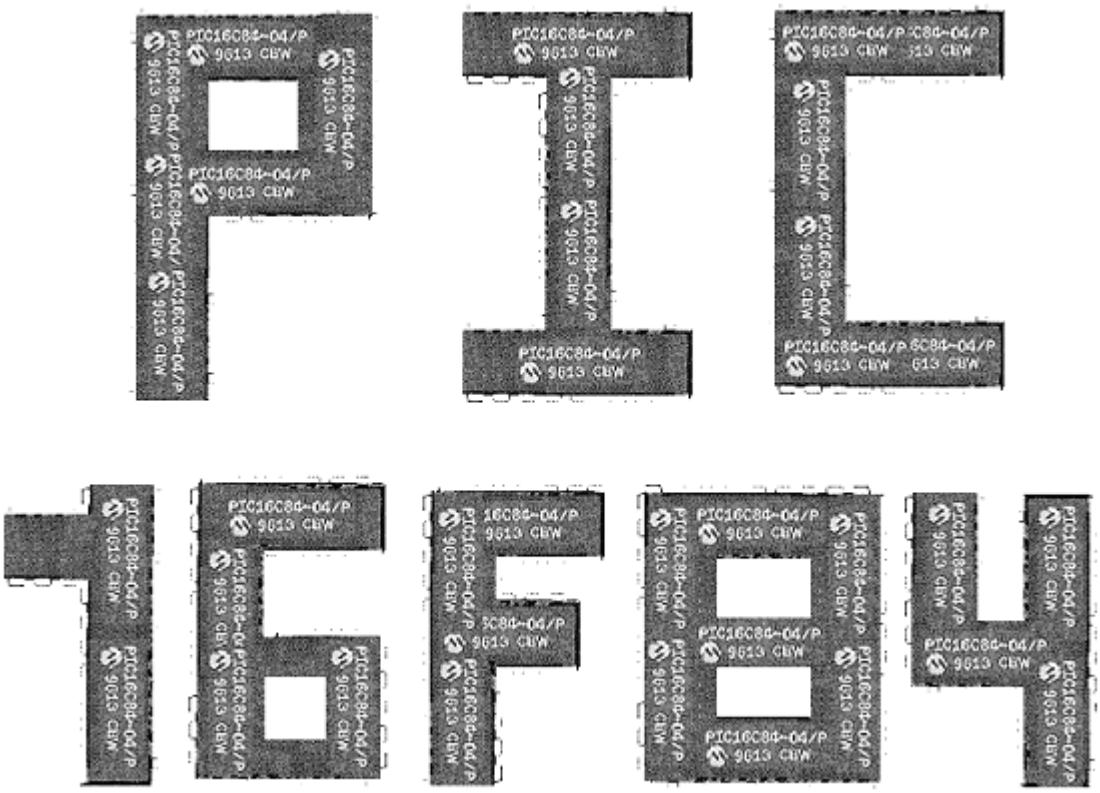
Mikroprocesorová technika.

Ing. Jan Hrdina



Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

Úvod

Mikropočítače jsou v dnešní době velmi důležité řídící jednotky, poněvadž se tak výrazně redukuje externí hardware(vnější zařízení), který je už v mikropočítačích implementován, a zároveň je zde velká univerzálnost použití. My se budeme konkrétně zabývat mikropočítačem PIC16F84 firmy MICROCHIP, který je v dnešní době dosti využívaný.

Popis mikropočítače

Procesor PIC16F84 je vyroben technologií **CMOS,** takže by se s ním mělo řádně zacházet, ale nebojte, snese toho hodně. Tento procesor má tzv. **Harvardskou architekturu,** což znamená, že má oddělenou programovou a datovou (RAM) paměť => výkon procesoru se tím zvyšuje, protože může najednou číst data i program v instrukčním cyklu a zároveň **velikost slova pro datovou a programovou paměť může být rozdílná.** Tento procesor je založen na **8-bitovém jádře typu RISC**, v překladu - **procesor s redukovanou instrukční sadou** => má pouze 35 základních instrukcí. Proto je tento procesor rychlý, ale aby byl opravdu velmi rychlý je zapotřebí splnit hned několik podmínek:

- použití jen **základních instrukcí** => čím složitější instrukce, tím je požadováno více strojových cyklů na jejich vykonání
- co **nejkratší čas na vykonání jedné instrukce** => většina instrukcí je vykonána během 1 strojového cyklu
- v tomto případě je nutné použít 14. bitové instrukce
- při načítání jedné instrukce se hned připravuje následující instrukce

Většina **instrukcí** trvá pouze **4 takty** oscilátoru, pak se při použití **4 MHz** krystalu rovná **1 strojový cyklus = 1µs.** Avšak některé instrukce trvají **2 strojové cykly**, protože procesor nevyužije připravenou instrukci a musí znovu načíst novou instrukci => jsou to **skokové instrukce** a není – li splněna podmínka obsažená v instrukci => **při nesplnění podmínky.** Dospěli jsme k názoru, že jádro procesoru je opravdu velmi rychlé. Abychom si dokázali udělat představu, jak pracuje procesor, podíváme se na blokové schéma **(Architektura PIC16F84),** protože je tam názorně ukázáno, jak je rozdělena **instrukce** na **dvě části** a do jakých bloků jdou tyto informace. **Instrukci**, která je tvořena **14. bity,** můžeme rozdělit tedy na **dvě části**. Jednu část tvoří **DATA – 8 bitů** (můžeme zadat číslo v rozsahu 00H až max. FFH) a druhá část instrukce je **OPERAČNÍ KÓD - 6-bitů.** Během 4 taktů oscilátoru se postupně načtou **DATA** a **KÓD instrukce a mezi tím se připravuje další instrukce, proto většina instrukcí trvá 1µs.**

Tento procesor má několik periferií a funkčních bloků, které jsou **níže podrobně popsány.** Periferiemi jsou **porty** neboli **brány** určené pro vstup a výstup dat a pro řízení. **PIC16F84** má bránu **A**, bránu **B**, časovač/čítač, **EEPROM** paměť. Má samozřejmě i jiné funkční bloky, mezi něž patří : **SLEEP** mód, zapnutí procesoru až po naběhnutí napájecího napětí **PWRTE,** ochrana kódu **CONFIG,** časovač zapnutí oscilátoru **OST,** sériové programování **ICSP,** Watchdog Timer **(WDT)** a jiné.

Procesor je schopen pracovat v rozsahu napájecího napětí již od **2V** do **6V** a proudový odběr je menší než **2mA** při použití **4Mhz krystalu.** S klesající frekvencí krystalu odběr proudu klesá. Zmínili jsme se zde o tzv. ochraně kódu (tj. slovo **CONFIG**), tato ochrana spočívá ve znemožnění přečtení zdrojového kódu z mikroprocesoru, ale taktéž pomocí slova **CONFIG** si můžeme určit např. jaký typ oscilátoru použijeme atd. Toto slovo si nyní podrobně popíšeme.

Konfigurační slovo – CONFIG

Procesor obsahuje tzv. konfigurační slovo, které lze měnit ve zdrojovém textu, nebo jej můžeme měnit v programátoru před vypálením zdrojového kódu do mikropočítače. Toto konfigurační slovo je 14-bitové a pomocí něj lze nastavit několik funkcí procesoru.

CP	CP	CP	CP	CP	CP	DP	CP	CP	CP	/PWRTE	WDTE	F0SC1	F0SC0
b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

bity 0,1 F0SC0: F0SC1

00 = **LP** oscilátor (do 200 Khz)
01 = **XT** oscilátor (do 4 Mhz)
10 = **HS** oscilátor (do 20 Mhz)
11 = **RC** oscilátor (RC článek)

bit 2 WDTE (Watchdog Timer)

0 = zakázán
1 = povolen

bit 3 /PWRTE (Power UP Timer) zapnutí procesoru až po naběhnutí napájecího napětí

0 = povol
1 = zakaž

bit 4:6 CP (Code Protection) ochrana programu

0 = ochrana programu zapnuta
1 = ochrana programu vypnuta

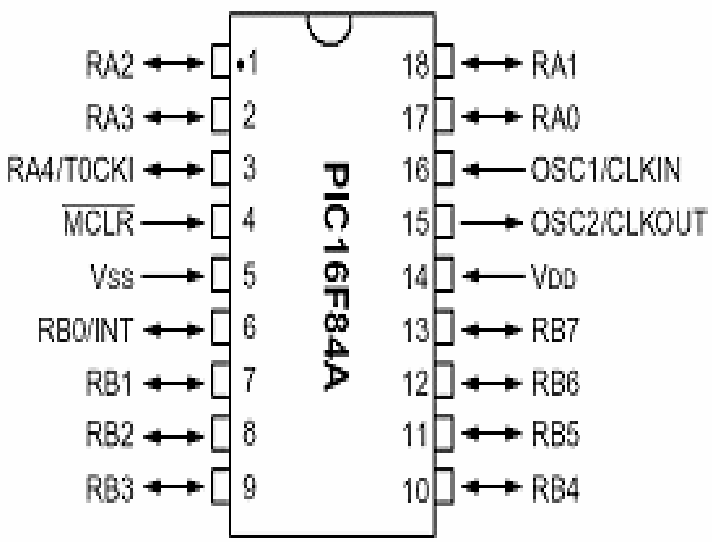
bit 7 DP (Data Protection) ochrana dat

0 = ochrana dat je zapnuta
1 = ochrana dat je vypnuta

bity 8:13 CP (Code Protection) ochrana programu

0 = ochrana programu zapnuta
1 = ochrana programu vypnuta

Popis vývodů procesoru

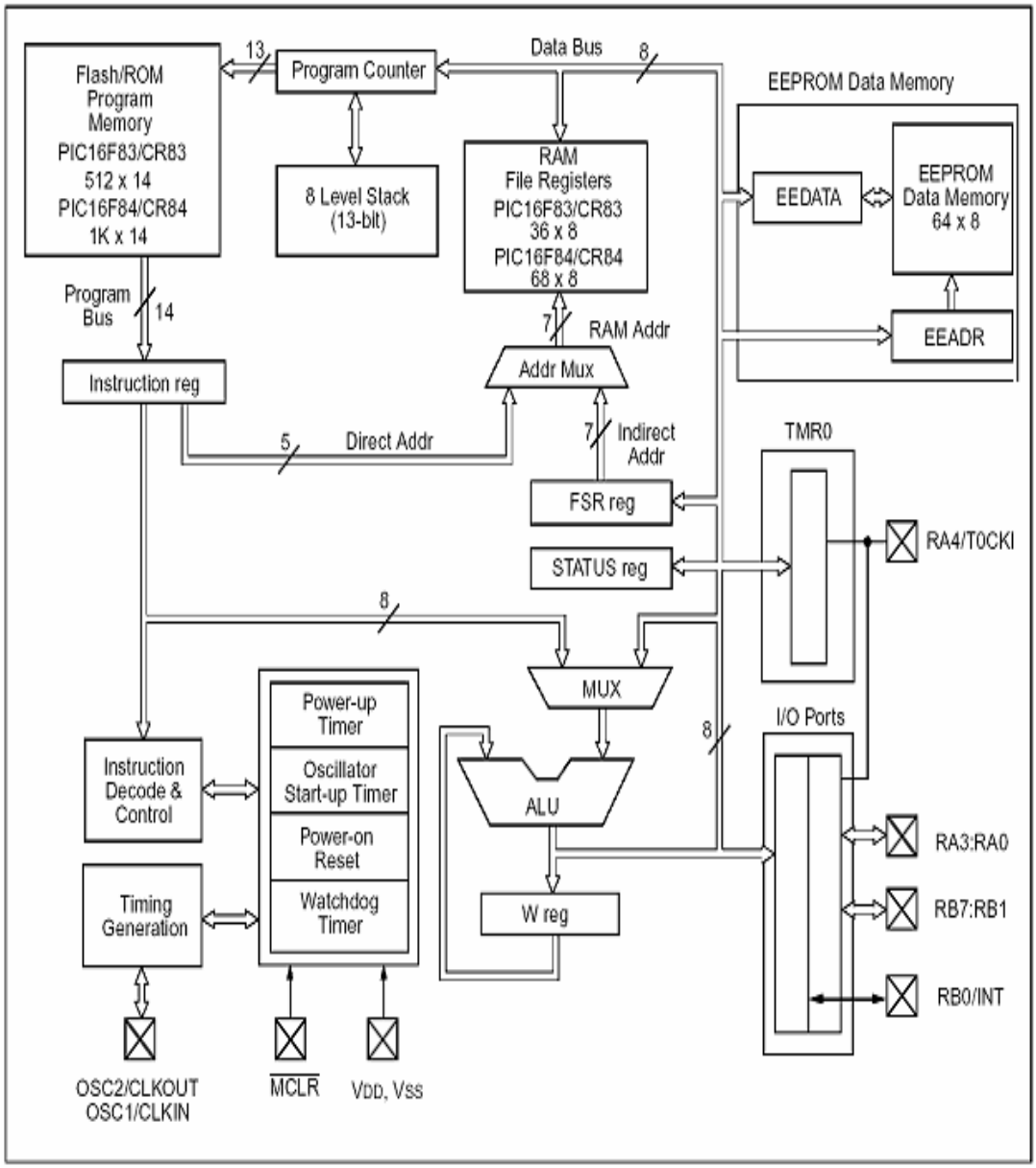


vývod	pin	typ I/O/P	provedení	popis
				PORTA je obousměrný vstupně/výstupní port
RA0	17	I/O	TTL	
RA1	18	I/O	TTL	
RA2	1	I/O	TTL	
RA3	2	I/O	TTL	
RA4/TOCKI	3	I/O	ST	Může být jako zdroj CLK signálu pro TMR1. Jako výstupní má otevřený kolektor!!!
MCLR/V_{pp}	4	I/P	ST	**RESET/**vstup programovacího napětí. Tento vývod je aktivní v nule, kdy provádí RESET obvodu.
V_{SS}	5	P	-	zem
				PORTB je obousměrný vstupně/výstupní port. PORTB může mít programově připojen slabý vnitřní pull-up odpor na všech vstupech.
RB0/INT	6	I/O	TTL/ST	může být vybrán jako zdroj vnějšího přerušení externí přerušení
RB1	7	I/O	TTL	
RB2	8	I/O	TTL	
RB3	9	I/O	TTL	přerušení při změně vstupu
RB4	10	I/O	TTL	přerušení při změně vstupu
RB5	11	I/O	TTL	přerušení při změně vstupu/ při programování
RB6	12	I/O	TTL/ST	(CLOCK)
RB7	13	I/O	TTL/ST	přerušení při změně vstupu/ při programování DATA
V_{DD}	14	P	-	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

I = Input (vstup) O = Output (výstup) I/O = Input/Output (vstup/výstup) P = Power
- = nevyužito TTL = TTL input ST = Schmitt Trigger input (na vstupu je Schmittův klopný obvod)

Architektura procesoru

Zde si jen řekneme, že aritmetickologická jednotka (označována **ALU**) umožňuje sčítání, odčítání, logické operace či snad jen posouvat obsah registru. Aritmetické operace mají dva operandy, z nichž jeden je vždy v pracovním registru (**W**) a druhý operand je registr v paměti nebo námi zadaná konstanta. Pracovní registr je určen pro práci s **ALU**. Veškeré registry tj. registry pro ovládání periferií nebo volně dostupné registry jsou **8-bitové.**
U typů **RISC** je taktéž typický znak v tom, že veškeré matematické a přesunové operace se provádějí přes pracovní (working) registr **W .**



Organizace paměti

Programová paměť:

Paměť programu má velikosti **1KB** tj. **0000H – 3FFH po 14-bitech.** Abychom mohli zjistit adresu v programu, slouží nám zde **13-bitový Program counter (PC),** u něhož je obsah zvyšován za každou instrukcí. Po **RESETU** nebo při zapnutí napájecího napětí procesoru, začíná procesor vždy na adrese **0000H. Vektor přerušení** (tj. **adresa pro obsluhu přerušení)** je **0004H** a je nutné tuto adresu **(při povolení přerušení)** akceptovat. V případě, že se budeme chtít dostat nad adresu **3FFH,** nastane automatický skok na začátek tj.**0000H.** Procesory **PIC** vyšší řady mají až **8KB** programové paměti.

Datová paměť – RAM:

Datová paměť je rozdělena u **PIC16F84** do dvou bank, označovaných jako **BANKA 0** a **BANKA1.** Každá banka má **128 bytů,** avšak celý prostor v jednotlivých bankách není implementován. Některé registry jsou obsaženy v obou bankách.

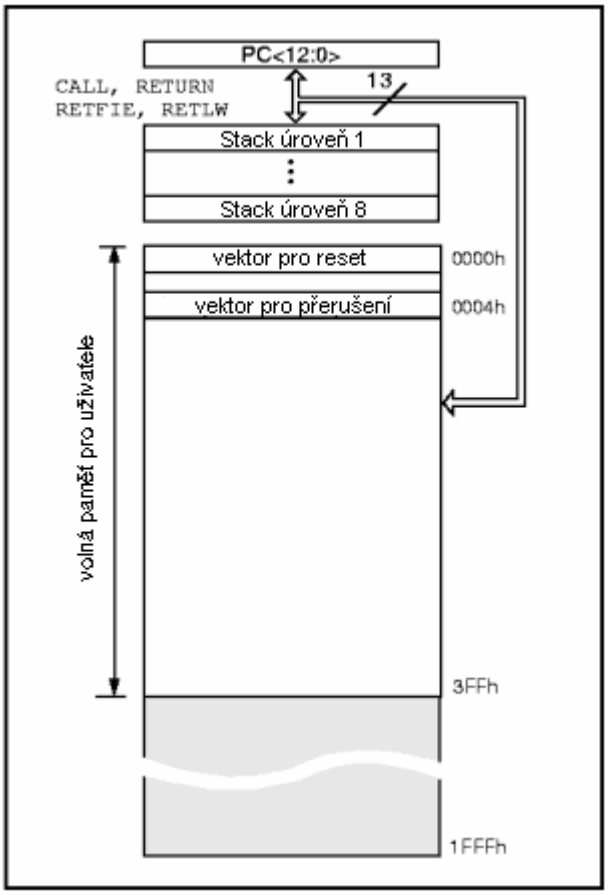
Od adresy: **(00H – 0BH) a (80H – 8CH)** jsou přístupné ovládací registry.
(0CH – 4FH) a (8CH – CFH) jsou přístupné registry pro volné použití
(50H – 7EH) a (D0H – FFH) není implementováno

Adresa	BANKA 0	BANKA 1	Adresa
00h	INDF *	INDF *	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 (RAM)**	**Mapováno do stránky 0**	8Ch až CFh
50h až 7Eh	**neimplementováno (čte jako 0)**	**neimplementováno (čte jako 0)**	D0h až FFh

<u>Zásobník pro návratové adresy:</u> (stack pointer)

Procesor má **8-úrovňový zásobník s šířkou 13-bitů.** Tento zásobník není možné adresovat, tedy nemůžeme z něj číst ani do něj zapisovat. Tento zásobník slouží čistě jen pro účely procesoru. V případě, kdy budeme volat podprogram pomocí instrukce **CALL,** je do zásobníku uložena **13-bitová adresa** tj. hodnota převzatá od programového čítače **(PC).** Obsah zásobníku je vybrán pouze při návratu z podprogramu tj. instrukce **RETURN, RETLW** nebo **RETFIE.** Proto se zásobníku říká **zásobník návratových adres.**

O**bslužné podprogramy musíme vždy ukončit, jinak dojde ke kolizi zásobníku !!!**



PERIFERIE PROCESORU PIC16F84
VSTUPNÍ / VÝSTUPNÍ BRÁNY = PORTY

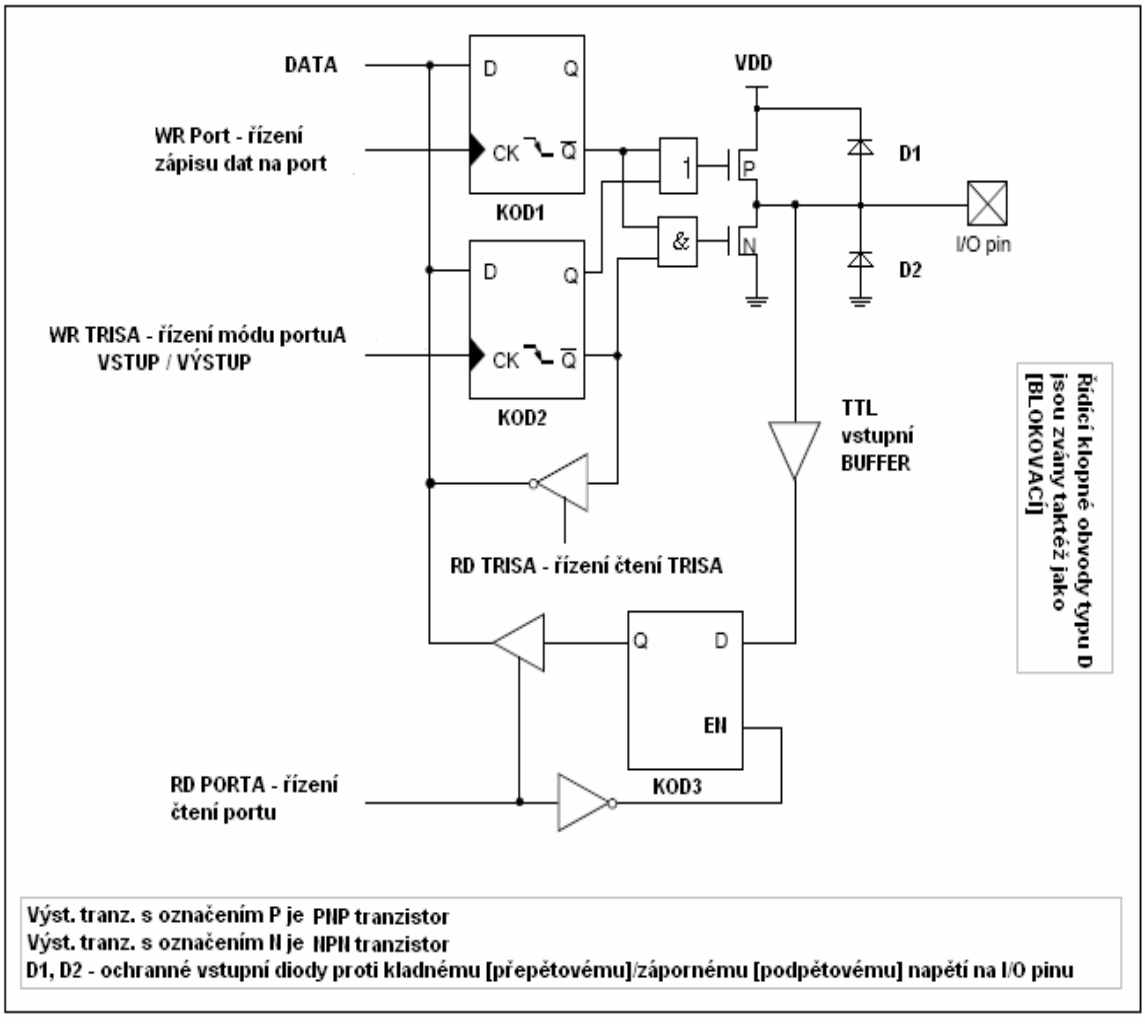
Teorie PORTA a PORTB :

Procesor PIC16F84 má dvě vstupní/výstupní **(duplexní)** brány. Každá z bran má svého zástupce v registrech. Zde jsou to registry **PORTA** a **PORTB**. Chceme – li měnit log.úroveň některého z pinu či pinů bran, lze to pomocí právě těchto registrů **PORTA PORTB**. Jednotlivé **piny = bity** portů mají taktéž označení. Samozřejmě si může každý označit piny podle sebe, ale je dobré používat následující označení, protože je převzato od výrobce. Piny **PORTA** značíme **RA** a **PORTB** značíme **RB. PORTA** je **5 – bitová** brána u níž je pin **RA4/INT využíván i jako čítač externího zdroje signálu. PORTB** je **8 – bitová** brána u níž jsou piny **(RB4 – RB7) využívány i jako externí interrupty a RB0/INT jako externí přerušení.** Už od samotného názvu (vstupní/výstupní) plyne, že brány můžeme používat jako vstupy nebo naopak jako výstupy. Nastavení bran **(tj. určíme který z pinů dané brány bude vstup a který výstup)** se provádí pomocí dvou registrů označovaných názvem **TRIS.** Registr **TRISA** slouží pro nastavení **PORTA**. Registr **TRISB** slouží pro nastavení **PORTB**. Piny, které chceme použít jako vstupy, musíme nastavit v registru **TRIS do úrovně log.1.** A naopak, při použití pinů jako výstupy, musíme zapsat na požadované bitové adresy v registru **TRIS úrovně log.0.** V případě, že nastavíme nějaký pin nějaké brány jako **vstup**, tak už nelze měnit log.úroveň (pomocí registrů **PORT**) na pinu, protože to blokuje **klopný obvod typu D (označení KOD2).** V opačném případě lze měnit log.úroveň pinu pomocí registrů (**PORT**), protože **KOD2** již neblokuje **KOD1.** Brány jsou konstruovány pro poměrně velké proudové zatížení, **max. 25mA do pinu a max. 20mA z pinu.** Na tyto hodnoty se musí dávat pozor, protože v případě jejich překročení nastane stav, kdy daný pin **již není možné ovládat!** Avšak dané max. hodnoty jsou dostatečné např. k rozsvěcení LED diod. Diody s označením **(D1 , D2)** jsou ochranné diody, které slouží k ochraně pinu. Dioda **D1** slouží jako ochrana proti napětí, které je **> 5V.** Dioda **D2** slouží jako ochrana proti napětí, které je **< 0V => tj. proti zápornému napětí.** V případě použití některé z **bran** jako vstupní (nebo jen pinu), použijeme většinou rezistor a tlačítko v paralelní kombinaci s kondenzátorem. V případě použití jako výstup použijeme většinou nějaký integrovaný budič, tranzistor nebo něco podobného. Záleží to především na typu a velikosti zátěže. Nyní se budeme zabývat přímou specifikací **brány A = PORTA a brány B = PORTB.**

PORT A

Z blokového schématu je vidět, jak piny **(RA0 – RA3)** fungují. Nejlepší bude, když si podrobně popíšeme, jakým způsobem vlastně ovládání pinů **(RA0 – RA3)** probíhá. Dále již budeme říkat jen **pin** – myslíme tím některý z pinů **(RA0 –RA3). My budeme brát třeba pin RA0**. Než si řekneme, jak pin **RA0** funguje při **zápisu** a **čtení**, je třeba si uvědomit, že řídící elektronika pinu je tvořena **třístavovými obvody, číslicovými obvody a výstupní část je tvořena tranzistory => hybridní obvod.** Máme zde dva klopné obvody **KOD1** a **KOD2**. **KOD1** slouží pro zápis dat (tj. log.úroveň) => využívá jej registr **PORTA. KOD2** slouží k nastavení pinu, používá jej registr **TRISA => vstup/výstup**. Zároveň můžeme **KOD2** označit jako **blokovací**, protože v případě nastavení **KOD2 (nastavení jako vstup)** je automaticky **vyřazen KOD1**.Při nulování **KOD2 (nastavení jako výstup)** je **KOD1 odblokován** a my můžeme pomocí registru **PORTA** zapisovat data na tento port. Čtení výstupu (pinu) se provádí pomocí **KOD3**. K tomu, aby mohlo čtení vůbec proběhnout, potřebujeme další obvody => třístavové obvody. Tyto obvody slouží především jako spínací prvky, které odpojí výstup z **KOD1** tak, aby neovlivňoval vstupní signál na tomtéž bitu.

Blokové schéma pinů RA0 - RA3



Nastavení pinu (RA0) jako VSTUP

Ve zdrojovém textu to vypadá takto :
movlw B‘00000001‘ ; určení, které piny mají být vstupní a které zas výstupní => jediný RA0 je vstup
movwf TRISA ;uložení výše dané hodnoty do TRISA

Jak to funguje uvnitř v mikropočítači – dívejme se na blokové schéma pinů (RA0 – RA3):
Bohatě si vystačíme s instrukcí **(movwf TRISA)**. Tato instrukce způsobí, že se přesunou data z pracovního registru (označován **W**) do registru **TRISA.** Nás zajímá jen **nultý bit (RA0)**, protože jen on je nastaven jako vstup. Takže, když použijeme tuto instrukci, tak se stane následující:
Na datovou sběrnici (**DATA**) jsou poslána požadovaná data**=> pin RA1 – RA4 = log.0, pin RA0 = log.1** Vygeneruje se řídící impuls na (**WR TRISA**) => tímto jsou data přenesena na výstup **KOD2**. Výstup **Q(non)** je tímto v **log.0** a **Q v log.1 KOD2.** Jelikož je hradlo **AND** spojeno s výstupem **Q(non) KOD2**, tak na výstupu hradla **AND** bude již nyní **pořád log.0.** Hradlo **OR** je zase spojeno s výstupem **Q KOD2**. Z tohoto všeho nám celkově vyplývá, že **tranzistor (N)** a **tranzistor (P)** budou **stále zavřeny=>** na **výstupu (pinu)** je **log.0 => záleží pak na programátorovi, zda v tomto režimu bude přivádět na tento pin Vdd přes rezistor 47kΩ .** Všimněme si, že **KOD1** slouží pro změnu log.úrovně na výstupu. **Ovšem za předpokladu**, **že jsme pin nastavili jako výstup.** V tomto případě jsme pin nastavili jako **vstup**, takže **nás KOD1 nezajímá.**

Nastavení pinu (RA0) jako VÝSTUP

Ve zdrojovém textu to vypadá takto :
movlw B‘00000000‘ ; určení, které piny mají být vstupní a které zas výstupní => RA0 je výstupní
movwf TRISA ;uložení výše dané hodnoty do TRISA

Jak to funguje uvnitř v mikropočítači – dívejme se na blokové schéma pinů (RA0 – RA3):
Na datovou sběrnici (**DATA**) jsou poslána požadovaná data**=> pin RA0 – RA4 = log.0 => all output.** Vygeneruje se řídící impuls na (**WR TRISA**) => tímto jsou data přenesena na výstup **KOD2**. Výstup **Q(non)** je tímto v **log.1** a **Q v log.0 KOD2**. Jelikož je hradlo **AND spojeno** s výstupem **Q(non) KOD2** a výstupem **Q(non) KOD1**, tak na výstupu hradla **AND** bude **log.1** v případě, že **Q(non) KOD1** bude **log.1 => KOD1 rozhoduje nyní o tom, jaký tranzistor bude otevřený/zavřený tzn. KOD1 řídí obě hradla**. Hradlo **OR** bude řídit tranzistor **(P)** a hradlo **AND** bude řídit tranzistor **(N)**. Z tohoto všeho nám vyplývá jedna věc => přichází na řadu řízení pomocí registru **PORTA**, který řídí **KOD1.**

Nastavení výstupu **RA0 = log.1, musí** být splněna předešlá podmínka tj. pin **RA0 je nastaven jako výstup**
Při nastavení **pinu** (tj.log1) se nám tato **log.1** přesune pomocí registru **PORTA** na datovou sběrnici. Následně se vygeneruje řídící impuls na **(WR PORTA)** => tímto jsou data přenesena na výstup **KOD1**. Hradlo **OR** si však bere log.úroveň pro vyhodnocení z výstupu **Q(non) KOD1** => sčítá tedy **(0 + 0) = log.0 => tranzistor (P) je otevřen** a **tranzistor** (**N**) **je zavřen**, protože hradlo **AND** má na výstupu **log.0.**

Ve zdrojovém textu to vypadá takto (je zde uvedeno i nastavení pinů jako výstupy):
movlw B‘00000000‘ ; všechny piny budou výstupy
movwf TRISA ; uložení hodnoty do TRISA
movlw B‘11111111‘ ; zde určujeme, jaká data se mají přenést na celou bránu => všechny piny budou v **log.1**

movwf PORTA ; zapsání dat na **PORTA.**

Nastavení výstupu **RA0 = log.0**, **musí** být splněna předešlá podmínka tj. pin **RA0 je nastaven jako výstup**

Při nastavení **pinu** (tj. log.0) se nám tato **log.0** přesune pomocí registru **PORTA** na datovou sběrnici. Následně se
vygeneruje řídící impuls na **(WR PORTA)** => tímto jsou data přenesena na výstup **KOD1**. Hradlo **OR** si však bere log.úroveň pro vyhodnocení z výstupu **Q(non) KOD1=>** sčítá tedy **(0 + 1) = log.1 => tranzistor (P) je zavřen** a **tranzistor (N) je otevřen**, protože hradlo **AND** má na výstupu **log.1.**

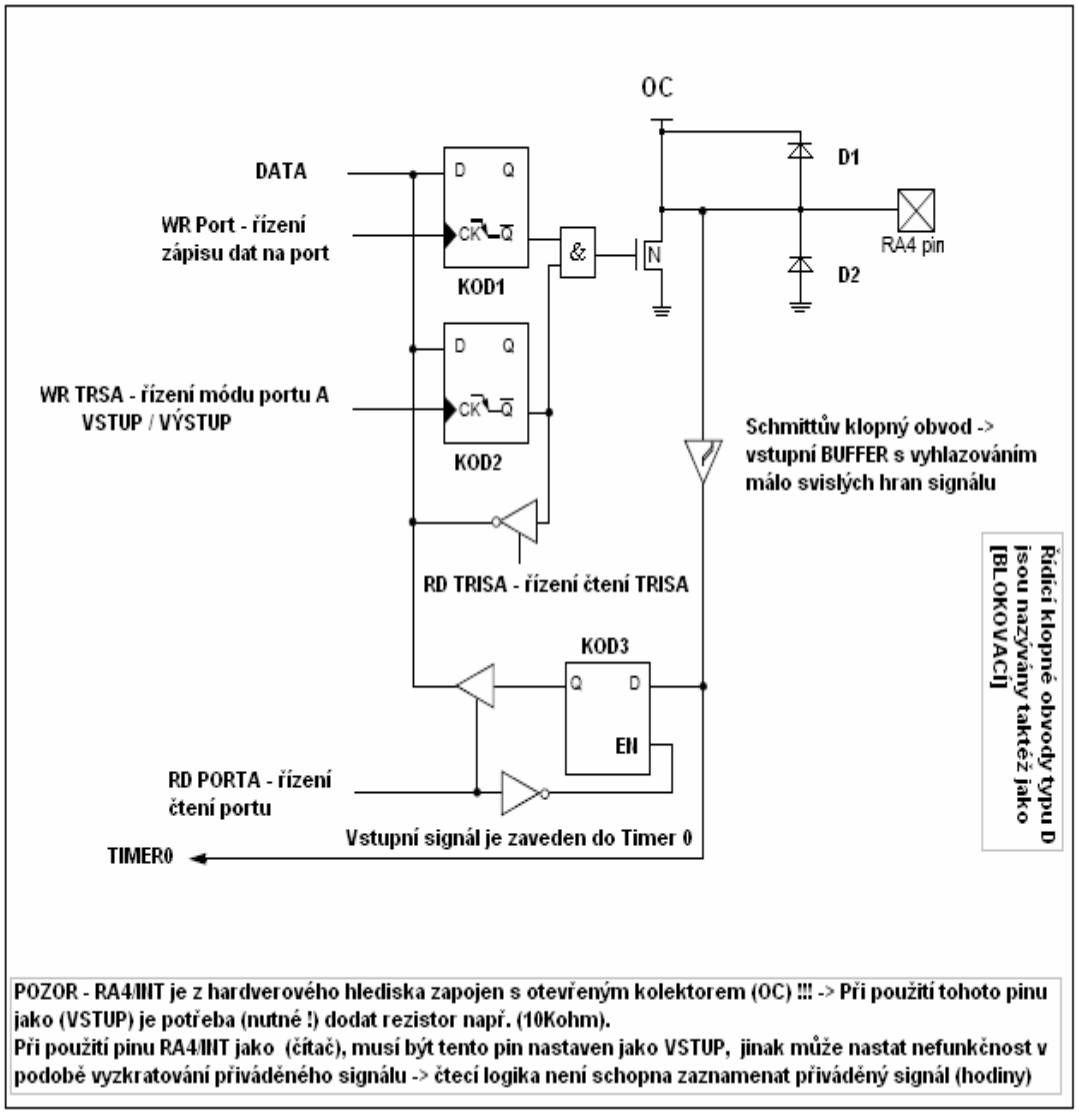
Ve zdrojovém textu to vypadá takto (je zde uvedeno i nastavení pinů jako výstupy): **movlw B‘00000000‘** ; všechny piny budou výstupy **movwf TRISA** ; uložení hodnoty do TRISA **movlw B‘0000000‘** ; zde určujeme, jaká data se mají přenést na celou bránu => všechny piny budou v **log.0** **movwf PORTA** ; zapsání dat na **PORTA.**

Jak funguje čtení pinu RA0:

Ve zdrojovém textu to vypadá takto : **movf porta,w** ; tato instrukce přesune data z pinů, tedy porta do **pracovního registru = akumulátoru.** ; znamená to tedy, že jsme přečetli stav jednotlivých pinů = bitů **PORTA**

Jak to funguje uvnitř v µpočítači – dívejme se na blokové schéma pinů (RA0 – RA3):
Při čtení je nejprve procesorem vygenerován impuls na **(RD PORTA).** Tento impuls způsobí, že data na **výstupu pinu RA0 = vstupu KOD3** jsou přesunuta na jeho výstup. Taktéž dojde k otevření třístavového obvodu =>**teprve po otevření KOD3 a třístavového obvodu jsou data přesunuta na datovou sběrnici.** Proč zde používáme třístavový obvod? Docházelo by k ovlivňování čteného signálu na datové sběrnici => **nefunkčnost datové sběrnice.**

Blokové schéma pinu RA4/INT - ČASOVAČ/ČÍTAČ



POZOR - RA4/INT je z hardverového hlediska zapojen s otevřeným kolektorem (OC) !!! -> Při použití tohoto pinu jako (VSTUP) je potřeba (nutné !) dodat rezistor např. (10Kohm).
Při použití pinu RA4/INT jako (čítač), musí být tento pin nastaven jako VSTUP, jinak může nastat nefunkčnost v podobě vyzkratování přiváděného signálu -> čtecí logika není schopna zaznamenat přiváděný signál (hodiny)

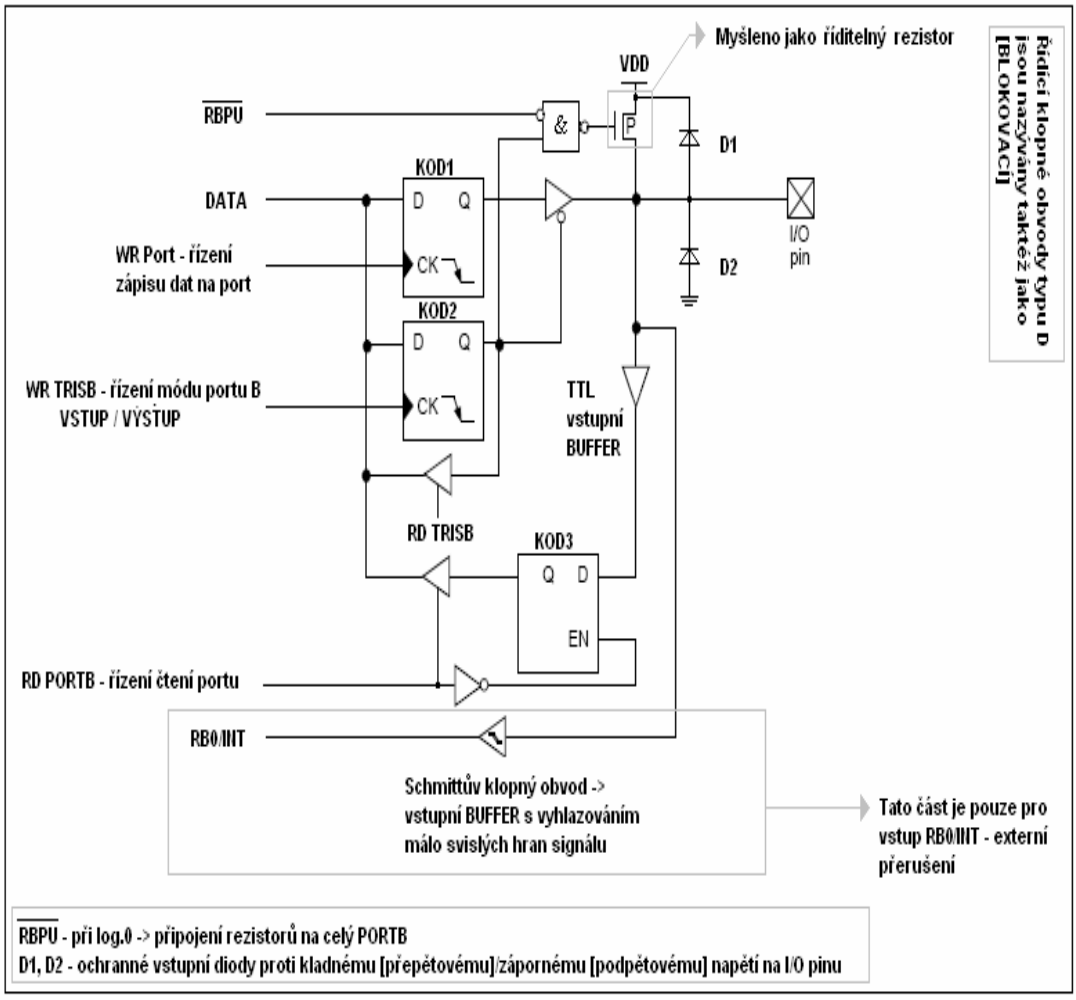
Z blokového schéma **RA4/INT** jsou vidět odlišnosti od pinů **(RA0 – RA3).** Chybí zde hradlo **OR** a tranzistor **(P).** Místo členu **(TTL vstupní BUFFER)** je zde **(Schmittův KO) =>** kdybychom chtěli pin **RA4/INT použít v režimu ČÍTAČ.** Tento pin je odlišný i tím, že je zapojen jako otevřený kolektor **OC, pro univerzální použití je nejlepší připojit na tento pin Vdd (napájecí napětí) přes rezistor např. 47KΩ.** Čtení je na tomto pinu stejné jako na pinech **(RA0 – RA3),** avšak **zápis** je trochu odlišný. Nebudeme si jej dopodrobna vysvětlovat, protože je podobný jako u pinů **(RA0 – RA3).** Odlišnost spočívá v tom, že **KOD1** ovládá **jen** hradlo **AND,** protože zde máme jen jeden tranzistor. Při nastavení pinu **RA4/INT** jako **vstup** je hradlo **AND** blokováno **KOD2.** Při nastavení jako **výstup** je hradlo **AND** odblokováno=> vidíme, že **KOD2** má stejnou funkci jako u pinů **(RA0 – RA3).**

Jelikož je čtení bran (PORTA, PORTB) podobné, nebude již dále popisováno – jen v nutných případech. Taktéž čtení registrů (TRISA, TRISB) je podobné, avšak s tím rozdílem (myslíme tím oproti čtení PORT), že se data čtou z výstupu Q(non) KOD2 ! V případě, že pin RA4/INT využijeme jako ČÍTAČ, musí být tento pin nastaven jako VSTUP !!!

PORT B

Z blokového schématu je vidět, jak piny **(RB0 – RB3)** fungují. Je zde podobná struktura jako u **PORTA**. Liší se ve výstupní části, která je tvořena **třístavovým obvodem, který nahrazuje ony dříve zmíněné tranzistory P a N u PORTA** , tzv.**řiditelným rezistorem a napojením pinu RB0/INT na přerušovací systém.** Řiditelný rezistor je ve skutečnosti zastoupen **tranzistorem P**, který je ovládán bitem **RBPU** v reg. **OPTION.** Jak jste si jistě všimli, výrazem „**řízení rezistoru“** myslíme **otevírání** či **zavírání tranzistoru P.** Záleží čistě na programátorovi, jakou aplikaci navrhne, zda daný **tranzistor P** využije či ne. Například při využití **pinů (RB0– RB3)** jako vstup, využijeme taktéž i **tranzistor P.** Nejlepší bude, když si podrobně popíšeme, jakým způsobem vlastně ovládání pinů **(RB0 – RB3)** probíhá. Jako v případě **PORTA** si i zde vybereme například **pin RB0/INT.**

Blokové schéma pinů RB0 - RB3



Nastavení pinu (RB0/INT) jako VSTUP

Ve zdrojovém textu to vypadá takto :

```
movlw     B‘00000001‘    ; určení, které piny mají být vstupní a které zas výstupní => jediný RB0/INT je vstup
movwf     TRISB          ;uložení výše dané hodnoty do TRISB
```

Jak to funguje uvnitř v mikropočítači – podívejme se na blokové schéma pinů **(RB0 – RB3)**:

Bohatě si vystačíme s instrukcí **(movwf TRISB)**. Tato instrukce způsobí, že se přesunou data z pracovního registru (označován **W**) do registru **TRISB.** Nás zajímá jen **nultý bit (RB0)**, protože jen on je nastaven jako vstup. Takže, když použijeme tuto instrukci, tak se stane následující:
Na datovou sběrnici (**DATA**) jsou poslána požadovaná data**=> pin RB1 – RB7 = log.0, pin RB0 = log.1** Vygeneruje se řídící impuls na (**WR TRISB**) => tímto jsou data přenesena na výstup **KOD2** tzn. výstup **Q KOD2**
je nyní nastaven na log.1 => tato log.úroveň znamená **zavření třístavového obvodu** tzn., že výstup **Q KOD1 je odpojen od výstupu pinu.** Při nastavení pinu **RB0/INT** jako **vstup** je **KOD1 jakoby blokováno KOD2.**

Nastavení pinu (RB0/INT) jako VÝSTUP

Ve zdrojovém textu to vypadá takto :
movlw B‘00000000‘ ; určení, které piny mají být vstupní a které zas výstupní => RB0 je výstupní
movwf TRISB ;uložení výše dané hodnoty do TRISB

Jak to funguje uvnitř v mikropočítači – podívejme se na blokové schéma pinů **(RB0 – RB3):**

Na datovou sběrnici (**DATA**) jsou poslána požadovaná data**=> pin RB0 – RB7 = log.0 => all output**. Vygeneruje se řídící impuls na (**WR TRISB**) => tímto jsou data přenesena na výstup **KOD2** tzn. výstup **Q KOD2** je nyní nastaven na log.0 => tato log.úroveň znamená **otevření třístavového obvodu** tzn., že výstup **Q KOD1 je připojen na výstup pinu.** Při nastavení pinu **RB0/INT** jako **výstup** je již možné zapisovat na **výstup pinu** data tj. přichází na řadu řízení výstupu pinu pomocí **KOD1.**

Nastavení výstupu **RB0 = log.1, musí** být splněna předešlá podmínka tj. pin **RB0 je nastaven jako výstup**
Při nastavení **pinu** (tj.log1) se nám tato **log.1** přesune pomocí registru **PORTB** na datovou sběrnici. Následně se vygeneruje řídící impuls na **(WR PORTB)** => tímto jsou data přenesena na výstup **KOD1**. Jelikož je třístavový obvod otevřen => data se dostanou na výstup pinu **RB0/INT.**

Ve zdrojovém textu to vypadá takto (je zde uvedeno i nastavení pinů jako výstupy):
movlw B‘00000000‘ ; všechny piny budou výstupy
movwf TRISB ; uložení hodnoty do TRISB
movlw B‘11111111‘ ; zde určujeme, jaká data se mají přenést na celou bránu -> všechny piny budou v **log.1**
movwf PORTB ; zapsání dat na **PORTB.**

Nastavení výstupu **RB0 = log.0, musí** být splněna předešlá podmínka tj. pin **RB0 je nastaven jako výstup**
Při nastavení **pinu** (tj.log0) se nám tato **log.0** přesune pomocí registru **PORTB** na datovou sběrnici. Následně se vygeneruje řídící impuls na **(WR PORTB)** => tímto jsou data přenesena na výstup **KOD1**. Jelikož je třístavový obvod otevřen => data se dostanou na výstup pinu **RB0/INT.**

Ve zdrojovém textu to vypadá takto (je zde uvedeno i nastavení pinů jako výstupy):

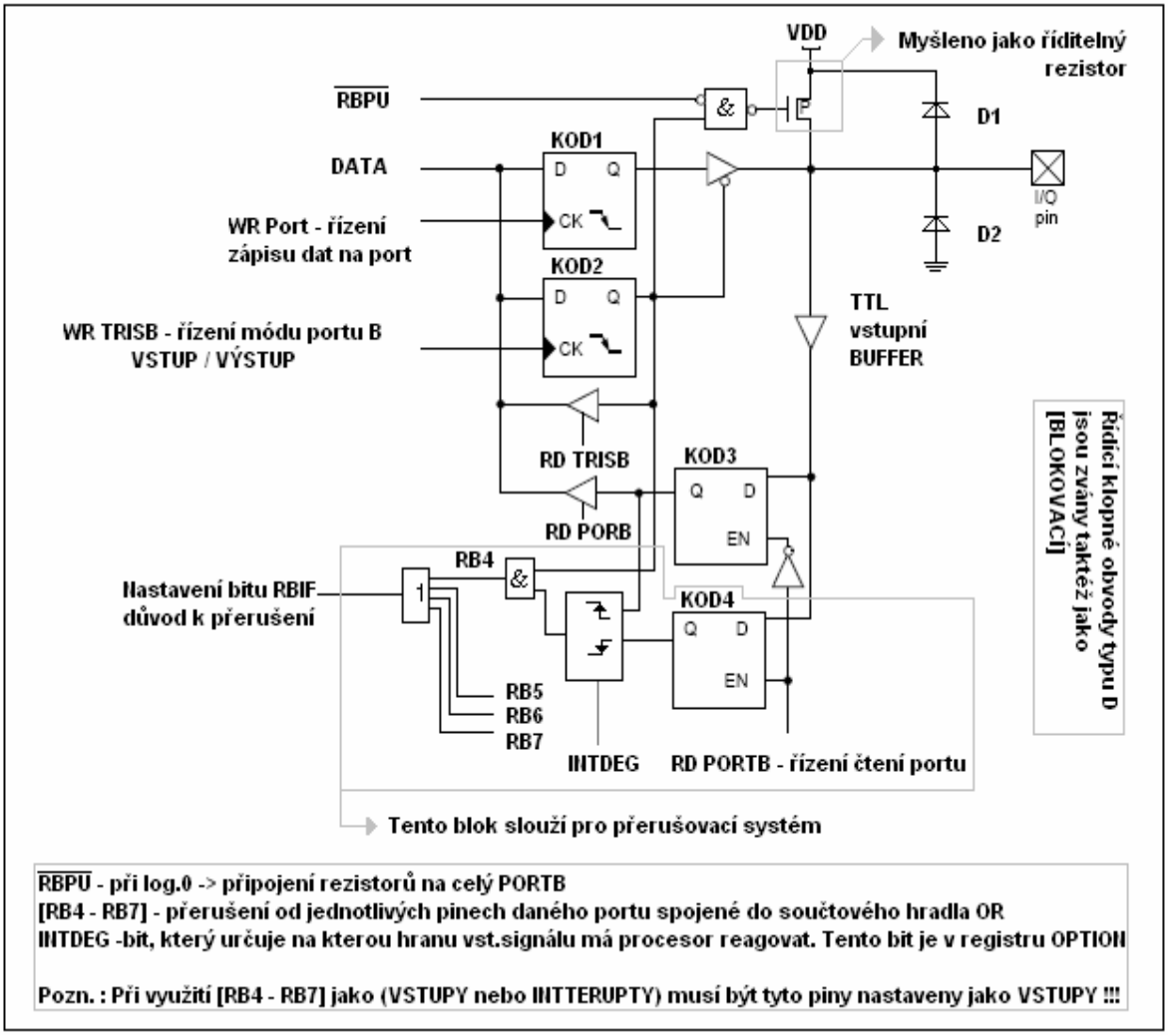
```
movlw     B‘00000000‘    ; všechny piny budou výstupy
movwf     TRISB          ; uložení hodnoty do TRISB
movlw     B‘00000000‘    ; zde určujeme, jaká data se mají přenést na celou bránu => všechny piny budou v log.0
movwf     PORTB          ; zapsání dat na PORTB.
```

Připojení rezistorů na PORTB - platí pro celou bránu
Abychom mohli připojit na námi daný pin (piny) říditelný rezistor, musí být splněny dvě podmínky.

1. nulování **bitu RBPU** v registru **OPTION**
2. příslušný pin (piny) musí být nastaven jako **VSTUP**
Jedině po splnění těchto dvou podmínek je na výstupu pinu/ pinů připojeno **VDD** přes tento říditelný rezistor.

Nastavení RB0/INT jako externí přerušení :
Všimněme si, že piny brány **PORTB** jsou označovány **RB**, avšak u **pinů (RB0 – RB3)** je jediný pin **RB0** označován i jako **RB0/INT.** Označení RB0/**INT** znamená, že tento pin umí pracovat i v jiném módu, než – li jen jako **VSTUP** či **VÝSTUP.** Jak jistě tušíte, zkratka **/INT** znamená **INTTERUPTS** což znamená **přerušení,** proto se tomuto **pinu RB0** říká taktéž **zdroj EXTERNÍHO PŘERUŠENÍ.** Budeme – li tedy pin **RB0** využívat jako zdroj externího přerušení, musí být pin nastaven jako **VSTUP a taktéž** se musíme povolit přerušení tohoto pinu, tj. nastavit bit **INTE** v registru **INTCON.** Všimněme si, že v případě využití **RB0** jako **ext.přerušení** se signál z pinu odvádí **nikoli KOD3** (tj. jako při čtení **PORTB**), ale pomocí tzv. **Schmittova klopného obvodu.** Tento Schittův klopný obvod slouží pro úpravu externího signálu, jenž má málo strmé hrany. Tento obvod tedy minimalizuje chybové čtení na pinu v tomto nastavení.

Blokové schéma pinů RB4 - RB7



Nyní se budeme zabývat strukturou pinů **(RB4 – RB7).** Vidíme, že **zápis** či snad **čtení** z pinů je naprosto stejné jako u pinů **(RB0 – RB3),** proto je zbytečné se jím zabývat. **Všechny** tyto piny tj.**(RB4 – RB7)** mohou být využity jako **zdroj pro přerušení.** Chceme-li je tedy využít podobně jako pin **RB0/INT,** musíme nastavit příslušné piny (to jsou ty, které chceme využít) jako **VSTUPY,** a taktéž povolit přerušení od **(RB4 – RB7)** tj. nastavit bit **RBIE** v registru **INTCON.** Při splnění těchto podmínek (jako u **RB0/INT**) jsou piny využity jako zdroje přerušení.

(RB4 – RB7) využity jako zdroje pro přerušení:

V případě, že **piny (RB4 – RB7)** využijeme jako zdroje přerušení, musíme splnit dříve stanovené podmínky. V případě, že tyto podmínky splníme, bude nás zajímat **blok pro přerušovací systém**. Tento **blok** je samozřejmě zakreslen do blok.schéma **(RB4 – RB7)** a my si nyní jen řekneme, že onen blok slouží k podávání informace o stavu log.úrovní na daném pinu dalším řídícím blokům. Tyto další řídící bloky jsou součástí celého přerušovacího systému a budeme o něm mluvit později. V tomto případě nás zajímá pouze **blok pro přerušovací systém.** Podívejme se na blokové schéma **(RB4 – RB7).**

Jak funguje blok pro přerušovací systém :

V tomto případě nás zajímá **KOD4,** protože na jeho vstupu (clock) je permanentně přivedena log.1 (pro zjednodušení výkladu) tzn. , že **KOD4** neustále přesouvá log.úroveň ze svého vstupu na svůj výstup. Tím podává data z výstupu pinu na vstup **rozhodovacího členu** tj. ten, který je ovládán bitem **INDF.** Tento rozhodovací člen má na výstupu **log.1** v případě, že je předem daná podmínka splněna. Tato podmínka je vlastně nastavení na jakou hranu vstupního signálu má tento člen reagovat **(tj. vzestupná nebo sestupná hrana signálu)** a samozřejmě **její splnění !** Hradlo **AND** má jeden vstup připojeno na výstup rozhodovacího členu a druhý vstup je připojen na výstup **Q KOD2.** Na výstupu hradla **AND** je tedy **log.1** pouze v případě, kdy je pin nastaven jako **vstup a musí** být splněna podmínka rozhodovacího členu – obě podmínky byly již zmíněny. Výstup hradla **AND** je spojen do čtyř – vstupého hradla **OR.** Je – li na výstupu hradla **AND log.1 =>** tato informace je taktéž kopírována na výstup hradla **OR**, čímž **nastává nastavení tzv. příznakového bitu RBIF** v registru **INTCON.** Tento bit **RBIF** může nastavit **jakýkoliv pin (RB4 – RB7), proto je hradlo OR čtyřvstupé !**

Shrneme–li tedy **blok pro přerušovací systém**, můžeme říci, že je to vlastně určitá část přerušovacího systému, která slouží k vyhodnocování vstupního signálu z pinu. Vyhodnocení signálu z pinu má za následek **nastavení** bitu **RBIF (nastala změna log.úrovně na pinu)** nebo naopak jeho **nulování (nenastala změna log.řovně na pinu)**.V případě, kdy je nastaven bit **RBIF** (změna log.úrovně) a taktéž bit **RBIE** (povolení přerušení od pinů **RB4 – RB7**) v registru **INTCON =>** nastane skok na obslužný podprogram pro přerušení tj. skok v paměti programu na **adresu 0004H.**

Pozn: V případě, že využijeme některý z pinů PORTB jako vstup, musíme na daný pin připojit VDD přes rezistor např. 47KΩ nebo využít říditelných rezistorů.

ČASOVAČ/ČÍTAČ

PIC16F84 má **pouze jeden časovač/čítač**, který může pracovat ve dvou režimech => **čítač vnějších událostí,** nebo **časovač vnitřních hodin.** Pod pojmem **časovač** nebo **čítač** si lze představit **8-bitový** registr, který je označován jako **TMR0.** Ve funkci **časovače** je jeho hodnota po každém strojovém cyklu zvyšována (inkrementována) o **1**. Jeden strojový cyklus = 4 periody oscilátoru. Pokud použijeme oscilátor s f = 4Mhz => 1 strojový cyklus = 1µs => je to vhodný kmitočet krystalu na **konstrukci přesných časových aplikací**. Z níže uvedeného blok.schéma je nastíněna celková funkce časovače/čítače. Registr TMR0 nazýváme **čítačem** v případě, jestliže **načítá externí zdrojový signál** od pinu **RA4/INT**. V případě, že registr TMR0 **načítá signál od vnitřních hodin**, jej nazýváme **časovačem**. Při přetečení tohoto registru (tj. z hodnoty 256 => 0) je **vždy** nastaven tzv. příznakový bit (označení **T0IF**), který je zahrnut v registru (INTCON). Bit **T0IF** slouží jako indikace pro přerušovací systém. Je –li nastaveno povolení přerušení od TMR0 a bit T0IF => skok na obslužný podprogram. Přerušovací systém bude vysvětlen v kapitole **PŘERUŠENÍ. Je-li do TMR0 zapsána hodnota, je inkrementace jeho obsahu povolena až po 2 strojových cyklech!** K registru TMR0 lze přiřadit tzv.**programovou děličku (přidělení se provádí bitem PSA = 0)**, která má za úkol celočíselně dělit přivedený vstupní signál. Je jedno zda to bude v režimu **časovač** nebo **čítač**.

Z tohoto nám vyplývá, že dělička způsobuje delší dobu čítání **reg. TMR0.** Tuto dobu čítání lze nastavit jak hodnotou TMR0, která se dá do tohoto registru přímo zapsat, tak také nastavením dělícího poměru na děličce. Pro nastavení dělícího poměru na děličce nám slouží bity **(PS2, PS1, PS0) =>** vidíme, že máme 3 nastavovací bity => 8 různých kombinací nastavení dělícího poměru. Dělící poměr je od (**1:2 - 1:256**). Dělička je programátorem nastavována, avšak **není přístupná**. Je–li dělička přiřazena k TMR0 tak nulováním TMR0 nulujeme současně i děličku. Ovládání **časovače/čítače** je prováděno bity, které jsou obsaženy v registru **OPTION**.

Následující tabulka nám ukazuje výpočetní vztah:

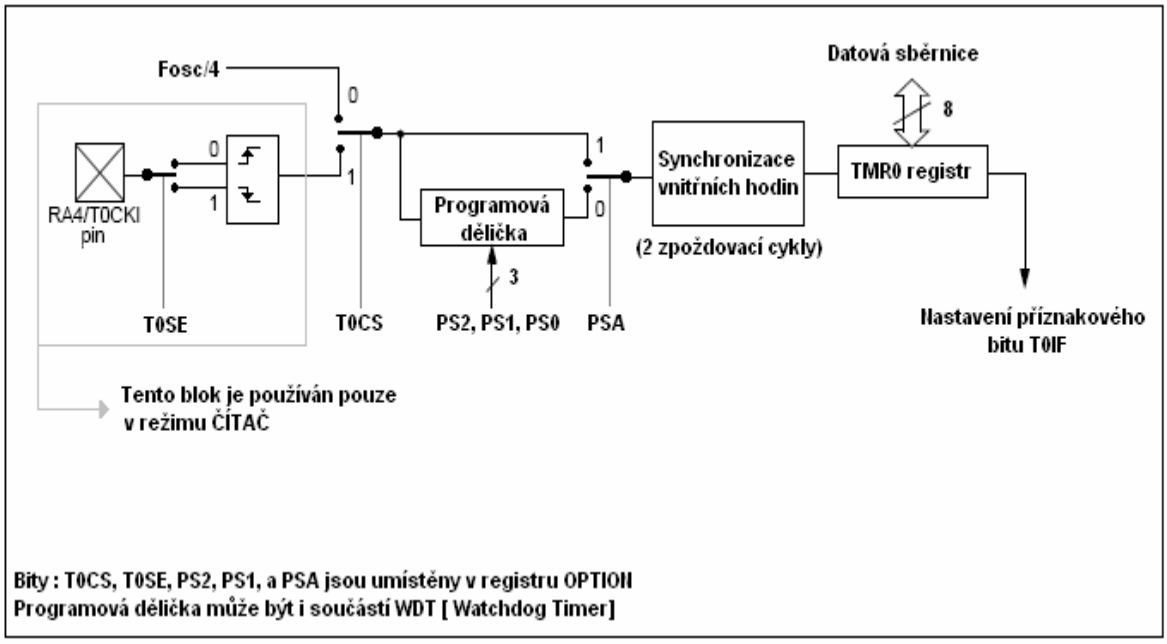
Při **přiřazení děličky k TMR0** platí následující jednoduchá rovnice:

(256 – N1)*(N2) = N3

N1: hodnota,která by se případně nastavila (zapsala) do registru TMR0
N2: nastavení dělícího poměru (1:2, 1:4, 1:8, 1:16, 1:32, 1:64, 1:128, 1:256) na děličce pomocí (PS2 – PS0)
N3: čas, který bude TMR0 čítat (µs), než nabude plné hodnoty 256

Přiřadíme–li děličku k registru TMR0 je totéž, jako bychom vzali dva 8-bitové čítače, které spojíme do série. První čítač by zastupoval **TMR0** a ten druhý by zastupoval **děličku**. **Navíc u druhého čítače bychom mohli měnit jeho (modulo) v závislosti na bitech (PS2 – PS0).** Při přetečení vyššího registru, tedy TMR0 je nastaven příznakový bit **T0IF**, který slouží především pro přerušovací systém, jako indikace přetečení TMR0.

Blokové schéma časovače/čítače Timer 0



<u>Režim – ČASOVAČ – v tomto režimu bereme signál z oscilátoru (označován OSC)</u>

Registr TMR0 nastavený jako **ČASOVAČ**, je též nazýván **<u>časovačem vnitřních hodin</u>** , tzn. reg. TMR0 je každou čtvrtinou celkové frekvence [(označení **fOSC/4**) **= doba instrukčního cyklu]** zvyšován **o 1**. Při přetečení TMR0 je nastaven příznakový bit T0IF. Tento režim využíváme většinou pro vytvoření časové smyčky. Proč jen obyčejné nevnořené čas.smyčky ? Protože **TMR0 má zpětnou vazbu na přerušovací systém a tím může procesor vykonávat jinou činnost, než-li počítat časovou smyčku.** Daný režim lze nastavit v reg. OPTION.

Nastavovací bity v OPTION:
T0SE – tento bit nás v tomto režimu nezajímá, je pouze pro čítač
T0CS – tento bit nastavíme na **log.0**, protože jinak bychom pracovali v režimu čítač
PSA - tento bit můžeme/nemusíme využít. **Při log.0 je dělička přiřazena časovači**

PS2, PS1, PS0 – při využití těchto bitů [**musí** být (PSA = 0)] se řídíme výše zmíněnou rovnicí pro určení času. **T0IF** – tento bit nastavuje procesor -> indikace pro přerušovací systém, **je nutné jej v obslužném prog. nulovat**

Režim - ČÍTAČ - v tomto režimu bereme signál ze vstupu RA4 tj.externí signál přivedený na tento pin

Registr TMR0 nastavený jako **ČÍTAČ**, je též nazýván **čítačem vnějších událostí.** , tzn. reg. TMR0 je každou čtvrtinou celkové frekvence [(označení **fOSC/4**) = **doba instrukčního cyklu**] zvyšován **o 1**. Při přetečení TMR0 je nastaven příznakový bit T0IF. Tento režim se využívá všude, kde chceme např. zjistit frekvenci z externího zdroje signálu. Daný režim lze nastavit příslušnými bity v registru OPTION.

Nastavovací bity v OPTION:
T0SE – na jakou hranu externích hodin má procesor reagovat **log.0 – náběžná hrana, log.1 – sestupná hrana**
T0CS – tento bit nastavíme na **log.1**, protože jinak bychom pracovali v režimu časovač
PSA - tento bit můžeme/nemusíme využít. **Při log.0 je dělička přiřazena čítači**
PS2, PS1, PS0 – při využití těchto bitů [**musí** být (PSA = 0)] se řídíme výše zmíněnou rovnicí pro určení času.
T0IF – tento bit nastavuje procesor -> indikace pro přerušovací systém, **je nutné jej v obslužném prog. nulovat**

Pozn: Při (PSA = 1) je dělička přiřazena WDT – (Watchdog Timer) -> dělící poměr k TMR0 = 1:1

EEPROM paměť

Teorie:

EEPROM jsou elektricky zapisovatelné paměti a uchovávají data i po odpojení napájecího napětí => tímto se **EEPROM** paměti liší od klasických **RAM** pamětí. Při měření nějaké veličiny potřebujeme dost často uschovat důležité informace, které by sloužily k dalšímu zpracování, nebo slouží tyto paměti také dost často k uchování námi zadaných předvoleb. **EEPROM paměti jsou elektricky programovatelné (tj. zápis) a elektricky mazatelné (tj. mazání).** Procesor **PIC16F84** má EEPROM paměť o velikosti **64Bytů po 8-bitech**. Abychom mohli tuto paměť ovládat (tj. zápis a čtení), je zapotřebí 4 registrů.
Tyto čtyři registry jsou:
EEDATA – slouží k zadávání námi zvolené adresy **(max. do 39H)**
EEDATA – slouží k zadávání námi zadaných dat
EECON1 – slouží jako řídící registr EEPROM paměti
EECON2 – slouží jako ochranný registr (prvek) proti nežádoucímu přepsání dat

Abychom mohli začít používat EEPROM paměť, je nutné si podrobně popsat registr **EECON1**, protože o zbývajících třech registrech nám bohatě stačí výše uvedené informace. U registru **EECON2** si řekneme, že ochrana proti nežádoucímu přepsání spočívá v postupném ukládání předem daných (nelze je měnit!) dat **=> tj. 55H, AAH.**

EECON1 – řídí přístup k paměti EEPROM adresa: 88H

----	**----**	**----**	**EEIF**	**WRERR**	**WREN**	**WR**	**RD**
7	6	5	4	3	2	1	0

(číslo bitu)

EEIF – [EEPROM Write Operation Interrupt Flag bit] indikuje zápis **do EEPROM** paměti
0 – zápis do paměti zatím nenastal
1 – dokončení zápisu do paměti => **data byla do EEPROM zapsána**
WRERR – [Write Error bit] při zápisu do paměti je tento bit kontrola, zda byl zápis **úspěšný**
0 – operace zápisu byla kompletní (úspěšná)
1 – operace zápisu se nezdařila, byla **přerušena /MCLR nebo WDT**
WREN – [Write Enable] řízení zápisu do **EEPROM**
0 – zápis do **EEPROM** je zakázán
1 – zápis do **EEPROM** je povolen
WR – zápis do **EEPROM** paměti, bit nelze nulovat **=> je automaticky nulován procesorem**
0 – zápis není požadován [programátorem samozřejmě!] – **nelze jej ovlivnit**
1 – zápis je požadován

RD – čtení z **EEPROM** paměti, bit nelze nulovat **-> je automaticky nulován procesorem**
0 – není požadováno čtení z paměti – **nelze jej ovlivnit**
1 – je požadováno čtení z pamět

Nastavení čtení DAT z EEPROM paměti a následné načtení:

Před operací čtení je nutno nastavit do registru **EEADR** adresu **[max. do 39H!],** z které bude probíhat čtení. Nastavením bitu **RD** v registru **EECON1** začne čtení z **EEPROM paměti** tj.uložení dat do registru **EEDATA. Čtení trvá pouze 1 strojový cyklus.**

EEADR <- adresa ; zápis požadované adresy
EECON1.RD <- 1 ; nastavení požadavku čtení
EEDATA <- EEPROM ; přesun požadovaných dat do registru EEDATA

Nastavení zápisu DAT do EEPROM paměti a následné zapsání:

Při zápisu dat se do registru **EEADR** uloží adresa a do registru **EEDATA** data k zápisu. Doporučuje se dále zakázat všechna přerušení tj. **(GIE = 0)** v registru **INTCON.** Nyní se provede povolení zápisu do **EEPROM** paměti nastavením bitu **WREN** v registru **EECON1** a provede se postupný zápis hodnot **55H** a **AAH** do registru **EECON2 [ochrana proti náhodnému přepsání]**. Následně se nastaví bit **WR (zapisuje se do EEPROM)** v registru **EECON1** a je možné opět zakázat zápis a povolit přerušení. Provedení zápisu trvá několik **ms,** jeho dokončení je možné sledovat pomocí přerušovacího systému -> tj. příznakový bit **EEIF.**

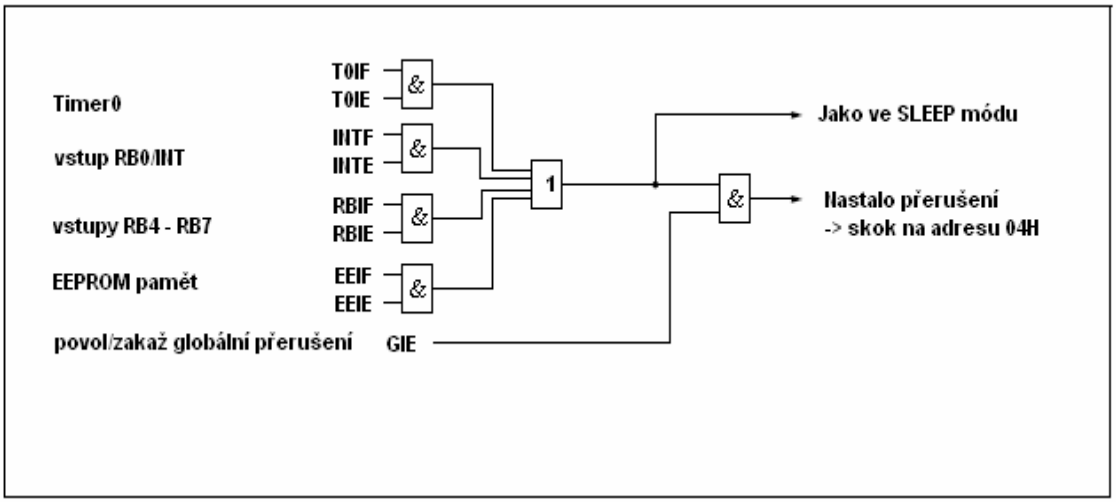
EEADR <- adresa ; na jakou adresu se mají data zapsat
EEDATA <- data ; požadovaná data
INTCON.GIE <- 0 ; zákaz globálního přerušení, **kvůli náhodnému přerušení**
EECON1.WREN <- 1 ; povolení zápisu do EEPROM
EECON2 <- 55h ; **ochrana zápisu – tato hodnota je dána napevno**
EECON2 <- AAh ; **ochrana zápisu – tato hodnota je dána napevno**
EECON1.WR <- 1 ;požadování zápisu dat do EEPROM. **Zápis dat do EEPROM**
EECON1.WREN <- 0 ; zákaz zápisu do EEPROM
INTCON.GIE <- 1 ; povolení globálního přerušení -> po této instr. lze testovat bit EEIF v registru EECON1

Pozn: Vidíme, že zápis trvá několik ms, proto je nejlepší testovat bit **EEIF tj. v případě, kdy jsme použili přerušovací systém, ale zároveň testujeme po tomto bitu bit WRERR, poněvadž nám dává velmi důležitou informaci o dokončení či nedokončení zápisu do EEPROM paměti !** V případě kdy budeme z **EEPROM** paměti pouze číst, je čtení doslova okamžité tj. data jsou nám dána následující strojový cyklus.

PŘERUŠENÍ

Přerušovací systém používáme, protože usnadňuje komunikaci s jinými periferiemi (např. externí paměť, jiný µkontroler atd.) a tím pomáhá µkontroleru dost velkou mírou. Jednoduše řečeno to znamená, že přerušovací systém registruje změnu jen na určitých periferiích, které nastavují patřičné signály tj.(příznakové bity) tomuto přerušovacímu systému. Procesor **PIC16F84** má tři periferie, jenž mohou sloužit i jako zdroje pro přerušení.Jsou to tyto: piny **RB0/INT, RB4 – RB7, EEPROM paměť a čítač/časovač TMR0 -> 4 zdroje pro přerušení.** Využijeme-li tedy nějakou z nabízených periferií jako zdroj pro přerušení, musíme nejprve námi použité přerušení povolit. Velmi dobře to vystihuje blokové schéma celkového přerušovacího systému z něhož je na první pohled vidět jak pracuje logika celkového přerušovacího systému.

Blokové schéma řídící logiky pro celkový přerušovací systém



Pouze pomocí registru **INTCON** můžeme zakazovat nebo povolovat veškerá přerušení, která má procesor k dispozici. Rozhodneme – li se využít **alespoň** jeden zdroj pro přerušení, musíme nejspíše toto přerušení povolit => tj. musíme splnit dvě nutné podmínky.

1. povolit globální přerušení – nastavení bitu **GIE**

2. povolit jednotlivá přerušení – závisí jen na nás, jaké budeme povolovat/zakazovat nastavení bitu/bitů **T0IE, INTE, RBIE, EEIE.**

Teprve po splnění třetí podmínky nastane přerušení. Procesor zjistí přerušení tím, že příslušný zdroj pro přerušení nastaví svůj předem daný tzv. příznakový bit v registru **INTCON.** Dvě podmínky z oněch třech jsou pro všechny zdroje pro přerušení stejné.Po splnění těchto dvou podmínek můžeme říci, že jsme **pouze** povolili zdroje pro přerušení. **Teprve po splnění třetí podmínky od jakéhokoliv zdroje přerušení procesor PIC16F84 skáče na adresu programové paměti 04H, která je pevně určena výrobcem.** Avšak pozor. Třetí podmínka je u každé periferie trochu jiná, protože každá periferie se používá na něco jiného a tudíž si tuto podmínku u každé periferie vysvětlíme.

Přerušení nastalo od zdroje - splnění třetí podmínky :

RB0/INT: periferie je zde **brána B.**
Třetí podmínka zde znamená změnu log.úrovně na pinu RB0, přičemž je zde možnost nastavení pomocí bitu **INTDEG** v registru **OPTION,** na jakou hranu vstupního signálu (tj. čelo, týl) má přerušovací systém reagovat. **Při přerušení se nastaví příznakový bit INTF.**
RB4 – RB7:
U tohoto zdroje pro přerušení je podmínka stejná jako u pinu **RB0/INT, avšak při přerušení se nastaví příznakový bit RBIF.**
čítač/časovač TMR0 : takto se označuje tato periferie.
Třetí podmínka zde znamená přetečení obsahu 8 – bitového registru TMR0, přičemž je zde možnost nastavení pomocí bitu **T0SE** v registru **OPTION,** na jakou hranu **vstupního signálu/vnitřních hodin** má probíhat inkrementace celkového obsahu registru **TMR0. Při přerušení se nastaví příznakový bit T0IF.**
EEPROM: takto se označuje tato periferie.
Třetí podmínka zde znamená ukončení zápisu námi požadovaných dat do této paměti. Při přerušení se nastaví příznakový bit EEIF. Při využití tohoto přerušení se před zápisem dat do paměti EEPROM doporučuje vypnout ostatní lokální přerušení, z důvodů bezchybného zápisu dat do této paměti.
Nyní jsme si podrobně ukázali, jak nastavit jednotlivá přerušení. Procesor **PIC16F84** nemá v přerušovacím systému zahrnut blok (např. procesory s jádrem 8051 ho mají), jenž nám umožňuje definovat **prioritu** pro přerušení. Otázkou priority se zabýváme v případě, kdy procesor přijme najednou dva a více signálů (tj. příznakové bity) od jednotlivých zdrojů pro přerušení a my tímto blokem určujeme jaké přerušení má být dříve obslouženo tj. které má z nich přednost. V případě, kdy probíhá obsluha některého ze zdrojů pro přerušení nemůže být obsluhováno jiné přerušení, které by přišlo po tomto přerušení. Z tohoto nám vyplývá, že priorita **pouze** určuje, které přerušení má být jako první obsluhováno. Proč se zde vůbec o nějaké prioritě zmiňujeme? Protože hraje dost podstatnou roli v případě, kdy máme např. až 15 zdrojů pro přerušení a chceme některé zdroje pro přerušení zpracovávat dříve, než-li ty, které mají nižší prioritu. I když tento blok **PIC16F84** neobsahuje, tak si jej můžeme **pomocí instrukcí sami vytvořit => můžeme v průběhu obsluhy programu pro přerušení sami rozhodovat, jaké přerušení (tj. příznakový bit) se má dříve zpracovávat, ale taktéž zároveň můžeme obsluhovat i přerušení, které by nemělo být zpravidla obsluhováno.** Když se nad tím zamyslíme, dospějeme k názoru, že by bylo dobré mít tento blok po ruce, avšak opak je pravdou => toto je vykompenzováno tím, že příznakové bity se automaticky **nemažou,** to znamená, že v obslužném podprogramu pro přerušení musíme tento příznakový **bit/bity vždy vymazat, jinak by nastala nekonečná opakující se smyčka obslužného programu.** Taktéž nám z toho vyplývá, že v případě, kdy provádíme obsluhu některého ze zdrojů pro přerušení a po něm hned přijmeme další žádost od jiného zdroje pro přerušení, se **vždy** nastaví příslušný příznakový bit, který zůstává nastaven i v případě, kdy již žádost o přerušení zanikla => toto je výhoda oproti jádru 8051. V případě, že přijde žádost o přerušení, procesor dokončí instrukci, kterou měl případně rozdělanou v hlavním programu či snad podprogramu a skočí vždy na adresu v programu paměti **04h. Po obsluze přerušení nastane skok na adresu, kde procesor posledně skončil.**

Závěrem můžeme říci toto:
Přerušovací systém jakéhokoliv procesoru je dosti důležitý, protože nám zpracovává signály, které jsou většinou potřeba k okamžitému vyhodnocení, proto je tento přerušovací systém tak důležitý. V případě, kdy se jedná o aplikaci, která vyžaduje velmi přesné řízení a je náročná na zpracování velkého počtu signálů, je využití přerušovacího systému na místě.

REGISTRY

Naučit se dobře programovat **PIC16F84** neznamená naučit se jen instrukce, ale znamená to především velmi dobrou znalost hardware a s tím souvisí právě zmíněné registry. Pomocí registrů děláme veškeré ovládání celého mikropočítače, protože každá periferie či snad jen říditelný blok má svého zástupce **(registr),** pomocí něhož lze např. nastavovat různé módy, zapisovat konkrétní hodnoty do periferiích atd. Procesor **PIC16F84** má několik ovládacích registrů.Zmíníme se samozřejmě o všech registrech. O některých registrech se stačí pouze informativně zmínit, avšak některé si popíšeme podrobně, protože jsou velmi důležití při ovládání mikropočítače. Než si budeme popisovat jednotlivé registry, přiomeneme si, že jsou uloženy ve dvou bankách. Vyšší řady procesorů PIC mají čtyři banky. **Blokové schéma paměti RAM vystihuje (organizaci) na jakých adresách jsou registry umístěny.**

Adresa	BANKA 0	BANKA 1	Adresa
00h	INDF *	INDF *	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 (RAM)**	**Mapováno do stránky 0**	8Ch až CFh
50h až 7Eh	**neimplementováno (čte jako 0)**	**neimplementováno (čte jako 0)**	D0h až FFh

*** Není fyzický registr**

STATUS – nastavení bank, příznakové bity (nikoli od přerušení !) po určitých matematických operacích.

IRP	RP1	RP0	T0	PD	Z	DC	C

adresa: 03H<->13H

7 6 5 4 3 2 1 0

bit.adresa

IRP – Výběr bank pro nepřímé adresování
0 – Banka 0 a 1
1 – Banka 2 a 3

RP1 – RP0 – Výběr banky registrů
00 – Banka0
01 – Banka1
10 – Banka2
11 – Banka3

T0 – [TIME OUT] indikuje důvod TIME - OUT
0 – přetečení **WDT**
1 – reset, **CLRWDT** nebo **SLEEP**

PD – [POWER - DOWN] indikuje důvod startu
0 – instrukce **SLEEP**
1 – reset, **CLRWDT**

Z –[Zero bit] indikuje nulový výsledek operace
0 – výsledek není nulový
1 – výsledek je nulový

DC – [Digit carry] indikuje přetečení z **3**bitu -> **4**bit daného bytu = registru.
0 – poloviční přenos mezi bity nenastal
1 – poloviční přenos mezi bity nastal

C – [Carry] indikuje přetečení z **7**bitu -> **8**bit -> daný registr přetekl tzn. je nulován
0 – přenos mezi bity nenastal
1 – přenos mezi bity nastal

Pozn:
V našem případě nás bit IRP nezajímá, protože PIC16F84 nemá banku 2 a 3 -> nebudeme jej používat. Taktéž bit RP1 nás nezajíme (předešlé důvody).

OPTION - nastavování a přidělování předděličky + další nastavení

RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

adresa: 81H

7 6 5 4 3 2 1 0

bit.adresa

RBPU – povolení vnitřních **[PULL – UP]** odporů na portB
0 – povoleno
1 – zakázáno

INTDEG – určuje aktivní hranu **vnějšího signálu tj. [RB0/INT a RB4 – RB7]** pro aktivaci přerušení
0 – spádová hrana
1 – náběžná hrana

T0CS – určuje zdroj signálu pro Tmr0
0 – vnitřní oscilátor **[1/4 OSC]**
1 – vstup **RA4 – Tmr0**

T0SE – určuje aktivní hranu pro práci s **Tmr0**
0 – náběžná hrana
1 – sestupná hrana

PSA – určuje připojení předděličky
0 – před **Tmr0**
1 – za **WDT [Watchdog Timet]**

PS2,PS1,PS0 – určují dělící poměr předděličky viz. tabulka

PS2	PS1	PS0	Tmr0	WDT
0	0	0	1:2	1:1
0	0	1	1:4	1:2
0	1	0	1:8	1:4
0	1	1	1:16	1:8
1	0	0	1:32	1:16
1	0	1	1:64	1:32
1	1	0	1:128	1:64
1	1	1	1:256	1:128

Pozn: Dělící poměr 1:1 pro Tmr0 lze získat tím, že připojíme předděličku za WDT tj.[PSA = 1]

INTCON - povolení/zákaz veškerých přerušení + příznakové bity od jednotlivých přerušení

GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF

adresa:0BH <->8BH

7	6	5	4	3	2	1	0

bit.adresa

Jednotlivá lokální a globální přerušení

GIE – povolení globálního přerušení
0 – přerušení zakázán**a**
1 – přerušení povolen**a**

EEIE – povolení přerušení od ukončení zápisu do EEPROM
0 – přerušení zakázáno
1 – přerušení povoleno

T0IE – povolení přerušení od Tmr0
0 – přerušení zakázáno
1 – přerušení povoleno

INTE – povolení vnějšího přerušení **– RB0/INT !!!**
0 – přerušení zakázáno
1 – přerušení povoleno

RBIF – povolení přerušení od změny na pinech **RB4 – RB7**
0 – přerušení nenastalo
1 –přerušení nastalo

Příznakové bity od jednotlivých lokálních přerušení

T0IF – indikuje **[tj. dává zprávu o]** přetečení Tmr0
0 – k přetečení nedošlo
1 – došlo k přetečení

INTF – indikuje požadavek vnějšího přerušení **[tj.změna log.úrovně]** na **RB0/INT**
0 – přerušení nenastalo
1 – přerušení nastalo

RBIF – indikuje změnu log.úrovně na pinech **RB7 – RB4**
0 – přerušení nenastala
1 – přerušení nastalo

Pozn: Programátor <u>musí !!!</u> tyto příznakové bity v obslužných programech nulovat. Kdyby se tak nestalo => cyklická smyčka v přerušovacím systému. V případě použití WDT by byl procesor nejspíše[záleží na nulování WDT] neustále resetován !!! Příznakový bit pro EEPROM paměť není bohužel zahrnut v registru INTCON, nýbrž v registru EECON1, kde příznakový bit je EEIF. Pro využití přeruš.systému musí programátor taktéž tento bit EEIF v reg. EECON1 vynulovat.

<u>EECON1</u> – řídí přístup k paměti EEPROM + příznakový bit pro EEPROM paměť

----	**----**	**----**	**EEIF**	**WRERR**	**WREN**	**WR**	**RD**

adresa: 88H

7	6	5	4	3	2	1	0

bit.adresa

EEIF – [EEPROM Write Operation Interrupt Flag bit] indikuje zápis **do EEPROM** paměti
0 – zápis do paměti zatím nenastal
1 – dokončení zápisu do paměti -> **data byla do EEPROM zapsána**

WRERR – [Write Error bit] při zápisu do paměti je tento bit kontrola, zda byl zápis **úspěšný**
0 – operace zápisu byla kompletní (úspěšná)
1 – operace zápisu se nezdařila, byla **přerušena /MCLR nebo WDT**

WREN – [Write Enable] řízení zápisu do **EEPROM**
0 – zápis do **EEPROM** je zakázán
1 – zápis do **EEPROM** je povolen

WR – zápis do **EEPROM** paměti, bit nelze nulovat **-> je automaticky nulován procesorem**
0 – zápis není požadován [programátorem samozřejmě!] – **nelze jej ovlivnit**
1 – zápis je požadován

RD – čtení z **EEPROM** paměti, bit nelze nulovat **-> je automaticky nulován procesorem**
0 – není požadováno čtení z paměti – **nelze jej ovlivnit**
1 – je požadováno čtení z paměti

<u>EEDATA</u> – adresa: 8H

Tento registr slouží k **přístupu k datům paměti EEPROM [64 Bytů – Bytové pole].** V případě, že budeme provádět zápis do **EEPROM** paměti, data k zápisu vložíme právě do tohoto registru a následně budou při **(WR=1)** uložena do **EEPROM** paměti. Při čtení **EEPROM** paměti se data přenesou do tohoto registru.

Working Register – pracovní registr (označován W), který je taktéž nazýván jako **akumulátor či střadač.** Přes tento registr se uskutečňují veškeré instrukce **matematického a přenosového typu !**

INDF* – adresa: 00H <->80H
Tento registr **není fyzický.** Představuje jiný registr, jehož index je uložen v registru **FSR**. Tento registr je používán při nepřímém adresování jako registr pro **čtení/zápis.**

FSR – adresa: 04H<->84H
Tento registr slouží jako **ukazatel pro nepřímé adresování, tj. nemusíme přepínat banky.** Zapsání hodnoty do tohoto registru (tj. 0h – FFh) se nám zpřístupní odpovídající registr jako registr **INDF.** Z tohoto všeho nám vyplývá, že v případě, kdy budeme používat nepřímé adresování, pak registr **FSR** slouží na zadávání příslušného registru tj. **ADRESA** a registr **INDF** umožňuje **čtení/zápis** adresovaného registru tj. **DATA.**

U registrů **INDF** a **FSR** si uděláme názorný příklad **nepřímého adresování.** Nepřímé adresování funguje nezávisle na přepínání bank -> je jedno v jaké bance se nyní nacházíme. **U přímého adresování je důležité, v jaké bance se nyní nacházíme,** protože přímé adresování je platné pouze v bance, která je aktivní -> nastavení bank se provádí v registru **STATUS, RP0.**

Ve zdrojovém textu vypadá NEPŘÍMÉ ADRESOVÁNÍ takto:

```
movlw H‘0C‘   ; zadáváme adresu (0h – FFh)
movwf FSR     ; adresa se přesouvá do registru FSR
movlw H‘55‘   ; zadávání dat (0h – FFh)
movwf INDF    ; zapsání dat na adresu 0Ch
```

TMR0 – adresa: 01H
Tento registr představuje **časovač/čítač,** u kterého lze nastavit mód pomocí registru **OPTION.**

PORTA – adresa: 05H
Tento registr umožňuje **čtení/zápis brány A.**

PORTB – adresa: 06H
Tento registr umožňuje **čtení/zápis brány B.**

TRISA – adresa: 85H
Tento registr **umožňuje řízení bran = portů.** Piny **PORTA,** které mají být jako vstupy se nastaví na příslušné bitové pozice v **TRISA log.1** a naopak.

TRISB – adresa: 86H
Tento registr **umožňuje řízení bran = portů.** Piny **PORTB,** které mají být jako vstupy se nastaví na příslušné bitové pozice v **TRISB log.1** a naopak.

EEADR – adresa: 09H

Pomocí tohoto registru lze zadávat námi zadané adresy pro **EEPROM paměť -> můžeme číst z námi zadané adresy, nebo můžeme provádět zápis do EEPROM paměti na námi zadanou adresu.**

EECON2* – adresa: 89H

Tento registr slouží jako **řídící registr EEPROM paměti. Není fyzický.** Jeho význam spočívá v ochraně proti nežádoucímu přepsání dat. Budeme-li provádět zápis do **EEPROM** paměti, je nutné před touto operací provést zápis hodnot **55H** a **AAH** do registru **EECON2.** Teprve po tomto zápisu do **EECON2** můžeme provést zápis do **EEPROM** paměti pomocí registru **EEDATA.**

PCL – adresa: 02H<->82H

Procesor PIC16F84 používá **13 . bitový čítač instrukcí (označován PC) => za každou instrukcí je obsah PC = PC+1.** Takto se zjišťujeme, na jaké adrese programové paměti procesor právě pracuje => velmi důležité při skoku v programu. Registr **PCL** nám zpřístupňuje **nižších 8 bitů z PC.** Zbylých **vyšších 5 bitů** z **PC** je tvořeno registrem **PCLATH.** Vezmeme-li v úvahu, že **PIC16F84 má 1KB velkou paměť programu,** pak z registru **PCLATH** využijeme u tohoto typu procesoru **pouze 2 bity!** Je dobré se zmínit o instrukcích **CALL** a **GOTO,** protože tyto instrukce ukládají do **PC 11 bitů.** Z tohoto nám vyplývá, že u typu **PIC16F84 nemusíme stránkovat paměť, protože tyto skokové instrukce mohou kamkoliv skočit v programové paměti. V případě, kdybychom chtěli skočit nad 2KB programové paměti, je nutné ručně zadat adresu do registru PCLATH.** Registr **PCL** používáme většinou v případě, kdy si vytvoříme tabulku o **max.256 Bytech!!!** a následně v této tabulce potřebujeme vybírat jen určité hodnoty.

PCLATH – adresa: 0AH <=> 8AH

Tento registr **zpřístupňuje 5 vyšších bitů PC.** Veškeré věci spjaté s tímto registrem byly již zmíněny ve výše uvedeném popisu.

Tabulka registrů:

BANKA 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	--	neimplementováno, čte jako 0							
08h	EEDATA	uložená data paměti EEPROM							
09h	EEADR	uložená adresa paměti EEPROM							
0Ah	PCLATH	--	--	--					
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF

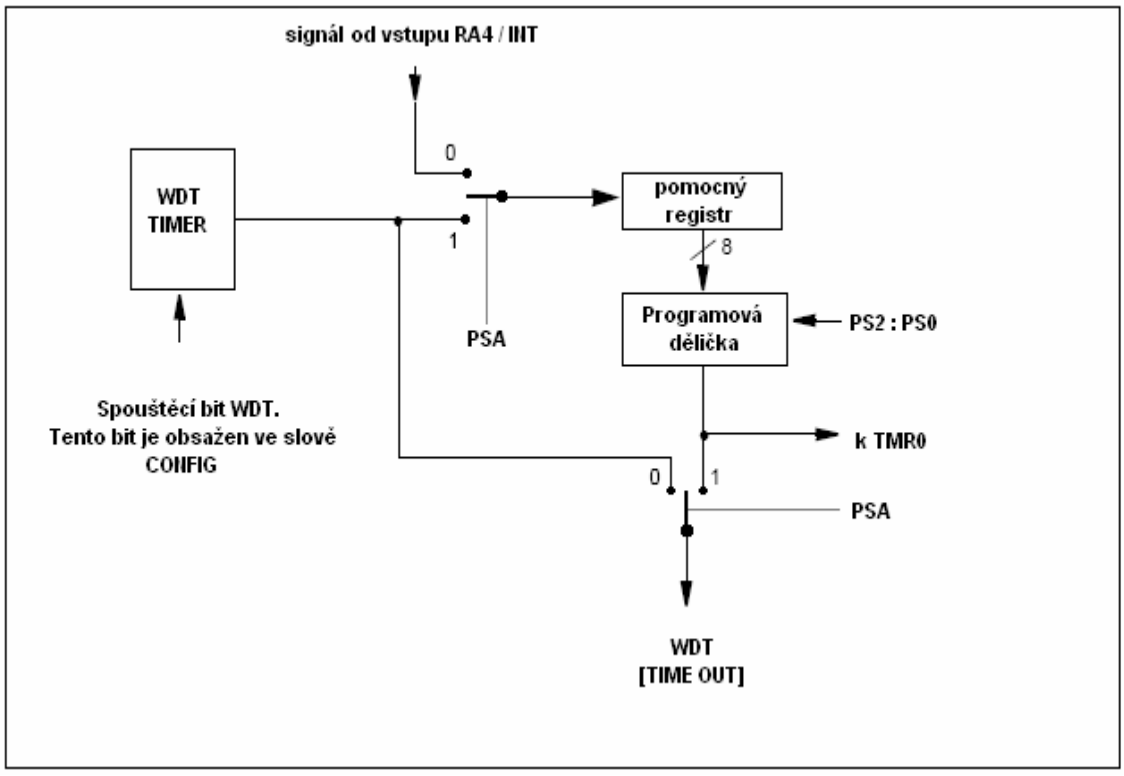
BANKA 1									
Adresa	**registr**	**bit 7**	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
80h	INDF	(není fyzický registr)							
81h	OPTION	/RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
82h	PCL								
83h	STATUS	IRP	RP1	RP0	/TO	/PD	Z	DC	C
84h	FSR								
85h	TRISA	--	--	--	nastavení PORTu A				
86h	TRISB	nastavení PORTu B							
87h	--	neimplementováno, čte jako 0							
88h	EECON1	--	--	--	EEIF	WRERR	WREN	WR	RD
89h	EECON2	(není fyzický registr)							
8Ah	PCLATH	--	--	--					
8Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF

Tato tabulka registrů podrobně ukazuje jednotlivé bity v daných registrech, a proto je velmi vhodná při programování.

WATCHDOG TIMER

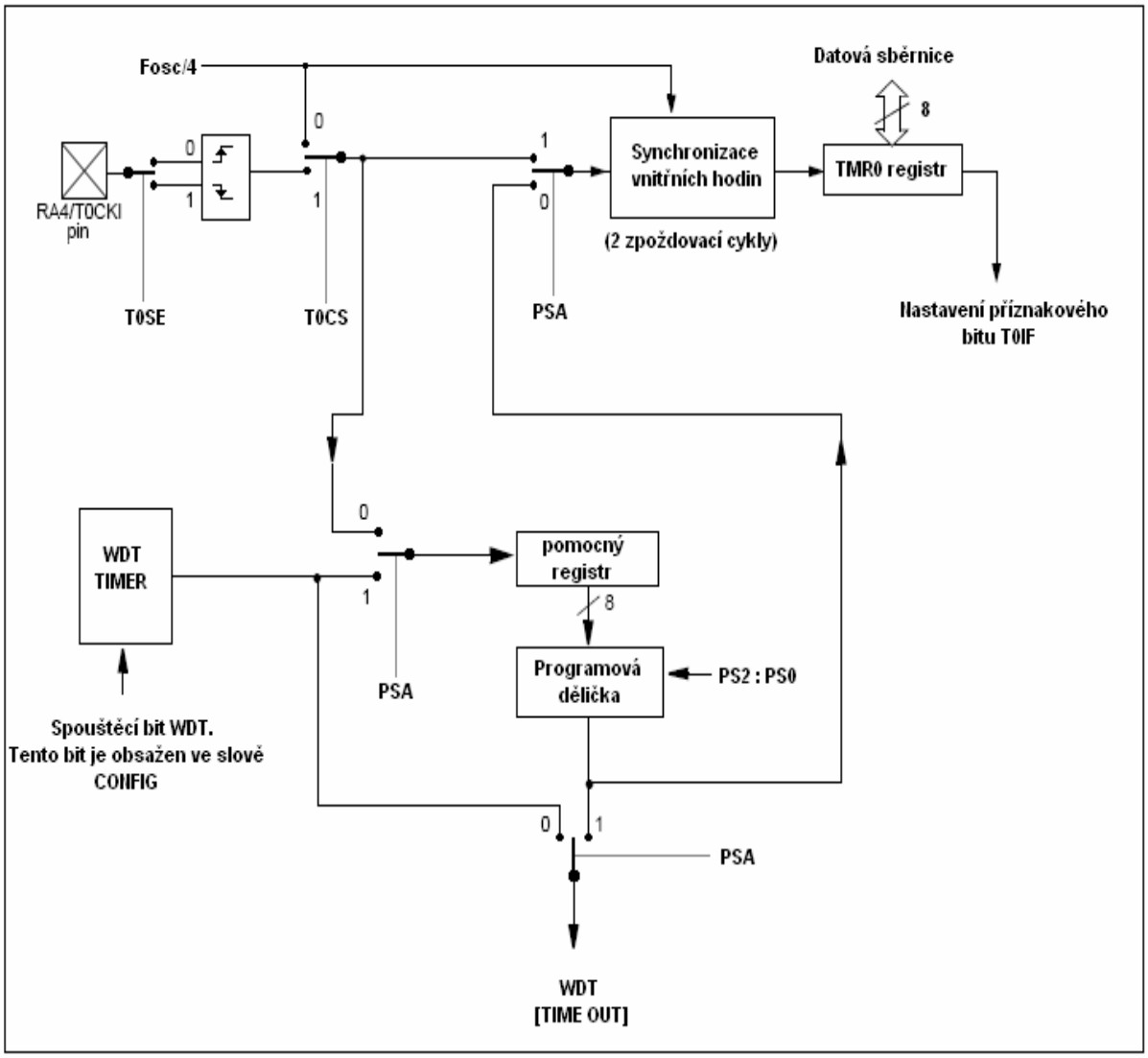
Wadtchdog Timer (označován WDT) je čítač, který si bere impulsy pro čítání od **RC oscilátoru.** Tento **RC** oscilátor **běží nezávisle na oscilátoru pro procesor, tedy i v době SLEEP. WDT přeteče za 18ms.** V případě kdy **WDT** přeteče, nastane **resetování procesoru => mikroprocesor skáče na adresu 0000H programové paměti tj. celkové nulování.** Tato funkce procesoru se aktivuje bitem **WDTE** v **konfiguračním slově CONFIG.** Z tohoto všeho nám vyplývá, že **WDT slouží ke správnému chodu programu => nastane – li zacyklení procesoru v určité části programu a není včas vynulován WDT, nastane resetování procesoru. WDT** lze nulovat speciální přidruženou instrukcí **CLRWDT.** Využijeme-li stav, jenž nazýváme **SLEEP,** a máme zapnutý **WDT** nastane **probuzení procesoru z tohoto režimu, nikoli resetování. Dobu WDT lze z 18ms zvýšit pomocí (PSA = 1) předděličky** až na **18ms *126 = 2.304s. Z blokového schéma je názorně vidět celková funkce WDT.**

Blokové schéma Watchdog Timeru a děličky



Z blokového schéma vidíme, že při použití předděličky musíme nastavit **(PSA = 1)** a pak už vybíráme dělící poměr pomocí bitů **(PS0, PS1, PS2). Při přetečení WDT (PSA = 0)** vyšle **WDT** signál k resetování procesoru -> v blok.schématu tj. **[TIME OUT].** Zde si ještě uvedeme **úplné blokové schéma WDT a TIMER0.** Toto schéma je pouze spojení již dříve zmíněných bloků, a proto není nutné si vysvětlovat funkci těchto bloků => toto celkové schéma slouží pouze pro představu funkce.

Blokové schéma WDT a časovače/čítače



Pozn: Před přepnutím předděličky mezi periferiemi **WDT** a **TIMER0** se doporučuje vynulovat tyto bloky, protože by mohlo dojít k náhodnému resetování procesoru.

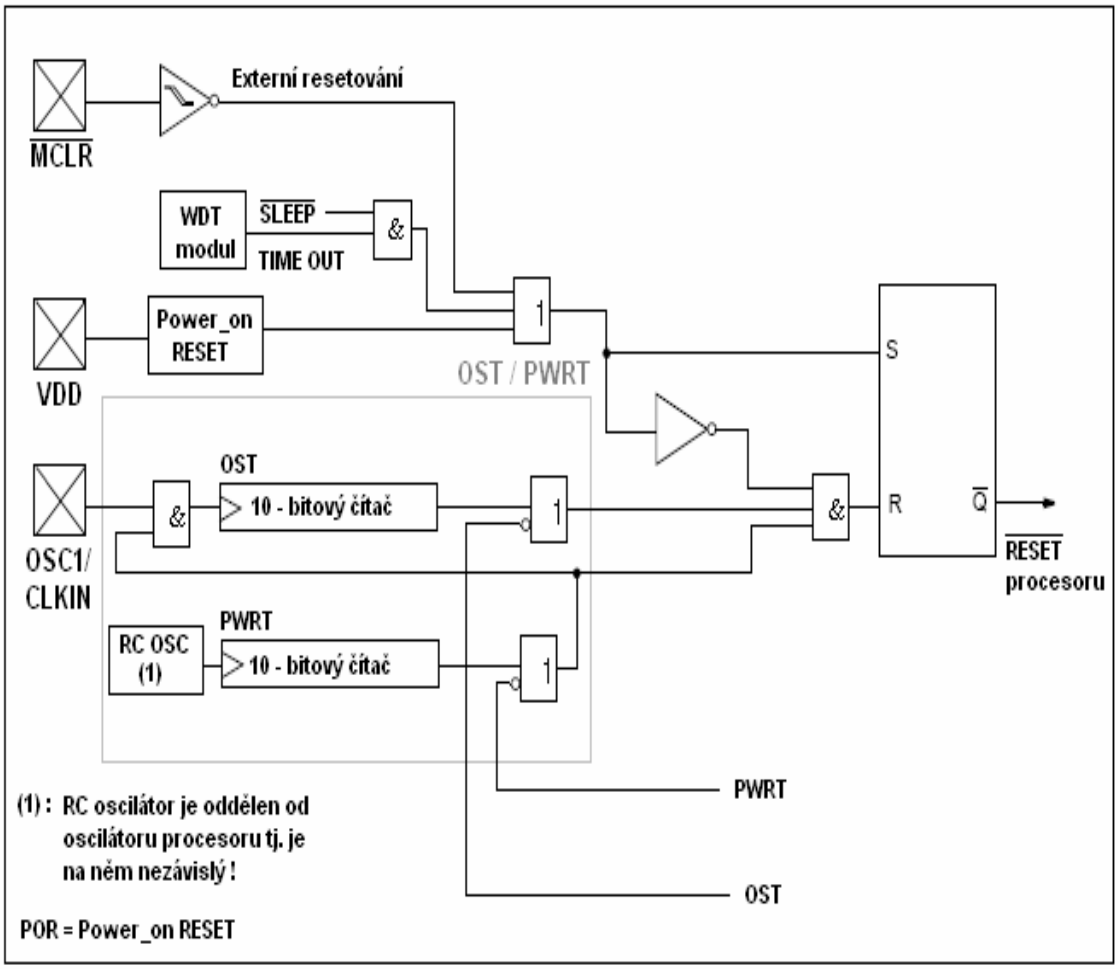
SLEEP

SLEEP (spaní) znamená u procesorů **PIC režim se sníženým příkonem (µA).** Tento režim se aktivuje speciální instrukcí, která je k tomuto systému přidružena tj. instrukce **SLEEP.** V případě, kdy jsme použili **WDT** a zároveň režim **SLEEP,** je **WDT** nulován. Následné dosažení času na **WDT** způsobí probuzení procesoru z režimu **SLEEP**. Během doby **SLEEP** je zastaven chod programu, čímž je tento stav způsoben vyřazením oscilátoru pro procesor. Probuzení procesoru z režimu **SLEEP** je možné udělat několika způsoby **=> RESET, WDT, RB0, RB4 – RB7 nebo ukončení zápisu do EEPROM paměti .** V případě, kdy jsme použili probuzení procesoru z režimu **SLEEP** vnějším příkazem na **pin /MCLR tj. RESET** je procesor resetován a začíná od adresy **0000H** programové paměti. V ostatních případech kromě **WDT (tj. přerušení)** udělá procesor za instrukcí **SLEEP** jednu instrukci a pak skočí na adresu **0004H, avšak pouze je-li povoleno přerušení => z tohoto nám tedy vyplývá, že procesor se probudí z režimu SLEEP i tehdy, je-li globální přerušení zakázáno tj. (GIE = 0).** V případě, kdy se procesor probudí z režimu **SLEEP** pomocí některého ze **zdrojů pro přerušení** nebo pomocí **WDT** a **není povoleno globální přerušení tj. (GIE = 0),** procesor pokračuje dále v programu za instrukcí **SLEEP, neskáče** v tomto případě na adresu **04H!**

RESET PROCESORU, SPECIFICKÉ FUNKČNÍ BLOKY

Blokové schéma pro **RESET** procesoru velmi názorně ukazuje, jak reset probíhá, ale taktéž ukazuje, od koho je reset požadován a taktéž určité specifické funkční bloky. Abychom si vysvětlili různé funkce, bude nejlepší, když si nejprve podrobně popíšeme **RESET** procesoru (tj. zdroje pro reset - **Power_on RESET, WDT a /MCLR**) a potom specifické funkce tohoto procesoru jako jsou **OST** a **PWRT.**

Blokové schéma resetovacího systému + specifické funkční bloky OST a PWRT



RESET procesoru

Reset je požadován od vnějšího pinu **/MCLR tj. externí resetování:**
Pro snadnější pochopení se dívejme na blokové schéma. V případě, že na tento pin **/MCLR** přivedeme **log.0,** je tato úroveň negována Schittovým KO s invertorem => převedení z log.0 na log.1 + vyhlazení hran signálu. Tato **log.1** se přesouvá na vstup **3-vstupového hradla OR,** které tuto **log.1** přesouvá na svůj výstup. Výstup tohoto **3-vstupového** hradla je spojen se **vstupem S** klopného obvodu typu **RS** (budeme označovat **RSKO**). **Log.1** zároveň směřuje na vstup invertoru. Invertor tuto **log.1** převede na **log.0**, čímž je **log.0** kopírována na vstup **3-vstupového hradla AND** => tímto jsme **vyřadili** ostatní vstupy hradla **AND** a na výstupu hradla **AND** bude nyní **stále log.0.** Z tohoto všeho nám tedy vyplývá, že **RSKO** je nastaven (tj. vstup **S = log.1, R = log.0) => výstup Q=log.1 RSKO => výstup Q non je tedy v log.0 => procesor je tedy resetován tzn. začíná na adrese 0000H programové paměti.**

Reset je požadován od modulu **WDT:**
Průběh resetování je stejný jako u pinu **/MCLR,** s tím rozdílem, že **WDT** při přetečení nastaví na vstupu (tj. **TIME OUT**) **2-vstupého hradla AND log.1.** Druhý vstup (tj. **/SLEEP**) hradla **AND** je (pro zjednodušení) **neustále v log.1 ->** od této chvíle je popis resetování naprosto stejný jako u pinu **/MCLR tj. nastává reset procesoru.** V případě, kdy využijeme mód **SLEEP,** je po přetečení **WDT** procesor probuzen z tohoto režimu (**SLEEP**)**, nikoli resetován!**

Reset je požadován od modulu **Power_on RESET (POR):**
Tento blok je **velmi** zapotřebí, protože **spouští procesor pouze od nejmenšího provozního napětí.** U procesoru **PIC16F84** je dán napěťový interval, který znemožňuje činnost procesoru, a to **(1.2V – 1.7V) =>** v tomto intervalu generuje blok **POR log.1,** čímž způsobuje reset procesoru. Takže je-li **VDD** (tj. napájecí napětí na procesoru) v tomto intervalu, je činnost procesoru **vyřazena.** Teprve po překročení 1.7V **(nejlépe alespoň 2V!)** je činnost procesoru spuštěna a procesor začína na adrese 0000H programové paměti => v tomto intervalu generuje blok **POR log.0,** čímž začíná procesor pracovat. **Z tohoto je zřejmé, že tento funkční blok je velmi důležitý, protože znemožňuje hazardní stavy na procesoru, které by byly způsobeny velmi rychlou skokovou změnou napájecího napětí VDD na tomto čipu.**

Specifické funkční bloky

Než začneme s popisem funkčních bloků, je třeba si uvědomit, že na výstupu **3-vstupového hradla OR** je **log.1 tj. bloky pro resetování nejsou aktivní.** Tím není procesor resetován, ale taktéž **není** ještě v činnosti, protože hradlo **AND** čeká na svých zbylých dvou vstupech úrovně **log.1,** které ovlivňují právě **specifické funkční bloky OST** a **PWRT.** Oba funkční bloky mají svůj opodstatněný význam, a proto si jejich funkci nyní popíšeme.

OST

Tato funkce umožní spuštění procesoru (nevyužíváme blok **PWRT**) teprve až po **1024 period oscilátoru (XT, HS, LP),** protože z **pinu OSC1** se přenáší signál na **10-bitový čítač =>** po přetečení tohoto čítače je na jeho **výstupu** dána **log.1,** která se přesouvá na **poslední zbylý vstup** hradla **AND.** Z tohoto je nám tedy jasné, že procesor není schopen pracovat bez oscilátoru, protože je blokem **OST** resetován tj. **OST slouží jako pojistka.** V případě, kdy budeme **využívat** i blok **PWRT,** je funkce **OST stejná,** ale s tím rozdílem, že se k času **1024 period OSC přičte čas 72ms.** Tento blok nelze **deaktivovat,** protože je **vždy** potřebný při spouštění procesoru tj. je **nepřístupný programátorovi.**

PWRT

Blok **PWRT** slouží jako **časovač** pro **pozdější zapnutí procesoru,** i když je **VDD** na procesoru alespoň **2V,** spustí se procesor až za určitý čas (tj. **72ms**). Tento blok můžeme chápat jako **takovou pojistku pro ustálení napájecího napětí.** Tento blok lze **de/aktivovat** pomocí **bitu /PWRTE** v konfiguračním slovu **CONFIG => aktivace je /PWRTE = log.0.** Všimněme si, že v případě, kdy využijeme blok **PWRT,** tak nastává **podřizování bloku OST** tzn., že blok **OST čeká na log.1 od bloku PWRT.** V případě, kdy bude **PWRT** deaktivován, je tento blok nefunkční a máme zde situaci naprosto stejnou jako v popisu **OST** tzn., že za **1024period OSC je procesor spuštěn.** V případě, že budeme blok **PWRT** využívat, nastává jiná situace. **Řekli jsme si, že PWRT je časovač.**
Tento časovač si bere signál z **oscilátoru RC, který pracuje nezávisle na oscilátoru pro procesor OSC.** Časovač přeteče za dobu úměrnou **72ms.** Za tuto dobu dává tento
časovač = PWRT úroveň **log.1 hradlům AND.** Tím je **2-vstupé hradlo AND** v činnosti tj. začíná přesun period z **OSC** do **10-bitového čítače** bloku **OST** a **zároveň** se **log.1** přesouvá na vstup **3-vstupého hradla AND =>** po přetečení čítače **(OST)** se vygeneruje na **poslední** vstup **3-vstupového hradla AND log.1 =>** v tomto případě je stav na vstupech hradla **AND (1.1.1) = log.1=> teprve nyní je procesor spuštěn.** Pro názornost si uvedeme tabulku, která vystihuje čas při **ne/použití PWRT,** čímž se **ne/prodlužuje čas pro spuštění bloku OST.**

Typ oscilátoru (OSC)	* nabíhání VDD	
	aktivován PWRT	deaktivován PWRT
XT, HS, LP	72ms + 1024 Tosc	1024 Tosc
RC	72ms	—

*** Uvědomme si, že funkce PWRT je při použití aktivní pouze v době, kdy napájecí napětí VDD začne nabíhat a jeho hodnota je alespoň 2V tj. povolení od bloku POR ! Teprve poté (tj. 2V) začíná blok PWRT pracovat tj. načasování 72ms a tím svoji funkci ukončuje, ale zároveň spouští blok OST tj. 1024 Tosc -> 72ms + 1024Tosc teprve po tomto času se spouští procesor a taktéž blok OST ukončuje svoji funkci.**

Shrnutí

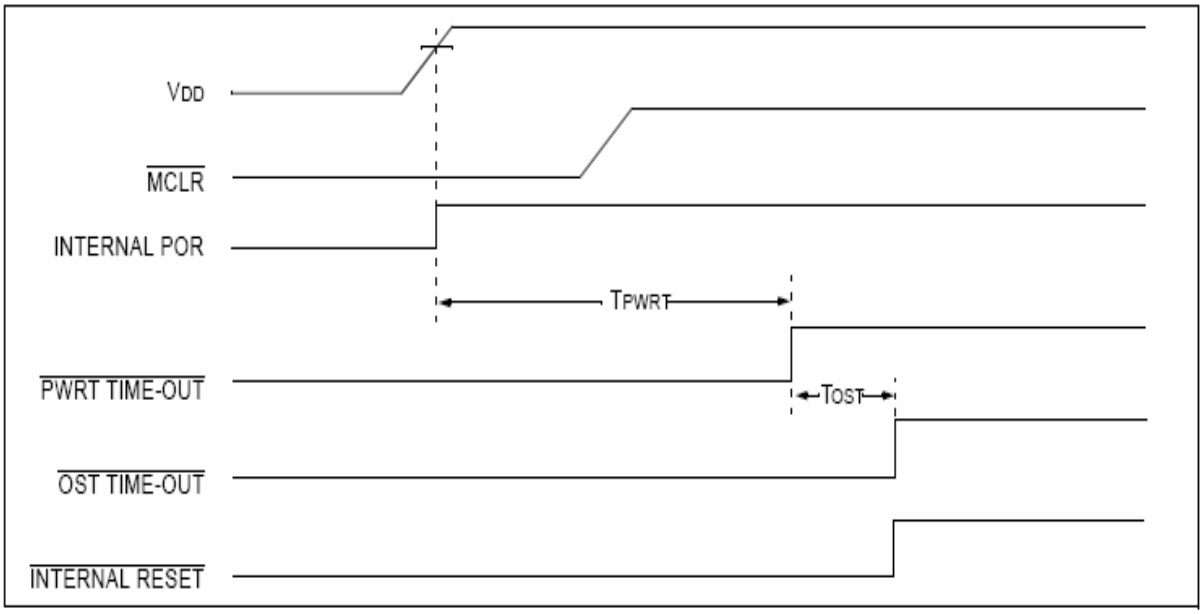
Můžeme říci, že resetování procesoru a jeho následné spouštění je ovlivněno celkem **6. funkčními bloky,** z nichž nemůžeme ovlivnit **OST** a **POR, protože jsou vždy nezbytné při spouštění procesoru !!!** Ostatní bloky tj.**(/MCLR,WDT, /SLEEP, PWRT)** můžeme využít a nebo taky ne. Jen pro připomenutí je dobré **zdůraznit,** že při **použití PWRT** se blok **OST** chová jako **slave** a **PWRT** jako **master.** S těmito **šesti bloky** taktéž souvisí tzv. příznakové bity, které jsou v registru **STATUS.** Jsou to bity **/T0** a **/PD =>** jsou zde možné **4 různé kombinace** z těchto bitů, a proto se podívejme na níže uvedenou tabulku, kde jsou podrobně tyto kombinace popsány.

Tabulka:

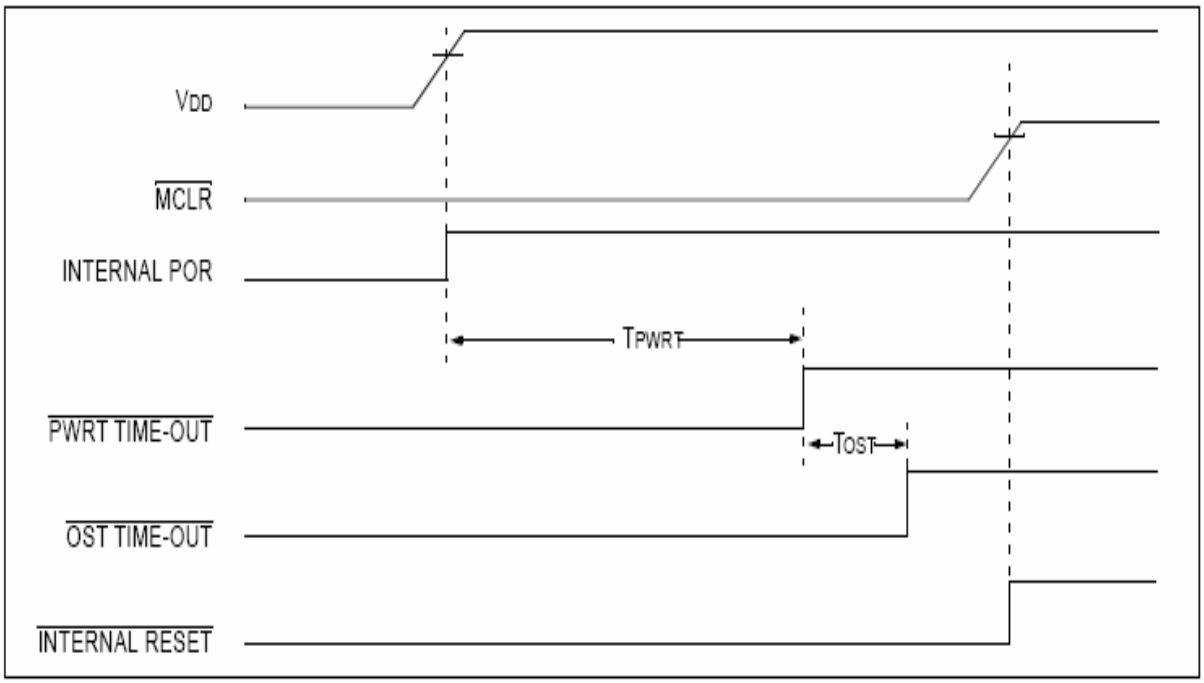
/T0	/PD	Podmínka
0	**0**	**WDT je aktivní,** zároveň používáme **SLEEP mód tj. při přetečení WDT nastane probuzení ze SLEEP módu**
0	**1**	**WDT RESET (přetečení WDT)** během provádění programu tj. operací
1	**0**	**/MCLR RESET** během **SLEEP** módu nebo při **přerušení** nastane **probuzení procesoru** ze **SLEEP módu**
1	**1**	**/MCLR RESET** během provádění programu tj. operací
1	**1**	**Power_on RESET (tento blok je aktivní tj. blokuje při VDD < 2V)**

Pro lepší pochopení se podívejme na 4 grafy, které nám podrobněji ukazují, jak probíhá sekvence impulsů v resetovacím systému za určitých podmínek.

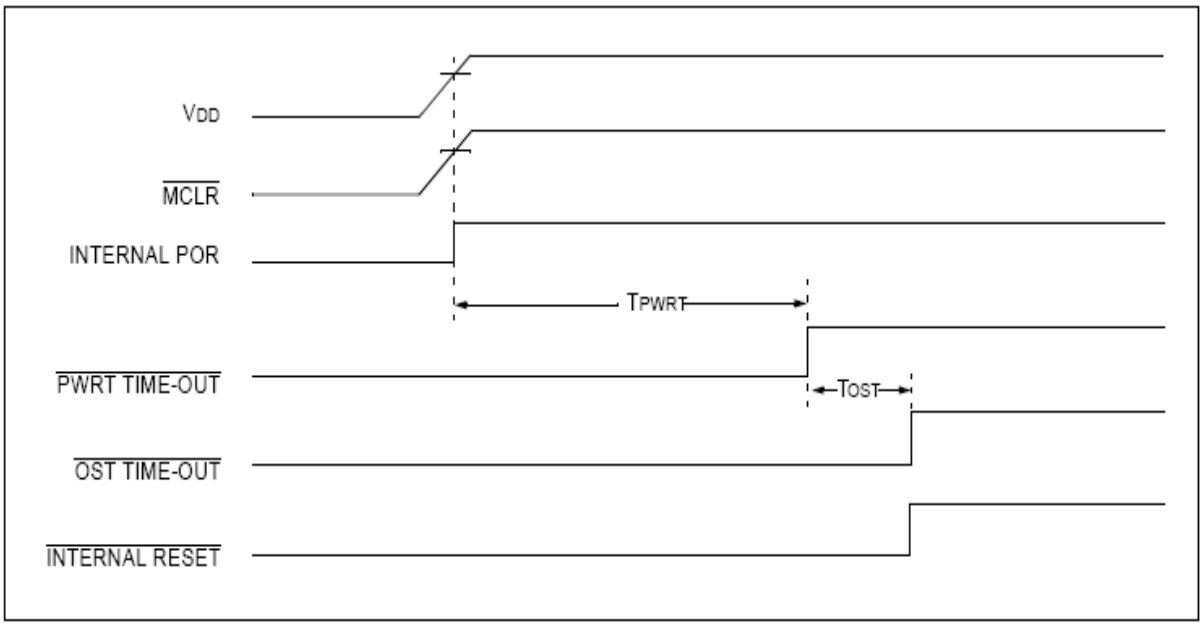
Sekvence impulzů při nabíhání VDD (pin $\overline{\text{MCLR}}$ není napojen na VDD)



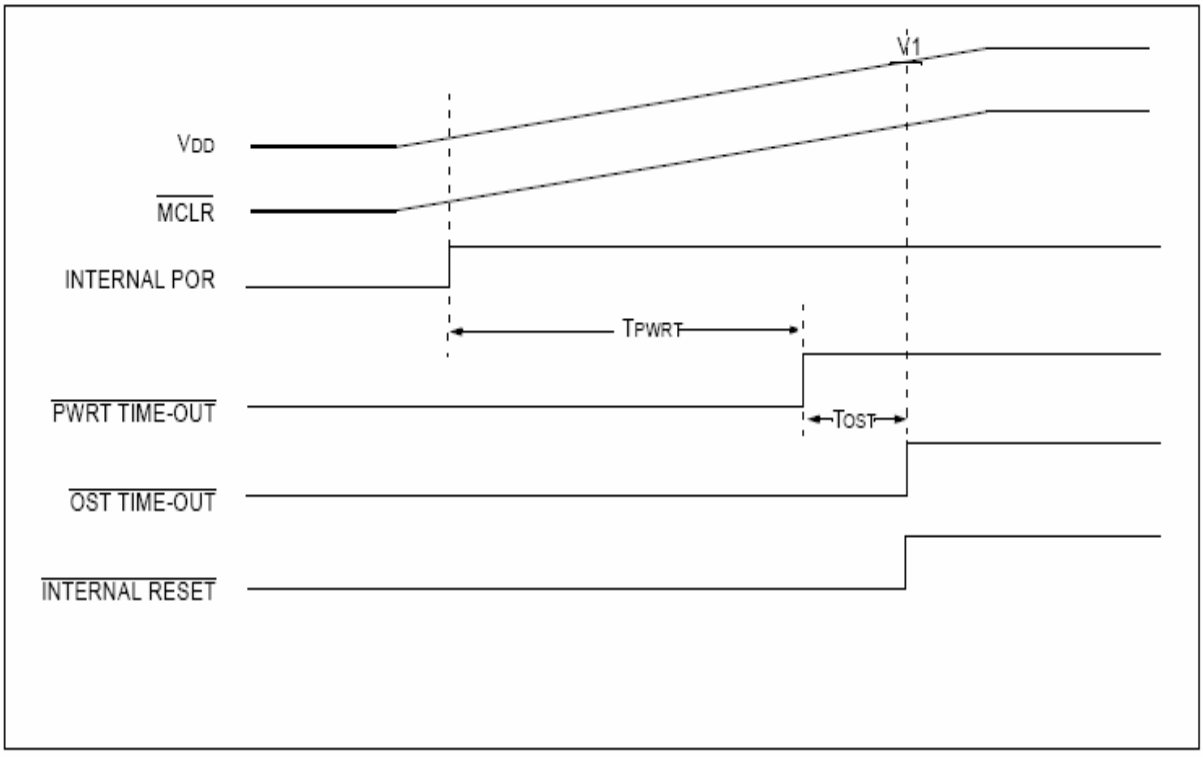
Sekvence impulzů při nabíhání VDD (pin $\overline{\text{MCLR}}$ není napojen na VDD)



Sekvence impulzů při RYCHLÉM nabíhání VDD (pin $\overline{\text{MCLR}}$ je napojen na VDD)



Sekvence impulzů při POMALÉM nabíhání VDD (pin $\overline{\text{MCLR}}$ je napojen na VDD)



Pozn: Na začátku neuvádíme **modul /SLEEP,** protože tento modul nezpůsobuje **RESET procesoru,** nýbrž má zde **blokovácí funkci v případě,** kdy **využíváme současně SLEEP mód a WDT.**

V této publikaci jsme si dali za úkol seznámit čtenáře se základy práce jednočipového mikropočítače PIC16F84, jednoho z nejrozšířenějších v této oblasti. K dokonalejšímu pochopení a zvládnutí práce s touto technikou je vhodná práce se simulačními programy firmy ***Microchip.*** Tyto programy jsou volně ke stažení na www.microchip.com, dále lze získat informace u české firmy na www.asix.cz pod názvem ***MPlab***. Tato firma vyrábí i emulátory pro celou řadu obvodů PIC, včetně programů pro programování těchto mikropočítačů pomocí PC. Další odkazy a návody na konstrukci programátorů těchto obvodů je možné získat v odborné literatuře i na webových stránkách s touto tématikou.

Závěrem bych chtěl poděkovat svému spolupracovníkovi Pavlu Matyášovi za obětavou práci při psaní této příručky a za vytvoření a odzkoušení celé řady programů pro tento mikropočítač. Dalším rozšířením k této problematice by bylo programování mikropočítačů ve vyšších programovacích jazycích a cvičení na emulátoru a simulátoru těchto obvodů. Chtěl bych poděkovat pozorným čtenářům za přečtení tohoto materiálu a požádat je o případné připomínky k tomuto textu.

Ing. Jan Hrdina
j-hrdina@seznam.cz