Mikroradiče

Jazyk C pre mikroradiče

Príkaz goto, smerníky, polia, reťazce, definície registrov,

poznámky k tvorbe zdrojového kódu

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY
ÚSTAV ELEKTRONIKY A FOTONIKY
Laboratórium DSP a mikroradičov

:: príkaz goto

- príkaz `goto` umožňuje nepodmienený skok v prípade, kedy je potrebné v toku programu vynechať istú časť kódu (v rámci jednej funkcie – prečo?)

- využíva **návestia** – pomenované miesta v kóde nasledované znakom ':'

```
start :
{
   if (podmienka)
          goto vonkajsi_blok ;
          /* dalsi zdrojovy kod */
   goto start ;
}
vonkajsi_blok :
/* prikazy vonkajsieho bloku */
```

V roku 1968 Edsger Dijkstra publikoval známy článok

Go To Statement Considered Harmful [1]

Kritizoval v ňom časté používanie príkazu `goto` v programovacích jazykoch a obhajoval používanie štruktúrovaného programovania:

- časté používanie príkazu `goto` vedie k tzv. špagetovému kódu
- používanie príkazu `goto` vedie k ťažšie čitateľnému kódu a náročnejšiemu odlaďovaniu
- akýkoľvek zdrojový kód, ktorý používa `goto` môžeme prepísať bez jeho použitia

[1] *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148. Copyright © 1968, Association for Computing Machinery, Inc.

:: príkaz goto

- jazyky ako C++ a Java poskytujú mechanizmy umožňujúce zotavenie sa z chybových stavov, jazyk C takouto možnosťou nedisponuje

- v jazyku C príkaz `goto` poskytuje vhodný spôsob ako opustiť zahniezdené bloky

```
for (..)
{
   for (..)
   {
         if (priznak_chyby)
               goto chyba;
               /* preskocime dva bloky */
   }
}
chyba :
```

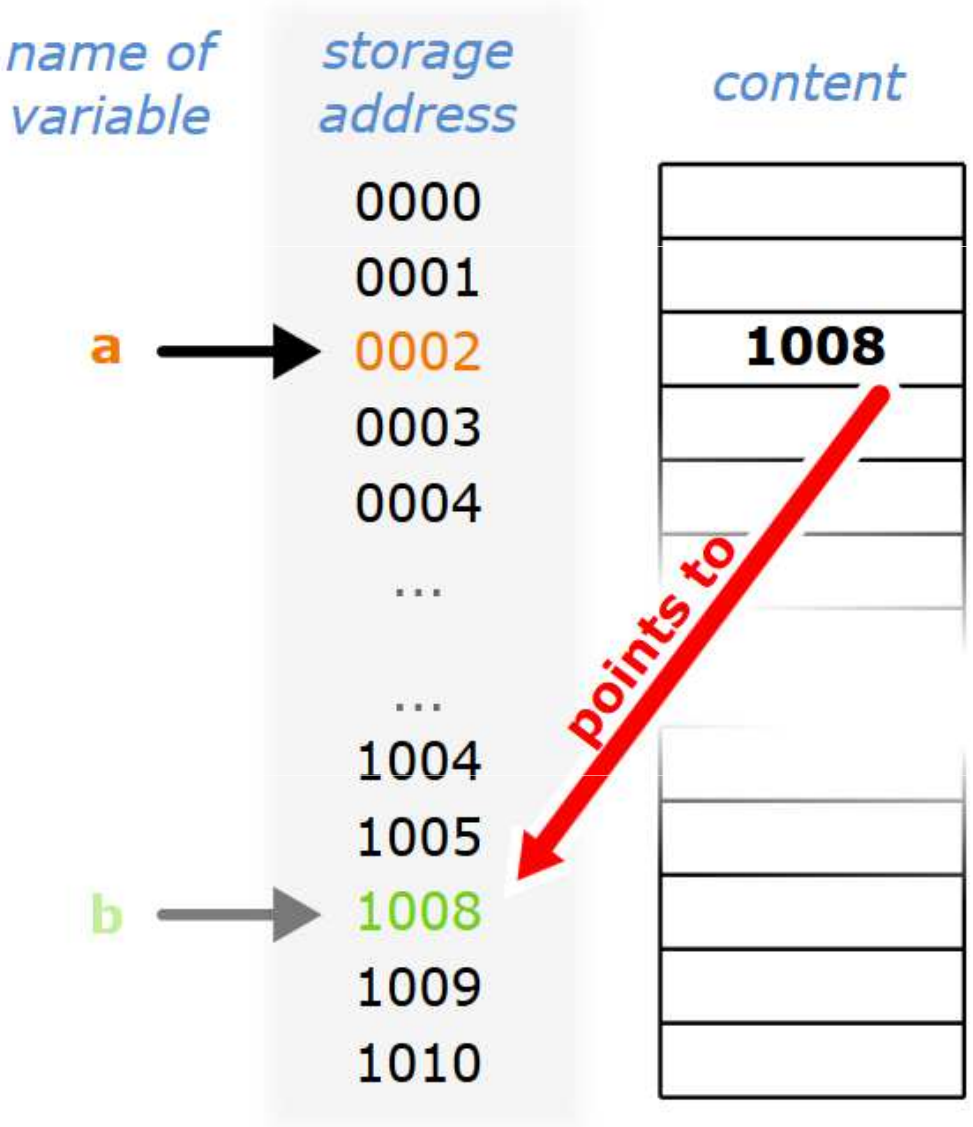
:: Opustenie cyklu bez použitia príkazu goto

```
priznak_pokracovania = 1;
for (..)
{
   for (init; priznak_pokracovania; iteracia)
   {
         if (priznak_chyby)
         {
               priznak_pokracovania = 0;
               break;
         }
         /* vnutorna slucka */
   }
   if (!priznak_pokracovania)
               break;
         /* vonkajsia slucka */
}
```

:: Smerníky – úvodné poznámky

- smerník – adresa premennej v pamäti
- pomocou adresy môžeme pristupovať k premennej alebo ju modifikovať odkiaľkoľvek z priestoru všetkých pamäťových adries
- smerníky výrazne zvyšujú efektivitu zdrojového kódu najmä v prípade dátových štruktúr
- pozn. k terminológii
  - stack: časť pamäte, v ktorej sú umiestnené lokálne premenné
  - heap: časť pamäte určená pre dynamickú alokáciu premenných

:: Smerníky – pohľad do pamäťového priestoru



:: Adresy dátových objektov

- každá premenná nachádzajúca sa v pamäti má svoju adresu, niektoré dátové objekty adresu nemajú:
  - konštanty / literály / definície preprocesora
  - výrazy (ak výsledkom nie je premenná)

Definícia konštanty pomocou direktívy preprocesora `#define`

```
#define  DLZKA 10
#define  NOVY_RIADOK '\n‘
```

Definícia konštanty pomocou kvalifikátora `const`

```
const int SIRKA = 5;
```

:: Smerníky – adresa premennej

- adresa premennej – unárny & operátor

```
int n= 4;
p_n = &n; /* adresa celeho cisla n */
```

:: Smerníky – operátor nepriameho prístupu

- prístup/modifikácia adresovanej premennej:

  unárny dereferenčný operátor (operátor nepriameho prístupu) *

```
int n = 4;
double pi = 3.14159;
int *p_n = &n; /* adresa celeho cisla n */
double *p_pi = &pi; /* adresa premennej pi */

*p_pi = *p_pi + *p_n ; /* pi bude 7.14159 */
```

- adresa premennej typu t má typ t*

- s dereferencovaným smerníkom pracujeme ako s každou inou premennou

:: Smerníky – inicializácia

- jazyk C neinicializuje premenné automaticky, je teda dôležité kontrolovať, aby adresa, na ktorú smerník ukazuje bola platná
- smerník je vhodné na začiatku inicializovať na tzv. NULL hodnotu

```
int *p_n = NULL;
```

- pozor na dereferencovanie smerníka s null hodnotou!

:: Smerníky – príklad

```
int a = 5;
int *p_n = NULL;
```

Adresa	Obsah
0x8130 (a)	0x00000005
0x8134 (p_n)	0x00000000

```
p_n = &a;
```

Adresa	Obsah
0x8130 (a)	0x00000005
0x8134 (p_n)	0x00008130

```
*p_n = 8;
```

Adresa	Obsah
0x8130 (a)	0x00000008
0x8134 (p_n)	0x00008130

:: Smerníky – pretypovanie

- v jazyku C je možné explicitne pretypovať akýkoľvek typ smerníka na akýkoľvek iný typ:

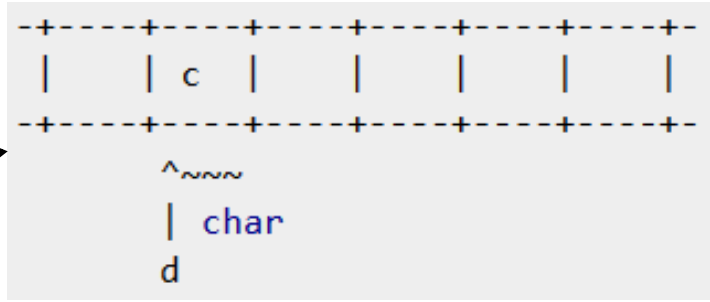
```
int *p_n = NULL;
p_m = (double *)p_n; /* pn bol povodne typu (int *) */
```

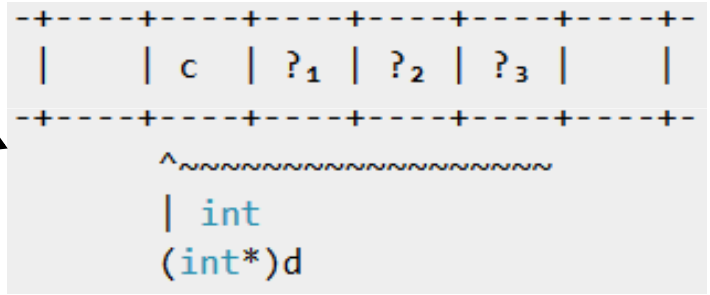
- dereferencovaný smerník má nový typ bez ohľadu na skutočný typ dát

- môže to viesť k segmentačným chybám, ktoré sú ťažko identifikovateľné

:: Smerníky – pretypovanie (príklad)

```
#include <stdio.h>

int vypis(int *);

void main(void){
      char c = '5';
      char *d = &c;

      vypis((int *)d);
      return 0;
}

int vypis(int *n){
    printf("%d\n", *n);
    return *n;
}
```





- v jazyku C nemajú volané funkcie možnosť priamo ovplyvňovať premenné vo volajúcich funkciách, pretože argumenty sú predávané hodnotou

- typický príklad: funkcia, ktorá vymení medzi sebou hodnoty dvoch premenných

```
void vymena(int x, int y) {
   int pomocna = x;
   x = y;
   y = pomocna ;
}
```

- x a y sú lokálne premenné funkcie vymena()!

- aké máme možnosti predávania hodnôt?

- verzia funkcie vymena() s použitím smerníkov

```
void vymena(int *p_x, int *p_y) {
   int pomocna = *p_x;
   *p_x = *p_y;
   *p_y = pomocna ;
}
```

- ak teraz zavoláme funkciu vymena():

```
int a = 5, b = 7;
vymena(&a, &b);
/* teraz a = 7, b = 5 */
```

:: Smerníky a polia

- jednoduché polia sú v C implementované pomocou smerníka na súvislý blok pamäte
- pole 8 prvkov typu int

```
int pole[8];
```

- k prvkom poľa môžeme pristupovať priamo

```
int a = pole[0];
```

- premenná `pole` je v podstate smerník na 0-tý prvok poľa

```
int *p_p = pole; ⇔ int *p_p = &pole[0];
```

:: Smerníky a polia

- predpokladajme, že

```
int *p_p = pole;
```

- k smerníku môžeme pripočítať alebo od neho odpočítať int:

`p_p + i` ukazuje na `pole[i]`

- adresa je inkrementovaná o `i` x veľkosť prvku
- ak `pole` je typu `int` a napr. `pole[0]` má adresu 100, potom `pole[3]` má adresu 112
- príklad: v prípade MSP430 je `sizeof(char) = 1` a `sizeof(int) = 2`

```
char *p_c = (char *)p_p;
```

aká hodnota `i` vyhovie podmienke: `(int *)(p_c + i) == p_p + 3`?

:: Reťazce a polia

- reťazce sú v podstate polia znakov ukončené znakom ’\0’

```
char retazec[] = "Toto je retazec.";
char * pc = retazec;
```

- s reťazcom môžeme potom manipulovať ako s poľom

```
char pomocna;
pomocna = *(pc+10);
```

- aká bude hodnota premennej `pomocna`?

:: Kopírovanie reťazcov

- verzia s využitím polí

```
void kopiruj_retazec(char *r, char *s)
{
   int i = 0;
   while((r[i] = s[i]) != '\0')
     i++;
}
```

- verzia so smerníkmi č. 1

```
void kopiruj_retazec(char *r, char *s)
{
   while((*r = *s) != '\0')
     r++;
     s++;
}
```

:: Kopírovanie reťazcov

- verzia so smerníkmi č. 2

```
void kopiruj_retazec(char *r, char *s)
{
   while((*r++ = *s++) != '\0')
     ;
}
```

:: Kopírovanie reťazcov

- verzia so smerníkmi č. 3

```
void kopiruj_retazec(char *r, char *s)
{
   while(*r++ = *s++)
     ;
}
```

:: Poznámky k deklaráciám – `const` a `volatile`

- kvalifikátory `const` a `volatile` sú kritické v prípade špeciálnych funkčných registrov - ich adresy sú konštantné, ale ich obsah je často premenlivý

- `const`: znamená, že hodnota by nemala byť modifikovaná, pretože ju programátor považuje za konštantu

- `volatile`: znamená, že premenná sa môže spontánne zmeniť bez priamej akcie zo strany programu, t.j. kompilátor nemusí držať kópiu premennej v registri kvôli efektivite ani nemôže predpokladať, že premenná zostane konštantná, keď vykonáva optimalizáciu štruktúry kódu

:: Definícia periférneho registra v jazyku C

- ak chceme priamo zadefinovať adresu periférneho registra, máme v jazyku C problém, pretože adresu premennej nie je možné zadefinovať:

```
// takto zadefinovať adresu P2OUT nepôjde
unsigned char &P2OUT = 0x0029;
```

- môžeme sa pokúsiť zadefinovať aspoň smerník na register

```
// smerník na P2OUT
unsigned char * p_P2OUT = (unsigned char *) 0x0029;
```

  - *prečo nie je táto definícia vhodná?*

:: Definícia periférneho registra v jazyku C

- definíciu môžeme vylepšiť použitím kvalifikátora `const`

```
// konstantny smernik na P2OUT
unsigned char * const p_P2OUT = (unsigned char *) 0x0029;

const unsigned char * p_P2OUT = (unsigned char *) 0x0029;
```

  - *aký bude vplyv kvalifikátora* `const` *v uvedených dvoch prípadoch?*
  - *ktorá z definícií je správna?*

- takýto spôsob definície je vhodný pre periférne registre typu P2OUT, ktoré je možné čítať, je možné do nich zapisovať a sú úplne pod kontrolou programu

- ako postupovať v prípade registrov, ktoré môžu byť modifikované hardvérovými alebo externými udalosťami (v MSP430 napr. register TACTL)?

:: Definícia periférneho registra v jazyku C

- definícia musí obsahovať kvalifikátor `volatile`, aby sme upozornili kompilátor, že hodnota v registri sa môže zmeniť počas udalosti, ktorá nie je pod kontrolou programu

```
volatile unsigned int * const p_TACTL = (unsigned int *) 0x0160;
```

- mnohé registre majú definície vyššie uvedeného typu, ale sú také, ktoré potrebujú ešte podrobnejšiu definíciu, napr. vstupné registre portov (napr. P2IN) nie sú iba `volatile`, ale sú tiež určené iba na čítanie

- zabrániť programu zapisovať do týchto registrov môžeme pridaním ďalšieho kvalifikátora `const`:

```
const volatile unsigned char * const p_P2IN = (unsigned char *) 0x0028;
```

  - v definícii vidíme dvakrát `const`, čo to znamená
  - tiež vidíme vedľa seba `const` aj `volatile`, je to v poriadku?

:: Definícia periférneho registra v jazyku C

- definícia register môžeme aj definovaním symbolickej konštanty:

```
#define p_P2IN ((const volatile unsigned char *) 0x028)
```

- môžeme deklarovať aj meno registra priamo:

```
#define P2IN (*((const volatile unsigned char *) 0x028))
```

:: Bitové polia

Definícia

```
__no_init volatile union {
    unsigned short TACTL; // Timer_A Control
    struct {
        unsigned short TAIFG : 1;          // Timer_A counter interrupt flag
        unsigned short TAIE : 1;           // Timer_A counter interrupt enable
        unsigned short TACLR : 1;          // Timer_A counter clear
        unsigned short : 1;
        unsigned short TAMC : 2;           // Timer_A mode control
        unsigned short TAID : 2;           // Timer_A clock input divider
        unsigned short TASSEL : 2;         // Timer_A clock source select
        unsigned short : 6;
    } TACTL_bit;
} @ 0x0160;
```

Nastavenie bitu: `TACTL_bit.TAIFG = 1`
Zmazanie bitu: `TACTL_bit.TAIFG = 0`
Zmena hodnoty bitu: `TACTL_bit.TAIFG ^= 1`

Priama modifikácia registra: `TACTL = MC_2 | ID_3 | TASSEL_2 | TACLR;`

:: Poznámky k tvorbe softvéru

- architektúra zdrojového kódu moderných mikroradičov je obvykle založená na prerušeniach (interrupt driven), t.j. procesor sa nachádza väčšinu času vo zvolenom nízkopríkonovom režime a v aktívnom režime sa nachádza iba pri obsluhe jednotlivých udalostí, ktoré spúšťajú prerušenia

- vieme, že v prípade MSP430 sa bity, ktorými volíme nízkopríkonový režim nachádzajú v stavovom registri SR

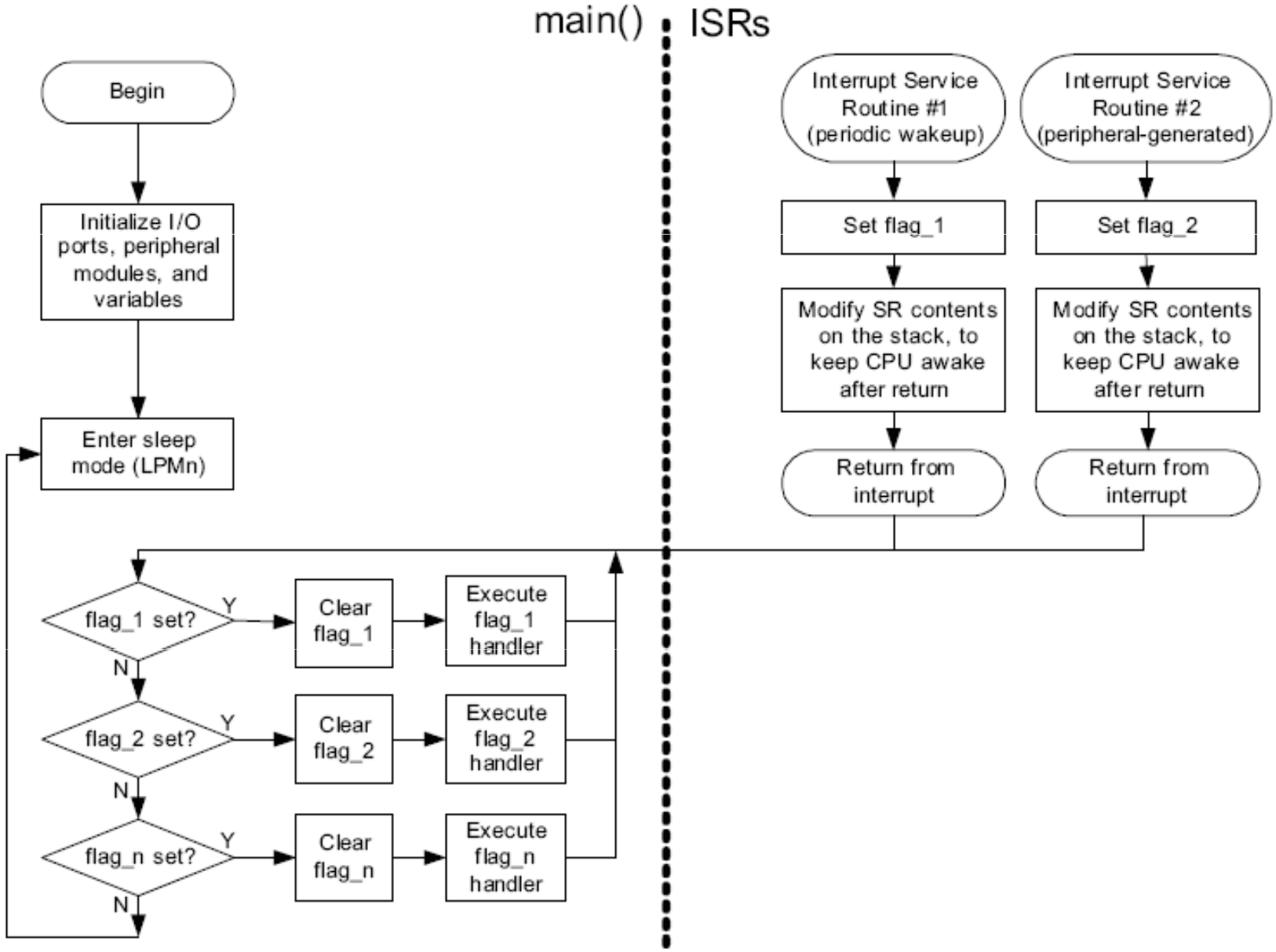
- výhoda tohto usporiadania spočíva v jednoduchom fakte: ak nastane prerušenie, odloží sa aktuálna hodnota SR a teda aj nastavený nízkopríkonový režim do zásobníka, stavový register SR sa vymaže a tým prejde procesor do aktívneho režimu, počas ktorého obslúži prerušenie

- po ukončení ISR dôjde k obnoveniu pôvodného obsahu registra SR a tým aj k opätovnému prechodu do zvoleného nízkopríkonového režimu

- tento tok programu však môže programátor ľahko zmeniť, ak počas vykonávania ISR zmení hodnotu registra SR, ktorá je odložená v zásobníku

- to znamená, že programátor môže zvoliť dve metódy spracovávania udalostí:
  - buď napíše zdrojový kód pre úplnú obsluhu udalosti priamo do ISR
  - alebo v ISR iba nastaví príznak udalosti a zmení odloženenú hodnotu SR tak, aby procesor po ukončení zostal v aktívnom režime a v rámci funkcie main() vykonal po otestovaní príznakov jednotlivých udalostí obsluhu udalosti, ktorá nastala

:: Architektúra zdrojového kódu MSP430



:: Poznámky k tvorbe softvéru

- pravidlo: first thing first
  - *jednou z prvých vecí po štarte programu by malo byť nastavenie WDT, ktorý je po resete aktívny a ak nezmeníme jeho nastavenie, resp. ak ho nedeaktivujeme, dôjde po jeho expirácii k resetu a následne k vytvoreniu nekonečnej slučky resetov*
  - *ďalšou dôležitou vecou je korektné nastavenie jednotlivých oscilátorov a celého systému generovania hodinových signálov*
- pravidlo: používame štandardné definície z hlavičkových súborov dodaných výrobcom procesora
  - *hlavičkové súbory obsahujú prehľadne usporiadané konštanty pre všetky registre a bity zvoleného procesora, pričom názvy registrov a bitov zodpovedajú názvom, ktoré sú uvedené v technickej špecifikácii procesora, čo výrazne uľahčuje písanie zdrojového kódu a zlepšuje jeho čitateľnosť*

- pravidlo: využívame intrinzické funkcie implementované v jazyku C pre daný procesor
  - *poskytujú možnosť vykonať niektoré nastavenia a kritické úlohy veľmi efektívne*
  - *najjednoduchším príkladom kritickej úlohy, ktorú môžeme vykonať efektívne s použitím intrinzickej funkcie je vstup/výstup do/z nízkopríkonového režimu*

```
_bis_SR_register(LPM3_bits + GIE);
```

  - *alebo ak potrebujeme modifikovať hodnotu bitov registra SR uloženú v zásobníku tak, aby zostal procesor po ukončení ISR v aktívnom stave:*

```
_bic_SR_register_on_exit(CPUOFF);
```

- pravidlo: využívame intrinzické funkcie implementované v jazyku C pre daný procesor
  - *môžeme ich využiť ak potrebujeme v zdrojovom kóde vytvoriť časove oneskorenie, ktoré bude trvať presne daný počet cyklov procesora (parameter definujúci počet cyklov musí byť konštanta v čase prekladu):*

```
_delay_cycles(unsigned long);
```

  - *alebo ak potrebujeme povoliť alebo zakázať prerušenia:*

```
_enable_interrupts(void);
_disable_interrupts(void);
```

- pravidlo: využívame nízkoúrovňové inicializačné funkcie (Low-level Initailization Function)
  - *keď kompilátor prekladá zdrojový kód z jazyka C a generuje asembler, vytvorí na začiatku kód, ktorý inicializuje všetky deklarované pamäťové miesta a napĺňa ich požadovanými hodnotami. Tento kód je umiestnený pred prvou inštrukciou funkcie main().*
  - *ak je množstvo deklarovanej pamäte veľké (množstvo premenných, dlhé polia a pod.) môže tento postup predstavovať problém z hľadiska watchdogu*
  - *čas potrebný na inicializáciu dlhého zoznamu premenných môže byť totiž taký dlhý, že dôjde k expirácii časovača WDT ešte pred vykonaním prvej inštrukcie funkcie main()*

- pravidlo: využívame nízkoúrovňové inicializačné funkcie (Low-level Initailization Function)
  - *tzn. že nikdy nedôjde k vykonaniu zdrojového kódu, ktorý inicializuje WDT (obvykle umiestnený na začiatku funkcie main) následkom čoho vznikne nekonečná slučka, v ktorej sa bude procesor neustále resetovať*
  - *pozn.: z hľadiska rýchlosti počítadla WDT sa tento problém týka iba procesorov MSP430, ktoré disponujú RAM pamäťou väčšou ako 2kB*
  - najjednoduchší spôsob ako zabrániť vzniku tohto problému je použiť direktívu kompilátora, ktorá zakáže inicializáciu pamäťových elementov, ktoré nie je potrebné inicializovať, napr.:

```
__no_init int x_array[2500];
```

- pravidlo: využívame nízkoúrovňové inicializačné funkcie (Low-level Initailization Function)
  - *ak takáto direktíva nie je v danom vývojovom prostredí implementovaná, je možné využiť **nízkoúrovňovú inicializačnú funkciu**, ktorá zabezpečuje, že jej telo bude vykonané pred samotnou inicializačnou časťou zdrojového kódu, ktorú vkladá kompilátor*
  - *tým zabezpečíme, že incializácia WDT prebehne ešte pred inicializáciou premenných, napr.:*

```
void __low_level_init(void)
{
        WDTCTL = WDTPW+WDTHOLD;
}
```

:: Príklad použitia nízkoúrovňovej inicializačnej funkcie

```
#include <msp430x16x.h>

int x_array[2500];                          // inicializacia pola zaberie viac ako 32ms.  32ms je expiracna perioda WDT!
unsigned int i,x;

void main(void)
{
//WDTCTL = WDTPW + WDTHOLD;                 // ak umiestnime zakazanie WDT sem namiesto do funkcie __low_level_init()
                                            // dojde k vytvoreniu nekonecnej resetovacej slucky

  P1DIR = 0x01;                             // konfiguracia vystupu pre LED
  x_array[0] = 1;                           // ak pole nie je v aplikacii pouzite, kompilator by ho nevytvoril

  // nasledovna slucka zabezpecuje blikanie LED, najskor rychlo potom coraz pomalsie a cyklus sa opakuje
  while(1)
  {
    P1OUT ^= 0x01;                                      // zmenime stav na LED
    TACCR1=TACCR1+400;                                  // predlzime oneskorenie
    if(TACCR1>18000) TACCR1=0;                          // ak je blikanie uz prilis pomale, zacneme znova
    TACCTL1 = CCIE;                                     // komparacny rezim, povolene prerusenie
    TACTL = TASSEL_2 + TACLR + +ID_3 + MC_2;            // SMCLK/4, zmazanie TA, rezim kontinualneho pocitania
    _bis_SR_register(LPM0_bits+GIE);                    // vstup do rezimu LPM0, cakame na prerusenie CCR1
   }
}

// telo nizkourovnovej funkcie bude vykonane pred inicializaciou pamate
void __low_level_init(void)
{
  WDTCTL = WDTPW + WDTHOLD;                             // zastavenie casovaca WDT
}

// obsluzna rutina prerusenia casovaca A1
#pragma vector = TIMERA1_VECTOR
__interrupt void Timer_A(void)
{
   _bic_SR_register_on_exit(LPM0_bits);                 // po ukonceni ISR zostane procesor v aktivnom stave
   CCTL1 &= ~CCIFG;                                     // zmazeme priznak prerusenia CCR1
}
```

:: Užitočné referencie

1. Kernighan B., Ritchie D.: Programovací jazyk C, Computer Press, 2006, ISBN: 80-251-0897-X

2. Herout, P.: Učebnice jazyka C, 4. vydání, Kopp, 2004,ISBN 80-7232-220-6

3. Texas Instruments, Inc.: MSP430 Optimizing C/C++ Compiler v.4.2 User's Guide (Rev. H)

Koniec prednášky

Jazyk C pre mikroradiče

Príkaz goto, smerníky, polia, reťazce, definície registrov,

poznámky k tvorbe zdrojového kódu