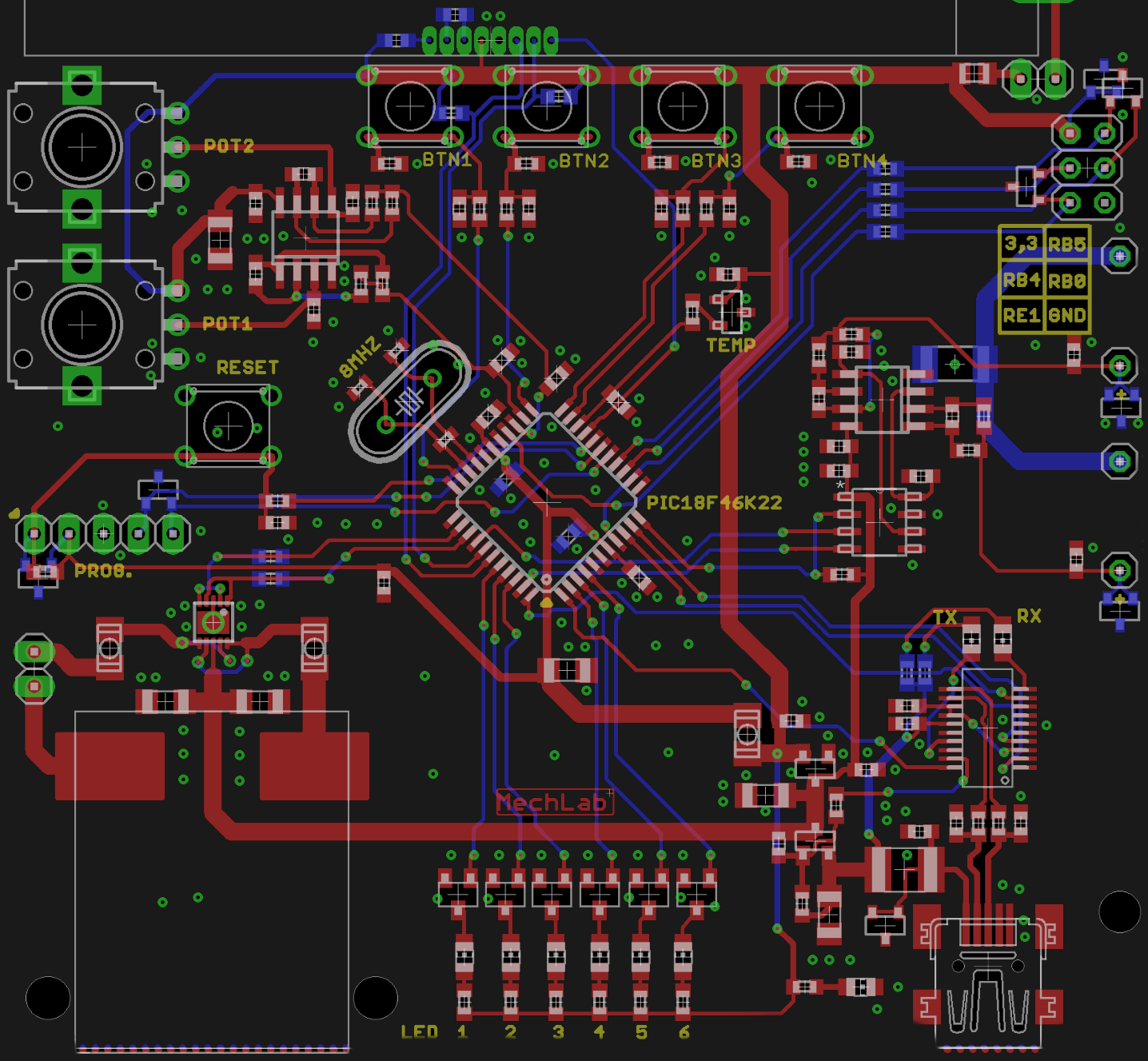
Apliklace embedded systémů v mechatronice



Obsah

1

Programovací jazyk C

1.1 Jazyk C

Jazyk C je nízko úrovňový programovací procedurální jazyk, který se stal velice populárním a v embedded systémech je dnes stále číslo jedna i přes to, že jeho stáři je více než 40 let. Vznikal v době vývoje operačního systému Unix, jehož jádro je napsáno převážně v tomto jazyku. Na desktopu je již dnes nahrazen svým následovatelem C++, který podporuje například objektově orientovaný návrh.

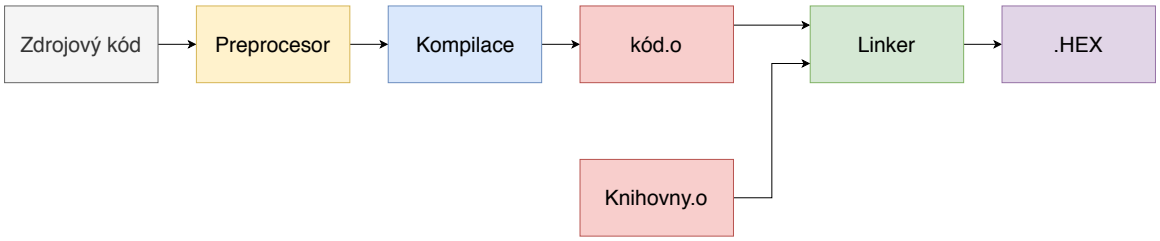
Výhodou jazyka C je jeho blízkost k hardwaru, z tohoto důvodu jsou programy v něm napsané velice efektivní a výpočetně rychlé v porovnání s dalšími moderními jazyky. Velkou výhodou a zároveň i nevýhodou je možnost přímo pracovat s pamětí. V jazyce C se zavadí k tomuto účelu tzv. ukazatel. Nevhodná práce s ukazateli však může vést k chybám. Jazyk C je přenositelný a kompilovaný. Překladač jazyka vytvoří strojový kód pro příslušnou platformu. Na desktopu se používají při programování standardní knihovny jazyka. Tyto knihovny obsahují mnoho funkcí pro práci s HW, výpis do terminálu, matematické funkce apod.

K zachování přenositelnosti bylo zavedeno několik standartu např. ANSI/ISO C a jejich verze C89, C90 a C99. Dodržení těchto norem zaručuje přenositelnost mezi platformami jako OS Windows a linux. Kód se tedy napíše pouze jednou a kompiluje se pro příslušnou platformu.

V procedurálních jazycích se vykonávají předdefinované instrukce krok po kroku. V jazyce C lze pak nadefinovat funkce, které vykonávají další posloupnosti příkazů.

1.1.1 Překlad kódu

K překladu kódu z vyššího jazyka se používá softwarový nástroj překladač anglicky compiler. Ten ze zdrojového kódu vytvoří strojový kód pro procesor. Mít k dispozici překladač určen pro naši platformu je tedy nezbytné k vytvoření spustitelného kódu. Průběh překladu kódu lze vidět na 1.1. Lze jej rozdělit na preprocesor, který pracuje s textem zpracovává například uživatelská makra. Kompilátor, který překládá vyšší jazyk na nižší strojový kód a také assembler. Linker spojuje jednotlivé soubory a knihovny dohromady a také určuje umístění/alokaci proměnných v paměti.



Obr. 1.1: Průběh kompilace programu

1.2 Základní práce s C

V prvé řadě je nutné podotknout, že tato kapitola představuje opravdu rychlý přehled programovacího jazyka C. Poslouží tak spíše jako základní přehled pro studující, kteří mají již nějaké zkušenosti s programováním. Zájemce o komplexní výklad se bude muset porozhlédnout po jedné z mnoha učebnic jazyka C. Naopak člověk, který již nějaký ten program vytvořil třeba v Matalbu, a nebo v jiném vysokoúrovňovém programovacím jazyce, by se měl rychle zorientovat.

Následující jednoduchý kód v jazyce C vytiskne hlášku, bez které by nebylo možné žádný kurz programování vůbec začít. Povšimněte si prosím funkce main(). Jedná se v jazyce C o funkci zvláštní. Tato funkce udává místo odkud (zjednodušeně řečeno) kód začíná. Tak jako každá funkce v C může mít návratovou hodnotu, v tomto případě proměnnou typu int. V závorce se nachází slovo void, které v C značí prázdný datový typ. V tomto případě funkce žádné hodnoty neočekává. Příkazem return je program ukončen. Funkcím bude věnován samostatný výklad dále.

Minimalistický kód v jazyce C:

```
// komentar na jeden radek
/* komentar na vice radku
================ funkce main =============*/
int main(void) {

    printf("Ahoj svete!");  // Tisk na terminal
    return 0;       // funkce main ocekava vraceni hodnoty int
}
```

1.2.1 Datové typy

Jazyk C není dynamicky typovaný tak jako například Matlab nebo Python. Prakticky to znamená, že programátor musí určit datový typ proměnné, kterou v programu zakládá. Rozsah datových typů může však být ovlivněn platformou na kterou kód překládáme.

Záleží tedy i na konkrétním překladači. V dokumentaci pro překladač microchip XC8 nalezneme tabulku podporovaných datových typů.

Datové typy v XC8:

Typ	Velikos(bit)	Popis
bool	1	boolean
char	8	-128..127
unsigned char	8	0..255
int	16	-32768..32767
unsigned int	16	0..65535
long	32	-2147483648..2147483647
unsigned long	32	0..4294967295
float	24/32	realné číslo
double	24/32	realné číslo

Prace s proměnnými v C:

```
// nacteni pouzitych knihoven  stdio.h obsahuje funkci printf
#include <stdio.h>

void main(void){
    // definice promennych typu int
    int a = 10;
    int b = 3;
    int vysledek1;
    float vysledek2;

    vysledek1 = a + b;
    vysledek2 = (float)a / (float)b;

    //vypis a + b na terminal
    printf("Vysledek je: %d\n",vysledek1);
    //vypis a / b na terminal
    printf("Vysledek je: %f\n",vysledek2);
}
```

1.2.2 Operátory

Jazyk C se vyznačuje poměrně bohatou sadou operátorů. Nejčastěji používané uvedeme v následujících tabulkách.

Matematické operátory:

Operátor	Význam
+	plus
++	ikrementuje hodnotu o jedna
-	mínus
--	dekrementuje hodnotu o jedna
*	násobení
/	dělení
%	modulo

Bitové operátory:

Operátor	Význam
&	bitové AND
|	bitové OR
^	bitové XOR
~	bitová negace
<<	bitový posun doleva
>>	bitový posun doprava

Porovnávací operátory:

Operátor	Význam
<	operátor menší
<=	menší nebo rovno
>	operátor větší
>=	větší nebo rovno
==	operátor rovnosti
!=	operátor nerovnosti
&&	logické AND
||	logické OR
!	logické NOT

Další speciální operátory:

Operátor	Význam
&	reference
*	dereference
sizeof()	získání velikosti
(typ)	přetypování

1.2.3 Větvení programu IF-ELSE

Abychom něco kloudného naprogramovali, musíme být schopni řídit tok programu. K tomuto větvení budeme používat reakci na určité podmínky. V jazyce C si k tomuto účelu představíme podmínky if-else a také switch-case. Čtenáři, kteří již mají zkušenosti s textovým programovacím jazykem, museli na řízení toku programu pomocí podmínek již narazit. Pomocí podmínek ověřujeme zda nastala konkrétní situace. Můžeme například vyhodnotit zda logická proměnná nabyla hodnoty True. Pokud ano, tak provedeme blok programu. Pokud ne, tento blok přeskočíme a vyhodnotíme jiný blok.

Použití IF-ELSE v C:

```
// nacteni pouzitych knihoven  stdio.h obsahuje funkci printf
#include <stdio.h>

void main(void){
    // definice promennych typu int
    int a = 10;
    int b = 5;
    int vysledek;

    if (a<=b){
        vysledek = a + b; //secteni a + b
        printf("Soucet je: %d",vysledek);
    }
    else{
        vysledek = a - b; //secteni a - b
        printf("Soucet je: %d",vysledek);
    }
}
```

1.2.4 SWITCH-CASE

Použití je vhodné v případě pokud tok programu řídí jedna proměnná a možností je větší množství. Pozor na vynechání příkazu break. Bez klíčového slova break se bude provádět vyhodnocování dále. Program který bude jako řídící proměnnou používat datový typ float, nebo double nepůjde zkompilovat.

Použití switch-case v C:

```
// nacteni pouzitych knihoven  stdio.h obsahuje funkci printf
#include <stdio.h>

void main(void){
    // vytvoreni promnne typu char
    char znak = 'B';

    // pokud se ridim pomoci jedne promenne je switch lepsi volba (ta musi byt cele cislo
    )
    switch (znak){

        case 'A':
            printf("A");
            break; // vzdy ukonsit prikazem break
        case 'B':
            printf("B"); //toto bnude vytisknuto
            break;
        case 'C':
            printf("C");
            break;
    }

}
```

1.2.5 Cykly for a while

Další důležitou součástí programování je využití cyklů. Pokud chci určitou část programu vícekrát opakovat je jistě vhodné zavést koncept, který mi to umožní provést jinak než tak, že daný kód 100x zopakuji. K tomuto slouží cykly. Dva nejdůležitější jsou FOR a WHILE. Každý je určen k rozdílnému problému.

Cyklus for se využívá v případě, že znám počet cyklů, které chci uskutečnit. Počet opakování proběhne na základě řídící proměnné.

For cyklus v C:

```
// nacteni pouzitych knihoven  stdio.h obsahuje funkci printf
#include <stdio.h>

/*===============main=========================================*/
void main(void){
    // definice promenne typu int
    int i;
  // for smycka vytiskne na  do terminalu 0..10
    for(i=0;i<=10;i++){
        printf("Hodnota i je:%d\n",i);
    }
}
```

Cyklus while se použije v případě, že počet cyklů neznám dopředu. Cyklus se provádí

tak dlouho dokud platí podmínka.

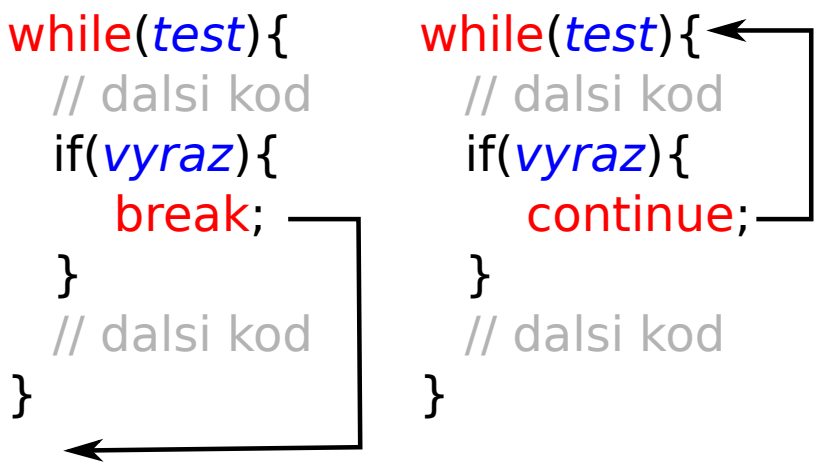
While cyklus v C:

```c
#include <stdio.h>
void main(void)
{
  // definice promenne typu int
    int i = 1;
    // vytiskne se 1..10
    while (i <= 10)
    {
        printf("%d\n", i);
        ++i;
    }
}
```

1.2.6 Příkazy break a continue

Klíčová slova break a continue lze používat ve smyčkách for a while. Pomocí příkazu continue se pokračuje další iterací, naopak příkaz break cyklus ukončí.

Použití příkazů break a continue:



1.3 Funkce

Pokud náš program obsahuje větší množství kódu, který se navíc často opakuje, využijeme možnost funkcionální dekompozice. V podstatě to znamená, že uživatel může posloupnost příkazů uzavírat do funkcí, také podprogramů. Abychom mohli funkci používat je nutné jí definovat, tedy zapsat její hlavičku i kód někam do globálního prostoru.

Pokud chci použít funkci v kódu před její definicí (překlad programu probíhá od shora), mohu využít pouze deklaraci zapsáním hlavičky funkce bez definice. Překladači tento ”prototyp”někde existující funkce postačí. Oba přístupy definicí před funkcí main a pod ní uvedeme v následujících příkladech.

Použití funkce v C:

```c
// nacteni pouzitych knihoven  stdio.h obsahuje funkci printf
#include <stdio.h>

int soucet(int a, int b);   //deklarace funkce soucet()
/*===============main=======================================*/
void main(void){
    // definice promennych typu int
    int a = 10;
    int b = 5;
    //vypis a + b na terminal s puzitim funkce soucet()
    printf("Soucet je: %d",soucet(a,b));
}

// definice funkce soucet()
int soucet(int a, int b){
    return a + b;
}
```

Použití funkce v C (Definice před funkcí main()):

```c
// nacteni pouzitych knihoven  stdio.h obsahuje funkci printf
#include <stdio.h>

// definice funkce soucet()
int soucet(int a, int b){
    return a + b;
}
/*===============main=======================================*/
void main(void){
    // definice promennych typu int
    int a = 10;
    int b = 5;
    //vypis a + b na terminal s puzitim funkce soucet()
    printf("Soucet je: %d",soucet(a,b));
}
```

Při definici funkce uvádíme návratovou hodnotu, respektive její datový typ. Dále datové typy proměnných, které funkci předáváme. Funkce bez návratové hodnoty a bez vstupních parametru se zapíše jako void fun(void). Funkci bez návratové hodnoty se někdy říká procedura.

1.4 Pole a ukazatele

[0]	[1]	[2]	[3]	[4]	[5]
19	22	0	255	10	47

Obr. 1.2: Prvky pole jsou v paměti seřazeny za sebou

Použití ukazatelů v C:

```c
#include <stdio.h>
/*prototyp funkce prohod vstupem je dvojice ukazatelu*/
void prohod(int* a, int* b);

int main(void){
    /* deklarace/definice promennych */
    int jedna = 1;
    int dva = 2;
    /* deklarace ukazatelu */
    int *p_jedna;
    int *p_dva;
    /* ziskani adres pomoci reference */
    p_jedna = &jedna;
    p_dva = &dva;
    /* funkce prohod */
    prohod(p_jedna, p_dva);
    printf("jedna = %d; dva = %d\n", jedna, dva);
    return 0;
}

void prohod(int* a, int* b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Použití ukazatele na pole v C:

```c
#include <stdio.h>

void uloz_do_pole(int pole[], int index, int cislo);

int main() {

    int cisla[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    printf("%d\n", cisla[7]);

    uloz_do_pole(cisla, 7, 3);

    printf("%d\n", cisla[7]);

    return 0;
}
```

```
void uloz_do_pole(int pole[], int index, int cislo){
    pole[index] = cislo;
}
```

1.5 Struktury

Struktury jsou v programovacím jazyku C kolekce (složený datový typ), které sdružují více proměnných pod jedním názvem. Proměnné ve struktuře můžou být různých datových typů. K této struktuře se potom přistupuje prostřednictvím ukazatele. Struktury jsou uloženy v souvislém kusu paměti.

Strukturu v C:

```
#include <stdio.h>
#include <string.h>

/*vytvoreni uzivatelskeho datovejo typu*/
typedef struct{
    char    jmeno[25];
    int     vek;
    int     vyska;
} clovek;

int main() {
    /*promene typu clovek a ukazatel*/
    clovek Petr = {"Petr Novak", 25, 178};
    clovek Michal;
    clovek *p_Petr = &Petr;
    /*pristup k promennym ve strukture pres tecku .*/
    Michal.vek = 30;
    Michal.vyska = 189;

    strcpy(Michal.jmeno, "Michal Mukl"); // pouziti knihovni funkce

    printf("Petr ma %d let\n",  Petr.vek);
    printf("Michal se jmenuje %s\n",  Michal.jmeno);
    /*ukazatel na strukturu pouziva se ->*/
    printf("Petr ma %d centimetru\n",  p_Petr->vyska);

    return 0;
}
```

1.6 Direktivy preprocesoru

text

1.6.1 Podmíněný překlad kódu

Použití podmíněného překladu

```
#include <stdio.h>
#include <stdlib.h>

#define PRELOZIT

int main()
{
    #ifdef PRELOZIT
    printf("Prvni\n");
    #endif
    printf("Druhy\n");
    return 0;
}
```

1.7 Vlastní knihovny

Jazyk C umožňuje tvorbu vlastních knihoven. programátor si může v knihovnách naprogramovat vlastní funkce, které mohou sloužit jako ovladače pro práci s hardwarem a nebo algoritmy, které častěji využívá. Pro vytvoření uživatelské knihovny budete potřebovat založit hlavičkový soubor s příponou.h a zdrojový soubor s příponou .c. Oaba tyto soubory mají stejný název např. MojeFunkce. V hlavičkovém souboru je zvykem ošetřit podmíněným překladem možnost vícenásobného načtení knihovny. V hlavičkovém souboru poté zavedu makra, proměnné a prototypy funkcí. Ve zdrojovém souboru poté programuji definice funkcí. Pokud chci potom v programu funkci z knihovny použít stačí předtím knihovnu zahrnout pomocí include ”MojeFunkce.h”.

Hlavičkový soubor Knihovna.h:

```
// direktiva preprocesoru pro podmineny preklad
// jedna se o ochranu proti vicenasobnemu nacteni knihovny
#ifndef KNIHOVNA_H_INCLUDED
#define KNIHOVNA_H_INCLUDED
// deklarace funkce najdi()
int najdi(char str[], char pismeno, char velikost);

#endif // KNIHOVNA_H_INCLUDED
```

Zdrojový soubor Knihovna.c:

```
//nacteni hlavickoveho souboru
#include "Knihovna.h"
//definice funkce najdi
int najdi(char str[], char pismeno, char velikost){
    int i;
    for (i=0; i<(velikost-1);i++){
```

```
        if (str[i] == pismeno){
            return i; //vratim prvni nalez; return funkci ukonci
        }
    }
    return -1;
}
```

Zdrojový soubor main.c:

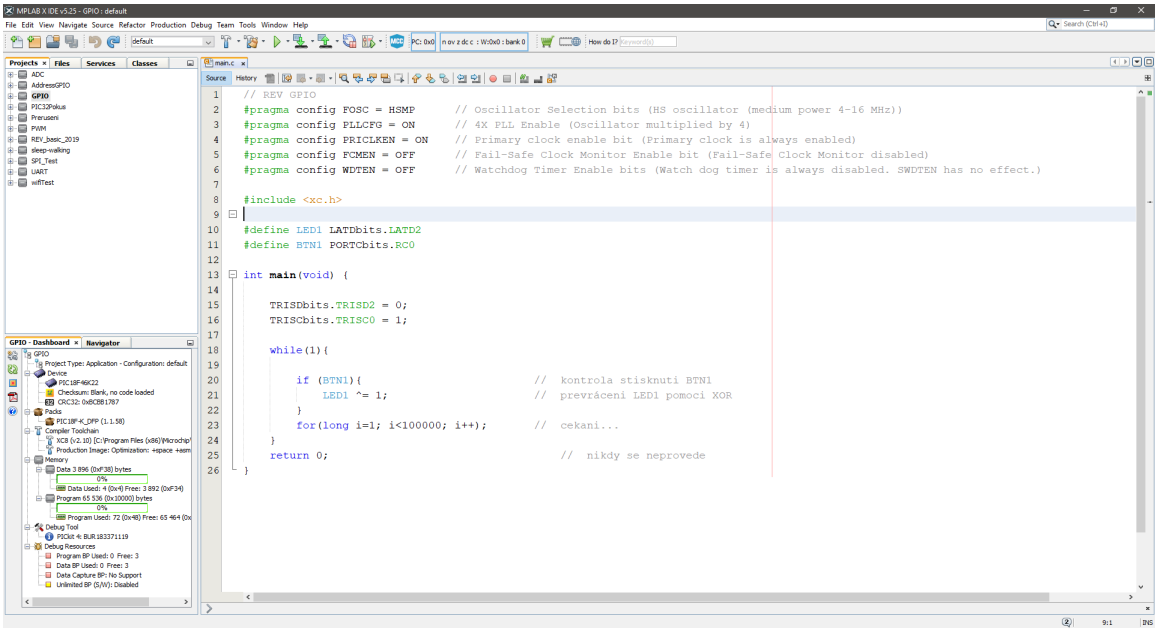
```
#include <stdio.h>
#include "Knihovna.h" // nacteni knihovny

int main(void)
{
    char slovo[20] = "ahoj jak se mas?";

    int index = najdi(slovo,'k', sizeof(slovo)); // pouziti funkce z knihovny

    printf("Index je: %d\n", index);
    return 0;
}
```

2

Programování PIC18

2.1 Vývojové nástroje

Pro práci s mikrokontroléry Microchip nabízí zdarma vývojové prostředí MPLAB. Ke stažení je také kompilátor XC8, který je pro bezplatné užití omezen. Omezení se týká úrovně možných optimalizací překladu do strojového kódu. Pro nahrání programu do paměti mikrokontroléru a pro debugging programu je potřeba použít speciálního zařízení tzv. programátoru. Michrochi nabízí například PicKit3 a PicKit4.



Obr. 2.1: MPLAB

2.2 Základní nastavení MCU

2.2.1 Pojistky

Pojistky, v případě názvosloví firmy Mikrochip konfigurační bity. Jsou speciální místa v programové paměti, která se nastavují během zavadění programu. Tyto pojistky obsahují důležité informace pro běh mikrokontroléru jako zdroj a nastavení oscilátoru, nastavení watchdog timeru a nebo uzamčení paměti. Konfigurační byty lze v prostředí MPLAB nastavit pomocí přehledné tabulky. Nastavení nalezneme v záložce windows/target memory views/configuratin bits. Po vyplnění lze konfigurační byty vložit do globálního prostoru zdrojové kódu. Pokud kód neobsahuje nastavení všech konfiguračních bitů, bude použita defaultní hodnota. Během kompilace je uživatel upozorněn prostřednictvím warningu.

Základní nastavení pro PIC18:

```
#pragma config FOSC = HSMP      // Oscillator Selection bits
#pragma config PLLCFG = ON      // 4X PLL Enable
#pragma config PRICLKEN = ON    // Primary clock enable bit
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enable bit
#pragma config WDTEN = OFF      // Watchdog Timer Enable bits
```

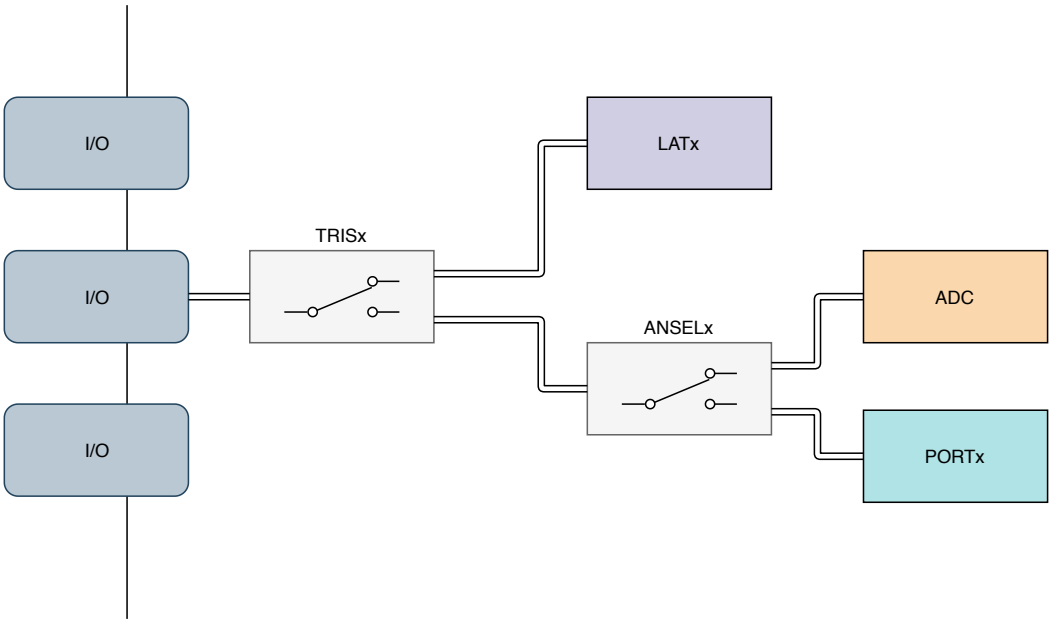
2.2.2 Oscilator

Obr. 2.2: todo

2.3 GPIO piny

Nejzákladnější periferii mikrokontroléru je obecný vstup/výstup. Ve výstupním režimu mohu z programu zapsat na příslušný pin logickou 0, nebo 1. To odpovídá úrovni napětí na pinu. V režimu vstupním mohu naopak získat současnou úroveň napětí na pinu. Jedná se tedy o základní možnost interakce s ostatními zařízeními (spínače, bezpečnostní prvky).

2.3.1 Inicializace GPIO



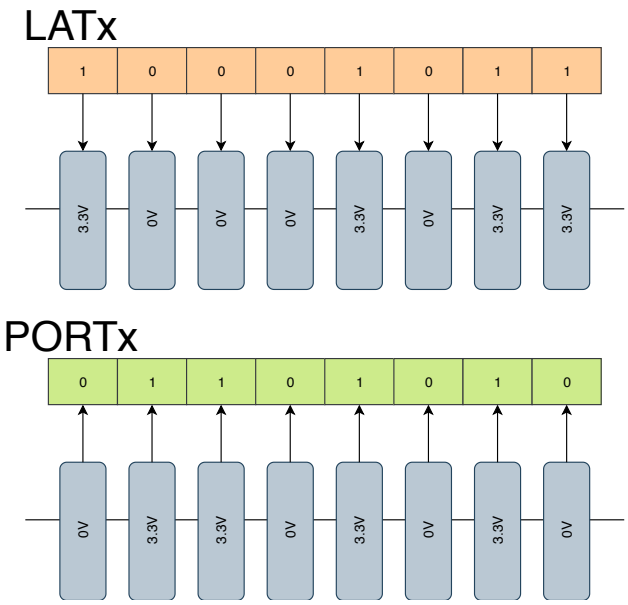
Obr. 2.3: Zjednodušený model GPIO pinu

Příklad inicializace GPIO v C:

```
void init(void){

    // Vypnuti ADC funkci na pinech, ktere budou konfigurovany jako vstupy
    ANSELA = 0x00; // Vypnuti ADC na portu RAx
    ANSELC = 0x00; // Vypnuti ADC na portu RCx

    // Nastaveni pinu RD2,RD3,RC4,RD4,RD5,RD6 jako vystup
    TRISDbits.TRISD2 = 0;
    TRISDbits.TRISD3 = 0;
    TRISCbits.TRISC4 = 0;
    TRISDbits.TRISD4 = 0;
    TRISDbits.TRISD5 = 0;
    TRISDbits.TRISD6 = 0;

    // Nastaveni pinu RA4,RA3,RA2,RC0 jako vstup
    TRISAbits.TRISA4 = 1;
    TRISAbits.TRISA3 = 1;
    TRISAbits.TRISA2 = 1;
    TRISCbits.TRISC0 = 1;
}

void main(void)
{
  // provedeni inicializacni funkce
    init();

    while(1){
```

```
    // dalsi uzivatelsky kod
    }

}
```

2.3.2 Zápis/čtení GPIO



Obr. 2.4: Čtení/zápis GPIO

GPIO v C:

```
#include <xc.h>

#pragma config FOSC = HSMP      // Externi oscilator
#pragma config PLLCFG = ON      // 4X PLL
#pragma config FCMEN = ON       // Fail-Safe Clock
#pragma config WDTEN = OFF      // Watchdog Timer OFF
#define _XTAL_FREQ 32E6

void init(void){
    ANSELA = 0x00;
    ANSELC = 0x00;
    // RD2 a RD3 jako vystup
    TRISDbits.TRISD2 = 0;
    TRISDbits.TRISD3 = 0;
    // RC0 jako vstup
    TRISCbits.TRISC0 = 1;
}
/*--------------------main---------------*/
void main(void)
{
    init();
```

```
    while(1){
        LATDbits.LATD2 = PORTCbits.RC0;     // Pin RD2 stejny jako RC0
        LATDbits.LATD3 = ~LATDbits.LATD3;   // Prevraceni pinu RD3
        __delay_ms(100);                    // Funkce z xc8.h nutne #define _XTAL_FREQ
        }
    }
```

GPIO s použitím uživatelských maker:

```
#include <xc.h>

#pragma config FOSC = HSMP      // Externi oscilator
#pragma config PLLCFG = ON      // 4X PLL
#pragma config FCMEN = ON       // Fail-Safe Clock
#pragma config WDTEN = OFF      // Watchdog Timer OFF
#define _XTAL_FREQ 32E6

// uzivatelska makra
#define LED1 LATDbits.LATD2
#define LED2 LATDbits.LATD3

#define BTN PORTCbits.RC0

void init(void){
    ANSELA = 0x00;
    ANSELC = 0x00;
    // RD2 a RD3 jako vystup
    TRISDbits.TRISD2 = 0;
    TRISDbits.TRISD3 = 0;
    // RC0 jako vstup
    TRISCbits.TRISC0 = 1;
}
/*--------------------main---------------*/
void main(void)
{
    init();

    while(1){
        LED1 = BTN;     // Pinu RD2 je nastavena stejna hodnota jako RC2
        LED2 = ~LED2;   // Prevraceni pinu RD3
        __delay_ms(100);// Knihovni funkce,  je treba #define _XTAL_FREQ
        }
    }
```

2.4 Čítače

text

2.4.1 Inicializace a využití čítače

Využití čítače:

```
void init(void){


    TRISD = 0x00  // nastaveni RD pinu jako vystup

    // Inicialzace Timeru 1
    T1CONbits.T1CKPS = 0b11;    // TMR1 nastaveni preddelicky
    T1CONbits.TMR1ON = 1;       // TMR1 zapnuti
}

void main(void)
{
  // provedeni inicializacni funkce
    init();

    while(1){

    if (TMR1 < 0x8000); // cekani na nacteni poloviny rozsahu 32768
          LATD2 = ~LATD2; // otoceni digitalniho pinu RD2
          TMR1 = 0;   // vynulovani timeru

      }
    }

}
```

2.5 Přerušení

text

Využití přerušení:

```
// zapis preruseni v XC8 C99 pokud neni upresneno HIGHT priorita na 0x0008
void __interrupt() ISR(void){
  // kontrola povoleni preruseni a priznak
   if(TMR1IE & TMR1IF){
        TMR1 = 0x8000;    //Nastaveni hodnoty registru TMR1
        LED1 ^= 1;      // zmena stavu LED1(makro)
        TMR1IF = 0;     // smazani priznaku preruseni dulezite!!!
   }
}

void init(void){
```

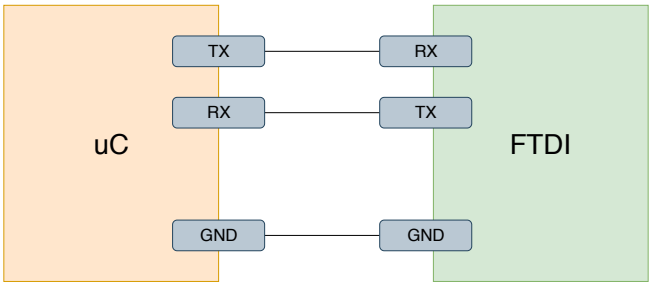
```
    TRISD = 0x00  // nastaveni RD pinu jako vystup

    // Inicialzace Timeru 1
    T1CONbits.TMR1CS = 0b00;  // TMR1 zdroj signalu
    T1CONbits.T1CKPS = 0b11;    // TMR1 prescaler
    T1CONbits.TMR1ON = 1;       // TMR1 on

    // konfugurace preruseni
    PEIE = 1;           // povoleni preruseni od periferii
    GIE = 1;            // povoleni preruseni globalne
    TMR1IE = 1;         // preruseni od TMR1 (preteceni registru TMR1)
}

void main(void)
{
  // provedeni inicializacni funkce
    init();

    while(1){
        // prazdna hlavni smycka
      }
    }

}
```

Více zdrojů přerušení:

```
// zapis preruseni s vice zdroji TMR1 a TMR5
void __interrupt() ISR(void){
   if(TMR1IE & TMR1IF){
        TMR1 = 0x8000;
        LED1 ^= 1;
        TMR1IF = 0;
   }
   if(TMR5IE & TMR5IF){
        TMR5 = 0;
        LED2 ^= 1;
        TMR5IF = 0;
   }
}
```

2.6 Uart

text



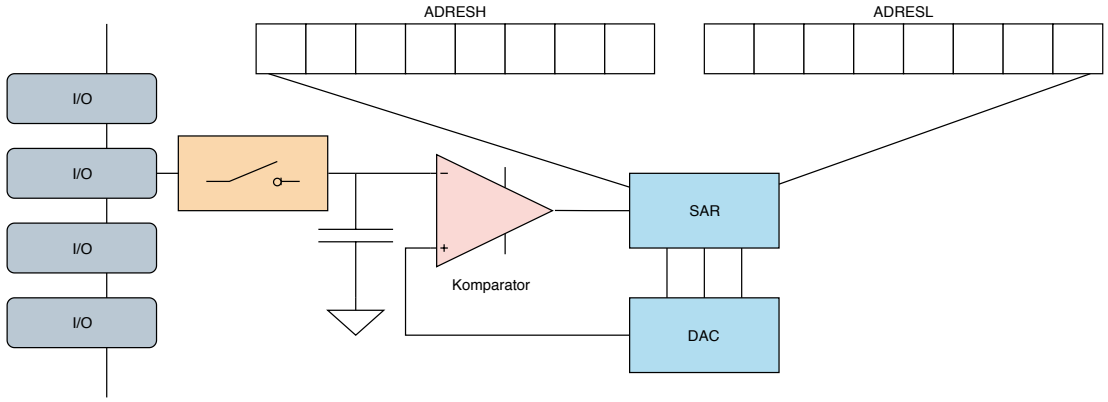
Obr. 2.5: UART

UART příjem a odeslaní znaku:

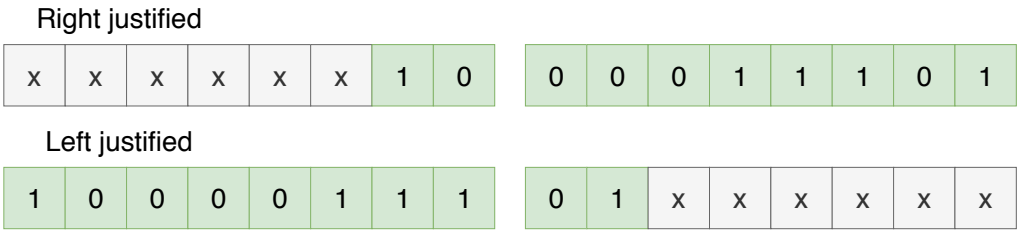
```
#include <xc.h>                //-- pro prekladac XC8

#pragma config FOSC = HSMP      // Externi oscilator
#pragma config PLLCFG = ON      // 4X PLL
#pragma config FCMEN = ON       // Fail-Safe Clock
#pragma config WDTEN = OFF      // Watchdog Timer OFF

/*--------main--------*/
int main(void) {

    ANSELC = 0x00;
    TRISB = 0x00;
    TRISCbits.TRISC6 = 0;   // TX pin jako vystup
    TRISCbits.TRISC7 = 1;   // rx pin jako vstup

    /*baudrate*/
    SPBRG = 52;     // (16_000_000 / (64 * 9600)) - 1
    RCSTAbits.SPEN = 1;       // zapnuti UART
    TXSTAbits.TXEN = 1;       // zapnuti TX
    RCSTAbits.CREN = 1;       // zapnuti RX

    while(1){
        if (PIR1bits.RCIF){
            LATB0 = ~LATB0;     // LED
            TXREG = RCREG;      // precist a poslad zpet
        }
    }
}
```

2.7 AD převodník

text



Obr. 2.6: Schema ADC



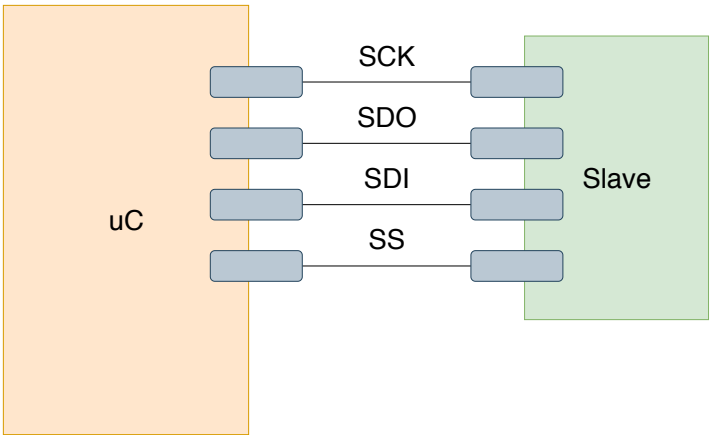
Obr. 2.7: Schema ADC

2.8 Capture/compare a PWM

Obr. 2.8: GPIO

2.9 SPI a I²C

text



Obr. 2.9: SPI

Generovaní pilového průběhu na DAC:

```
#include <xc.h>

#pragma config FOSC = HSMP      // Externi oscilator
#pragma config PLLCFG = ON      // 4X PLL
#pragma config FCMEN = ON       // Fail-Safe Clock
#pragma config WDTEN = OFF      // Watchdog Timer OFF

#define _XTAL_FREQ 32E6         // definice fosc pro knihovnu
#define DAC_SS LATB3            // DAC slave select pin
#define DAC_CH1 0b1001          // kanal A
#define DAC_CH2 0b0001          // kanal B

void init(void){
    /* vyber pinu jako vystupy */
    TRISCbits.TRISC3 = 0;
    TRISCbits.TRISC5 = 0;
    TRISBbits.TRISB3 = 0;
    TRISDbits.TRISD2 = 0;

    LATBbits.LATB3 = 1;         // DAC SS off

    SSP1CON1bits.SSPM = 0b0010; // SPI Master mode, clock = Fosc/16
    SSP1CON1bits.SSPEN = 1;     // SPI zapnuto
}
void SPIWrite(char chanel ,char data);

void main(void) {
    init(); // provedeni inicializace
    unsigned char i = 0;

    /* hlavni smycka */
```

```
    while(1){
        if (i>=255) i = 0;
        SPIWrite(DAC_CH1,i);     // zapis na DAC
        LATD2 = ~LATD2;          //prevraceni led
        __delay_ms(1);
        i++;
    }
}
/* funkce zapisu SPI funkce zapisuje dva bajty za sebou */
void SPIWrite(char channel,char data){

    unsigned char msb, lsb;
    msb = (channel<<4) | (data>>4); // prvni bajt
    lsb = (data<<4);             // druhy bajt
    DAC_SS = 0;                      // slave select
    PIR1bits.SSPIF = 0;          // vynulovani priznaku SPI
    SSPBUF = msb;                // zapis do bufferu
    while(PIR1bits.SSPIF == 0)NOP(); // pockat nez SPI posle prvni bajt

    PIR1bits.SSPIF = 0;          // vynulovani priznaku SPI
    SSPBUF = lsb;                // zapis do bufferu
    while(PIR1bits.SSPIF == 0)NOP(); // pockat nez SPI posle druhy bajt

    NOP();               // paza jedna instrukce
    DAC_SS = 1;          // vzpnout slave select
}
```

Obr. 2.10: todo

2.10 SLEEP mód

Obr. 2.11: todo

2.11 Watchdog

Watchdog timer je bezpečnostní periferie mikrokontroléru. Nastavení je u PIC18 prováděno pomocí konfiguračních bytů. Jedná se o čítač, který je nezávislý na běhu ostatních částí zařízení. Po přetečení registru čítače dojde k resetování. Registr čítače je proto třeba pravidelně smazat. Pro PIC18 je pro tento účel k dispozici instrukce CLRWDT. Pomocí nastavení WDTPS v konfiguračních bitech mohu nastavit periodu, kdy k resetu dojde.

Watchdog:

```
#include <xc.h>

#pragma config FOSC = HSMP
#pragma config PLLCFG = ON
#pragma config PRICLKEN = ON
#pragma config FCMEN = OFF
#pragma config WDTEN = ON        // zapnuti watchdog timeru
#pragma config WDTPS = 512       // nastaveni delicky na 1/((31kHz/128)/512) = 2.1s

// uzivatelska makra
#define _XTAL_FREQ 32E6
#define LED1 LATDbits.LATD2
#define BTN PORTCbits.RC0

void init(void){
    /*vypnout analogove funkce na portu c*/
    ANSELC = 0x00;
    // RD2 jako vystup
    TRISDbits.TRISD2 = 0;
```

```
    // RC0 jako vstup
    TRISCbits.TRISC0 = 1;
}
/*deklarace funkce*/
void past(void);

void main(void){
    init();

    while(1){
        LED1 = ~LED1;
        __delay_ms(500);
        asm("CLRWDT");      // instrukce na vynulovani watchdog timeru
        if(BTN) past();     // pokud je tlacitko stisknuto spadni do pasti
    }
}
/* definice funkce past() ve ktere je nekonecna smycka */
void past(void){
    while(1){};
}
```