Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta – Katedra fyziky

Soubor úloh postavených na jednočipech PIC

Bakalářská práce

Vedoucí práce: Ing. Michal Šerý

Autor: Zdeněk Boháč

Anotace

Bakalářská práce je zaměřena na zpracování souboru úloh postavených na jednočipových mikrokontrolérech PIC. Práce nejprve krátce seznamuje s použitým hardwarem, jímž je mikroprocesor PIC16F84A, který je brán jako vhodný pro výuku základů programování mikrokontrolérů na střední škole. Ostatně celá práce si klade za cíl podpořit výuku v oblasti programování mikrokontrolérů na střední škole. Proto zde nechybí zpracované manuály pro programování v jazyce nízké úrovně – jazyce Assembleru, pro jazyk vyšší úrovně - jazyk C a manuály pro praci s použitými vývojovými prostředími (MPLab a MikroC). Výstupem práce je celá řada řešených úloh, které jsou řazeny od jednodušších po složitější a vzhledem k rozsahu jsou uvedeny v příloze. Úlohy jsou napsány jak v Assembleru, tak v jazyce C.

Abstract

This work contains collection of problems with the microcontrollers PIC. The work first shortly acquaint with used hardware, whereby is microprocessor PIC16F84A. This microcontroller is very good for education of programming for students on secondary school. This work have to help with study in programming microcontrollers on the secondary school. Next chapters contains manuals for programming in low-level language assembler and for language higher level - C language plus manuals for development systems MPLab and MikroC. The output of this work is many problems solution, that are ranged from simpler to complicated. Whole collection of problems is in apendix. Exercise are writing how in assembler, so in language C.

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě pedagogickou fakultou elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích 20. listopadu 2009 . . . .

Zdeněk Boháč

Poděkování

Děkuji vedoucímu práce Ing. Michalovi Šerému za cenné připomínky při realizaci práce.

Obsah

1 Úvod ... 7

2 Popis procesoru PIC16F84A ... 9

2.1 Základní vlastnosti ... 9

2.2 Popis periférií ... 9

2.3 Popis vývodů ... 9

2.4 Architektura ... 10

2.5 Uspořádání paměti ... 12

2.5.1 Paměť programu ... 12

2.5.2 Zásobník návratových adres ... 12

2.5.3 Speciální funkční registry a paměť dat ... 12

2.6 Porty ... 14

2.6.1 Port A ... 14

2.6.2 Port B ... 14

2.7 Čítač/časovač TMR0 ... 15

2.8 Přerušení (interrupt) ... 15

3 Assembler – jazyk symbolických adres ... 16

3.1 Základní charakteristika assembleru ... 16

3.2 Zpracování programu v Assembleru ... 17

3.3 Příklad vývojového prostředí ... 18

3.4 Přehled assembleru MPASM ... 18

3.4.1 Přehled instrukční sady ... 18

3.4.2 Direktivy assembleru ... 27

3.4.3 Používané číselné formáty ... 28

4 Základy jazyka C pro programování mikrokontrolerů ... 29

4.1 Jazyk C ... 29

4.2 Základy jazyka ANSI-C ... 29

4.2.1 Zápis programu v jazyce C ... 30

4.2.2 Příklad jednoduchého programu ... 30

4.2.3 První program pro PIC16F84A ... 31

4.3 Elementy jazyka C ... 32

4.3.1 Klíčová slova ... 32

4.3.2 Idetifikátory ... 33

4.3.3 Odsazovače (bílé znaky) ... 33

4.3.4 Konstanty ... 34

4.3.5 Proměnné 34
4.3.6 Komentáře 35
4.3.7 Pole 36
4.3.8 Příkaz přiřazení 37
4.3.9 Operátory 37
4.4 Řízení toku programu 40
4.4.1 Podmíněný operátor if 40
4.4.2 Přepínač - switch 40
4.4.3 Cykly 41
4.4.3.1 Cyklus while 42
4.4.3.2 Cyklus do-while 42
4.4.3.3 Cyklus for 43
4.5 Funkce 44
5. Použitá vývojová prostředí 46
5.1 MPLAB 46
5.1.1 MPLAB - Založení nového projektu v IDE MPLAB 46
5.1.2 Založení nového projektu bez použití průvodce 49
5.1.3 Překlad (kompilace) zdrojového souboru 51
5.1.4 Simulace 52
5.1.5 Zobrazení hodnot Speciálních funkčních registrů a proměnných 53
5.1.6 Rychlý přehled 55
5.2 Mikroelektronika MikroC 56
5.2.1 Mikroelektronika - Založení nového projektu v MikroC for PIC 56
5.2.2 Překlad (kompilace) zdrojového souboru 59
5.2.3 Simulace 59
5.2.4 Rychlý přehled 61
5.3 NI MULTISIM 62
6 Porovnání programovacích jazyků 64
7 Závěr 71
8 Seznam použité literatury 72
9 Přílohy 73

1 Úvod

Bakalářská práce je zaměřena na zpracování souboru úloh postavených na jednočipových mikrokontrolérech PIC. V poslední době zvláště v oblasti číslicové techniky dochází ke změně, kdy část číslicových obvodů není realizována za použití tradičních logických obvodů, ale využívá se při realizaci tzv. programovatelných součástek. Programovatelné součástky mají jednu základní výhodu – při realizaci obvodů nepotřebujeme mnoho různých typů logických obvodů, ale obvykle si vystačíme pouze s jediným integrovaným obvodem (v našem případě převážně PIC16F84A), který v závislosti na nahraném softwaru zastoupí i několik logických obvodů. Zapojení a vlastní realizace obvodů se stává jednodušší (méně použitých součástek) a výsledná zapojená z hlediska funkce jsou poměrně jednoduše modifikovatelná (pro změnu funkce obvykle stačí pouze změna softwaru).

Tato práce si klade za cíl podporu výuky v oblasi mikrokontrolérů na střední škole. Základním záměrem bylo zpracování souboru úloh, které by napomohli zvládnutí základů programování jednočipových mikrokontrolérů PIC. V krátké úvodní kapitole je krátce popsán použitý hardware, kterým je PIC16F84A, mikrokontrolér, který bývá často nazýván výukovým a proto se na středních školách také často používá. Výhodou PIC16F84A je ne moc rozsáhlý katalogový list (asi 40 stránek bez elektrických charakteristik), jehož obsah je nutné pro práci se součástkou zvládnout, a navíc je dostupný i v českém jazyce.

Třetí a čtvrtá kapitola seznamuje se základy programovacích jazyků, nejprve ve třetí kapitole jazyka nejnižší úrovně – Assembleru, ve čtvrté kapitole jazyka vyšší úrovně – jazyka C. Třetí kapitola je navíc doplněna o kompletní přehled instrukcí z tzv. redukované sady (RISC), kdy je každá instrukce vždy doplněna příkladem. Ve čtvrté kapitole se jedná o krátký přehled jazyka C (nikoli referenční) opět doplněný praktickými ukázkami.

Pátá kapitola se věnuje použitým vývojovým prostředím, jimiž jsou MPLab pro vývoj aplikací v Assembleru, MikroC pro jazyk C a je zde i zmínka o použití špičkového simulačního programu MULTISIM, který zvládne jak Assembler, tak jazyk C (je zde integrován překladač PICC lite firmy HiTech. Pro MPLab i MikroC jsou vždy na závěr zařazeny rychlé přehledy, které mají začátečníkovi usnadnit práci s vývojovým prostředím.

Šestá kapitola přináší srovnání jazyka Assembler s jazyky vyšší úrovně - jazykem C, Pascalem a Basicem. Na třech jednoduchých úlohách srovnáme výše zmíněné programovací jazyky. Vždy jsou uvedeny zdrojové kódy a velikost využité paměti.

Záměrem této bakalářské práce je zpracování souboru úloh postavených na jednočipech PIC. Kompletní přehled řešených úloh je uveden v příloze, následovaný

ukázkovou první úlohou. Veškeré zpracované úlohy jsou uvedeny na doprovodném CD disku v elektronické podobě. Jsou zpracovány postupně od nejjednodušších po složitější, rozděleny pro učitele a pro žáka, první část úloh je zpracována v Assembleru, druhá v jazyce vyšší úrovně - jazyce C. V zadání pro učitele je po hlavičce uvedeno zadání úlohy, schéma zapojení, následuje výukový účel úlohy s teoretickým rozborem a na závěr je uveden zdrojový kód aplikace, případně jsou důležité části zdrojového kódu okomentovány a vysvětleny. V zadání pro žáka je po nezbytné hlavičce uvedeno zadání úlohy doplněné schématem.

2 Popis procesoru PIC16F84A

2.1 Základní vlastnosti

Mikrokontrolér PIC16F84A používá redukovanou sadu 35 instrukcí (RISC), jejichž převážná většina se vykonává za jeden instrukční cyklus programu, vyjímku tvoří poze instrukce větvení programů a skoků. Je použita Harwardská architektura, pro níž je charakteristická oddělená paměť programu a dat. PIC16F84A má paměť programu o velikosti 1024 bajtů (umožňuje 1000 přepisů), paměť RAM o velikosti 68 bajtů a velikost paměti EEPROM pro uložení dat po odpojení napájení je 64 bajtů. Pro konfiguraci využívá 15 speciálních funkčních registrů (SFR).

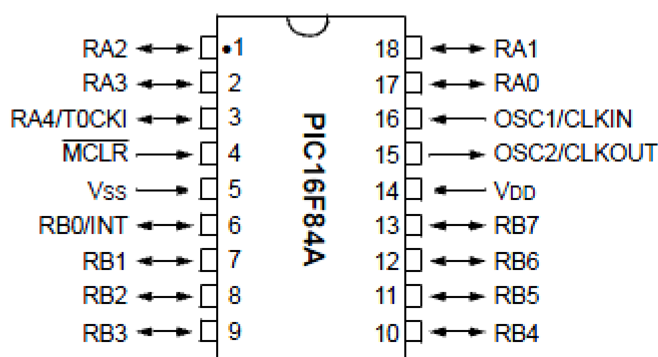
K další výbavě PIC16F84A se řadí přerušní za 4 zdrojů (dvou vnějších zdrojů: vstup RB0/INT a změna hodnoty na čtyřech vyšších bitech PORTB <7:4> a dva vnitřní zdroje: přetečení od časovače TMR0 a ukončení zápisu dat do paměti EEPROM), dále můžeme využít integrovaný 8-bitový čítač/časovač s 8-bitovou předděličkou.

Obvod je vyroben technologií CMOS (rozsah pracovního napětí: 2,0 V - 5,5 V, nízký odběr < 2 mA při 5 V při frekvenci oscilátoru 4 MHz, 15 µA při 2 V a frekvenci oscilátoru 32 kHz a méně než 1 µA při stand-by a napájení 2 V).

2.2 Popis periférií

Mikrokontrolér PIC16F84A obsahuje 13 nastavitelných vývodů, které mohou být nakonfigurovány jako vstupní (a poté z nich hodnota programově čtena) nebo jako výstupní (a hodnota do nich naopak programově zapisována). Z pohledu zatížení snese vývod nakonfigurovaný jako výstup pro ovládání LED max. proud 25 mA na výstup, celkově 80 mA pro 5 vývodů na Portu A a 100 mA 8 vývodů na Portu B.

2.3 Popis vývodů



Obr. č. 1: Pouzdro mikrokontroléru PIC16F84A [1]

vývod	pin	typ I/O/P	provedení	popis
				PORTA je obousměrný vstupně/výstupní port
RA0	17	I/O	TTL	
RA1	18	I/O	TTL	
RA2	1	I/O	TTL	
RA3	2	I/O	TTL	
RA4/TOCKI	3	I/O	ST	Na RA4 je připojen čítač / časovač. V zapojení jako výstup má otevřený kolektor!
MCLR/V_{pp}	4	I/P	ST	RESET/vstup programovacího napětí. RESET je proveden v logické nule.
V_{ss}	5	N	-	zem
				PORTB je obousměrný vstupně/výstupní port.
RB0/INT	6	I/O	TTL/ST	RB0 může být vybrán jako zdroj vnějšího přerušení
RB1	7	I/O	TTL	
RB2	8	I/O	TTL	
RB3	9	I/O	TTL	
RB4	10	I/O	TTL	přerušení při změně vstupu
RB5	11	I/O	TTL	přerušení při změně vstupu
RB6	12	I/O	TTL/ST	přerušení při změně vstupu/CLK při programování součástky
RB7	13	I/O	TTL/ST	přerušení při změně vstupu/DATA při programování součástky
V_{dd}	14	N	-	napájení +5V
OSC2/CLKOUT	15	O	-	Výstup krystalového oscilátoru. Připojení krystalu nebo keramického rezonátoru. V RC módu výstup CLK signálu, který je 1/4 kmitočtu na OSC1
OSC1/CLKIN	16	I	CMOS	vstup pro krystalový oscilátor příp. RC oscilátor

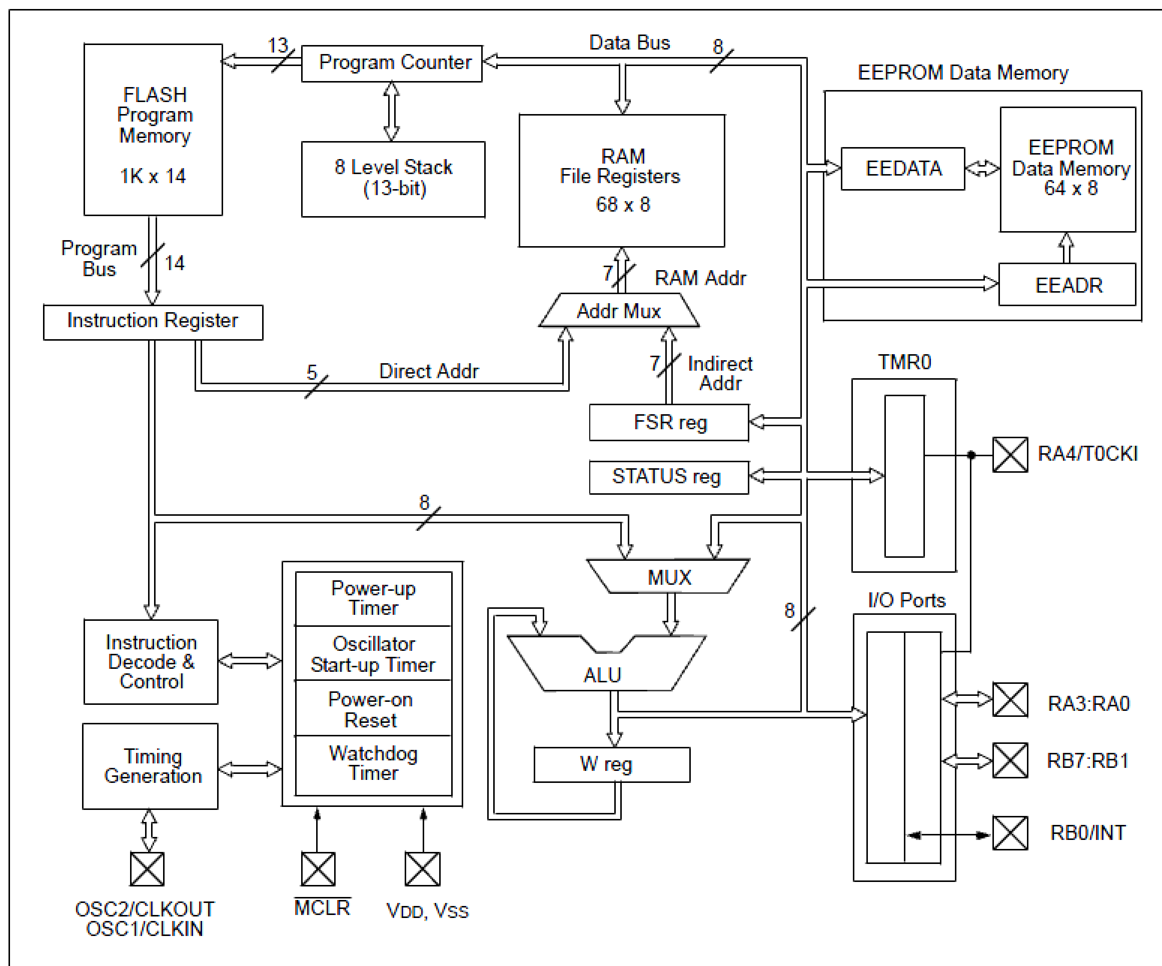
Obr. č. 2: Význam zkratek: I = Input (vstup) O = Output (výstup) I/O = Input/Output (vstup/výstup)
N = napájení TTL = TTL input ST = Schmitt Trigger input (na vstupu Schmittův obvod) [2]

2.4 Architektura

PIC16F84A využívá redukované sady instrukcí - RISC. Data a program jsou ukládány v oddělených částech paměti (tzv. Harwardská architektura). Data jsou šířky 8 bitů a kód programu má šířku 14 bitů. Tato šířka instrukčního slova (14 bitů) umožňuje vykonání většiny instrukcí během jediného instrukčního cyklu. To platí pro většinu instrukcí kromě instrukcí, které provádějí skoky či větvení programu (ty pro vykonání vyžadují 2 instrukční cykly).

ALU (aritmeticko-logická jednotka) umožňuje provádět základní aritmetické operace (sčítání, odčítání, rotace a základní logické operace). Aritmetické operace mají dva operandy, z nichž jeden je nutno vždy uložit do pracovního registru W a druhý operand je registr v paměti nebo konstanta. U jednoduchých instrukcí je operandem vlastní pracovní registr (W-registr), nebo registr v paměti. Pracovní registr W je 8-bitový a je

určen pro práci ALU a zároveň je využíván značnou částí instrukcí pro odkládání dat. Pracovní registr W není možné programově adresovat (nemá adresu).



Obr. č. 3: Blokové schéma PIC16F84A [1]

V závislosti na vykonávání instrukcí ALU jsou ovlivňovány hodnoty příznakových bitů v registru STATUS [3]:

C (CARRY – přenos hodnoty do vyššího/nižšího řádu přes 8 bitů)

DC (Digit Carry – přenos hodnoty do vyššího/nižšího řádu přes 4 bity)

Z (ZERO - příznak nulového výsledku po aritmetické operaci)

Pro přímé adresování se používá pátý bit registru STATUS nazvaný RP0. Pro nepřímé adresování se vyuřívá registru FSR (adresa 0x04 SFR), kam je zapsána adresa registru, do kterého chci zapsat a následně je vlastní hodnota zapsána do registru nepřímá adresa (0x00 v SFR).

2.5 Uspořádání paměti

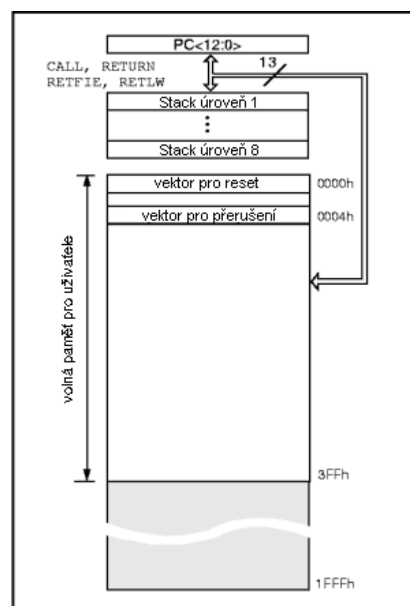
PIC16F84A používá Harwardské architektury, to znamená, že má navzájem oddělenou paměť programu (velikost 1024 B) a paměť dat (RAM 68 B a EEPROM 64 B pro trvalé uložení dat po odpojení napájení.

2.5.1 Paměť programu

Procesor PIC16F84A má 13-bitový programový čítač (PC), který je schopen adresovat programovou paměť o velikosti 8K x 14. Pro PIC16F84A se používá první 1K x 14 bitů (0000h-03FFh). Po resetu procesor začína na adrese 0000h. Vektor přerušení se nachází na adrese 0004h. Při volání podprogramů či vykonání je využíván 8-úrovňový zásobník (Stack) typu LIFO.

2.5.2 Zásobník návratových adres

Procesor má 8-mi úrovnový zásobník s šířkou 13 bitů. Zásobník není součástí programové ani datové paměti a není možné programově provádět jeho čtení nebo do něho zapisovat. Při instrukci volání podporgramu CALL je do zásobníku uložena celá 13 bitová hodnota programového čítače (Program Counter) zvětšená o jedničku. Obsah zásobníku využívají instrukce návratu z podprogramu RETURN a RETLW nebo instrukce návratu z přerušení RETFIE.



Obr. č. 4: Organizace paměti programu [2]

2.5.3 Speciální funkční registry a paměť dat

Paměť (obr. 5) je rozdělena do dvou stránek (tzv. banků), z nichž v každé je 128 adresovatelných buněk, ke kterým přistupujeme pomocí adresování v hexadecimální soustavě. Adresy 00h - 0Bh a 80h – 8Bh obsahují Speciální funkční registry a provádí se zde konfigurace procesoru, registry 0Ch - 4Fh jsou určeny pro data uživatele (paměť RAM).

adresa	registr stránka 0	registr stránka 1	adresa
00h	INDF (nepřímá adresa) *	INDF (nepřímá adresa) *	80h
01h	TMR0	OPTION	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h

06h	PORTB	TRSIB	86h
07h	--	--	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2 *	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch až 4Fh	68 bajtů paměti (SRAM)	namapováno do stránky 0	8Ch až CFh
50h až 7Eh	neimplementováno (čteno jako 0)	neimplementováno (čteno jako 0)	D0h až FFh

* Není fyzický registr

Obr. č. 5: Organizace speciálních funkčních registrů [2]

BANK 0									
Adresa	**registr**	**bit 7**	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
00h	INDF	(není fyzický registr)							
01h	TMR0	8-bitový časovač							
02h	PCL								
03h	STATUS	IRP	RP1	RP0	/TO	/PD	Z	DC	C
04h	FSR								
05h	PORTA	--	--	--	RA4/T0CKI	RA3	RA2	RA1	RA0
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
07h	--	není implementován							
08h	EEDATA	pracovní registr pro uložení dat paměti EEPROM							
09h	EEADR	pracovní registr pro uložení adresy paměti EEPROM							
0Ah	PCLATH	--	--	--					
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF

BANK 1									
Adresa	**registr**	**bit 7**	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
80h	INDF	(není fyzický registr)							
81h	OPTION	/RBPU	INTEDG	T0CS	TS0E	PSA	PS2	PS1	PS0
82h	PCL								
83h	STATUS	IRP	RP1	RP0	/TO	/PD	Z	DC	C
84h	FSR								
85h	TRISA	--	--	--	nastavení PORTu A				
86h	TRISB	nastavení PORTu B							
87h	--	není implementován							
88h	EECON1	--	--	--	EEIF	WRERR	WREN	WR	RD
89h	EECON2	(není fyzický registr)							
8Ah	PCLATH	--	--	--					
8Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF

Obr. č. 6: Přehled speciálních funkčních registrů (SFR) [2]

2.6 Porty

Mikrokontrolér PIC16F84 má 2 porty, port A a port B. Celkem nabízí 13 vstupně-výstupních (I/O) vývodů. Funkce vývodů (tedy vstup či výstup) u těchto portů mohou být měněny v závislosti na požadavku programu, a to i během samotného vykonávání programu.

2.6.1 Port A

Port A je 5-bitový (má vývody RA0 až RA4, zbylé vývody PORTA nejsou implementovány). Vývod RA4 má na vstupu Schmittův obvod a jako výstup má otevřený kolektor. Všechny ostatní vývody portu A mají jako vstupní úrovně TTL a jako výstup budiče CMOS. Všechny I/O vývody portu A se nastavují v registru TRISA, registru na adrese 85h v banku 1 SFR, kde můžeme nastavit každý z vývodů jako vstup nebo výstup nezávisle na ostatních. Nastavením příslušného bitu na "1" (I – Input) v registru TRISA se nastaví příslušný vývod jako vstupní, nastavením příslušného bitu na "0" (O – Output) se nastaví příslušný vývod jako výstupní. Po inicializaci procesoru jsou všechny I/O piny nastaveny jako vstupy.

Vývod portu RA4 je možné přepnout jako vstup pro hodinový signál TMR0.

2.6.2 Port B

Port B je 8-bitový (RB0 až RB7). Všechny I/O výstupy se nastavují v TRISB, registru na adrese 86h v banku 1 Speciálních funkčních registrů, kde můžeme nastavit každý z vývodů jako vstup nebo výstup nezávisle na ostatních. Nastavením příslušného bitu na "1" v registru TRISB se nastaví příslušný vývod jako vstupní, nastavením příslušného bitu na "0" se nastaví příslušný vývod jako výstupní. Po inicializaci procesoru jsou všechny I/O piny nastaveny jako vstupy.

Každý z vývodů portu B může mít programově připojen slabý vnitřní pull-up rezistor (cca 100 uA) na všech vývodech konfigurovaných jako vstupní. Toto je automaticky vypnuto u těch vývodů, které jsou nastaveny jako výstupní.

Čtyři vývody portu B (RB4 - RB7) mají při příslušném nastavení schopnost vyvolat přerušení při změně stavu. Avšak tato schopnost je dána pouze vývodům nastaveným jako vstupní. Přerušení může být vyvoláno i změnou hodnoty vstupního bitu RB0 (aktivní hrana se nastavuje v registru OPTION na adrese 81h SFR).

2.7 Čítač/časovač TMR0

Modul TMR0 je 8-bitový registr čítače/časovače, kde je povolen zápis i čtení obsahu registru, navíc obsahuje 8-bitovou programovatelnou předdělička a výběr aktivní hrany při externím vstupním hodinovém signálu.

V režimu časovač (TIMER) je obsah registru TMR0 inkrementován při každém instrukčním cyklu (f_{osc}/4). Toto platí, nebude-li použita předdělička. Je-li do registru TMR0 zapsána hodnota, je zvýšení jeho obsahu povoleno až po dvou následujících instrukčních cyklech.

V režimu čítač (Counter) bude obsah registru TMR0 zvyšován s každou vzestupnou případně sestupnou hranou na vývodu RA4/T0CKI. Přeteče-li obsah registru čítače/časovače z hodnoty FFh -> 00h, dochází k vyvolání rutiny přerušení čítače TMR0, kdy je nastaven příznakový bit TOIF.

Předděličku je volitelně možné přiřadit buď k funkčnímu modulu TMR0 nebo WDT. Obsah předděličky není možné číst ani její hodnotu měnit. Je-li předdělička předřazena čítači / časovači TMR0 je možné programově zvolit tyto dělící poměry 1:2, 1:4, ..., 1:256. Je-li předdělička předřazena časovači Watchdog (WDT) je možné programově zvolit dělící poměry 1:1, 1:2, ..., 1:128.

2.8 Přerušení (interrupt)

PIC16F84A má 4 zdroje (dva vnější: vstup RB0/INT a čtyři vyšší bity PORTu B <7:4> a dva vnitřní: přetečení od časovače TMR0 a ukončení zápisu dat do paměti EEPROM), které mohou vyvolat přerušení na základě vnější události.

Při přijetí přerušení (zde se předpokládá, že je povoleno) mikrokontrolér skočí na adresu 0x04 programové paměti a po vykonání zde začínajícího obslužného podprogramu se program vrátí zpět do místa, kde k přerušení došlo [4].

3 Assembler – jazyk symbolických adres

Assembler je programovací jazyk nejnižší úrovně (generace programovacích jazyků: 1. generace – strojový kód, 2. generace – Assembler, 3. generace – strukturované programovací jazyky Pascal, C, atd.). Historie Assembleru sahá až do poloviny 20. století. Poté byl Assembler v 70. letech 20. století postupně vytlačen jazyky vyšší úrovně (jazyky C, Pascal, atd.), které byli odvozeny právě z Assembleru.

Dnes se s Assemblerem již nesetkáme v takové míře jako v dřívějších dobách. Původně, ještě předtím, než vznikly vyšší programovací jazyky (např. Pascal, C, Basic), se v Assembleru psalo prakticky vše. V době prvních PC se již používaly programovací jazyky vyšší úrovně jako Pascal, C a později C++. Assembler byl používán jen na některé části programu, a to právě na ty, které musely být velmi rychlé, např. pro práci s grafikou.

V dnešní době se Assembler používá pouze v několika málo oblastech. Za zmínku stojí psaní ovladačů zařízení, virů, či oblast méně legální, kterou je cracking (disassemblování programu a následnou úpravu zdrojového kódu). Poslední oblastí je použití Assembleru pro psaní programů pro menší zařízení, která nedisponují takovým výkonem (a kapacitou paměti) jako dnešní počítače. Těmito zařízeními jsou např. mikrokontroléry, ale i zde se začínají čím dál více využívat vyšší programovací jazyky (jazyk C, Basic či Pascal).

3.1 Základní charakteristika assembleru

Jazyk Assembler je jazyk symbolických adres, kde samotné adresy nahrazujeme symboly a tím se stává vlastní kód čitelnější. Řádek v Asesembleru má pevně danou strukturu, dělí se na čtyři oddíly vzájemně oddělené bílými znaky (kvůli zarovnání se doporučuje jako bílý znak tabulátor). Na jednom řádku může být maximálně jeden příkaz, je tedy možné využít všechny čtyři oddíly řádku, či vynechat kterýkoliv z čtyř oddílů, ale řádek může zůstat i prázdný.

Syntaxe řádku v assembleru:

```
Label příkaz P1,P2 ; komentář (co je za středníkem překladač ignoruje)
```

Label – návěští se používá pro přenesení řízení programu na jiné místo, např. po instrukci `goto k`,

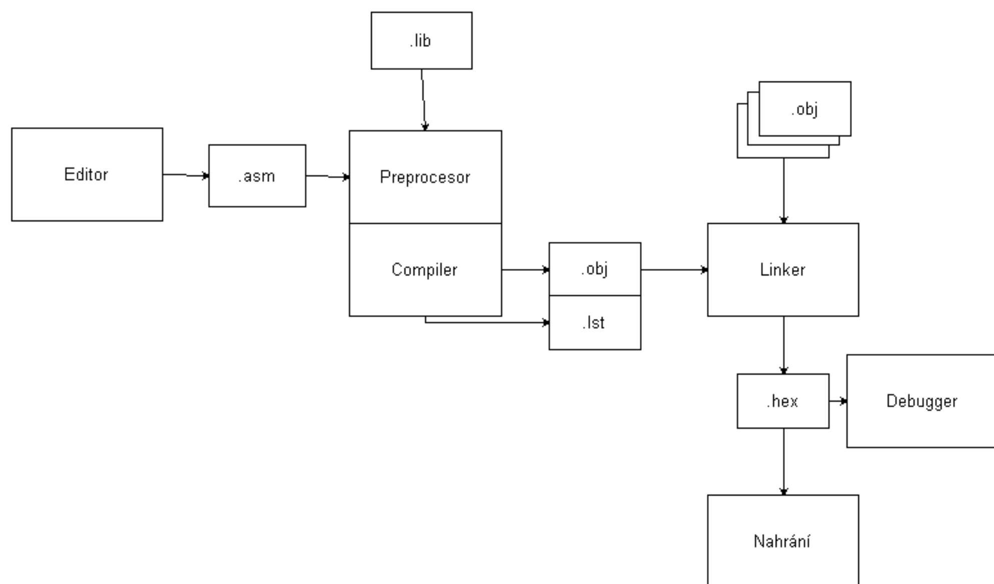
P1,P2 – operandy, parametry,

Komentář – slouží pro vytvoření poznámek pro autora programu, používá se zejména pro zvýšení čitelnosti programu.

V Assembleru na místě **příkazu** může být:

- **instrukce** – PIC16F84 má 35 instrukcí,
- **pseudoinstrukce** – ty se moc nepoužívají, jedná se o nadstavbu jazyka Assembler, kdy výsledná instukce je složeninou více instrukcí
- **makroinstrukce –** makra – několik instrukcí zařazených v bloku programu.
- **direktivy** – jsou rezervovaná slova, která nemůžou být využita pro jinou činnost, než pro jakou jsou nadefinována, např. `END` – direktiva, která ukončuje program.

3.2 Zpracování programu v Assembleru



Obr. č. 7: Zpracování programu v assembleru

Zdrojový kód píšeme v *editoru* (může to být i poznámkový blok Windows), vlastní zdrojový soubor má příponu .asm. *Preprocesor* je součástí překladače, která předzpracovává zdrojový soubor tak, aby měl překladač lehčí práci. Např. vynechává komentáře, zajišťuje správné vložení maker apod. *Compiler*, česky kompilátor (překladač), vykonává překlad zdrojového souboru do relativního (objektového) kódu počítače – při překladu vzniká soubor s příponou .obj. Vzniká tzv. protokol o překladu (listing - *.lst* soubor). *Linker* sestaví program, jehož výsledkem je přímo program v hexadecimální soustavě (příponoa *.hex*). *Debugger* slouží k ladění programu a hledání syntaktických chyb, které nastanou při běhu programu. Po nalezení chyby se celý cyklus (editor, compiler, linker, debugger) opakuje až do doby, kdy náš program už žádnou syntaktickou chybu neobsahuje [5].

3.3 Příklad vývojového prostředí:

Vývojové prostředí **MPLAB** verze 8.30 (90Mb) pro Windows 98 SE, Windows 2000 SP2, Windows NT SP6, Windows ME, Windows XP Pro, Windows Vista, Windows 7. Program MPLAB obsahuje překladač **MPASM verze 5.21.** Je možné stáhnout přímo od výrobce mikrokontrolérů firmy Microchip z www adresy http://www.microchip.com.

Poznámka: Pro psaní programů v Assembleru můžeme použít i poznámkový blok Windows a pro vlastní překlad provést v samostatném překladači MPASM.

3.4 Přehled assembleru MPASM

3.4.1 Přehled instrukční sady

PIC16F84A obsahuje redukovanou sadu instrukcí (RISC) na 36 instrukcí. Osmnáct instrukcí je bajtově orientovaných (operandy f,d nebo f), čtyři instrukce jsou bitově orientované (operandy f,b), třináct instrukcí je řídících, navíc je tu jedna instrukce, která nic nedělá (NOP).

Seznam použitých zkratek:

k - konstanta;

f - adresa registru, se kterým instrukce pracuje;

d - určuje, kam je uložen výsledek operace. Je li d = 0,výsledek je uložen do pracovního registru W, je-li d = 1, výsledek je uložen do registru f. Výchozí nastavení tohoto parametru je 1;

b – pořadí bitu v registru *f*.

3.4.1.1 Instrukce aritmetických a logických operací

ADDLW k (ADD Literal to W)

Sečte obsah pracovního registru *W* s konstantou *k*, výsledek se uloží do *W*.

Ovlivňuje stavové bity: C, DC, Z   Počet cyklů: 1

Příklad:

```
; priklad secteni cisel 136 a 121 (136 + 121 = 257)
; priznakove bity STATUS Z = 0, DC = 0, C = 0
       movlw D'136' ; v pracovnim registru w je ulozeno cislo 136
       addlw D'121'
; obsah registru W je 1
; priznakove bity STATUS Z = 0, DC = 1, C = 1
; byl prenos přes 256 i přes 16, ale vysledek je nenulovy
```

ADDWF f,d (ADD W to F)

Sečte obsah registru *W* s registrem *f*. Je-li d = 0, výsledek se uloží do W, je-li d = 1, výsledek se uloží do *f*.

Ovlivňuje stavové bity: C, DC, Z   Počet cyklů: 1

Příklad:

```
       cislo_1    equ  0x0c
; priklad secteni cisel 136 a 121 (136 + 121 = 257)
; priznakove bity STATUS Z = 0, DC = 0, C = 0
       movlw     D'121'
       movwf     cislo_1 ; do registru cislo_1 nahrajeme hodnotu 121
       movlw     D'136' ; v pracovnim registru w je ulozeno cislo 136
       addwf     cislo_1,0
; obsah registru W je 1
; priznakove bity STATUS Z = 0, DC = 1, C = 1
; byl prenos přes 256 i přes 16, ale vysledek je nenulovy
```

ANDLW k (AND Literal and W)

Provede AND (logický součin) mezi pracovním registrem *W* a konstantou *k*. Výsledek uloží popracovního registru *W*.

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Pravdivostní tabulka logické funkce AND:

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Příklad:

```
; priklad logického soucinu binárních cisel  00001111 & 01010101 = 00000101
; priznakove bity STATUS Z = 0
       movlw   b'00001111' ; do pracovniho registru W nahrajeme hodnotu 00001111
       andlw   b'01010101' ; provedu log. soucin cisla ve W s cislem 01010101
; vysledek ulozi do W, obsah registru W je 00000101
; priznakove bity STATUS Z = 0 , vysledek je nenulovy
```

ANDWF f,d (AND W with F)

Provede AND (logický součin) mezi pracovním registrem *W* a registrem f. Je-li d = 0, výsledek se uloží do *W*, je-li d = 1, výsledek se uloží do *f*.

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Příklad:

```
; priklad logického soucinu binárních cisel  00001111 & 01010101 = 00000101
; priznakove bity STATUS Z = 0
              movlw   b'01010101'
              movwf   cislo_1 ; do promenne cislo_1 nahrajeme hodnotu 00001111
              movlw   b'00001111'
              andwf   cislo_1,0 ; provedu log. soucin cisla ve W s cislem 01010101
; vysledek ulozi do W, obsah registru W je 00000101
; priznakove bity STATUS Z = 0 , vysledek je nenulovy
```

COMF f,d (COMplement F)

Provede negaci (zamění hodnotu bitů 0 za 1 a naopak) registru f. Je-li d = 0, výsledek se uloží do W, při d = 1se výsledek uloží do f.

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Příklad:
```
; priklad negace binárního cisla  01010101
; priznakove bity STATUS Z = 0
              movlw  b'01010101'
              movwf  cislo_1 ; do promenne cislo_1 nahrajeme hodnotu 01010101
              comf   cislo_1,0 ; provedu negaci binárního cisla 01010101
; vysledek ulozi do W, obsah registru W je 10101010
; priznakove bity STATUS Z = 0 , vysledek je nenulovy
```

DECF f,d (DECrement F)

Odečte jedničku od obsahu registru f. Je-li d = 0, výsledek se uloží do W, při d = 1 se výsledek uloží do f.

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Příklad:
```
; priklad dekrementace cisla  10
; priznakove bity STATUS Z = 0
       movlw  d'10'
       movwf  cislo_1 ; registr cislo_1 ma hodnotu 10
dokola decf   cislo_1,1 ; provede cislo_1 - 1 a vysledek ulozi do cislo_1
       movfw  cislo_1 ; zkopiruje obsah cislo_1 do W
       goto   dokola ; provadi neustale odecitani cisla 1
; priznakove bity STATUS Z = 0 , vysledek je nenulovy
```

INCF f,d (DECrement F)

Přičte jedničku k obsahu registru f. Je-li d = 0, výsledek se uloží do W, při d = 1 se výsledek uloží do f.

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Příklad:
```
; priklad inkrementace cisla  10
; priznakove bity STATUS Z = 0
       movlw  d'10'
       movwf  cislo_1 ; registr cislo_1 ma hodnotu 10
dokola incf   cislo_1,1 ; provede cislo_1 + 1 a vysledek ulozi do cislo_1
       movfw  cislo_1 ; zkopiruje obsah cislo_1 do W
       goto   dokola ; provadi neustale pricitani cisla 1
; priznakove bity STATUS Z = 0 , vysledek je nenulovy
```

IORLW k (Inclusive OR Literal with W)

Provede OR (logický součet) mezi registrem W a konstantou k. Výsledek uloží do W.

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Příklad:
```
; priklad logického souctu binárních cisel  00001111 or 01010101 = 01011111
; priznakove bity STATUS Z = 0
       movlw  b'00001111' ; do pracovniho registru W nahrajeme hodnotu 00001111
       iorlw  b'01010101' ; provedu log. soucet cisla ve W s cislem 01010101
; vysledek ulozi do W, obsah registru W je 01011111
; priznakove bity STATUS Z = 0 , vysledek je nenulovy
```

IORWF f,d Inclusive OR W with F)

Provede OR (logický součet) mezi registrem W a registrem f. Je-li d = 0, výsledek se uloží do W, při d = 1 se výsledek uloží do f.

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Pravdivostní tabulka logické funkce OR:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Příklad:
```
; priklad logického souctu binárních cisel  00001111 or 01010101 = 01011111
; priznakove bity STATUS Z = 0
              movlw  b'01010101'
              movwf  cislo_1 ; do promenne cislo_1 nahrajeme hodnotu 00001111
              movlw  b'00001111'
              iorwf  cislo_1,0 ; provedu log. soucet cisla ve W s cislem 01010101
; vysledek ulozi do W, obsah registru W je 01011111
; priznakove bity STATUS Z = 0 , vysledek je nenulovy
```

SUBLW k (SUBtract W from Literal)

Odečte obsah registru pracovního registru W od konstanty k. Výsledek je uložen do registru pracovního registru W.

Ovlivňuje stavové bity: C, DC, Z   Počet cyklů: 1

Příklad:
```
; priklad odecteni cisel 100 a 10 (100 - 10 = 90)
; priznakove bity STATUS Z = 0, DC = 0, C = 0
       movlw  d'10' ; obsah registru W je 10
       sublw  d'100' ; 100 – 10 = 90
; obsah registru W je 00001010 – 10
; priznakove bity STATUS Z = 0, DC = 0, C = 0
; nebyl prenos přes 256 ani přes 16 a vysledek je nenulovy
```

SUBWF f,d (SUBtract W from F)

Odečte obsah pracovního registru W od registru f. Je-li d = 0, výsledek se uloží do W, při d = 1 se výsledek uloží do f.

Ovlivňuje stavové bity: C, DC, Z   Počet cyklů: 1

Příklad:
```
; priklad odecteni cisel 100 a 10 (100 - 10 = 90)
; priznakove bity STATUS Z = 0, DC = 0, C = 0
       movlw  d'10'
       movwf  cislo_1 ; do promenne cislo_1 nahrajeme hodnotu 10
       movlw  d'100'
       subwf  cislo_1,0 ; 100 – 10 = 90
; obsah registru W je 00001010 – 10
; priznakove bity STATUS Z = 0, DC = 0, C = 0
; nebyl prenos přes 256 ani přes 16 a vysledek je nenulovy
```

XORLW k (eXclusive OR Literal with W)

Provede XOR mezi registrem W a konstantou k. Výsledek uloží do W.

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Pravdivostní tabulka logické funkce XOR:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Příklad:
```
; priklad XOR binárních cisel  00001111 XOR 01010101 = 01011010
; priznakove bity STATUS Z = 0
       movlw  b'00001111' ; do pracovniho registru W nahrajeme hodnotu 00001111
       xorlw  b'01010101' ; provede XOR cisla ve W s cislem 01010101
; vysledek ulozi do W, obsah registru W je 01011010
; priznakove bity STATUS Z = 0 , vysledek je nenulovy
```

XORWF f,d (eXclusive OR W with F)

Provede XOR mezi registrem W a registrem f. Je-li d = 0, výsledek se uloží do W, při d = 1 se výsledek uloží do f.

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Příklad:

```
; priklad XOR binárních cisel  00001111 XOR 01010101 = 01011010
; priznakove bity STATUS Z = 0
       movlw   b'00001111'
       movwf   cislo_1 ; do registru cislo_1 nahrajeme hodnotu 00001111
       movlw   b'01010101'
       xorwf   cislo_1,0 ; provedu XOR cisla_1 s cislem 01010101
; vysledek ulozi do W, obsah registru W je 01011010
; priznakove bity STATUS Z = 0 , vysledek je nenulovy
```

3.4.1.2 Instrukce nulování a nastavení

BCF f,b (Bit Clear F)

Vynuluje bit b registru f.

Ovlivňuje stavové bity: -   Počet cyklů: 1

Příklad:

```
       bcf     0x03,5 ; bank 0
```

BSF f,b (Bit Set F)

Nastaví bit b registru f do 1.

Ovlivňuje stavové bity: -   Počet cyklů: 1

Příklad:

```
       bsf     0x03,5 ; bank 1
```

CLRF f (CLeaR F)

Vynuluje obsah registru f.

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Příklad:

```
       clrf    PORTB ; vynulovani registru PORTB
```

CLRW (CLeaR W register)

Vynuluje obsah registru W.

Ovlivňuje stavové bity: Z Počet cyklů: 1

Příklad:

```
       clrw ; vynulovani pracovniho registru W
```

CLRWDT (CleaR WatchDog Time)

Vynuluje WDT a předděličku, když je k WDT připojená.

Ovlivňuje stavové bity: TO = 1, PD = 1   Počet cyklů: 1

Příklad:

```
clrwdt ; vynulovani WDT
```

3.4.1.3 Instrukce přesunu dat

MOVF f,d (MOVe F)

Přesune obsah registru f je-li d = 0, do registru W, je-li d = 1 zpět do registru F

Ovlivňuje stavové bity: Z   Počet cyklů: 1

Příklad:

```
movf cislo_1,w ; presun cislo_1 do pracovniho registru W
```

MOVLW k (MOVe Literal to W)

Přesune konstantu k do registru W

Ovlivňuje stavové bity: -   Počet cyklů: 1

Příklad:

```
movlw b'00001111' ; presun binárního cisla 00001111 do pracovniho registru W
```

MOVWF f (Move W to F)

Přesune obsah registru W do registru f.

Ovlivňuje stavové bity: -   Počet cyklů: 1

Příklad:

```
movwf cislo_1 ; presun hodnoty z pracovniho registru W do registru cislo_1
```

RLF f,d (Rotate Left F through carry)

Rotuje obsah registru f o jeden bit doleva přes C bit stavového registru. Je-li d = 0, výsledek se uloží do W, je-li d = 1, výsledek se uloží do f.

Ovlivňuje stavové bity: C   Počet cyklů: 1

Příklad:

```
; priklad provedeni rotace doleva na binárním cisle 00001111
; priznakove bity STATUS C = 0
        movlw   b'00001111'
        movwf   cislo_1 ; do registru cislo_1 nahrajeme hodnotu 00001111
        rlf     cislo_1,1  ; provedeni rotace vlevo
; vysledek ulozi do registru cislo_1, obsah registru cislo_1 je 00011110
; priznakove bity STATUS C = 0, nebyl prenos
```

RRF f,d (Rotate Right F through carry)

Rotuje obsah registru f o jeden bit doprava přes C bit stavového registru. Je-li d = 0, výsledek se uloží do W, je-li d = 1, výsledek se uloží do f.

Ovlivňuje stavové bity: C   Počet cyklů: 1

Příklad:

```
; priklad provedeni rotace doprava na binárním cisle 00001111
; priznakove bity STATUS C = 0
       movlw   b'00001111'
       movwf   cislo_1 ; do registru cislo_1 nahrajeme hodnotu 00001111
       rrf     cislo_1,1  ; provedeni rotace vpravo
; vysledek ulozi do registru cislo_1, obsah registru cislo_1 je 00000111
; priznakove bity STATUS C = 1, probehl prenos
```

SWAPF f,d (SWAP F)

Prohodí horní a dolní půlbyte registru f. Je-li d = 0, výsledek se uloží do W, je-li d = 1, výsledek se uloží do f.

Ovlivňuje stavové bity: -   Počet cyklů: 1

Příklad:

```
; priklad prohozeni hornich a dolnich čtyř bitu
       movlw   b'00001111'
       movwf   cislo_1 ; do registru cislo_1 nahrajeme hodnotu 00001111
       swapf   cislo_1,1  ; prohozeni hornich a dolnich čtyř bitu
```

3.4.1.4 Instrukce podprogramů a přerušení

CALL (subroutine CALL)

Volání podprogramu.

Ovlivňuje stavové bity: -   Počet cyklů: 2

Příklad:

```
        call cekej ; volani podprogramu cekej
```

RETLW k (RETurn Literal to W)

Navrátí se z podprogramu. Registr W naplní konstantou k.

Ovlivňuje stavové bity: -   Počet cyklů: 2

Příklad:

```
        retlw 0 ; navrat z podprogramu a zapis hodnoty 0 do pracovniho registru W
```

RETURN (RETurn from subroutine)

Navrátí se z podprogramu.

Ovlivňuje stavové bity: -   Počet cyklů: 2

Příklad:

```
return ; navrat z podprogramu
```

RETFIE (RETurn From Interrupt)

Navrátí se z podprogramu obsluhujícího přerušení.

Ovlivňuje stavové bity: GIE = 1   Počet cyklů: 2

Příklad:

```
retfie ; navrat z preruseni
```

3.4.1.5 Instrukce skoků

BTFSC f,b (Bit Test F, Skip if Clear)

Je-li bit b registru f (tedy f,b = 0), přeskočí následující instrukci (provede místo ní instrukci NOP)

Ovlivňuje stavové bity: -   Počet cyklů: 1(2)

Příklad:

```
; priklad pouziti podminky btfsc
testuj btfsc  tlacitko ; je stisknute tlačítko?
       goto   svetlo ; není - rozsvit LED
       bcf    led ; zhasni LED
       goto   testuj ; opet testuj
svetlo bsf    led    rozsvit LED
       goto   testuj  ; opet testuj
```

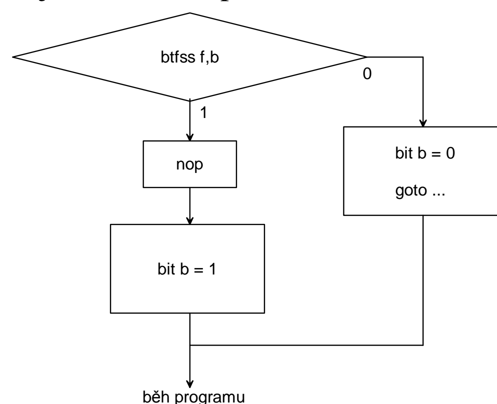
BTFSS f,b (Bit Test F, Skip if Set)

Je-li bit b registru f (tedy f,b = 1), přeskočí následující instrukci (provede místo ní instrukci NOP).

Ovlivňuje stavové bity: -   Počet cyklů: 1(2)

Příklad:

```
; priklad pouziti podminky btfsc
testuj btfsc  tlacitko ; je stisknute tlačítko?
       goto   svetlo ; ano - rozsvit LED
       bcf    led ; zhasni LED
       goto   testuj ; opet testuj
svetlo bsf    led    rozsvit LED
       goto   testuj  ; opet testuj
```



Obr. č. 8: Blokový diagram instrukce BTFSS

GOTO k (GO TO)

Provede nepodmíněný skok na adresu k.

Ovlivňuje stavové bity: - Počet cyklů: 1(2)

Příklad:

```
goto   skok ; prenese rizeni programu na navesti skok
```

DECFSZ f,d (DECrement F, Skip if Zero)

Od obsahu registru f odečte jedničku. Je-li d = 0, výsledek se uloží do W, je-li d = 1, výsledek se uloží do f. Je-li výsledek po odečtení 0, přeskočí se následující instrukce (provede se místo ní instrukce NOP).

Ovlivňuje stavové bity: - Počet cyklů: 1(2)

Příklad:

```
; priklad pouziti instrukce decfsz f,d – nejcastejsi pouziti: casove smycky
Cekej  movlw  d'10'
       movwf  pocet
opakuj nop ; nedelej nic, cekej 1 instrukcni cyklus
       nop
       nop
       decfsz pocet,1 ; počet = pocet - 1
       goto   opakuj  ; opakuj dokud počet > 0, pro počet = 0 nahrad instrukci nop
       return
```

INCFSZ f,d (INCrement F, Skip if Zero)

K obsahu registru f přičte jedničku. Je-li d = 0, výsledek se uloží do W, je-li d = 1, výsledek se uloží do f. Je-li výsledek po přičtení 0, přeskočí se následující instrukce (provede se místo ní instrukce NOP).

Ovlivňuje stavové bity: - Počet cyklů: 1(2)

Příklad:

```
       incfsz pocet,1 ; pocet = počet + 1
```

3.4.1.6 Zvláštní instrukce

NOP

No OPeration

Prázdná operace. Nic se neprovede.

Ovlivňuje stavové bity: - Počet cyklů: 1

Příklad:

```
nop ; instrukce nic nedela, pouze trva 1 instrukcni cyklus
```

SLEEP

Mikrokontrolér přejde do stavu SLEEP. Vynuluje WDT a předděličku.

Ovlivňuje stavové bity: PD = 0, TO = 1 Počet cyklů: 1

Příklad:

```
sleep ; uvedeni mukrokontroleru do rezimu spanku
```

[2 - instrukce převzaty z katalogového listu]

3.4.2 Direktivy assembleru

INCLUDE

Syntaxe: INCLUDE „soubor"

Vloží soubor s defincemi, podprogramy, knihovnami

Příklad:

```
include „C:\MPLAB\PICPIC16F84A.EQU
```

EQU

Syntaxe: název konstanty EQU hodnota

Definice konstanty v programu

Příklad:

```
status equ 0x03 ; status je na adrese 03h
```

ORG

Syntaxe: (návěští) ORG hodnota

Nastaví adresu na kterou se uloží následující program

Příklad:

```
org 0x0004
```

#define

Syntaxe: #define název registr,bit

definuje název pro bit registru

Příklad: `#define LED porta, 0`

MACRO

ENDM

Syntaxe: název MACRO

vytvoří makroinstrukci z instrukcí uzavřených mezi

MACRO a ENDM.

Příklad:

```
bank0 macro
            bcf RP0
     endm
```

LIST P

Syntaxe: LIST P= typ procesoru

říká překladači, pro jaký procesor je program napsán

Příklad:

```
LIST P = PIC16F84A
```

END

Konec programu

;

Oddělí komentář od vlastního programu od středníku do konce řádku.

3.4.3 Používané číselné formáty

Dekadické: D’100’

Dekadický (desítkový) formát je dobré používat kvůli lepší čitelnosti, pro zadávání hodnot do čekacích smyček.

Hexadecimální:0FEH, 0xFE

Hexadecimální tvar můžeme použít tam, kde zadáváme hodnotu celého bytu, popřípadě v adresách..

Binární: B’10001011’

Binární tvar je vhodné použít při nastavování bitů ve stavovém slově, registrech speciálních funkcí atd.

4 Základy jazyka C pro programování mikrokontrolerů

Tento manuál obsahuje pouze základy jazyka C potřebné s ohledem na programování jednočipových mikropočítačů PIC, proto celá řada standardních kapitol byla z tohoto manuálu vypuštěna. Nejedná se tedy o referenční manuál, který by pokrýval celou problematiku programování v jazyce C.

4.1 Jazyk C

První verzi jazyka C původně navrhl a implementoval Denis Ritchie (1972) pod operačním systémem UNIX na počítači PDP-11. Podoba jazyka byla časem vylepšována a normována, v současné době se používá na téměř všech komerčně dostupných operačních systémech. V současné době se používá ANSI-C (ANSI – American National Standard for Information System – Programing Language C) verze z roku 1999.

Jazyk ANSI-C je pro požadavky v oboru mikrokontrolérů velmi vhodný. Všeobecně je jazyk C v současné době nejčastěji používaný programovací jazyk, a to nejen při programování mikrokontrolérů, ale i pro tvorbu win aplikací [6].

Přechod z assembleru na jazyk C se zdá v prvopočátku velmi obtížný. Většinou jsou zde obavy týkající se větší velikosti kódu a také rychlosti zpracování. Velikost výsledného kódu napsaného v jazyce C je sice o něco větší, než z assembleru, ale v současné době již mikrokontroléry disponují obvykle dostatkem paměti, takže pojmou i náročné aplikace vytvořené v jazyce C [6]. Obvykle platí, že začne-li se programátor nejprve učit psát programy v assembleru a pak následně přejde na C, nechce se již k assembleru nikdy vrátit. Naopak při začátku psaní programů v jazyce C a následném přechodu k assembleru zjistíme, že zde většinou již panuje neochota se učit assembler z důvodu „zhýčkanosti“ z jazyka C (přece jen u programování v assembleru je potřeba detailnější znalost hardwaru mikrokontroléru a zvláště jeho instrukční sady) .

Naopak program v jazyce C je jednodušší a velmi dobře přenositelný pro různé typy mikrokontrolérů, ať již mezi mikroprocesory od stejného výrobce nebo mikroprocesory ostatních výrobců.

4.2 Základy jazyka ANSI-C

Každý program v jazyce C je složen z určitých základních součástí. Program v jazyce C se skládá z jedné nebo více funkcí (program v jazyce C je sám o sobě funkcí). z

nichž každá obsahuje jeden nebo více příkazů. Funkce je v jazyce C pojmenovaný podprogram, který lze volat z jiných částí programu. Funkce jsou tedy základními stavebními kameny jazyka C.

Všechny příkazy jazyka C končí středníkem. Jazyk C nepovažuje konec řádku za ukončení příkazu. Na jednom řádku tedy můžeme umístit i dva nebo více příkazů nebo jeden příkaz může pokračovat přes více řádků.

4.2.1 Zápis programu v jazyce C

```
typ jméno_funkce (parametry){
příkazy
}
```

Položka **typ** určuje typ dat vrácených funkcí.

Položka **jméno_funkce** určuje jméno funkce.

Parametry lze funkci předávat pomocí seznamu parametrů (v závorce).

Na místě položky příkazy může být jeden nebo více příkazů. Teoreticky by funkce nemusela obsahovat žádný příkaz, ale protože by v takovém případě neprováděla žádnou činnost, nemělo by to v praxi žádný význam.

4.2.2 Příklad jednoduchého programu

```
#include <stdio.h>
int main (void){
printf ("AHOJ") ;
return 0;
}    // [7]
```

Poznámka: Výše uvedený program obsahují snad všechny učebnice, které se věnují základům jazyka C. Výsledkem je vypsání řetězce „AHOJ" na obrazovku monitoru.

Vysvětlení jednotlivých řádků:

`#include <stdio.h>` Způsobí, že překladač jazyka C přečte hlavičkový soubor **stdio.h** a tento soubor začlení do programu. Tento soubor obsahuje mimo jiné informace o funkci **printf**().

`int main (void) {` Začíná funkci **main()**. Položka **int** určuje, že **main()** vrátí celočíselnou hodnotu. Položka **void** říká překladači, že **main()** nemá žádné parametry. (void = prázdný)

`printf("AHOJ"};` Je příkaz jazyka C. Volá standardní knihovní funkci **printf()**, která zobrazí zadaný řetězec znaků (v našem případě AHOJ).

`return 0;` Indikuje nulovou vracející hodnotu z funkce **main(),** což indikuje normální ukončení programu. Jakákoliv jiná hodnota představuje chybu. Položka **return** je jedním z klíčových slov jazyka C.

`}` Nakonec je program formálně ukončen, když dojde na uzavírací závorku funkce **main()**.

4.2.3 První program pro PIC16F84A

```
//***************************************************************
// program pro rozsviceni LED pripojene anodou na RB0
//***************************************************************
#include<PIC1684.h>   //hlavickovy soubor pro 16f84A – PICC Lite
void main(void){      //zacatek hlavni funkce
// nastaveni vstupu a vystupu, 1 - vstup(input), 0 / vystup(output]
TRISB=0b00000000;     //PORTB nastaven jako vystupni
PORTB=0b00000001;     // rozsvicena led dioda na RB0
}
```

Přidám opět vysvětlení jednotlivých řádků:

// vše, co je za těmito znaky do konce řádku, je považováno za komentář (nepřekládá se)

`#include<PIC1684.h>` vložíme hlavičkový soubor pro 16f84A (použitý překladač je PICC, při použití jiného překladače je potřeba použít odpovídající hlavičkový soubor)

`void main(void){` Začíná hlavní funkci **main()**. Položka **void** určuje, že **main()** nevrátí žádnou hodnotu. Parametr **void** říká překladači, že **main()** nemá žádné parametry. V naší funkci navíc netestujeme řádné ukončení funkce main() (void = prázdný)

`TRISB=0b11111110;` Tento řádek nastavuje RB0 jako výstupní (registr TRISB obsahuje na pozici RB0 hodnotu 0, RB1-RB7 jsou ponechány jako vstupní , 1 – vstup, 0 – výstup). Tělo hlavní funkce může obsahovat řadu příkladů navzájem oddělených od sebe středníkem.
Poznámka: Hodnotu můžeme zadat i zápisem v desítkové soustavě, potom by řádek vypadal následovně: TRISB=0;

PORTB=0b00000001; Tímto řádkem pošleme na RB0, který je nastaven jako výstup, hodnotu log. 1.

} konec programu

4.3 Elementy jazyka C

V této kapitole se budeme zabývat základními prvky jazyka C, tedy základními prvky, z nichž se jazyk C skládá. Jsou to [8]:

- klíčová slova,
- identifikátory,
- odsazovače,
- konstanty,
- proměnné,
- komentáře,
- operátory.

4.3.1 Klíčová slova

Klíčová slova jsou zvláštní identifikátory, které mají v jazyce C speciální význam. Proto nesmí být použity v jiném významu, než jak určuje norma ISO C, což znamená, že nesmí být použita jako jména proměnných, konstant nebo funkcí. Použití malých písmen v klíčových slovech je také důležité. Například RETURN nebude považováno za klíčové slovo return. (poznámka: klíčová slova pro různé překladače se mohou nepatrně lišit).

Přehled klíčových slov jazyka C [8]:

auto	double	int	struct
break	else	long	switch

case	enum	register	typedef
char	extern	return	uninon
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

4.3.2 Idetifikátory

Identifikátory jsou jména, která přiřazujeme například proměnným, funkcím a typům.

Je možné použít:

- písmena a..z a A..Z,
- číslice 0..9,
- znak podtržítka (_).

Prvním symbolem smí být pouze **písmeno** nebo **podtržítko**, poté následuje libovolná kombinace písmen, číslic a podtržítek. Nesmí být používána česká diakritika.

Délka identifikátoru je v normě ISO C omezená 31 znaky. Každý typ kompilátoru však upravuje toto omezení.

POZOR! Rozlišují se malá a velká písmena!

4.3.3 Odsazovače (bílé znaky)

Odsazovače, jinak zvané také **bílé znaky** (mezera, tabulátor, nový řádek, posun řádku, návrat vozíku, vertikální tabulátor, nová stránka)

V obvyklém zdrojovém textu se nejčastěji můžeme setkat s prvními třemi představiteli odsazovačů. Odsazovače spolu s operátory a oddělovači stojí mezi identifikátory, klíčovými slovy, řetězci a konstantami použitými ve zdrojovém textu. Pro překladač představují dále nedělitelné celky.

4.3.4 Konstanty

Konstanty jsou čísla, znaky nebo řetězce znaků, které se nemění po celou dobu běhu programu. Konstanty definujeme po klíčovém slově **const** následovaném typem konstanty, jejím identifikátorem a po rovnítku její hodnotou ukončenou středníkem. Nejlépe si možné definice konstant vysvětlíme na příkladech:

```
const int   konstanta = 5;
const char  pismeno='A';
```

4.3.5 Proměnné

Proměnné jsou paměťová místa, která jsou přístupná prostřednictvím **identifikátoru**. Hodnota proměnné může být během výpočtu měněna. Tím se proměnné zásadně liší od **konstant**, které mají po celou dobu chodu programu **hodnotu stejnou** - konstantní.

Během provádění programu se veškeré informace nacházejí v paměti. Tamtéž jsou umístěny pomocné hodnoty, mezivýsledky i výsledky. Rozhodně by nebylo vůbec příjemné k takovým hodnotám přistupovat prostřednictvím jejich adresy. Proto je vhodné si odpovídající paměťová místa pojmenovat a pak se na ně odkazovat jménem. Nejprve tedy deklarujeme typy a identifikátory proměnných, čímž pro ně vyhrazujeme v paměti počítače místo a současně si překladač spojuje s identifikátorem proměnné informaci o jejím umístění v paměti (adrese) [6].

Proměnné **deklarujeme** uvedením **datového typu**, za kterým následuje **identifikátor**, nebo **seznam identifikátorů**, navzájem oddělených **čárkami**. Deklarace **končí středníkem**. Současně s deklarací proměnné můžeme, ale nemusíme, definovat i její počáteční hodnotu:

```
int a, b, c;
int a=10;
```

Přehled užívaných datových typů v jazyce C [9]:

Datový typ	Počet bitů v paměti	Rozsah hodnot
bit	1	0 nebo 1
sbit	1	0 nebo 1
(unsigned) char	8	0 až 255
signed char	8	- 128 až 127

Datový typ	Počet bitů v paměti	Rozsah hodnot
(signed) short (int)	8	- 128 až 127
(signed) int	16	-32768 až 32767
unsigned (int)	16	0 až 65535
(signed) long (int)	32	-2147483648 až 2147483647
unsigned long (int)	32	0 až 4294967295
float	32	$-1.5 * 10^{45}$ až $+3.4 * 10^{38}$
double	32	$-1.5 * 10^{45}$ až $+3.4 * 10^{38}$
long double	32	$-1.5 * 10^{45}$ až $+3.4 * 10^{38}$

Poznámka: Přehled datových typů byl převzat z nápovědy k programu MikroC. Reálné datové typy float (32 bit), double (64 bit) a long double (80 bit) mají obvykle jinou hodnotu [9].

4.3.6 Komentáře

K dobrým programátorským zvykům patří vkládat do programů komentáře. Komentář je poznámka, kterou přidáváme do zdrojového kódu programu. Všechny komentáře jsou překladačem ignorovány. Usnadňují čtení programů při jejich ladění a případných pozdějąích úpravách.

Komentář je část programu umístěná mezi dvojici párových symbolů /* */. V tomto případě jde o komentář, který může zasahovat přes více řádků.

```
/* Komentář */

/* Dlouhý komentář
   zapsaný na více řádcích */
```

Některé verze jazyka C dovolují podle vzoru jazyka C++ zapisovat i komentáře ve tvaru

```
// toto je jednořádkový komentář
```

Tyto komentáře končí vždy koncem řádku (je však možno jich zapsat více za sebou, pokud na každém řádku zopakujeme úvodní //).

4.3.7 Pole

V jazyce C je jednorozměrné pole seznamem proměnných stejného datového typu, ke kterým se přistupuje přes společné jméno. Jednotlivé proměnné v poli se nazývají **prvek pole**. K jednotlivým prvkům pole se přistupuje pomocí indexů. V jazyku C jsou prvky pole vždy číslovány od nuly (**nulovým** indexem). To znamená, že chceme-li pracovat s prvním prvkem pole, zadáme jako index nulu, pokud s druhým, tak zadáme jedničku, atd..

```
typ jméno_proměnné[velikost_pole]
```

```
int   celaCisla[4];
```

Grafické zobrazení pole včetně indexů je na následujícím obrázku:

0	1	2	3

Vícerozměrné pole

Jazyk C umožňuje deklarovat pole pouze **jednorozměrné pole**. Jeho prvky ovšem mohou být libovolného typu (všechny však musí být stejného datového typu). Mohou tedy být jednorozměrnými poli. To však již dostáváme pole polí, tedy **matici**. Budeme-li uvedeným způsobem postupovat dále, vytvoříme datovou strukturu prakticky libovolné dimenze. Pak již je třeba mít jen dostatek paměti. A operační systém který nám ji umí poskytnout.

typ jmeno_promenne[3][4];

typ - určuje datový typ položek pole

jmeno - představuje identifikátor pole

[4][5] - určuje rozsah jednotlivých vektorů na čtyři řádky a pět sloupců

Grafické zobrazení takového dvourozměrného pole včetně indexů jednotlivých prvků je na následujícím obrázku:

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

4.3.8 Příkaz přiřazení

Přiřazovací příkaz je nejčastěji používaným příkazem ve většině programovacích jazyků. **Operátorem přiřazení** v jazyce C je **symbol =**. **Vlevo** od = musí být výraz, odkazující se někam do paměti. Hodnotu zprava přece musíme někam zapsat. A kam jinam ji může program zapsat, než do paměti na adresu, kterou určí právě z levé strany přiřazovacího výrazu.

Příklady užití přiřazovacího příkazu:

```
a = 5;
d = 'a';
```

4.3.9 Operátory

4.3.9.1 Aritmetické operátory

V aritmetických výrazech je nutné zdůraznit, že výraz ukončený středníkem se stává příkazem. Konstruujeme je z operandů a aritmetických operátorů

```
i=2         // vyraz s přiřazením
i=2;        // příkaz
```

Aritmetické výrazy konstruujeme z operandů a aritmetických operátorů

Operátor	Činnost
+	sčítání
-	odčítání
*	násobení
%	zbytek po celočíselném dělení

Příklad: `c = a + b;`

4.3.9.2 Logické operátory

Logické hodnoty jsou dvě, **pravda** a **nepravda**. Norma ISO C říká, že hodnota **nepravda** je představována **nulou**, zatímco **pravda jedničkou**.

Operátor	Činnost
&	and (logický součin)

Operátor	Činnost
||	or (logický součet)
!	not (negace)

Pravidla pro určení výsledku známe z **Booleovy algebry**.

4.3.9.3 Relační operátory

Relační operátory slouží pro porovnání operandů, tedy zjištění, zda mají operandy stejnou hodnotu, zda je jeden větší než druhý nebo naopak (použití např. v podmínkách).

Operátor	Činnost
<	menší
>	větší
<=	menší nebo rovno
>=	větší nebo rovno
==	rovno
!=	nerovno

4.3.9.4 Bitové operátory

Již podle názvu můžeme usuzovat, že nám bitové operátory umožňují provádět operace nad jednotlivými bity. Bitové operace je však možné provádět pouze s celočíselnými hodnotami.

Operátor	Činnost
<<	bitový posun vlevo
>>	bitový posun vpravo
&	bitový and (logický součin, konjunkce)
|	bitový or (logický součet, disjunkce)
^	bitový not (negace, inverze)

Operátor	Činnost
~	bitový xor (nonekvivalence)

4.3.9.5 Priorita operandů

Přehled všech operátorů a jejich priorit od **nejvyšší** po **nejnižší** [8]

Operátory	Asociativita
( ) [ ] ->.	zleva doprava
! ~ + - ++ -- (přetypování) *(pointer) &(adresní operátor) sizeof	zprava doleva
* / %	zleva doprava
+ -	zleva doprava
<< >>	zleva doprava
< <= > >=	zleva doprava
== !=	zleva doprava
&	zleva doprava
^	zleva doprava
|	zleva doprava
&&	zleva doprava
||	zleva doprava
?:	zprava doleva
= += -= *= /= %= >>= <<= &= |= ^=	zprava doleva
,	zleva doprava

Priorita určuje, že například násobení se vyhodnotí dříve, než třeba sčítání. **Asociativita** říká, vyhodnocuje-li se výraz zleva doprava, nebo naopak.

4.4 Řízení toku programu

4.4.1 Podmíněný operátor if

Příkaz if je používaný pro větvení programu. Ternární operátor potřebuje tři operandy.

```
if(podminka) prikaz1;
else prikaz2;
```

Podmínka je výraz, který je vyhodnocen jako pravdivý nebo nepravdivý. Je-li **pravdivý**, je vykonán příkaz1, je-li **nepravdivý**, vykoná se větev else, tedy příkaz2.

Příklad 1:

```
if(a<b)
   x=a;
else
   x=b;
```

Tento příklad si můžeme vysvětlit následovně: je-li proměnná a menší než b, přiřaď proměnné x hodnotu proměnné a, v opačném pčípadě ji přiřaď hodnotu proměnné b.

Příklad 2:

```
if(TLACITKO==1) {   //test stisku tlacitka
         LED=1;     //rozsviceni LED (je-li stisknuto tlacitko)
     }
   else {
         LED=0;     //zhasnuti LED (neni-li stisknuto tlacitko)
     }
```

Je-li stisknuto tlačítko (`TLACITKO==1`) LED se rozsvítí (`LED=1`), v opačném případě bude vykonána větev else (`LED=0`).

4.4.2 Přepínač - switch

I když je příkaz **if** dobrý pro výběr ze dvou možností, stává se těžkopádným, když je potřeba pracovat s několika možnostmi. Jazyk C řeší tento problém příkazem **switch** (v Pascalu case). Příkaz **switch** je příkaz pro **vícenásobný výběr**. Používá se pro volbu **jedné z několika** variant a pracuje následovně. Hodnota je postupně testována podle seznamu celočíselných nebo znakových konstant. Když je nalezena shoda, provede se posloupnost příkazů spojená s touto hodnotou.

```
switch (celociselny_vyraz){
          case hodnota1: prikaz1;
          case hodnota2: prikaz2;
              .             .
              .             .
              .             .
          case hodnotaN: prikazN;
          default: prikazD;
}
```

Příklad:

```
// ukazka pouziti prikazu switch – maticova klávesnice 3x3
switch (kod_klavesy) {
      case  1: jedna(); break;  // 1
      case  2: dva(); break;    // 2
      case  3: tri(); break;    // 3
      case  4: ctyri(); break;  // 4
      case  5: pet(); break;    // 5
      case  6: sest(); break;   // 6
      case  7: sedm(); break;   // 7
      case  8: osm(); break;    // 8
      case  9: devet(); break;  // 9
      default: nula();
    }
```

Je-li kód klávesy 1 (tedy `kod_klavesy = 1`), vykoná se první řádek (`case 1: jedna(); break;`), tedy bude zavolána fce `jedna()`, v opačném případě se testují další řádky. Obsahuje-li proměnná `kod_klavesy` jinou hodnotu než je interval celých čísel 1 až 9, vykoná se řádek default, tedy bude zavolána fce `nula()`. Příkaz `break` říká, že tok programu nemá pokračovat následujícím řádkem.

4.4.3 Cykly

Cyklus je část programu, která je v závislosti na podmínce prováděna opakovaně. U cyklu obvykle rozlišujeme **řídicí podmínku cyklu**. Řídicí podmínka cyklu určuje, bude-li provedeno tělo cyklu, či bude-li řízení předáno za příkaz cyklu. **Tělo cyklu** je příkaz, zpravidla zapsaný v podobě bloku.

Cykly můžeme rozdělit podle toho, provede-li se tělo **alespoň jedenkrát**, a cykly, kde tělo **nemusí** být provedeno **vůbec**. Výběr typu cyklu ponechejme na vhodnosti použití v dané situaci i na zvyklostech programátora.

4.4.3.1 Cyklus while

Podmínka cyklu **while** se testuje **před** průchodem cyklu, což znamená, že cyklus tedy **vůbec nemusí proběhnout**. Programový kód pro **výraz** musí být uzavřen v kulatých závorkách.

```
while (výraz)
      příkaz
```

Příklad:

```
while(1){                //zacatek nekonecneho cyklu while  (nekonecny = 1 )
    while (TLACITKO==1) {       //test tlacitka pripojeneho na RA0
              LED=1;            //rozsviceni LED (je-li stisknuto tlacitko)
          }
              LED=0;           //zhasnuti LED (neni-li stisknuto tlacitko)
}                       // konec cyklu while
```

Dokud je stisknuto tlačítko (`TLACITKO==1`), svítí LED (`LED=1`) dioda připojená k výstupu anodou. Pokud tlačítko není stisknuté, LED (`LED=0`) zhasne. Celý tento úsek programu bude probíhat neustále, protože je zde použit nekonečný cyklus while (`while(1)`), který je z obou stran uzavřen složenými závorkami.

4.4.3.2 Cyklus do-while

Příkaz **do** je jediným z cyklů, který zajišťuje alespoň **jedno** provedení těla cyklu. Jinak řečeno, jeho testovací příkaz **příkaz** je testován až po průchodu tělem cyklu. Pokud je test, představovaný hodnotou získanou z **výrazu** splněn, provádí se tělo cyklu, tedy **příkaz**, který je typicky blokem.

```
do
       příkaz
while (výraz);
```

Příklad:

```
while(1){         //zacatek nekonecneho cyklu while  (nekonecny = 1 )
    do {            //zacatek cyklu while - probehne minimalne jednou
          LED=0;      //zhasnuti LED0 (neni-li stisknuto tlacitko)
        }
    while (TLACITKO==0); //test tlacitka pripojeneho na RA0 - podminka
    LED=1;        //rozsviceni LED0 (je-li stisknuto tlacitko), ale i LED=0}
}                 // konec nekonečného cyklu while
```

Nejprve se LED zhasne `(LED=0)`, až poté se testuje stav tlačítka `(TLACITKO==0)`. Dokud tlačítko není stisknuté, LED nesvítí `(LED=0)`. Po stisku tlačítka dojde k rozsvícení LED, problém ale je ten, že v těle nekonečného cyklu nejprve dochází v rychlém sledu za sebou ke zhasnutí LED *(LED=0)* a poté k rozsvícení LED `(LED=1)`, což se projeví nižší intenzitou svitu LED diody.

4.4.3.3 Cyklus for

Cyklus **for** se používá pro **zadaný počet opakování** příkazu nebo bloku příkazu. Má následující syntaxi.

```
for(inicializace; test-podmínky; inkrementace) příkaz;
```

Část **inicializace** se používá pro zadání počáteční hodnoty proměnné, která řídí průběh cyklu. Tato hodnota se obvykle označuje jako **řídící proměnná cyklu**. Část inicializace se provádí pouze jednou před začátkem cyklu. Část **test-podmínky** testuje řídící proměnnou cyklu na koncovou hodnotu. Je-li test podmínky vyhodnocen jako pravdivý, cyklus se opakuje. Je-li nepravdivý, cyklus se ukončí a zpracování programu pokračuje příkazem následujícím za cyklem. Test podmínky se provádí na začátku cyklu neboli před každým opakováním těla cyklu. Část **inkrementace** cyklu **for** se provádí na konci cyklu. To znamená, že část inkrementace se provádí poté, co se provede příkaz nebo blok, který tvoří tělo cyklu. Účelem inkrementace je zvyšovat (nebo snižovat) řídící proměnnou cyklu o určitou hodnotu.

Část **inkrementace** je nepovinná. Pokud neuvedeme testovací výraz **test-podmínky**, použije překladač hodnotu jedna, a tedy bude provádět nekonečný cyklus. [6]

Příklad:

```
// ukazka funkce casove smycky
void cas(void){                 //funkce casu
  for( i=0; i<50000; i++) { }        //cyklus pro funkci casu
}
```

K volání fce slouží příkaz cas(), následně bude vykonána časová smyčka, která má za úkol zaměstnat procesor na dobu, která je dána nastavením cyklu for *(i=0; i<50000; i++) { }*. Část inicializační obsahuje hodnotu 0 *(i=0)*, k této hodnotě je každým průchodem přičítána hodnota 1 *(i++)* a vše se opakuje tak dlouho, dokud platí podmínka *(i<50000)*.

4.5 Funkce

Funkce jsou základním stavebním kamenem jazyka C. Každý program v jazyce C minimálně jednu funkci obsahuje - main(). Nyní se podíváme na tvoření funkcí a na to, jak je volat.

Definice funkce je následující:

```
navratovy_typ jmeno ([parametry])
{
        telo funkce
}
```

Jméno funkce slouží k její identifikaci. Při volání funkce v programu musíme uvést za funkcí kulaté závorky, a to i tehdy, když funkce nemá žádné argumenty.

Volání funkce je vypadá pak takto:

```
Vyraz (seznam_skutecnych_parametru);
```

Výraz může být identifikátorem funkce (její jméno) nebo adresa vzniklá jeho vyhodnocením.

Parametry funkce jsou očekávaná data, která bude funkce zpracovávat. Každý parametr má své jméno a musí být určitého datového typu. Pokud funkce nemá žádné parametry, uvádí se v závorkách slůvko void. Pokud je jich více než jeden, oddělují se čárkou.
Funkce může mít jednu návratovou hodnotu. Její typ se uvádí před jménem funkce (*návratový_typ*).

Po ukončení funkce (ať již příkazem return, nebo tím že se vykonají všechny příkazy v jejím těle) se pokračuje v provádění kódu za místem kde byla funkce volána.

Návratová hodnota funkce main() se vrací operačnímu systému. Zaběhnutá praxe je, že návratová hodnota 0 (tj. return 0) znamená úspěšné ukončení programu, jakákoliv jiná hodnota značí chybu.

Příklad:

```
//***************************************************************************
// Rozblikani osmi LED diody pripojene na RB0 anodou
//***************************************************************************

//promenne
     int i=0;
     sbit LED at RB0_bit;  // LED je pripojena na pinu RB0 – prostředí MikroC

//vedlejsi funkce – casova smycka
void cas(void){              //funkce casu
  for( i=0; i<50000; i++) { }      //cyklus pro funkci casu
}

//hlavni funkce
void main(void){     //zacatek hlavni funkce

TRISA=0b11111111;     //nastaveni portu A jako vstupy ( 1=vstup , 0=vystup )
TRISB=0b11111110;     //nastaveni portu B jako vyvstupy  ( 1=vstup , 0=vystup )

PORTB=0b00000000;    //zhasnuti LED na portu B

while(1){      //zacatek nekonecneho cyklu while  (nekonecny = 1 )

    LED=1;         //rozsviceni LED pripojene na RB0
    cas();           //volani funkce casu
    LED=0;         //zhasnuti LED připojené na RB0
    cas();           //volani funkce casu

}   //konec cyklu while
}   //konec hlavni funkce
```

5. Použitá vývojová prostředí

V této části se zaměříme na práci s vývojovými prostředími. Kompletní manuál v anglickém jazyce je obvykle možné stáhnout z www stránek výrobce softwaru. Na tomto místě uvedeme jen krátký manuál, který popisuje základy práce s příslušným vývojovým prostředím jako je např. postup založení nového projektu, kompilace zdrojového kódu, simulace atd.

V první části se zaměříme na program MPLAB, typické vývojové prostředí pro psaní programů v assembleru, které je možné doplnit např. o freewarový překladač PICC lite od firmy Hi-Tech a poté je možné programovat aplikace v jazyce C.

V druhé části popíšeme vývojové prostředí společnosti Mikroelektronika, která nabízí jednotné vývojové prostředí pro jazyky vyšší úrovně MikroC, MikroBasic a MikroPascal (navíc pro mikrokontroléry různých výrobců). K prostředí navíc existuje velmi kvalitně zpracovaný systém nápovědy (Help), kde je navíc uvedeno velké množství zpracovaných úloh, které můžou v začátku významně pomoci.

V poslední části zmíníme možnost testovat napsané úlohy v simulačním programu Multisim společnosti National Instruments, v němž se dají simulovat veškeré obvody od slaboproudých až po silnoproudá zapojení a kromě obvyklých vlastností, jako je sestavení obvodů ze součástek v knihovnách, obsahuje Multisim i měřící přístroje, které lze vložit na pracovní plochu, zapojit je do schématu a měřit s nimi [10].

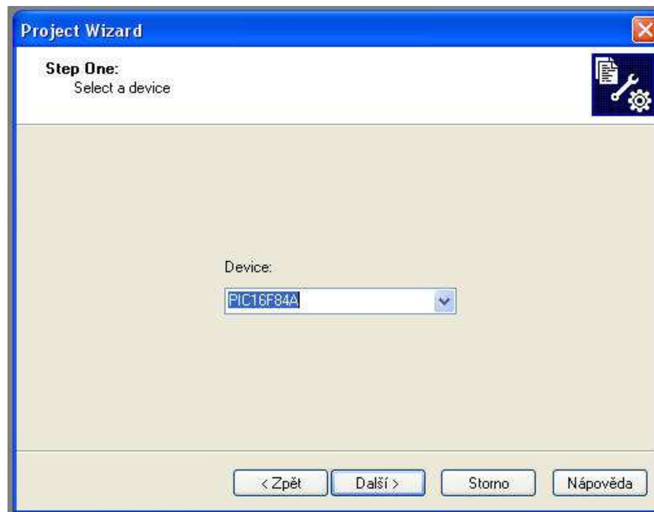
5.1 MPLAB

5.1.1 MPLAB - Založení nového projektu v IDE MPLAB

Naše programy budeme psát v MPLABu, vývojovém prostředí od firmy Microchip. Prvním krokem je založení nového projektu.

MPLAB IDE / Založení nového projektu

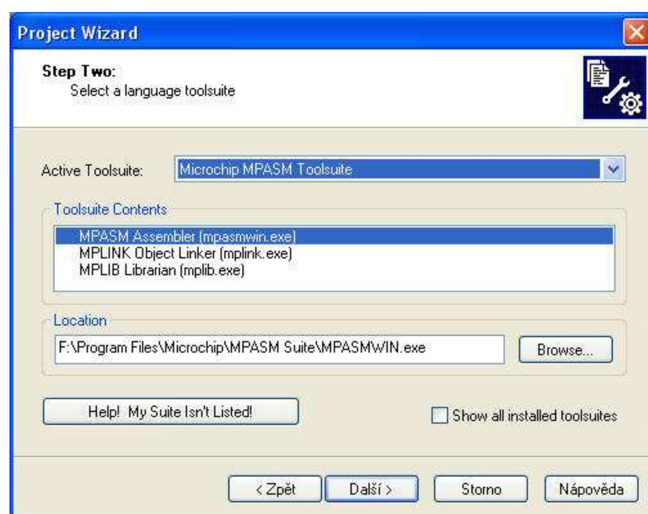
Nový projekt můžeme (doporučuji) založit pomocí průvodce v menu **Project/Project Wizard …** Zde projdeme průvodce. V prvním přivítacím okně jen klikneme na tlačítko **Další**. Následuje okno s názvem **Step One: Select Device**, kde vybereme typ součástky, se kterou budeme pracovat, v našem případě to bude PIC16F84A (obr. č. 9)



Obr. č. 9: Krok 1 – výběr součástky

Pokračujeme stiskem tlačítka **Další.**

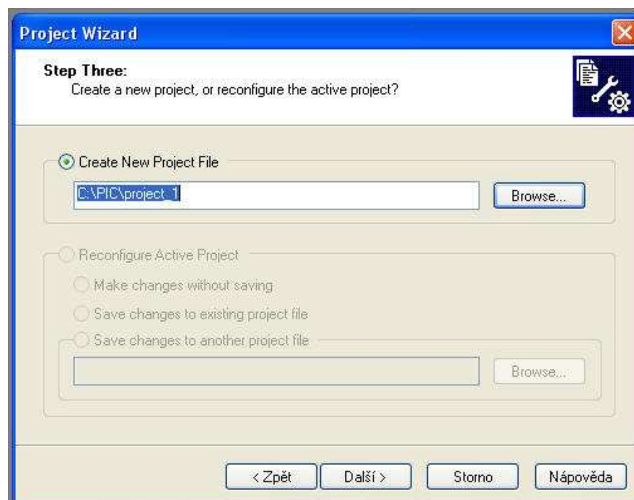
Ve druhém kroku (obr. č. 10) vybereme **Microchip MPASM Toolsuite** (ta suita obsahuje: překladač MPASM, linker MPLINK, progam na knihovny MPLIB) Všechny ty programy jsou někde nainstalovány, a musí se zadat, kde je jejich adresář, obvykle C:\Program Files\Microchip\MPASM Suite\)



Obr. č. 10: Krok 2 – výběr překladače

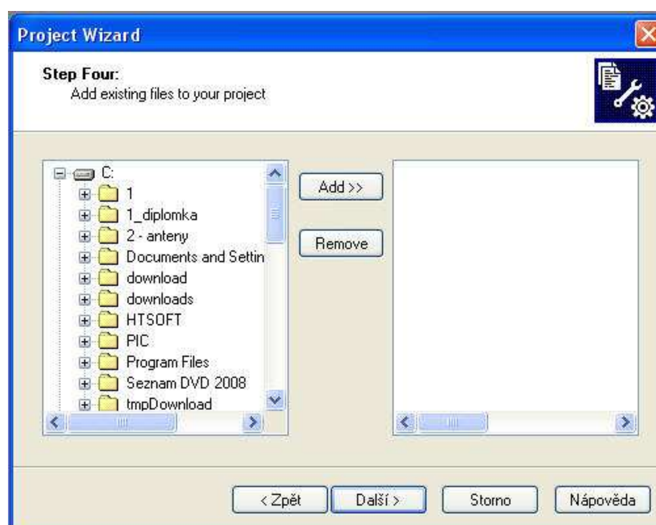
Poznámka: Pokud budeme chtít použít překladač pro jazyk C, v Aktive Toolsuite vyvereme HI-TECH PICC Toolsuite, překladač však musí být nejprve doinstalován (není součástí distribučního souboru od firmy Microchip).

Ve třetím kroku průvodce (obr. č. 11) **zvolíme jméno** pro náš nový projekt (např. projekt_1 – zde nepoužíváme českou diakritiku ani mezery a tečky), a zadám cestu k adresáři, kde bude náš projekt uložen (k tomu využiji tlačítko Browse)



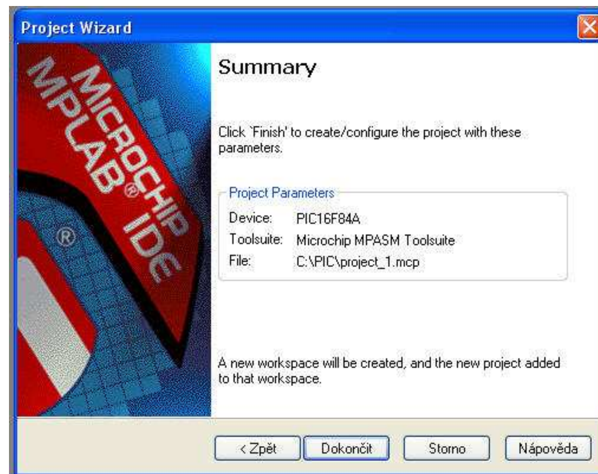
Obr. č. 11: Krok 3 – výběr umístění pro uložení projektu

Ve čtvrtém kroku (obr. č. 12) můžeme do projektu **přidat zdrojové soubory**, pokud již existují. Neexistují-li, pokračujeme kliknutím na tlačítko **Další**.



Obr. č. 12: Krok 4 – vložení existujících zdrojových souborů do projektu

Poslední okno průvodce (obr. č. 13) zobrazuje jen souhrn nastavených parametrů pomocí průvodce. Na závěr kliknu na tlačítko **Dokončit**. Tímto máme založen prázdný projekt.



Obr. č. 13: Summary – rekapitulace zadaných údajů

5.1.2 Založení nového projektu bez použití průvodce:

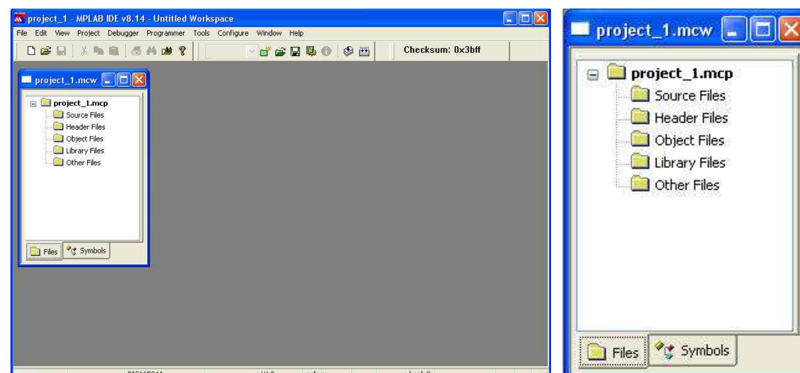
Po zapnutí se musí nastavit součástka: To se provede nahoře v menu **Configure / Select Device** - zde vyberu typ PIC16F84A

Dále musíme založit nový projekt: Nahoře v menu **Project / New** musíme zadat nějaký název, potom zvolit adresář, ve kterém ten projekt bude uložem, adresář musí existovat (stejné okno jako v kroku tři průvodce)

Nahoře v menu **Project / Set Language Tools Location** / vybereme Microchip MPASM Toolsuite (2x na něj klikneme) a vybereme Executables (ta suita obsahuje: překladač MPASM, linker MPLINK, progam na knihovny MPLIB). Všechny ty programy jsou někde nainstalovány, a musíme zvolit, kde je jejich adresář, obvykle C:\Program Files\Microchip\MPASM Suite\ a tam se všechny nacházejí a je třeba zadat k nim pomocí Browse správnou cestu. Tímto máme založený nový projekt.

Zobrazení vytvořeného projektu

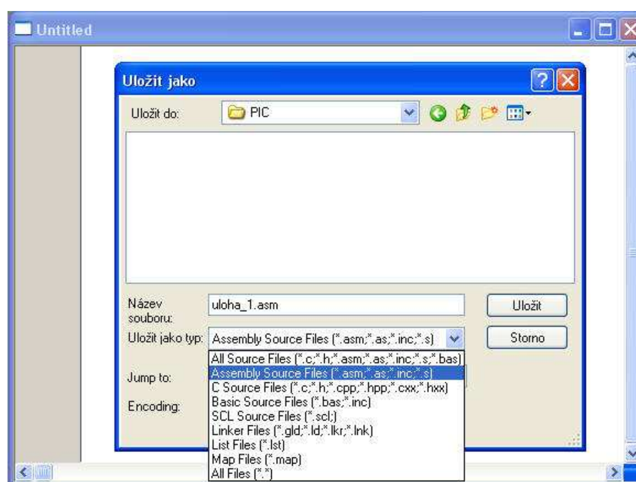
Pro zobrazení právě vytvořeného projektu s názvem Project_1 zvolíme v menu **View/Project** (obr. č. 14).



Obr. č. 14: Zobrazení vytvořeného projektu

Založení nového zdrojového souboru

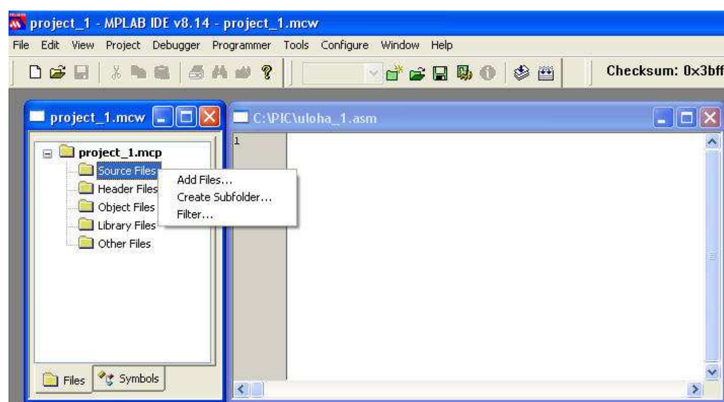
Nahoře v menu **File / New** se zobrazí nové okno pro napsání zdrojového souboru (obr. č. 15). Před psaním zdrojového kódu pomocí menu **File / Save As** zdrojový soubor uložíme. Zde přiřadíme zdrojovému souboru vyžadované jméno a příponu **asm** či **c** (ale použít můžeme i další, jak je patrné z obrázku) podle toho, v jakém programovacím jazyce budeme programovat. MPLab nám bude barevně odlišovat jednotlivé elementy zdrojového kódu. Tato funkce nám velmi zpřehlední psaní zdrojových souborů, navíc zaneseme do zdrojového souboru méně syntaktických chyb, které po následném překladu musíme opravit.



Obr. č. 15: Založení nového zdrojového souboru

Vložení zdrojového souboru do projektu

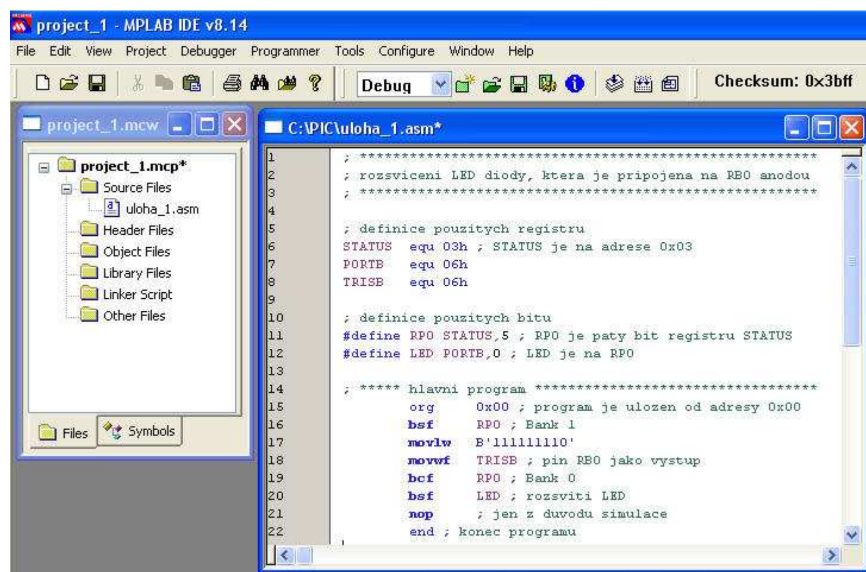
Před překladem hotového zdrojového souboru musí být tento zdrojový soubor vložen do projektu (obr. č. 16).



Obr. č. 16: Vložení zdrojového souboru do projektu

Pro vložení zdrojového souboru do projektu označíme v okně projektu **Source Files** a po kliknutí pravým tlačítkem myši se zobrazí nabídka, kde zvolíme **Add Files**, a v následně zobrazeném okně zvolíme zdrojový soubor, který chceme vložit do projektu.

Nyní je vše již připraveno pro napsání zdrojového kódu požadované aplikace, jak je vidět z obrázku č. 17.



Obr. č. 17: Založený projekt s napsaným zdrojovým kódem

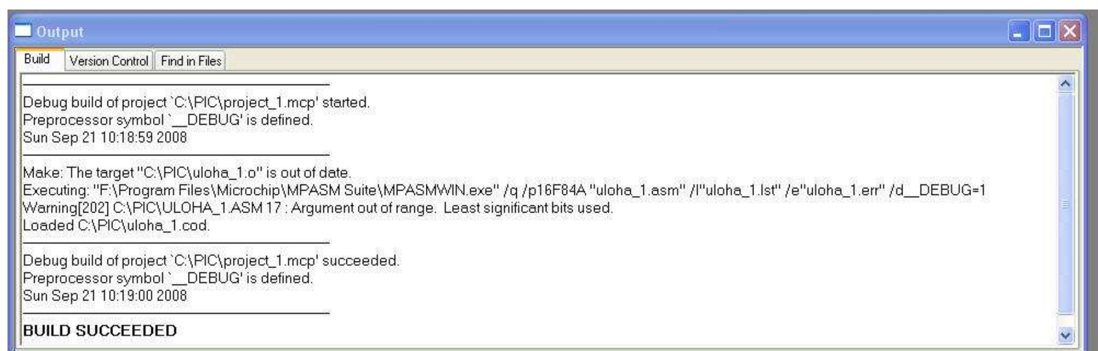
Při prvním spuštění vývojového prostředí MPLab nejsou zobrazovány v okně zdrojového souboru čísla řádků. Postup pro zobrazení: klikneme kamkoliv do okna zdrojového souboru (okno musí být aktivní), dále klikneme pravým tlačítkem myši a z kontextového menu zvolíme **Properties** (možno i z menu **Edit / Properties**), zobrazí se okno **Editor Options** a na záložce **'ASM' File Types** zatrhneme **Line Numbers**.

5.1.3 Překlad (kompilace) zdrojového souboru

Když máme zdrojový soubor napsaný, přeložíme ho.

Z menu **Project / Build all** (nebo stisk klávesy Ctrl+**F10**). V okénku **Output** se objeví hlášení překladače, pokud je všechno v pořádku, objeví se **BUILD SUCCEEDED** a pokud jsou někde nějaké chyby, překladač vypíše **BUILD FAILED** a nahoře nad tím je napsáno, co kde je špatně, to se ostatně pozná už při psaní, např. instrukce jsou modře. Zdrojový program opravujeme tak dlouho, až je vypsáno **BUILD SUCCEEDED** (obr. č.18)**.**

Rychlejší volbou je **Project / Make** (nebo stisk klávesy **F10**). Zde se překládá pouze zdrojový soubor.



Obr. č. 18: Překlad zdrojového kódu

5.1.4 Simulace

MPLab nám umožňuje zobrazit obsah Speciálních funkčních registrů, pamětí, či proměnných. Pomocí simulace si můžeme otestovat náš program, zda plní správně svou funkci.

MPLAB simulátor spustíme z horního menu **Debugger / Select Tool / MPLAB SIM** Tím zapneme nástroj pro simulaci (objeví se lišta na obrázku 19)

Obr. č. 19: Simulátor MPLABu

F7 - Step Into (krokování) - při ladění programu provede jeden řádek (příkaz) programu. Pokud je na řádku volána programová rutina, provede se její otevření a přesun na první příkaz.

F8 - step over – při ladění programu provede jeden řádek (příkaz) programu. Pokud je na řádku volána programová rutina, provede se celá.

Step Out - při opuštění programové rutiny.

Reset - vykoná reset procesoru.

Animate - zpomaleně probíhá program, mění se obsahy registrů (v menu Debugger / Settings / Animation/RealTime Updates se dá nastavit rychlost animace).

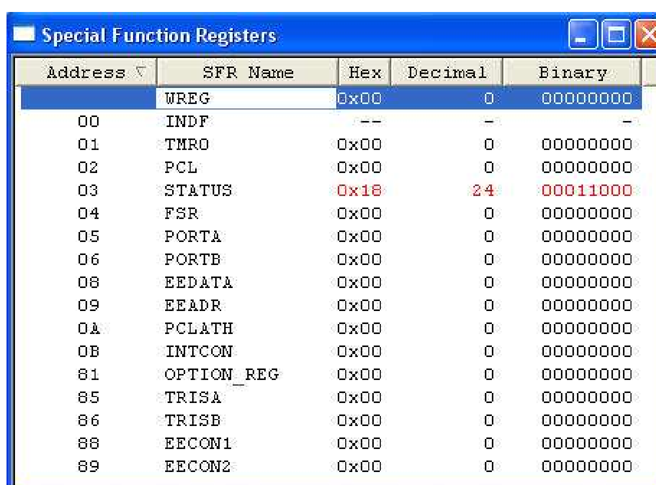
F5 - **Halt** - zastavení animace.

F9 - **Run** - rozběhnutí programu až do nalezení dalšího breakpointu.

Breakpoint - značka, která způsobí zastavení bežícího programu. Breakpointy obvykle vkládáme na významné řádky. Nastavuje se v okénku **Source File** - klikneme pravým tlačítkem na příslušnou instrukci a dáme **Set Breakpoint** nebo 2x klik na příslušné instrukci klikneme levým tlačítke myši. Breakpoint se zruší stejným způsobem - **Remove Breakpoint** nebo 2x levým tlačítkem myši. Po nastavení breakpointu můžeme program pustit pomocí Run nebo Animate. Program se v daném místě se zastaví. Breakpionty se dají také všechny zapnout (Enable) nebo vypnout (Disable) přičemž zůstávají zadány na svých místech. Odstraní se pomocí **Remove**.

5.1.5 Zobrazení hodnot Speciálních funkčních registrů a proměnných

Horní menu **View / Special Function Registers** – zobrazí hodnoty všech Speciálních funkčních registrů (obr. č. 20).



Address	SFR Name	Hex	Decimal	Binary
	WREG	0x00	0	00000000
00	INDF	--	-	-
01	TMR0	0x00	0	00000000
02	PCL	0x00	0	00000000
03	STATUS	0x18	24	00011000
04	FSR	0x00	0	00000000
05	PORTA	0x00	0	00000000
06	PORTB	0x00	0	00000000
08	EEDATA	0x00	0	00000000
09	EEADR	0x00	0	00000000
0A	PCLATH	0x00	0	00000000
0B	INTCON	0x00	0	00000000
81	OPTION_REG	0x00	0	00000000
85	TRISA	0x00	0	00000000
86	TRISB	0x00	0	00000000
88	EECON1	0x00	0	00000000
89	EECON2	0x00	0	00000000

Obr. č. 20: Speciální funkční registry

Případně **View / Watch** – v tomto okně si můžeme nechat zobrazit hodnotu jednotlivých nadefinovaných proměnných - **ADD SYMBOL**, či obsah Speciálních funkčních registrů procesoru - **ADD SFR**. V okně Watch je potom vidět, jakou hodnotu daná buňka obsahuje.

A nyní již můžeme přistoupit k vlastní simulaci.

Pokud budeme chtít odsimulovaný a zkompilovaný program nahrát na mikroprocesor PIC, použijeme výstupní ***.hex** soubor který vznikl při překladu z horního menu **Project / Make (či stiskem klávesy F10)**. Překladač vyprodukoval soubor ***.hex** (kromě jiných), ten se již pomocí programátoru nahraje do PIC16F84A.

Ještě zde upozorním na soubor s příponou .lst. Zde je výpis pro naší aplikaci:

```
MPASM  5.21                    ULOHA_1.ASM   9-21-2008  10:24:17         PAGE  1


LOC  OBJECT CODE     LINE SOURCE TEXT
  VALUE

                      00001 ; *****************************************************
                      00002 ; rozsviceni LED diody, ktera je pripojena na RB0 anodou
                      00003 ; *****************************************************
                      00004
                      00005 ; definice pouzitych registru
  00000003            00006 STATUS  equ     03h ; STATUS je na adrese 0x03
  00000006            00007 PORTB   equ     06h
  00000006            00008 TRISB   equ     06h
                      00009
                      00010 ; definice pouzitych bitu
                      00011 #define RP0     STATUS,5 ; RP0 je paty bit registru STATUS
                      00012 #define LED     PORTB,0 ; LED je na RP0
                      00013
                      00014 ; ***** hlavni program *********************************
0000                  00015                 org             0x00 ; program je ulozen od
adresy 0x00
0000   1683           00016                 bsf             RP0 ; Bank 1
Warning[202]: Argument out of range.  Least significant bits used.
0001   30FE           00017                 movlw   B'111111110'
0002   0086           00018                 movwf   TRISB ; pin RB0 jako vystup
0003   1283           00019                 bcf             RP0     ; Bank 0
0004   1406           00020                 bsf             LED ; rozsviti LED
0005   0000           00021                 nop             ; jen z duvodu simulace
                      00022                 end     ; konec programu

MPASM  5.21                    ULOHA_1.ASM   9-21-2008  10:24:17         PAGE  2


SYMBOL TABLE
  LABEL                             VALUE

LED                              PORTB,0
PORTB                            00000006
RP0                              STATUS,5
STATUS                           00000003
```

```
TRISB                             00000006
__16F84A                          00000001
__DEBUG                           1


MEMORY USAGE MAP ('X' = Used,  '-' = Unused)

0000 : XXXXXX---------- ---------------- ---------------- ----------------

All other memory blocks unused.

Program Memory Words Used:     6
Program Memory Words Free:  1018


Errors   :     0
Warnings :     1 reported,     0 suppressed
Messages :     0 reported,     0 suppressed
```

5.1.6 Rychlý přehled

Založení nového projektu: v menu **Project/Project Wizard**

Založení nového zdrojového souboru:
Menu **File / New**. Uložíme s příponou *.asm
(po uložení jako *.asm se nám jednotlivé části zdrojového kódu zabarví – zobrazuje syntaxi)

Přidání zdrojového souboru **do projektu**: V okně Project označíme **Source Files /** pravé tlačítko myši - **Add file**

Překlad zdrojového souboru: z horního menu **Project / Make (F10)**.

Simulace: otevření simulátoru z horního menu **Debugger / Select Tool / MPLAB SIM**.

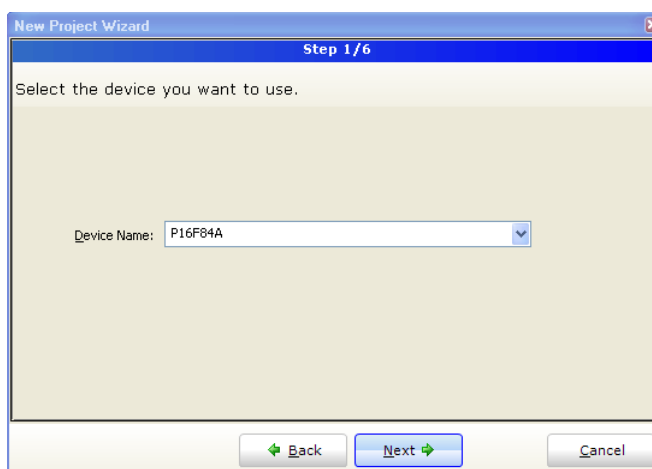
Zobrazení hodnot SFR: Nahoře **View / Special Function Registers** - všechny funkční registry procesoru.

5.2 Mikroelektronika MikroC

5.2.1 Mikroelektronika - Založení nového projektu v MikroC for PIC

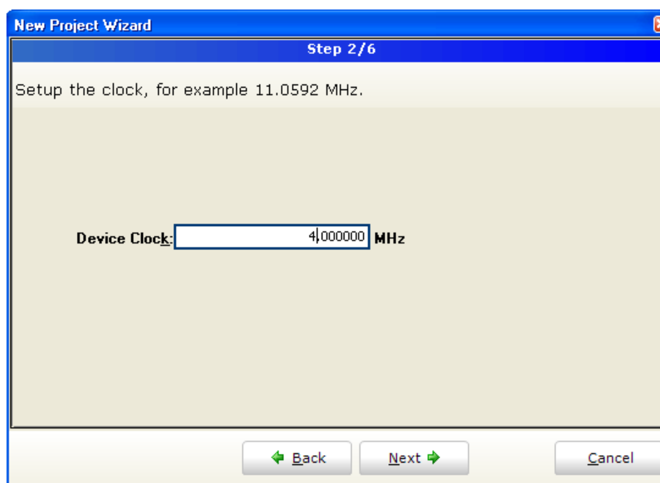
Pro psaní projektů v jazyce C použijeme nyní vývojové prostředí od společnosti Mikroelektronika s názvem MikroC for PIC. Prvním krokem bude založení nového projektu.

Nový projekt založíme pomocí průvodce, který se automaticky spustí po kliknutí v menu **Project/New Project** (**Shift+Ctrl+n**). Dále projdeme průvodce. Po prvním přivítacím okně, kde jen klikneme na tlačítko **Další**, následuje okno s názvem **Step One: Select Device**, kde vybereme typ součástky, se kterou budeme pracovat, v našem případě to bude PIC16F84A (obr. č. 21).



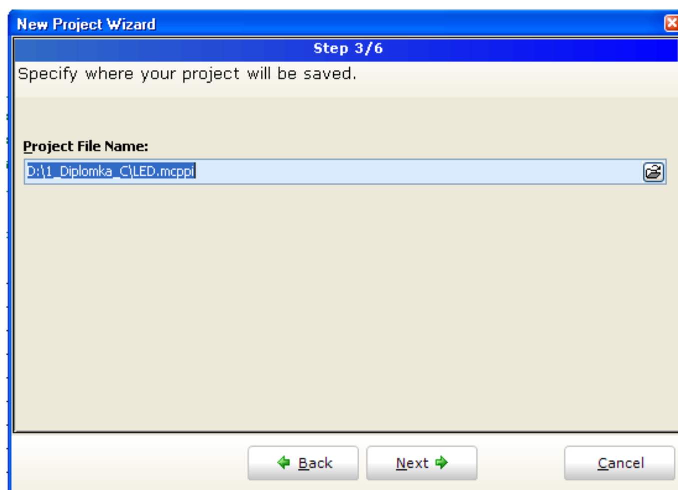
Obr. č. 21: Krok 1 – výběr součástky

Pokračujeme stiskem tlačítka Další (Next). Ve druhém kroku (obr. č. 22) nastavíme **taktovací frekvenci**, která v našem případě bude 4 MHz.



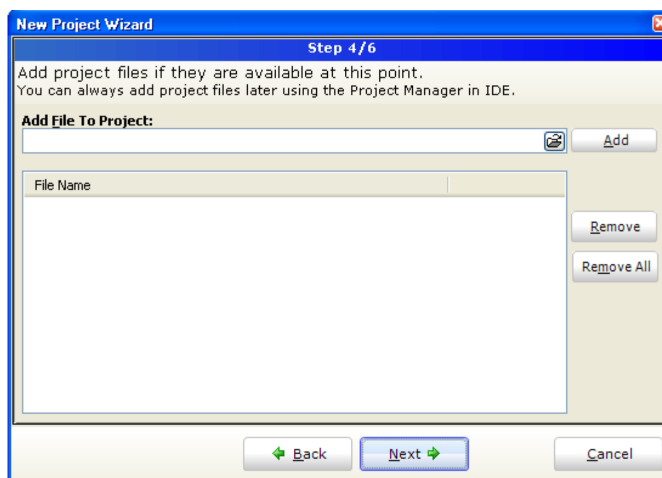
Obr. č. 22: Krok 2 – nastavení taktovací frekvence

Ve třetím kroku průvodce (obr. č. 23) **zvolíme jméno** pro náš nový projekt (např. LED – ve jméně nepoužíváme českou diakritiku ani mezery a tečky), a zadáme cestu k adresáři, kde bude náš projekt uložen (k tomu využijeme tlačítko Browse)



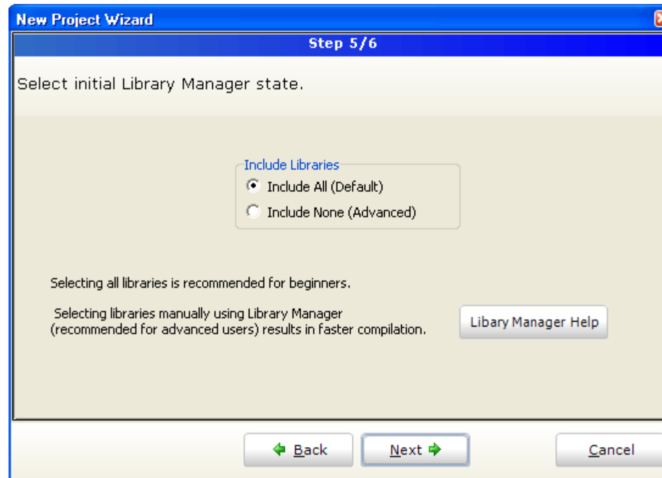
Obr. č. 23: Krok 3 – výběr umístění pro uložení projektu

Ve čtvrtém kroku (obr. č. 24) můžeme do projektu **přidat zdrojové soubory**, pokud již existují. Neexistují-li, pokračujeme kliknutím na tlačítko **Další**.



Obr. č. 24: Krok 4 – vložení existujících zdrojových souborů do projektu

V předposledním okně (obr. č. 25) jen volíme, které knihovny připojíme do projektu. Pro začátečníky je doporučena implicitně nastavená volba **Include All** (tedy všechny knihovny).



Obr. č. 25: Summary – rekapitulace zadaných údajů

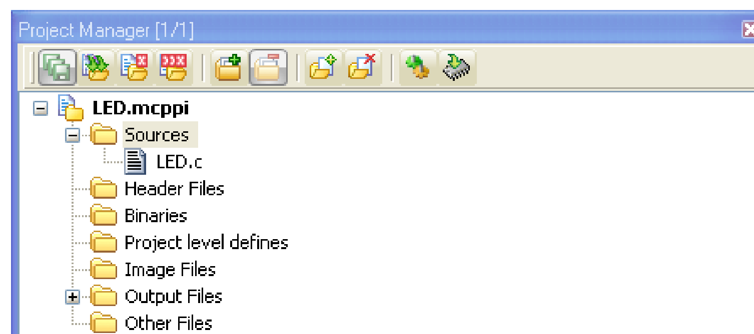
Poslední šestý krok průvodce jen oznámí úspěšné založení nového projektu. Po kliknutí na tlačítko **Dokončit (Finish)** se průvodce uzavře. Tímto máme založen nový projekt se zdrojovým souborem, kde se nachází prázdná hlavní funkce main{} (obr. č. 26), do které můžeme psát zdrojový kód.

```
LED.c
1   void main() {
.
.   }
```

Obr. č. 26: Hlavní funkce main {}pro psaní zdrojového kódu

Vložení zdrojového souboru do projektu

Před překladem hotového zdrojového souboru musí být tento zdrojový soubor vložen v projektu. Vložení zdrojového souboru probíhá sice automaticky v době zakládání nového projektu, ale někdy může vzniknout potřeba do projektu zahrnout jiný zdrojový soubor. Toto je možné dosáhnout v okně Project Manageru, který (obr. č. 27) zobrazíme z menu **View/ Project Manager**.



Obr. č. 27: Okno Project Manageru

Pro vložení zdrojového souboru do projektu označíme v okně projektu **Sources** a po kliknutí pravým tlačítkem myši se zobrazí nabídka, kde zvolíme **Add Files To Project**, a v následně zobrazeném okně zvolíme zdrojový soubor, který chceme vložit do projektu. Zdrojový soubor z projektu vymažeme tak, že ho označíme a zvolíme z kontextového menu **Remove File From Project**.

5.2.2 Překlad (kompilace) zdrojového souboru

Z menu **Project / Build all** (nebo stisk klávesy **Ctrl+F9**). V okně **Output** se objeví hlášení překladače, pokud je všechno v pořádku, objeví se **FINISHED SUCCESFULLY** a pokud jsou někde nějaké chyby, překladač napíše **FINISHED (WITH ERRORS)** a navíc červeně jsou vypsány čísla řádků a typ chyby, které se ve zdrojovém kódu nacházejí (obr. č. 28).

Messages | Quick Converter

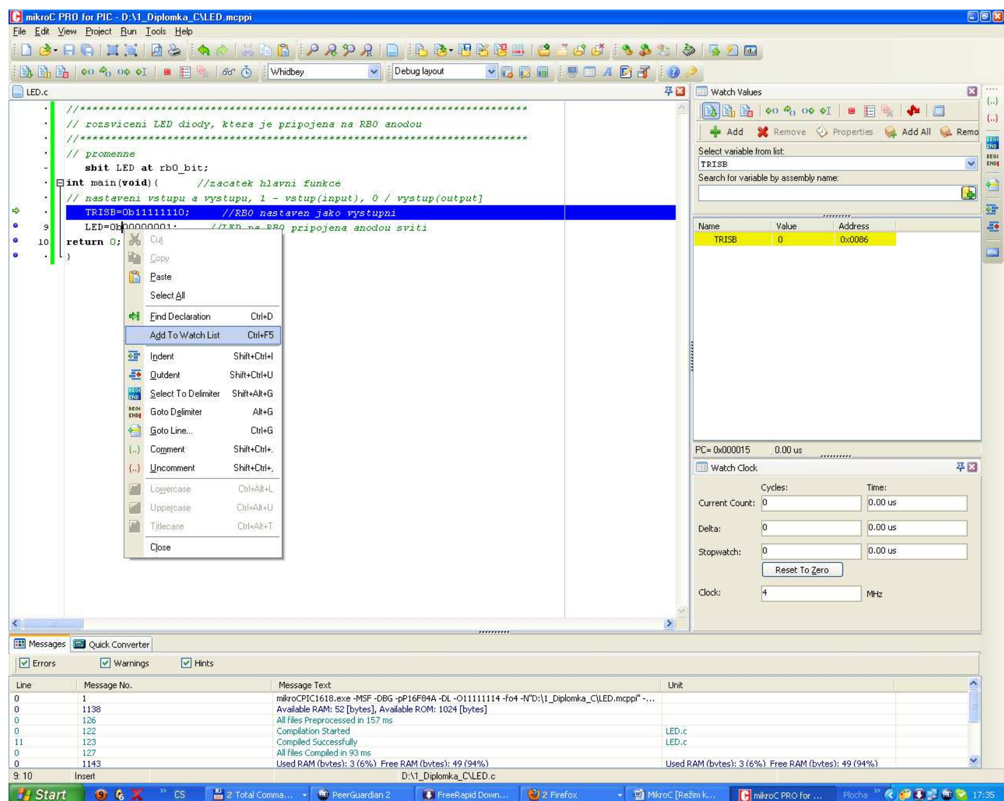
Errors | Warnings | Hints

Line	Message No.	Message Text	Unit
0	1	mikroCPIC1618.exe -MSF -DBG -pP16F84A -DL -O11111114 -fo4 -N"D:\1_Diplomka_C...	
0	1138	Available RAM: 52 [bytes], Available ROM: 1024 [bytes]	
0	126	All files Preprocessed in 157 ms	
0	122	Compilation Started	LED.c
11	123	Compiled Successfully	LED.c
0	127	All files Compiled in 93 ms	
0	1143	Used RAM (bytes): 3 (6%) Free RAM (bytes): 49 (94%)	Used RAM (bytes): 3 (6%) Free RAM (bytes): 49 (94%)
0	1143	Used ROM (program words): 26 (3%) Free ROM (program words): 998 (97%)	Used ROM (program words): 26 (3%) Free ROM (progra...
0	125	Project Linked Successfully	LED.mcppi
0	128	Linked in 188 ms	
0	129	Project 'LED.mcppi' completed: 547 ms	
0	103	Finished successfully: 08 XI 2009, 17:30:56	LED.mcppi

Obr. č. 28: Překlad zdrojového kódu

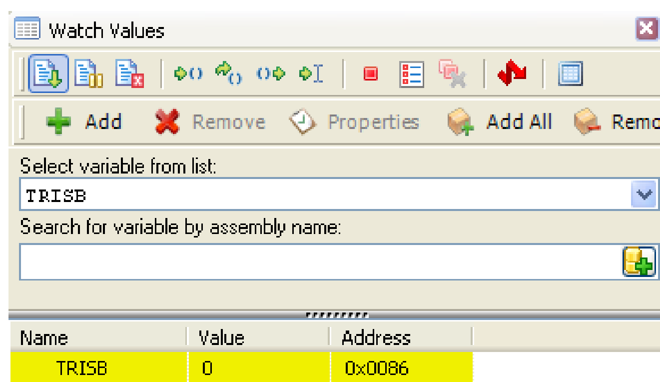
5.2.3 Simulace

MikroC Pro podobně jako MPLab nám umožňuje zobrazit obsah Speciálních funkčních registrů, či nadefinovaných proměnných. Pomocí simulace si můžeme otestovat program, zda plní správně svoji funkci. MPLAB simulátor spustíme z horního menu **Run / Start Debugger**.



Obr. č. 29: Simulátor MikroC Pro

Speciální funkční registry či proměnné vkládáme buď výběrem ze seznamu **Select variable from list** a poté klikneme na **zelené plus** v okně **Watch Values** (obr. č. 30) nebo přímo ze zdrojového kódu tak, že klikneme na příslušnou proměnnou ve zdrojovém kódu a po kliknutí na pravé tlačítko z kontextového menu vybereme položku **Add To Watch List** (obr. č. 29).



Obr. č. 30: Okno Watch Values

Horní řada ikon nám pak dovoluje simulaci programu. Najdeme zde podobná tlačítka jako tomu bylo v MPLabu, ty nejběžnější, které se využívají při simulaci, si nyní popíšeme:

Step Into (F7) - při ladění programu provede jeden řádek (příkaz) programu. Pokud je na řádku volána programová rutina, provede se její otevření a přesun na první příkaz.

Step Over (F8) - při ladění programu provede jeden řádek (příkaz) programu. Pokud je na řádku volána programová rutina, provede se celá.

Step Out (F8) - při Opuštění programové rutiny.

Start Debugger - vykoná reset procesoru.

Run / Pause Debugger – rozběhnutí programu až do nalezení dalšího breakpointu / zastavení programu.

Breakpoint – značka, která způsobí zastavení bežícího programu, breakpointy obvykle vkládáme na významné řádky, kde chceme zjistit hodnotu proměnné.

5.2.4 Rychlý přehled

Založení nového projektu: v menu **Project/New Project**
(nový zdrojový soubor je založen automaticky při založení nového projektu – přípona .c)

Zobrazení Project Manageru, z menu **View/ Project Manager**.

Přidání zdrojového souboru **do projektu**: V Project Manageru **Sources /** pravé tlačítko myši - **Add Files To Project.**

Překlad zdrojového souboru: Z menu **Project / Build all** (nebo stisk klávesy **Ctrl+F9**).

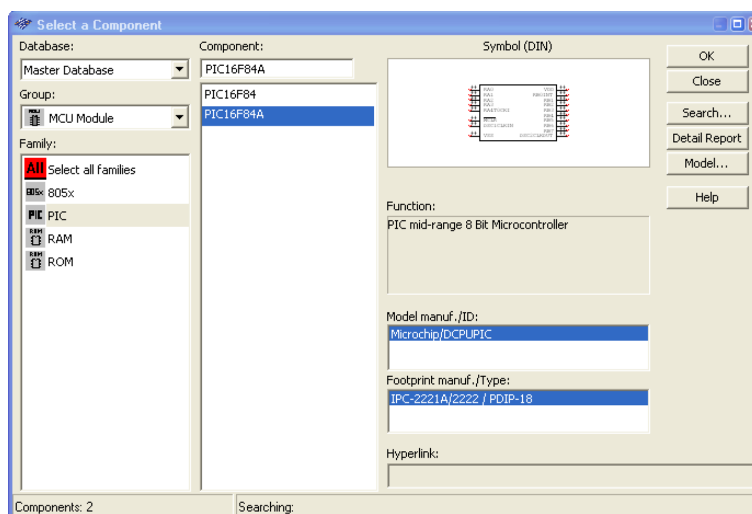
Simulace: spuštění simulátoru z horního menu **Run / Start Debugger**.

Zobrazení hodnot SFR či proměnných: označíme příslušnou proměnnou ve zdrojovém kódu a po kliknutí na pravé tlačítko z kontextového menu vybereme položku **Add To Watch List**.

5.3 NI MULTISIM

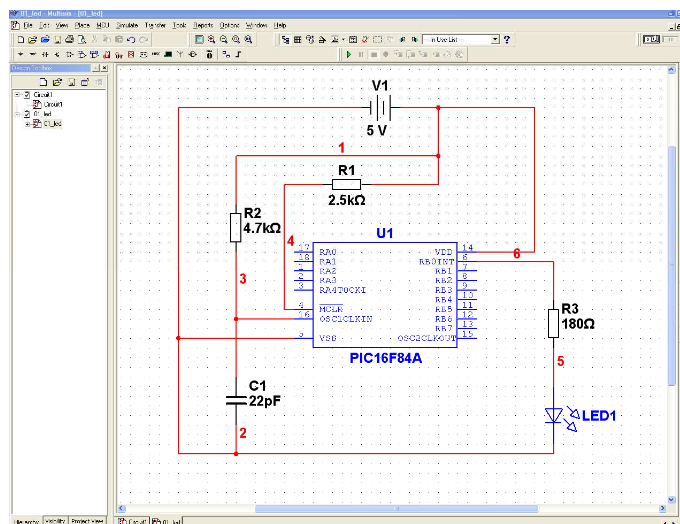
O tomto programu, který v České republice distribuuje firma Cadware, se zmíníme jen krátce, neboť se jedná o poměrně drahý komerční program. Nabízí nám však možnost postavit otestovat naprogramované úlohy, aniž bychom museli vyrábět desku s plošným spojem. Pro školy navíc existuje výrazná sleva.

Program NI Multisim (National Instruments) je výkonný simulátor analogových, digitálních a smíšených elektronických obvodů, pokračovatel známých simulátorů EWB a Multisim. NI Multisim nabízí příjemné prostředí pro snadné a rychlé kreslení schémat s možností následné simulace a analýzy jejich chování [10].



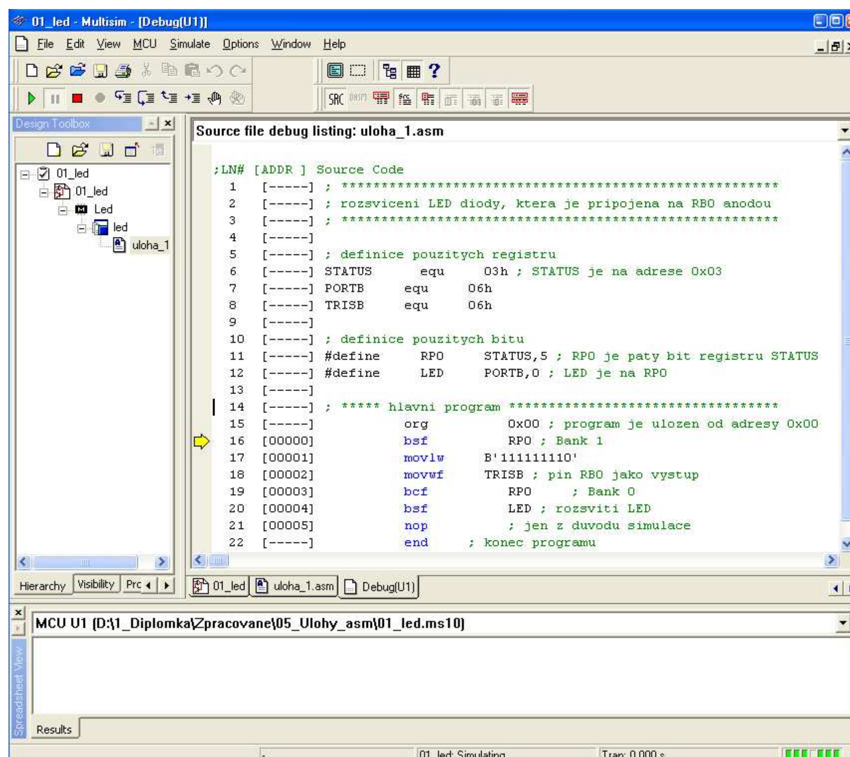
Obr. č. 31: Okno pro vložení součásty programu Multisim

V prostředí programu je možné postavit obvod, který bude řízen mikrokontrolérem. Výběr mikrokontrolérů se však omezuje jen na několik tzv. „školních“ typů z rozdílných architektur. Od firmy Microchip zde nalezneme pouze PIC16F84A. Do schématu tento obvod můžeme vložit z poměrně rozsáhlé knihovny součástek (obr. č. 31).



Obr. č. 32: Schema zapojení v programu Multisim

Po zapojení schématu (obr. č. 32) můžeme přiřadit mikrokontroléru zdrojový kód aplikace, stačí jen poklepat na PIC16F84A a v nově otevřeném okně na záložce Code klikneme na Properties a v okně MCU Code Manager již můžeme přiřadit zkompilovaný zdrojový kód (soubor s příponou *.hex), další možností je přímo přiřadit zdrojový kód napsaný v jazyce assembler a následně zkompilovat v MPASM Assembler překladači, nebo v jazyce C, kde je pro překlad využito kompilátoru Hi-Tech PICC Lite. Multisim má navíc zabudován simulátor, který umožňuje kompletní debugging zdrojového kódu (obr. č. 33).



Obr. č. 33: Schema zapojení v programu Multisim

6 Porovnání programovacích jazyků

Pro porovnání programovacích jazyků jsem se snažil vytvořit pro všechny programy stejné podmínky. Napsal jsem postupně 3 jednoduché programy (rozsvícení LED diody, rozsvícení LED diody po stisku tlačítka a blikání LED diody) nejprve v Assembleru, poté i ve třech jazycích vyšší úrovně: v jazyce C, Basicu a Pascalu. Pro řešení stejných úloh jsem se snažil použít stejné algoritmy. Navíc překladače pro jazyky vyšší úrovně byly od stejné firmy (Mikroelektronika).

Prvním programem bylo rosvícení LED diody připojené na pin RB0. Ve vlastním programu jsem nejprve pin RB0 nastavil jako výstupní, poté jsem na RB0 zapsal log. 1, tj. + napájecí napětí, protože LED dioda byla připojena anodou.

Zdrojový text aplikace v assembleru:

```
; ******************************************************
; rozsviceni LED diody, ktera je pripojena na RB0 anodou
; ******************************************************
; definice pouzitych registru
STATUS        equ    03h ; STATUS je na adrese 0x03
PORTB  equu    06h
TRISB  equ    06h

; definice pouzitych bitu
#define       RP0    STATUS,5 ; RP0 je paty bit registru STATUS
#define       LED    PORTB,0 ; LED je na RP0
; ***** hlavni program *********************************
              org          0x00 ; program je ulozen od adresy 0x00
              bsf          RP0 ; Bank 1
              movlw  B'111111110'
              movwf  TRISB ; pin RB0 jako vystup
              bcf          RP0    ; Bank 0
              bsf          LED ; rozsviti LED
              nop          ; jen z duvodu simulace
              end    ; konec programu
```

Zdrojový text první aplikace ve vyšších programovacích jazycích (postupně MikroC, MikroBasic, MikroPascal):

```
//********************************************************************
// rozsviceni LED diody, ktera je pripojena na RB0 anodou  - jazyk MikroC
//********************************************************************
// promenne
   sbit LED at rb0_bit;
int main(void){       //zacatek hlavni funkce
// nastaveni vstupu a vystupu, 1 - vstup(input), 0 / vystup(output]
   TRISB=0b11111110;     //RB0 nastaven jako vystupni
```

```
   LED=0b00000001;     //LED na RB0 pripojena anodou sviti
return 0;
}
```

```
' ***************************************************************************
' rozsviceni LED diody, jazyk MikroBasic
' ***************************************************************************
program led
main:
    TRISB = 0b11111110
    TRISA = 255
    PORTB = 1
end.
```

```
{*************************************************************************
 rozsviceni LED diody - jazyk MikroPascal
 *************************************************************************}

program led;
begin
  TRISA := %11111111;  // PORTA vstupni
  TRISB := %11111110;  // RB0 jako vystup
  PORTB := 1;          // rozsviceni LED
end.
```

Druhým programem bylo rosvícení LED diody připojené na pin RB0 po stisku tlačítka připojeného na RA0. Ve vlastním programu jsem nejprve pin RB0 nastavil jako výstupní, PortA jsem ponechal jako vstupní, poté jsem v závislosti na stisku tlačítka připojeního k RA0 na RB0 zapsal log. 1 (bylo-li stisknuté), nebo log. 0 (nebylo-li stisknuté), tj. buď připojená LED svítila nebo nesvítila.

Zdrojový text aplikace v assembleru:

```
; *******************************************************
; rozsviceni LED diody, ktera je pripojena na RB0 anodou
; pote, co je stisknuto tlacitko
; *******************************************************
; ********* Speciální funkcni registry **********************************
STATUS       equ    03h ; STATUS je na adrese 0x03
PORTA  equ    05h
TRISA  equ    05h ;použijeme prime adresovani
PORTB  equ    06h
TRISB  equ    06h ;použijeme prime adresovani
; ********* Definice bitu **********************************************
#define      RP0    STATUS,5 ; RP0 je paty bit registru STATUS
#define      LED    PORTB,0 ; LED je na RB0
#define      TL     PORTA,0 ; Tlacitko je na RA0
; ********* Hlavni program *********************************************
```

```
            org         0x00 ; program je ulozen od adresy 0x00
            bsf         RP0 ; Bank 1
            movlw  B'111111111'
            movwf  TRISA ; cely PORTA jako vstup
            movlw  B'111111110'
            movwf  TRISB ; pin RB0 jako vystup
            bcf         RP0   ; Bank 0
Testuj btfss  TL
            goto   Zhasni
            bsf         LED ; rozsviti LED
            goto   Testuj
Zhasni bcf         LED
            goto   Testuj
            nop         ; jen z duvodu simulace
            end    ; konec programu
```

Zdrojový text druhé aplikace ve vyšších programovacích jazycích (postupně MikroC, MikroBasic, MikroPascal):

```
//********************************************************************************
// stiskem tlacitka na RA0 se rozsviti LED pripojena anodou na RB0 - jazyk MikroC
//********************************************************************************
// promenne
   sbit TLACITKO at ra0_bit;    //TLACITKO je na RA0
   sbit LED at rb0_bit;    //LED je na RB0

int main(void){      //zacatek hlavni funkce
// nastaveni vstupu a vystupu, 1 - vstup(input), 0 / vystup(output]
   TRISA=0b11111111;     //PORTA nastaven jako vstupni
   TRISB=0b11111110;     //RB0 nastaven jako vystupni
   PORTB=0b00000000;
while(1){         //zacatek nekonecneho cyklu while  (nekonecny = 1 )
   if(TLACITKO==1) {       //test tlacitka pripojeneho na RA0
         LED=1;        //rozsviceni LED0 (je-li stisknuto tlacitko)
      }
   else {
         LED=0;        //zhasnuti LED0 (neni-li stisknuto tlacitko)
        }
}                  // konec cyklu while

return 0;

' ***************************************************************************
'  stiskem tlacitka na RA0 se rozsviti LED pripojena anodou na RB0
' - jazyk MikroBasic
' ***************************************************************************
program tlacitko
    dim   TL as sbit at PORTA.B0
    dim   LED as sbit at PORTB.B0
main:
```

```
'   hlavni program
    TRISB = %11111110
    TRISA = %11111111
while true
    if TL = 1 then
       PORTB = 1
       else
       PORTB = 0
    end if
wend
end.

//*************************************************************************
// stiskem tlacitka na RA0 se rozsviti LED pripojena anodou na RB0
// - jazyk MikroPascal
//*************************************************************************
program tlacitko;
  var TL: SBIT AT PORTA.0;
begin
  TRISA := %11111111;  { PORTA vstupni }
  TRISB := %11111110;  // RB0 jako vystup
  While TRUE do
    begin
       if TL = 1 then       // je stisknute tlacitko?
       PORTB := 1          // rozsviceni LED
       else
       PORTB := 0          // zhasnuti LED
    end;                 // konec nekonecneho cyklu
end.
```

Třetím programem bylo rozblikání LED diody připojené na pin RB0. Ve vlastním programu jsem nejprve pin RB0 nastavil jako výstupní, poté jsem na RB0 zapsal log. 1, použil časovou smyčku dlouhou 1s, na RB0 zapsal log. 0, opět zavolal časovou smyčku a to vše běželo neustále dokola.

Zdrojový text aplikace v assembleru:

```
; *******************************************************
; rozblikani LED diody, ktera je pripojena na RB0 anodou
; *******************************************************

; definice pouzitych registru
Status equ    0x03;
TrisA  equ    0x05;
PortA  equ    0x05;
TrisB  equ    0x06;
PortB  equ    0x06;
CounterA      equ    0x0C
```

```
CounterB     equ    0x0D
CounterC     equ    0x0E
; definice bitu
#define LED  PortB,0
#define RP0  Status,5
; ***** hlavni program ********************************
             org    0
             bsf    RP0          ; Bank 1
             movlw  b'11111111'
             movwf  TrisA        ; PortA je vstupni
             movlw  b'11111110'
             movwf  TrisB      ;RB0 jako vystup
             bcf RP0             ;Bank 0
             goto   Start
;PIC Time Delay = 1,00000200 s with Osc = 4000000 Hz
Cekej        movlw  D'6'
             movwf  CounterC
             movlw  D'19'
             movwf  CounterB
             movlw  D'173'
             movwf  CounterA
loop         decfsz CounterA,1
             goto   loop
             decfsz CounterB,1
             goto   loop
             decfsz CounterC,1
             goto   loop
             retlw  0
Start  bsf LED             ; rosvit LED
             call   Cekej  ; celej 1 sekendu
             bcf    LED          ; zhasni LED
             call   Cekej  ; celej 1 sekendu
             goto   Start  ; neustale opakuj
             end
```

Zdrojový text druhé aplikace ve vyšších programovacích jazycích (postupně MikroC, MikroBasic, MikroPascal):

```
//************************************************************************
// Rozblikani osmi LED diod pripojenych na PORTB anodou - jazyk MikroC
//************************************************************************
void main(){
  TRISA = 0xFF;           // nastaveni PORTA jako vstup
  TRISB = 0;              // nastaveni PORTB jako vystup
  PORTB = 0;              // uvodni inicializace PORTB
  do {                    // nekonecna smycka
         PORTB = 255;           // rosvit vsechny LED diody na PORTB
         Delay_ms(1000);        // cekej 1 sekundu
         PORTB = 0;             // zhasni vsechny LED diody
         Delay_us(1000000);        // cekej 1 sekundu
```

```
  } while(1);         // konec nekonecneho cyklu
}

'*************************************************************************
' Rozblikani osmi LED diod pripojenych na PORTB anodou - jazyk MikroBasic
'*************************************************************************
program blik
dim   LED as sbit at PORTB.B0
main:
    TRISA = %11111111
    TRISB = %11111110
while true
    LED = 1      ' rosvit led
    Delay_ms(1000) ' cekej 1 sekundu
    LED = 0      ' rosvit led
    Delay_ms(1000) ' cekej 1 sekundu
wend
end.

//*************************************************************************
// Rozblikani osmi LED diod pripojenych na PORTB anodou - jazyk MikroPascal
//*************************************************************************
program blik;
begin
  TRISA := %11111111;  { PORTA vstupni }
  TRISB := %11111110;  // RB0 jako vystup
  While TRUE do
    begin
    PORTB := 1;          // rozsviceni LED
    Delay_ms(1000);      // 1 sekunda pauza
    PORTB := 0;          // zhasnuti LED
    Delay_ms(1000);      // 1 sekunda pauza
    end;                 // konec nekonecneho cyklu
end.
```

Výsledky jsou shrnuty v přehledové tabulce:

Program	Program 1: LED		Program 2: Tlacitko		Program 3: Blikání LED	
	Velikost .hex	Počet slov	Velikost .hex	Počet slov	Velikost .hex	Počet slov
Assembler	67 B	6	108 B	13	203 B	27
MikroC	92 B	8	129 B	14	268 B	39
MikroBasic	117 B	11	129 B	14	264 B	38
MikroPascal	109 B	9	129 B	14	268 B	39

V tabulce jsou shrnuty informace poskytnuté překladačem při překladu, tedy počet užitých slov v paměti programu. Tady platí, že čím méně užitých slov, tím rychleji

pracující program (vykonání jedné instrukce trvá určitou dobu). Dále je v tabulce uvedena velikost souboru s příponou hex.

Jednoznačně se potvrzuje, že zdrojový kód napsaný v Assembleru je nejefektivnější (a tedy i nejrychlejší). Assembler je také vhodný pro aplikace, které potřebujeme distribuovat ve velkých sériích (můžeme obvykle použít mikrokontrolér s menší kapacitou pamětí a tím uspořit výrobní náklady).

U vyšších programovacích jazyků si můžeme všimnout délky programu, které potřebují pro svou činnost výrazně méně vývojářem napsaných řádků zdrojového kódu, jak plyne z přiložených zdrojových textů aplikací. Tvorba aplikací ve vyšších programovacích jazycích tedy zabere méně času vývojáře. Vyšší programovací jazyky jsou tedy velmi vhodné pro aplikace psané na zakázku, kdy nejde o větší série a nevadí vyšší cena použitého hardware.

7 Závěr

Bakalářská práce je zaměřena na zpracování souboru úloh postavených na jednočipových mikrokontrolérech PIC. Pro většinu úloh byl použit mikrokontrolér PIC16F84A, při použití A/D převodníku byl použit mikrokontrolér PIC12F675, a to z důvodu absence A/D převodníku v PIC16F84A, pro komunikaci pomocí seriové sběrnice I2C byl vybrán mikroprocesor PIC16F877 (opět z důvodu absence SDA a SCL u PIC16F84A), který byl spojen se sériovou EEPROM pamětí 24C02. Vlastní práce s PIC12F675 a PIC16F877 se neliší s PIC16F84A.

V bakalářské práci jsou po krátkém úvodním popisu PIC16F84A zpracovány stručné manuály pro programování v jazyce Assembler včetně popisu instrukcí, jazyce C a navíc i manuály pro použitá vývojová prostředí MPLAB a MikroC. Tyto manuály je možné použít jako materiál pro podporu výuky na střední škole.

Úlohy byly řešeny nejprve v jazyce nejnižší úrovně – jazyce Assembler, poté podobné úlohy byly řešeny v jazyce vyšší úrovně – jazyce C. Ukazuje se, že úlohy řešené v Assembleru jsou velmi rychlé, šetří kapacitu paměti, které u malých zařízení typu mikrokontrolér není nazbyt. Assembler se hodí zvláště tam, kde potřebujeme vyrobit velké série, zde ušetříme na vlastním hardwaru (můžeme zvolit levnější mikrokontrolér s nižší kapacitou paměti) nebo tam, kde potřebujeme maximální rychlost vykonání programu. Psaní úloh v Assembleru většinou zabere více času vývojáře a i počet napsaných řádků zdrojového kódu je vyšší.

Také z řešených úloh zcela jasně vyplynulo, že právě složitější úlohy jsou daleko jednoduševji řešitelné v jazyce C. Úlohy řešené ve vyšších programovacích jazycích nevyžadují takové úsilí a množství času vývojáře. Ve vyšších programovacích jazycích vývojář může využít a zahrnout do programu již předdefinované funkce, které společně s překladačem dodává výrobce softwaru. Pokud jde o počet slov použitých v paměti, tak programy napsané v jazycích vyšší úrovně zaberou asi o 30 % větší prostor v paměti programu, což se dá při dnešních cenách mikroprocesorů obvykle akceptovat. Vzhledem k tomu je použití vyšších programovacích jazyků velmi vhodné pro použití v jednočipových mikrokontrolérech.

V příloze bakalářské práce je uvedena pouze první zpracovaná úloha, a to pouze v jazyce Assembler. Všechny zpracované úlohy jsou dostupné na doprovodném CD disku. Jsou postupně řešeny nejprve v jazyce symbolických adres – Assembleru, poté v jazyce vyšší úrovně – jazyce C, od jednodušších po složitější. Navíc jsou dostupné vždy v zadání určeném jak pro učitele, tak pro žáka.

8 Seznam použité literatury:

[1] Microchip: Katalogový list PIC16F84A: http://ww1.microchip.com/downloads/en/DeviceDoc/35007b.pdf

[2] V. Čebiš , J. Kadlecová: Překlad originální dokumentace PIC16F84A, 1998

[3] Doveda Boys: Stránky procesorů PIC: http://www.cmail.cz/doveda/

[4] ASIX: *PIC krok za krokem,* komentované příklady programů pro PIC

[5] Jazyk C – přednáška: http://homel.vsb.cz/~rep75/proglang/prednasky/prednaska1.htm

[6] Metodika výuky mikroprocesorů PIC v oborech elektro, výpočetní a automatizační technika na SPŠ a SOŠ s využitím PC: http://www.spshk.cz

[7] Herbert Schildt: *Nauč se sám C*, Praha: nakladatelství SFTPRESS, 2001, ISBN 80-86497-16-X

[8] Dalibor Kačmář: Jazyk C: Učebnice pro střední školy, Praha: nakladatelství CP Books, a.s. 2000, ISBN 80-7226-295-5

[9] MikroElektronika: http://www.mikroe.com/

[10] Cadware: http://www.cadware.cz/

[11] Václav Kadlec: *Učíme se programovat v jazyce C*, Brno: nakladatelství CP Books, a.s., 2005, ISBN 80-7226-715-9

[12] Burkhard Mann: *C pro mikrokontroléry*, Praha: nakladatelství Ben – technická literatura, 2000, ISBN 3-7723-4154-3

[13] Jiří Hrbáček: *Komunikace mikrokontroléru s okolím,* Praha: nakladatelství Ben – technická literatura, 1999, ISBN 80-86056-42-2

[14] Jiří Hrbáček: *Moderní učebnice programování jednočipových mikrokontrolérů PIC 1*, BEN 2004, ISBN 80-7300-136-5

[15] Jiří Hrbáček: *Moderní učebnice programování jednočipových mikrokontrolérů PIC 2*, BEN 2007, ISBN 80-7300-137-3

[16] Elektronika E-ZIN: http://elektronika.ezin.cz/

[17] Wikipedia: 1-Wire: http://en.wikipedia.org/wiki/1-Wire

[18] Václav Vacek: *Učebnice programování PIC*, BEN 2000, ISBN 80-86056-87-2

[19] Dogan Ibrahim: *PICBasic Projects – 30 Projects using PICBasic and PICBasic Pro*, Newnes 2006, ISBN 0750668792

9 Přílohy

V první části uvedu kompletní seznam řešených úloh v rámci této bakalářské práce, následovat bude jedna ukázková úloha řešená v jazyce Assembler. Všechny zpracované úlohy se pak nacházejí na doprovodném CD nosiči.

Přehled zpracovaných úloh:

1 Assembler

1.1 Nastavení I/O brány

1.1.1 Rozsvícení LED diody

1.1.2 Tlačítko s LED

1.2 Kombinační logické obvody s PIC16F84A

1.3 Procvičování aritmetických isntrukcí

1.3.1 Součet dvou čísel

1.3.2 Součin dvou libovolných čísel

1.4 Přímé a nepřímé adresování

1.5 Časové smyčky

1.5.1 Blikání LED

1.5.2 Běžící světlo třemi způsoby

1.6 Čítač

1.7 Přerušení

1.8 Tlačítka

1.8.1 Multifunkční tlačítko

1.8.2 Multifunkční tlačítko s ošetřením zákmitů

1.8.3 Maticová klávesnice (3 způsoby ovládání)

1.9 Krokový motor

1.10 Ovládání servomotoru

1.11 Piezo

1.12 Zobrazovače

1.12.1 Sedmisegmentový zobrazovač

1.12.2 Sedmisegmentový zobrazovač – multiplexní řízení

1.13 Paměť EEPROM

2 Jazyk C

2.1 Nastavení I/O brány

2.1.1 Rozsvícení LED diody

2.1.2 Rozsvícení LED diody po stisku tlačítka

2.2 Kombinační logické obvody s PIC16F84A

2.3 Procvičování aritmetických instrukcí

2.4 Přímé a nepřímé adresování

2.5 Časové smyčky

2.5.1 Blikání LED

2.5.2 Běžící světlo

2.6 Čítač

2.7 Přerušení

2.8 Tlačítka

2.8.1 Multifunkční tlačítko

2.8.2 Multifunkční tlačítko s ošetřením zákmitů

2.8.3 Maticová klávesnice

2.9 Krokový motor

2.10 Ovládání servomotoru

2.11 Piezo

2.12 Zobrazovače

2.12.1 Sedmisegmentový zobrazovač

2.12.2 Sedmisegmentový zobrazovač – multiplexní řízení

2.12.3 LCD zobrazovač

2.13 Paměť EEPROM

2.14 AD převodník s PIC12F675

2.15 Komunikace mikroprocesoru s okolím

2.15.1 Komunikace PIC16F84A pomocí jednovodičové sběrnice DALLAS

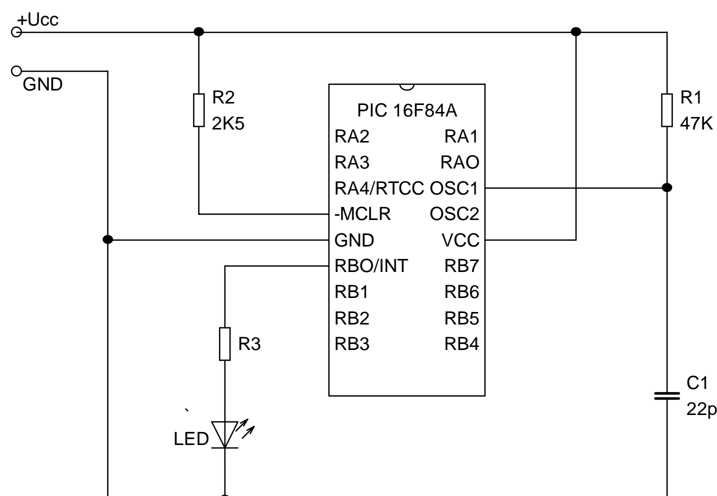
2.15.2 Komunikace PIC16F877 se sériovou pamětí 24C02 pomoci I2C

Programování jednočipů PIC

ÚLOHA č.1 TÉMA : Nastavení I/O - LED

Číslo úlohy: 1.1.1	Programovací jazyk: Assembler	Zadání: učitel

Zadání úlohy: Rozsviťte LED diodu připojenou na RB0. LED dioda je připojena na RB0 dle schematu (obr. 1).



Obr. 1 Schéma zapojení

Výukový účel úlohy: Naučit se pracovat se vstupy / výstupy mikrokontroléru.

Teoretický rozbor úlohy:

K tomu, abychom rozsvítili LED diodu, která je připojená na pin RB0 PIC16F84A anodou (katoda je potom zapojena na svorku GND), musíme nejdřív nastavit vstupně-výstupní (V/V) vývod RB0 jako výstupní. To provedeme na Stránce 1 paměti RAM a Speciálních funkčních registrů, konkrétně v registru TRISB na adrese 86h.

TRISB 86h

RB7	RB6	RB5	RB4	RB3	RB2	RB1	**RB0**
1	1	1	1	1	1	1	**0**

V/V piny RB1 - RB7 ponecháme jako vstupní, RB0 nastavíme jako výstup.

Zapíšeme-li do odpovídajícího bitu těchto registrů hodnotu 1, V/V pin se stane vstupem (což je také stav po zapnutí obvodu), zapíšeme-li nulu, V/V pin se stane výstupem. Když je

pin vstupní, můžeme ho číst, když je výstupní, můžeme do něj zapsat buď logickou 0 nebo 1. To uděláme zápisem do registru PORTB na adrese 06h.

PORTB 06h

RB7	RB6	RB5	RB4	RB3	RB2	RB1	**RB0**
							1

Logickou 1 zapíšeme na pozici RB0 v registru PORTB na adrese 06h. To nám zajistí, že se na RB0 objeví napětí +Ucc (to odpovídá log. 1).

Máme tu ale jeden problém a ten spočívá v tom, že TRISB se nachází na stránce 1 paměti RAM, kdežto PORTB na stránce 0 paměti RAM. Pokud chceme zapisovat do registru TRISB musíme zapisovat do stránky 1 paměti RAM. To lze buď přímo nebo nepřímo. My zvolíme přímý zápis. Musíme použít přímé adresování. To lze provést pomocí pátého bitu registru STATUS na adrese 03h, který se nazývá RP0. Je-li RP0 = 0, pracujeme se stránkou 1 paměti RAM. Je-li RP0 = 1, pracujeme se stránkou 1 paměti RAM.

STATUS 03h

IRP	RP1	**RP0**	TO	PD	Z	DC	C
		0/1					

Význam jednotlivých bitů registru STATUS:

IRP a RP1 – u PIC16F84A nemají pevnou funkci, jsou pro nás univerzálními bity, které můžeme použít, jak chceme

RP0 - výběr podstránky paměti, s níž pracujeme: 0 – stránka 0
1 – stránka 1

TO a **PD** jsou stavové bity, které indikují události, jako zapnutí napájení, reset, probuzení ze sleep, atd.

Z je nastaven do 1, je-li výsledek aritmetické nebo logické operace roven 0.

DC indikuje přenos (výpůjčku) v operacích sčítání (odčítání) pro dolní 4 bity.

C indikuje přenos (výpůjčku) v operacích sčítání (odčítání), požívá se také v operacích rotace.

Každý program by měl obsahovat tzv. hlavičkou, ve které bude uveden název programu, jméno autora, datum poslední úpravy, použitý překladač atd. Program je uveden celý včetně hlavičky:

```
;****************************************************************************
;*                                LED.asm                                   *
;****************************************************************************
;* Program: LED              Verze 1.00           Autor: Zdeněk Boháč       *
;* Popis: Program rozsviti LED diodu pripojenou anodou na vyvod RB0         *
;*        mikroprocesoru.                                                   *
;* Nastaveni:RC                                                             *
;*                  WDT - off                                               *
;*                  PWR - on                                                *
;*                                                                          *
;*                  +-----+ +-----+                                         *
;*              RA2 |1    +-+   18| RA1                                     *
;*              RA3 |2          17| RA0                                     *
;*        RA4/T0CKI |3          16| OSC1/CLKIN                              *
;*            /MCLR |4   16C84  15| OSC2/CLKOUT                             *
;*              GND |5          14| VCC                                     *
;*          RB0/INT |6   16F84  13| RB7                                     *
;*              RB1 |7          12| RB6                                     *
;*              RB2 |8          11| RB5                                     *
;*              RB3 |9          10| RB4                                     *
;*                  +-------------+                                         *
;*                                                                          *
;****************************************************************************

; ********* Speciální funkcni registry ******************************
status  equ     0x03            ;status je na adrese 0x03
portb   equ     0x06
trisb   equ     0x06            ;použijeme prime adresovani

; ********* Definice bitu ******************************************
#define LED     portb,0         ;LED je na pinu RB0
#define RP0     status,5        ;RP0 je 5. bit registru status

; ********* Hlavni program ******************************************
        org     0               ;program je ulozen od adresy 0x00
        bsf     RP0             ;stranka 1 pameti RAM
        movlw   B'11111110'
        movwf   trisb           ;pin RB0 je nastaven jako vystup
        bcf     RP0             ;stranka 0 pameti RAM
        bsf     LED             ;rozsviti LED
        end                     ;konec programu
```

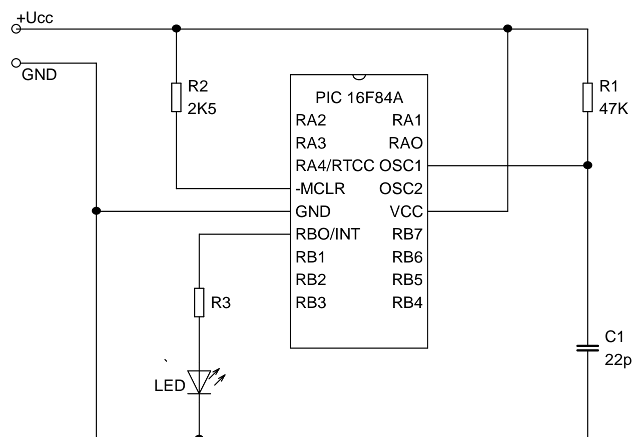
<table>
<tr><td colspan="3">

Programování jednočipů PIC

ÚLOHA č.1   TÉMA : Nastavení I/O - LED

</td></tr>
<tr><td>Číslo úlohy: 1.1.1</td><td>Programovací jazyk: Assembler</td><td>Zadání: žák</td></tr>
</table>

Zadání úlohy: Rozsviťte LED diodu připojenou na RB0. LED dioda je připojena na RB0 dle schématu (obr. 1).



Obr. 1 Schéma zapojení