Středoškolská technika 2009

Setkání a prezentace prací středoškolských studentů na ČVUT

Učebnice jazyka C pro mikroprocesory dsPIC30F3013

István Módos

SŘEDNÍ PRŮMYSLOVÁ ŠKOLA SDĚLOVACÍ TECHNIKY
Praha1, Panská 3

Obsah

1. Úvod 4
2. První program 5
Založení nového projektu a kompilace kódu 6
Naprogramování procesoru 6
3.1 Datové typy 8
Celá čísla 8
Desetinná čísla 8
Ostatní datové typy 8
Optimalizace 9
3.2 Konstanty 9
Definice proměnných 10
4.1 Přiřazovací příkaz 11
4.2 Operátory 11
Priorita operátorů a jejich asociativita 15
5.1 Základní pojmy 16
Identifikátor 16
Klíčová slova 16
Komentáře 16
Hlavičkové soubory 16
5.2 Obecná struktura kódu 17
6. Cykly 19
Příkazy break a continue 21
7. If a else 23
8.1 Switch 25
8.2 Goto a return 26
Goto 26
Return 26
9. Preprocesor 27
Obecné schéma zpracování kódu 27
Makra bez parametrů 28
Makra s parametry 29
Podmíněný překlad 30
10.1 Paměťový model – pohled dsPICa 31
Programová paměť 31
Datová paměť 33
10.2 Paměťový model – pohled Céčka 35
Oblast platnosti identifikátorů 37
Modifikátory paměťových tříd 38
11.1 Funkce 40
Datový typ Void 42
Definice funkce 43
Rekurze 44
Proměnný počet parametrů 44
Standardní knihovny 44
11.2 Přerušení 44
Priorita přerušení 45

Vnořená přerušení ... 45
12. Pointery ... 49
Pointerová aritmetika ... 50
Konverze pointerů ... 52
Pointer a funkce ... 52
Dynamické přidělení paměti ... 53
13.1 Pole ... 59
Inicializace pole ... 60
Operace s polem ... 60
Pole jako parametr funkce ... 61
Dynamické pole ... 64
13.2 Řetězce ... 64
Řetězec jako parametr funkce ... 65
13.3 Vícerozměrná pole ... 66
Inicializace vícerozměrného pole ... 68
Pole řetězců ... 68
14.1 Operátor typedef ... 69
14.2 Struktury ... 69
Přístup k jednotlivým prvkům struktury ... 70
Inicializace struktury ... 71
Pointer na strukturu ... 71
Funkce a struktury ... 72
Bitové struktury ... 73
15.1 Výčtový typ ... 75
15.2 Union ... 75
16. Závěr a doporučená literatura ... 77

•1. Úvod

Předtím, než se dozvíte, co všechno v této učebnici najdete, si zodpovězme otázku, proč jsem se vlastně do těchto skript pustil. Důvodem pro jejich vznik je fakt, že v českém prostředí neexistuje žádný text věnující se problematice programování mikroprocesorů dsPIC v jazyce C. Všude vidíte samá AVRka a knížek, věnující se procesorům od firmy Microchip, je poměrně málo (jsou sice kvalitní, ale jsou psané na zastaralé řady, a ještě k tomu se věnují pouze programování v assembleru). Pokud se tedy zvídavý čtenář chce dozvědět více, nemá jinou možnost, než sáhnout po některé z anglicky psaných knížek, popř. začít studovat oficiální manuály a programovat stylem "četba-pokus-omyl-četba-oprava-jakžtakž funguje". Takže když přišla chvíle na to, abych se rozhodnul, o čem bude pojednávat moje dlouhodobá maturitní práce, volba padla právě na tuto problematiku.
Asi se ptáte, co všechno musíte umět, než se pustíte do pročítání těchto papírů. Především je to určitá znalost procesoru dsPIC a jeho vnitřní architektury (znalost DSP jádra není nutností, neboť se jím v této knížce nezabýváme), elementární chápání elektroniky (prostě abyste věděli, proč a jak to vlastně všechno děláme) a hlavně chuť naučit se něco nového. Vše ostatní bude postupně vysvětlováno v knize. Tím "všechno ostatní" míním hlavně syntaxi (stavbu) jazyka C a jeho praktické využití s mikroprocesory. Knížka obsahuje 14 odzkoušených příkladů (nepočítám první "Hello world"), na kterých je ukázáno použití právě probírané látky (příklady jsou patřičně okomentovány, aby se čtenář neztratil). Kvůli těmto příkladům bych Vám doporučil si někde sehnat nějaké místo na "prototypování", popř. si vyrobit vývojovou desku, která je součástí mé dlouhodobé maturitní práce.
Předem chci upozornit, že tato práce rozhodně není dokonalá, úplná a věřím, že kdyby ji četl nějaký zasvěcenější člověk, tak by mu hrůzou vstávaly vlasy na hlavě nad některými pasážemi. Skripta jsou hlavně odrazovým můstkem pro všechny, které zajímají procesory dsPIC a chtějí si o nich počíst něco v přijatelné a lidské (doufám) podobě.

No, konec žvanění, pustíme se do práce...

•2. První program

Bývá zvykem, že každá učebnice programování začíná program "Hello word!" a ani já nepůjdu proti proudu. Takový program v jazyce C vypadá takto:

```
#include <stdio.h>

int main(void)
{
  printf("Hello World!\n");
  return 0;
}
```

Po kompilaci (převedení do spustitelné podoby) zdrojového kódu se nám na standardní výstup (u PC je to terminál) vypíše řetězec "Hello world", odřádkuje se a poté je program ukončen. Takto napsaný kód je použitelný pouze u PC kompilerů, neboť naše mikrořadiče pracují trošku jinak (za jistých okolností je spustitelný i na dsPICovi, ale touto možností se zabývat nebudeme, neboť je pro programátora poměrně nevýhodná). Pokud chceme vytvořit program, který bude dělat skoro to stejné, musíme nejprve definovat standardní výstup (bude to terminál počítače), nastavit registry periferií a je-li to třeba, vytvořit nějaké funkce, které nám budou dané znaky tisknout. Nebudu to prodlužovat, program "Hello world!" pro dsPIC vypadá např. takto (nezabývejte se tím, že nerozumíte tomu, jak program funguje - jak se budete prokousávat těmito skripty tak vše pochopíte):

Příklad 02 - 01:

```
#include  <p30fxxxx.h>
#define   FCY       11059200
#define   BRATE     9600
#define   RYCHLOST  (FCY / 16 / BRATE) - 1

void UART_vysli(unsigned char *p_retezec);

int main(void)
{

  ADCON1bits.ADON = 0;
  ADPCFG = 0xFFFF;
  U2BRG = RYCHLOST;
  U2MODE = 0x8000;
  U2STA = 0x0400;
  UART_vysli("Hello world!\n");

  while (1)
    ;
}

void UART_vysli(unsigned char *p_retezec)
{

  while (*p_retezec != '\0')
  {
    while (U2STAbits.UTXBF == 1)
      ;

    U2TXREG = *(p_retezec++);
  }
}
```

Trošku se nám to rozrostlo, nemyslíte? Teď Vás určitě zajímá, jak tento program dostat do mikroprocesoru, nejprve ale musíme zdrojový kód zkompilovat.

●Založení nového projektu a kompilace kódu

Chceme-li si výše napsaný program vyzkoušet, budeme si jej muset nejprve zkompilovat. Zkompilovat znamená převést zdrojový kód do podoby, které mikroprocesor „rozumí“, což je binární soubor zvaný hex. Je tedy jedno, jestli programujete v C, Pascalu, Basicu nebo v čemkoliv jiném, důležité je, jestli máte kompiler, který vám Váš zdrojový kód přeloží. Kompilerů na trhu existuje dostatek na to, aby si každý našel to své. Já si vybral C30 kompiler od firmy Microchip a to především proto, že jej nabízejí pro studenty zdarma (kdo si připlatí, dostane možnost svůj kód optimalizovat). Ostatní firmy sice také nabízejí volně stažitelné verze, ale přidávají do nich různá omezení, např. pouze 2k instrukcí atd. Prakticky je ale jedno, jaký kompiler zvolíte, důležité je, aby dodržoval normu ANSI (pak jsou rozdíly pouze v míře optimalizace kódu plus nějaké specialitky), s určitými úpravami je poté možné přenášet jeden a ten samý kód mezi různými kompilery.

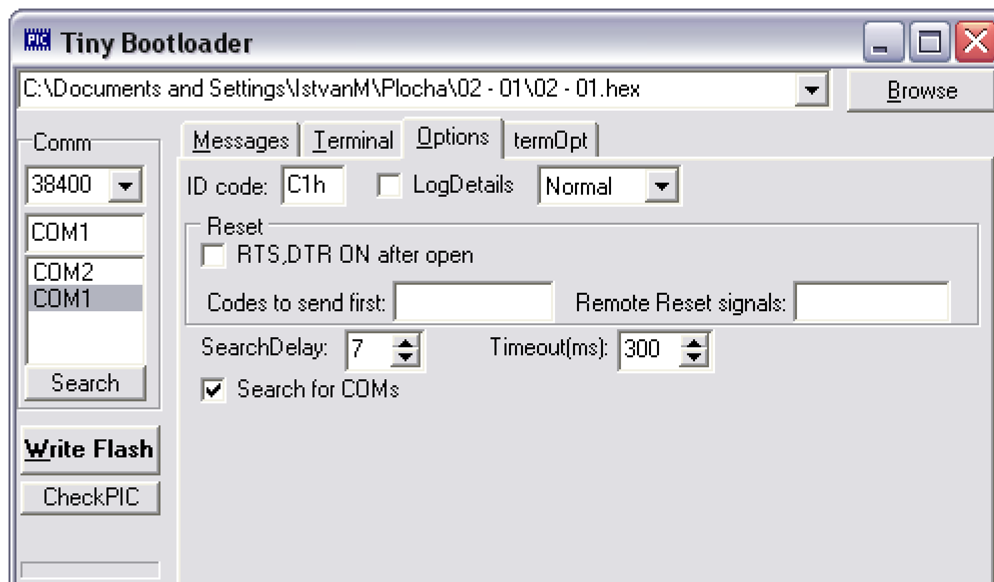
Předpokládám, že založit projekt v MPLABu umíte (postupujte, jako kdybyste zakládali normální assemblerovský projekt), takže Vám ukáži pouze nastavení kompileru. Vyberte *Project?Select Language Toolsite*, kde v liště *Active Toolsit*e vybere položku *Microchip C30 Toolsite*. Teď nás čeká vyhledávání souborů potřebných ke správné funkci kompileru. Musíte projít všechny položky ze seznamu a přidělit jim správný soubor (jeho jméno je uvedeno v závorce), tyto soubory se nalézají ve složce „bin“ v adresáři, kam jste nainstalovali kompiler. Tímto jste si nastavili kompiler a již můžete psát svůj program. Vytvořte nový zdrojový soubor (POZOR! Přípona souboru nebude *.s ale *.c!!!), do kterého vložíte příklad kód z příkladu 02 - 01. Nyní už pouze dáme Build all a máme hotovo! Výsledkem našeho snažení je nám známý hex soubor, který již pouze naprogramujeme do dsPICa.

●Naprogramování procesoru

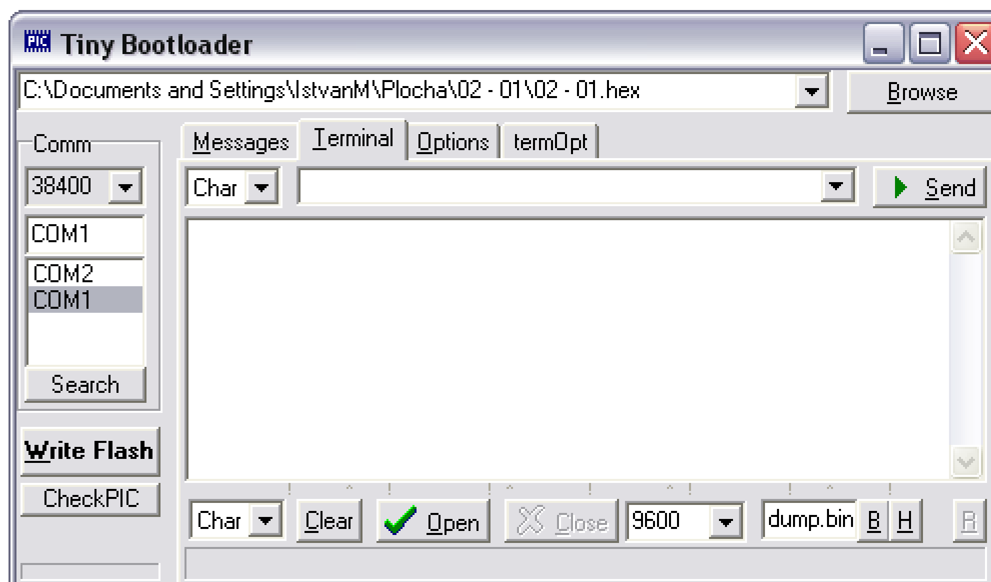
Když jsme teď dostali hex soubor, vyvstává otázka, jak ho do procesoru dostat. Následující postup je pro ty, kteří mají vytvořenou vývojovou desku (součást DMP) a chtějí využít výhody bootloaderu, ostatní nechť se s tím porvou sami. Bootloader je malinkatý (no, závisí na tom, jaký zvolíte) program, díky kterému je možné programovat mikroprocesor pouze pomocí UART sběrnice. Princip spočívá v tom, že nahrajete bootloader do paměti procesoru (tohle je poprvé a naposledy, kdy musíte použít normální programovátko) a při spuštění dsPICa je bootloader schopen pomocí table instrukcí připisovat do programové paměti Váš vlastní program (a nebojte, funkci UART sběrnice jako takové neohrozíte, je možné s ní v programu dále pracovat). Když je bootloader zaváděn do procesoru, je nutné mu specifikovat pojistky procesoru (které bohužel nelze měnit, takže chcete-li např. použít jiný oscilátor, je nutné přepsat celý bootloader) a také rychlost sériové sběrnice (ve Vašem programu pak můžete použít jinou rychlost). Pokud zabrouzdáte do složky *Bootloader*, objevíte jeden předpřipravený zdrojový kód, který má následující vlastnosti:

*Oscilátor HS/2*8 (takže při použití krystalu 11,0592MHz, bude* F_{cy} *opravdu 11,0592MHz)*
Watchdog vypnut
Boren vypnut
MCLR zapnut (vyžaduje bootloader)

Podařilo se-li nám úspěšně naprogramovat bootloader (neděste se, že to trvá trochu delší dobu, musí se projít celá paměť dsPICa, protože bootloader sídlí na jejím konci), můžeme vložit náš mikroprocesor do vývojové desky a zapnout napájení. Pomalu přistupujeme k poslední části našeho snažení a tou je samotné programování našeho programu. Spustíme program TinyBootloader, přejdeme do záložky *Options* nastavujeme dle obrázku:


Tiny Bootloader
C:\Documents and Settings\IstvanM\Plocha\02 - 01\02 - 01.hex
Browse
Comm
38400
COM1
COM2
COM1
Search
Write Flash
CheckPIC
Messages
Terminal
Options
termOpt
ID code:
C1h
LogDetails
Normal
Reset
RTS,DTR ON after open
Codes to send first:
Remote Reset signals:
SearchDelay:
7
Timeout(ms):
300
Search for COMs

Jak je vidno, bootloader s PC komunikuje rychlostí 38400 Bd (znovu opakuji, tato rychlost je pouze na komunikaci bootloader?PC, Váš program může používat úplně jiné rychlosti). Pro jistotu doporučuji otestovat komunikaci tlačítkem *CheckPIC*. Pokud vše proběhlo v pořádku, najděte tlačítkem *Browse* hex soubor zkompilovaného programu a dejte *Write Flash, p*o krátké chvilce se mikroprocesor naprogramuje. Teď už zbývá pouze otestovat správnou funkčnost programu, přejděte proto do záložky *Terminal* a nstavte ji takto:



Číslo 9600 dole vedle tlačítka *Close* značí rychlost, se kterou komunikujeme s procesorem a *Char* v nabídce dole znamená způsob, jak budou přijímaná data zobrazována (zde to budou znaky). Slavnostně tedy otevřete COM port tlačítkem *Open* a zmáčkněte hardwarový reset procesoru na vývojové desce a ejhle! Hello world! (až Vás přestane bavit vysílat neustále jeden a ten samý textový řetězec, uzavřete COM tlačítkem *Close*).

V dalších kapitolách se již budu zabývat syntaxí jazyka.

•3.1 Datové typy

Stejně jako ve všech programovacích jazycích představuje datový typ obor hodnot, kterých může proměnná nabývat. Definujeme-li třeba nějakou proměnnou jako datový typ integer (celá čísla), nemůže nabývat hodnoty 3.14 (ve skutečnosti nabude hodnotu, která je dána implicitní konverzí, v našem případě by se „oddělila“ desetinná část a hodnota, kterou by proměnná měla, by byla 3). Rozsah těchto hodnot závisí především na tom, kolik bytů zabírá daná proměnná v paměti a také to, jestli počítáme se znaménkovými čísly.

●Celá čísla

Datový typ	Počet bitů	Minimum	Maximum
Signed char	8	-128	127
Unsigned char	8	0	255
Signed short	16	-32768	32767
Unsigned short	16	0	65535
Signed int	16	-32768	32767
Unsigned int	16	0	65535
Signed long	32	-2^{31}	$2^{31} - 1$
Unsigned long	32	0	$2^{32} - 1$
Signed long long	64	-2^{63}	$2^{63} - 1$
Unsigned long long	64	0	$2^{64} - 1$

Každý datový typ má v podstatě dvojí formu (jak již jste z tabulky jistě vypozorovali), a to znaménkovou (signed) a neznaménkovou (unsigned). Znaménkový typ má vždy menší rozsah, neboť nejvyšší bit zde plní funkci znaménka. Co se týče umístění dané proměnné v paměti, je uplatňován formát „Little Endian“, neboli nejméně významný bit (anglicky „Least significant bit“ – LSB) je umístěn na nejnižší adrese a nejvíce významný bit („Most significant bit“ – MSB) na nejvyšší adrese. Máme-li tedy například proměnnou typu unsigned int, jejíž hodnota je 0xE54F (toto je hexadecimální zápis, bude vysvětlen dále), bude její rozložení v paměti vypadat následovně:

Adresa	0x800	0x801
Hodnota	4F	E5

Jak vidíte, všechno odpovídá. Za předpokladu, že velikost registru, který leží na jedné adresní buňce, je 8 bitů, pak nám vychází, že proměnná leží na dvou buňkách, tedy 2 * 8 = 16 bitů (vzhledem k tomu, že dsPIC je hlavně 16-ti bitový mikroprocesor, tak výše zmíněné řádky nejsou nejsprávnější formulace. Opravdová velikost jednoho registru je 16 bitů, ale tento registr leží na DVOU adresách, proto je možné k datům přistupovat pouze po sudých adresách).

●Desetinná čísla

Vedle celých čísel lze ještě využít také čísla desetinná:

Datový typ	Počet bitů	Minimum	Maximum
Float	32	2^{-126}	2^{128}
Double	32	2^{-126}	2^{128}
Long double	64	2^{-1022}	2^{1024}

Všimněte si, že desetinná čísla nemají dvojí formu, neboť jsou pouze znaménková.

●Ostatní datové typy

Pokud vás zajímá, jak si Céčko poradí s typem Boolean (logické TRUE a FALSE), tak vás asi zklamu, neboť jej nelze přímo definovat. Pro FALSE má hodnotu 0 a pro TRUE je to jakékoliv číslo nezávisle na tom, na jaký datový typ je proměnná definována (příklady: FALSE – vždy 0; TRUE – 5, 68, -78, 5.698).

Pro bitové proměnné se používají bitové struktury (neexistuje tedy datový typ unsigned bit), o kterých bude řeč až v kapitole o strukturách.

Kombinací existujících datových typů lze vytvářet vlastní datové typy pomocí operátoru typedef, o něm ale také později.

●Optimalizace

V zájmu optimalizace doporučuji vždy pracovat s co nejmenším datovým typem. Když třeba definujeme nějakou proměnnou pro práci se znaky (rozsah 0 – 255, jako ASCII tabulka), je úplně zbytečné použít typ long long, místo toho použijte typ unsigned char, čímž se šetří jak paměť, tak výkon procesoru, neboť práce s typy většími než jsou int (zvláště reálné typy) je mnohem náročnější (to, co s int-em uděláte za pár instrukčních cyklů, může trvat s většími datovými typy i stovky instrukcí). Čím dříve si začnete uvědomovat, že je nutné šetřit s omezenými systémovými zdroji (hlavně s pamětí), tím dřív se z vás stane lepší programátor embedded zařízení.

•3.2 Konstanty

Konstanty prakticky představují zápis určitého čísla (či znaku). Konstanty se dají zapsat několika způsoby:

1Celočíselné konstanty

ADesítkové – klasické desítkové čísla, se kterými se setkáte všude. Příklady: 15, 6, 78, -89
BHexadecimální – začínají buď 0x nebo 0X. Příklady: 0xFFE5, 0X678F
COktalové – začínají číslicí 0 (tento zápis čísel je snad nejméně používaný). Příklady: 075, 052

1Reálné konstanty

Zapisujeme buď s desetinnou tečkou, nebo v semilogaritmickém tvaru. Příklady: 5.2, 68.498, 2e12, 5E3

1Znakové konstanty

Znakové konstanty se píší do uvozovek. Příklad: ‘s‘, ‘9‘, ‘L‘
Každá znaková konstanta je ve skutečnosti nějaké číslo, jehož hodnota se dá určit pomoci ASCII tabulky. Definujeme-li třeba do nějaké proměnné znakovou konstantu ‘A‘, bude její „číselná“ hodnota rovna 65 (dekadicky). Funguje to i naopak, vložíme-li do proměnné číslo 52 (dekadicky), bude v ní znak ‘4‘.
V ASCII tabulce se vyskytují ještě jakési řídící znaky, které vlastně ani žádné znaky nejsou. Provádějí nějakou akci, třeba posun o řádek, písknutí, tabulátor apod. V jazyce C se jim říká *escape sekvence* a jejich zápis se provádí následujícím způsobem (tyto znaky mají své velké využití u terminálu):

Zápis	Hexadecimální hodnota	Význam
‘\n‘	0x0A	Posun o řádek
‘\r‘	0x0D	Návrat na začátek řádku
‘\f‘	0x0C	Nová stránka
‘\t‘	0x09	Tabulátor
‘\b‘	0x08	Posun doleva
‘\a‘	0x07	Písknutí
‘\0‘	0x00	Nulový znak

Znakové konstanty nabízejí ještě jednu zvláštnost a to je psaní zpětného lomítka a apostrofu. Je nutné užít tyto escape sekvence: lomítko ‘\\‘, apostrof ‘\‘‘

1Řetězce

Řetězec je jednorozměrné pole znaků zakončených nulovým znakem (řetězcům je věnována celá kapitola, tam se dozvíte pravý význam těchto slov), které se definují v uvozovkách. Příklad: “Tohle je pekny retezec“
Jazyk C je dokonce tak chytrý, že oddělíme-li řetězce bílými znaky (mezery, tabulátory, nové řádky apod.), tak je spojí do jednoho dlouhého řetězce:
“Toto“ “je“   “pekny“
“retezec“
Bude ve skutečnosti vypadat takto: “Tohle je pekny retezec“
Stejně jako znakové konstanty nabízejí řetězcové zvláštnost v podobě psaní uvozovek pomocí escape sekvence: “\““

●Definice proměnných

Když teď již známe datové typy a zápis různých konstant, můžeme si ukázat, jak vypadá definice. A co je to definice proměnné? Je to příkaz, který přidělí proměnné určitého typu jméno a paměť. Jménu se také častěji říká identifikátor (nemusí se jednat pouze o jméno proměnné, může jít o jméno funkce, struktury, pole - to vše je identifikátor). Jazyk C je case sensitive, což znamená, že rozlišuje malá a velká písmena (takže třeba proměnná X je něco jiného, než proměnná x). Příklady definice proměnných:

```
unsigned int    cislo;
char            znak;
float           pi;
```

Všimněte si, že jsem v druhém příkladu nenapsal, jestli je proměnná znaménková nebo ne. Pokud to napíšete takto, je implicitně nastaven znaménkový typ, takže výše zmíněný příklad je ekvivalent zápisu signed char znak;

Proměnné lze i inicializovat na určitou hodnotu (to znamená, že po spuštění programu v nich bude uložena hodnota zapsaná v inicializaci):

```
signed int      cislo = -258;
unsigned char   znak = 'X';
float           pi = 3.14;
```

•4.1 Přiřazovací příkaz

Přiřazovací příkaz je asi nejčastější konstrukcí, kterou můžete vidět v jakémkoliv jazyce. Ukažme si ji na jednoduchém příkladu:

```
x = 5 - 8;
```

Levá strana tohoto příkazu (příkaz se pozná podle středníku) je nějaká proměnná (pole, struktura, pointer…), do které se ukládá obsah pravé strany neboli výrazu. Levá strana tedy spíše představuje adresu, na které leží daný prvek (v našem případě proměnná pojmenovaná identifikátorem x), čemuž se také říká l-hodnota. Výsledkem toho to příkazu bude přiřazení výrazu 5 – 8 (tedy -3) do proměnné x. V Céčku lze také provádět několikanásobné přiřazení:

```
x = y = 5;
```

Několikanásobné přiřazení se ale vyhodnocuje zprava doleva, takže výše zmíněný příklad se dá rozepsat do dvou přiřazovacích příkazů:

```
y = 5;
x = y;
```

•4.2 Operátory

Už na základní škole jsme se s operátory setkali. Operátor vykonává nějakou operaci mezi operandy (+ třeba vykonává součet dvou operandů). V normálním životě jsme se především setkali s klasickými aritmetickými operátory (+, -, /, *), jazyk C ale nabízí spoustu jiných, pro nás zatím neznámých operátorů. Nepředbíhejme ale, hezky všechno popořadě:

1Aritmetické operátory

Již jsme si je představili, jedná se o sčítání +, odčítání -, násobení *, dělení / a dělení modulo % (zbytek po celočíselném dělení, př: 9 % 2 = 1), jejich významy jsou snad jasné. Problém může nastat, máme-li operandy jiného datového typu, třeba při dělení čísla typu float číslem typu int (nebo obráceně). V takovýchto případech se automaticky převede druhý operand na typ float, tudíž bude celý výraz reálný. Další komplikace může nastat, když tento výraz přiřazujeme proměnné s celočíselným typem. Ukažme si na příkladech:

```
int     x = 3;
float   y = 1.25;
float   vys;

vys = x / y;
```

V tomto případě nenastává žádný problém. Hodnota proměnné x je převedena na typ float (nikoliv proměnná samotná!), čímž bude výraz vyhodnocen jako reálný a tak bude také uložen v proměnné vys (2.4). Druhý příklad ale už funguje jinak:

```
int     x = 3;
float   y = 1.25;
int     vys;

vys = x / y;
```

Hodnota proměnné x se zase převede a výraz bude také vyhodnocen jako reálné číslo, ale vzhledem k tomu, že je přiřazován do proměnné celočíselného typu, tak se od hodnoty výrazu „odtrhne“ desetinná část a ve vys budeme mít hodnotu 2.

Obrácený problém nastane, pokud budeme dělit dvě celá čísla (třeba 5 a 2) a budeme výsledek přiřazovat reálné proměnné (výsledek bude 2 a ne 2.5). Abychom dostali desetinný výsledek, je nutno jeden z operandů přetypovat na reálný tvar (viz. dále).

1Relační operátory ("porovnávací")

Jsou to: vetší než >, menší než <, rovnost == (všimněte si rozdílu mezi operátorem přiřazení = a operátorem rovnosti ==), menší nebo rovno <=, větší nebo rovno >= a nerovnost !=. Výsledkem těchto výrazů je vždy buď hodnota TRUE (1, popř. nenulové číslo), nebo FALSE (nula) dle toho, je-li splněna podmínka. Máme-li třeba příklad:

```
int   i = 5;
int   k = 89;
int   vys;

vys = i < k;
```

tak bude v proměnné hodnota 1, neboť byla splněna podmínka (5 je menší než 89)

1Logické operátory

Jsou pevně spjaty s relačními operátory. Jedná se prakticky o logické funkce: and && (logický součin, „a zároveň“), or || (logický součet, „nebo“) a negace !. Pro zopakování zde uvádím pravdivostní tabulky všech výše zmíněných operátorů:

P	Q	P && Q	P || Q	!P
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

Jak už jsem napsal, je zde velká spojitost mezi logickými a relačními operátory, ukažme si ji na příkladu:

```
int   x = 7;
int   y = 15;
int   vys;

vys = (x < y) && (x == y);
```

(závorky nejsou nutné, logické operátory mají nižší prioritu, ale mně více vyhovují jasně oddělené a viditelné části)

Výsledek v proměnné vys bude 0, neboť:
1) (x < y) je TRUE
2) (x == y) je FALSE
3) (TRUE && FALSE) je FALSE, tedy nula

Je zde také více než vhodné zmínit se o zrychleném vyhodnocování výrazů. Vtip spočívá v tom, že byla-li by třeba v našem příkladě hned první závorka nepravda, tak by k porovnání s druhou závorkou ani nedošlo, neboť dle pravdivostní tabulky 0 && cokoliv je pořád 0. Pomocí zrychleného vyhodnocování a priority jednotlivých operátorů lze vytvářet opravdu krkolomné a na první pohled těžko čitelné výrazy.

1Bitové operátory

Jsou nám asi dobře známy, neboť jsme se s nimi setkávali snad na každém kroku při programování v assembleru.

A<u>Bitový posun</u>

Jsou dva, vlevo << a vpravo >>. Ukažme si je na příkladu:

```
Bchar   x = 5;
char    y = 12;
char    vysx;
char    vysy;

vysx = x << 3;
vysy = y >> 3;
```

C

Jak můžete vidět, ve výrazu je první operand vždy objekt, který se bude posouvat a druhý operand je počet posuvů doprava či doleva (dle operátoru). Výsledky tedy budou:

Posun vlevo *Posun vpravo*

00000101 – 5
<< 3
=
00101000 – 40

00001100 – 12
>> 3
=
00000001 – 1

Asi si říkáte, jak je možné, že druhý příklad není 129 (10000001). Odpověď je jednoduchá: jazyk C se nestará o to, aby vám jednotlivé bity „přehazoval“ z LSB na MSB (popř. obráceně), je proto možné velkým počtem posunů vytvořit třeba nulu, ze které se už nehnete. Přehazování jednotlivých bitů, je tedy ponecháno na programátorovi. Bitové posuny se dají použít jako násobení (dělení) x-tou mocninou dvou, kde x je počet posunů (s tím rozdílem, že bude-li x = 0, tak vám to neposune ani o jeden bitík).

V následujících částech si už někdo může splést logické a bitové operátory. Pro jistotu budu pro všechny zbývající operátory předkládat porovnání.

DBitový součin

Značí se & a provádí s proměnnou funkci zvanou bitový součin. Používá se většinou pro maskování.

Logický součin

00001011 – 11
&&
00100111 – 39
=
00000001 – 1 (TRUE)

Bitový součin

00001011 – 11
&
00100111 – 39
=
00000011 – 3

EBitový součet

Značí se | a provádí s proměnnou funkci zvanou bitový součet. Používá se pro nastavení určitých bitů na 1, aniž by nějak změnil jiné bity.

Logický součet

00001011 – 11
||
00100111 – 39
=
00000001 – 1 (TRUE)

Bitový součet

00001011 – 11
|
00100111 – 39
=
00101111 – 47

FBitová negace

Značí se ~ a provádí s proměnnou funkci zvanou bitovou negaci.

Logická negace

00001011 – 11
!(00001011) – 0 (FALSE)

Bitová negace

00001011 – 11
~(00001011) – 244 (11110100)

GBitový exkluzivní součet

Značí se ^ a provádí s proměnnou funkci zvanou bitový exkluzivní součet. Používá se pro porovnání dvou čísel (je-li výsledek výrazu hodnota TRUE, jsou oba operandy rozdílné)

Bitový exkluzivní součet

00001011 – 11
^
00100111 – 39
=
00101100 – 44 (TRUE - čísla jsou rozdílná)

1Operátor čárka

Značí se opravdu překvapivě, a zajišťuje postupné vyhodnocení (zleva doprava). Příklad:

```
x =(6, 9);
```

Výsledek v proměnné x bude 9. Proč to? Nejprve byla vyhodnocena jako hodnota výrazu číslo 6, ale díky operátoru čárka je výsledná hodnota 9 (proto se to jmenuje postupné vyhodnocení). Používá se třeba pro vícenásobnou inicializaci v cyklech for, nebo třeba také pro definování několika proměnných stejného datového typu:

```
unsigned long   x, y, z = 69;
```

1Ternární operátor („podmíněný operátor“)

Jak už název napovídá, vše se točí kolem nějaké podmínky. Nejlepší bude ukázat si tento operátor na příkladu:

```
x = (5 > 2) ? 9 : 5;
```

Ternální operátor se skládá z podmínky (5 > 2) a podle její pravdivosti (TRUE či FALSE) je výsledek výrazu jedna z hodnot (může tam být ale třeba i funkce atd.). Tyto hodnoty jsou odděleny dvojtečkou. V příkladě výše je podmínka (5 > 2) pravdivá, proto je hodnota v proměnné x rovna 9 (první hodnota je pro splněnou podmínku, druhá pro nesplněnou).

1++ (popř. --)

Používají se ve spojení s l-hodnotou (tedy proměnná, prvek pole apod.) a mají dva významy (analogicky lze ++ nahradit --):

A++i (prefix)

Hodnota v i bude zvýšena o 1 ještě před vyhodnocením výrazu.

B i++ (postfix)

Hodnota v i bude navýšena o 1 až po vyhodnocení výrazu.

Příklady:

```
int i = 2, j = 4, k;

i++;            i bude 3
k = ++j;        k bude 5, j bude 5
k = j++;        k bude 5, j bude 6
k = i++ + --j   k bude 8, i bude 4, j bude 5
```

1Ostatní speciální operátory

AOperátor přetypování

Určitě si vzpomenete na náš problém z povídání o aritmetických operátorech, kdy jsme dělili dvě celá čísla a chtěli jsme, aby byl výsledek výrazu reálný. Bylo řečeno, že je nutné jeden z operandů přetypovat:

```
Bint    x = 5, y = 2;
float   vys;

vys = (float) x / y;
C
```

DPomocí (float) jsme přetypovali hodnotu v proměnné x na reálný typ, což nám změnilo výraz z celočíselného typu na typ reálný (v proměnné vys již máme správný výsledek 2.5).

Jak jste si určitě všimli, přetypování se provádí pomocí (typ_na_který_chceme_přetypovat).

EOperátor sizeof

Sizeof je velice šikovný operátor, který nám zjistí velikost daného objektu v bytech (lze velice jednoduše zjistit třeba velikost datových typů, což se nám bude náramně hodit při alokování dynamické paměti). Příklad:

```
Fx = sizeof(int)
```

GV proměnné x bude hodnota 2, neboť datový typ zabírá v paměti dva byty (musíme si ale dát pozor na různé kompilery, které mohou mít různě velké datové typy. C30 má sice int velký dva byty, to ale neznamená, že třeba CCS kompiler musí mít také 2 byty).

HOperátor přiřazení
Už jsme se s ním setkali a snad všichni pochopili, že se používá pro přiřazení hodnoty výrazu do adresy (l-hodnoty). Proč se ale o něm zde zmiňuji, je důvod, že existují různé „modifikace“. Představme si situaci:

```
Ix = x + 3;
```

JPokud je první z operandů l-hodnota z levé strany přiřazovacího příkazu, lze místo toho napsat:

```
Kx += 3;
```

L

MCož je naprosto stejné, jako výše zmíněný příklad. Zde je výpis operátorů, které lze použít s operátorem přiřazení:

<table>
<tr><th>Zápis</th><th>Význam</th></tr>
<tr><td>l-hodnota += výraz</td><td>l-hodnota = l-hodnota + výraz</td></tr>
<tr><td>l-hodnota -= výraz</td><td>l-hodnota = l-hodnota - výraz</td></tr>
<tr><td>l-hodnota *= výraz</td><td>l-hodnota = l-hodnota * výraz</td></tr>
<tr><td>l-hodnota /= výraz</td><td>l-hodnota = l-hodnota / výraz</td></tr>
<tr><td>l-hodnota %= výraz</td><td>l-hodnota = l-hodnota % výraz</td></tr>
<tr><td>l-hodnota <<= výraz</td><td>l-hodnota = l-hodnota << výraz</td></tr>
<tr><td>l-hodnota >>= výraz</td><td>l-hodnota = l-hodnota >> výraz</td></tr>
<tr><td>l-hodnota |= výraz</td><td>l-hodnota = l-hodnota | výraz</td></tr>
<tr><td>l-hodnota &= výraz</td><td>l-hodnota = l-hodnota & výraz</td></tr>
<tr><td>l-hodnota ^= výraz</td><td>l-hodnota = l-hodnota ^ výraz</td></tr>
</table>

AReferenční a dereferenční operátor
Referenční operátor se používá pro získání adresy, na které leží nějaký objekt. Zapisuje se jako &x (x je objekt, jehož adresu chceme znát). Dereferenční operátor se používá pro přístup do adresy, jejíž hodnota je uložená v proměnné (říká se mu pointer). Zápis je *p (v p je uložena adresa objektu, ke kterému chceme přistupovat). To nám prozatím k těmto operátorům stačí, více se s nimi seznámíme v kapitole o pointerech.

BOperátor typedef
Pomocí tohoto operátoru jsem schopni si vytvořit svůj vlastní datový typ (kombinací nám již známých). Jeho použití je především u složitých definic typu "pointer na funkci, která vrací pointer na pole struktur" apod. Díky typedef lze mnohé komplikované definice značně zjednodušit.

●Priorita operátorů a jejich asociativita

Často se ve výrazech objevuje více než jeden operátor, proto je nutné říci, který má před ostatními „přednost“, což znamená, že bude vyhodnocen jako první. Kupříkladu prefixy mají větší prioritu než aritmetické násobení, proto v příkazu x = ++i / 6; bude nejprve vyhodnocena hodnota i (i +1) a až poté bude podělena šesti.
Asociativita zase značí, v jakém směru bude výraz vyhodnocován. Máme-li příkaz x = ++i + ++j bude nejprve vyhodnocena levá strana výrazu a až poté pravá (asociativita zleva doprava). Následuje přehled operátorů s jejich prioritou (řazení od operátorů s nejvyšší prioritou po ty s nejnižší) a asociativitou:

<table>
<tr><th>Operátory</th><th>Asociativita</th></tr>
<tr><td>. -> () []</td><td>zleva</td></tr>
<tr><td>++ -- ! ~ (přetypování) & * sizeof</td><td>zprava</td></tr>
<tr><td>* / %</td><td>zleva</td></tr>
<tr><td>+ -</td><td>zleva</td></tr>
<tr><td><< >></td><td>zleva</td></tr>
<tr><td>< > <= >=</td><td>zleva</td></tr>
<tr><td>== !=</td><td>zleva</td></tr>
<tr><td>&</td><td>zleva</td></tr>
<tr><td>^</td><td>zleva</td></tr>
<tr><td>|</td><td>zleva</td></tr>
<tr><td>&&</td><td>zleva</td></tr>
<tr><td>||</td><td>zleva</td></tr>
</table>

?:	zprava
Přiřazení a jeho modifikace	zprava
,	zleva

Pravidlo říká, že pokud si nejste jisti prioritou, tak závorkujte (závorky mají nejvyšší prioritu). Nejen, že vám dají stoprocentní jistotu, ale také zpřehlední složitější výrazy.

•5.1 Základní pojmy

V této podkapitole se seznámíme s některými základními pojmy jazyka C, o kterých ještě nebyla řeč, nebo jsem je jen „nakousl".

●Identifikátor

S identifikátorem jsme se již setkali, když jsme pojmenovávali nějaký objekt (prozatím to byla vždy proměnná, ale může se jednat i o funkci, pole apod.). Jméno identifikátor by mělo jednoznačně říkat, co daný objekt je a k čemu slouží, nesnažíme se vytvářet dlouhé, nicneříkající jména typu „ruzovoucky_slon" pro výsledek nějakého součtu, ale na první pohled jasné jako „cena", „pocet_piv" apod (existují ale běžně užívané identifikátory pro různou činnost, třeba identifikátory „i, j, k" pro indexy nebo „s" pro řetězce). Nevyřčená dohoda také říká, že na rozdíl od symbolických konstant (více v kapitole o preprocesoru) píšeme jména objektů malými písmeny.

●

●Klíčová slova

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Jsou speciální slova určená pro překladač. Jsou to třeba jména datových typů, paměťových tříd a různých „zvláštních" slov využívaných v příkazech. ANSI norma definuje tato jména pro klíčová slova (žádný identifikátor nesmí mít stejné znění jako je klíčové slovo!):

●Komentáře

Komentář použijeme tehdy, chceme li popsat (vysvětlit) nějakou část programu, popř. nějaký příkaz. Snažíme se svůj kód už ve fázi programování komentovat, protože později si na to určitě čas nenajdeme. Komentáře nejsou zpracovávány preprocesorem (nejsou součástí samotného kódu). Komentáře se dají psát dvojím způsobem:

➔Při tomto způsobu komentář umisťujeme mezi /* a */, takže příklad by vypadal /*toto je komentář*/.
Chceme-li psát komentář na několik řádků, postupujeme dle následujícího schématu:
/*komentář_1
komentář_2*/

➔Zde se píše komentář za dvojici lomítek //. Za komentář se poté bere vše, co je nalevo od lomítek (osobně používám raději tento způsob, ale závisí jenom na vás, co si vyberete). Příklad:

```
➔x = a + b;    //Tady je komentář
```

➔

●Hlavičkové soubory

Hlavičkové (*header*) soubory obsahují deklarace a definice objektů, využívající výhod odděleného překladu. Voláme-li např. nějakou funkci ze standardních knihoven, je nutné vložit příslušný hlavičkový soubor pomocí klíčového slova include. Příklad:

```
#include <string.h>     //Vloží hlavičkový soubor pro práci s řetězci. Od této chvíle
                        //můžeme volat funkce z této knihovny
```

Všimněte si použití znaku #, který značí preprocesorový příkaz.
U mikrokontrolérů se ještě vkládá soubor, který definuje jména jednotlivých SFR, bitů a makra (prakticky se jedná o céčkovou náhradu za soubory *.inc). Pro naše dsPICy se jedná o soubor <p30f3013.h>:

```
#include <p30f3013.h>   //Nyní můžeme pracovat se SFR jejich jmény, jak jsou uvedeny v
                        //datasheetu, třeba TMR1 = 0;
```

Je dobré se ještě zmínit o jedné zajímavé vlastnosti a tou je přístup k identifikátorům jednotlivých bitů. Představme si situaci, kdy chceme třeba vypnout AD převodník a nechceme ovlivnit ostatní bity. Namísto zdlouhavého maskování a binárních operací můžeme využít faktu, že všechny logicky ucelené celky jednotlivých registrů se dají ovládat pomocí bitové struktury (co to přesně je, se dozvíme v kapitole o strukturách). Pokud víme, jak přistupovat k jednotlivým prvkům struktury (využívá se operátoru tečka), je vše ostatní již hračkou:

```
ADCON1bits.ADON = 0;
```

Obecný zápis je tedy **SFRbits.bit_ktery_chceme_nastavit**
Logicky ucelený celek se myslí skupina bitů, které ovlivňují jedno nastavení, třeba bity TCKPS<5:4> v registru T1CON:

```
T1CONbits.TCKPS = 0x3;     //Nastavena předdělička 256
```

•5.2 Obecná struktura kódu

Každý kód musí obsahovat funkci main (pokud ji neobsahujete, jedná se buď o chybu, nebo o soubor, ve kterém jsou uloženy definice funkcí), na kterou se „skočí“ vždy na začátku. Velice podobnou strukturu má i programování v assembleru, např. u 8mi bitových PICů (řada 16F a její ekvivalenty) by se dala taková „funkce main“ nalézt na adrese 00h (kde se nachází skok na návěští), na kterou bude instrukční čítač ukazovat hned po spuštění programu. Pokud ji nenalezne, provede reset a bude jí hledat zase. U dsPICů zase tento kód leží na adrese 100h atd. Pokud ale programujeme pomocí C, tak vše funguje trošičku jinak. Funkce main zůstala, ale první, co se po zapnutí procesoru spustí, je takzvaný c0 kód, jenž má za úkol inicializovat proměnné, konstanty, zpracovat paměť a celkově tak nějak připravit procesor (chování tohoto kódu nelze ovlivnit, aniž bychom tím ohrozili správné fungování našeho programu). Až po skončení c0 se skočí na funkci main. Prakticky nás při programování tento c0 kód nijak neobtěžuje (programujeme, jako by ani nebyl), ale musíme si uvědomit, že nám i při sebemenším programu něco ukousne z paměti.
Další důležitou věcí, co si musíme uvědomit, je fakt, že neexistuje něco jako konec programu, což je zásadní rozdíl oproti programování na počítači. Zatímco u PC se při skončení programu pouze uvolní systémové zdroje (paměť, výkon procesoru), které může operační systém přidělit jinému programu, tak u mikrokontroléru se provede reset, čímž se spustí celý program znovu. Jinými slovy, programy pro mikrokontroléry musí „běhat“ v nekonečné smyčce do doby, než je vypnuto napájení, proto je také funkce main nekonečná (veškeré příkazy, volání funkcí se provádí uvnitř této funkce, skončí-li funkce main, nastane reset).
Jsme li obeznámeni s těmito fakty, můžeme se podívat na to, jak vlastně takový kód vypadá:

```
Vložení hlavičkových souborů
Definice symbolických konstant a maker

Deklarace funkčních prototypů
Definice funkcí
Definice globálních proměnných, polí, struktur…

int main(void)
{
   Definice lokálních proměnných, polí, struktur…

   Příkazy;
   Nekonečná smyčka
   {
      Příkazy;
   }
```

```
}
```

Toto je obecné schéma, jak by měl každý zdrojový kód vypadat. Vzhledem k tomu, že se všemu budeme více věnovat v samostatných kapitolách, zmíním se zde pouze o složených závorkách.
Abychom definovali, co patří do funkcí a co ne, používají se složené závorky {}. Do těchto závorek tedy píšeme všechny příkazy, které se mají ve funkci vykonat. {} mohou mít dvojí význam:

A**Blok**

Pokud je uvnitř složených závorek také definice objektů, jedná se o blok. Příkladem je výše uvedený blok funkce main, který obsahuje definice lokálních proměnných a příkazy.

B**Složený příkaz**

Neobsahují-li složené závorky definici, jedná se pouze o složený příkaz, který se navenek chová jako jeden příkaz.
Příklad:

```
C{
   x = a + b;
   a++;
}
```

1.1
2

•6. Cykly

O cyklu jsme si již povídali, že je důležitou součástí kódu. Každý program musí běhat v cyklu, jinak se bude neustále restartovat a bude velice neefektivní. Pomocí cyklu jsme již schopni také psát „použitelné“ programy, takže od této kapitoly si budeme moct ukázkové příklady naprogramovat a vyzkoušet na desce.
Logika cyklů je taková, že budou opakovat určitý sled příkazů do doby, než bude porušena podmínka (např. budeme přičítat k nějaké proměnné jedničku do doby, než bude tato proměnná větší než 5). Ačkoliv je druhů cyklu několik, tak všechny mají společné, že se cyklus bude opakovat do doby, co bude podmínka TRUE (tedy jakékoliv nenulové číslo). Je tedy načase, abychom se s jednotlivými cykly seznámili.

1Cyklus while
První a nejjednodušší cyklus je while, u něhož je podmínka vyhodnocována při vstupu a na začátku každého nového cyklu. Jeho obecné schéma vypadá takto:

```
while (podmínka)
{
   příkaz;    //Příkaz, který bude opakován do doby, než bude porušena podmínka
}
```

Příklad 06 – 01:
Snad nejzákladnější program, který se naučí každý ze začátku programování mikroprocesorů. Program v nekonečné smyčce kopíruje stav z dolních 4 bitů PORTB (napojených na přepínače) na horní 4 bity PORTB (napojených na diody).

```
#include <p30f3013.h>           //Vložení hlavičkového souboru pro dsPIC30F3013

int main(void)
{

   ADCON1bits.ADON = 0;         //Vypnutí AD převodníku
   ADPCFG = 0xFFFF;             //Všechny piny digitání
   TRISB = 0x000F;              //Nastavení směru jednotlivých pinů
                                //RB0 - RB3 = vstupní, RB4 - RB7 = výstupní

   //Prakticky nekonečný cyklus, podmínka bude
   //pořád TRUE
   while (1)
   {
      PORTB <<= 4;              //Posunutí dolních 4 bitů o 4 pozice výš
   }
}
```

Jumpery:
Jumper 1 - 4 = Přepínače
Jumper 5 - 8 = Diody

Toto je typický příklad nekonečného cyklu. Díky podmínce, která nebude nikdy FALSE (i kdyby pršely trakaře), bude do vypojení napájení kopírovat jednotlivé bity. Program díky tomu nikdy neskončí a je uzavřen v nekonečné smyčce.

Příklad 06 – 02:
Tento příklad bude každou (přibližně) vteřinu přičítat k PORTB jedničku (celý je napojen na diody). Konstanta 43402 je vypočítána dle schématu: Fcy = 11059200Hz ? přibližně Tcy = 90ns ? 1s / 256(předdělička)/90ns ? 43402.

```
#include <p30f3013.h>

int main(void)
{

   ADCON1bits.ADON = 0;     //Vypne AD převodník
   ADPCFG = 0xFFFF;         //Všechny piny digitální
   TRISB = 0;               //Nastavení směru jednotlivých pinů
                            //Všechny jsou výstupní
```

```
   PORTB = 0;               //Nulování PORTB
   TMR1 = 0;                //Nulování čítače
   T1CON = 0x8030;          //Spustí čítač 1, předdělička 256

   //Nekonečný cyklus
   while (1)
   {
      //Procesor bude běhat v tomto cyklu do té doby,
      //než nebude číslo v TMR1 větší (či rovno) než
      //43402
      while (TMR1 < 43402)
         ;                  //Takto se vytváří cyklus, který nemá žádné příkazy

      TMR1 = 0;             //Nulovná čítače
      PORTB++;              //Přičte jedničku do PORTB
   }
}
```

Jumpery:

Jumper 1 - 8 = Diody

1Cyklus do – while

Tento způsob je velice podobný předchozímu s jedním (ale o to zásadním rozdílem). U cyklu while byla podmínka testována hned při vstupu do cyklu a pokud je tato podmínka nepravdivá, tak se celý cyklus přeskočí a nebude proveden žádný příkaz. Naproti tomu do – while testuje podmínku až na konci cyklu, to znamená, že i kdyby byla podmínka FALSE, tak se alespoň jednou příkazy z toho cyklu vykonají. Názorně na obecném schématu:

```
do
{
   příkaz;            //Tento příkaz bude alespoň jednou vykonán
} while (podmínka);
```

Příklad 06 – 03:

Zde je vidět, že ačkoliv je podmínka zjevně nepravdivá, tak přesto bude cyklus jednou vykonán. Důkazem toho bude indikování čísla 0xAA diodami a následným skokem do nekonečné smyčky.

```
#include <p30f3013.h>

int main(void)
{

   ADCON1bits.ADON = 0;       //Vypne AD převodník
   ADPCFG = 0xFFFF;           //Všechny piny digitální
   TRISB = 0;                 //Nastavení směru jednotlivých pinů
                              //Všechny jsou výstupní
   PORTB = 0;                 //Nulování PORTB

   //Ačkoliv je podmínka nepravdivá, cyklus se jednou vykoná
   do
   {
      PORTB = 0xAA;
   }  while (2 > 5);          //Podmínka je vyhodnocena jako FALSE,
                              //cyklus se již nebude opakovat

   //Nekonečná smyčka
   while (1)
      ;
}
```

Jumpery:

Jumper 1 - 8 = Diody

3) **Cyklus for**

Je to takové „3 v 1“. Představte si situaci dle následujícího příkladu:

```
i = 0;
while (i < 5)
{
   i++;
}
```

Nejprve se provede inicializace i na 0 a poté se bude k i přičítat 1 do doby, než bude hodnota v i větší (či rovno) hodnotě 5. Stejný cyklus se dá zapsat mnohem elegantněji pomocí for:

```
for (i = 0; i < 5; i++)
   ;
```

První část v závorce je inicializace (která se vykoná pouze jednou a to při vstupu do cyklu), druhá je podmínka a třetí je příkaz, který se provede na konci cyklu. Všechny tyto části jsou odděleny středníkem a pomocí operátoru čárka lze vytvářet třeba další inicializace atd. Je možné některé části (kromě podmínky) vynechat, píšeme potom pouze středník. Podmínka je vyhodnocována při vstupu a na začátku každého nového cyklu.

Příklad 06 – 04:

Pomocí cyklu for a while budeme na LED diodách zobrazovat mocniny 2 (do 128) s přibližně vteřinovým intervalem. Výpočet konstanty je stejný jako v příkladu 06 – 02.

```
#include <p30f3013.h>

int i;  //Definice proměnné i

int main(void)
{

   ADCON1bits.ADON = 0;          //Vypne AD převodník
   ADPCFG = 0xFFFF;              //Všechny piny digitální
   TRISB = 0;                    //Nastavení směru jednotlivých pinů
                                 //Všechny jsou výstupní
   PORTB = 0;                    //Nulování PORTB
   TMR1 = 0;                     //Nulování čítače
   T1CON = 0x8030;               //Spustí čítač 1, předdělička 256


   //Nekonečný cyklus
   while (1)
   {
      //Cyklus, který bude na LED zobrazovat
      //mocninu dvou
      for (PORTB = 1, i = 0; i <= 7; i++)
      {
         while (TMR1 < 43402)    //Čekáme 1 vteřinu
            ;

         TMR1 = 0;               //Nulování čítače
         PORTB <<= 1;            //Vynásobení PORTB dvěmi
      }
   }
}
```

Jumpery:
Jumper 1 - 8 = Diody

Možná si říkáte, že jsem vytvořil proměnnou i úplně zbytečně, že stačilo pouze testovat podmínku PORTB != 256. Důvod pro zavedení proměnné i je ukázání inicializací více proměnných v cyklu for.

●Příkazy break a continue

Tyto příkazy jsou úzce vázány na cykly a rozšiřují možnosti ovládání průběhu cyklu.

➔Break

Je-li tento příkaz použit uvnitř nějaké smyčky tak jeho použití spočívá ve „vyskočení“ z tohoto cyklu (nastane třeba nějaká situace, kterou je nutné ošetřit) a pokračováním příkazy, které se nacházejí za touto smyčkou (pokud máme dva cykly a jeden je „vložen“ do druhého, tak použití příkazu break v nejvnitřnějším cyklu skočí do cyklu vnějšího). Lze jej ještě použít u příkazu switch, ale o tom jindy.

➔Continue

Tento příkaz zase naopak skočí na na konec cyklu, čímž prakticky započne „nové kolo“ (je-li použit v cyklu for, vykoná se také „konečný“ příkaz, v příkladu 06-04 by to byl příkaz i++)

•7. If a else

Při programování budete muset občas udělat nějaký příkaz v závislosti na nějaké podmínce, k čemuž se používá příkaz if. Jeho obecné schéma:

```
if (podmínka)
   příkaz;         //Příkaz, který se vykoná, je-li podmínka TRUE
příkaz;            //Příkaz, který se provede nezávisle na podmínce
```

Pro podmínku prakticky fungují stejná pravidla, jako u cyklů (použití logických operátorů). Pokud budeme chtít vykonat více příkazů, použijeme složený příkaz:

```
if (podmínka)
{
   příkaz;      //Příkazy, který se vykonají, je-li podmínka TRUE
   příkaz;
   příkaz;
}
příkaz;         //Příkaz, který se provede nezávisle na podmínce
```

K if se váže ještě jedno klíčové slovo a to je else. Používá se tehdy, chceme-li vykonat nějaký příkaz, když nebude platit první podmínka:

```
if (podmínka)
   příkaz;         //Příkaz, který se vykoná, je-li podmínka TRUE
else
   příkaz;         //Příkaz, který se vykoná, je-li podmínka FALSE
příkaz;            //Příkaz, který se provede nezávisle na podmínce
```

<u>Příklad 07 – 01:</u>

V závislosti na stavu RB0 (tlačítko) bude na RB1-7 (diody) číslo 0xAA (RB0 je rovno 1) nebo 0x54 (RB0 je rovno 0).

```
#include <p30f3013.h>

int main(void)
{

   ADCON1bits.ADON = 0;          //Vypne AD převodník
   ADPCFG = 0xFFFF;              //Všechny piny digitální
   TRISB = 0x1;                  //Nastavení směru jednotlivých pinů
                                 //RB0 - vstup, RB1-7 = výstup
   PORTB = 0;                    //Nulování PORTB

   //Nekonečný cyklus
   while (1)
   {
      if (PORTBbits.RB0 == 1)    //Je stav na RB0 rovný 1?
         PORTB = 0xAA;           //Ano, je
      else
         PORTB = 0x54;           //Ne, není
   }
}
```

Jumpery:

Jumper 1 = Přepínač

Jumper 2 - 8 = Diody

Příkazy if lze do sebe vnořovat a také lze používat poměrně oblíbenou konstrukci else if.

Příklad 07 – 02:

V tomto příkladě máme dva přepínače (RB0 a RB1) a 6 diod (RB2 – RB7). Dle stavu na RB0 nám buď diody „čítají“ (RB0 = 1) nebo „stojí“ (RB0 = 0). Pokud platí podmínka RB == 1, tak dále testujeme stav na RB1. Pokud je jeho stav logická 1, tak se bude z hodnoty na výstupu odčítat 1, pokud je stav logická 0, bude naopak 1 přičítat. Zbytek je už pouze ošetření, aby mohl výstup nabývat pouze některou z hodnot intervalu 0 - 63.

```
#include <p30f3013.h>
```

```c
unsigned char cislo = 0;  //Definice a inicializace
                          //proměnné cislo

int main(void)
{

  ADCON1bits.ADON = 0;            //Vypne AD převodník
  ADPCFG = 0xFFFF;                //Všechny piny digitální
  TRISB = 0x3;                    //Nastavení směru pinů
                                  //RB0-1 = vstupní, RB2-7 = výstupní
  PORTB = 0;                      //Nulovní PORTB
  T1CON = 0x8020;                 //Spustí čítač 1, předdělička 64
  TMR1 = 0;                       //Nuluje čítač

  //Nekonečná smyčka
  while (1)
  {
    while (TMR1 < 43210)          //Čekáme, než uplyne 0,25s
      ;

    if (PORTBbits.RB0 == 0)       //Je RB0 rovno 0?
      continue;                   //Ano, je. Nic nepřičítej ani neodečti,
                                  //čítač "stojí" a díky příkazu continue
                                  //začíná nový cyklus
    else                          //Ne, není.
    {
      TMR1 = 0;                   //Nulujeme čítač
      if (PORTBbits.RB1 == 0)     //Je RB1 rovno 0?
      {                           //Ano je, proto k výstupu přičti 1
        cislo++;
        if (cislo == 64)          //Přetekla hodnota v proměnné
                                  //cislo povolených 63?
          cislo = 0;              //Ano, nuluj cislo

        PORTB = (cislo << 2);     //Upravené cislo na
                                  //PORTB
      }
      else
      {                           //Ne, není, proto od výstupu odečti 1
        cislo--;
        if (cislo == 255)         //Podtekla hodnota v proměnné
                                  //cislo povolenou 0?
          cislo = 63;             //Ano, nastav novou hodnotu

        PORTB = (cislo << 2);     //Upravené cislo na
                                  //PORTB
      }
    }
  }
}
```

Jumper 1 - 2 = Přepínače
Jumper 3 - 8 = Diody

Zde jsem se snažil demonstrovat použití příkazu continue, který okamžitě skáče na začátek cyklu (tedy na čekání, než uplyne 0,25s).
Budete-li si projíždět některé cizí zdrojové kódy, uvidíte, že se ve více případech používá jakýchsi zkrácených zápisů podmínek. Namísto if (výraz != 0) se píše jenom if (výraz), což využívá vlastnosti TRUE, které může nabývat jakékoliv nenulové číslo. Naopak if (výraz == 0) je delší náhrada za if (!výraz).

•8.1 Switch

Switch (*přepínač*) je příkaz, který se použije pro větvení programu. Doslova „přepíná“ mezi jednotlivými větvemi v závislosti na vstupní podmínce. Pro lepší pochopení zde předkládám porovnání s příkazem if:

```
If (znak == 'A')
   Větev1;
else
{
   if (znak == 'B')
      Větev2;
   else
      Větev3;
}
```

V tomto příkladu testujeme hodnotu v proměnné znak. Pokud je rovna ‘A‘ (0x41), tak se provede Větev1. Pokud je podmínka vyhodnocena jako FALSE, testujeme dále znak na rovnost se znakovou konstantou ‘B‘ (0x42). Při pravdivosti podmínky se vykoná Větev2. Neplatí-li tedy ani jedna podmínka, bude provedena Větev3. Pomoci příkazu if lze tedy také vytvářet jakési větvení programu, ale nedáme-li si pozor, dá se v tom poměrně dobře zamotat. Mnohem elegantnější způsob je použití již zmíněného přepínače:

```
switch (znak)
{
   case 'A':
      Větev1;
      break;
   case 'B':
      Větev2;
      break;
   default:
      Větev3;
      break;
}
```

Ačkoliv oba příklady dělají totéž, je nutné si při pracování s přepínačem uvědomit pár zásadních věcí. První je použití příkazu break, který vlastně ukončuje jednu větev a opouští přepínač. Kdybychom jej v první části (case ‘A‘) vynechali, vykonala by se i větev2 a až poté by se vyskočilo s přepínače (protože zde už se break nachází). Občas se to hodí, chceme-li pro různé hodnoty vykonat stejný sled příkazů. Další zvláštností je „případ“ default. Pokud se hodnota v znak nerovná žádné z nabízených možností, bude vykonána větev nacházející se v části default. Příkaz switch ji nemusí obsahovat, ale je to takové „blbuvzdorné“ řešení, takže se celkově doporučuje používat. Zbylá omezení se již pouze týkají podmínky. U toho příkazu musí být v každém případě podmínka typu něco == něco (nelze tedy psát case znak < 8 apod.), což je oproti if určitá nevýhoda. Poslední věcí, kterou je nutné dodržet, je typ vyhodnocovaného objektu. Ten musí být pouze typu int, není proto možné testovat reálné datové typy.

Příklad 08 – 01:

Pomocí příkazu switch budeme měnit výstup na LED diodách (RB3 – RB7). Vstupem budou tři tlačítka (RB0 – RB2) a dle čísla vytvořeného na nich bude možné vybrat 5 různých výstupů.

```
#include <p30f3013.h>

unsigned char vstup;    //Definice proměnné vstup

int main(void)
{

   ADCON1bits.ADON = 0;          //Vypne AD převodník
   ADPCFG = 0xFFFF;              //Všechny piny digitální
   TRISB = 0x7;                  //Nastavení směru jednotlivých pinů
                                 //RB0-2 - vstup, RB3-7 = výstup
   PORTB = 0;                    //Nulování PORTB

   //Nekonečný cyklus
   while (1)
   {
      vstup = PORTB & 0x7;       //Maskování PORTB a přiřazení
                                 //stavu RB0-2 do proměnné vstup
      switch (vstup)             //Příkaz přepínače, vybere větev v závislosti
```

```
                                  //na tom, jaké číslo je uložené ve vstup
     {
       case 0:                    //Větev pro vstupní číslo = 0
         PORTB = 0xA8;
         break;
       case 1:                    //Větev pro vstupní číslo = 1
         PORTB = 0x50;
         break;
       case 2:                    //Větev pro vstupní číslo = 2
         PORTB = 0xD8;
         break;
       case 3:                    //Větev pro vstupní číslo = 3
         PORTB = 0x20;
         break;
       default:                   //Větev pro vstupní čísla 4-7
         PORTB= 0xF8;
         break;                   //Zde sice příkaz break nutný není,
                                  //kvůli "kráse" se přidává
     }
   }
}
```

Jumpery:
Jumper 1 - 3 = Přepínače
Jumper 4 - 8 = Diody

Samozřejmě že jsme mohli použít několik if příkazů, ale není snad přehlednější použít přepínač?

•8.2 Goto a return

●Goto

V strukturovaném jazyce (jako je C) je použití goto naprosto zbytečné, neboť všechny problémy se dají řešit právě strukturovaným programováním. Předkládám zde obecné schéma, ale snažte se goto vyhnout velkým obloukem:

```
goto navesti;

navesti:
Příkazy;
```

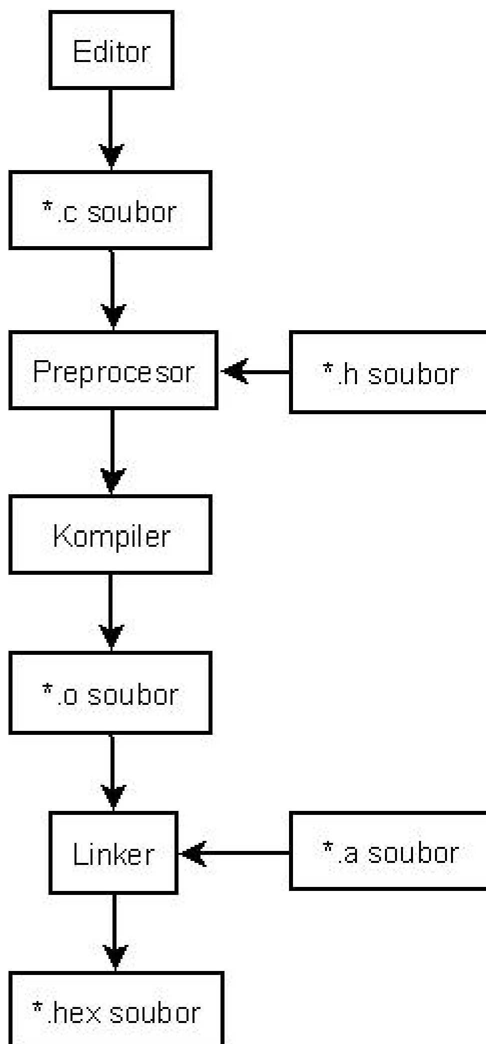
●Return

Ačkoliv to vypadá, že příkaz return je další „pozůstatek“ po assembleru, je hojně využíván ve funkcích. Používá se především pro návrat z funkce do místa, odkud byla funkce volána (v assembleru prakticky call ? return instrukce). Kromě toho je schopen vracet hodnotu, ale o tom až někde v kapitole o funkcích.

•9. Preprocesor

Ještě před tím, než si řekneme, co je preprocesor a k čemu slouží, nás čeká jedna malá odbočka – zpracování zdrojového kódu.

●Obecné schéma zpracování kódu



Jedná se o podstatně zjednodušené schéma, neboť prakticky každý krok (hlavně co se týče Linkeru) lze ovlivnit různými nastaveními a tím také získat různé vedlejší produkty (rozuměj soubory), což ale nás jako amatéry moc nezajímá (zájemce odkazuji na oficiální helpy Microchipu a přeji jim hodně štěstí při jejich studování). Nyní se tedy popíšeme jednotlivé kroky:

1Editor

2Je to program, do kterého zapisujeme svůj zdrojový kód. Může to být jakýkoliv textový editor, který si zdrojový kód neobohatí o různé formátovací „pimprlátka". Výsledkem je *.c soubor.

3Preprocesor

4Tato část překladače nedělá nic jiného, než že upraví zdrojový kód do formátu, se kterým si poradí kompiler. Tato úprava zahrnuje (mimo jiné) činnosti jako rozvoj maker, vynechání komentářů a také vložení hlavičkových souborů (které je nutno pomocí nám již známé instrukce include vložit). Výsledkem je tentýž *. c soubor, ale s výše jmenovanými úpravami.

5Kompiler
6Nejdůležitější krok je tzv. přeložení (kompilace) zdrojového kódu do souboru *.o neboli souboru „relativních adres“ (v této fázi je již zdrojový kód přeložen, ale jednotlivým registrům, proměnným apod. nejsou ještě přiřazeny adresy).

7Linker
1.Posledním úkolem překladače je přiřazení absolutních adres (např. zamění relativní adresu TRISB za absolutní adresu 0x2C6). Aby mohl linker správně fungovat, potřebuje tzv. *.a soubor, který je vlastně knihovna, obsahující všechny adresy jednotlivých objektů. Pokud nikde nenastala žádná chyba, tak dostaneme *.hex soubor, který už můžeme naprogramovat do mikroprocesoru.

To by bylo vše, teď se již opravdu pustíme do preprocesoru.

Jak již bylo řečeno výše, preprocesor jaksi „přežvýká“ zdrojový kód tak, aby mohl být následně zpracován kompilerem. Pojďme se podívat na to, co všechno preprocesor umí.

●Makra bez parametrů

Představte si situaci, že na několika místech v programu pracujete s nějakou konstantou (třeba π). Namísto toho, abyste pokaždé psali 3.14159265358979, tak lze využít mnohem příjemnější konstrukce a tou je definování makra (neboli symbolické konstanty):

```
#define PI   3.14159265358979
```

Následně můžeme kdekoliv v programu (kromě řetězců) použít symbolickou konstantu PI. Při zpracování zdrojového kódu se takováto symbolická konstanta „rozvine“, takže preprocesor nahradí PI za hodnotu 3.14159265358979. Co se týče zápisu, název konstanty a její hodnota je oddělena mezerou, takže cokoliv za první mezerou od PI doprava je její hodnota.
Definovat můžeme nejenom číselné konstanty, ale i výrazy, příkazy (více příkazů v jednom makru je třeba spojit pomocí bloku {}), textové řetězce apod. Ukažme si na příkladu, jak vypadá použití symbolické konstanty (jejíž hodnota je výraz) a jak bude vypadat tentýž kód po rozvinutí makra.
Před:

```
#define SOUCET  (6 + 7)

main()
{
   char i;

   i = SOUCET;
   while(1)
      ;
}
```

Po:

```
main()
{
   char i;

   i = (6 + 7);
   while(1)
      ;
}
```

Všimněte si, že preprocesor za symbolickou hodnotu dosadí přesně to, co jsme definovali jako hodnotu.

Mimo námi definovanými makry existují také tzv. předdefinované (to znamená, že je nemusíme definovat konstrukcí #define) symbolické konstanty dodávané spolu s kompilerem, ukažme si tedy některé z nich:
A**__DATE__**
Při rozvinutí tohoto makra dostaneme datum překladu (kompilace) zdrojového kódu. Formát hodnoty makra je měsíc

(3 znaky, bohužel anglicky) den (2 znaky) rok (4 znaky), př: Oct 26 2008 (samozřejmě s mezerami). Vzhledem k faktu, že toto makro má jako hodnotu posloupnost znaků (neboli řetězec), nelze jej jednoduše umístit do proměnné, ale do pole. S polem ještě neumíme pracovat, proto zatím ponecháme toto makro stranou.

B **__TIME__**
Jeho hodnota je čas kompilace zdrojového kódu ve formátu hh:mm:ss, př: 13:18:47 (je to zase řetězec).

C **__FILE__**
Používá se pro zjištění jména zdrojového souboru, formát jméno.c (překvapivě zase řetězec).

Existují i další, ale jejich použití není tak časté.

●Makra s parametry

Makra s parametry jsou takové jednodušší funkce. Podívejme se nejprve na obecný zápis (na rozdíl od symbolických konstant píšeme jméno makra s parametrem malými písmeny):

```
#define jmeno_makra(parametr_1, ....parametr_N)   hodnota_makra
```

Parametry jsou něco jako vstupní hodnoty, se kterými makro „něco udělá“ (hodnota_makra) a vyhodí výsledek. Takové klasické makro může třeba sloužit k součinu dvou čísel:

```
#define soucin(x, y)    x * y

main()
{
   int i = 3, j = 4, k;

   k = soucin(i, j);
   while(1)
      ;
}
```

V proměnné k budeme mít uloženou hodnotu součinu (12) vypočítanou makrem soucin. Vstupní parametry jsou proměnné i a k, které se dosadí do hodnoty makra (po rozvinutí makra bude tedy mít levá strana přiřazovacího příkazu hodnotu i * j) a normálně se výraz spočítá. Dosud vše funguje krásně, ovšem v některých situacích se makra chovají trošku jinak. Další příklad:

```
#define soucin(x, y)    x * y

main()
{
   int i = 3, j = 4, k;

   k = soucin(i + 1, j - 1);
   while(1)
      ;
}
```

Pokud někteří z vás očekávali, že budou mít v k uloženou hodnotu 12, nechť si pro dnešek odpustí večeři, neboť výsledkem je číslo 6. Jak to? Podívejme se, co se stane při rozvinutí makra:

```
k =  i + 1 * j - 1;
```

Vzhledem k tomu, že operace násobení má vyšší prioritu než sčítání, tak se provede nejprve operace 1 * j, což se následně přičte k hodnotě proměnné i a nakonec se odečte jednička. Tato zrada se řeší uzávorkováním všech vstupních parametrů v hodnotě makra. Správná definice by měla tedy vypadat takto:

```
#define soucin(x, y)    ((x) * (y))
```

Definice zapsaná tímto způsobem se již rozvine do námi požadovaného tvaru k = ((i + 1)*(j – 1)); kde i + 1 je hodnota operandu x a j – 1 je hodnota operandu y. Možná se ptáte, proč je hodnota makra uzávorkovaná celá. Důvodem je to, že makro může být zase operandem nějakého výrazu, což by se po rozvinutí nemuselo chovat tak, jak chceme.

●Podmíněný překlad

Občas se stane, že vyvíjíme nějaký software pro různé platformy (u PC buď pro Windows nebo Linux, u embedded systémů to může být třeba PIC nebo AVR). Můžeme tedy daný software napsat pro každou platformu zvlášť, což je poměrně časově náročné. Uvědomíme-li si, že Céčko je poměrně dobře přenositelný jazyk (na rozdíl od assembleru, u něhož nemůžeme přenášet zdrojový kód mezi jednotlivými výrobci mikroprocesorů), můžeme za určitých podmínek (samozřejmě je-li to vůbec možné) psát jeden zdrojový kód pro různé systémy. V takovém případě musíme ale zajistit, aby se při kompilaci pro nějakou platformu vzalo ze zdrojového kódu jenom to, co je sama schopna využít. Jinými slovy, když vařím kuře na houbách pro dva lidi a vím, že jeden nejí kuřecí maso a druhý nejí houby a já jsem líný na to, abych vařil dva různé obědy (a zároveň mám chuť na již zmiňované kuřátko), tak budu očekávat, že při požívání jídla sní každý jenom to, co mu chutná. Podobně je to i mikrokontrolérů. Základ kódu (např. matematické operace pro výpočet filtru) je pro obě platformy stejný, liší se jenom způsobem komunikace s okolím a periferiemi (AVRka budou mít jistě jiné funkce pro práci s UARTem než PICy). Na druhou stranu, někdy je rychlejší a méně problematické napsat dva různé kódy, avšak s touto možností momentálně nebudeme počítat. Jak tedy zajistit, aby se při výsledné kompilaci pro jeden ze systémů vzali jenom houby? Odpovědí je podmíněný překlad, který se postará o to, že v závislosti na podmínce bude přeložena pouze určitá část kódu. Obecné schéma podmíněného překladu:

```
#if podmínka
   část kódu, která se přeloží, je-li podmínka TRUE;      //Větev 1
#else
   část kódu, která se přeloží, je-li podmínka FALSE;     //Větev 2
#endif                                                    //Ukončení podmíněného překladu
                                                          /pro tento úsek
```

Nepřipomíná vám to něco? No samozřejmě že podmíněný příkaz If – else. I zde tedy můžeme využít logické operátory jako jsou &&, ||, == atd. Důležitější je spíše to, co může být operandem takové podmínky. Jako operand lze totiž zvolit pouze symbolickou konstantu, takže nelze testovat kupříkladu proměnnou (hodnota operandu musí být známá v době překladu).

Vedle již známých logických operátorů se ještě zavádí operátor defined(KONSTANTA), jehož hodnota je TRUE v případě, že je KONSTANTA definována (pomocí #define) a FALSE v opačné situaci. Příklad složitější podmínky:

```
#define PIC  1
#if defined(PIC) && (PIC == 1)
   část kódu, která se přeloží, je-li definována konstanta PIC a má hodnotu 1;
#else
   část kódu, která se přeloží při nesplnění podmínky (není definován PIC nebo má jinou
   hodnotu něž 1);
#endif
```

V tomto případě je podmínka splněna, takže se přeloží první větev.

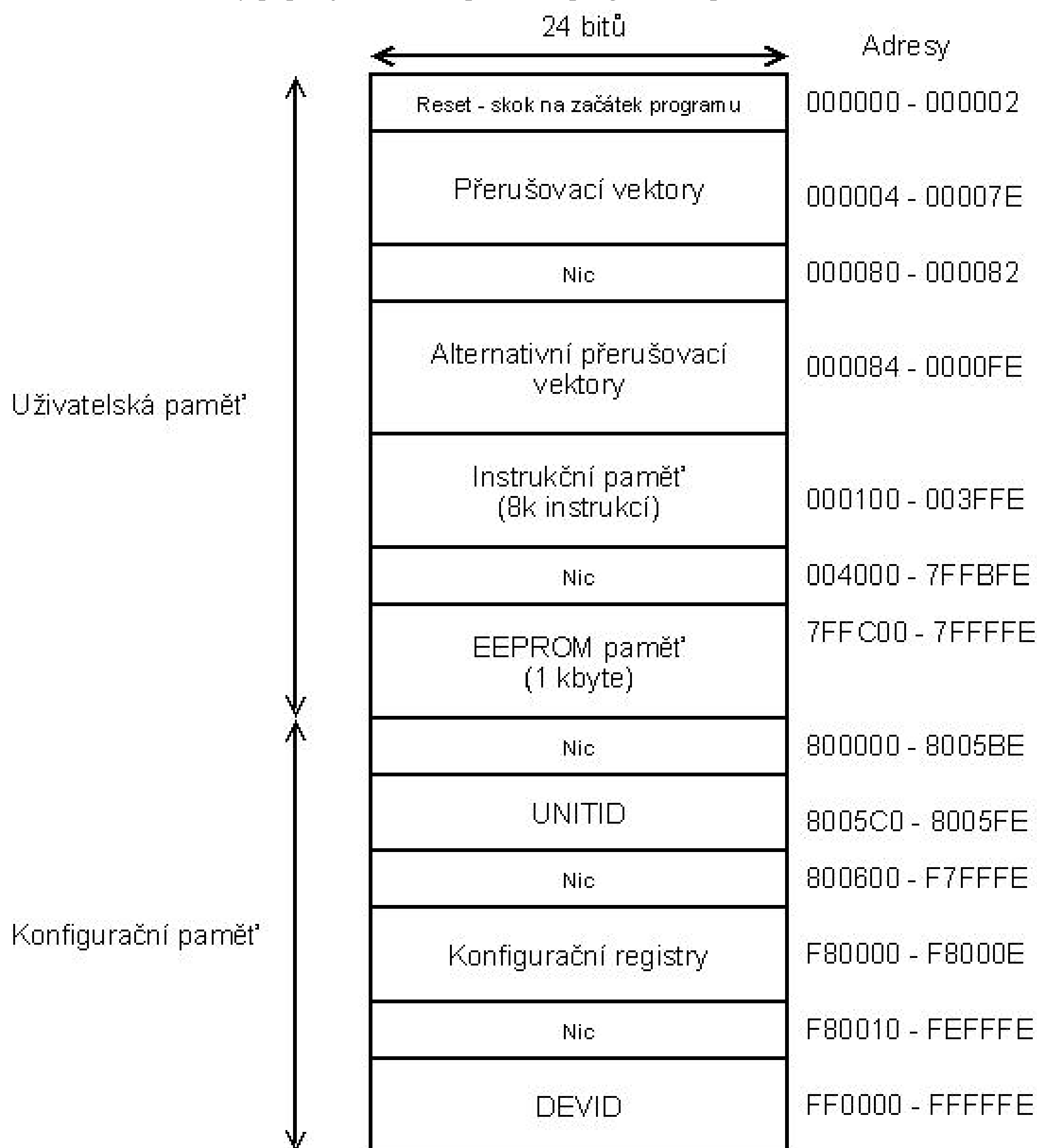
S podmíněným překladem souvisí ještě dvě konstrukce a to #undef („dedefinování" makra, samozřejmě že od místa použití #undef) a #elif (náhrada za příkaz else if, čímž se dají podmíněné překlady opravdu náramně větvit).

•10.1 Paměťový model – pohled dsPICa

Jak již určitě víme, dsPIC má paměť založenou na Harvardské architektuře, což znamená, že paměťový prostor pro programovou (instrukční) a datovou část je rozdělen do dvou samostatných sekcí (na rozdíl od von Neumannovy koncepce, která má obě paměti v jednom prostoru). Díky tomu můžou obě části přistupovat ke své paměti různým způsobem a komunikace mezi nimi probíhá pomocí sběrnice. Podívejme se tedy, jak vypadá programová a datová paměť.

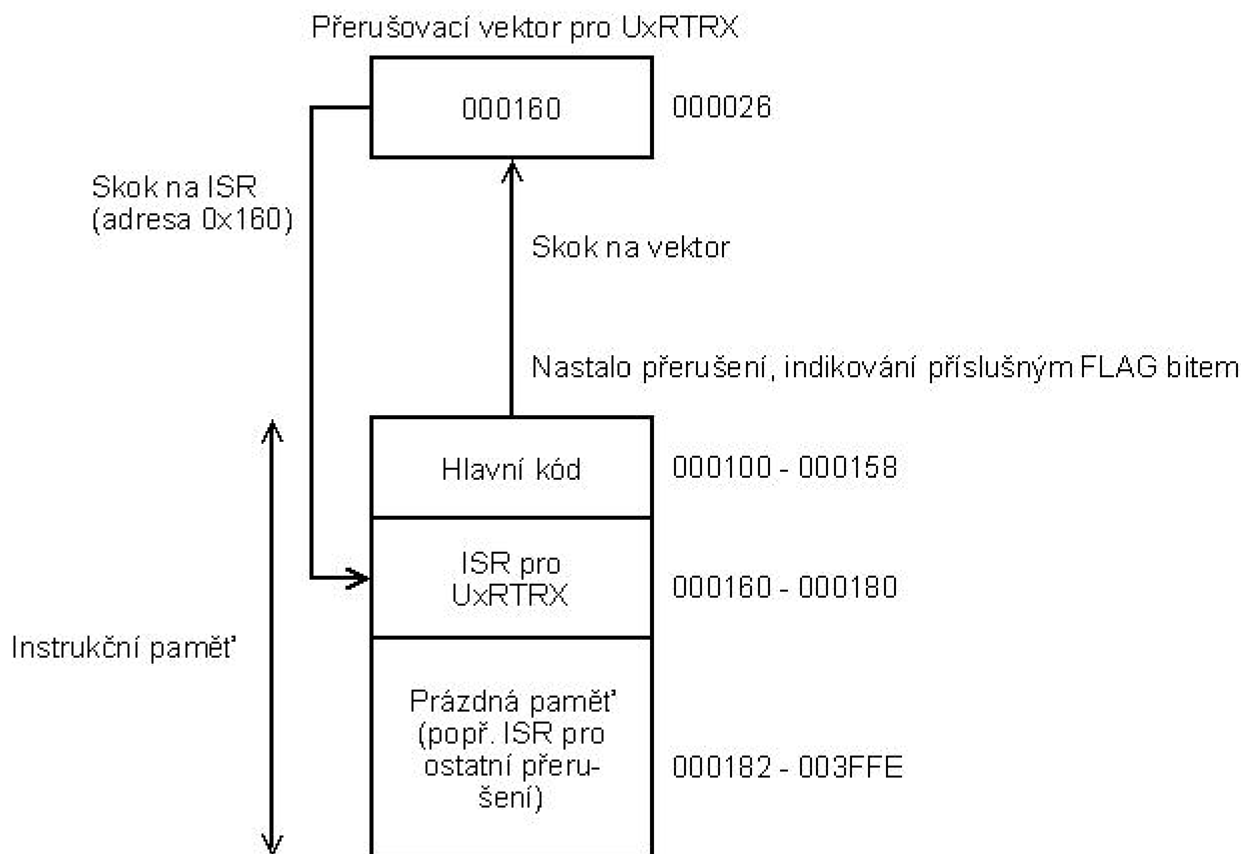
●Programová paměť

Nejprve si ukážeme obrázek, který popisuje vnitřní uspořádání programové paměti:



Jak je vidět, je šířka paměťové sběrnice (počet bitů na jeden registr) 24 bitů. Adresy jednotlivých registrů okupují POUZE sudé adresy (př. 0x0, 0x2, 0x4...), takže nelze přistupovat k lichým adresám (ve skutečnosti jsou na lichých adresách horní bity 24-bitového slova, takže bity <23:16>).
Na prvních dvou adresách (čti na adresách 0x0 – 0x3, protože každý registr leží na dvou buňkách) je reset, který skáče na začátek programu (tento začátek jsme v assembleru definovali a abychom neplýtvali pamětí, bývá umístěn na adrese 0x100, což je počátek instrukční paměti, v Céčku se tím již nezabýváme). Následují tzv. přerušovací vektory, na které se skočí při přerušení, takže když třeba přijde po UARTu nějaký znak, nastane přerušení (za předpokladu, že je povoleno) a skočí se na přerušovací vektor UxRTRX, kde leží další skok tentokrát již na přerušovací rutinu (tedy

kód pro přerušení nazývaný Interrupt Service Routine - ISR). Tuto situaci dobře ilustruje následující obrázek (adresa přerušovací rutiny a vektoru pro UxRTRX nemusí být stejná, jako je na obrázku):



Přerušovací vektory tedy obsahují skoky na adresy, kde začínají jejich ISR. Po skončení přerušovací rutiny se bude zase pokračovat v hlavním kódu.
Alternativní přerušovací vektory nás nezajímají (používají se při debuggingu), takže následuje instrukční paměť (jejíž délka je pro dsPIC 30F3013 8k instrukcí). Ta obsahuje jak hlavní kód (funkce main a definice ostatních funkcí), tak ISR pro jednotlivá přerušení, konstanty atd. Další sekce obsahuje EEPROM („datová" paměť, ve které zůstanou data i po vypojení napájení). Zbytek je konfigurační paměť, o které se ale nebudu rozepisovat.

Nyní vyvstává otázka, jak se dostat k informacím, které jsou uloženy v programové paměti (třeba z důvodu čtení již zmíněných konstant nebo přepisování jednotlivých instrukcí – princip bootloaderu). Možnosti jsou dvě, buď pomocí tzv. Table instrukcí nebo Program Space Visibility (PSV). Každá přistupuje k tomuto problému trochu jiným způsobem, mají jiné výhody a použití, proto si je popíšeme zvlášť.

1Program Space Visibility

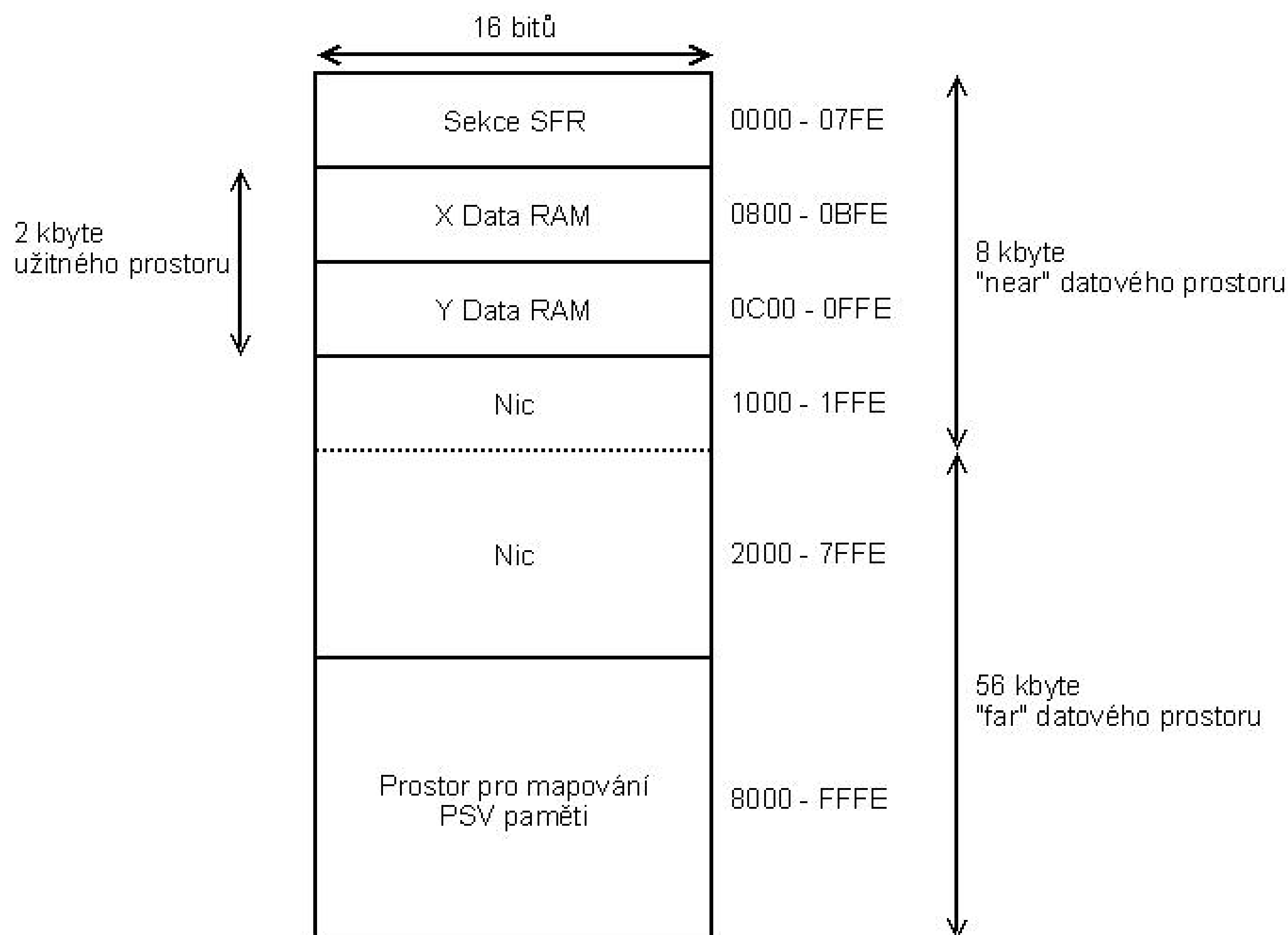
Použijeme-li tento způsob, jsme schopni přečíst (tedy nikoliv zapisovat) z jedné adresy pouze dolních 16 bitů (což jsou užitná data, horních 8 bitů bývají instrukce, popř. význam nižších bitů) a to pouze z instrukční paměti. Výhodou je relativně snadný přístup k těmto datům (u assembleru je to otázka nastavení pár registrů a u Céčka ani nepoznáme, že něco jako PSV existuje, prostě daná data čteme pomocí normálních konstrukcí). Princip spočívá v „mapování" určité části datové paměti (konkrétně adres 0x8000 až 0xFFFE) do programové paměti, kde se nacházejí PSV data, ze které si je poté pomocí nepřímého adresování vybíráme k dalšímu zpracování. Do PSV paměti se ukládají třeba řetězce, vzorky pro převodníky a tak nějak všechno, co má v programu význam konstanty (nezaměňujte tyto konstanty s makry, neboť makra se do žádné paměti neukládají, nahrazují se svými hodnotami při preprocesingu).

1Table instrukce

Na rozdíl od PSV jsme schopni číst (a zapisovat) všech 24 bitů (navíc nejsme limitováni pouze instrukční pamětí, ale můžeme se pohybovat i v konfigurační sekci). Nevýhoda je použití speciálních table instrukcí, takže nelze použít

céčkové přístupové metody (je nutné „přepnout“ do assembleru). Tento způsob se používá především pro přístup ke „zkomprimovaným“ datům (jako užitná data mohou sloužit všechny bity 24-bitové sběrnice), přepis instrukcí (již zmíněný princip bootloaderu) a ke změně konfiguračních hodnot.

●Datová paměť

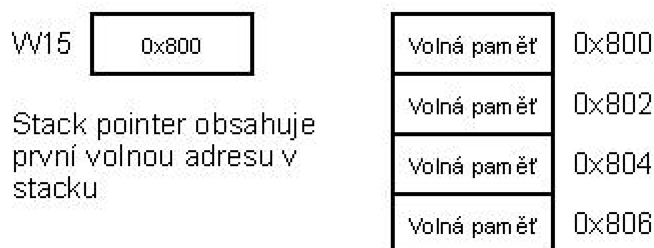


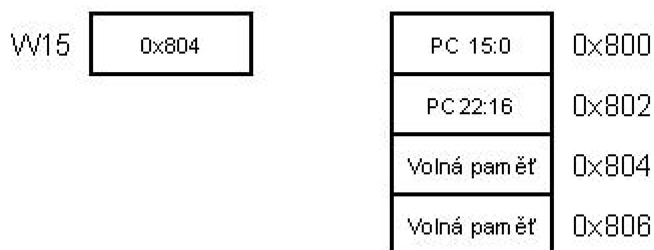
Nejdříve obrázek:

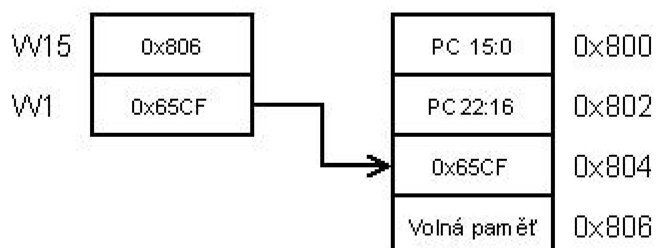
Oproti programové má datová paměť délku jedné adresy 16 bitů. K jednotlivým slovům (word – 16 bitů) se zase přistupuje pouze po sudých adresách s tím, že dolní byte (obsahuje LSB) je na sudé adrese a horní byte (MSB) na adrese liché.
První část datové paměti je sekce SFR, která obsahuje veškeré nastavovací a indikační registry, jenž ovládají periferie a procesor (v tomto prostoru tedy najdete kupříkladu registr pro ovládání čítačů nebo SPI).
Následují 2 kbyty paměti, do které si můžete ukládat prakticky co chcete (proto ji nazývám užitným prostorem).
Nebyl by to ale dsPIC, aby v tom nebyl nějaký háček. Tato část je rozdělena (jak je vidět z obrázku) na X a Y RAM.
Pokud jsme v assembleru používali pouze klasické instrukce (tedy ne takové, které vyžadovali použití DSP jádra), bylo možné k tomuto prostoru přistupovat jako k „jednolité“ paměti, takže jsme se nějakým rozdělením na XY vůbec nezabývali, ale při použití DSP instrukcí byla každá část ovládána samostatně, což naštěstí u Céčka vůbec neřešíme, do paměti si přistupujeme jak je nám libo (o správný přístup se nám stará kompiler). Kromě tohoto rozdělení je toto malé skromné místo bydlištěm pro Stack, který si za chvíli popíšeme.
Spolu s „malým nic“ nám XY RAM a SFR sekcí tvoří tzv. near paměť (není to nic významného, je to pouze pojmenování určitého úseku paměti).
Pod „velkým nic“ se nachází prostor pro mapování PSV paměti (tyto dvě části vytvářejí far paměť). Pokud tedy povolíme PSV, budeme moct přistupovat ke konstantám uloženým v programová paměti právě pomocí této sekce

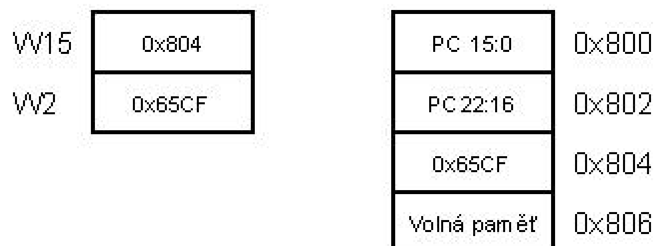
Stack

Pokud jste programovali se starými PICy (16Fxxx a jejich ekvivalenty), určitě si vzpomenete na 8-mi úrovňový (často proklínané číslo aneb „proč to sakra nemůže mít aspoň o úroveň víc?!“) stack (zásobník). Byla to samostatná paměť, do které se ukládaly jednotlivé návratové adresy (hodnoty instrukčního čítače) při volání podprogramů (pomocí instrukce call) nebo při přerušení, aby se měl procesor po vykonání určité činnosti kam vracet. Tím ale veškeré použití stacku končilo, nebylo třeba možné do něj cpát nějaká data.
S příchodem nových, nablýskaných 16-bitových mikrokontrolérů dostal stack novou podobu. Kromě toho, že do něj můžeme vkládat i vlastní data (pomocí instrukcí push a pop), lze jeho velikost vlastnoručně měnit.
Stack pracuje registrem W15, což je tzv. Stack Pointer. Jeho činnost spočívá v tom, že ukazuje na následující volné místo v paměti stacku (to znamená, že v registru W15 je uložena adresa této volné buňky, po resetu by měl ukazovat na adresu 0x800) a díky tomu, že přístup k jednotlivým adresám je pouze po sudých číslech, je hodnota 0-tého bitu v stack pointeru vždy v nule. Stack Pointer „roste“ a „klesá“ dle toho, jestli do něj zapisujeme (push), nebo čteme (pop).
Pro názornost předkládám následující obrázky:







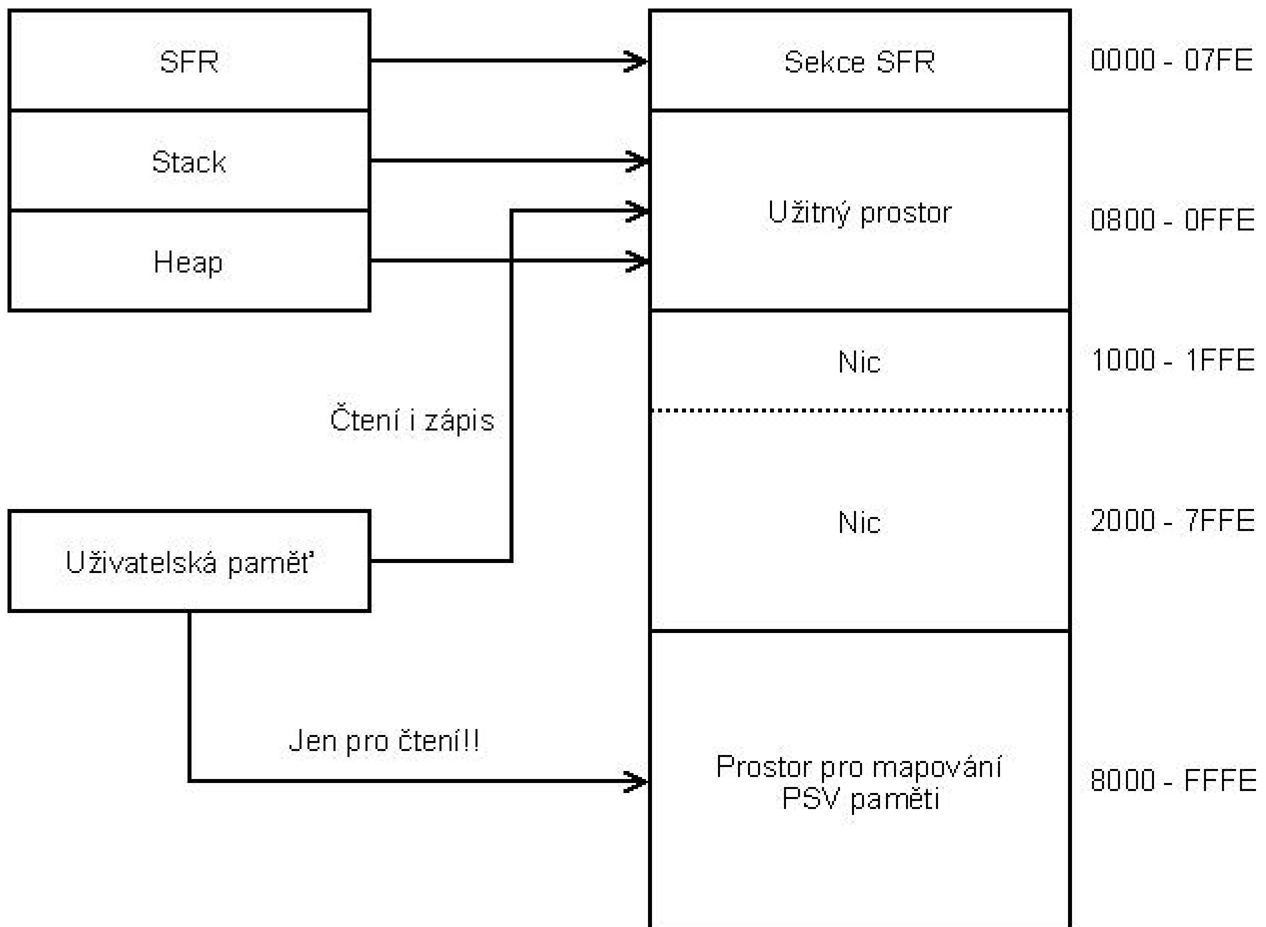


U předposledních dvou obrázků může vyvstat otázka: jak je možné, že W15 ukazuje na paměť 0x806 (obrázek č. 3), ale do W2 se uloží obsah adresy 0x804 (obrázek č. 4)? Je to proto, že procesor předpokládá, že stack pointer ukazuje vždy na volné místo (to, že tam může být nějaké číslo, jej nijak netrápí, proto v obrázku č. 4 stack pointer ukazuje na adresu 0x804, přestože tam je číslo 0x65CF), takže za první užitná data (pro stack) považuje obsah adresy, která je o buňku níže (tedy 0x804). Při dalším čtení (pop) by se tedy vzala hodnota z adresy 0x802, nikoliv z 0x804, na kterou ukazuje W15. Tento princip je prakticky popis LIFO (Last in, First out) bufferu. K W15 ještě malou připomínku: je to registr jako každý jiný, takže z něj lze číst a zapisovat dle libosti (můžeme tedy nastavit stack od jiné adresy, než je 0x800, s čímž je ale spojen problém podtečení stacku).
Jak jsem zmínil výše, je možné velikost stacku měnit, což zajišťujeme pomocí registru SPLIM. Jeho hodnota udává, kam až může stack ukládat jednotlivá data (adresa v SPLIM je poslední použitelnou buňkou pro stack). Příklad: Stack pointer je nastaven na hodnotu 0x800 a SPLIM na 0x816, takže do stacku můžeme uložit maximálně 12 16-bitových slov (počítáno hexadecimálně: (0x816 – 0x800)/2 + 1) = C, což je 12 dekadicky), takže adresa 0x818 již stacku nepřísluší.
Se stackem ještě souvisí jedna věc, a to je přetečení a podtečení. Stack přeteče v tu chvíli, když bude hodnota v stack pointeru větší alespoň o 2 než v SPLIM (v našem příkladě tedy ve W15 budeme ukazovat na 0x818). Podtečení naopak nastane, když bude hodnota ve W15 menší jak 0x800 (proto se definuje stack od adresy 0x800, abychom věděli, kdy nastalo podtečení). U obou případů se skočí na tzv. Stack Error Trap (prakticky přerušení, trap pouze značí, že někde nastala chyba), ve kterém je nutné tyto problémy nějak řešit.

•10.2 Paměťový model – pohled Céčka

Teď, když víme, jak vypadá paměť v našem procesoru, vyvstává otázka, jak se k ní bude přistupovat pomocí jazyka C. Vzhledem k tomu, že jazyk C je přenositelný mezi různými platformami (viz. kapitola o preprocesoru), bylo zapotřebí vytvořit navenek jakousi "unifikovanou" představu o paměti, aby se programátor, který píše v Céčku, nemusel zabývat tím, jak vypadá paměť uvnitř jednotlivých procesorů. V praxi to tedy vypadá tak, že píšeme software pro tuto "unifikovanou" paměť a kompiler se poté postará o správnou alokaci (přidělení) paměti. "Céčková" paměť se dá znázornit následujícím obrázkem:

Tato paměť nemá přesně stanovené adresy, neboť velikost jednotlivých bloků závisí pouze na stavbě paměti uvnitř mikroprocesoru. Aplikujeme-li tento model na náš procesor, tak zjistíme, že prakticky odpovídá datové paměti (programovou paměť jako takovou obecný model jazyka C nezná, ale náš kompiler C30 je schopen za určitých okolností uživatelskou paměť rozšířit i na tu programovou - PSV). Tuto skutečnost nejlépe vystihuje následující obrázek:



Ještě před tím, než si rozepíšeme jednotlivé bloky, slušelo by se říci něco o oblasti platnosti identifikátorů a modifikátorech paměťových tříd.

●Oblast platnosti identifikátorů

Jak již asi tušíte, jednotlivým objektům můžeme pomocí tohoto rozdělení říci "ty budeš viditelný odcamcaď pocamcaď a ty budeš vidět všude". Objekty, které jsou viditelné všude (s jednou výjimkou) se nazývají globální a těm, které vidíme pouze v některých situacích říkáme lokální.

1Globální objekty

Definují se vně jakékoliv funkce (tedy i mimo funkci main, prostě jakoby do "volného prostoru"), což má za následek, že s tímto objektem můžeme pracovat v jakékoliv funkci. Příklad definice:

```
#include <p30fxxxx.h>

int globalni_prom;   //Definice globální proměnné typu int

int main(void)
{
   while (1)          //Nekonečný cyklus
      ;
}
```

Takováto proměnná se alokuje do uživatelské paměti (tam končí všechny globální objekty) při spuštění programu a bude tam sídlit do doby, než program skončí (vzhledem k tomu, že programy u mikroprocesorů mívají tendenci obsahovat nekonečný cyklus, tak paměťové místo této proměnné bude uvolněno až po restartu či vypnutí mikroprocesoru).

1Lokální objekty
Definice lokálního objektu je vždy uvnitř funkce, příklad:
```
#include <p30fxxxx.h>

int main(void)
{
   int lokalni_prom;  //Definice lokální proměnné typu int

   while (1)          //Nekonečný cyklus
      ;
}
```

S lokálními objekty můžeme pracovat pouze uvnitř dané funkce, ve které jsme je definovali (v našem příkladě s proměnnou lokalni_prom si můžeme hrát pouze ve funkci main), jinde ne. Tyto objekty se nevytváří při spuštění programu, ale při vstupu do funkce, kde jsou definovány. Následně se jim přidělí paměťové místo ve stacku a při opuštění funkce je jejich paměťový prostor uvolněn, neexistují tedy po celou dobu běhu programu.
Je velice důležité si zapamatovat, že definujeme-li třeba dvě proměnné typu int, kdy jedna je globální a druhá lokální a ještě je definujeme se stejným jménem, tak můžou nastat dvě situace (v tomto příkladě jsem si definoval svoji funkci, což ještě neumíte, takže vám prozatím stačí pouze vědět, že příkazem fce(); skáču do funkce fce. Prostě jako kdyby tam bylo call):
```
#include <p30fxxxx.h>

int i;                //Definice globální proměnné typu int

void fce(void)
{
   int i;             //Definice lokální proměnné typu int

   i++;               //Přičtení 1 k lokální proměnné, hodnota v i je neznámá, neboť i nebylo
                      //inicializováno
   return;            //Vracení se do funkce main do místa, kde jsme volali funkci fce
                      //Stack se začíná vyprazdňovat, což má za následek "zničení" lokální
                      //proměnné
}

int main(void)
{

   i = 3;             //Přiřazení hodnoty 3 do globální proměnné
   fce();             //Volání funkce fce

   i++;               //Přičtení 1 ke globální proměnné, i se tedy rovná 4
   while (1)          //Nekonečný cyklus
      ;
}
```

Jak je z příkladu vidět, lokální proměnná ve funkci fce nám "zastínila" globální proměnnou (samozřejmě že kdybychom ve funkci fce nedefinovali lokální proměnnou i, tak by se jednička přičetla ke globální proměnné. Hodnota v i by tedy byla 5 před vstupem do nekonečného cyklu). Ve funkci main zase nevidíme lokální a pracujeme pouze s tou globální.

●Modifikátory paměťových tříd
Kromě toho, že může být objekt globální či lokální, tak může obsahovat také modifikátor paměťové třídy.

1Modifikátor static
Určitě si pamatujete, že lokální objekt se alokuje (do stacku) při vstupu do funkce a při opuštění se zase dealokuje (to znamená, že se jeho obsah "zničí" a při novém volání funkce bude mít tento objekt jinou hodnotu). Tento modifikátor zajistí to, aby byl objekt alokován při spuštění programu a poté setrval v "životaschopném" stavu po dobu běhu programu. Takový objekt se již nemůžeme nacházet ve stacku (ačkoliv je to lokální objekt a ty se ukládají do stacku), ale pouze v uživatelské paměti (tím zajistíme jeho "nesmrtelnost"). Uvedu příklad:
```
#include <p30fxxxx.h>
```

```
void fce(void)
{
   static int i = 0;  //Definice statické lokální proměnné typu int a její inicializace

   i++;               //Přičtení 1 do proměnné i
   return;            //Vracení se do funkce main do místa, kde jsme volali funkci fce
}


int main(void)
{

   while (1)          //Nekonečný cyklus, který bude neustále volat funkci fce
      fce();          //Volání funkce fce
}
```

Proměnná i se bude neustále zvětšovat o 1 a svoji hodnotu nebude zapomínat, protože je definována s modifikátorem static (jinak má tato proměnná podobné vlastnosti jako ostatní lokální proměnné, tedy že ji nevidíme v jiných funkcích než v té, kde byla definována).

1Modifikátor const
Jak už název napovídá, definujeme tím konstantní objekty, což znamená, že jeho hodnota je dána inicializací a tu si ponechá po celou dobu. Hodnotu takového objektu nemůžeme měnit, ale pouze číst (měnit lze pouze s pomocí pointeru, což bohužel neplatí u našeho C30 kompileru). U PC kompilerů se většinou tyto objekty nacházejí v uživatelské paměti, C30 ale konstantní objekty umisťuje do PSV kvůli šetření s drahocennou datovou pamětí (nemusíte mít strach, že byste museli používat speciální instrukce, kompiler vše zařídí za vás).

1Modifikátor volatile
Označuje objekt, který může být ovlivněn nějakou asynchronní událostí (třeba přerušením), což ale neznamená, že když nebude proměnná označkována tímto modifikátorem, tak že ji nebudete moct v přerušení měnit (abych se přiznal, tak opravdový význam tohoto modifikátoru mi tak trochu uniká, ale pravdou zůstává, že všechny objekty definované v hlavičkových souborech od Microchipu tento modifikátor obsahují).
1Modifikátor registr
U mikročipů naprosto zbytečný modifikátor, neboť umisťuje dané objekty do registrů procesoru. U PC kompilerů je tato vlastnost docela užitečná (zrychluje práci s objekty), ale u našeho je k ničemu, protože VŠECHNY objekty se nacházejí v registrech procesoru (jenom zaměňuji slovo registr za paměť)

No a teď si už konečně rozepíšeme obecnou céčkovou paměť (není moc o čem se rozepisovat, téměř vše důležité bylo již zmíněno v předchozích odstavcích):

SFR
Název mluví za vše, tato část tedy obsahuje veškeré registry procesoru. Tyto registry se chovají jako proměnné (popř. jako bitové struktury, pokud chceme přistupovat k jednotlivým bitům), takže je možné z nich číst a zapisovat běžným způsobem, ovšem na to, abychom mohli k tomu bloku přistupovat jako k SFR paměti, je nutné "natáhnout" definiční soubor p30fxxxx.h (pokud bychom to neudělali, kompiler by tuto část paměti považoval za obyčejnou uživatelskou paměť, což by vedlo ke změně chování procesoru. Tato skutečnost ale neplatí u klasických PC kompilerů, protože tam žádná SFR paměť není!!).

Uživatelská paměť
Tato paměť obsahuje veškeré globální a statické objekty. Jejich alokace se provede při spuštění programu a přestanou existovat až po jeho skončení (máte tedy jistotu, že je tam budete mít pořád).

Stack

Stack jsme si již rozebrali v kapitole 10.1 a prakticky vše, co jsme o něm řekli, platí i pro céčkový stack. Stack je tedy paměť, která dynamicky "roste a klesá" v závislosti na tom, jestli do ní něco ukládáme. Obsahuje návratové adresy, lokální proměnné, parametry pro funkce (viz v další kapitole) a ještě spoustu dalšího.

Heap

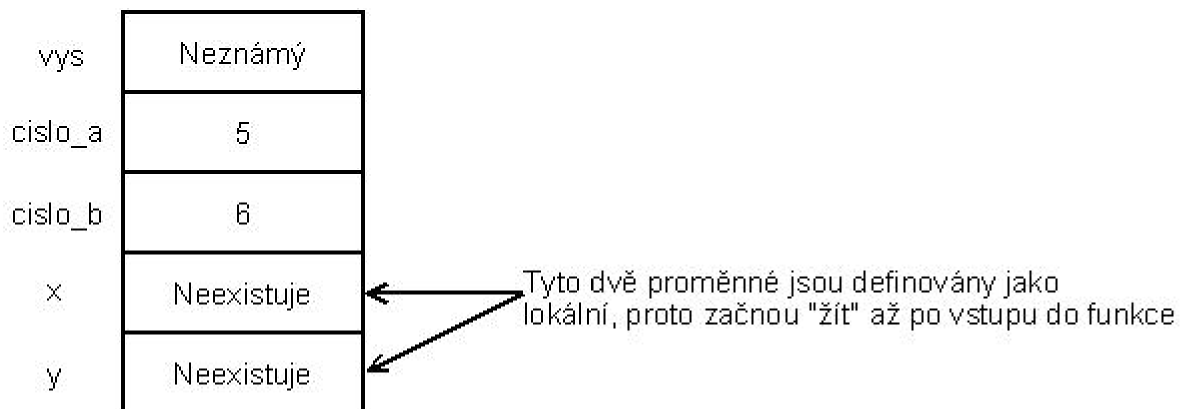
Je to ode mě docela neslušné, že jsem se zatím vůbec nezmínil o způsobech alokace paměti, proto to teď v rychlosti napravím. Paměť můžeme objektům přidělit dvěma způsoby a to staticky a dynamicky. Statické přidělení se dá vyjádřit jako "dostal jsem své místečko při spuštění programu a do jeho konce jej neopustím" (statické objekty se nacházejí v uživatelské paměti), kdežto u dynamického přidělení nemají objekty přesně dané adresy. Stack je speciálním případem dynamické paměti, protože se o něj stará kompiler (ačkoliv nemá pevně vázané objekty). Skutečná dynamická paměť je tedy heap, který by se dal přirovnat k pískovišti (prázdný paměťový prostor o předem dané velikosti a nedefinovaného typu), na kterém můžeme libovolně stavět bábovičky a hrady (tím myslím jakékoliv objekty) a následně je bourat (a uvolnit tento prostor pro případný další objekt). Pro práci s heapem je zde soubor standardních funkcí, o kterých si povíme někdy v dalších kapitolách (práce s heapem je také poměrně riziková pro začátečníky, kteří tu a tam zapomenou dealokovat nějakou paměť a v kombinaci s pointery se jedná doslova o smrtící zbraň, protože kompiler Vámi prováděné kroky vůbec nekoriguje, zde jste pouze svými pány)

•11.1 Funkce

Základní myšlenka funkcí je taková, že funkce dostane na vstupu nějaký parametr (nebo nemusí dostat nic), což bývá nějaký objekt, se kterým si funkce "něco udělá" a pokud budeme chtít, tak nám taky tento výsledek vrátí jako návratovou hodnotu (funkce ale nemusí nic vracet, většinou ale vrací buď výsledek operace, nebo alespoň příznak, byla-li funkce provedena úspěšně či nikoliv). Jako nejednodušší funkci si můžeme představit součet dvou čísel, kde výsledek je návratová hodnota. Pojďme si takovou funkci napsat a posléze podrobně rozebrat:

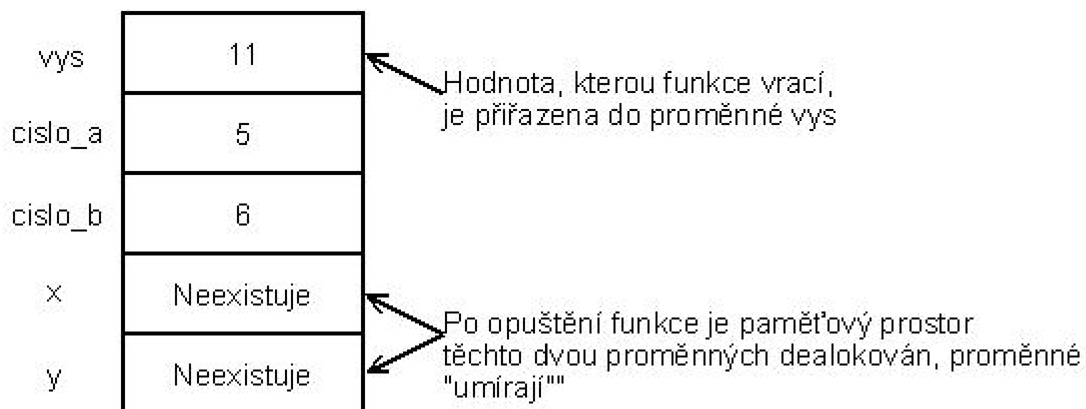
```
int soucet(int x, int y)
{
   return (x + y);
}

int main(void)
{
   int vys, cislo_a = 5, cislo_b = 6;

   vys = soucet(cislo_a, cislo_b);
   while (1)
      ;
}
```

Prvním čtyřem řádkům říkáme definice funkce a skládá se z hlavičky (první řádek) a kódu (vše v bloku). Hlavička obsahuje datový typ návratové hodnoty (to je ten první int), název funkce (soucet) a parametry (obsah závorky za názvem, někdy se jim také říká argumenty). Vracíme-li nějakou hodnotu, musí funkce obsahovat příkaz return, který tuto hodnotu vrací (sčítáme-li tedy dvě celá čísla, musí být návratový typ definován na int nebo jiný celočíselný ekvivalent, protože jinak by došlo ke konverzi čísla a tudíž ke ztrátě informace). Parametry funkce jsou ale trošku komplikovanější záležitost, proto se podívejme, co se stane při zavolání funkce (příkazem soucet(cislo_a, cislo_b);) a při jejím průchodem:



Zatím je snad všechno jasné, cislo_a i cislo_b byly inicializovány na své hodnoty a proměnné x y zatím neexistují (protože jsou to lokální proměnné).

Při zavolání funkce soucet se do stacku uloží nejenom návratová adresa, ale také se v něm vytvoří prostor pro proměnné x a y (obě o velikosti jednoho intu, což jsme definovali v hlavičce). Zatím mají tyto proměnné neznámé hodnoty (byly pouze vytvořeny, ne inicializovány). Pak ale nastane jev, který se nazývá předání hodnoty, kdy hodnoty vstupních parametrů (proměnných cislo_a a cislo_b) jsou doslova předány do proměnných x a y (cislo_a do x a cislo_b do y). To, že se předávají pouze hodnoty, je velmi důležité si zapamatovat. Proměnné x a y jsou naprosto nezávislé na proměnných cislo_a a cislo_b, mají pouze společnou počáteční hodnotu, která se ale může v průběhu funkce změnit (to znamená, že změna v proměnné x se neprojeví v cislo_a). Teď nám nic nestojí v cestě, abychom spočítali součet (výraz (x + y)) a jeho hodnotu vrátili jako návratovou hodnotu (pomocí příkazu return):



Jakmile funkci opouštíme, lokální proměnné zanikají a vracíme se do místa, odkud jsme do funkce skočili (což lze, protože do stacku jsme si uložili návratovou adresu). Pokud vracíme hodnotu, je nutné si ji třeba někam uložit (v našem příkladě do proměnné vys) nebo ji třeba testovat na rovnost apod.

Ve skutečnosti je předávání parametrů trošičku složitější, neboť se všechno děje přes work registry. Každý parametr je uložen do samostatného work registru (pokud je nějaký parametr jiného typu než int, tak se jeho hodnota rozdělí do více registrů) a až poté jsou jejich hodnoty předány lokálním proměnným ve stacku. Návratová hodnota se vždy vrací pomocí registru W0 (takže se do stacku vůbec neuloží). Zmiňuji se zde o tom z toho důvodu, že pokud si před zavoláním funkce něco uložíte do W registrů, tak je zde téměř 100% jistota, že se s danými daty můžete rozloučit, protože zde není zajištěna záloha a následná obnova těchto registrů (na rozdíl od přerušení, kde jsou některé registry zálohovány). Celkově proto není doporučováno užívat těchto registrů, neboť jsou velice často využívány jako pomocné registry kompilerem.

●Datový typ Void

Void v angličtině znamená prázdný prostor a přesně to je také jeho významem, tedy prázdný typ. Použití toho datového typu je hlavně u funkcí a u dynamického přidělení paměti (nelze mít třeba proměnnou void-ového typu). Pokud máme např. funkci, která nevrací žádnou hodnotu, definujeme její hlavičku takto:

```
void funkce(parametry)
```

Podobně je to s funkcí, které nepředáváme žádné parametry:

```
datový_typ_návratové_hodnoty funkce(void)
```

Takovou funkci poté v programu voláme příkazem funkce();

Příklad 11 - 01:

Tento program vyšle každou vteřinu po sériové sběrnici jeden znak rychlostí 9600 Bd. Tento znak získáme z PORTB, tedy z přepínačů. Jako parametr funkce je předán onen znak k vyslání (protože nám naše funkce nic nevrací, má její návratová hodnota typ void)

```
#include <p30fxxxx.h>
#define FCY        11059200              //Frekvence procesoru
#define BRATE      9600                  //Požadovaná přenosová rychlost v Baudech
#define RYCHLOST   (FCY / 16 / BRATE) - 1  //Výpočet konstanty

//Funkce, která vyšle hodnotu parametru
//po sériové sběrnici
void vysli_znak(unsigned char znak)
{

   U2TXREG = znak;                     //Tímto jsme odstartovali přenos
   return;                             //Návrat do hlavní funkce
}

int main(void)
{

   ADCON1bits.ADON = 0;                //Vypne AD převodník
   ADPCFG = 0xFFFF;                    //Všechny piny digitální
   TRISB = 0xFFFF;                     //Nastavení směru jednotlivých pinů
   U2BRG = RYCHLOST;                   //Přiřazení konstanty
   U2MODE = 0x8000;                    //Povolení UARTu, 8 - bitové znaky
                                       //bez parity
   U2STA = 0x0400;                     //Zapnutí UART vysílače
   TMR1 = 0;                           //Nulování čítače
   T1CON = 0x8030;                     //Spustí čítač 1, předdělička 256

   //Nekonečný cyklus
   while (1)
   {
      while (TMR1 < 43402)             //Čekáme, než uplyne 1 vteřina
         ;

      TMR1 = 0;                        //Nulování čítače
      vysli_znak(PORTB & 0x00FF);      //Tímto příkazem předáme hodnotu
                                       //na PORTB (po vymaskování) jako
                                       //parametr funkce
   }
}
```

Jumpery:
Jumper 1 - 8 = Přepínače

Příklad 11 - 02:

Na rozdíl od předchozího programu, kde jsme znaky vysílali, tak zde budeme znaky přijímat a zobrazovat je na LED diodách (rychlost je pořád 9600 Bd). V hlavním kódu voláme funkci, která čeká na příjem znaku (příjem znaku zjišťujeme tak, že neustále oťukáváme bit URXDA, který říká, je-li buffer prázdný či nikoliv) a tento přijatý znak vracíme (funkci tentokrát žádný parametr nepředáváme, proto bude v závorce void).

```
#include <p30fxxxx.h>
```

```
#define FCY       11059200                    //Frekvence procesoru
#define BRATE     9600                        //Požadovaná přenosová rychlost v Baudech
#define RYCHLOST  (FCY / 16 / BRATE) - 1      //Výpočet konstanty

//Funkce, která čeká na znak a následně
//jej vrátí jako svoji hodnotu
unsigned char prijmi_znak(void)
{

   //Zde čekáme, než se nastaví flag do 1,
   //což značí, že v bufferu je alespoň jeden znak
   while (!U2STAbits.URXDA)
      ;

   return U2RXREG;                   //Návrat do hlavní funkce a
                                     //vrácení přijatého znaku
}

int main(void)
{

   ADCON1bits.ADON = 0;              //Vypne AD převodník
   ADPCFG = 0xFFFF;                  //Všechny piny digitální
   TRISB = 0;                        //Nastavení směru jednotlivých pinů
   U2BRG = RYCHLOST;                 //Přiřazení konstanty
   U2MODE = 0x8000;                  //Povolení UARTu, 8 - bitové znaky
                                     //bez parity
   U2STA = 0;                        //Nulování U2STA (pro jistotu)

   //Nekonečný cyklus
   while (1)
   {
      PORTB = prijmi_znak();         //Zde voláme funkci prijmi_znak
                                     //a hodnotu, kterou nám vrátí,
                                     //přiřadíme do PORTB

   }
}
```

Jumpery:

Jumper 1 - 8 = Diody

●Definice funkce

Jeden způsob jsme si již ukázali a to je definování funkce před hlavní funkcí main. Pokud ale řekněme budete chtít svou definici vepsat někam pod funkci main (třeba kvůli přehlednosti), je nutné kompileru sdělit, jak vypadá hlavička vaší funkce (to znamená, že mu řeknete, jestli něco vrací, jaké má parametry atd.), aby ji správně volal (jinak by se mohlo stát, že by se některé parametry nesprávně konvertovaly). Toto "sdělení" se provádí funkčním prototypem, předvedu na obecném příkladě:

```
datový_typ_návratové_hodnoty funkce(parametry);   //Deklarace funkčního prototypu funkce

int main(void)
{
   /*Zdrojový kód funkce main*/
}

datový_typ_návratové_hodnoty funkce(parametry)
{
   /*Zdrojový kód funkce funkce*/
}
```

Všimněte si, že za deklarací funkčního prototypu je středník!

Existují různé zkrácené verze funkčního prototypu (třeba vynechání datových typů pro parametry a návratové hodnoty), ale důsledně doporučuji vypisovat vše, čímž zamezíte výskytu chyb.

●Rekurze

Vysvětlení pojmu rekurze ve Výkladovém slovníku výpočetní techniky: "Rekurze - viz rekurze".

Rekurze se používá u funkcí, které volají sebe sama. Důvodů může být mnoho, důležité ale je, aby ten důvod byl dobrý, protože rekurzivní volání funkce většinou zabere velké místo v zásobníku, kam se ukládají jednotlivé parametry a navíc je rekurze hrozný žrout času. U našich malých, slaboučkých mikroprocesorů je použití rekurze (třeba pro výpočet faktoriálu) opravdu na dlouho, proto bychom se ji měli snažit vyhnout (existuje pravidlo, že veškeré algoritmy, které obsahují rekurzi, se dají převést na cykly bez rekurze)

●Proměnný počet parametrů

Představte si situaci, že si chcete napsat funkci, která sčítá parametry a jejich výsledek vrací. Můžete vytvořit buď stálý počet parametrů (např. 4, takže funkce bude sčítat 4 čísla, které musí dostat na vstupu), nebo můžete použít speciálních konstrukcí, díky kterým můžete funkci předat teoreticky nekonečný počet čísel (prakticky je jejich počet omezen zásobníkem). Způsob, jakým se tohle dělá, je nad rámec této učebnice, proto zájemce odkazuji na odbornou literaturu.

●Standardní knihovny

Jakýsi standardizující útvar jménem ANSI *(Americký Národní Standardizační Institut)* se po dostatečném rozšíření jazyka C začal zajímat o jeho normalizaci, což znamená, že každý kompiler bude muset dodržovat určitá pravidla (třeba syntaxe jazyka, hlavičkové soubory atd.) a tyto normy začal vydávat (nejnovější norma se jmenuje C99). Byly vytvořeny jakési knihovny, např. pro práci s konzolí, řetězci, časem atd. Náš C30 kompiler tyto knihovny obsahuje také, ale jejich použitelnost pro embedded zařízení je spíše sporná. Vezměme si třeba knihovnu pro práci s terminálem stdio.h (umí také pracovat se soubory): s touto knihovnou lze na PC kompilerech tisknout znaky na terminál, číst klávesnici (neboli standardní výstup a vstup) nebo otvírat soubory. Jakmile tyto funkce použijeme v našem procesoru, narazíme na problémy. Jaký terminál? Jaká klávesnice? To, co za nás na PC odře operační systém, si musíme všechno udělat sami, takže by to nakonec vypadalo tak, že bychom tyto funkce přepisovali, aby věděli, kam mají daná data posílat nebo číst (prostě definovat, co je standardní vstup a výstup). Když už vyřešíte tyto problémy, tak zjistíte, že třeba funkce printf (formátovaný tisk na výstup, tedy na obrazovku) Vám sežere OHROMNÉ množství paměti. To, co se Vám pokouším říct je, aby jste se snažili vyhnout použití těchto knihoven (u PC kompilerů je samozřejmě použití těchto knihoven nutnost) a třeba si vytvořit knihovny vlastní, které budou lépe použitelné pro náš malý svět mikroprocesorů (hlavně si z toho proboha neberte poznatek, že všechny funkce ze standardních knihoven jsou na nic, musíte se jenom naučit rozlišovat, co je lepší si udělat sám a kdy použít standardní funkce)
Vedle normovaných knihoven také C30 kompiler přichází s vlastními knihovnami pro ovládání periferií (třeba LCD, čítačů, UARTu nebo také pro digitální filtraci a Fourierovu transformaci). Účelem těchto knihoven je snaha odprostit se od klasických nastavování SFR registrů a místo toho použít funkce, kterým předáte nastavení pro různé periferie. Osobně jsem tyto knihovny ještě moc nezkoumal (razím filosofii "naprogramuj si vše sám"), ale jejich použitelnost je na míle vzdálená (v dobrém slova smyslu) od standardních ANSI C knihoven pro PICy. Kdo má zájem, může začít studovat oficiální helpy Microchipu, kde je vše popsáno spolu s příklady a zdrojovými kódy jednotlivých funkcí.

•11.2 Přerušení

Přerušení je asynchronní (to znamená, že může přijít kdykoliv) činnost, na kterou je nutné reagovat vhodnými prostředky. Typickým přerušením může být přetečení čítače, přijetí znaku či jiná asynchronní činnost. Když tato událost nastane, bylo by vhodné ji ošetřit speciální procedurou, takže přetečený čítač vynulujeme, přijatý znak si někam uložíme a podobně. U starých 8-mi bitových PICů jsme měli pouze jednu přerušovací rutinu pro všechny zdroje přerušení, takže jsme na začátku této rutiny většinou testovali, od jakého zdroje přerušení přišlo a podle toho jsme skočili do různých podprogramů. Výhoda nových mikroprocesorů je použití tzv. přerušovacích vektorů (jak jsem již letmo zmínil v kapitole 10.1), které odkazují na místo v paměti, kde se nachází přerušovací rutina, neboli ISR (Interrupt Service Routine), takže pro každý zdroj přerušení musíme psát vlastní rutinu. Navíc rozlišujeme dva druhy přerušení, od periferií a tzv. trapy, na které se skočí, když nastane nějaká chyba (třeba dělení nulou, chyba oscilátoru, chyba zásobníku apod.). Aby ale přerušení mohlo vůbec nastat, je nutné jej povolit (na tento účel jsou registry IECn) a

pak už jenom čekáme na flagový bit (registry IEFn), dle jehož změny procesor skáče na jednotlivé vektory (tyto flagové bity je většinou nutné v ISR nulovat).
Ještě než přejdeme k definici ISR v céčku, bylo by záhodno si zde ještě uvést dvě věci, které s přerušeními souvisí:

●Priorita přerušení

Jednotlivé zdroje přerušení můžou mít přidělený různé priority v závislosti na tom, která z nich je momentálně důležitější. V praxi to funguje tak, že pokud v jednom momentu nastanou přerušení od různých zdrojů, tak dle priority, která jim byla přidělena, se vykonají postupně. Tato priorita je dána buď defaultně (viz datasheet) nebo programátorem (pomocí registrů IPCn, kde má každý zdroj vyhrazeny 3 bity a podle jejich hodnoty má různou prioritu - 7 nejvyšší a 1 nejmenší).

●Vnořená přerušení

Pokud tuto volbu povolíte (defaultně je povolena), tak je umožněno, aby přerušení nastalo i během vykonávání rutiny jiného přerušení (neumožníte-li toto procesoru, bude nejprve dokončeno dosavadní přerušení až poté se skočí do ISR druhého přerušení). Zde hraje velkou roli již zmíněná priorita, takže když do ISR s menší prioritou skočí jiné přerušení s prioritou větší, tak bude to s menší pozastaveno a začne se vykonávat ISR druhého přerušení. Chci Vás upozornit, že vnořená přerušení mohou být vskutku velké zlo, proto pokud to lze, snažíme se jim vyhýbat. Zapínání a vypínání vnořených přerušení je pomocí bitu NSTDIS v registru INTCON1.

No a teď se vrhneme na programování ISR.

Přerušovací rutiny jsou z hlediska jazyka C téměř normální funkce, rozdíl je v tom, že ISR funkce v céčku nemá žádné parametry a také nic nevrací. Samotné odlišení, které z ní udělá přerušovací rutinu, je použití atributu __interrupt__ (z obou stran je podtržítko dvakrát!). Atributy jako takové nejsou součástí normy ANSI C a jsou výmyslem C30 kompileru (atributy se dá přidělit ještě poměrně dost věcí, ale o tom možná někdy jindy). Obecná hlavička takové ISR funkce vypadá takto:

```
void __attribute__((__interrupt__, dodatečné_možnosti)) název_přerušení(void)
```

Nebojte, na nadefinování hlavičky existují i makra, ale ty nás teď nezajímají. Jak jsem již řekl, atributem interrupt řekneme kompileru, že daná funkce je ISR pro dané přerušení. Dodatečnými možnosti mám na mysli tyto (hlavička může obsahovat naráz několik těchto možností):

1Uložení proměnných

Velká výhoda při vstupu do přerušovací rutiny je to, že kompiler sám zajistí zálohu některých registrů a při vracení se jejich obnovu. Jmenovitě jde o všechny Work (které kompiler využívá), Status a Repeat Count registry. Kromě této automatické možnosti lze využít i ručního uložení proměnných a to dvojím způsobem:

A Pomocí save

Zde se dané proměnné uloží do stacku a my sami rozhodneme, které proměnné to budou. Zápis:

```
void __attribute__((__interrupt__(__save__(proměnné)))) název_přerušení(void)
```

A

B Pomocí shadow registrů

Některé registry mají tzv. shadow registr, což je jakoby takové zálohovací místečko. Jedná se především o první čtyři Work registry a ještě některé jiné procesorové registry. Pokud tuto možnost chcete využít, pište hlavičku takto:

```
void __attribute__((__interrupt__, __shadow__)) název_přerušení(void)
```

1"Preprolog"

Občas se stane, že ještě před tím, než instrukční čítač skočí do ISR generované kompilerem, tak musíme vykonat nějakou instrukci. Takové instrukci se říká preprolog a vykoná se hned poté, co procesor zjistí, že nastalo přerušení. Nevýhoda je, že tato instrukce může být pouze v jazyce assmebler, ale lze přistupovat i k proměnným, které jsme definovali v Céčkovém kódu:

```
void __attribute__((__interrupt__(__preprologue__("instrukce")))) název_přerušení(void)
```

Chceme-li tedy, řekněme, zvýšit obsah nějaké proměnné definované jako int x; tak místo instrukce napíšeme inc _x (podtržítko u jména proměnné nám zajistí přístup k céčkové proměnné).

1Změna PSVPAGu

Pokud očekáváme přístup do PSV paměti v ISR (je lhostejno, jestli se jedná o objekty s modifikátorem const nebo k nim přistupujeme pomocí assemblerovských funkcí pro PSV), tak musíme zajistit zálohu PSVPAGu. Tuto zálohu provedeme atributem auto_psv:

```
void __attribute__((__interrupt__, auto_psv)) název_přerušení(void)
```

V opačném případě píšeme místo auto_psv atribut no_auto_psv.

Kromě těchto možností existují i některá další, ale řekl bych, že pro naše účely bohatě stačí znát tyto. Teď už nám zbývá pouze jméno funkce pro rutinu, které si nemůžete halabala vymyslet. Tato jména jsou přesně stanovena a nacházejí se v gld souboru našeho mikroprocesoru. Chceme-li třeba napsat hlavičku pro ISR, který obsluhuje přerušení pro čítač 1 (v gld souboru jsme našli jméno _T1Interrupt), bude vypadat následovně (píši ji i s dalšími atributy, aby bylo názorně vidět, jak se píší složitější definice):

```
void __attribute__((__interrupt__(__save__(prom), __preprologue__("dec _x")), auto_psv,
__shadow__)) _T1Interrupt(void)
```

Při vracení se z ISR nemusíme používat příkaz return, protože nic nevracíme.

Na závěr ještě dodám jednu obecnou radu: snažte se rutiny přerušení dělat co možná nejkratší, bez dlouhých matematických výpočtů a volání jiných funkcí. Toto jsou věci, které by se měli nacházet v hlavní funkci (tedy main) a ISR by tedy mělo spíše obsahovat pár jednoduchých instrukcí. Je-li třeba dělat nějakou akci, která zabere větší množství času, je lepší si v přerušení nastavit příznak, že přerušení proběhlo a až v hlavní funkci tyto události řešit (také nezapomínejte nulovat flagové bity na začátku ISR!).

Příklad 11 - 03:

Zde jsem si na Vás přichystal něco trošku složitějšího. Zadání samotné je poměrně lehké. V závislosti na tom, jestli vyšleme z terminálu (rychlost 9600 Bd zůstává) znak + nebo -, nám budou diody dělat hada, který se bude pohybovat vpravo čí vlevo (při nesprávném znaku nám mikrokontrolér odpoví znakem 0 a had se zastaví). Rychlost pohybu hada je přibližně 0,25 ms (pohybem myslím přeskok svitu jedné diody na druhou), výpočet je jednoduchý: TCY je přibližně 90 ns (1 / 11059200) a potřebujeme počet instrukcí, které je nutné utratit do 0,25 ms, což je 0,25 ms / 90 ns = přibližně 2777778. Tak velké číslo do PR1 nenarveme, proto využiji předděličku 256 a po podělení 2777778 / 256 dostaneme přibližně číslo 10850 (což uložíme do PR1). Co Vás ale spíše zarazí, je proměnná flag_bity. Takto konstruované proměnné lze velmi často pozorovat v cizích kódech, neboť to šetří paměť. Každý bit této proměnné má svoji funkci, takže nás nezajímá její celková hodnota, ale hodnota jednotlivých bitů (jak tyto bity čtu a nastavuji si zkusit vysvětlit sami, potřebujete na to pouze chvilku přemýšlení a kapitolu 4.2). Všimněte si také, že proměnné, které v přerušení měním, jsem definoval modifikátorem volatile.

Co se týče přerušení, snažil jsem se je udělat co nejkratší a veškeré testování umístit do hlavní funkce. Díky tomu je zmenšená pravděpodobnost, že přerušení bude čekat, je-li vykonávána ISR jiného přerušení.

```
#include <p30fxxxx.h>
#define FCY       11059200                  //Frekvence procesoru
#define BRATE     9600                      //Požadovaná přenosová rychlost v Baudech
#define RYCHLOST  (FCY / 16 / BRATE) - 1    //Výpočet konstanty

//Definice symbolických maker pro flagové bity
#define PRIJATY    1                   //Definování makra pro přijatý znak
#define PRETECENI  2                   //Definování makra pro přetečení čítače

volatile unsigned char flag_bity = 0;  //Toto je speciální proměnná, do které si ukládám
                                       //dva flagové bity a mohou mít tyto významy:
                                       //0. bit:  0 - přerušení od UARTu nenastalo
                                       //         1 - byl přijat znak
                                       //1. bit:  0 - přerušení od TMR1 nenastalo
                                       //         1 - čítač jedna přetekl
```

```c
volatile unsigned char znak;               //Proměnná pro přijatý znak

//Defince funkce error, která vyšle
//po UARTu znak 0
void error(void)
{

   U2TXREG = '0';
}

int main(void)
{

   ADCON1bits.ADON = 0;            //Vypne AD převodník
   ADPCFG = 0xFFFF;                //Všechny piny digitální
   TRISB = 0;                      //Nastavení směru jednotlivých pinů
   PORTB = 1;
   INTCON1bits.NSTDIS = 1;         //Zakázaní vnořených přerušení

   U2BRG = RYCHLOST;               //Přiřazení konstanty
   U2MODE = 0x8000;                //Povolení UARTu, 8 - bitové znaky
                                   //bez parity
   U2STA = 0x0400;                 //Zapnutí UART vysílače
   IFS1bits.U2RXIF = 0;            //Nulování flagového bitu pro UART2 (příjem)
   IEC1bits.U2RXIE = 1;            //Povolení přerušení při příjmu znaku po UART2

   TMR1 = 0;                       //Nulování čítače
   PR1 = 10850;                    //Nastavení maximální hodnoty čítače 1
   T1CON = 0x8030;                 //Spuštění čítače, předdělička 256
   IFS0bits.T1IF = 0;              //Nulování flagového bitu pro TMR1
   IEC0bits.T1IE = 1;              //Povolení přerušení při přetečení čítače 1

   //Nekonečný cyklus
   while (1)
   {
      if (flag_bity & PRIJATY)             //Díváme se na flagový bit, jestli byl přijat znak
      {
         if (znak != '+' && znak != '-')   //Pokud je přijatý znak jiný, než +-, tak
            error();                       //vyšleme po UARTu znak 0

         flag_bity &= ~PRIJATY;            //Nulování flagového bitu od přijatého znaku
      }

      if (flag_bity & PRETECENI)           //Díváme se na flagový bit, jestli přetekl čítač
      {
            switch (znak)                  //Testujeme, jak máme čítat (jestli vůbec)
            {
               case '+':                   //Čítáme "nahoru"
                  PORTB = (PORTB == 128) ? 1 : PORTB << 1;
                  break;
               case '-':                   //Čítáme dolů
                  PORTB = (PORTB == 1) ? 128 : PORTB >> 1;
                  break;
               default:                    //Přijatý znak byl neznámý, nečítáme vůbec
                  break;
            }

         flag_bity &= ~PRETECENI;          //Nulování flagového bitu přetečení
      }
   }
}

//Přerušovací rutina pro UART2 při přijmutí znaku
//Prakticky pouze nastavíme příznaky a uložíme si přijatý znak
void __attribute__((__interrupt__, auto_psv)) _U2RXInterrupt(void)
{

   IFS1bits.U2RXIF = 0;       //Nulováni flagového bitu
   znak = U2RXREG;            //Uložení si přijatého znaku
   flag_bity |= PRIJATY;      //Nastavení příznaku, že přišel znak
}


//Přerušovací rutina pro TMR1 při jeho přetečení
//Prakticky pouze nastavíme příznaky
void __attribute__((__interrupt__, auto_psv)) _T1Interrupt(void)
{
```

```
   IFS0bits.T1IF = 0;          //Nulováni flagového bitu
   flag_bity |= PRETECENI;     //Nastavení příznaku, že čítač přetekl
}
```

Jumpery:
Jumper 1 - 8 = Diody

Snažte se pochopit všechny konstrukce, kterých jsem v tomto programu využil, protože následující kapitoly budou čím dál tím složitější, a pokud se budete ztrácet v základech, je téměř jisté, že nebudete zvládat ani následující části.

•12. Pointery

Konečně se dostáváme k srdci jazyka C neboli pointerům. Pokud si pamatujete na assembler, určitě si vzpomenete na tzv. nepřímé adresování. Šlo o konstrukci, kdy jste měli v nějaké proměnné uloženou hodnotu adresy a pomocí nepřímého adresování jste mohli obsah této adresy měnit. V Céčku dělá prakticky to stejné (a ještě mnohem víc) proměnná (budeme-li se držet správné terminologie, tak pointer není proměnná, ale její obsah, což je adresa, ale já si na slovíčkaření moc nehraji, proto je pro mě pointer proměnná) které se říká pointer (doslova ukazatel). Důležité ale je si zapamatovat, že pointer se váže na nějaký datový typ, takže říkáme třeba pointer na int, pointer na char atd. Pro názornost předvedu příklad (pointer na typ se definuje pomocí *, neboli tzv. dereferenční operátor):

```
int   *pointer_int;             //Takto jsem si nadefinoval pointer na int
int   pom_1 = 5, pom_2 = 10;    //Definice dvou proměnných typu int a jejich inicializace

int main(void)
{
   pointer_int = &pom_1;        //Získání adresy (pomocí referenčního operátoru &) proměnné pom_1
                                //a její přiřazení do pointeru
   *pointer_int = pom_2;        //Přiřazení hodnoty v proměnné pom_2 (10) a její přiřazení na
                                //adresu, na kterou "ukazuje" pointer
}
```

Definice proměnných a snad i pointeru je doufám jasná (pointer není na nic inicializován, obsahuje tedy náhodné číslo, což by mohlo při špatně napsaném kódu vést až k zápisu na místo v paměti, které našemu programu vůbec nepatří. Lepší je proto inicializovat pointer buď na nějakou konkrétní, nebo na tzv. NULL adresu). Následuje obrázek, který ukazuje tyto tři proměnné s hodnotami a s jejich adresami (adresy jsou smyšlené, kompiler nemusí přiřadit

Název	Hodnota	Adresa
pointer_int	Neznámé číslo	0x800
pom_1	5	0x802
pom_2	10	0x804

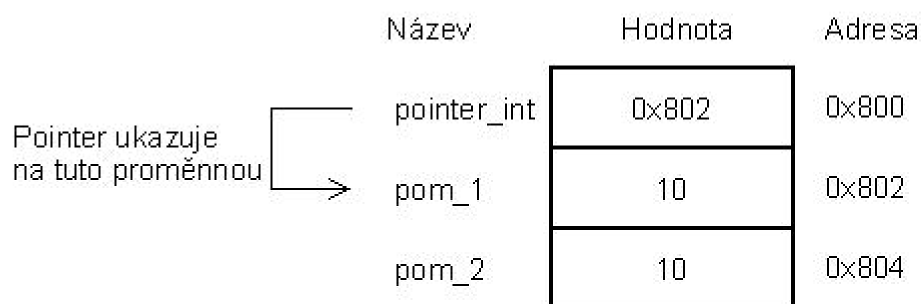
přesně tyto):

V obrázku si všimněte, jak jsou zarovnány jednotlivé proměnné. Z kapitoly 10. víme, že datová paměť má velikost jedné "paměťové buňky" 16 bitů (což je velikost jednoho intu v C30 kompileru). Nižší byte těchto 16 bitů okupují sudé adresy, liché adresy obsahují vyšší byte, takže třeba proměnná pom_1 leží na adrese 0x802 (nižší byte) až 0x803 (vyšší byte). Pointer, ať už bude ukazovat na jakýkoliv datový typ, tak bude vždy zabírat pouze jednu paměťovou buňku (tedy jakoby dvě adresy, protože každá adresa obsahuje 8 bitů). Zbylé dvě proměnné jsou zarovnány také na sudé adresy, protože obě zabírají 16 bitů (kdybychom definovali ještě jednu proměnnou typu long a umístili ji hned pod prom_2, tak by další volná adresa byla 0x810).
Nyní si pomocí referenčního operátoru získáme adresu proměnné prom_1 a přiřadíme si ji do pointeru. Obsah

Název	Hodnota	Adresa
pointer_int	0x802	0x800
pom_1	5	0x802
pom_2	10	0x804

proměnných bude následující:

A nakonec si vložíme obsah prom_2 na adresu, na kterou ukazuje pointer (toto "ukazování" se provádí dereferenčním operátorem *):



Že to není tak těžké, viďte? Pokud bychom chtěli provést opačnou operaci, tedy přiřadit hodnotu proměnné, na kterou ukazuje pointer, do jiné proměnné, místo *pointer_int = pom_2; napíšeme pom_2 = *pointer_int;

●Pointerová aritmetika

S pointery lze provádět několik operací. Můžeme je sčítat, odčítat a porovnávat. Sčítat a odčítat lze pointer a celé číslo, odčítat a porovnávat pouze dva pointery stejného typu (ukazuje-li každý na jiný datový typ, je nutné jeden z nich konvertovat na typ, jaký má ten druhý). Jak už možná tušíte, asi v tom bude jakýsi háček. Vezměme si nejprve součet pointeru (na typ int) a celého čísla (třeba 3). Přičteme-li toto číslo k pointeru, nedostaneme jeho hodnotu (tedy adresu nějakého prvku, na který pointer ukazoval) zvětšenou o tři, ale "posuneme" se tím v paměti o 3 délky daného objektu (adresa v pointeru se tedy zvýší o 6). Nedělám si iluze, že jste to pochopili (přeci jenom, vstřebat pointery chvíli trvá), proto si vše ukážeme na příkladu, kde definuji čtyři proměnné stejného typu (předpokládejme, že tyto proměnné budou v paměti zarovnány pod sebou) a jeden pointer, který na tento datový typ ukazuje:

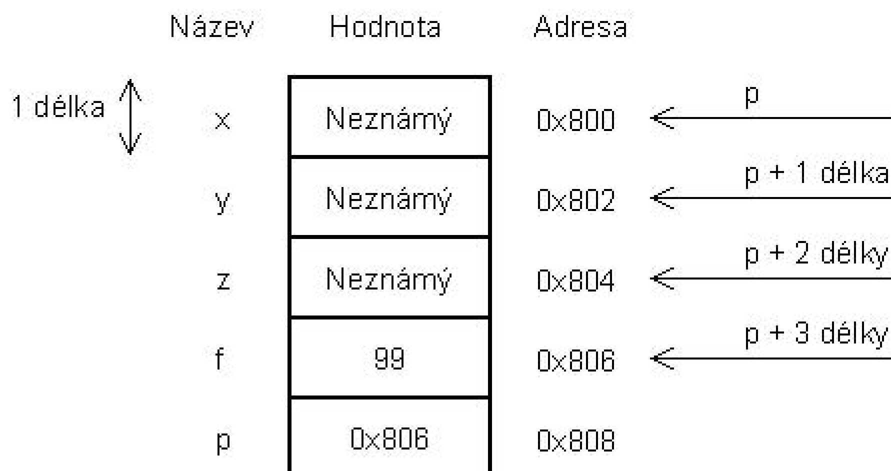
```
int x, y, z, f;    //Definice čtyř proměnných typu int (bez inicializace)
int *p = &x;       //Pointer na typ int s inicializací, takže nyní pointer ukazuje na proměnnou x

int main(void)
{
   p = p + 3;      //Přičtení čísla 3 do hodnoty pointeru
   *p = 99;        //Zápis čísla 99 do adresy, na kterou ukazuje pointer (proměnná f)
}
```

Název	Hodnota	Adresa
x	Neznámý	0x800
y	Neznámý	0x802
z	Neznámý	0x804
f	Neznámý	0x806
p	0x800	0x808

Situace před vstupem do funkce main bude vypadat takto

Hodnoty v proměnných x y z f neznáme, protože jsme je neinicializovali, pointer p zase ukazuje na adresu proměnné x. Nyní přičteme k hodnotě pointeru číslo 3. Víme-li, že velikost jednoho intu jsou dva byty a že každá adresa (0x800, 0x801, 0x802 atd...) vyjadřuje "osmibitový prostor" (to znamená, že na každé adrese se nachází 8 bitů), tak při přičtení

čísla tři k pointeru, zvětšíme jeho velikost na 0x806, protože 2 (počet bytů na datový typ int) * 3 (počet délek) + 0x800. Pokusím se to znázornit na následujícím obrázku:



Ke konci funcke main již pouze přiřadím číslo 99 do adresy, na kterou ukazuje pointer.
Doufám, že v tuto chvíli Vám už docvaklo, proč se říká pointer na datový typ. Když pointer někam ukazuje, očekává na tom místě objekt stejného typu, jakým je definovám a dle toho k němu také přistupuje. Vezmeme-li si ještě jeden příklad, kdy bude pointer typu long ukazovat na proměnnou typu long (velikost 32 bitů = 4 byty) na adrese 0x800 a přičteme k tomuto pointeru číslo 5, budeme ve výsledku ukazovat na adresu 0x820, jelikož 4 (počet bytů na datový typ) * 5 (počet délek) + 0x800 = 0x820. Zajímavé je, že pokud sčítáme pointer na char a celé číslo, dostaneme opravdu součet adresy a daného čísla, protože délka typu char je jeden byte.
Pokud budeme odečítat celé číslo od pointeru, dostaneme analogicky obrácený postup, takže adresa v pointeru je snížena o daný počet délek datového typu, na který pointer ukazuje.

Budou-li oba operandy daného výrazu dva pointery stejného datového typu, můžeme je buď porovnávat nebo odčítat (součet dvou pointerů je nesmyslná hodnota). Při rozdílu (samozřejmě počítáme větší - menší, obráceně je to blbost) je hodnota výrazu počet délek mezi těmito pointery. Mějme pointery na typ int, kdy jeden je inicializován na hodnotu 0x820 a druhý na 0x800. Při rozdílu větší - menší dostaneme jako výsledek číslo 10, nikoliv 20, protože při velikosti jedné délky objektu 2 byty je výsledek (0x820 - 0x800) / 2 = 10.

Příklad 12 - 01:
Znovu využijeme UART sběrnici na vysílání (nedivte se, bez LCD displeje je UART terminál nejlepší zobrazovací zařízení, které momentálně pomocí naší vývojové desky můžeme použít), rychlost 9600. Po zapnutí napájení nám procesor vyšle na terminál text "Ahoj svete!" a poté skočí do nekonečné smyčky, ve které nic nedělá. Jednotlivé znaky tohoto textu jsem nadefinoval do sekce "znaky", která začíná na adrese 0x800. Každý následující znak leží na adrese 0x800 + n, kde n je pořadí daného znaku (0x800, 0x801, 0x802 atd....), takže mezi jednotlivými písmeny není žádné "prázdné místo" (asi se Vám nezdá použití atributu u jednotlivých proměnných, což chápu, protože jsem o tom nic neříkal, zmiňoval jsem pouze atributy ISR. Já osobně si myslím, že pro začátek není třeba zabrouzdávat do nejtemnějších koutů C30 kompileru, proto Vám teď bohatě postačí vědět, že něco jako atributy u objektů existují a že tyto dva vytvoří sekci a přidělí adresu, tečka). Pomocí pointeru si v cyklu "vybíráme" hodnotu proměnné, na kterou zrovna daný pointer ukazuje a tuto hodnotu poté předáme funkci jako parametr (následně se pointer inkrementuje o 1, neboť postfix ++ značí, že inkrementace proměnné nastane až po vyhodnocení výrazu). Toto bude cyklus dělat do doby, než pointer "nenarazí" na znak '\0' (tzv. nulový znak, u řetězců označuje jejich konec, ale o tom více až v

kapitole o řetězcích). Tento program si dobře zapamatujte, protože se k němu budu v následujících kapitolách občas vracet.

```
#include <p30fxxxx.h>
#define FCY       11059200               //Frekvence procesoru
#define BRATE     9600                   //Požadovaná přenosová rychlost v Baudech
#define RYCHLOST  (FCY / 16 / BRATE) - 1 //Výpočet konstanty

//Definice jednotlivých znaků do sekce "znaky", která
//začíná na adrese 0x800
char s_0 __attribute__((section("znaky"), address(0x800))) = 'A';
char s_1 __attribute__((section("znaky"))) = 'h';
char s_2 __attribute__((section("znaky"))) = 'o';
char s_3 __attribute__((section("znaky"))) = 'j';
char s_4 __attribute__((section("znaky"))) = ' ';
char s_5 __attribute__((section("znaky"))) = 's';
char s_6 __attribute__((section("znaky"))) = 'v';
char s_7 __attribute__((section("znaky"))) = 'e';
char s_8 __attribute__((section("znaky"))) = 't';
char s_9 __attribute__((section("znaky"))) = 'e';
char s_10 __attribute__((section("znaky"))) = '!';
char s_11 __attribute__((section("znaky"))) = '\0';

char *p = &s_0;                     //Definice pointeru na char s inicializací,
                                    //pointer ukazuje na proměnnou s_0

//Funkce, která vyšle hodnotu parametru
//po sériové sběrnici
void vysli_znak(unsigned char znak)
{

   //Zde čekáme do doby, než se ve vysílacím
   //bufferu uvolní místo
   while(U2STAbits.UTXBF)
      ;

   U2TXREG = znak;                  //Tímto jsme odstartovali přenos
   return;                          //Návrat do hlavní funkce
}

int main(void)
{

   ADCON1bits.ADON = 0;             //Vypne AD převodník
   ADPCFG = 0xFFFF;                 //Všechny piny digitální
   U2BRG = RYCHLOST;                //Přiřazení konstanty
   U2MODE = 0x8000;                 //Povolení UARTu, 8 - bitové znaky
                                    //bez parity
   U2STA = 0x0400;                  //Zapnutí UART vysílače

   //V tomto cyklu vysíláme všechny
   //znaky ze sekce "znaky" (kromě
   //posledního, to je ukončovací znak)
   while (*p != '\0')
      vysli_znak(*p++);             //Předá hodnotu proměnné, na kterou ukazuje
                                    //pointer funkci vysli_znak a poté se pointer
                                    //zvýší o 1 (takže bude ukazovat na další proměnnou)

   //Nekonečný cyklus
   while (1)
      ;
}
```

●Konverze pointerů

Kromě pár situací (porovnávání pointerů, přetypování pointeru, který vrací funkce pro dynamické přidělení paměti atd.) nemá konverze pointeru velký význam (různé druhy kompilerů přistupují k pointerům různě, proto se konverze může stát zdrojem chyb). Pointer se přetypuje takto: (datový_typ_na_který_chceme_přetypovat *) pointer.

●Pointer a funkce

Určitě si vzpomenete na kapitolu o funkcích, kde jsem Vám tvrdil, že není možné ve volané funkci měnit obsah lokální proměnné jiných funkcí (samozřejmě pouze těch, které v tu danou chvíli existují, což jsou prakticky pouze ty, které se nacházejí ve volající funkci) a že jediné, co můžete předat v parametru, je hodnota této proměnné. To by ale

nebyl pointer, aby s tím něco neudělal! Když totiž předáváte pointer jako parametr funkce, tak funkci vlastně dáváte adresu nějakého objektu, takže když známe adresu, je možné k ní přistupovat. Ukážu na příkladu:

```
void pricti(int *p)
{
   (*p)++;
}
int main(void)
{
   int x = 0;

   pricti(&x);
}
```

V hlavní funkci jsem definoval jednu lokální proměnnou (inicializovanou na 0) a poté volám funkci pricti(). Všiměte si, co předávám jako parametr. Je to adresa proměnné x, takže tímto vlastně funkci pricti() sděluji, kde v paměti (konkrétně ve stacku) se nachází tato proměnná. Hlavička funkce pricti() obsahuje návratový typ void (nic nevracíme) a jeden parametr, kterým je ukazatel na typ int. Do tohoto pointeru se uloží adresa, kterou jsme předali jako parametr, takže nyní pointer p ukazuje na proměnnou x (aniž by samozřejmě věděl, jak se jmenuje). Nakonec hodnotu v x už pouze zvýším o jedna, zase pomocí poinetru p (podívejte se na konstrukci, co jsem použil - za (*p) si prakticky můžete představit proměnnou x. Rozlišujte ale (*p)++, což je zvýšení hodnoty proměnné, na kterou ukazuje pointer a *p++, tedy zvýšení hodnotu pointeru o jednu délku, viz příklad 12 - 01). Pokud bychom chtěli ale opravdu předávat pointer (nikoliv adresu pomocí referenčního operátoru), nezbyde nám nic jiného, než ve funkci main() dopsat definici nového pointeru:

```
void pricti(int *p)
{
   (*p)++;
}
int main(void)
{
   int x = 0;
   int *p_int = &x;

   pricti(p_int);
}
```

Dělá to úplně to samé, co příklad předešlý, ale tentokrát je pointer skutečný parametr funkce.

Pointer se ale nemusí funkci pouze předávat, může být také vracen. To znamená, že nám daná funkce jako svoji návratovou hodnotu "vyflusne" adresu nějakého objektu (je chybou vracet pointer na některou z lokálních proměnných z volané funkce, protože po opuštění funkce zanikají). Tento způsob je hojně využívaný při přidělování dynamické paměti (funkce vytvoří někde v heapu nějaký blok volné paměti a jako návratovou hodnotu nám vrátí adresu začátku bloku. K tomuto bloku pak již přistupujeme pouze pomocí pointeru, ale to už předbíhám). Pokud by chom chtěli takovou funkci definovat, je nutné její hlavičku opatřit *, takže by vypadala třeba takto: int *adresa() - čte se jako "funkce, vracející pointer na int" (jako návratovou hodnotu vracíme prostě pointer na tuto adresu). Příklad:

```
int *vrat_adresu(int *p)
{
   return p;
}
int main(void)
{
   int x = 0;
   int *p_int = &x;

   p_i = vrat_adresu(p_i);
}
```

V p_i budeme mít stále hodnotu adresy, kde se nachází proměnná x, protože jsme si ji vrátili funkcí vrat_adresu();

●Dynamické přidělení paměti

To, že naše data se mohou nacházet v normální datové paměti, stacku nebo v heapu už víme (občas i v PSV), ale o poslední možnosti jsme si zatím prakticky nic neřekli. Představte si situaci, kdy najednou potřebujeme pro náš program uvolnit nějaký blok paměti (velikost tohoto bloku se může dle okolností měnit), třeba na ukládání vzorků s

AD převodníku. Můžeme buďto využít normální datovou paměť, s čímž ale přichází problém, protože nikdy nemůžeme vědět, jestli se na dané adrese nenachází něco jiného. Stack použít prakticky nelze, protože ten svůj obsah mění doslova pořád, proto někdo přišel s myšlenkou heapu: když se programátor rozhodne pro přidělení určitého množství paměti, je mu tento blok vytvořen v tzv. heapu. Důležité je si zapamatovat, že toto přidělení může přijít kdykoliv programu a kompiler v době překládání nemusí znát velikost tohoto bloku (proto se tomu říká dynamické přidělení paměti). Na práci s heapem jsou standardní funkce, které se nacházejí v knihovně stdlib.h, takže při použití heapu ji musíme "includovat" do našeho zdrojového kódu. Také je nutné sdělit kompileru maximální velikost heapu (ten se totiž nachází v normální datové paměti, klasický paměťový model dsPICa něco jako heap nezná), což nastavíme v MPLABu takto: Project ? Build options... ? Project ? záložka MPLAB LINK30 ? Heap size (vytvoří heap paměť o velikosti x bytů). Nyní už se můžeme pustit do samotné práce heap pamětí.

Nastane-li v programu situace, že si potřebujeme přidělit určité množství paměti, tak zavoláme funkci malloc(). Parametrem této funkce je počet bytů, které chceme přidělit, takže při požadavku přidělení paměti o velikosti 2 intů budeme funkci volat malloc(4). Zde ale nastává problém různých velikostí datových typů. Někdy na začátku učebnice jsem se zmiňoval, že různé kompilery mohou mít různě velké datové typy, proto je místo toho lepší využít operátor sizeof, který "vrací" (není to funkce, proto v uvozovkách) velikost objektu v bytech. Správné volání funkce by tedy vypadalo takto malloc(sizeof(int) * 2), čímž si vytvoříme blok paměti o velikosti dvou intů. Co se týče návratové hodnoty této funkce, je jí prázdný (void, takže neukazuje na žádný konkrétní datový typ) pointer (tady je to význam adresy, nikoliv proměnné), který ukazuje na začátek tohoto bloku. Vzhledem k tomu, že tento pointer je typu void, je nutné jej přetypovat, takže když jsme vytvořili blok paměti o velikosti dvou intů, tak by měl být tento pointer přetypován na int (aby překladač věděl, že objekty v této paměti budou inty), výraz s touto funkcí bude potom vypadat (int *) malloc(sizeof(int) * 2); No a nakonec si tuto adresu někam uložíme, třeba do našeho pointeru:

```
int *p;      //Definice pointeru na int

int main(void)
{
   p = (int *) malloc(sizeof(int) * 2);      //Přidělení dynamické paměti o velikosti 4 bytů
}
```

Poté už přistupujeme k jednotlivým proměnným pomocí klasické pointerové aritmetiky (důrazně doporučuji si "zazálohovat" adresu počátku bloku a pro pohyb v něm využít jiný pointer, protože když už tuto paměť nebudeme potřebovat, je nutné ji "uvolnit", k čemuž potřebujeme právě adresu počátku).
Možná se ptáte, co se stane, když už nepůjde přidělit více paměti (heap bude plný). V takovém případě funkce malloc() vrací pointer na NULL. Je dobré nespoléhat se na náhodu a testovat návratovou hodnotu, a pokud bude heap plný, tak tuto situaci nějak řešit, takže nejsprávnější volání funkce malloc() by vypadalo takto:

```
int *p;      //Definice pointeru na int

int main(void)
{
   if ((p = (int *) malloc(sizeof(int) * 2)) == NULL)   //Lze ještě přidělit paměť?
      error();                                          //Nelze, voláme funkci error, která
                                                        //problém řeší
}
```

Samozřejmě že si nemusíme vytvořit pouze jeden blok paměti, funkci malloc() můžeme volat tolikrát, kolik bytů jsme přidělili heapu.

Po skončení práce s již dříve přiděleným blokem paměti (ať už z důvodu nepoužívanosti dat nebo z nutnosti uvolnit místo novému bloku) je nutné tuto část uvolnit. Tím dáme vědět překladači, že tuto paměť již nebudeme dále potřebovat, takže ji může považovat za prázdnou dynamickou paměť. Uvolnění se provede funkcí free(), jejíž parametrem je pointer na začátek bloku přidělené paměti. Tento pointer je nutné přetypovat na typ void, takže funkci zavoláme jako free((void *) pointer). Po uvolnění paměti nám ale náš pointer stále ukazuje do tohoto bloku, ačkoliv již nám není přidělen, proto je vhodné je "nasměřovat" někam jinam (třeba na hodnotu NULL), aby nemohlo dojít k nechtěnému zápisu do paměti, která nám nepatří.

Příklad 12 - 02:

Tak tento program je opravdu chuťovka. Jeho úkolem je simulovat paměť (bytovou, takže velikost datové sběrnice je 8 bitů), jejíž velikost si zadá sám uživatel. Ke komunikaci požívám zase terminál (pro lepší psaní doporučuji přepnout do módu Mix2), rychlost 9600 Bd. Heap mám nastaven na 500 bytů, což je možná trochu zbytečné, když maximální velikost paměti je 255 (protože po UARTu běhají byty, jejichž maximální hodnota je 255), ale pro jistotu je nastaven na větší hodnotu.

Než se začnu rozkecávat, jak program funguje uvnitř, tak pár slov k ovládání. První hodnota, kterou program očekává, je velikost paměti, takže uživatel musí zadat hodnotu od 1 do 255 (při nule bude program stále čekat). Poté má tři možnosti, tedy zapisovat (znak "W"), číst ("R") nebo uvolnit paměť a vrátit se na začátek ("F"), při všech ostatních znacích procesor vyšle znak 1 (signalizace je uzpůsobena tak, že při jakékoliv chybě vysílá 1, při potvrzení správnosti pošle 0). Vybereme-li zápis, program bude čekat na adresu, kam má zapisovat (bude-li přijatá adresa větší, než je maximální adresa, tak hlásí chybu). Po přijetí adresy pošle potvrzení a vyčkává na zapisovaný znak, jehož hodnota může být jakákoliv. Po skončení zápisu se program vrátí zpět na čekání jednoho ze tří znaků W, R nebo F. Při čtení čekáme pouze na adresu (ta je zase testována, je-li větší než maximální, tak se Vám na terminálu zobrazí 1), po jejímž přijetí je obsah této adresy vložen na PORTB, tedy LED diody (následně se vracíme na čekání jednoho z tří znaků). No a konečně pokud zvolíme možnost "F", tak se nám naše paměť uvolní a my budeme moci zase zadat novou velikost.

Jak je vidno, program používá jakýsi primitivní koncept "menu", u něhož ukazatel na výběr je proměnná flag_bity. S touto konstrukcí jsme se už setkali v příkladu 11 - 03. V hlavní smyčce se při přijetí znaku neustále dívám na jednotlivé bity této proměnné (pomocí maskování) a a koukám se, kde zrovna v menu se nacházím (a podle toho udělám příslušnou akci). Někomu se to může zdát kapánek nepřehledné, ale takový už je holt život.

Pro pohyb v dynamické paměti využívám dvou pointerů, jeden (p_zac) neustále ukazuje na začátek a druhý (p_akt) při zápisu ukazuje na místo, kam se má zapisovat. Co je ale asi nejzajímavější, je funkce pridel(). Asi Vás zarazil parametr **zacatek. Není to nic jiného, než "pointer na pointer", neboli tím kompileru oznamujeme, že funkce jako druhý parametr nedostává adresu proměnné, ale adresu pointeru (neboť jako parametr předáváme &p_zac, abychom mohli získat adresu počátku bloku). Pak už pomocí normálního dereferenčního operátoru jsme schopni k tomuto pointeru přes pointer zacatek přistoupit.

(pokud bychom uměli pracovat s řetězci, tak by mohla signalizace probíhat pomocí textu, např. "Zadejte velikost paměti", "Zadal jste špatný znak" apod., ale to se naučíme až v další kapitole. Pokud budete chtít, můžete si po jejím přečtení vytvořit textovou signalizaci)

```
#include <p30fxxxx.h>
#include <stdlib.h>
#define FCY         11059200                //Frekvence procesoru
#define BRATE       9600                    //Požadovaná přenosová rychlost v Baudech
#define RYCHLOST    (FCY / 16 / BRATE) - 1  //Výpočet konstanty

//Definování symbolických konstant
#define PRERUSENI        1
#define ZAPISUJEME       2
#define CTEME            4
#define CEKAME_CISLO     16
#define NIC_NENI         6
#define UVOLNENI         8

//Definice proměnných
unsigned char *p_zac;                       //Pointer na unsigned char
                                            //Ukazuje na začátek bloku
unsigned char *p_akt;                       //Pointer na unsigned char
                                            //Ukazuje na aktuální adresu
volatile int znak = 0;                      //Proměnná obsahující přijatý znak,
                                            //inicializace na 0
int max;                                    //Obsahuje velikost bloku
volatile unsigned char flag_bity = 0;       //Proměnná na flagové bity
//Významy jednotlivých bitů:
//0. bit:       0 - znak ještě nebyl přijat
//              1 - znak byl přijat
//1. - 2. bit:  00 - nic se neděje
//              01 - zapisujeme
//              10 - čteme
//3. bit:       0 - paměť je používána
//              1 - paměť byla uvolněna
//4. bit:       0 - čekáme na adresu (pouze pokud zapisujeme)
```

```
//              0 - čekáme na číslo (pouze pokud zapisujeme)


//Funkční prototypy
int pridel(int pocet, unsigned char **zacatek);
void vysli(char data);

int main(void)
{

   ADCON1bits.ADON = 0;               //Vypne AD převodník
   ADPCFG = 0xFFFF;                   //Všechny piny digitální
   TRISB = 0;                         //Nastavení směru jednotlivých pinů

   U2BRG = RYCHLOST;                  //Přiřazení konstanty
   U2MODE = 0x8000;                   //Povolení UARTu, 8 - bitové znaky
                                      //bez parity
   U2STA = 0x0400;                    //Zapnutí UART vysílače
   IFS1bits.U2RXIF = 0;               //Nulování flagového bitu
   IEC1bits.U2RXIE = 1;               //Povolení přerušení od příjmu

   //Nekonečný cyklus
   while (1)
   {
      //Teoreticky nekonečný cyklus, pokud bude uživatel neustále
      //zadávat velikost paměti 0.
      while(1)
      {
         //Čekáme, než nám uživatel vyšle po UARTu velikost paměti
         while(!znak)
            ;

         flag_bity &= ~PRERUSENI;     //Nulování příznaku o přerušení
         IEC1bits.U2RXIE = 0;         //Zákaz přerušení od příjmu
         max = znak;                  //V max máme velikost bloku

         //Zde voláme funkci pridel(), která nám vytvoří paměť o max prvcích
         //a uloží adresu počátku do pointeru p_zac. Pokud funkce nebude
         //schopna paměť přidělit, vrací hodnotu 1 a pak se vracíme zpět
         //na začátek cyklu
         if ((pridel(max, &p_zac)) == 1) //Šlo přidělit paměť?
         {
            vysli('1');                  //Ne nešlo, vyšli chybovou hlášku
            IFS1bits.U2RXIF = 0;         //Nulování flagového bitu
            IEC1bits.U2RXIE = 1;         //Povolení přerušení od příjmu

            continue;                    //Vracíme se zpět na začátek cyklu
         }

         p_akt = p_zac;                  //Aktuální pozice pointeru na začátek bloku

         IFS1bits.U2RXIF = 0;            //Nulování flagového bitu
         IEC1bits.U2RXIE = 1;            //Povolení přerušení od příjmu
         vysli('0');                     //Vyšli potvrzovací znak
         break;                          //Vyskočení z nekonečného cyklu
      }

      //Práce s přidělenou pamětí
      while(1)
      {
         //Čekáme, než přijde přerušení
         while (!(flag_bity & PRERUSENI))
            ;

         flag_bity &= ~PRERUSENI;              //Nulování příznaku o přerušení

         switch (flag_bity & NIC_NENI)         //Ptáme se, co je zrovna prováděno
         {
            //Zrovna není prováděno nic, přiřadíme tedy nějakou činnost
            case 0:
               switch (znak)                   //Co bude prováděno závisí na
               {                               //obsahu proměnné znak
                  case 'W':
                     flag_bity |= ZAPISUJEME;  //Budeme zapisovat
                     vysli('0');               //Vyšli potvrzovací znak
                     break;
                  case 'R':
```

```
                  flag_bity |= CTEME;       //Budeme číst
                  vysli('0');               //Vyšli potvrzovací znak
                  break;
               case 'F':                    //Uživatel chce uvolnit paměť
                  free((void *) p_zac);     //Uvolnění paměti
                  flag_bity |= UVOLNENI;    //Nastavení příznaku, aby mohl program vyskočit
                                            //ze smyčky
                  vysli('0');               //Vyšli potvrzovací znak
                  break;
               default:                     //Byl poslán špatný znak
                  vysli('1');               //Vyšli chybovou hlášku
                  break;
            }
            break;

         //Čteme
         case CTEME:
            if (znak >= max)                //Je adresa, ze které chceme číst větší
                                            //než maximální?
            {                               //Ano je
               vysli('1');                  //Vyšli chybovou hlášku
               break;
            }
            PORTB = *(p_zac + znak);        //Čtení adresy
            flag_bity &= ~CTEME;            //Nulování příznaku, už jsme dočetli
            vysli('0');                     //Vyšli potvrzovací znak
            break;

         //Zapisujeme
         case ZAPISUJEME:
            if (flag_bity & CEKAME_CISLO)   //Čekáme číslo nebo adresu
            {                               //Čekáme číslo
               *p_akt = znak;               //Zápis čísla na adresu, kam ukazuje pointer
               vysli('0');                  //Vyšli potvrzovací znak
               flag_bity &= ~CEKAME_CISLO;  //Nulování příznaku, již skončil zápis
               flag_bity &= ~ZAPISUJEME;    //Nulování příznaku, již skončil zápis
               break;
            }
            //Číslo nečekáme, čekáme adresu
            if (znak >= max)                //Je adresa, zna kterou chceme zapisovat
                                            //větší než maximální?
            {                               //Ano je
               vysli('1');                  //Vyšli chybovou hlášku
               break;
            }
            p_akt = p_zac + znak;           //Pointer ukazuje na adresu, kam
                                            //budeme zapisovat
            flag_bity |= CEKAME_CISLO;      //Nastavení příznaku, že čekáme číslo
            vysli('0');                     //Vyšli potvrzovací znak
            break;
      }

      if (flag_bity & UVOLNENI)             //Uvolnil uživatel paměť?
      {                                     //Ano
         flag_bity &= ~UVOLNENI;            //Nastavení příznaku, že paměť byla uvolněna
         znak = 0;                          //Nulování znaku
         break;                             //Návrat na úplný začátek
      }
    }    //Konec smyčky "práce s pamětí"
  }      //Konec hlavní smyčky
}        //Konec funkce main


//Funkce vytvoří blok dynamické paměti o pocet prvcích a
//uloží adresu začátku do místa, kam ukazuje pointer zacatek.
//Pokud nelze přidělit paměť, vrací 1, jinak 0
int pridel(int pocet, unsigned char **zacatek)
{
   unsigned char *docasny;          //Pointer, který ukazuje na začátek bloku.
                                    //Je to pouze dočasné, abychom nemuseli při
                                    //neúspěchu s přidělením měnit obsah adresy, na který
                                    //ukazuje zacatek
   //Zkoušíme přidělit paměť
   if ((docasny = (unsigned char *) malloc(sizeof(char) * pocet)) == NULL)
      return 1;                     //Paměť nešlo přidělit

   *zacatek = docasny;              //V p_zac máme adresu počátku bloku
   return 0;
```

```
}

//Funkce vyšle znak po sériové sběrnici
void vysli(char data)
{

   U2TXREG = data;                //Tímto odstartujeme vysílání
   return;
}

//ISR pro UART - příjem, provede se uložení přijatého znaku
//a nastavení příznaků
void __attribute__((interrupt, auto_psv)) _U2RXInterrupt(void)
{

   IFS1bits.U2RXIF = 0;           //Nulování flagového bitu
   znak = U2RXREG;                //Uložení si přijatého znaku
   flag_bity |= PRERUSENI;        //Nastavení flagu, že přišlo přerušení
}
```

Jumpery:
Jumper 1 - 8 = Diody

S pointery se dá zažít ještě spoustu srandy, takže touto kapitolou s nimi rozhodně nekončíme (spíš je to začátek).

•13.1 <u>Pole</u>

Pro vysvětlení pojmu pole využijme příklad 12 - 01. Zde jsem si definoval jakousi sekci ("znaky"), která začínala od nějaké adresy. V této sekci byly pod sebou zarovnány různé proměnné jednoho typu (v příkladu to byl char) a k jednotlivým prvkům této sekce jsem přistupoval pomocí pointeru (jeho zvýšením o jedna jsem ukazoval na následující proměnnou v sekci, neboli pointer se "posunul" o jednu délku datového typu char). Svým způsobem se jedná o jakési pole, ale nebyla na něj aplikována typicky céčková syntaxe. Pole je tedy sekce paměti, která obsahuje předem stanovený počet objektů (nemusí být jenom pole charů, může být i pole struktur, pole řetězců nebo třeba pole pointerů na funkce). Pokud ještě pořád tápete, tak si pole představte doslova jako jednořadé pole jabloní, kde hodnota jednotlivých prvků pole (tedy stromů) je počet jejich jablek.
Definice pole je v Céčku následující:

```
typ jmeno[POCET_PRVKU];
```

Typem myslíme datový typ jednotlivých prvků pole (nezapomeňte, že všechny prvky pole mají stejný datový typ, takže každý zabírá v paměti stejné místo). Jméno je identifikátor námi definovaného pole a POCET_PRVKU odpovídá množství prvků v poli (pokud takto definujeme statické pole, je nutné znát počet prvků v době překladu, aby mohl překladač vymezit dostatečně velké místo v paměti. Pole definované v heapu, tedy dynamicky, může mít proměnlivou velikost). Když už máme vytvořené pole, vyvstává otázka, jak přistupovat k prvkům (nebo chcete-li ke stromům). Jsou dva způsoby:

1Pomocí indexu
Index bývá nějaká proměnná (nebo to může být pouze číslo), jejíž hodnota značí, se kterým prvkem pole pracujeme (index ale musí být celočíselný). Zde nastává velice důležitá věc, kterou je nutné si uvědomit: číslování pořadí jednotlivých prvků začíná od nuly, takže prvnímu prvku pole náleží index 0, druhému prvku pole náleží index 1, třetímu index 2 atd. Definujeme-li tedy pole o 10 prvcích (POCET_PRVKŮ bude mít hodnotu 10), tak poslední prvek pole se nachází pod indexem 9, nikoliv 10!!! S tím souvisí další věc a to je kontrola meze polí. V Céčku taková kontrola není, takže klidně můžete použít index 10, ale budete pracovat s objektem, který leží <u>mimo</u> nám přidělený kus paměti pro pole (prostě se hrabeme na cizím písečku). Kompiler nic nehlásí, jako chybu to nebere, takže bývá na nás, abychom s indexem neutekli někam, kam nemáme. Příklad práce s prvky pole pomocí indexu:

```
char pole[10];          //Pole 10 charů

int main(void)
{
   int i = 4;           //Definice indexu a jeho inicializace na 4

   pole[0] = 6;         //Přiřazení hodnoty 6 do prvního prvku pole
   pole[2] = 9;         //Přiřazení hodnoty 9 do třetího prvku pole
   pole[i] = pole[0];   //Přiřazení hodnoty z nultého prvku pole (6) do pátého prvku pole
   i = pole[2];         //Přiřazení hodnoty z třetího prvku pole (9) do proměnné i
   pole[i] = 18;        //Přiřazení hodnoty 18 do desátého (posledního, index = 9) prvku pole
   pole[10] = 99;       //Zápis mimo naše pole (chyba, kterou kompiler nehlásí)
   pole[-1] = 2;        //Také nehlášeno kompilerem, zapíše hodnotu 2 do adresy, která je
                        //vzdálená od nultého prvku pole o jednu délku typu char směrem "dozadu"
}
```

Poslední dva příkazy jsou chybou, kterou musíme jako programátoři náležitě ošetřit (třeba testováním, nacházíme-li se s daným indexem uvnitř našeho pole).
Pomocí referenčního operátoru & lze také získat adresy prvků, takže pro zjištění adresy nultého prvku použijeme zápis &pole[0]

1**Pomocí pointeru**
Tento způsob jsme již nevědomky provozovali již v zmíněném příkladu 12 - 01. Pokud známe počáteční adresu pole, jsme schopni se pomocí klasické pointerové aritmetiky (sčítání a odčítání celého čísla) pohybovat v poli. Jak ale získat počáteční adresu? Buď z nultého prvku pole (tedy výrazem &pole[0]) nebo "z názvu", což je vlastně takový pointer na začátek pole. Tyto adresy si přiřadíme do nějakého pointeru (který ale musí být stejného datového typu jako pole) a pak pomocí pointerové aritmetiky pracujeme s hodnotami (počáteční adrese pole říkáme "bázová adresa").

```
char pole[10];          //Pole 10 charů
```

```
char *pointer;          //Pointer na char

int main(void)
{
   pointer = &pole[0];  //Přiřazení počáteční adresy pole do pointeru (1. způsob)
   *(pointer + 2) = 6;  //Přiřazení hodnoty 3 do třetího prvku pole
   pointer = pole;      //Přiřazení počáteční adresy pole do pointeru (2. způsob)
   pointer[1] = 66;     //Přiřazení hodnoty 66 do druhého prvku pole
}
```

Poslední příkaz je důkazem, že i s pointery se dá prakticky pracovat pomocí indexů (bez dereferenčního operátoru, protože kdyby byl zde použit, tak by to znamenalo, že na druhém prvku pole leží adresa nějakého objektu a do něj pomocí tohoto operátoru přiřadíme hodnotu 66). Tyto výrazy jsou tedy ekvivalentní (berte to tak, že v pointeru máme uloženou bázovou adresu pole):

```
pole[2] == *(pole + 2) == *(pointer + 2) == pointer[2]
```

●Inicializace pole

Nějakého koumáka určitě napadne, že pole by se dalo inicializovat pomocí cyklu, kde neustále se zvyšující proměnnou používám jako index a přiřazuji jednotlivé hodnoty. Tak by to určitě šlo (hlavně pro velká pole, která inicializovat ručně by byl horor), ale lze to i v definici. Používají se na to "blokové" závorky {}, do kterých píšeme hodnoty prvků (jednotlivé hodnoty oddělíme čárkou):

```
char pole[5] = {1, 3, 'A', 8, 6};      //Definice pole charů a inicializace

int main(void)
{
   char i;

   i = pole[2];                        //V i bude uložena hodnota 'A'
}
```

Pokud chceme řekněme inicializovat pouze první dva prvky, jednoduše zkrátíme závorku:

```
char pole[5] = {1, 3};       //Definice pole charů a inicializace prvních dvou prvků

int main(void)
{
   char i;

   i = pole[1];              //V i bude uložena hodnota 3
}
```

Při inicializaci (ruční) lze také vynechat POCET_PRVKŮ, neboť kompiler si jej automaticky doplní dle počtu inicializovaných hodnot (tohle je zvláště užitečné jsme-li líné osoby). Nevýhoda ale spočívá v tom, že nevíme, kolik má dané pole ve skutečnosti prvků (ve spuštěném programu to nelze zjistit, pouze při programování spočítáním inicializovaných hodnot), což se třeba řeší ukončovacím znakem, který značí, že "tady pole končí" (takto fungují řetězce, ale o tom později):

```
char pole[] = {1, 3, 9};        //Definice pole 3 charů a inicializace

int main(void)
{
   char i;

   i = pole[2];                 //V i bude uložena hodnota 9
}
```

●Operace s polem

Zde bude trochu překvapivé sdělení: s poli nelze používat žádné operátory. Neexistuje žádný výraz pole_1 = pole_2, který by Vám zkopíroval jedno pole do druhého (nebo porovnání polí operátorem ==, to lze pouze u jednotlivých prvků), to vše si musí programátor řešit sám, např. takto (pro jednoduchost uvádím pole o stejném počtu prvků):

```
char pole_1[4];                        //Definice čtyř-prvkového pole
char pole_2[4] = {'A', 'F', 69, 7};    //Definice čtyř-prvkového pole a inicializace

int main(void)
{
   char i;                   //Definice proměnné, která bude sloužit jako index
```

```
   //For cyklus bude ukládat jednotlivé prvky z pole_2 do pole_1 do doby, než index nedosáhne
   //hodnoty 4, to cyklus ukončí
   for (i = 0; i < 4; i++)
      pole_1[i] = pole_2[i];
}
```

Rozdíl bázových adres polí dělá to samé, jako rozdíl pointerů, tedy výsledkem je počet prvků mezi těmito adresami.

●Pole jako parametr funkce

I pole lze předávat jako parametr a to dvěma způsoby (závisí na tom, jestli radši pracujete s indexem nebo s pointerem). Pokud preferujeme pointery, předáváme funkci bázovou adresu pole a ve funkci samotné s tímto polem pracujeme pomocí pointerů. Příklad (funkce init() inicializuje pole sudými čísly počínaje nulou):

```
void init(int *p, char n)         //Funkce nebude nic vracet, jako parametr předáváme adresu
{                                 //pole, na kterou se inicializuje pointer p. Druhý parametr
                                  //je počet prvků v poli
   char i;                        //Pomocná proměnná

   for (i = 0; i < n; i++)
      *(p + i) = i * 2;
}

int main(void)
{
   int pole[10];                  //Definice 10-prvkového pole intů

   init(pole, 10);                //Volání funkce init()
}
```

Je dobrým zvykem při psaní vlastních funkcí předávat také počet prvků pole namísto pevného vložení počtu dovnitř funkce (takže můžete funkci volat s poli s různým počtem prvků).

U druhého způsobu také předáváme bázovou adresu, ale jako parametr funkce je vyloženě pole (nikoliv lokální, je to pouze "vnitřní" identifikátor pro funkci init()). Zajímavé je, že toto pole v hlavičce funkce nemusí mít pevně stanovenou velikost:

```
void init(int prac[], char n)
{
   char i;                        //Pomocná proměnná

   for (i = 0; i < n; i++)
      prac[i] = i * 2;
}

int main(void)
{
   int pole[10];                  //Definice 10-prvkového pole intů

   init(pole, 10);                //Volání funkce init()
}
```

Zapamatujte si, že v Céčku nelze pole předat hodnotou (ať budete dělat cokoliv, tak se Vám nevytvoří lokální pole ve stacku), ale pouze odkazem (což má za následek, že lze měnit jednotlivé prvky ve funkci).

Příklad 13 - 01:

Dnes si konečně vyzkoušíte reproduktůrek na vývojové desce. Tento program totiž hraje donekonečna úvodní 4 takty ze skladby Gyöngyhajú lány (česky "Dívka s perlami ve vlasech") od kapely Omega. Na to jsem potřeboval dva čítače a SPI převodník. Celé kouzlo tvorby jednotlivých frekvencí (skladba je hraná pomocí sinusových průběhů, nikoliv ze vzorků nějakého hudebního souboru, na to jaksi nemá dsPIC paměť) spočívá v měnění časové vzdálenosti mezi jednotlivými vzorky jakési "předpřipravené" sinusovky. Tato "předpřipravená" sinusovka má 20 vzorků amplitudy a dle toho, jaký tón je hrán, tak se upravuje perioda mezi jednotlivými vzorky (praktická realizace je pomocí TMR1, takže přijde-li od něj přerušení, tak vyšle následující vzorek z pole amplitud. TMR1 čítá do doby, která je uložena v PR1 a ta vyznačuje vzdálenost mezi jednotlivými vzorky). Přerušení od TMR2 naopak značí, že budeme hrát následující tón v poli a dle toho upravím jednotlivé PRx registry. Také jsem si definoval tři pole (a k nim náležitě

pojmenované indexy): amp(obsahuje jednotlivé vzorky amplitudy, neboli napětí sinusovky), tony(vzdálenosti mezi jednotlivými vzorky, takže jakoby frekvence tónů) a delka(doba trvání jedné noty). Pokud Vás zajímají detaily, jak jsem přišel na konkrétní čísla, projděte si soubor Vzorky.ods, který obsahuje všechny důležité výpočty a dovysvětlující texty.
Ještě bych chtěl upozornit, že jednotlivé konstanty nejsou úplně přesné (vinou je zaokrouhlování a nevyrovnání rozdílů), ale pro naše studijní účely to postačuje (další problém je v tom, že konstanty jsou počítány pro napěťový rozsah 0 - 4,095V při V_{ref} = 4,096V, což bohužel na naší desce není, takže ve skutečnosti ta napětí trochu "lítají", ale zvuk to vyluzuje takový, jaký chceme, takže co...)

```
#include <p30fxxxx.h>

//Definování symbolických konstant
#define INT_TMR1   1          //Konstanta pro přerušení od TMR1
#define INT_TMR2   2          //Konstanta pro přerušení od TMR2
#define DAC_HLAV   0x3000     //Tato hlavička se přiřadí ke každému vysílanému vzorku

//Pole vzorků amplitudy
unsigned int amp[20] = {2048, 2680, 3251, 3704, 3995, 4095, 3995, 3704, 3251, 2680, 2048, 1415,
                        844, 391, 100, 0, 100, 391, 844, 1415};

//Pole obsahující konstanty pro PR1
unsigned int tony[18] = {531, 709, 669, 796, 596, 709, 796, 709, 893, 796, 1061, 1002, 1192,
                         65535, 1338, 1417, 1338, 1591};

//Pole s délkami jednotlivých not
unsigned int delka[18] = {21700, 21700, 21700, 21700, 21700, 10850, 10850, 21700, 21700, 21700,
                          21700, 21700, 21700, 21700, 10850, 10850, 21700, 21700};

//Indexy polí
char i_amp = 0;                         //Index pole amplitud
char i_tony = 0;                        //Index pole tónů
char i_delka = 0;                       //Index pole délek

volatile unsigned char flag_bity = 0;   //Významy jednotlivých bitů:
                                        //0.bit:   0 - nic
                                        //         1 - přišlo přerušení od TMR1
                                        //1.bit:   0 - nic
                                        //         1 - přišlo přerušení od TMR2

int cisteni;

int main(void)
{

   ADCON1bits.ADON = 0;            //Vypne AD převodník
   ADPCFG = 0xFFFF;                //Všechny piny digitální
   TRISDbits.TRISD9 = 0;           //Nastavení směru RD9 (LDAC)
   PORTDbits.RD9 = 0;              //Zápis ze vstupních registrů na výstup DAC bude při
                                   //změně CS z nuly do jedné
   TRISBbits.TRISB9 = 0;           //Nastavení směru RB9 (CS)
   PORTBbits.RB9 = 1;              //DAC je v inaktivním stavu
   INTCON1bits.NSTDIS = 1;         //Zakázaní vnořených přerušení

   IFS0bits.SPI1IF = 0;            //Nulování flagového bitu
   IEC0bits.SPI1IE = 1;            //Povolení přerušení od SPI
   SPI1CON = 0x53F;                //Master mód, 16-bitová komunikace, data samplována na
                                   //náběžnou hranu, rychlost SCK je rovna FCY
   SPI1STAT = 0x8000;              //Povolení SPI modulu

   IFS0bits.T1IF = 0;              //Nulování flagového bitu
   IEC0bits.T1IE = 1;              //Povolení přerušení od TMR1
   PR1 = tony[i_tony];             //Nastavíme přehrávání od 0.tého tónu
   T1CON = 0x8000;                 //Zapne TMR1

   IFS0bits.T2IF = 0;              //Nulování flagového bitu
   IEC0bits.T2IE = 1;              //Povolení přerušení od TMR2
   PR2 = delka[i_delka];           //Nastavíme nultou délku
   T2CON = 0x8030;                 //Zapne TMR2, předdělička 256

   //Hlavní nekonečný cyklus
   while (1)
   {
      //Čekáme, než nastane nějaké přerušení
      while(!flag_bity)
```

```c
        ;

      //Dle toho, jaké přerušení nastalo, tak skočíme do jednotlivých větví

      //Přišlo přerušení od TMR2, nastavíme následující délku. Protože
      //má pro nás větší důležitost, testujeme ho dřív
      if (flag_bity & INT_TMR2)
      {
         flag_bity &= ~INT_TMR2;              //Nulování příznaku
         //Nejprve navýšíme hodnotu indexu i_tony a poté testujeme,
         //jestli nezačínáme od nultého tónu
         if (++i_tony >= 18)
            i_tony = 0;

         //Nejprve navýšíme hodnotu indexu i_delka a poté testujeme,
         //jestli nezačínáme od nulté délky
         if (++i_delka >= 18)
            i_delka = 0;

         //Zde řešíme, jestli máme "vysílat" pomlku místo tónu. Pokud ano,
         //zastavujeme čítač TMR1 a pokud ne, tak jej spouštíme (za předpokladu,
         //že ještě nebyl spuštěn)
         if (tony[i_tony] == 65535)
         {                                    //Ano, je pomlka
            T1CONbits.TON = 0;                //Pozastavujeme čítač
         }
         else if (T1CONbits.TON == 0)         //Ne, je tón
            T1CONbits.TON = 1;                //Spouštíme čítač

         PR1 = tony[i_tony];                  //Nastavujeme nový tón
         PR2 = delka[i_delka];                //Nastavujeme novou délku
      }

      //Přišlo přerušení od TMR1, tudíž pošleme do převodníku další vzorek
      if (flag_bity & INT_TMR1)
      {
         flag_bity &= ~INT_TMR1;              //Nulování příznaku
         //Nejprve navýšíme hodnotu indexu i_amp a poté testujeme,
         //jestli náhodou nemáme vysílat od nultého vzorku
         if (++i_amp >= 20)
            i_amp = 0;                        //Vysíláme od nultého vzorku

         PORTBbits.RB9 = 0;                   //CS do nuly - začínáme vysílat
         cisteni = SPI1BUF;                   //Tuto a následující instrukci je nutno provést,
         SPI1STATbits.SPIROV = 0;             //abychom vyčistili buffer
         SPI1BUF = DAC_HLAV | amp[i_amp];     //Vysíláme vzorek s hlavičkou po SPI
      }
   }
}

//ISR pro SPI, která jenom vynuluje příznak a nastaví DAC do inaktivního stavu
void __attribute__((interrupt, auto_psv)) _SPI1Interrupt(void)
{

   IFS0bits.SPI1IF = 0;            //Nulování flagového bitu
   PORTBbits.RB9 = 1;              //CS do jedné - dokončili jsme vysílání celého slova
}

//ISR pro TMR1, zde nuluji flagový bit a nastavuji příznak, že přišlo přerušení
void __attribute__((interrupt, auto_psv)) _T1Interrupt(void)
{

   IFS0bits.T1IF = 0;              //Nulování flagového bitu
   flag_bity |= INT_TMR1;          //Nastavení příznaku, že přišlo přerušení od TMR1
}

//ISR pro TMR2, zde nuluji flagový bit a nastavuji příznak, že přišlo přerušení
void __attribute__((interrupt, auto_psv)) _T2Interrupt(void)
{

   IFS0bits.T2IF = 0;              //Nulování flagového bitu
   flag_bity |= INT_TMR2;          //Nastavení příznaku, že přišlo přerušení od TMR2
}
```

Jumpery:

Jumper 9 - 11 = DA převodník

Jumper 12 = Nastaven buď na kanál A DA převodníku nebo vytáhnutý (pro poslech z jacku)

●Dynamické pole

Vytvoření pole o určitém počtu prvků v heap paměti není nic těžkého, už jsme se s tím dokonce setkali. Protože s polem se dá pracovat jako s pointerem, tak se to dělá tak, že do tohoto pointeru si necháme uložit adresu počátku této přidělené paměti (pomocí funkce malloc()) a poté přistupujeme k jednotlivým prvkům pomocí indexu (nebo sčítáním pointeru a celého čísla, vyberte si způsob, který se Vám více zamlouvá). Příklad:

```
int *p;              //Pointer na typ int

int main(void)
{
   p = (int *) malloc(sizeof(int) * 10);        //Vytvoření 10-prvkového pole intů
   p[0] = 99;                                   //Přiřadí prvnímu prvku číslo 99
   p[7] = p[0];                                 //Přiřadí osmému prvku číslo z prvního prvku (99)
   free((void *) p);                            //Uvolnění přidělené paměti
}
```

Stejně jako všude, tak i u dynamického pole Céčko nekontroluje meze polí.

•13.2 Řetězce

V Céčku je řetězec pole charů, které obsahuje ASCI znaky a konec tohoto pole (tedy poslední použitelný prvek pole) je zakončen tzv. EOS znakem (EOS - end of string, neboli konec řetězce, je definován jako dekadická 0 nebo '\0') - řetězec je tedy něco jako text. Vytvořme si tedy řetězec:

```
char retezec[5] = {'A', 'h', 'o', 'j', '\0');     //Definice řetězce o 5 znacích (všimněte si, že
                                                  //kvůli ukončovacímu znaku jsme jaksi přišli o
                                                  //jednu pozici, na které se mohl nacházet znak)
```

Řetězec se tedy zarovná v paměti takto (počátek řetězce je na adrese 0x800):

Hodnota	'A'	'h'	'o'	'j'	'\0'
Adresa	0x800	0x801	0x802	0x803	0x804

Abychom pokaždé nemuseli při inicializaci psát ukončovací znak, lze ji napsat takto:

```
char retezec[5] = "Ahoj";
```

Ačkoliv jsme napsali text o čtyřech znacích, tak je nutné definovat pole o velikost pěti prvků (kvůli EOS - v tomto případě jej kompiler sám do pole dodá). Vynecháme-li počet prvků, automaticky se doplní:

```
char retezec[] = "Ahoj";
```

Ve všech třech případech máme pěti-prvkové pole čtyř znaků a jednoho EOS znaku.

●Řetězec jako parametr funkce

Pokud rozumíte předávání pole funkci, tak tato část rozhodně nebude žádnou novinkou, protože stejně jako pole, tak i řetězec obsahuje bázovou adresu, kterou funkci předáváme (nezapomínejte, že řetězec je pole se vším všudy, jenom se používá k něčemu trochu jinému). Rozdíl oproti normálním polím je ale v tom, že funkce předem nemusí znát počet prvků v řetězci, neboť může využít konstrukce "čti do doby, než narazíš na EOS" (uvidíte v následujícím příkladu). Občas se setkáte s tím, že se řetězec nepředává funkci pomocí identifikátoru (neboli bázové adresy), ale že se do uvozovek přímo vypíše daný text, např. uvažujme funkci vysli_znaky(), která nic nevrací a její jediný parametr je pointer na typ char, které předáme nějaký řetězec:

```
vysli_znaky("Nazdar!");
```

Toto není nějak konkrétně pojmenovaný řetězec, který by byl v nějakém námi definovaném poli (kompiler sám vytvoří pole obsahující tento řetězec, nejčastěji v PSV paměti, neboť je konstantní - nelze jej změnit), "Nazdar!" tedy není nic jiného, než adresa počátku tohoto pole s textem.

Příklad 13 - 02:

Zase jednou využijeme terminál (a znovu s rychlostí 9600 Bd). Nejprve vyšleme jakousi uvítací zprávu, která po nás bude žádat jeden ze tří znaků a dle toho, který z nich pošleme, tak nám dsPIC odpoví příslušnou zprávou (pro všechny ostatní znaky vyšleme jednu chybovou hlášku). V programu nehledejte nějaký zádrhel, jediná novinka je využití řetězců (předávání funkci jsem udělal dvěma způsoby, tedy buď přes bázovou adresu nebo rovnou jako vepsaný text, abyste viděli, že lze obojí). Ve funkci vysli() využívám toho, že řetězce jsou ukončeny znakem EOS, takže vysílám všechny znaky, než na něj narazím. (malá poznámečka k escape sekvencím: pro terminál Tiny bootloaderu stačí pro posun na další řádek napsat '\n', ale ne každému programu to stačí. Pokud Vám to nebude odřádkovávat v jiném terminálu, je nutné za každým '\n' znakem přidat další speciální escape sekvenci '\r')

```
#include <p30fxxxx.h>
#define FCY       11059200     //Frekvence procesoru
#define BRATE     9600         //Požadovaná přenosová rychlost v Baudech
#define RYCHLOST  (FCY / 16 / BRATE) - 1  //Výpočet konstanty

//Definice symbolických konstant pro příznaky
#define PRIJEM  1

volatile unsigned char flag_bity = 0;     //Významy jednotlivých bitů:
                                          //0. bit:  0 - nic se neděje
                                          //         1 - přišlo přerušení od UART příjmu
volatile char znak;                       //Do této proměnné budeme ukládat příchozí znak

//Defincie jednotlivých vzkazů (řetězců)
char zprava_1[] = "Zdravim vsechny pozemstany!\n";
char zprava_2[] = "Jen tak si vysilaaam...\n";
char zprava_3[] = "SPSST Panska je nejlepsi!\n";     //Tuto reklamu jsem si neodpustil :)
char error[] =  "Zadal jsi neplatny znak!\n"
                "Zadej bud A, B nebo C!\n";

//Funkce vysli(), která pošle po sériové sběrnici
//řetězec, který je uložen v poli. Využíváme toho, že
//řetězce jsou zakončeny znakem '\0'
void vysli(char retezec[])
{
   int i;        //Index pro retezec

   //Budeme vysílat jednotlivé znaky z řetězce do doby,
   //než narazíme na EOS
   for (i = 0; retezec[i] != '\0'; i++)
   {
      //Čekáme, než se vyprázdní buffer
      while(U2STAbits.UTXBF)
         ;

      U2TXREG = retezec[i];
   }
}


int main(void)
{

   ADCON1bits.ADON = 0;          //Vypne AD převodník
   ADPCFG = 0xFFFF;              //Všechny piny digitální

   U2BRG = RYCHLOST;             //Přiřazení konstanty
   U2MODE = 0x8000;              //Povolení UARTu, 8 - bitové znaky
                                 //bez parity
   U2STA = 0x0400;               //Zapnutí UART vysílače
   IFS1bits.U2RXIF = 0;          //Nulování příznaku
   IEC1bits.U2RXIE = 1;          //Povolení přerušení při příjmu znaku po UARTu

   vysli("Pro jeden ze tri znaku zadejte A, B nebo C\n");    //Vyšle uvítací zprávu

   //Nekonečný cyklus
```

```
   while (1)
   {
      //Čekáme, než přijde přerušení
      while (!(flag_bity & PRIJEM))
         ;
      flag_bity &= ~PRIJEM;      //Nulování příznaku

      //Dle toho, jaký znak byl přijat, tak skočíme do příslušné větve
      switch (znak)
      {
         //Pro znak 'A'
         case 'A':
            vysli(zprava_1);     //Vyšle první zprávu
            break;

         //Pro znak 'B'
         case 'B':
            vysli(zprava_2);     //Vyšle druhou zprávu
            break;

         //Pro znak 'C'
         case 'C':
            vysli(zprava_3);     //Vyšle třetí zprávu
            break;

         //Pro všechny ostatní znaky
         default:
            vysli(error);        //Vyšle chybovou hlášku
            break;
      }
   }
}

//ISR pro UART příjem, je zde pouze nastavení příznaků a
//uložení si příchozího znaku
void __attribute__((interrupt, auto_psv)) _U2RXInterrupt(void)
{

   IFS1bits.U2RXIF = 0;          //Nulování flagového bitu
   znak = U2RXREG;               //Uložení si přijatého znaku
   flag_bity |= PRIJEM;          //Nastavení flagu, že přišlo přerušení
}
```

•13.3 Vícerozměrná pole

O jednořadém poli jabloní jsem již vyprávěl na začátku kapitoly o polích. Teď si ale představte, že budeme chtít přidat další řádky stromků. Řešením je použití vícerozměrných polí, v našem příkladě tedy dvourozměrného (jeden rozměr na počet řad a druhý na počet sloupců). Definice vícerozměrného pole o třech řádcích a pěti sloupcích vypadá následovně:

```
int pole[3][5];
```

Je tedy vidno, že první číslo udává počet řádků a druhý počet sloupců. Tímto jsme vytvořili jakousi "tabulku", kde každá buňka má přesně stanovené souřadnice vytvořené z čísla řádku a sloupce:

	Sloupce				
Řádky	pole[0][0]	pole[0][1]	pole[0][2]	pole[0][3]	pole[0][4]
	pole[1][0]	pole[1][1]	pole[1][2]	pole[1][3]	pole[1][4]
	pole[2][0]	pole[2][1]	pole[2][2]	pole[2][3]	pole[2][4]

Teď by si to hlavně chtělo ujasnit, co znamená vlastně řádek a co sloupec. Hodnota řádku (např. pole[1]) není nic jiného, než pointer na jednorozměrné pole o pěti prvcích (pole[1] je adresa začátku tohoto pole, je normálně tisknutelná, takže si ji můžete nechat vypsat). Definujeme-li tedy tři řádky, prakticky jsme vytvořili jednorozměrné pole o třech prvcích, kde každý prvek tohoto pole je pointer na další, tentokrát pěti-prvkové pole (v tomto pěti-

prvkovém poli si jednotlivé buňky vybíráme pomocí souřadnice, kterou jsem nazval "sloupec"). Pro lepší pochopení

této situace je následující obrázek:

Co se týče zarovnání v paměti, Céčko zarovnává po řádcích, takže v paměti to vypadá takto: pole[0][0], pole[0][1], pole[0][2], pole[0][3], pole[0][4], pole[1][0], pole[1][1] atd....

Rozměrů může být i více, než dva, takže lze udělat i třírozměrné (tím se vytvoří pole pointerů, kde každý pointer odkazuje na další pole pointerů a tam každý pointer odkazuje na pole takového datového typu, jakým bylo toto vícerozměrné pole definováno) či vícerozměrné (prakticky můžeme jít až donekonečna nebo do doby, co nám stačí paměť)

●Inicializace vícerozměrného pole

Stejně jako obyčejná pole a řetězce, tak i vícerozměrná pole lze inicializovat na začátku. Chceme-li naplnit výše uvedenou tabulku čísly 1 až 15, použijeme následující definici:

```
int pole[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
```

Lze vynechat první rozměr, v takovém případě si jej překladač doplní sám (druhý údaj je ale nutno uvést).

```
int pole[][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
```

Stejně tak i při předávání vícerozměrného pole funkci lze první rozměr (v hlavičce funkce) vynechat, důležitý je pouze rozměr druhý.

●Pole řetězců

Nejčastější použití dvourozměrného pole bývá právě pole řetězců, ačkoliv jeho definice se neprovádí přes dvourozměrné pole, ale přes pole pointerů na řetězce, ukáži proč:

```
char pole[3][23] = {{"Ahoj"}, {"B"}, {"Tady je dlouhy retezec"}};
```

Tímto jsem definoval dvourozměrné pole o třech řádcích, kde každý řádek představuje pointer na jednotlivé řetězce. Očividnou nevýhodou této definice je naprosté plýtvání pamětí, neboť všechny řetězce se skládají z pole o 23 prvcích (neboť definicí vícerozměrných polí lze vytvářet pouze pravidelné "obdélníkové" pole. Jiné tvary lze vytvářet, ale použití těchto konstukcí je pro nás v tuto chvíli zbytečná, zájemce odkazuji na doporučenou literaturu), ačkoliv např. v případě řetězce "Ahoj" je zaplněno pouze prvních 5 prvků (samotný text + EOS). Mnohem lepší je definovat jednorozměrné pole pointerů na char:

```
char *pole[3] = {"Ahoj", "B", "Tady je dlouhy retezec"};
```

Zde již pamětí neplýtváme, neboť pro každý prvek našeho třírozměrného pole byl vytvořen pointer na jiná pole (řetězce), které můžou mít různou velikost. Nyní již lze přistupovat k jednotlivým znakům pomocí obou souřadnic (první je číslo řetězce a druhá je pořadí znaku v tomto řetězci), takže např. pole[2][1] obsahuje hodnotu 'a'.

•14.1 Operátor typedef

Ještě před tím, než se pustíme do struktur (na které jste již určitě natěšení), tak si povíme něco o operátoru typedef. Bude to velice krátká odbočka ale nutná, protože znalosti z ní budeme používat po zbytek kapitoly.

Toužili jste si někdy vytvořit svůj datový typ? Chtěli jste mít vedle těch všech intů, charů a doublů také vlastní? Jestli jste odpověděli kladně, tak právě pro Vás zde mám operátor typedef. Jeho použití je velice snadné. Příklad: chceme si ušetřit práci při definici proměnných typu volatile unsigned int tím, že nebudeme pokaždé vypisovat celý název tohoto datového typu. Pomůžeme si operátorem typedef, se kterým si vytvoříme vlastní typ, který se bude jmenovat V_U_INT:

```
typedef volatile unsigned int V_U_INT;
```

Od této chvíle můžeme jakékoliv proměnné přiřadit datový typ V_U_INT. Pokud se během programování rozhodneme, že by bylo dobré si vytvořit datový typ, který by ukazoval na V_U_INT, tak to napíšeme takto:

```
typedef V_U_INT *P_V_U_INT;
```

Dávejte ale pozor při definici pointeru tohoto datového typu, neboť jej definujeme BEZ dereferenčního operátoru (ten se prakticky ukrývá v P_V_U_INT):

```
P_V_U_INT pointer;      //Definice BEZ dereferenčního operátoru
```

Na předchozích příkladech lze krásně vidět, k čemu ten operátor typedef vlastně je. Jeho úlohou je zjednodušit definice objektů, jejichž datový typ je jedno velké, komplikované "něco". Je to jako se slupkami cibule - postupně nabalujeme další vytvořené datové typy ("slupky"), až dostaneme celou defínci ("cibuli"). Snažte se jej používat co nejčastěji, protože není nic horšího, než luštit složité definice...

•14.2 Struktury

Předně chci říct, že struktura je jakýsi datový typ, který v sobě zahrnuje různé objekty (pro připomenutí, objekty zde myslím třeba proměnné, pole, pointery ale můžou to být i další struktury atd. - objekt v mém podání nemá nic společného s objektově orientovaným programováním), ke kterým můžeme libovolně přistupovat (tedy z nich číst a do nich psát). Jako takový typická struktura může být v reálném světě třeba nějaký formulář, kde jednotlivé kolonky (prvky) tohoto formuláře (struktury) můžou značit např. jméno (ve struktuře by to byl řetězec, tedy pole charů), věk (proměnná typu unsigned char) apod. Struktura se dá definovat mnoha způsoby (společné mají použití klíčového slova struct, označující definici struktury):

1"Nepřidělitelná" struktura

```
struct {
   char jmeno[20];
   unsigned char vek;
};
```

V příkladě vidíte převedení formuláře do struktury, jednotlivé prvky se píší do "blokových" závorek {}. Takto nadefinovaná struktura je nám naprosto na nic, protože neexistuje žádný specifický objekt, který by byl tohoto typu.

1S definicí proměnných

Při definici struktury můžeme rovnou některým proměnným přiřadit typ zrovna definované strukury, dělá se to takhle:

```
struct {
   char jmeno[20];
   unsigned char vek;
} osoba_1, osoba_2;
```

kde osoba_1 a osoba_2 jsou proměnné, jejichž typ je výše definovaná struktura.

1S pojmenováním
Použijeme-li tento způsob, tak jsme schopni díky identifikátoru strukury (píše se mezi klíčovým slovem struct a otvírací složenou závorkou) definovat kdekoliv v programu proměnnou stejného typu:

```
struct formular {
   char jmeno[20];
   unsigned char vek;
} osoba_1, osoba_2;
```

Pokud budeme chtít třeba za 100 řádků kódu definovat novou proměnnou, provedeme to následovně:

```
struct formular osoba_3;
```

Všimněte si, že bylo nutné použít klíčové slovo před identifikátorem.

1Pomocí operátoru typedef
Asi nejlepší způsob a také nejčastější. Operátorem typedef si vytvoříme nový datový typ formular (struktura dvou prvků - pole charů a jeden neznaménkový char) a pak můžeme proměnné definovat na tento typ (tentokrát ale bez již bez klíčového slova struct):

```
typedef struct {
   char jmeno[20];
   unsigned char vek;
} FORMULAR;             //Zde FORMULAR neoznačuje proměnnou s názvem FORMULAR, ale datový typ

FORMULAR osoba_1, osoba_2;
```

Za předpokladu, že prvkem dané struktury je struktura stejného typu (což lze), je nutné uvést i jméno struktury před otvírací závorkou. Tuto vnořenou strukturu nelze definovat normálním způsobem, je nutné využít pointer. Pro zajímavost zde předkládám definici takové struktury a vícekrát se k tomu již nebudu vracet (zájemce přesměrovávám na doporučenou literaturu):

```
typedef struct formular {
   char jmeno[20];
   unsigned char vek;
   struct formular *otec;
   struct formular *matka;
} FORMULAR;
```

Jak matka, tak i otec jsou neinicializované pointery, které momentálně neukazují nikam, je proto nutné dynamickým přidělením paměti tyto dvě struktury "vytvořit".
Dohodou je, že pro pojmenování struktury (formular) a datového typu (FORMULAR) se použije stejné slovo, ale s různou velikostí písmen.

●Přístup k jednotlivým prvkům struktury

Mějme datový typ struktury FORMULAR (lehká modifikace předchozích struktur, zde bude totiž prvek jmeno tvořit pointer na řetězec) a proměnnou tohoto typu osoba_1 a nyní chceme v hlavním programu vyplnit jednotlivé položky. Jak tedy přistupovat k jednotlivým prvkům? Velice jednoduše a to pomocí operátoru tečka, jehož použití můžete vidět na následujícím příkladu:

```
typedef struct formular {
   char *jmeno;              //Pointer na char, bude obsahovat adresu prvního prvku řetězce
   unsigned char vek;
} FORMULAR;

FORMULAR osoba_1;

int main(void)
{

   osoba_1.jmeno = "Pavel Liska";  //Zde vidíte použití operátoru tečka, prvek je oddělen od
                                   //identifikátoru struktury tečkou. Jinak se jedná o přiřazení
                                   //adresy prvního prvku řetězce (tedy pole) do pointeru jmeno
   osoba_1.vek = 41;               //Přiřazení hodnoty 41 do prvku vek struktury osoba_1
}
```

Přiřazení hodnoty z prvku struktury se dělá úplně stejně, tedy s tečkovým operátorem. Oproti polím mají struktury jednu ohromnou výhodu a tou je přiřazení struktury do jiné struktury. V praxi to znamená. že můžeme psát

```
osoba_2 = osoba_1;
```

V tomto příkazu se přiřadí veškeré hodnoty prvků struktury osoba_1 do osoba_2.

●Inicializace struktury

Je velice podobná inicializaci polí. Veškeré hodnoty inicializačního procesu píšeme do složených závorek a oddělujeme čárkou. Je-li prvkem struktury jiná struktura (nebo pole), lze jej také inicializovat, ale je nutné jej oddělit novými {} závorkami. Pro pochopení dokládám příklad:

```
typedef struct adresa {
  char *ulice;
  char *mesto;
} ADRESA;

typedef struct formular {
   char *jmeno;
   unsigned char vek;
   ADRESA bydliste;           //Tento prvek je struktura ADRESA
} FORMULAR;

FORMULAR osoba_1 = {"Florian Utrinos", 26, {"Kopacska 1897/2", "Brno"}};

int main(void)
{

   vysli_retezec(osoba_1.bydliste.ulice);     //Takto jsem předal funkci vysli_retezec() (její
                                              //definici jsem vynechal) adresu prvního prvku
                                              //řetězce, na který ukazuje pointer ulice
}
```

Tento příklad také demonstruje přístup ke struktuře v jiné struktuře, což se děje pomocí dvojitého použití tečky (je to logické, první tečkou přistoupíme na prvek struktury FORMLUAR, kterým je další struktura bydliste typu ADRESA a pomocí další tečky přistoupíme k prvku ulice, což je pointer na řetězec).

●Pointer na strukturu

To, že pointer je mocná (a zároveň nebezpečná) věc, jsme zjistili už dávno, a co by to bylo za pointer, kdyby nedokázal odkazovat na strukturu. Struktury, ke kterým se přistupuje pomocí pointeru (a ne jejich identifikátoru), využívají pro práci se svými prvky operátor ->, což je rozdíl oproti klasickému operátoru tečka:

```
typedef struct formular {
   char *jmeno;
   unsigned char vek;
} FORMULAR;

FORMULAR osoba_1;
FORMULAR *p;         //Pointer na datový typ FORMULAR

int main(void)
{

   p = &osoba_1;              //Přiřazení adresy struktury osoba_1 do pointeru p
   p->jmeno = "Ales Novak";   //Přístup k prvku jmeno struktury osoba_1 přes pointer p
}
```

Toto je nejběžnější přístup pomocí pointeru, ale lze využít i trošku komplikovanější způsob:

```
(*p).jmeno = "Ales Novak";
```

Osobně si myslím, že přístup pomocí tohoto způsobu nemá své opodstatnění, ale závisí na Vás, pro který se rozhodnete. Chybou (v tomto konkrétním případě) by bylo určitě vynechání závorky:

```
*p.jmeno = "Ales Novak";
```

neboť takto přiřazujeme adresu řetězce na buňku, na kterou odkazuje hodnota v prvku jmeno.

Pro práci s dynamickými strukturami se zásadně používají pointery (ono to ani jiným způsobem nejde), tudíž operátor ->. Je ale nutné znát velikost struktury v bytech, abychom mohli správně alokovat paměť, což provedeme operátorem sizeof:

```
typedef struct formular {
   char *jmeno;
   unsigned char vek;
} FORMULAR;

FORMULAR *p;          //Pointer na datový typ FORMULAR

int main(void)
{

   p = (FORMULAR *) malloc(sizeof(FORMULAR));   //Přidělení paměti o velikosti typu FORMULAR
   p->vek = 15;                                 //Přístup k prvku vek dynamické struktury pomocí
                                                //pointeru p
   free((void *) p);                            //Uvolnění paměti
}
```

●Funkce a struktury

Na rozdíl od polí, která jdou předávat pouze odkazem (a tím pádem lze měnit jejich jednotlivé prvky přímo), tak u struktur můžeme předávat jak odkazem, tak i hodnotou. Druhý jmenovaný způsob je málo používaný, protože při větších velikostech struktury máte vyžranou paměť ve stacku natotata, ale občas lze použít pro malé struktůrky. Zdrojový kód, jehož obsahem by byla funkce, která by sečetla prvky x a y ze struktury PROMENNE a jejich výsledek vrátila, je následující (všimněte si použití operátoru tečka):

```
typedef struct promenne {
   int x;
   int y;
} PROMENNE;

PROMENNE moje = {56, 44};       //Definice proměnné moje na datový typ PROMENNE a její
                                //inicializace

int secti(PROMENNE lokalni_strukt)
{

   return (lokalni_strukt.x + lokalni_strukt.y);     //Vracíme součet prvků x a y
}

int main(void)
{
   int vys;

   vys = secti(moje);           //Jako parametr předáváme hodnotu prvků proměnné moje
}
```

Pokud bychom ale chtěli měnit jednotlivé prvky, je nutné předat adresu struktury a v hlavičce volané funkce definovat parametr jako pointer na strukturu (tím myslím daný datový typ). K přístupu k jednotlivým prvků, používáme operátor ->, neboť využíváme pointeru (tento příklad dělá to samé, akorát že funkce nic nevrací a výsledek se uloží do prvku vys proměnné moje):

```
typedef struct promenne {
   int x;
   int y;
   int vys;
} PROMENNE;

PROMENNE moje = {56, 44};       //Definice proměnné moje na datový typ PROMENNE a její
                                //inicializace

int secti(PROMENNE *p)                 //Parametr funkce je pointer na datový typ PROMENNE, do
{                                      //kterého se uloží adresa struktury moje

   p->vys = p->x + p->y;
}


int main(void)
{
```

```
   secti(&moje);                     //Jako parametr předáváme adresu proměnné moje
}
```

Funkce může ale i vracet pointer na strukturu (to je případ funkce, která alokuje dynamickou strukturu a vrací na ní pointer), což děláme buď return &struktura, nebo return pointer_na_tuto_strukturu.

●Bitové struktury

Určitě si vzpomenete na příklady z předchozích kapitol, kde jsem používal jakousi proměnnou flag_bity, ve které měl každý bit jiný význam. Pro nastavování a testování jednotlivých bitů této proměnné jsem využíval binární součet a součin. Pokud se Vám toto nastavování zdálo komplikované, tak právě pro Vás byly stvořeny bitové struktury. Tentokrát se ale nejedná o strukturu, která by obsahovala nějaké objekty, ale pouze jednu proměnnou typu int. Bity v ní můžou být pouze typu unsigned nebo signed int (ačkoliv se říká int, tak to int úplně není, spíš si představte, že to může být pouze znaménkové či neznaménkové celé číslo). Následující příklad ukazuje, jak bychom mohli takovou strukturu definovat:

```
typedef struct flag_bity {
   unsigned int inter_uart    : 1;      //Vyhrazení jednoho bitu pro inter_uart
   unsigned int inter_spi     : 1;      //Vyhrazení jednoho bitu pro inter_spi
   unsigned int mod           : 6;      //Vyhrazení šesti bitů pro mod
   unsigned int nevyuzito     : 8;      //Zbylých 8 bitů je přiřazeno proměnné nevyuzito
} FLAG_BITY;
```

Jak tedy vidíte, hodnota čísla za dvojtečkou značí počet bitů, které bude přiděleno danému objektu ze struktury. Přístup k jednotlivým proměnným je již velice jednoduchý:

```
FLAG_BITY nastavovaci;

int main(void)
{
   int x;

   nastavovaci.inter_uart = 0;          //Nastavení proměnné inter_uart
   x = nastavovaci_mod;                 //Přiřazení hodnoty z proměnné mod do proměnné x
}
```

Pokud by Vás zajímalo, jak je zarovnána tato struktura v paměti, tak se podívejte na následující obrázek:

Bity se tedy začínají přidělovat od LSB, nikoliv MSB.

Bitové struktury jsem používal od počátku učebnice, většinou se jednalo o vypnutí AD převodníku příkazem ADCON1bits.ADON = 0; což je vlastně přístup do bitové struktury ADCON1bits. Tato struktura je v hlavičkovém souboru definována takto:

```
typedef struct tagADCON1BITS {
   unsigned DONE   :1;
   unsigned SAMP   :1;
   unsigned ASAM   :1;
   unsigned        :2;
   unsigned SSRC   :3;
   unsigned FORM   :2;
   unsigned        :3;
   unsigned ADSIDL :1;
   unsigned        :1;
   unsigned ADON   :1;
} ADCON1BITS;
extern volatile ADCON1BITS ADCON1bits __attribute__((__sfr__));
```

První část je definice struktury ADCON1bits, ke které jsou přiřazeny jednotlivé proměnné s různou velikostí. Podíváte-li se do datasheetu, tak zjistíte, že tato struktura svou definicí odpovídá registru ADCON1. Druhou částí je

již pouze deklarace externí (to znamená, že tato proměnná je definována v jiném modulu) proměnné ADCON1bits s atributem __sfr__, což značí, že tato proměnná je speciální registr procesoru. Pomocí gld souboru a jeho linkováním k projektu lze přiřadit ADCON1bits adrese 0x02A0 (není nutné vědět, jak to kompiler dělá).
Teď už pouze vyvstává otázka, jestli je výhodnější k příznakové proměnné přistupovat pomocí klasických bitových operací nebo struktur. Možná se Vám to bude zdát velice zvláštní, ale s těmito proměnnými je lepší pracovat pomocí struktur. Vymyká se to zdravému rozumu, neboť assembler nezná datový typ strukturu, takže se očekává, že překladač bude muset vytvořit speciální procedury pro práci se strukturami. Nevím, jestli je to dáno tím, že používám free verzi kompileru, který je bez jakékoliv úrovně optimalizace, ale nastavení jednoho bitu v bitové struktuře trvá pouze jednu instrukci, zatímco provedení toho samého pomocí bitových operací zabere instrukce tři! Co z toho vyplývá? Používejte raději bitové struktury, je to nejenom přehlednější, ale překvapivě i rychlejší.

•15.1 Výčtový typ

Kromě symbolických konstant existuje jakýsi výčtový typ, což je seznam symbolických konstant (ačkoliv nemají nic společného s preprocesorem, který nahrazuje hodnotu symbolické konstanty během překladu programu). Patří mu klíčové slovo enum a takový seznam lze definovat takto (dle konvence se jednotlivé položky výčtového typu píší velkými písmeny):

```
typedef enum {
   PANSKA, KRIZIKOVA, KARLIN        //Zde není žádný středník
} SKOLY;
```

Předem si je nutné uvědomit, že takto napsané položky nemají nic společného s řetězci (není to tedy žádný text, který by šel třeba tisknout). PANSKA, KRIZIKOVA a KARLIN jsou "symbolické konstanty" pro čísla 0 - 2 (první položka v seznamu je číslována od nuly a pak se hodnota zvyšuje o jednu), takže provedu-li pár přiřazovacích příkazů (s definicí proměnných typu SKOLY), budou jejich hodnoty vypadat takto:

```
SKOLY prvni, druha;                 //Definice dvou proměnných typu SKOLY

int main(void)
{
   prvni = PANSKA;                  //Hodnota v prvni bude 0, protože položky ve výčtovém typu
                                    //jsou číslovány od 0
   druha = KARLIN;                  //Hodnota v druha bude 2
   prvni = druha;                   //Hodnota v prvni bude 2
   druha = 78;                      //Toto bohužel lze také, ačkoliv je proměnná druha definována
                                    //jako typ SKOLY, který má hodnoty 0, 1 a 2
}
```

Jednotlivé položky výčtového typu lze i inicializovat:

```
typedef enum {
   PANSKA = 3;
   KRIZIKOVA = 9;
   KARLIN
} SKOLY;
```

Jak je vidno, proměnné lze inicializovat prakticky na jakoukoliv celočíselnou a znaménkovou hodnotu. Pokud se ptáte, jakou hodnotu představuje KARLIN, tak je to 10, protože hodnota položky je vždy o jednu větší (pokud není explicitně inicializována) než hodnota položky předchozí.
Já osobně výčtový typ nikdy nepoužil, což se ale nevylučuje s tím, že jsou v ostatních programech poměrně hojně používány (já si vystačím se symbolickými konstantami).

•15.2 Union

Poslední datový typ, jaký v Céčku známe, je union. Union je proměnná, která naráz může obsahovat proměnné různých typů. Co je ale důležité je fakt, že tyto proměnné sdílejí stejný paměťový prostor, takže zapíšeme-li něco do jedné proměnné, tak se "přemaže" již existující hodnota. Pokud jste to nepochopili, tak nezoufejte, ukážu Vám nejprve definici a poté pár příkazů:

```
typedef union {
   int     cele_cislo;
   float   desetinne_cislo;
} MUJ_UNION;

MUJ_UNION prom;
```

Tímto jsem definoval typ MUJ_UNION, což je union o dvou položkách, které sdílejí stejný prostor. Přístup k jednotlivým položkám je pomocí klasických "strukturových" operátorů, tedy -> (pomocí pointerů) a tečky (klasický přístup). Provedu-li nyní příkaz

```
prom.desetinne_cislo = 3.2;
```

tak se jakoby proměnná prom transformovala na datový typ float (ale je to pořád union!) a její hodnota je 3,2. Pokud ale budu teď přiřazovat tuto hodnotu přes položku cele_cislo do nějaké jiné proměnné příkazem

```
nejaka_promenna = prom.cele_cislo;
```

tak se do nejake_promenne nepřiřadí číslo 3.2, ale nějaká nesmyslná hodnota (resp. ta hodnota smysl má, ale pouze pro desetinná čísla, neboť ty se ukládají do proměnných jiným způsobem), protože jsem toto číslo četl přes položku cele_cislo (což je proměnná typu int). Samozřejmě přiřazením hodnoty

```
prom.cele_cislo = 9;
```

se přemaže původních 3,2.

Použití unionů je buď pro šetření s pamětí (velikost přidělené paměti unionu odpovídá velikosti největší položky) nebo (a to je pro nás mnohem použitelnější) pro čtení příznakové (nebo jakékoliv jiné bitové struktury, pro ostatní druhy to nemá smysl) proměnné, která je definována pomocí struktury. Mějme ještě jednou strukturu z kapitoly 14.2:

```
typedef struct flag_bity {
   unsigned int inter_uart    : 1;      //Vyhrazení jednoho bitu pro inter_uart
   unsigned int inter_spi     : 1;      //Vyhrazení jednoho bitu pro inter_spi
   unsigned int mod           : 6;      //Vyhrazení šesti bitů pro mod
   unsigned int nevyuzito     : 8;      //Zbylých 8 bitů je přiřazeno proměnné nevyuzito
} FLAG_BITY;
```

a proměnnou

```
FLAG_BITY nastavovaci;
```

Nyní jsme schopni pomocí klasických operátorů pro práci se strukturou číst nebo zapisovat do jednotlivých položek (nebo chcete-li bitů) nějaké hodnoty. Co ale nejsme schopni zjistit, je hodnota celé proměnné nastavovaci (nemůžeme prostě provést příkaz nejaka_promenna = nastavovaci, neboť tím přiřadíme pouze hodnotu první položky), což je výhoda příznakové nestrukturované proměnné (ty lze číst celé). Pomocí unionu ale můžeme tuto poměrně užitečnou vlastnost přenést i na bitové struktury:

```
typedef struct flag_bity {
   unsigned int inter_uart    : 1;      //Vyhrazení jednoho bitu pro inter_uart
   unsigned int inter_spi     : 1;      //Vyhrazení jednoho bitu pro inter_spi
   unsigned int mod           : 6;      //Vyhrazení šesti bitů pro mod
   unsigned int nevyuzito     : 8;      //Zbylých 8 bitů je přiřazeno proměnné nevyuzito
} FLAG_BITY;

typedef union {
   FLAG_BITY    jednotlive;
   unsigned int celkove;
} PRIZNAKOVA;

PRIZNAKOVA nastavovaci;

int main(void)
{
   unsigned int pom;

   nastavovaci.celkove = 0;                  //Nulování celé proměnné nastavovaci (všech bitů)
   nastavovaci.jednotlive.inter_uart = 1;    //Nastavení 0. bitu, hodnota v nastavovaci je 1
                                             //(0000 0000 0000 0001 binárně)
   nastavovaci.jednotlive.inter_spi = 1;     //Nastavení 1. bitu, hodnota v nastavovaci je 3
                                             //(0000 0000 0000 0011 binárně)
   nastavovaci.jednotlive.mod = 58;          //Nastavení 2. až 7.bitu, hodnota v nastavovaci je
                                             //235 (0000 0000 1110 1011 binárně)
   nastavovaci.jednotlive.nevyuzito = 255;   //Nastavení 8. až 15. bitu, hodnota v nastavovaci je
                                             //65 515 (1111 1111 1110 1011 binárně)
   pom = nastavovaci.celkove;                //Přiřazení hodnoty 65 515 do proměnné pom
}
```

Posledním příkazem nepřistupuji k proměnné nastavovaci jako ke struktuře, ale jako k jednolité proměnné typu unsigned int s hodnotou 65 515.

•16. Závěr a doporučená literatura

Tak, to by bylo všechno šmoulové. Znalosti, které jste momentálně nabyli, jsou dostatečné na to, abyste se pustili do vlastních malých projektů a začali zkoumat různá zákoutí jazyka C. Samozřejmě, že tyto znalosti nejsou úplné (hodně věcí jsem vynechal, protože mě přišlo zbytečné Vás momentálně seznamovat s některými specifickými drobnostmi, byť jsou důležité pro Váš budoucí vývoj), je proto nutné se zdokonalovat. Tímto bych Vás chtěl odkázat na některou ze zde uvedených knížek, které Vám pomůžou s rozšířením obzorů a kdoví, třeba se tím budete jednou i živit?

●

[1] Pavel Herout: Učebnice jazyka C, 1994
Patrně nejlepší česky psaná učebnice na jazyk C (na PC, ale výklad je samozřejmě použitelný i na mikrokontroléry). Principy a syntaxe jsou vysvětleny velice dobře (pochopí to i cvičený Labrador) a díky přiloženým úlohám k procvičení si můžete namáhat hlavu pěkně dlouho (řešení lze stáhnout ze stránek nakladatelství Kopp).

[2] Lucio Di Jasio: Programming 16-bit Microcontrollers in C, 2007
Bohužel, co se týče problematiky programování našich dsPICů v C, je česky psaná literatura prakticky nedostupná (nepočítám procesory ATMEL, na ty se sehnat nějaká česká literatura dá), takže musíme hledat jinde, nejlépe v anglickém jazyce. Tato kniha se zabývá spíše praktickým využitím jazyka C (syntaxe je zde zmíněna opravdu jen letmo) na různých periferiích (např. LCD, UART, SD karta a dokonce i televize!). Pokud tedy už jazyk ovládáte na poměrně slušné úrovni a dokážete se v něm orientovat, tak Vám tato lidsky psaná knížka velice pomůže (ikdyž to není na dsPIC procesory, ale na řadu 24, což jsou 16-bitové procesory, takže princip je prakticky stejný).

[3] Microchip Inc.: MPLAB C Compiler for PIC24 and dsPIC User´s Guide, 2008
Není nad oficiální literaturu od Microchipu. Zabývá se především propojení jazyka C s procesory, takže zde nenajdete popis syntaxe a ani příklady v ní nehledejte. Je to pouze suchý výčet, co všechno kompiler dokáže. Osobně bych si tuto knihu vzal do ruky až v posledním případě (tím nemyslím, že je to kniha nepřesná či chybná, ale že pro normálního člověka je těžko stravitelná - je to holt manuál)

[4] Microchip Inc.: datasheet dsPIC30F3013, 2005
Jediná kniha, která Vás bude pronásledovat ve dne i v noci kamkoliv se hnete. Bez datasheetu k dsPICu nejste schopni napsat vůbec nic, i kdybyste programovali v Céčku jako Bozi.