_control(true, false, false);
			SYSTICK_delay_ms(BLINK_DELAY);
			LED_control(false, true, false);
			SYSTICK_delay_ms(BLINK_DELAY);
			LED_control(false, false, true);
			SYSTICK_delay_ms(BLINK_DELAY);
			break;
		}	// switch

	}	// while

	/* Never leave main */
	return 0;
}
```

Vysvětlení kódu

Program je napsán jako **konečný automat** (podrobnosti najdete v dokumentu „Konečný automat“). Program má 2 stavy:

- ST_OFF, kdy LED nesvítí
- ST_EFFECT, kdy probíhá světelný efekt postupného rozsvěcení LED.

Program nejprve zjišťuje, jestli je stisknuto tlačítko. Pokud ano, pak nastaví proměnnou pro stav programu – state – na hodnotu „efekt“ tj. postupné rozsvěcování LED.

V další části kódu se pomocí switch vyhodnocuje aktuální stav a podle toho se provádí činnost.
V případě stavu ST_EFFECT se:

- rozsvítí první LED,
- pak se program pozastaví se na určitou dobu,
- rozsvítí druhou LED (a zhasne první)
- opět se pozastaví na určitou dobu, atd.

Pokud tlačítko není stisknuto, program nastaví stav na „nesvítí“ – ST_OFF a v tomto stavu pouze zhasíná všechny LED.

V programu jsou použity dvě pomocné funkce:

switch1_read vrací hodnotu SWITCH_PRESSED, pokud je stisknuto tlačítko SW1.

LED_control rozsvěcuje nebo zhasíná LED LD1 až LD3 podle hodnot vstupních parametrů. Parametry jsou hodnoty bool pro jednotlivé LED LD1 až LD3: void LED_control(bool d1, bool d2, bool d3);

Analýza odezvy programu

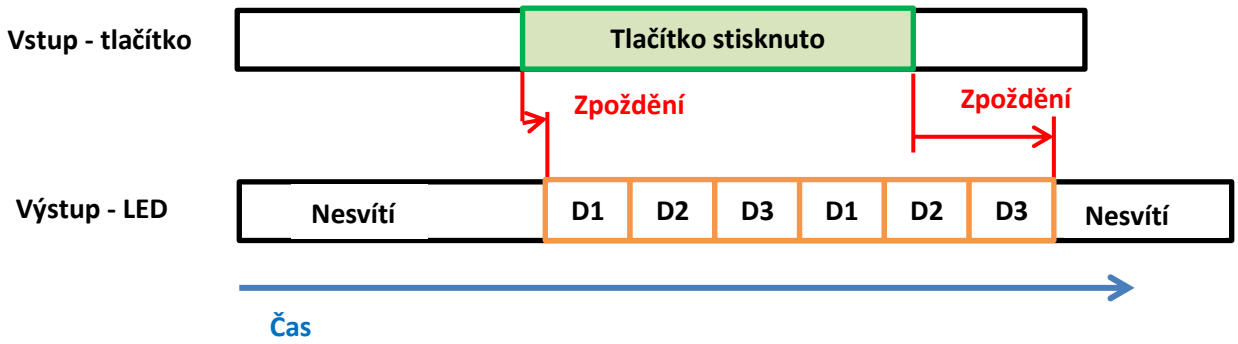
Jak rychle reaguje náš program na stisk a uvolnění tlačítka?

Pokud tlačítko není stisknuto, LED jsou zhasnuty. Když tlačítko stiskneme, rozsvítí se nejdřív LED LD1, za chvíli zhasne a rozsvítí se LD2 atd. Po zhasnutí LD3 se rozsvítí zase LD1 a sekvence se opakuje.

Reakce na stisk tlačítka je rychlá, protože ve stavu, kdy LED nesvítí, program provádí pouze test tlačítka a zhasnutí LED. Tyto operace trvají krátkou dobu a stav tlačítka se tedy testuje velmi často.

Při uvolnění tlačítka je odezva horší.

Jak vidíme na obrázku, když uvolníme tlačítko v době, kdy svítí D2, LED D2 bude svítit dál, pak se rozsvítí D3 a teprve po jejím zhasnutí přestanou LED svítit. Program totiž kontroluje stav tlačítka jen mezi jednotlivými sekvencemi bliknutí D1-D2-D3.



Jak zlepšit odezvu programu?

Jedna možnost – ale špatná – by byla přidat kontrolu tlačítka za každé rozsvícení LED v sekvenci.

Podívejte se na následující kód, který ukazuje **špatné řešení**:

```
case ST_EFFECT:
	LED_control(true, false, false);
	SYSTICK_delay_ms(BLINK_DELAY);
	// Pokud neni stisknuto tlacitko přeruš sekvenci
	if ( switch1_read() != SWITCH_PRESSED )
		break;
	LED_control(false, true, false);
	SYSTICK_delay_ms(BLINK_DELAY);
	// Pokud neni stisknuto tlacitko přeruš sekvenci
	if ( switch1_read() != SWITCH_PRESSED )
		break;
	LED_control(false, false, true);
	SYSTICK_delay_ms(BLINK_DELAY);
	break;
```

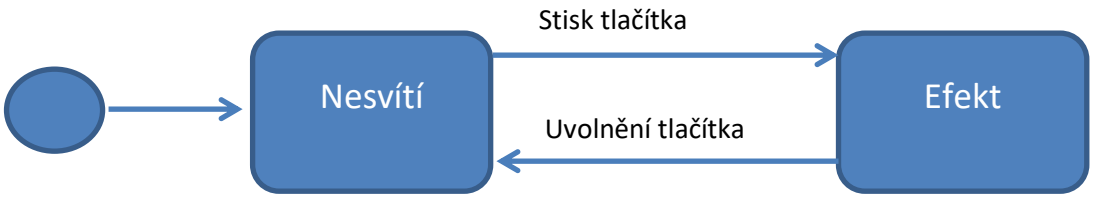
Proč je to špatné řešení?

Protože se tím míchají nesouvisející věci dohromady (ovládání LED s čtením tlačítek). Vznikne špatně strukturovaný kód označovaný jako **„špagetový kód“** (spaghetti code).

Jednotlivé činnosti v programu jsou propleteny dohromady jako špagety. Takový program je pak nepřehledný, špatně se upravuje a rozšiřuje. **Je vhodné v kódu programu oddělovat jednotlivé činnosti.** Např. aby každou činnost vykonávala jedna funkce.

Blikač verze 2 - vylepšení odezvy programu přidáním stavů pro jednotlivé LED

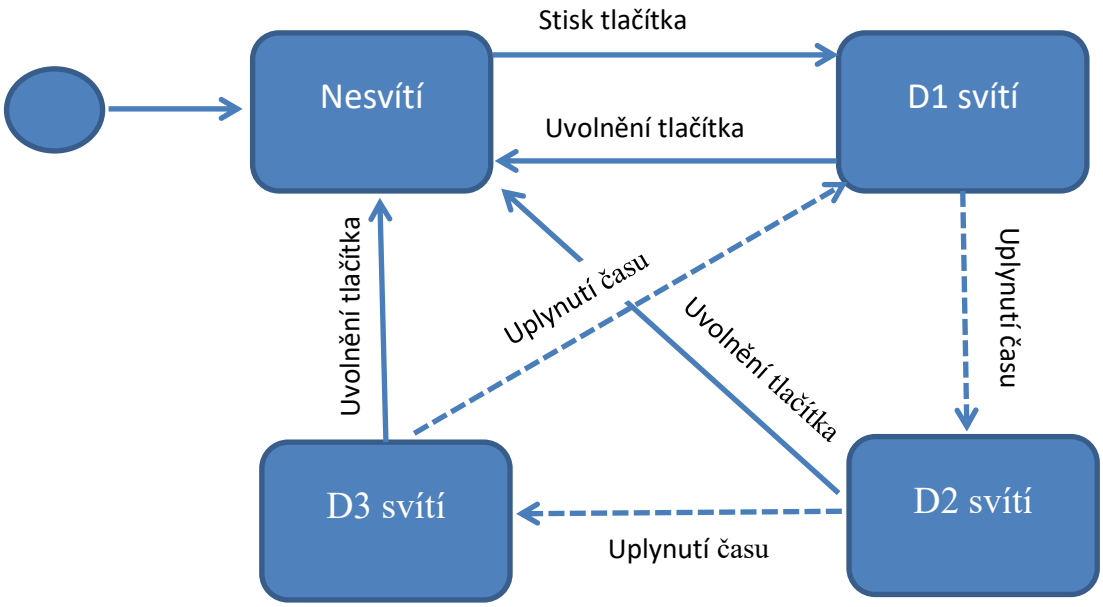
Náš program má zatím 2 stavy – Nesvítí a Efekt. Viz obrázek.



Problém je, že ve stavu Efekt se provádí celá sekvence postupného rozsvícení tří LED, která trvá dlouho, a proto se tlačítko testuje jen jednou za dlouhou dobu.

Co když program bude při každém průchodu funkcí hlavní smyčkou provádět jen část efektu – rozsvícení jedné LED?

Pak se bude tlačítko testovat častěji a odezva programu bude kratší. Budeme potřebovat více stavů. Upravený stavový diagram bude vypadat takto:



Program se vždy po uplynutí času zpoždění dostává do stavu, kdy svítí další LED. Pokud je ale uvolněno tlačítko, pak přejde do stavu Nesvítí.

Zde je kód upraveného programu. V ukázkovém projektu **response** jej aktivujete nastavením symbolu VERSION na 2: #define VERSION 2

```
// Stavy programu
#define ST_LED1_ON  1
#define ST_LED2_ON  2
#define ST_LED3_ON  3
#define ST_OFF      4
int state = ST_OFF;

int main(void) {
	// inicializace ovladace pinu a delay
	init();

	while (1) {
		if (switch1_read() == SWITCH_PRESSED) {
			// Jen pokud je stisknuto tlacitko a soucasny stav je vypnuto,
			// prejdeme na stav rozsviceni prvni LED, jinak uz nektera LED
			// sviti a stavy se meni ve switch.
			if ( state == ST_OFF )
				state = ST_LED1_ON;
		}
		else
			state = ST_OFF;

		switch (state) {

		case ST_OFF:
			LED_control(false, false, false);
			break;

		case ST_LED1_ON:
			// Bliknout LED1 a prejit na stav dalsi LED2
			LED_control(true, false, false);
			SYSTICK_delay_ms(BLINK_DELAY);
			state = ST_LED2_ON;
			break;

		case ST_LED2_ON:
			LED_control(false, true, false);
			SYSTICK_delay_ms(BLINK_DELAY);
			state = ST_LED3_ON;
			break;

		case ST_LED3_ON:
			LED_control(false, false, true);
			SYSTICK_delay_ms(BLINK_DELAY);
			state = ST_LED1_ON;
			break;
		}	// switch

	}	// while

	/* Never leave main */
	return 0;
}
```

Vysvětlení kódu

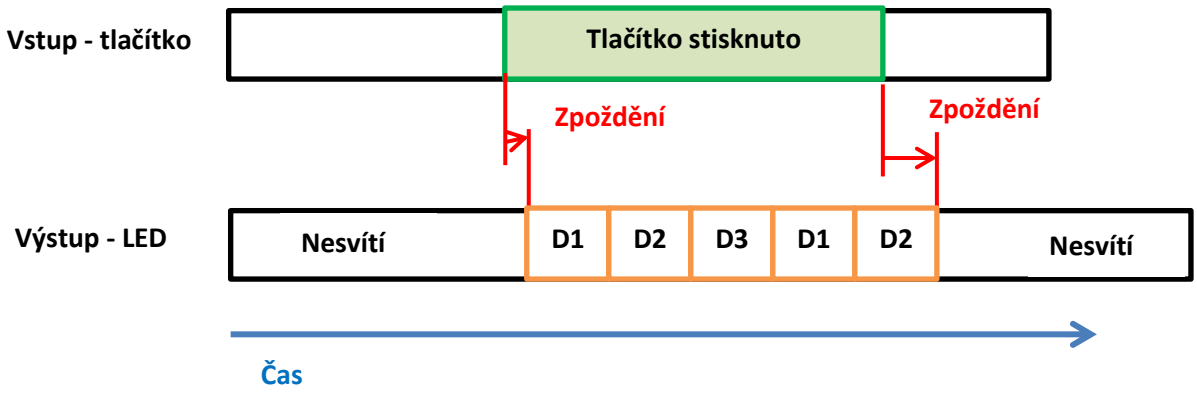
Oproti první verzi jsou přidány další stavy, resp. původní stav pro efekt je rozdělen na 3 stavy pro jednotlivé LED:

- ST_LED1_ON – svítí LED 1
- ST_LED2_ON – svítí LED 2
- ST_LED3_ON – svítí LED 3

V každém z těchto stavů se rozsvítí příslušná LED a pak se program pozastaví na určenou dobu. Pak nastaví proměnnou „state“ na následující stav a tím průchod smyčkou končí. Při novém vykonání smyčky se na začátku znovu testuje tlačítko.

Analýza odezvy programu

Jak rychle reaguje náš upravený program na stisk a uvolnění tlačítka?



Jak je vidět na obrázku, pokud uvolníme tlačítko v okamžiku, kdy svítí D2, pak program zareaguje na konci intervalu, kdy D2 svítí.

Odezva se tedy zlepšila – nemusíme čekat, než doběhne celá sekvence tří LED, program zareaguje po ukončení svitu jedné LED. Ale zpoždění je pořád dost velké. Odpovídá době svitu každé LED. Pokud bychom chtěli, aby každá LED svítila 5 sekund, pak by zpoždění mohlo být také až 5 s.

Jak ještě zlepšit odezvu programu?

Blikač verze 3 - další zlepšení odezvy programu využitím časovače

Abychom ještě zlepšili odezvu programu na uvolnění tlačítka, **musíme se zbavit čekání pomocí softwarového zpoždění** (funkce SYSTICK_delay_ms).

Softwarové zpoždění zastaví program, dokud neuplyne určený čas. Tomuto způsobu čekání se říká „busy-waiting“, protože procesor je v době čekání plně vytížen (busy) právě tímto čekáním.

Takové čekání je plýtváním procesorového času. Procesor jen čeká na uplynutí času, místo aby dělal něco užitečného.

Tento způsob čekání je možné použít v jednoduchých programech, které nic jiného vykonávat nemusí, nebo v případě velmi krátkých čekání (která ani jinak vyřešit nejdou). Jinak bychom ale takové čekání používat neměli.

Jak čekat bez delay?

Mikropočítače obsahují kromě procesoru další obvody, tzv. **periferie.** Tyto periferie mohou namísto procesoru vykonávat určité speciální úkoly.

Jednou takovou periferií je časovač.

Časovač dokáže např. vyvolat určitou událost po uplynutí nastaveného času, měřit délku pulzu na vstupu, generovat pulzy na výstupu apod.

Pracovat přímo s časovačem může být poměrně složité, ale s využitím funkcí ovladače „Systick" můžeme snadno odměřovat čas. K tomu použijeme funkci **SYSTICK_millis()**.

Funkce SYSTICK_millis

SYSTICK_millis vrací počet milisekund, které uplynuly od spuštění našeho programu.

Jak využít millis pro čekání?

Princip je tento:

- Na začátku čekání si uložíme aktuální čas (získaný pomocí SYSTICK_millis()).
- Při každém průchodu přes hlavní smyčku spočítáme, kolik času už uplynulo.
- Pokud už uběhlo více času, než jak dlouho chceme čekat, přejdeme do dalšího stavu.

Poznámka

Při tomto řešení v programu opakovaně zjišťujeme, jestli už uplynul určený čas. Tento princip – dotazování - se nazývá **polling**. Dá se použít pro načasování událostí v programu bez toho, abychom museli program zastavit pomocí delay.

Kód program verze 3

V této verzi programu využijeme místo delay dotazování (polling). V předchozí verzi programu jsme měli pro každou LED jeden stav, např. ST_LED1_ON. V této verzi potřebujeme místo toho pro každou LED dva stavy:

- rozsvícení LED
- čekání na uplynutí času svícení

Ve výsledku tak program bude procházet přes hlavní smyčku velmi rychle, nikde nečeká. A proto bude schopen rychle reagovat na tlačítko.

Zde je nová verze programu. V ukázkovém projektu **response** ji aktivujete nastavením symbolu VERSION na 3 (#define VERSION 3).

```
// Stavy programu
#define ST_LED1_ON    1
#define ST_LED2_ON    2
#define ST_LED3_ON    3
#define ST_OFF        4
#define ST_LED1_WAIT  5
#define ST_LED2_WAIT    6
#define ST_LED3_WAIT    7
int state = ST_OFF;

int main(void) {
```

```
	// inicializace ovladace pinu a delay
	init();

	uint32_t waitStart;		// cas, kdy se rozvitila LED
	uint32_t currentTime;		// aktualni cas, pomocna promenna

	while (1) {
		if (switch1_read() == SWITCH_PRESSED) {
			// Jen pokud je stisknuto tlacitko a soucasny stav je vypnuto,
			// prejdeme na stav rozsviceni prvni LED, jinak uz nektera LED
			// sviti a stavy se meni ve switch.
			if ( state == ST_OFF )
				state = ST_LED1_ON;
		}
		else
			state = ST_OFF;

		switch (state) {

		case ST_OFF:
			LED_control(false, false, false);
			break;

		case ST_LED1_ON:
			// Rozsvitit LED, ulozit aktualni cas a prejit do stavu cekani na
			// uplynuti casu svitu teto LED.
			LED_control(true, false, false);
			waitStart = SYSTICK_millis();
			state = ST_LED1_WAIT;
			break;
		case ST_LED1_WAIT:
			// Kontrola jestli uz ubehlo dost casu abychom rozsvitili dalsi LED
			// a pokud ano, prechod na dalsi stav
			currentTime = SYSTICK_millis();
			if ( currentTime - waitStart >= BLINK_DELAY )
				state = ST_LED2_ON;
			break;


		case ST_LED2_ON:
			LED_control(false, true, false);
			waitStart = SYSTICK_millis();
			state = ST_LED2_WAIT;
			break;
		case ST_LED2_WAIT:
			currentTime = SYSTICK_millis();
			if ( currentTime - waitStart >= BLINK_DELAY )
				state = ST_LED3_ON;
			break;

		case ST_LED3_ON:
			LED_control(false, false, true);
			waitStart = SYSTICK_millis();
			state = ST_LED3_WAIT;
			break;
		case ST_LED3_WAIT:
			currentTime = SYSTICK_millis();
			if ( currentTime - waitStart >= BLINK_DELAY )
				state = ST_LED1_ON;
			break;
		}	// switch

	}	// while
	return 0;
}
```

Vysvětlení kódu

Pro každou LED jsou teď v programu 2 stavy: ST_LED1_ON a ST_LED1_WAIT.

Ve stavu ST_LEDn_ON se jen rozsvítí LED a uloží se aktuální čas získaný pomocí funkce SYSTICK_millis() do proměnné waitStart. Pak se nastaví stav na následující - ST_LEDn_WAIT.

Ve stavu ST_LED1_WAIT se zjistí aktuální čas a odečte se od něj čas, kdy začalo čekání. Tím zjistíme, jak dlouho už čekáme.

Pokud už čekání trvalo požadovanou dobu (BLINK_DELAY), pak přejdeme do dalšího stavu - svítí následující LED. Zde je ukázka – kód pro stav, kdy svítí LED1 a čeká se, až bude čas ji zhasnout a rozsvítit LED2:

```
case ST_LED1_WAIT:
	// Kontrola jestli uz ubehlo dost casu abychom rozsvitili dalsi LED
	// a pokud ano, prechod na dalsi stav
	currentTime = SYSTICK_millis();
	if ( currentTime - waitStart >= BLINK_DELAY )
		state = ST_LED2_ON;
	break;
```

Nejprve je zjištěn aktuální čas a výsledek uložen do proměnné currentTime.

V podmínce if se odečte aktuální čas od času, kdy začalo čekání (uložen v proměnné waitStart při rozsvícení LED).

Pokud je výsledek větší nebo roven požadované době svícení (konstanta BLINK_DELAY), pak se stav nastaví na ST_LED2_ON – tedy při příštím průchodu smyčkou se rozsvítí LED2.

Analýza odezvy programu

Na obrázku níže vidíme, jak teď probíhají činnosti v programu.

Pruh **Čtení tlačítka** ukazuje, kdy probíhá v programu načtení stavu tlačítka.

Pruh **Ovládání LED** znázorňuje kód programu, který se stará o ovládání LED. Modré sloupce představují stav, kdy jsou LED zhasnuty (stav ST_OFF). Oranžové sloupce jsou stavy, kdy svítí LED a čeká se na uplynutí času.

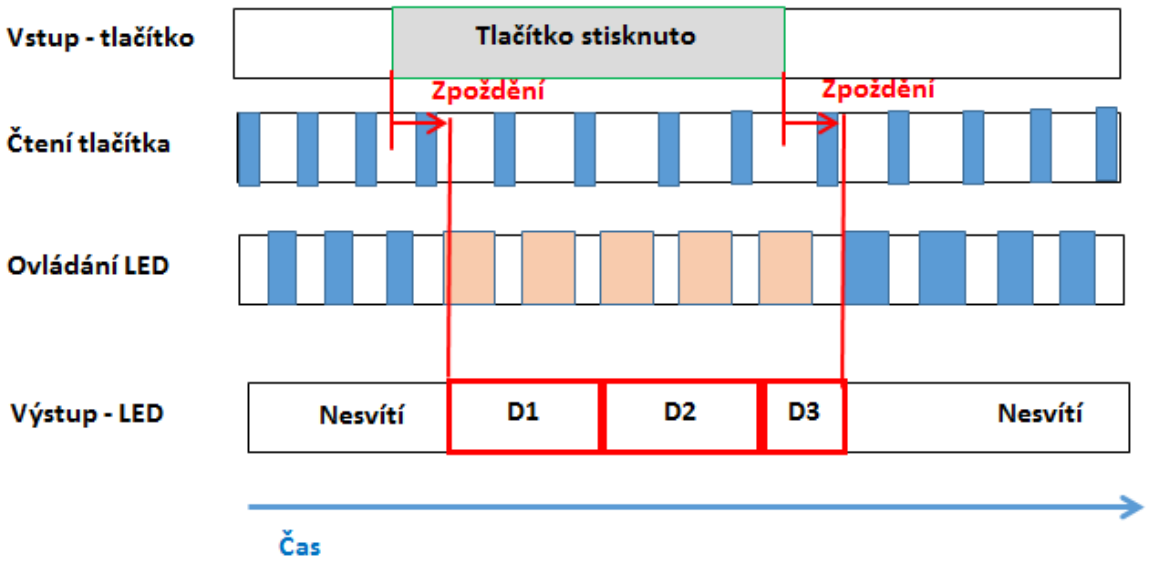
Poslední pruh zobrazuje, která LED právě svítí.

Jak je to se zpožděním reakce programu na tlačítko?

Při stisku tlačítka podle obrázku je program ve stavu ST_OFF. Nějakou dobu trvá, než se dostane znovu ke čtení tlačítka. Pak přejde do stavu, kdy svítí D1 – oranžový pruh.

Při uvolnění tlačítka vzniká zpoždění tím, že program právě nečte tlačítko, ale zabývá se ovládáním LED (a vyhodnocením, zda už uplynul čas pro svit aktuální LED).

Celkově je ale zpoždění velmi krátké, protože jednotlivé stavy programu trvají velmi krátce (na obrázku jsou to úzké obdélníčky v pruzích Čtení tlačítka a Ovládání LED).



Tím jsme vylepšili odezvu programu.

Zkusme ještě **vylepšit přehlednost programu**, aby se snadněji rozšiřoval a upravoval.

Blikač verze 4 - program jako několik úloh

Představte si, že teď když jsme program konečně dokončili, přišel zákazník s novým požadavkem…

Při stisknutém tlačítku má kromě efektu postupného rozsvěcování LED 1 až LED 3 ještě blikat zelená LED a to tak, že se rozsvítí na 200 ms a zhasnuta bude 700 ms – tedy bliká krátkými záblesky.

Jak to uděláme?

Můžeme přidat do příkazu switch další stavy - pro zelenou LED. Ale…

- jak to udělat, aby současně probíhal efekt běžícího světla a blikala zelená LED s jinou periodou?
- I když se nám to podaří, program bude **nepřehledný**. Budeme směřovat ke špagetovému kódu – mícháme efekt běžícího světla s efektem blikání LED, i když jsou to vlastně nesouvisející činnosti.

Pomůžeme si tak, že **rozdělíme program na jednotlivé činnosti,** neboli **úlohy (tasky)**. A pro každou činnost v programu použijeme jednu funkci.

Nejprve upravme program, který už máme, zatím bez blikání zelené LED. Hlavní smyčka bude vypadat takto:

```
while (1) {
    TaskSwitches();
    TaskEffect();
}
```

Ve funkci TaskSwitches je ukryt kód pro čtení tlačítka.

Ve funkci TaskEffect je ukryt kód pro efekt běžícího světla.

Funkce TaskSwitches a TaskEffect si můžeme představit jako nezávislé úlohy (task = úloha). Program tedy vykonává 2 úlohy – čte tlačítka a provádí efekt s LED.

Kód programu jsme skoro nezměnili, jen jsme jej z main() přesunuli do dvou funkcí.

Změnili jsme ale způsob uvažování o programu. Teď o něm **uvažujeme jako o dvou nezávislých úlohách**.

Když se objeví další požadavek, co má program ještě dělat, přidáme prostě další úlohu = funkci do hlavní smyčky.

Jak přidat blikání zelené LED podle požadavku zákazníka? Jednoduše!

```
while (1) {
	TaskSwitches();
	TaskEffect();
	TaskGreenLed();
}	// while
```

Musíme samozřejmě napsat kód funkce TaskGreenLed.

Funkce TaskGreenLed má blikat LED.

Napsat blikání LED je snadné. Rozhodně snadnější, než vymyslet, jak blikání „zabudovat“ do složitého programu s mnoha stavy.

Když uvažuji o programu jako o sadě úloh, pak v každé úloze řeším jen jeden malý úkol.

Pozor, aby to ale fungovalo, **je potřeba, aby se každá úloha provedla v co nejkratším čase.** Pokud se např. jedna úloha bude provádět 3 sekundy (protože do ní vložíme několik delay), pak nebude možné, aby jiná úloha bliknula LED např. dvakrát za sekundu.

Zde je kompletní upravený program se třemi úlohami (tasky). V ukázkovém projektu **response** jej aktivujete nastavením symbolu VERSION na 4 (#define VERSION 4).

```
// Stavy programu
#define ST_LED1_ON		1
#define ST_LED2_ON		2
#define ST_LED3_ON		3
#define ST_OFF			4
#define ST_LED1_WAIT		5
#define ST_LED2_WAIT		6
#define ST_LED3_WAIT		7

// globalni promenna pro stav tlacitka SW1
bool SW1_pressed;
// Promenna state je nove lokalni uvnitr tasku (funkce) pro blikani

// Prototypy funkci
void TaskSwitches(void);
void TaskEffect(void);
void TaskGreenLed(void);
```

```
int main(void) {
	// inicializace ovladace pinu a delay
	init();

	while (1) {

		TaskSwitches();
		TaskEffect();
		TaskGreenLed();

	}	// while

	/* Never leave main */
	return 0;
}

// Uloha, ktera se stara o obsluhu tlacitek
void TaskSwitches(void)
{
	if (switch1_read() == SWITCH_PRESSED)
		SW1_pressed = true;
	else
		SW1_pressed = false;
}


// Uloha, ktera se stara o blikani LED
void TaskEffect(void) {
	// Stav totoho tasku.
	// Promenna je static, aby si uchovala hodnotu mezi volanimi teto funkce,
	// tj. aby nezanikla na konci funkce
	static int state = ST_LED1_ON;

	static uint32_t waitStart;		// cas, kdy se rozvitila LED, musi byt static!
	uint32_t currentTime;			// aktualni cas, pomocna promenna

	// Uloha efekt LED se provadi jen pri stisknutem tlacitku
	if (SW1_pressed) {
		switch (state) {

		case ST_LED1_ON:
			// Rozsvitit LED, ulozit aktualni cas a prejit do stavu cekani na
			// uplynuti casu svitu teto LED.
			LED_control(true, false, false);
			waitStart = SYSTICK_millis();
			state = ST_LED1_WAIT;
			break;

		case ST_LED1_WAIT:
			// Kontrola jestli uz ubehlo dost casu abychom rozsvitili dalsi LED
			// a pokud ano, prechod na dalsi stav
			currentTime = SYSTICK_millis();
			if (currentTime - waitStart >= BLINK_DELAY)
				state = ST_LED2_ON;
			break;

		case ST_LED2_ON:
			LED_control(false, true, false);
			waitStart = SYSTICK_millis();
			state = ST_LED2_WAIT;
			break;

		case ST_LED2_WAIT:
			currentTime = SYSTICK_millis();
			if (currentTime - waitStart >= BLINK_DELAY)
				state = ST_LED3_ON;
```

```
                        break;
                case ST_LED3_ON:
                        LED_control(false, false, true);
                        waitStart = SYSTICK_millis();
                        state = ST_LED3_WAIT;
                        break;
                case ST_LED3_WAIT:
                        currentTime = SYSTICK_millis();
                        if (currentTime - waitStart >= BLINK_DELAY)
                                state = ST_LED1_ON;
                        break;

                }       // switch
        } else {
                // zhasnout LED pokud neni stisknuto tlacitko
                LED_control(false, false, false);
                state = ST_LED1_ON;   // reset stavu tasku
        }

}       // TaskEffect

// Ukazka dalsiho tasku - blika pri stisknutem tlacitku RGB LED
void TaskGreenLed() {
    static enum {
       ST_LED_ON, ST_ON_WAIT, ST_LED_OFF, ST_OFF_WAIT

    } stav = ST_LED_ON;

    static uint32_t startTime;

    // uloha se provadi jen pri stisknutem tlacitku
       if (SW1_pressed) {

                switch (stav) {
                case ST_LED_ON:
                        pinWrite(LED_GREEN, LOW);
                        startTime = SYSTICK_millis();
                        stav = ST_ON_WAIT;
                        break;
                case ST_ON_WAIT:
                        if (SYSTICK_millis() - startTime >= GREEN_ON_DELAY)
                                stav = ST_LED_OFF;
                        break;
                case ST_LED_OFF:
                        pinWrite(LED_GREEN, HIGH);
                        startTime = SYSTICK_millis();
                        stav = ST_OFF_WAIT;
                        break;
                case ST_OFF_WAIT:
                        if (SYSTICK_millis() - startTime >= GREEN_OFF_DELAY)
                                stav = ST_LED_ON;
                        break;
                }   // switch
        } else {
                pinWrite(LED_GREEN, HIGH);    // zhasni LED
                stav = ST_LED_ON;     // resetuj stav LED
        }
}
```

Co jsme v program změnili?

1) Vytvořili jsme proměnnou „stisknuto“ (bool SW1_pressed).
Tuto proměnnou nastavuje úloha TaskSwitches na true, pokud je stisknuto tlačítko. Informaci o stisku tlačítka potřebují obě další úlohy – efekt i blikání.

2) Proměnou „stav“ jsme nově definovali uvnitř funkce TaskEffect, protože je to stav této jedné úlohy, ne už celého programu. Proměnná je „static“, to znamená, že si zachová hodnotu, i když funkce skončí.

3) V nové úloze TaskGreenLed jsme zavedli její vlastní proměnnou „stav“. Tato proměnná je definována jako výčtový typ (enum).

```
static enum {
      ST_LED_ON, ST_ON_WAIT, ST_LED_OFF, ST_OFF_WAIT } stav = ST_LED_ON;
```

Použití enum je lepší než použití **int** a stavů definovaných pomocí direktivy #define, protože je lépe čitelné a překladač může pohlídat, aby do proměnné nebyla přiřazena nesmyslná hodnota.

Plánovač

V poslední verzi programu jsme vlastně vytvořili vlastní jednoduchý **operační systém**.

Používáme v něm 3 úlohy (tasky).

Funkce main(), která postupně spouští jednotlivé úlohy je zde v roli jádra operačního systému – tzv. plánovače. Plánovač (scheduler) je součást operačního systému, která zajišťuje přepínání úloh na procesoru.

Náš plánovač je velmi jednoduchý, prostě úlohy volá postupně jednu po druhé. Když první úloha skončí, spustí se druhá, atd.

Je to tzv. **nepreemptivní plánovač** (kooperativní plánovač).

Takový plánovač nemůže přerušit (preempt) běžící úlohu. Musí spoléhat na to, že úloha sama co nejrychleji skončí a uvolní tak procesor pro další úlohy. Tedy aby systém fungoval, musejí úlohy spolupracovat (kooperovat).

Rutiny (funkce)

Funkce (nebo také rutina, procedura, metoda, podprogram) je samostatná část kódu, kterou voláme za jediným účelem.

Podle [4] je funkce (rutina) největším vynálezem počítačové vědy kromě vynálezu samotného počítače. Díky rutině je program čitelnější, srozumitelnější. (viz [4], kapitola 7 – Vysoce kvalitní rutiny).

Pro vytváření rutin existuje mnoho dobrých důvodů. Uveďme alespoň několik základních podle [4]:

- **Omezení složitosti**. Rutina ukrývá informace, o kterých nebudeme chtít na vyšších úrovních programu uvažovat. Samozřejmě o nich musíme uvažovat, když rutinu vytváříme, ale pak na tyto informace můžeme zapomenout a používat rutinu bez znalosti toho, co dělá uvnitř – jako „černou skříňku“.

- **Vytvoření srozumitelné abstrakce**. Umístění kódu do dobře pojmenované rutiny je nejlepší způsob dokumentace kódu. Místo nějaké posloupnosti příkazů bude v kódu pouze volání funkce, což je kratší a srozumitelnější.
- **Vyhnutí se duplicitnímu kódu**. Pokud se stejný kód vyskytuje na několika místech programu, je vhodné jej přesunout do samostatné rutiny. Usnadní se tím úpravy a opravy kódu a ušetří se paměť pro program.
- **Zlepšení přenositelnosti**. Nepřenositelné vlastnosti programu izolujte do rutin – např. přístup na hardware. Pak při přenosu na jiný hardware stačí nahradit příslušné rutiny a zbytek programu zůstane beze změn.

Inline funkce

Inline funkce (vložené rutiny) jsou z pohledu programátora funkce, pracuje s nimi jako s funkcemi, ovšem překladač při překladu místo volání funkce vloží přímo kód dané funkce.

Důvodem pro použití inline funkcí je zvýšení výkonu (rychlosti provádění programu). Nevýhodou je větší velikost výsledného kódu, protože místo volání jedné „sdílené“ kopie kódu vznikne vlastní kopie kódu v každém místě volání. Proto také inline funkce mají smysl pouze pro krátké úseky kódu a při požadavku na skutečně velmi rychlé provádění. U programů pro mikropočítače se s nimi setkáme relativně často, ale přesto bychom je měli používat jen výjimečně. Kniha [4] doporučuje změřit zlepšení výkonu kódu s inline rutinami ve srovnání s použitím normálních volaných rutin a podle toho se rozhodnout, zda se jejich použití vyplatí. Pokud vám připadá úsilí potřebné k takovému testování zbytečné, pak zřejmě není vyšší výkon tak kritickým požadavkem, jak si myslíte, a je zbytečné inline funkce nasazovat.

Doporučený postup práce a náměty k procvičení

1. Vyzkoušejte si postupně jednotlivé varianty programu. V ukázkovém projektu response stačí změnit na začátku hodnotu symbolu VERSION na řádku:
   #define VERSION 1
   U každé verze programu si vyzkoušejte odezvu programu a srovnejte svoje pozorování s časovými digramy uvedenými v návodu.

2. Vytvořte program, který bude blikat třemi LED, každá s jinou frekvencí.
   - LED 1 bude blikat s periodou 1 s, se střídou 50% tj. 0,5 s svítí a 0,5 s nesvítí.
   - LED 2 bude blikat tak, že 200 ms svítí a 650 ms nesvítí.
   - LED 3 bude blikat tak, že 800 ms svítí a 300 ms nesvítí.

   Blikání se bude spouštět a zastavovat stiskem tlačítka. Po stisku tlačítka LED začnou blikat a budou blikat i po jeho uvolnění. Při dalším stisku tlačítka LED zhasnou, po dalším stisku se znovu rozsvítí, atd.
   Program vyřešte tak, aby reagoval na stisk tlačítka bez zpoždění a byl rozdělen na jednotlivé úlohy (tasky), jak je ukázáno v programu Blikač v tomto návodu.

Kontrolní otázky

1. Vysvětlete pojem odezva programu.

2. Jaká je nevýhoda použití „busy waiting“ (funkce delay) pro čekání?
3. Pokud ráno spustíte svůj program v mikropočítači a večer se podíváte na hodnotu vracenou funkcí SYSTICK_millis(), jakou přibližně hodnotu můžete očekávat?
4. Vysvětlete důvody pro používání funkcí v programu, jaké jsou hlavní přínosy?
5. Vysvětlete význam klíčového slova inline a static u funkce. Jaký je rozdíl mezi inline a „obyčejnou“ funkcí?

Zdroje a doporučená literatura

[1] PINKER, Jiří. Mikroprocesory a mikropočítače, Praha: BEN - technická literatura, 2004, 159 s. ISBN 80-730-0110-1.
[2] CATSOULIS J.: Designing Embedded Hardware, O'Reilly 2005.
[3] BARR M., MASSA A.: Programming Embedded Systems (with C and GNU development tools). O'Reilly, 2006.
[4] MCCONNELL, Steve. Dokonalý kód: umění programování a techniky tvorby software. Vyd. 1. Brno: Computer Press, 2005, 894 s. ISBN 80-251-0849-X. Případně originál: Code Complete, 2nd edition. (případně anglický originál, Code Complete, 2nd editon).
[5] PONT, M. J.: *Embedded C*. Addison-Wesley, 2002 a Kurz Programming embedded systems stejného autora dostupný online (ftp://ftp.ti.com/pub/data_acquisition/MSC_CD-ROM/C_Programming/PrgEmbeddeSys_1perPage.pdf).
[6] DEAN, Alexander G. Embedded Systems Fundamentals with ARM Cortex-M based Microcontrollers: A Practical Approach. ARM Education Media, ISBN 978-1911531036.