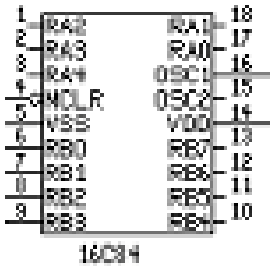
Microcontrolador PIC 16f84

O que são :
Pequeno componente eletrônico dotado de uma inteligência programável

Toda a vez que o microcontrolador é alimentado, o programa interno é executado

O pic 16f84

18 pinos
13 portas configuráveis como entrada ou saída
4 interrupções (tmr0, externa, mudança de estado e eeprom)
Memória flah que permite a gravação do programa diversas vezes no mesmo chip.
Memória EEPROM (Não volátil) interna.
Via de programação com 14 bits e 35 intruções.

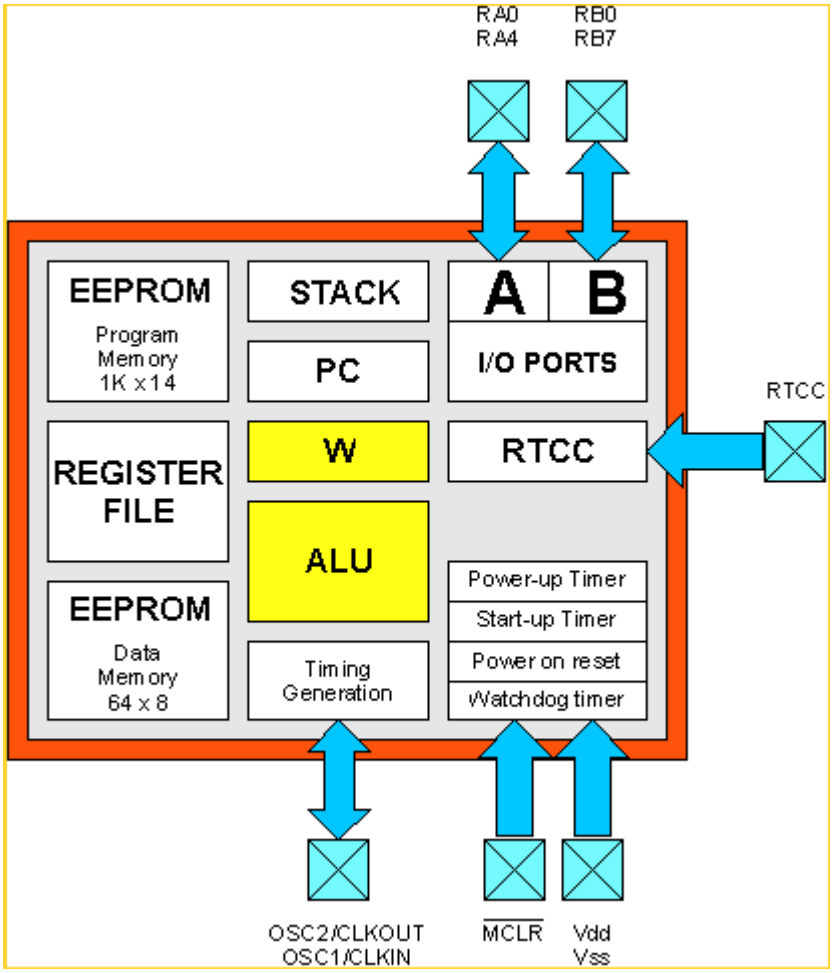


O pic 16f84 possui 13 portas de i/o separada em 2 grupos porta A e porta B

O portA possui 5 pinos que podem ser configurados como entrada ou saída e seus nomes são definidos como RA0, RA1, RA2, RA3 OU RA4. O PINO RA4 TAMBÉM PODE SER USADO PARA INCREMENTAR O CONTADOR TMR0.

O portB possui 8 pinos, configurável como entrada ou saída sendo RB0, RB1, RB2, RB3, RB4, RB5 E RB6.

O RB0 pode ser usado para gerar uma interrupção externa, e RB4 a RB7 podem gerar uma interrupção por mudança de estado.



O microcontrolador funciona com gnd no pino 5 e +5v no pino 14 vdd
Um oscilador deve ser ligado nos pinos OSC1 e OSC2 (pinos 16 e 15)

O pino 4 /mclr qunado colocado em GND (baixo) causará um reset. Com +5v, o programa recomeça d ponto inicial.
A corrente máxima de entrada de um pino é 25 ma
A corrente máxima de saída de um pino é de 20 ma
O portA aceita entrada máxima de 80ma com saída máxima de 50 ma
A portB aceita entrada de no máximo 150ma, com saída de no máximo 100 ma

clock
O microcontrolador pic divide o clock externo por 4.
7

Interrupções do pic

Interrupção de Timer 0

Acontece sempre que um contador de tempo interno, denominado TMR0 estoura, ou seja, sempre que passar de 0xFF para 0X00.

Interrupção Externa

Gerada por sinal externo ligado a uma porta específica do pic, no caso, RB0

Interrupção por mudança de estado

Funciona em uma borda de subida ou borda de descida
Esta interrupção esta ligada a RB4, RB5, RB6 E RB7

Interrupção de fim de escrita na EEPROM
Usado para confirmar quando os dados foram escritos

Como tratá-las ?
Quando acontece uma interrupção, o programa guarda na pilha a proxima linha e desvia para um endereço fixo.

Memória de programa do pic
14 bits – 1024 palavras
O endereço 00h é o vetor de reset
O endereço 04h é o vetor de interrupção
O restante até
3ffh é para uso geral

Memória de dados do pic

Volátil com registradores de 8 bits e 68 bytes, quando o pic é desligado, os seus dados são perdidos.

Bank0		Bank1	
Ender	Registradores	Registradores	Ender
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	trisb	86h
07h	(não usada)	(não usada)	87h
08h	eedata	Eecon1	88h
09h	eeadr	Eecon2	89h
0ah	pclath	Pclath	8ah
0bh	intcon	Intcon	8bh

0ch	uso geral 68 bytes	Espelho do banco 0	8ch
4fh			CFh
50h	não disponivel	não disponivel	D0h
07fh			0FFh

Registradores especiais
Gerais
Status
Endereços 03h e 83h

Bit7	bit6	bit5	bit4	bi3	bit2	bit1	bit0
R/w	R/w	R/w	R	R	R/w	R/w	R/w
Irp	Rp1	Rp0	/t0	/pd	Z	Dc	C

Irp – seletor de banco de memória usado para o endereçamento **indireto**
0 = banco 0 e 1 (00h – ffh)
1= banco 2 e 3 (100h e 1ffh)
este bit não é usado no 16f84 sendo mantido sempre em 0.

Rp1 e rp0 – seletor de banco de memória usado para endereçamento **Direto**
00 – banco 0 (00-7fh)
01 banco 1 (80h ffh)
10 banco 2 (100h –17fh) * não usado
11 banco 3 (180h – 1ffh) * não usado

* o pinc 16f84 só possui o banco 0 e 1 sendo que rp1 será mantido sempre em 0

/t0 : indicação de time-out
0 – indica que ocorreu um estouro de watchDog
1 – indica que ocorreu um power up ou foram executadas as intruções clrwdt ou sleep

/pd : indicação de power-down
0- sleep foi executada
1- indica que ocorreu um power up ou foi executada a intrução clrwdt

z: indicação de zero
0 – a ultima operação não foi zero
1 – a utima operação resultou em zero

DC – digit carry
0 – não houve digit carry - estouro do nibble
1 – houve digit carry entre o bit 3 e 4 -> passou o nibble
usado quando se trabalha com números de 4 bits

C – carry
0 – não houve estouro
1 – houve estouro e ultrapassou os oito bits

regitrador option

Bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
R/w	R/w	R/w	R/w	R/w	R/w	R/w	R/w
/rbpu	Intedg	Tocs	Tose	Psa	Ps2	Ps1	Ps0

/rbpu: Habilitação dos pull-ups internos para o portb:
0=-pulls-ups habilitador para todos os pinos do portb configurados como entrada
1- pull-ups desabilitados

intedg: Configuração de borda que gerará a interrupção externa no rb0:

0 – A interrupção ocorrerá na borda de descida
1 – A interrupção ocorrerá na borda de subida

tocs: Configuração do incremento para tmr0
0 – tmr0 será incrementado pelo clock da máquina
1 - tmr0 será incrementado externamente pela mudança no pino ra4/tock1

tose: Configuração da borda que incrementará o tmr0 no pino ra4/tock1, quando tocs=1
0 – o incremento ocorrerá na borda de subida
1 – o incremento ocorrerá na borda de descida

psa : configuração de aplicação do prescaler:
0 – preescaler será aplicado ao tmr0
1 – será a aplicado a wdt

ps2, ps1 e ps0: configuração do valor de prescaler

bits 2,1,0	tmr0	wdt
000	1:2	1:1
001	1:4	1:2
010	1:8	1:4
011	1:16	1:8
100	1:32	1:16
101	1:64	1:32
110	1:128	1:64
111	1:256	1:128

Registrador Intcon

ESSE REGISTRADOR SERVE PARA CONFIGURAR E IDENTIFICAR AS INTERRUPÇÕES

Bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
R/w	R/w	R/w	R/w	R/w	R/w	R/w	R/w
Gie	Eeie	To1e	Inte	Rbie	T01f	Intf	Rbif

Gie : habilitação das interrupções
0 – desabilitado
1 – habilitado

eeie: habilitação da interrupção de final de escrita na eeprom
0 - não trata

1 – trata

t01e : Interrupção de estouro de tmr0
0 – não trata
1 – trata

inte: interrupção externa de rb0
0 = não trata
1 = trata

rb1e: interrupção de mudança de estado nos pinos rb4 a rb7
0 – não trata
1 – trata

t01f Identificação da interrupção de estouro de tmr0
0 – não ocorreu a interrupção
1 – ocorreu

intf: interrupção externa rb0
0 – não ocorreu
1 ocorreu

rbif: identificação da ocorrencia da int. por mudança de estado em rb4 a rb7

0 – não ocorreu
1 - ocorreu

registradores pcl e pclath

O pcl armazena os 8 bits penos significativos do program counter **(enderecos 02h e 82h)**

O registrador pclath armazena os 5 bits restantes

Como a memória do pic é maior que 256 bytes, não da para acessar tudo com somente 8 bits. Por isso o pclath possui os 5 bits mais altos do pc. Esse registrador é controlado pelo hardware.

portas

conhecendo os tris

Esses registradores servem para confgurar os pinos das portas como entrada ou saída. Quando e colocado “1” em um bit do tris, o pino relacionado é colocado como entrada. 1= entrada 0-= saida.. para configurar o porA deve ser usado o trisa, para portb deve ser usado o trisb

Registrador trisA endereço 85h

Bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
			R/w	R/w	R/w	R/w	R/w
			Ra4	Ra3	Ra2	Ra1	Ra0

Registrador trisB endereço 86h

Bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
R/w	R/w	R/w	R/w	R/w	R/w	R/w	R/w
Rb7	Rb6	Rb5	Rb4	Rb3	Rb2	Rb1	Rb0

Portas

Registrador portA endereço 05h

Bit7	bit6	Bit5	bit4	bit3	bit2	bit1	bit0
			R/w	R/w	R/w	R/w	R/w
			Ra4	Ra3	Ra2	Ra1	Ra0

Registrador portB endereço 06h

Bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
R/w	R/w	R/w	R/w	R/w	R/w	R/w	R/w
Rb7	Rb6	Rb5	Rb4	Rb3	Rb2	Rb1	Rb0

Contadores

TMr0 – incremtno pelo clock da máquina ou sinal externo, seu estouro pode gerar uma interrupção

Conhecendo o wdt

O wdt ou cão de guarda é um contador incrementado por um clock independente. O pic pussui um contador de 18ms interno. Não é acessivel para escrita ou leitura. O programador so pode zerá-lo através de clrwdt. Se der overflow o sistema é resetado.
WDT pode ser desligado durante a gravação do pic
É usado para evitar que um sistema trave.

prescaler

configuração da escala do clock.
Tmr0 e wdt será configurado de acordo com o prescaler
Exemplo wdt estoura em 18ms se configurarmos o prescalres em 1:4 ele estourará em aproximadamente em 72 ms.

EEPROM
EEADR

Registrador que indica o endereço de memória onde ocorrerá a leitura ou escrita na eeprom

EEDATA:
deve ser preenchido com o dado a ser armazenado na escrita ou contem o dado lido.

EECONS :
Registradores de configuração da eeprom eecon1 e eecon2
Eecon1 é responsável pelas operações de escrita e leitura da eeprom

Registrador EECON1 endereço 88h

Bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
			R/w	R/w	R/w	R/s	R/s
			Eeif	Wrerr	Wren	Wr	Rd

*S- setado pelo sistema

Eeif – indica se ocorreu uma interrupção de final de escrita na eeprom (colocadoaqui por não haver espaço no registrador intcon)

Wrerr – indicação de erro de escrita da EEPROM

Wren – habilitação de escrita na eprom
0 – não permite
1 – habilita escrita na EEPROM

wr: cliclo de escrita na eeprom
0 – zerado pelo hardware quando terminou de escever
1 – inicia o ciclo de escrita

rd: ciclo de leitura da eeprom

0 - zerado pelo hardware quando terminou de ler
1 – inicia o ciclo de leitura

EECON2
Não é um registrador verdadeiro – é usado no ciclo de escrita na eeprom por questão de segurança evitando alteração acidental da mesma.

ENDEREÇAMENTO INDIRETO

FSR E INDF

Fsr é um ponteiro para um endereço de memória
Indf – valor do endereçamento indireto

Registrador **FSR**				Endereços : 04h e 84h			
Bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
R/w	R/w	R/w	R/w	R/w	R/w	R/w	R/w
Ponteiro para endereçamento indireto							
Registrador : **INDF**				Endereços: 00h e 80h			
Valor do Endereçamento Indireto							

Bank0				
00	INDF	valor do endereçamento indireto (não é um registro verdadeiro	---- ----	---- ----
01	TMR0	Contador Tmrr de 8 bits	XXXX XXXX	UUUU UUUU
02	PCL	Parte baixa do PC	0000 0000	0000 0000
03	STATUS	Irp-rp1-rp0-/to-/pd-z-dc-c	0001 1XXX	000Q UUUU
04	FSR	PONTEIRO PARA ENDEREÇAMENTO INDIRETO	XXXX XXXX	UUUU UUUU
05	PORTA	- - - -ra4-ra3-ra2-ra1-ra0	---X XXXX	---U UUUU
06	PORTB	Rb7-rb6-rb5-rb4-rb3-rb2-rb1-rb0	XXXX XXXX	UUUU UUUU
07		Não implementado valor sempre 0	---- ----	---- ----
08	EEDATA	Dado para escrita/leitura na eeprom	XXXX XXXX	UUUU UUUU
09	EEADR	Endereço para escrita/leitura na eeprom	XXXX XXXX	UUUU UUUU
0A	PCLATH	Parte alta do pc	---0 0000	---0 0000
0b	INTCON	Gie-eeie-toie-inte-rbie-toif-intf-rbif	0000 000X	0000 000U
BANK1				
80	INDF	valor do endereçamento indireto	---- ----	---- ----
81	OPTION	Rbpu-intedg-tocs-tose-psa-ps2-ps1-ps0	1111 1111	1111 1111
82	PCL	Parte baixa do PC	0000 0000	0000 0000
83	STATUS	Irp-rp1-rp0-/to-/pd-z-dc-c	0001 1XXX	000Q UUUU
84	FSR	PONTEIRO PARA ENDEREÇAMENTO INDIRETO	XXXX XXXX	UUUU UUUU
85	TRISA	- - - -ra4-ra3-ra2-ra1-ra0	---1 1111	---1 1111
86	TRISB	Rb7-rb6-rb5-rb4-rb3-rb2-rb1-rb0	1111 1111	1111 1111
87		Não implementado valor sempre 0	---- ----	---- ----
88	EECON1	- - - eeif-wrerr-wren-w-rd	---0 X000	---0 Q000
89	EECON2	Utilizado na incicialização da escrita na eeprom	---- ----	---- ----
8A	PCLATH	Parte alta do pc	---0 0000	---0 0000
8b	INTCON	Gie-eeie-toie-inte-rbie-toif-intf-rbif	0000 000X	0000 000U

Legenda : x-desconhecido, u-não modificado, q-dependen de outras condições, 1-um , 0-zero

O include pic16f84.inc

```
LIST
; P16F84.INC  Standard Header File, Version 2.00    Microchip Technology, Inc.
        NOLIST

; This header file defines configurations, registers, and other useful bits of
; information for the PIC16F84 microcontroller.  These names are taken to match
; the data sheets as closely as possible.

; Note that the processor must be selected before this file is
; included.  The processor may be selected the following ways:

;       1. Command line switch:
;               C:\ MPASM MYFILE.ASM /PIC16F84
;       2. LIST directive in the source file
;               LIST   P=PIC16F84
;       3. Processor Type entry in the MPASM full-screen interface

;==========================================================================
;
;       Revision History
;
;==========================================================================

;Rev:   Date:    Reason:

;2.00   07/24/96 Renamed to reflect the name change to PIC16F84.
;1.01   05/17/96 Corrected BADRAM map
;1.00   10/31/95 Initial Release

;==========================================================================
;
;       Verify Processor
;
;==========================================================================

        IFNDEF __16F84
           MESSG "Processor-header file mismatch.  Verify selected processor."
        ENDIF

;==========================================================================
;
;       Register Definitions
;
;==========================================================================

W                           EQU     H'0000'
F                           EQU     H'0001'

;----- Register Files------------------------------------------------------

INDF                        EQU     H'0000'
TMR0                        EQU     H'0001'
PCL                         EQU     H'0002'
STATUS                      EQU     H'0003'
FSR                         EQU     H'0004'
PORTA                       EQU     H'0005'
PORTB                       EQU     H'0006'
EEDATA                      EQU     H'0008'
EEADR                       EQU     H'0009'
PCLATH                      EQU     H'000A'
INTCON                      EQU     H'000B'

OPTION_REG                  EQU     H'0081'
TRISA                       EQU     H'0085'
TRISB                       EQU     H'0086'
EECON1                      EQU     H'0088'
EECON2                      EQU     H'0089'

;----- STATUS Bits --------------------------------------------------------

IRP                         EQU     H'0007'
RP1                         EQU     H'0006'
RP0                         EQU     H'0005'
NOT_TO                      EQU     H'0004'
```

```
NOT_PD                          EQU     H'0003'
Z                               EQU     H'0002'
DC                              EQU     H'0001'
C                               EQU     H'0000'

;----- INTCON Bits --------------------------------------------------------

GIE                             EQU     H'0007'
EEIE                            EQU     H'0006'
T0IE                            EQU     H'0005'
INTE                            EQU     H'0004'
RBIE                            EQU     H'0003'
T0IF                            EQU     H'0002'
INTF                            EQU     H'0001'
RBIF                            EQU     H'0000'

;----- OPTION Bits --------------------------------------------------------

NOT_RBPU                        EQU     H'0007'
INTEDG                          EQU     H'0006'
T0CS                            EQU     H'0005'
T0SE                            EQU     H'0004'
PSA                             EQU     H'0003'
PS2                             EQU     H'0002'
PS1                             EQU     H'0001'
PS0                             EQU     H'0000'

;----- EECON1 Bits --------------------------------------------------------

EEIF                            EQU     H'0004'
WRERR                           EQU     H'0003'
WREN                            EQU     H'0002'
WR                              EQU     H'0001'
RD                              EQU     H'0000'

;=========================================================================
;
;       RAM Definition
;
;=========================================================================

        __MAXRAM H'CF'
        __BADRAM H'07', H'50'-H'7F', H'87'

;=========================================================================
;
;       Configuration Bits
;
;=========================================================================

_CP_ON                          EQU     H'000F'
_CP_OFF                         EQU     H'3FFF'
_PWRTE_ON                       EQU     H'3FF7'
_PWRTE_OFF                      EQU     H'3FFF'
_WDT_ON                         EQU     H'3FFF'
_WDT_OFF                        EQU     H'3FFB'
_LP_OSC                         EQU     H'3FFC'
_XT_OSC                         EQU     H'3FFD'
_HS_OSC                         EQU     H'3FFE'
_RC_OSC                         EQU     H'3FFF'

        LIST
```

O conjunto de instruções

Termos Utilizados

Work – registrador acumulador temporário indicado pela letra w

File : referencia a um registrador (posição de memória) usado com a letra f
Literal – um numero qualquer em binário, decimal ou hexadecimal (L- na representação do nome de instruções – k nos argumentos)
Destino – Local onde o resultado de uma operação será armazenado (f- no registrador passado como argumento ou em W – registrador Work) o include se encarrega de converter 0 para W e 1 para F no destino)

Construção de um nome de instrução

Ex:
Decrementar (DEC) um registrador (F) = DECF

Ex:
DECFZ temp,W
Decrementa f (temp) , guardando o resultado em d (W) , salta (S) a próxima linha se o resultado for zero.(Z)

Resumo das Instruções

OPERAÇÕES COM REGISTRADORES (F)		
Instrução	**Argumentos**	**Descrição**

ADDWF	F,d	Soma W e f, guardando o resultado em d
		ex:
		ADDWF FSR,W
		Antes: W=0x17, FSR=0xC2 - depois W=0xD9, FSR=0xC2
		ADDWF FSR,F
		Antes: W=0x17, FSR=0xC2 - depois W=0x17, FSR=0xD9

ANDWF	F,d	Lógica "E" entre W e f guardando o resultado em d. Ex: ADDLW 0X15
CLRF	F	Limpa f. **Ex**. CLRF TRISB (trisb criado com DEFINE)
COMF	F,d	Calcula o complemento de f, guardando o resultado em d.Ex: comf BOTAO,W
DECF	F,d	Decrementa f, guardando o resultado em d. Ex: DECF temp,W
DECFZ	F,d	Decrementa f, guardando o resultado em d, pula a próxima linha se o resultado for zero. DECFZ temp,W
INCF	f,d	Incrementa f, guardando o resultado em d. **Ex:** INCF CONTADOR,W (F=DESTINO) 0 o incremento vai para o registrador temporário W, se for indicado INCF CONTADOR, F, o próprio registrador será incrementado
INCFSZ	f,d	Incrementa f, guardando o resultado em d, pula a próxima linha se o resultado for zero. EX: INCFSZ temp,W
IORWF	f,d	Lógica "OU" entre W e f guardando o resultado em d
MOVF	f,d	Move f para d (cópia). **Ex.** MOVF TEMPO1,W
MOVWF	f	Move W para f (cópia) . **Ex**: MOVWF OPTION_REG
RLF	f,d	Rotaciona f 1 bit para a esquerda
RRF	f,d	Rotaciona f 1 bit para a direita
SUBWF	f,d	Subtrai w de f (f-W), guardando o resultado em d.
XORWF	f,d	Lógia "OU exclusivo" entre W e f, guardando o resultado em d.
OPERAÇÕES COM LITERAIS		
Instrução	**Argumentos**	**Descrição**
ADDLW	K	Soma k com W, guardando o resultado em W. Antes: w contem 0x10 – depois W contem 0x25
ANDLW	K	Lógica "E" entre k e W, guardando o resultado em W.
IORLW	K	Lógica "OU" entre k e W, guardando o resultado em W.
MOVLW	K	Move k para W. **Ex**: movlw B'00010001'
SUBLW	K	Subtrai W de k (k-W), guardando o resultado em W.
XORLW	K	Lógica "OU exclusivo" entre k e W, guardando o resultado em W.
OPERAÇÕES COM BITS		
Instrução	**Argumentos**	**Descrição**
BCF	f,b	Impõe 0 (zero) ao bit b do registrador f
BSF	f,b	Impõe 1 (um) ao bit b do registrador f
BTFSC	f,b	Testa o bit b do registrador f e, pula a próxima linha se ele for 0 (zero).
BTFSS	f,b	Testa o bit b do registrador f e, pula a próxima linha se ele for 1 (Um).
CONTROLES		
Instrução	**Argumentos**	**Descrição**
CLRW	-	Limpa W.
NOP	-	Nenhuma operação (gasta um ciclo de máquina)
CALL	R	Executa a rotina R.
CLRWDT	-	Limpa o WDT para não acontecer o reset.
GOTO	R	Disvia para o ponto R, mudando o PC.
RETFIE	-	Retorna de uma Interrupção
RETLW	k	Retorna de uma rotina, com k em W.
RETURN	-	Retorna de uma rotina, sem afetar W.
SLEEP	-	Coloca o PIC em modo de economia de energia

f-file (referência a um registrador – posição de memória)
d- destino (F – próprio registrador W-registrador Work)
k – uma literal
b- numero do bit
R- nome de uma rotina (label)

Exemplo de uso das instruções

MOVLW

Exemplo:

```
        MOVLW 10
```

Move a constante para o registrador W
(Cuidado com a especificação da base numérica)

MOVF, MOVWF

Exemplos:

1 MOVF temp,W
2 MOVF temp,F
3 MOVWF temp

Efeito em status ou em outros registradores: Seta ou limpa o flag Z (MOVF); nenhum (MOVWF)

Estas instruções movem dados entre registradores e o registrador W.
1- move o valor em temp para W
3- Move de W para temp
2- Move de temp para temp (setando o flag z se for o caso)

Inicializando um registrador Exemplo:

```
temp  equ  0x70
movlw 0x10
    movwf temp
```

ADDWF, ANDWF, IORWF, SUBWF, XORWF

Exemplo:

```
        ADDWF temp,W
        XORWF x,F
```

Efeito em status ou em outros registradores: afeta C, DC, Z (ADDWF, SUBWF); afeta Z (ANDWF, IORWF, XORWF)

Estas instruções fazem a operação especificada no registrador W e outro registrador que você escolher.

O resultado pode ser colocado em W no de volta no registrador original dependendo de como a instrução foi escrita
Por Exemplo:

```
        MOVLW 1         ; põe 1 em W
        ADDWF temp,F            ; adiciona  1 em temp,  armazenando o resultado em temp
; W continua igual a 1 aqui
```

```
ADDWF x,W      ; W=x+W, x não é alterado
```
Quando somando, se o resultado não couber em um byte, o flag C é setado.

A operação de subtração sempre calcula com F-W, então tenha cuidado para que não signifique um W-F. se W contém 3 e x contém 10, fazer uma subtração irá resultar em 7. Quando subtrair, o bit C é um flag de subtransbordamento. Se o resultado for negativo, então C irá ser igual a 0. Se o flag C igual a 1, nenhum subtransbordamento ocorreu. Então, ao calcular 10-3 deixa C=1. Mas calcular 3-10 irá limpar C. O resultado, neste caso será 0xf9 que é a representação em complemento de dois para o número -7.

Para calcular o complemento de dois de um número negativo, escreva a magnitude como um número binário. Para 8, você terá 00000111. Então inverta todos os bits (11111000) e some 1 (11111001).

Todas as instruções nesta seção setam o flag z se o resultado é zero. senão o flag Z é limpo..

ADDLW, ANDLW, IORLW, SUBLW, XORLW

Exemplo:

```
ADDLW .30
XORLW 0x80
```
Efeito em status ou em outros registradores Modifica C, DC, Z (ADDLW, SUBLW); modifica Z (ANDLW, IORLW, XORLW)

Estas instruções agem como adição, subtração, E, OU inclusivo e OU Exclusivo Exceto quando operando com literais:

```
ADDLW 2
```
Adiciona 2 em W (e, é claro, deixa o resultado em W) A instrução de subtração calcula a literal com W e pode confundir. Ao invés de escrever:

```
SUBLW 2  ; Calcula 2-W
```
Você pode preferir escrever

```
ADDLW -2  ; Isto é o que você provavelmente irá preferir escrever
```
Adicionando 2 negativo irá dar o resultado esperado sem passos desnecessários.

CLRF, CLRW

Exemplo:

```
CLRF temp
CLRW
```
Efeito em Status. Afeta o flag Z setando-o em 1

Esta é uma operação muito comum para carregar 0 em um registrador e esta instrução pode fazer isto em uma única etapa. A instrução CLRW limpa W, que também pode ser feito por MOVLW 0 . CLRW e MOVLW 0 gastam o mesmo espaço de memória de programa. A única diferença é que MOVLW não irá afetar o flag Z, enquanto CLRW afetará.

COMF

Exemplo:

```
COMF temp,W
```

Efeito em status ou em outros registradores Altera Z

Esta instrução inverte os bits no registrador especificado. Você pode armazenar o resultado em W(,W) ou de volta no registrador (,F). O flag Z será setado se o resultado for zero (ou limpo caso contrário) .. Se você deseja inverter os bits do registrador W, execute XORLW 0xFF que tem o mesmo efeito.

INCF, DECF

Exemplo:

```
        INCF temp,W
        DECF x,F
```

Efeito em status ou em outros registradores Altera Z

Outra operação comum é adicionar ou subtrair 1 de um registrador. (INCF +1; DECF -1)..O resultado pode ser armazenado em W o colocado de volta no registrador. Se você deseja incrementar ou decrementar W, tente ADDLW 1 ou ADDLW -1.

INCFSZ, DECFSZ

Exemplo:

```
        INCFSZ temp,W
        DECFSZ x,F
```

Efeito em Status: Pode alterar o contador de programa

Estas instruções são similares a INCF eDECF. Elas adicionam ou subtraem um do registrador especificado e armazenam o resultado como direcionado. Estas instruções não afetam o flag Z. Mas, se o resultado é zero, saltam a próxima instrução. Isto é muito útil é claro, para laços. Veja o exemplo:

```
        CLRF  y
        MOVLW .10  ;
        MOVWF i          ; i=10
        MOVF x,W   ; W=X
LOOP:   ADDWF y,F  ; Y=Y+X
        DECFSZ i,F
        GOTO LOOP
; Sair aqui com a resposta em y
```

Você pode adivinhar o que este laço faz ?

NOP

Exemplo:

```
        NOP
```

Efeito em status ou em outros registradores: Nenhum

Esta instrução não faz nada. Mas é utilizada em loops de temporização para gastar tempo. Se estiver usando muito destes, você pode achar melhor usar outras maneiras para gastar tempo. Exemplo : Goto $+1 irá gastar 2 ciclos de tempo.

RLF, RRF

Exemplo:

```
        RLF temp,F
```
Efeito em Status: modifica C

Esta instrução move os bits no registrador 1 bit para a esquerda (RLF) ou direita (RRF). O bit extra que é deslocado vem do flag C e o bit deslocado para fora vai para o flag C Desse modo, se você setar C=1 e fazer um RLF em um registrador que contem 0x81, o resultado será 0x03 e o flag C será setado.

Esta instrução é muito usada para manipular bits, mas tem outra propriedade interessante : deslocar a esquerda um byte multiplica um numero por 2 e deslocar a direita divide um número por 2! Pode-se combinar esta propriedade com a adição para conseguir facilmente multiplicações. Por Exemplo, Suponha que você deseja calcular y=10*x. Bem , isto é o mesmo que y=8*x+2*x, certo ? então tente isto:

```
        BCF status,C    ; C=0
        RLF x,F         ; X=2*X
        BCF status,C
        RLF x,W                 ; W=2*X (então  4*X)
        MOVWF temp
        BCF status,C
        RLF temp,W      ; W=2*X (então  8*X)
        ADDWF x,W       ; W=8X+2X = 10X
        MOVWF y
```
Esta não é uma forma fácil de combinar deslocamentos a direita para divisão, mas se você precisa dividir por 2, 4, 8, 16, etc. então você apenas precisa combinar deslocamentos.

SWAPF

Exemplo:

```
        SWAPF temp,w
```
Efeito em status ou em outros registradores: nenhum

Este comando move os 4 bits mais baixos do registrador para os 4 bits mais altos e os 4 mais altos para os 4 mais baixos. Você pode colocar o resultado de volta no registrador ou em W. Se um registrador contém, digamos, 0xA5 e você executa SWAPF, o resultado será 0x5A.

Esta instrução pode ser útil ao trabalhar com números BCD ou Hexa. Entretanto, pode ser útil quando você quer mover um registrador para ou de W sem afetar nenhum flag ::

```
        SWAPF temp,F
        SWAPF temp,W  ; move temp para W com nenhuma mudança de flags
; Se você deseja que temp recupere o seu resultado, adicione:
        SWAPF temp,F
```

BCF, BSF

Exemplo:

```
        BCF status,Z
```
Efeito em Status: nenhum

Esta instrução limpa (BCF) ou seta (BSF) o bit indicado em um registrador. O bit pode ter um nome (como Z) ou você pode usar um número de 0 a 7 (0 é o menos significativo e 7 o mais significativo). Às vezes você poderá usar isto para economizar algumas instruções. Por Exemplo, Suponha que você escreveu:

```
MOVLW 0x7F
ANDWF temp,F
```

Você pode trocar isto por uma instrução simples BCF temp,7 . Não é apenas mais rápido e toma menos espaço. Mas também não destrói o valor do registrador W!

GOTO

Exemplo:

```
GOTO main
```

Efeito em Status: Nenhum

Como você deve saber, a instrução GOTO força seu programa a seguir a execução para o ponto indicado.

CALL, RETURN

Exemplo:

```
CALL hexout
RETURN
```

Efeito em status ou em outros registradores: Nenhum

Um CALL é similar a GOTO com uma grande exceção: o valor do contador de programa é salvo em uma pilha interna. Quando o programa encontrar um RETURN, a seqüência de execução será devolvida para a próxima instrução após CALL A pilha interna do PIC16f84 tem 8 níveis de profundidade.

Assim como GOTO, a instrução CALL pega a parte alta do endereço do registrador PCLATH.

RETLW

Exemplo:

```
RETLW .99
```

Efeito em Status: nenhum

RETLW é exatamente igual a RETURN exceto pelo fato de que carrega um valor literal em W antes de retornar.

RETFIE

Exemplo:

```
RETFIE
```

Efeito em Status. Seta GIE

RETFIE funciona exatamente igual a um return, mas também seta a habilitação geral de interrupção (global interrupt enable) (GIE). Quando uma interrupção de hardware ocorre, ela limpa o GIE e executa o conteúdo apontado por uma instrução CALL. O uso de RETFIE permite a reabilitação das interrupções e o retorno ao

programa principal, tudo em um único passo. Se você não deseja reabilitar as interrupções, simplesmente execute um RETURN.

BTFSC, BTFSS

Exemplo:

```
    BTFSS temp,7
```

Efeito em status ou em outros registradores: Modifica o contador de programa

Quando você desejar fazer um salto condicional, você precisará de uma destas instruções. Elas testam um bit e saltam para a próxima instrução se o bit estiver setado (BTFSS) ou limpo (BTFSC). Na maioria das vezes a próxima instrução é um goto ou um call, mas pode ser também uma instrução simples. Por Exemplo, este código testa temp para ver se é zero. Se é, então carrega temp com o valor 0x80. Caso contrário, temo não é alterado:

```
MOVF temp,F     ; seta o flag  Z , nenhum dado é realmente movido
BTFSC status,Z  ; teste o bit Z de status e pule próxima instrução se não for zero
BSF temp,7      ; se temp era zero, agora é 0x80!
```

CLRWDT

Exemplo:

```
    CLRWDT
```

Efeito em status ou em outros registradores: nenhum

Esta instrução informa ao timer do watchdog que o seu programa continua executando. Se você não estiver com o watchdog habilitado, você não precisa desta instrução

SLEEP

Exemplo:

```
    SLEEP
```

Efeito em status ou em outros registradores: nenhum

Colocar o processador para dormir permite que o consumo de energia caia muito., mas você irá precisar de uma interrupção ou de um reset para acordar.

Observe!

Alguns processadores PIC tem significativas diferenças no seu conjunto de instruções. Também alguns compiladores tratam as instruções de outra forma para os mesmos mnemônicos. Este tutorial foca no padrão Microchip de mnemônicos

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                       BOTÃO E LED - EX1                               *
;*                       DESBRAVANDO O PIC                               *
;*       DESENVOLVIDO PELA MOSAICO ENGENHARIA E CONSULTORIA              *
;*      VERSÃO: 1.0                              DATA: 11/06/99          *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                      DESCRIÇÃO DO ARQUIVO                             *
;*-------------------------------------------------------------------*
;*  SISTEMA MUITO SIMPLES PARA REPRESENTAR O ESTADO DE                  *
;*  UM BOTÃO ATRAVÉS DE UM LED.                                         *
;*                                                                      *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                     ARQUIVOS DE DEFINIÇÕES                           *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

#INCLUDE <P16F84.INC>         ;ARQUIVO PADRÃO MICROCHIP PARA 16F84

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                    PAGINAÇÃO DE MEMÓRIA                              *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;DEFINIÇÃO DE COMANDOS DE USUÁRIO PARA ALTERAÇÃO DA PÁGINA DE MEMÓRIA

#DEFINE     BANK0 BCF STATUS,RP0    ;SETA BANK 0 DE MEMÓRIA
#DEFINE     BANK1 BSF STATUS,RP0    ;SETA BANK 1 DE MAMÓRIA

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                         VARIÁVEIS                                    *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; DEFINIÇÃO DOS NOMES E ENDEREÇOS DE TODAS AS VARIÁVEIS UTILIZADAS
; PELO SISTEMA

      CBLOCK      0x0C        ;ENDEREÇO INICIAL DA MEMÓRIA DE
                              ;USUÁRIO

            W_TEMP            ;REGISTRADORES TEMPORÁRIOS PARA
            STATUS_TEMP ;INTERRUPÇÕES
                        ;ESTAS VARIÁVEIS NEM SERÃO UTI-
                        ;LIZADAS
      ENDC                    ;FIM DO BLOCO DE MEMÓRIA

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                        FLAGS INTERNOS                                *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; DEFINIÇÃO DE TODOS OS FLAGS UTILIZADOS PELO SISTEMA

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                         CONSTANTES                                   *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; DEFINIÇÃO DE TODAS AS CONSTANTES UTILIZADAS PELO SISTEMA

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                          ENTRADAS                                    *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; DEFINIÇÃO DE TODOS OS PINOS QUE SERÃO UTILIZADOS COMO ENTRADA
; RECOMENDAMOS TAMBÉM COMENTAR O SIGNIFICADO DE SEUS ESTADOS (0 E 1)
```

```
#DEFINE     BOTAO PORTA,2     ;PORTA DO BOTÃO
                  ; 0 -> PRESSIONADO
                  ; 1 -> LIBERADO

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                            SAÍDAS                                     *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; DEFINIÇÃO DE TODOS OS PINOS QUE SERÃO UTILIZADOS COMO SAÍDA
; RECOMENDAMOS TAMBÉM COMENTAR O SIGNIFICADO DE SEUS ESTADOS (0 E 1)

#DEFINE     LED   PORTB,0     ;PORTA DO LED
                  ; 0 -> APAGADO
                  ; 1 -> ACESO

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                         VETOR DE RESET                                *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

      ORG   0x00  ;ENDEREÇO INICIAL DE PROCESSAMENTO
      GOTO  INICIO

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                    INÍCIO DA INTERRUPÇÃO                              *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; AS INTERRUPÇÕES NÃO SERÃO UTILIZADAS, POR ISSO PODEMOS SUBSTITUIR
; TODO O SISTEMA EXISTENTE NO ARQUIVO MODELO PELO APRESENTADO ABAIXO
; ESTE SISTEMA NÃO É OBRIGATÓRIO, MAS PODE EVITAR PROBLEMAS FUTUROS

      ORG   0x04        ;ENDEREÇO INICIAL DA INTERRUPÇÃO
      RETFIE                  ;RETORNA DA INTERRUPÇÃO

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                     INICIO DO PROGRAMA                                *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

INICIO
      BANK1             ;ALTERA PARA O BANCO 1
      MOVLW B'00000100'
      MOVWF TRISA       ;DEFINE RA2 COMO ENTRADA E DEMAIS
                        ;COMO SAÍDAS
      MOVLW B'00000000'
      MOVWF TRISB       ;DEFINE TODO O PORTB COMO SAÍDA
      MOVLW B'10000000'
      MOVWF OPTION_REG  ;PRESCALER 1:2 NO TMR0
                        ;PULL-UPS DESABILITADOS
                        ;AS DEMAIS CONFG. SÃO IRRELEVANTES
      MOVLW B'00000000'
      MOVWF INTCON            ;TODAS AS INTERRUPÇÕES DESLIGADAS
      BANK0             ;RETORNA PARA O BANCO 0

 ;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                    INICIALIZAÇÃO DAS VARIÁVEIS                        *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


      CLRF  PORTA       ;LIMPA O PORTA
      CLRF  PORTB       ;LIMPA O PORTB
```

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                     ROTINA PRINCIPAL                                  *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
MAIN

      BTFSC BOTAO       ;O BOTÃO ESTÁ PRESSIONADO?
      GOTO  BOTAO_LIB   ;NÃO, ENTÃO TRATA BOTÃO LIBERADO
      GOTO  BOTAO_PRES  ;SIM, ENTÃO TRATA BOTÃO PRESSIONADO

BOTAO_LIB
      BCF   LED         ;APAGA O LED
      GOTO  MAIN        ;RETORNA AO LOOP PRINCIPAL

BOTAO_PRES
      BSF   LED         ;ACENDE O LED
      GOTO  MAIN        ;RETORNA AO LOOP PRINCIPAL

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                       FIM DO PROGRAMA                                 *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
      END               ;OBRIGATÓRIO
```

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                   CONTADOR MELHORADO - EX4                            *
;*                      DESBRAVANDO O PIC                                *
;*       DESENVOLVIDO PELA MOSAICO ENGENHARIA E CONSULTORIA              *
;*      VERSÃO: 1.0                              DATA: 11/06/99          *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                      DESCRIÇÃO DO ARQUIVO                             *
;*-------------------------------------------------------------------*
;*  CONTADOR QUE UTILIZA DOIS BOTÕES PARA INCREMENTAR E DECRE-          *
;*  MENTAR O VALOR CONTROLADO PELA VARIÁVEL "CONTADOR". ESTA            *
;*  VARIÁVEL ESTÁ LIMITADA PELAS CONSTANTES "MIN" E "MAX".              *
;*  O VALOR DO CONTADOR É MOSTRADO NO DISPLAY.                          *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                     ARQUIVOS DE DEFINIÇÕES                            *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

#INCLUDE <P16F84.INC>         ;ARQUIVO PADRÃO MICROCHIP PARA 16F84

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                     PAGINAÇÃO DE MEMÓRIA                              *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;DEFINIÇÃO DE COMANDOS DE USUÁRIO PARA ALTERAÇÃO DA PÁGINA DE MEMÓRIA

#DEFINE     BANK0 BCF STATUS,RP0    ;SETA BANK 0 DE MEMÓRIA
#DEFINE     BANK1 BSF STATUS,RP0    ;SETA BANK 1 DE MAMÓRIA

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
;*                          VARIÁVEIS                                  *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;
; DEFINIÇÃO DOS NOMES E ENDEREÇOS DE TODAS AS VARIÁVEIS UTILIZADAS
; PELO SISTEMA

      CBLOCK      0x0C        ;ENDEREÇO INICIAL DA MEMÓRIA DE
                              ;USUÁRIO

            W_TEMP            ;REGISTRADORES TEMPORÁRIOS PARA
            STATUS_TEMP ;INTERRUPÇÕES
                        ;ESTAS VARIÁVEIS NEM SERÃO UTI-
                        ;LIZADAS
            CONTADOR    ;ARMAZENA O VALOR DA CONTAGEM
            FLAGS       ;ARMAZENA OS FLAGS DE CONTROLE
            FILTRO1           ;FILTRAGEM PARA O BOTÃO 1
            FILTRO2           ;FILTRAGEM PARA O BOTÃO 2

      ENDC              ;FIM DO BLOCO DE MEMÓRIA

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                        FLAGS INTERNOS                               *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; DEFINIÇÃO DE TODOS OS FLAGS UTILIZADOS PELO SISTEMA

#DEFINE     ST_BT1      FLAGS,0           ;STATUS DO BOTÃO 1
#DEFINE     ST_BT2      FLAGS,1           ;STATUS DO BOTÃO 2

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                          CONSTANTES                                 *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; DEFINIÇÃO DE TODAS AS CONSTANTES UTILIZADAS PELO SISTEMA

MIN         EQU   .0    ;VALOR MÍNIMO PARA O CONTADOR
MAX         EQU   .15   ;VALOR MÁXIMO PARA O CONTADOR
T_FILTRO    EQU   .255  ;FILTRO PARA BOTÃO

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                           ENTRADAS                                  *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; DEFINIÇÃO DE TODOS OS PINOS QUE SERÃO UTILIZADOS COMO ENTRADA
; RECOMENDAMOS TAMBÉM COMENTAR O SIGNIFICADO DE SEUS ESTADOS (0 E 1)

#DEFINE     BOTAO1      PORTA,1     ;PORTA DO BOTÃO
                  ; 0 -> PRESSIONADO
                  ; 1 -> LIBERADO

#DEFINE     BOTAO2      PORTA,2     ;PORTA DO BOTÃO
                  ; 0 -> PRESSIONADO
                  ; 1 -> LIBERADO

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                           SAÍDAS                                    *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; DEFINIÇÃO DE TODOS OS PINOS QUE SERÃO UTILIZADOS COMO SAÍDA
; RECOMENDAMOS TAMBÉM COMENTAR O SIGNIFICADO DE SEUS ESTADOS (0 E 1)


;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                         VETOR DE RESET                              *
```

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

      ORG   0x00  ;ENDEREÇO INICIAL DE PROCESSAMENTO
      GOTO  INICIO

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                     INÍCIO DA INTERRUPÇÃO                            *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; AS INTERRUPÇÕES NÃO SERÃO UTILIZADAS, POR ISSO PODEMOS SUBSTITUIR
; TODO O SISTEMA EXISTENTE NO ARQUIVO MODELO PELO APRESENTADO ABAIXO
; ESTE SISTEMA NÃO É OBRIGATÓRIO, MAS PODE EVITAR PROBLEMAS FUTUROS

      ORG   0x04        ;ENDEREÇO INICIAL DA INTERRUPÇÃO
      RETFIE                  ;RETORNA DA INTERRUPÇÃO

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                  ROTINA DE CONVERSÃO BINÁRIO -> DISPLAY              *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
; ESTA ROTINA IRÁ RETORNAR EM W, O SIMBOLO CORRETO QUE DEVE SER
; MOSTRADO NO DISPLAY PARA CADA VALOR DE CONTADOR. O RETORNO JÁ ESTÁ
; FORMATADO PARA AS CONDIÇÕES DE LIGAÇÃO DO DISPLAY AO PORTB.
;        a
;     **********
;     *        *
;   f *        * b
;     *    g   *
;     **********
;     *        *
;   e *        * c
;     *    d   *
;     **********  *.

CONVERTE
      MOVF  CONTADOR,W  ;COLOCA CONTADOR EM W
      ANDLW B'00001111' ;MASCARA VALOR DE CONTADOR
                        ;CONSIDERAR SOMENTE ATÉ 15
      ADDWF PCL,F

;           B'EDC.BAFG' ; POSIÇÃO CORRETA DOS SEGUIMENTOS
      RETLW B'11101110' ; 00 - RETORNA SÍMBOLO CORRETO 0
      RETLW B'00101000' ; 01 - RETORNA SÍMBOLO CORRETO 1
      RETLW B'11001101' ; 02 - RETORNA SÍMBOLO CORRETO 2
      RETLW B'01101101' ; 03 - RETORNA SÍMBOLO CORRETO 3
      RETLW B'00101011' ; 04 - RETORNA SÍMBOLO CORRETO 4
      RETLW B'01100111' ; 05 - RETORNA SÍMBOLO CORRETO 5
      RETLW B'11100111' ; 06 - RETORNA SÍMBOLO CORRETO 6
      RETLW B'00101100' ; 07 - RETORNA SÍMBOLO CORRETO 7
      RETLW B'11101111' ; 08 - RETORNA SÍMBOLO CORRETO 8
      RETLW B'01101111' ; 09 - RETORNA SÍMBOLO CORRETO 9
      RETLW B'10101111' ; 10 - RETORNA SÍMBOLO CORRETO A
      RETLW B'11100011' ; 11 - RETORNA SÍMBOLO CORRETO b
      RETLW B'11000110' ; 12 - RETORNA SÍMBOLO CORRETO C
      RETLW B'11101001' ; 13 - RETORNA SÍMBOLO CORRETO d
      RETLW B'11000111' ; 14 - RETORNA SÍMBOLO CORRETO E
      RETLW B'10000111' ; 15 - RETORNA SÍMBOLO CORRETO F

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                     INICIO DO PROGRAMA                               *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
INICIO
      BANK1             ;ALTERA PARA O BANCO 1
      MOVLW B'00000110'
      MOVWF TRISA       ;DEFINE RA1 E 2 COMO ENTRADA E DEMAIS
                        ;COMO SAÍDAS
      MOVLW B'00000000'
      MOVWF TRISB       ;DEFINE TODO O PORTB COMO SAÍDA
      MOVLW B'10000000'
      MOVWF OPTION_REG  ;PRESCALER 1:2 NO TMR0
                        ;PULL-UPS DESABILITADOS
                        ;AS DEMAIS CONFG. SÃO IRRELEVANTES
      MOVLW B'00000000'
      MOVWF INTCON            ;TODAS AS INTERRUPÇÕES DESLIGADAS
      BANK0             ;RETORNA PARA O BANCO 0

 ;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                     INICIALIZAÇÃO DAS VARIÁVEIS                 *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

      CLRF  PORTA       ;LIMPA O PORTA
      CLRF  PORTB       ;LIMPA O PORTB
      CLRF  FLAGS       ;LIMPA TODOS OS FLAGS
      MOVLW MIN
      MOVWF CONTADOR    ;INICIA CONTADOR = MIN
      GOTO  ATUALIZA    ;ATUALIZA O DISPLAY INICIALMENTE

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                     ROTINA PRINCIPAL                            *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
MAIN
      MOVLW T_FILTRO
      MOVWF FILTRO1           ;INICIALIZA FILTRO1 = T_FILTRO
      MOVWF FILTRO2           ;INICIALIZA FILTRO2 = T_FILTRO

CHECA_BT1
      BTFSC BOTAO1            ;O BOTÃO 1 ESTÁ PRESSIONADO?
      GOTO  BT1_LIB           ;NÃO, ENTÃO TRATA COMO LIBERADO
                        ;SIM
      DECFSZ      FILTRO1,F   ;DECREMENTA O FILTRO DO BOTÃO
                        ;TERMINOU?
      GOTO  CHECA_BT1   ;NÃO, CONTINUA ESPERANDO
                        ;SIM
      BTFSS ST_BT1            ;BOTÃO JÁ ESTAVA PRESSIONADO?
      GOTO  DEC         ;NÃO, EXECUTA AÇÃO DO BOTÃO
      GOTO  CHECA_BT2   ;SIM, CHECA BOTÃO 2

BT1_LIB
      BCF   ST_BT1            ;MARCA BOTÃO 1 COMO LIBERADO

CHECA_BT2
      BTFSC BOTAO2            ;O BOTÃO 2 ESTÁ PRESSIONADO?
      GOTO  BT2_LIB           ;NÃO, ENTÃO TRATA COMO LIBERADO
                        ;SIM
      DECFSZ      FILTRO2,F   ;DECREMENTA O FILTRO DO BOTÃO
                        ;TERMINOU?
      GOTO  CHECA_BT2   ;NÃO, CONTINUA ESPERANDO
                        ;SIM
      BTFSS ST_BT2            ;BOTÃO JÁ ESTAVA PRESSIONADO?
```

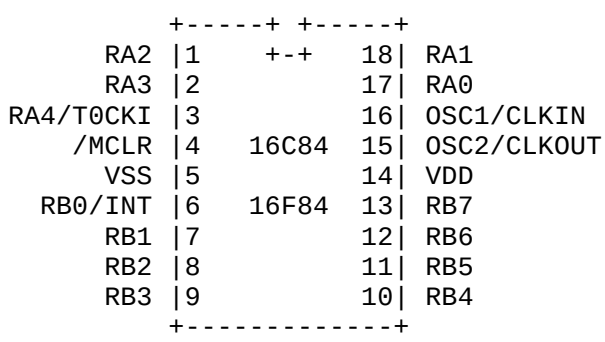
```
      GOTO  INC         ;NÃO, EXECUTA AÇÃO DO BOTÃO
      GOTO  MAIN        ;SIM, VOLTA AO LOOPING

BT2_LIB
      BCF   ST_BT2            ;MARCA BOTÃO 2 COMO LIBERADO
      GOTO  MAIN        ;RETORNA AO LOOPING

DEC                     ;AÇÃO DE DECREMENTAR
      BSF   ST_BT1            ;MARCA BOTÃO 1 COMO JÁ PRESSIONADO
      MOVF  CONTADOR,W  ;COLOCA CONTADOR EM W
      XORLW MIN         ;APLICA XOR ENTRE CONTADOR E MIN
                        ;PARA TESTAR IGUALDADE. SE FOREM
                        ;IGUAIS, O RESULTADO SERÁ ZERO
      BTFSC STATUS,Z    ;RESULTOU EM ZERO?
      GOTO  MAIN        ;SIM, RETORNA SEM AFETAR CONT.
                        ;NÃO
      DECF  CONTADOR,F  ;DECREMENTA O CONTADOR
      GOTO  ATUALIZA    ;ATUALIZA O DISPLAY

INC                     ;AÇÃO DE INCREMENTAR
      BSF   ST_BT2            ;MARCA BOTÃO 2 COMO JÁ PRESSIONADO
      MOVF  CONTADOR,W  ;COLOCA CONTADOR EM W
      XORLW MAX         ;APLICA XOR ENTRE CONTADOR E MAX
                        ;PARA TESTAR IGUALDADE. SE FOREM
                        ;IGUAIS, O RESULTADO SERÁ ZERO
      BTFSC STATUS,Z    ;RESULTOU EM ZERO?
      GOTO  MAIN        ;SIM, RETORNA SEM AFETAR CONT.
                        ;NÃO
      INCF  CONTADOR,F  ;INCREMENTA O CONTADOR
      GOTO  ATUALIZA    ;ATUALIZA O DISPLAY

ATUALIZA
      CALL  CONVERTE    ;CONVERTE CONTADOR NO NÚMERO DO
                        ;DISPLAY
      MOVWF PORTB       ;ATUALIZA O PORTB PARA
                        ;VISUALIZARMOS O VALOR DE CONTADOR
                        ;NO DISPLAY
      GOTO  MAIN        ;NÃO, VOLTA AO LOOP PRINCIPAL

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                       FIM DO PROGRAMA                                 *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

      END               ;OBRIGATÓRIO
```

My First PIC Project

David Tait
david.tait@man.ac.uk

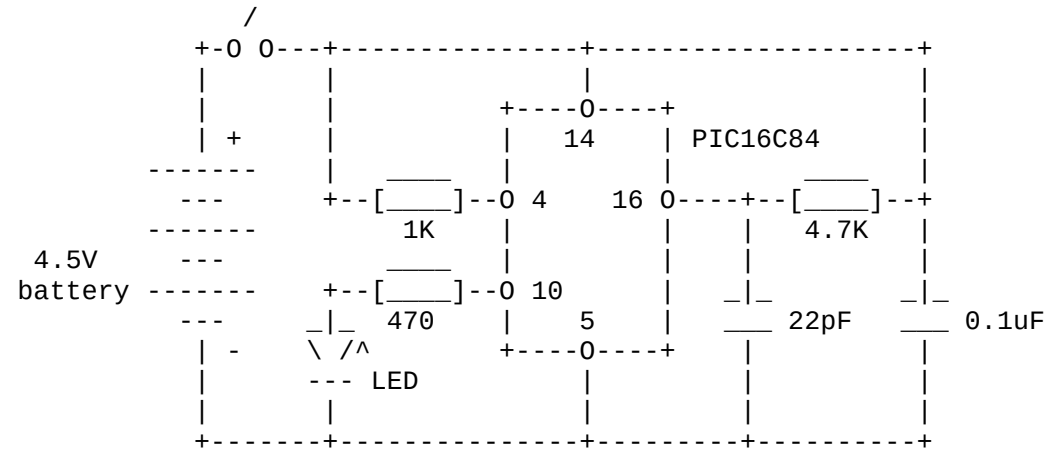
The PIC16C84 (or PIC16F84) from Microchip is a really great little processor. Being based on EEPROM (or "flash") technology means that it can be programmed in a matter of seconds and typically it can be reprogrammed around 1000 times. Of its 18 pins 13 can be used as general purpose I/O. When programmed as outputs the I/O pins are able to source 20mA and sink 25mA (more than enough to drive LEDs directly for Exemplo). It is inexpensive and can be programmed with simple DIY hardware. Obviously these features make the '84 attractive for many projects but they also mean that it is an ideal processor for anyone wanting to learn about microcontrollers.

This short document is meant for people who have just built or purchased a PIC programmer and are itching to get their '84 doing something if only to convince themselves that their programmer, PIC or both are working. To do this we obviously need to lash together some simple hardware and this means knowing a little about the PIC. Here's a pinout diagram (looking from above):



The RA* and RB* pins are I/O pins associated with the PIC registers PORTA and PORTB respectively (RA4 can also be used as an input to the internal timer and RB0 can be used as an interrupt). VDD and VSS are the power pins. The '84 works over a wide range of voltages but typically VSS is connected to 0V and VDD to +5V. The main reset pin, /MCLR, can simply be tied to VDD (either directly or through a resistor) because the PIC includes a reliable power-on reset circuit - all you need to do to reset the PIC is cycle its power. The processor needs a clock and the OSC1 and OSC2 pins can be configured for a variety of different options including crystal and low cost RC oscillator modes.

A simple circuit that you can use as the basis of your first PIC16C84 project is shown here:



(Grab http://www.man.ac.uk/~mbhstdj/files/test.gif for a more readable version).

The circuit uses an RC oscillator and one I/O pin (RB4) attached to a LED. This is all you need to get the PIC to do something and see it happening. Charles Manning (Electronics Australia, April 1996) wrote an amazingly short (6 word) LED flasher program that you can use with this circuit:

```
        LIST    P=16C84
        MOVLW   0
        TRIS    6
        OPTION
LOOP    SLEEP
        INCF    6,F
        GOTO    LOOP
        END
```

The program is written for MPASM (Microchip's free assembler available from http://www.microchip.com). To use the program you'll need to extract it from this file using a text editor (DOS EDIT is fine), save it to another file (LIGHTS.ASM for Exemplo) then assemble it with MPASM (using the command "MPASM LIGHTS.ASM") to produce a hex file LIGHTS.HEX which can then be downloaded to the PIC using your programmer. Ignore the warnings from MPASM about TRIS and OPTION being "not recommended". Make sure you program the PIC with the watchdog enabled and the RC oscillator selected.

If don't have MPASM yet here is a hex representation of the program I prepared earlier:

```
:0C0000000306600620063008 60A0328DE
:00000001FF
```

You can save these two hex records to the file LIGHTS.HEX and skip the MPASM step. If you are using one of my PIC programmers you can download this hex file with the correct configuration by using one of the following commands:

```
PP -RW8 LIGHTS.HEX                          (PP V-0.3)
PP -RW LIGHTS.HEX                           (PP V-0.4)
```

The program uses the watchdog timeout as a timing source to decide when to turn the LED on or off; in fact you can get the LED to flash at different rates by connecting it to a different bit of PORTB (RB0-RB7, pins 6-13). This is an unusual use of the watchdog. Normally the watchdog is used to make sure the PIC is behaving itself and, unless your program is specifically designed to use it, enabling the watchdog is a big mistake. The simple LIGHTS program uses the watchdog to wake it from "sleep" (i.e. power down) mode; on waking, the PIC increments the PORTB register - thus changing the states of RB0-RB7 - and promptly goes back to sleep awaiting the next watchdog timeout. The watchdog timer is clocked by an internal RC oscillator which has nominally the same period on all PICs therefore a consequence of using the watchdog for timing is that the program will still work correctly no matter what PIC oscillator configuration or frequency is actually used (well, the frequency should be at least a few kHz). This feature makes the LIGHTS program very useful for initial testing of almost any PIC protoboard.

The circuit can be modified to give a slightly more entertaining effect by adding more LEDs. Connect the first LED to RB0 (pin 6), a second to RB1 (pin 7), a third to RB2 (pin 8) and so on; it's best to use at least four LEDs and you can use up to eight (the last one connected to RB7, i.e. pin 13). Each LED should be connected in series with a 470 ohm resistor and wired between the PIC pin and the -ve battery connection (VSS) just like the one in the schematic above. The following program will illuminate each LED in turn obeying a to-and-fro pattern (remember the display on the car featured in the old "Knight Rider" TV series?):

```
        LIST P=16C84
;
PORTB   EQU     6
TRISB   EQU     86H
OPTREG  EQU     81H
STATUS  EQU     3
CARRY   EQU     0
RP0     EQU     5
MSB     EQU     3               ;BIT POSITION OF LEFTMOST LED
;
        CLRF    PORTB           ;ALL LEDS OFF
        BSF     STATUS,RP0      ;SELECT REGISTER BANK 1
        CLRF    TRISB^80H       ;SET PORTB TO ALL OUTPUTS
        MOVLW   0AH
        MOVWF   OPTREG^80H      ;ASSIGN PRESCALER (1:4) TO WDT
        BCF     STATUS,RP0      ;SELECT REGISTER BANK 0
        INCF    PORTB,F         ;TURN ON RIGHTMOST LED
        BCF     STATUS,CARRY    ;CLEAR CARRY
LEFT    SLEEP                   ;WAIT FOR WDT TIMEOUT
        RLF     PORTB,F         ;TURN ON LED TO LEFT
        BTFSS   PORTB,MSB       ;REACHED LEFTMOST?
        GOTO    LEFT            ;LOOP IF NOT
RIGHT   SLEEP                   ;WAIT FOR WDT TIMEOUT
        RRF     PORTB,F         ;TURN ON LED TO RIGHT
        BTFSS   PORTB,0         ;REACHED RIGHTMOST?
        GOTO    RIGHT           ;LOOP IF NOT
        GOTO    LEFT            ;START NEW CYCLE
        END
```

MPASM should assemble the program to give this hex representation:

```
:10000000860183168601 0A3081008312860A031056
:100010006300860D861D08286300860C061C0C28CC
:020020000828AE
:00000001FF
```

Again you need to tell your programmer to enable the watchdog timer and RC oscillator. If you save the four hex records to a file (WALKLEDS.HEX say) you can download the program using my programmers by running one of these commands:

```
PP -RW8 WALKLEDS.HEX                    (PP V-0.3)
PP -RW WALKLEDS.HEX                     (PP V-0.4)
TOPIC -RWG WALKLEDS.HEX                 (TOPIC V-0.2)
```

As it stands the "LED walking" program is suitable for four LEDs but you can change the value of MSB if you want to use more - MSB should be 4, 5, 6 or 7 for 5, 6, 7 or 8 LEDs.

The program avoids using the deprecated TRIS and OPTION instructions (Microchip don't want you to use them because they may not be supported by future PICs). Therefore, unlike the previous program, no warnings are generated when the program is assembled. To prevent MPASM generating annoying messages about the correct use of bank selection bits I have inverted the most significant bit of any bank 1 register address (e.g. I use TRISB^80H rather than simply TRISB where the "^" operator denotes bitwise exclusive-OR). This is just a trick I've picked up and there are several other ways to silence MPASM; in fact MPASM allows specific messages to be suppressed. However, I like my programs to assemble without generating warnings or messages even if many of them can be safely ignored. Getting MPASM to shut up without resorting to deliberately suppressing warnings and messages takes a little effort.

As a final Exemplo, the following program will give much the same effect as the 4-LED WALKLEDS program. You'll notice that even though it does the same as the previous program this one is much longer and it's certainly not meant as an Exemplo of efficient programming. Instead it is designed to illustrate a few key PIC idioms and techniques. Amongst other things it contains an interrupt handler, routines to read and write data EEPROM, and shows how table lookups are implemented on the PIC. The program contains Exemplos of some of the more useful MPASM features such as two kinds of macro. It also shows such things as how to override the default radix (hex) for numbers and embed PIC configuration information. Stylistically at least it looks more like a "real" PIC program.

```
; PATTERN.ASM
;
; A program designed to illustrate reading/writing data EEPROM
; and timer interrupts.  A table of values is written to EEPROM
; and the processor then executes a "do nothing" loop.  When the
; timer overflows it interrupts the processor and the next value
; in sequence is read from EEPROM then written to port B where it
; is displayed on LEDs.  By changing the table any pattern of up
; to 64 values can be displayed.
```

```
;
; Copyright (C) 1997 David Tait (david.tait@man.ac.uk)


        PROCESSOR 16C84
        __CONFIG  03FF3                 ;RC oscillator

PCL     equ     2
STATUS  equ     3                       ;standard register files
PORTB   equ     6
EEDATA  equ     8
EEADR   equ     9
INTCON  equ     0BH
OPTREG  equ     081H
TRISB   equ     086H
EECON1  equ     088H
EECON2  equ     089H

RP0     equ     5
Z       equ     2
GIE     equ     7
T0IE    equ     5
T0IF    equ     2
WREN    equ     2
WR      equ     1
RD      equ     0

#define bank0   bcf STATUS,RP0          ;select bank 0
#define bank1   bsf STATUS,RP0          ;select bank 1

magic   macro                           ;magic EEPROM write sequence
        movlw   55H
        movwf   EECON2^80H
        movlw   0AAH
        movwf   EECON2^80H
        endm

        cblock  0CH                     ;variable block
        n_vals
        n_tmp
        endc

        ;*************************;
        ; Main program entry point ;
        ;*************************;

        org     0
        goto    start


        ;***********************;
        ; Interrupt entry point ;
        ;***********************;

        org     4

; Normally context should be saved before the interrupt service
; routine and restored after but that's not necessary in this program
; because the processor is doing nothing between interrupts.  See
```

```
; the PIC datasheet for the recommended procedure.

        movf    EEADR,w
        xorwf   n_vals,w
        btfsc   STATUS,Z                ;EEADR == n_vals?
        clrf    EEADR                   ;yes, start again at 0
        call    ee_rd
        movf    EEDATA,w                ;read EEPROM
        movwf   PORTB                   ;display byte
        incf    EEADR,f                 ;new address
        bcf     INTCON,T0IF             ;clear interrupt flag
        retfie


start   clrf    PORTB
        bank1
        clrf    TRISB^80H               ;port B all outputs
        movlw   B'00000111'
        movwf   OPTREG^80H              ;timer 0 prescale 256:1
        bsf     EECON1^80,WREN          ;allow writing to EEPROM
        bank0
        call    ee_init                 ;transfer table to EEPROM
        bank1
        bcf     EECON1^80H,WREN         ;disallow writing to EEPROM
        bank0
        bsf     INTCON,T0IE             ;enable timer interrupt
        bsf     INTCON,GIE              ;globally allow interrupts

loop    goto    loop                    ;do nothing forever


; ee_init
;
; initialise EEPROM from table

ee_init clrw
        call    lut                     ;get number of table entries
        movwf   n_vals                  ;and save
        movwf   n_tmp                   ;and again
        clrf    EEADR
        decf    EEADR,f                 ;EEADR = -1
ee_in1  incf    EEADR,f                 ;next address
        movf    EEADR,w
        addlw   1
        call    lut                     ;get associated table entry
        movwf   EEDATA
        call    ee_wr                   ;write to EEPROM
        decfsz  n_tmp,f                 ;another?
        goto    ee_in1                  ;yes
        clrf    EEADR                   ;no, then finished
        return


; lut
;
; look up table

lut     addwf   PCL,f                   ;add W to PCL to get table entry
        retlw   D'12'                   ;number of entries in table
```

```
        retlw   B'1000'                 ;first entry
        retlw   B'1000'
        retlw   B'0100'
        retlw   B'0100'
        retlw   B'0010'
        retlw   B'0010'
        retlw   B'0001'
        retlw   B'0001'
        retlw   B'0010'
        retlw   B'0010'
        retlw   B'0100'
        retlw   B'0100'                 ;last entry


; ee_wr
;
; Writes byte in EEDATA to EEPROM location at EEADR.  Interrupts
; should be disabled before calling ee_wr.

ee_wr   bank1
        magic                           ;invoke magic sequence
        bsf     EECON1^80H,WR           ;start write
ee_wr1  btfsc   EECON1^80H,WR           ;write complete?
        goto    ee_wr1                  ;no
        bank0
        return


; ee_rd
;
; Reads EEPROM byte at EEPROM location EEADR into EEDATA

ee_rd   bank1
        bsf     EECON1^80H,RD           ;start read
        bank0
        return                          ;read will be complete on return

        end
```

Here is the MPASM generated hex file (save as PATTERN.HEX):

```
:020000000E28C8
:0800080009080C060319890127
:1000100044200808860089A0B110900860183160E
:1000200086010730810008158312 1C2083160811F1
:1000300083128B168B171B2800012C208C008D003F
:1000400089018903890A0908013E2C2088003A2089
:100050008D0B22288901080082070C3408340834EB
:1000600004340434023402340134013402340234DE
:1000700004340434831655308900AA30890088146A
:1000800088184028831208008316081483120800 79
:02400E00F33F7E
:00000001FF
```

With my programmers this can be downloaded using:

```
PP PATTERN.HEX                          (PP V-0.4)
TOPIC -G PATTERN.HEX                    (TOPIC V-0.2)
```

These projects may not seem very exciting but if you have just built or bought a PIC programmer and have hurriedly put together the simple test circuit then seeing a LED flash on and off is exceedingly gratifying. I hope you find that out for yourself. Good luck.

2nd Edition
4/Feb/97

FILES.TXT This file

PP.TXT Brief description of the programmer and the hardware dependent
parts of the software

PP: PIC16X8X PROGRAMMER HARDWARE
==========

David Tait
david.tait@man.ac.uk
http://www.man.ac.uk/~mbhstdj

I released information about a simple PIC programmer in early 1994 because I couldn't find any homebrew designs around at the time. That's all changed now. There has been an explosion of hobbyist interest in PICs and as a result there's plenty of DIY information available in magazines and on web sites. The problem for most people now is choosing between all the stuff that's available. Although a few of the other offerings you'll find are based on variants of my original stuff, and there are even commercial versions from Maplin, DonTronics and a others (but don't get the wrong idea - nobody has paid me a penny), that shouldn't be taken as any kind of endorsement of what you've got here. I can perhaps make your choice a

little easier by telling you that what I'm offering is some DOS command-line software which is only capable of programming the PIC16X8X family (16F83/84 and 16C84) using relatively simple hardware (see pp.pcx) or even trivial hardware (see qandd.pcx) connected to the parallel port of a PC. Now, if you really wanted something to program other PICs, look elsewhere; if you wanted a GUI, look elsewhere; if you wanted Windows software, look elsewhere; or if you would have preferred a serial port connection, look elsewhere. All of those alternatives are out there somewhere. On the other hand, if you've built a programmer which you suspect may be compatible with mine, and want some no-frills DOS software to program the 16X8X family, look here.

The essential features of what has become known as a "Tait"-style programmer can be summarised as follows:

- Uses the PC parallel port pins D0, D1, D2, D3, ACK and GND.
- The PIC programming pins RB6 and RB7 are connected to D0 and D1 via open collector buffers (either inverting like the 7406 or non-inverting like the 7407).
- RB7 is connected to ACK via one of the open collector buffers.
- The PIC VDD pin is connected to +5V either via a 4066 CMOS analogue switch (possibly the parallel combination of two or three 4066 switches) or via a PNP transistor.
- The PIC /MCLR pin is connected to the programming voltage (about 13-14V) either via a 4066 switch or a PNP transistor.

There are therefore at least four possible variants:

- 7406 buffers/4066 switches
- 7407 buffers/4066 switches
- 7406 buffers/PNP switches
- 7407 buffers/PNP switches

The original description contained an ASCII sketch of a simple 7406/4066 version (this is still around as pic84pgm.zip in my PIC archive - reach it via the URL at the head of this document). However, possibly the most popular variant is based on 7407/PNP hardware and an Exemplo of this type is shown in the file pp.pcx. The programmer should work with most PCs but the connection between the hardware and the PC should be fairly short. A couple of small value capacitors (100pF or so) one connected between RB6 and GND and the other between RB7 and GND may be needed to suppress the "ground bounce" exhibited by some buffers. PCB designs for various versions of the hardware can be found in several places (for Exemplo, there are a couple in my PIC archive: the 740X/4066 versions can be built using the layout in pic84art.zip produced by Michael Laidlaw; and Steve Willis has provided a layout for a PNP transistor variant in the file pp875.zip). The hardware can be used to program most midrange PICs but I don't supply any software to do that myself. Note, because of the higher current requirements of the non-EEPROM PICs I don't recommend using the 4066-based hardware with these devices although it may work.

There is now a lot of software around which is compatible with the programmer (for Exemplo, www.dontronics.com offers programs written by Nigel Goodwin for the DonTronics version; or see www.sistudio.com for PIP-02 by Silicon Studio). My own software is pretty basic and as mentioned previously only works with the 16X8X family. The software is designed to be run under MS-DOS and the user interface, if it can be dignified by such a description, is documented in the file program.txt. The current version of the software is V-0.5. There have been a few subtle changes since V-0.4 but the only essential difference between V-0.5 and the previous version is slightly better support for the 16F84. Another small change in V-0.5 is a more explicit separation between the front-end and the hardware dependent parts. I hope this makes it easier for people to port the program to other environments. It should also make it a bit easier for me to document my other simple 16x8x programmer designs (like the forthcoming update for my TOPIC board - see topic02.zip in the PIC archive) because the description in program.txt is common to them all.

The remainder of this document describes the hardware dependent features of the version of my software intended for use with any "Tait"-style

programmer. The executable for this version is in pp.exe and the C source is archived in src.zip.

MS-DOS environment variables are used to communicate with the driver software (i.e. the hardware dependent parts of the program). (As MS-DOS reserves very little memory for environment variables it may be necessary to increase the allocation: more memory can be reserved using the /E:nnnn option of command.com and using /E:1024 will quadruple the default allocation for Exemplo.) The hardware driver changes its behaviour in response to the following variables:

- PPLPT=nn Select printer port nn (default 1 thus selecting LPT1 which is often, though not always, at address 378 hex). The software will only use likely printer port addresses.

- PPDEBUG=nn Enable debugging if nn=1 (default nn=0). This invokes a mode that lets you verify correct operation of the hardware. In debug mode hitting enter repeatedly will toggle each of the LPT pins in turn and also checks the input line for consistency when D0 (equivalent to RB7) is toggled.

- PPDUMP=nn Dump in INHX16 format if nn=16 (default INHX8M).

- PPDELAY=nn Add nn*0.83 microseconds delay between LPT port accesses (possibly useful on fast PCs). The meaningful range of nn is 0 to 127 and the default is 6.

- PPSETUP=nn Select hardware setup:

  nn=0 selects 7406/4066 switches (default).
  nn=1 " 7407/4066 switches.
  nn=2 " 7406/PNP transistor switches.
  nn=3 " 7407/PNP transistor switches.

No environment variables NEED to be set for the original hardware described in pic84pgm.zip or the Maplin version. To use the programmer design shown in pp.pcx you should make sure PPSETUP=3. This can be done by several means: by adding

```
set ppsetup=3
```

to your autoexec.bat file; by simply typing the same thing before using the software; or by writing a batch file to set the variable and run the software which you can then use instead of invoking pp directly. The batch file approach is probably the most convenient. Here's an Exemplo designed for 7407/PNP hardware on LPT2:

```
@echo off
set ppsetup=3
set pplpt=2
pp %1 %2 %3 %4
set ppsetup=
set pplpt=
```

A dummy batch file is provided in mypp.bat so that you can customise it for your own version of the hardware.

Now you should read program.txt to learn more about the programmer software.

V-0.2 19 April 1998

PROGRAM.TXT How to use the programmer software

PP: GENERIC PIC16X8X PROGRAMMER SOFTWARE
==========

David Tait
david.tait@man.ac.uk
http://www.man.ac.uk/~mbhstdj

Introduction

This document describes some simple MS-DOS based software for programming Microchip PIC16x8x (i.e. the 16F83, 16F84 and 16C84) microcontrollers. Different versions of the software are available depending on the interface between the programming hardware and the PC. What is described here is the common front-end or hardware independent part. Throughout this document the software is called "pp" although the actual name may depend on the hardware specific variant. Also depending on the hardware driver certain features mentioned here may not be available. For Exemplo, some hardware cannot read back information from the PIC and therefore it is impossible to use features that rely on this capability.

Overview

The purpose of the software is to take an image of a PIC program from a file produced by an assembler or compiler and download it to the PIC. The image should be in a hex file: the most common hex file format is designated INHX8M by Microchip but an older style called INHX16 is still common. The software can read data from either type of hex file. There are four separate areas of a PIC that can be programmed:

- Program memory — This is where the PIC program goes (the PIC starts execution at location 0 of program memory). There are 1024 14-bit words of program memory (the 16F83 has 512).
- Data memory — 64 bytes of EEPROM memory.
- ID locations — 4 words of memory which can be used to tag the PIC by storing a unique number there. (there are few other locations called "test memory" in this area).
- Configuration word — Used to select PIC oscillator setup and enable or disable the watchdog, power-up timer and code protection.

The hex file can contain information about any or all of these areas and the software will program them accordingly. Embedding all the information in the hex file makes life very simple. In this case all you need to do to program a PIC with the hex file "prog.hex" is type:

```
pp prog.hex
```

How simple can you get? (Actually, if you are working on the same project for a while it's likely that you are repeatedly downloading the same hex file. In this case you can write a batch file called "p.bat" with the contents:

```
@echo off
pp prog.hex
```

and then all you need to type is:

```
p
```

which really is difficult to beat for simplicity. More about using batch files later). Most PIC development tools have the means to embed all the necessary information in the hex file. For Exemplo, MPASM, Microchip's free assembler, includes the three directives, __CONFIG, __IDLOCS and DE, to define configuration, ID and data memory respectively. In fact, the only way to use pp to program data memory and the ID locations is to embed information about these areas in the hex file. However, as it is so vital, the software does provide another way to define the configuration word if it is not included in the file. (The programs in pichex01.zip, which should be available where you found this stuff, contains utilities to build integrated hex files for the 16C84 if your development tools don't.)

The software lets you erase the PIC (i.e. set the PIC contents back to its unprogrammed state). Use this function sparingly; it isn't necessary to erase the entire PIC before it is reprogrammed because each word is overwritten during programming. The software is designed so that it only overwrites locations which are different to those in the hex file (which means that if you try to program the PIC twice with the same hex file nothing is overwritten the second time). Erasing the PIC can slow down the programming process as it will force all memory to be updated. Another point to consider is the finite lifetime of the PIC; a 16x8x can only be reprogrammed a finite number of times (minimum 100 but 1000 typically) therefore minimising the number of times each location is reprogrammed can extend the useful life of the PIC. There are times when the PIC must be erased however. In particular the erase function can be used to defeat code protection: part of the config word is used to prevent the PIC contents from being read which in turn prevents people from cloning your PIC. The contents of a code protected PIC cannot be overwritten but by erasing the PIC, code protection is removed thus making the PIC available for reprogramming again.

The PIC contents can be dumped to a hex file or compared with (verified against) another hex file. The verify feature is only useful if the PIC is not code protected although the software does verify as it goes along. It is important that the PIC is verified under the same conditions as it was programmed: if the software was used to define the config word during programming it must also be used to do the same during verification. If you don't do this verification will probably fail when the configuration word is compared.

Using the software

If all is well when you type pp you should see a banner similar to this

one:

PIC16F84 Programmer Version 0.5 Copyright (C) 1994-1998 David Tait.

Usage: pp [-lxhrwpcdevgosn!] hexfile

```
Config:   l = LP,   x = XT,     h = HS,      r = RC
          w = WDTE, p = PWRTE,  c = code protect
Others:   d = dump, e = erase,  v = verify,  g = go
          o = old,  s = silent, n = no read, ! = no wait
Defaults: RC, /WDTE, /PWRTE, unprotected,
          no erase, stop, new, verbose, read, wait
```

Bug reports to david.tait@man.ac.uk

This is meant to indicate that pp is invoked with a command line that consists of an optional set of switches signalled by the prefix '-' (though '/' works as well if you prefer) followed by a filename. The switches are used to either define the configuration word (overriding the config specified in the file) or invoke a programmer function or both. Some valid command lines might be:

```
pp prog.hex
pp -l -w prog.hex          LP oscillator and watchdog enabled
pp -d new.hex              dump PIC to new.hex
pp -ec prog.hex            erase and code-protect
```

This shows how switches can be defined individually (-l -w) or as a block (-ec). Some switch combinations don't make sense or may be contradictory and the exact action is not obvious. For Exemplo:

```
pp -xr prog.hex
```

tells the programmer to select both the RC oscillator and the XT oscillator configuration. Contradictions like this are resolved by obeying the last switch specified (in this case the RC oscillator is selected).

Before using the software to program a PIC it may be necessary to inform the hardware driver about the specific hardware attached. This is typically done using MS-DOS environment variables (details are in the hardware specific documentation). An easy way to do this is to run pp from a batch file which sets up the environment variables as required. Here's a typical Exemplo which I'll call mypp.bat:

```
@echo off
set ppsetup=3
pp -n %1 %2 %3 %4
set ppsetup=
```

Here the ppsetup variable tells the driver something about the hardware, then pp is invoked with a fixed switch (-n in this case) and other arguments are taken from the command line and when pp exits ppsetup is removed from the environment. The batch program can be run just like pp itself:

```
mypp -xw prog.hex      set ppsetup=3 then run pp -n -xw prog.hex
```

Though pp is not designed to be used with Windows it can be run in a DOS box. The best way to do this with Windows 3.1 is to create a batch file

appropriate to your setup then use the PIF editor to create a PIF file to run the batch program. The PIF file should specify exclusive execution and windowed display but if you want to see any error messages it should not specify close window on exit; the defaults are OK otherwise. File Manager can then be used to drag the PIF file onto a folder to make an icon which you can simply double-click to run pp with the command line entered with the PIF editor. I don't use Windows 95 but perhaps something equivalent can be done, otherwise just run the batch file or pp itself from within a Win95 DOS box.

After any hardware specific environment variables have been set the software should be run once before trying to program a PIC. This will let the driver initialise the hardware.

Switches

This section gives a detailed listing of all the switches used by pp. As mentioned previously, embedding information in the hex file means you don't have to use any switches to program a PIC. However, if the configuration information is not contained in the file then it should be defined on the command line (switches -lxhrwpo) otherwise a default word is written to the PIC and this is unlikely to be the one you want. Dumping, erasing and verification are invoked by switches -d, -e and -v respectively. Finally, some switches (-gsn!) change the behaviour of the program.

Configuration switches

The first few switches are used to set the 14-bit configuration word. In the 16F83/84 the config word is organised like this:

    CC CCCC CCCC PWOO

where the bits marked C are used to enable/disable code protection, the P bit (Microchip calls it PWRTE) controls the power-up timer, the W bit (or WDTE) enables/disables the watchdog timer and the two bits marked OO define the PIC oscillator configuration. By default (i.e. if neither the hex file nor the command line switches define the config word) pp uses 11 1111 1111 1011 (3FFB hex); this corresponds to no code protection, power-up timer disabled, watchdog disabled and RC oscillator. The 16C84 has a slightly different layout:

    -- ---- ---C PWOO

where the bits marked - are don't care (though they read back as 1). Another difference is that the sense of P is inverted (i.e. 1 enables the power-up timer). Thus if 3FFB hex is written to a 16C84 it would select no code-protection, power-up timer enabled, watchdog disabled and RC oscillator. The config switches are used to define the C, P, W and O bits as follows:

-l select the LP (low power) oscillator mode

-x select the XT (crystal) oscillator mode

-h select the HS (high speed) oscillator mode

-r select the RC (resistor/capacitor) oscillator mode

-w enable watchdog timer

-p enable the power-up timer

-c enable code protection

There are a few cases to consider. If any of the above switches are specified on the command line then the config word defined in the hex file is totally ignored (overridden). If none of the above are specified then the file config is used. If none are specified, and the config word is not defined by the file, the default word is used. To take care of the differences between the 16F83/84 and 16C84 another switch is available:

-o assume old style (16C84) config layout

The -o switch is only really needed if you are programming a 16C84 and must enable the power-up timer on the command line (using -p). It does have a couple of other side effects though: using -o means only the low order five bits of the config word are verified (all bits are programmed); usually the software prints a precis of the config word (like "CP-X" meaning code protection and power-up timer enabled and XT oscillator selected) and the precis will lie about the P bit if the -o switch is not used appropriately. If the default word is used to define the config (because neither the command line nor the file say what it should be) then -o inverts the P bit (i.e. the default word becomes 3FF3 hex). The -o switch on its own DOES NOT override or change a file-specified config word. Thus, provided the hex file defines the correct style of config (and sets all high order config bits to 1) you can just forget about -o regardless of whether you are programming a 16F84 or a 16C84. There is no switch to indicate a 16F83 is being programmed and it's up to you to make sure you don't try to write more than 512 words of program memory in this case.

Dumping, erasing and verifying

Other programmer functions are requested using the following switches:

-d dump the PIC contents to the given file.

-e erase the PIC

-v compare PIC contents with the given file

Usually -d is used on its own (or with one of the behavioural switches described later). If the file specified to receive the dump already exists the dump is aborted. If any parts of the PIC are in their unprogrammed state they are not dumped, however, unprogrammed holes in the program or data memory are dumped. If the PIC is entirely blank the dump file is not created. The -e switch can be used with or without a file argument; in the first case the PIC is erased then programmed and in the second case it is simply erased. The -v switch reads the PIC and compares it with what should have been programmed according to both the file and the other configuration switches. Verification halts at the first difference found and the config word is compared last.

Switches changing program behaviour

The remaining switches are used to change the normal program behaviour. The exact effect depends on the hardware driver. These switches are:

-g go - tells the driver to enable PIC execution

-s run silently

-n no read - don't read from the hardware

-! no wait - program the PIC without waiting for interaction

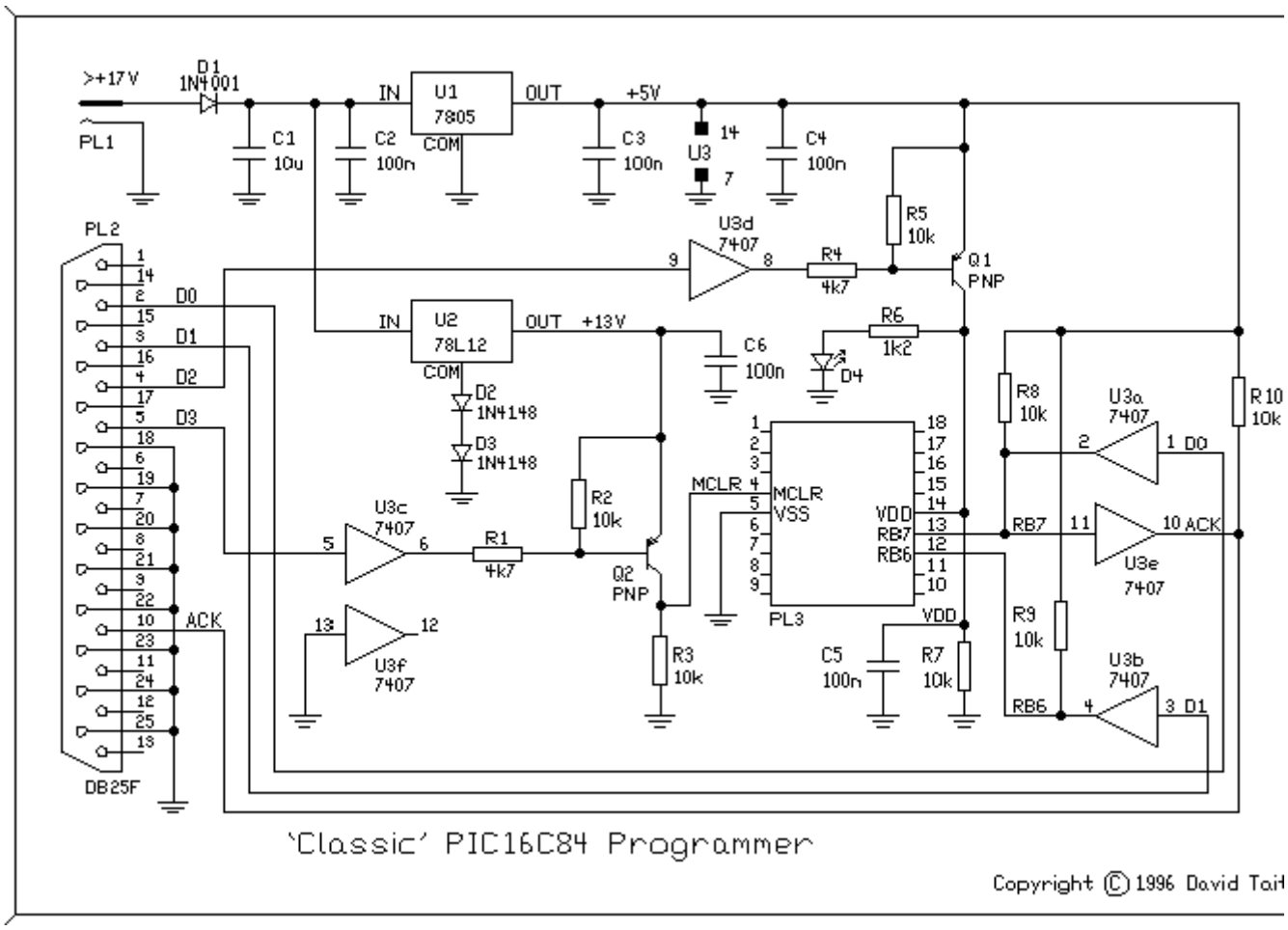
If the hardware can program the PIC in-circuit (Microchip call this in-circuit serial programming or ICSP) the -g switch tells the hardware to let the PIC run after programming, or, if no file argument is specified, simply enable the PIC to run. By default the PIC is held in reset after programming. If the driver doesn't support ICSP the action of -g is undefined and it's best not to use it. Usually pp gives some indication of progress and prints error messages (verbose mode) but -s suppresses this behaviour. If the -s switch is used success or failure is signalled using the ERRORLEVEL return value (0 if all is OK, 1 for error). As described earlier the PIC is read before it is programmed but not all hardware can support this. The -n switch tells pp that the hardware can't read from the PIC. This switch is essential for some hardware. Even if the hardware is supposed to be capable of reading from the PIC the -n switch is useful for debugging. For Exemplo, try -n if the programmer gives "verify failed during programming" errors. Obviously -n is not compatible with -d or -v. Some hardware needs to know when the PIC is ready to be programmed and asks the user to confirm this condition. Using -! will skip the confirmation step (note -s asserts -! too).

Summary

The software described in this document can be used to program PIC16x8x microcontrollers when used with suitable hardware. If all you want to do is program a PIC and the configuration word is defined in the hex file then the only command line argument needed is the filename of the hex file. Running the program from within a batch file makes it possible to customise the program behaviour (it is even easy to use a batch file to make a chatty version that prompts for the hex filename and any command lines switches). Ultimately if you don't like anything about the way pp works the source is provided so that you can change it for yourself.

V-0.2 19 April 1998

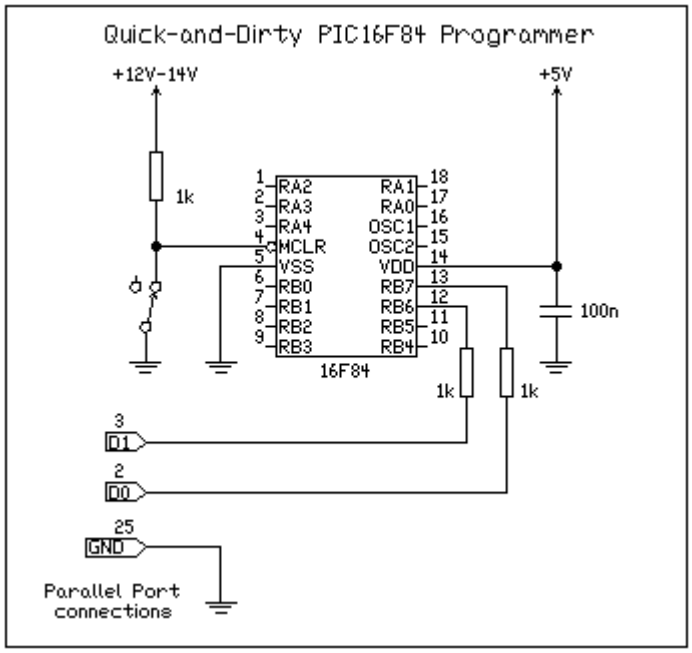
PP.PCX PIC16X8X programmer schematic



PP.EXE PIC programmer software

SRC.ZIP C source code for the programmer software

QANDD.PCX The quick-and-dirty programmer schematic



QANDD.TXT Brief description of the quick-and-dirty programmer

QUICK-AND-DIRTY 16F84 PROGRAMMER
==========

David Tait
david.tait@man.ac.uk
http://www.man.ac.uk/~mbhstdj

If you have a source of +12V-14V and +5V available you can program 16F84s with virtually no hardware. The schematic in qandd.pcx shows one possible setup. This is more or less the simplest circuit possible. In one design I've seen (Mark Cox's BLOWPIC) things are made even more simple by connecting the PIC directly to the parallel port but the 1k resistors add a little protection (the values are not critical but don't use less than 1k). Adding a few more components will make things a lot more convenient (see topic02.zip in my PIC archive for a more elaborate version; or, if I ever find time to document it, my hardware for programming 16F84s in-circuit via the PC serial port). The quick-and-dirty circuit has several limitations and one very big plus point: it should only take a few minutes to lash the thing together. The hardware is usable with the pp V-0.5 software described in pp.txt and program.txt but, because there is no way of reading the PIC, pp must be run with the -n switch and the PIC can't be dumped or verified. One other drawback with this setup is that program memory and data memory can't be programmed together. Using a hex file with both program and data memory specified will only give correct results for program memory. If this is a problem the hex utilities in pichex01.zip (from my PIC archive) can be used to split the hex file so that each area

can be programmed separately. As most PIC applications don't really need the data memory to be programmed it's not likely to be a major limitation in practice.

Although they were simply meant as Exemplos of how to customise pp the files mypp.bat and mypp.pif are in fact ready to use with the hardware described in qandd.pcx. To use the quick-and-dirty programmer: insert a PIC (any of the 16x8x family); then, making sure the switch is closed, turn on the power supplies; run mypp with a command line of the form:

```
mypp [ -xyz ] prog.hex
```

where -xyz is an optional set of valid pp switches excluding -! or -s and prog.hex is the name of the hex file to be downloaded (using mypp rather than pp itself ensures that the software uses the correct setup, i.e. ppsetup=3, and that the -n switch is specified); when pp asks you to "Insert PIC ..." open the switch and hit any key (or control-C to abort); when pp exits close the switch, turn off the supplies and remove the PIC. You can run mypp from a Windows DOS box or drag mypp.pif onto a Windows folder for lazy "double-click" execution (you'll need to use the Windows PIF editor to set the appropriate command line though).

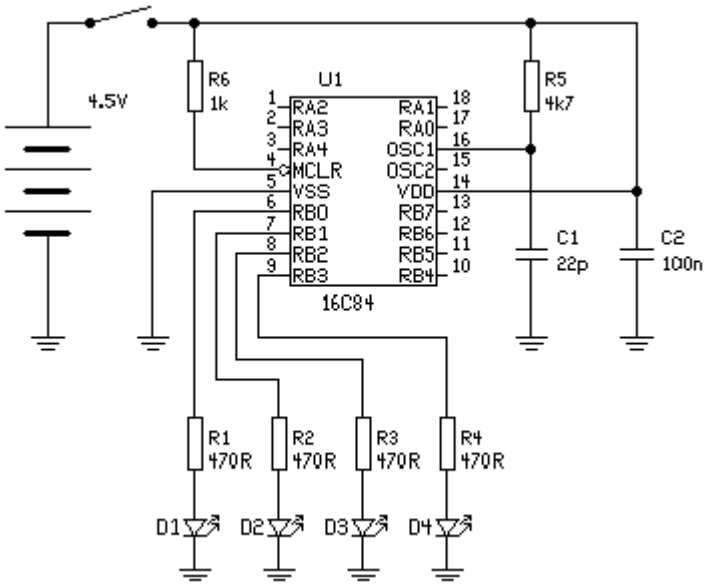
If you add another single-pole double-throw switch wired so that it selects whether the /MCLR resistor is connected to +12V-14V (as shown) or to +5V you get a crude form of in-circuit programmer. This is roughly speaking how Derren Crome's Everyday Practical Electronics setup works. When the new switch is in the +5V position the existing switch either resets the PIC (closed position) or lets it run (open position). Of course, for the PIC to do anything useful you'll need to add oscillator components and make connections to some of the remaining port pins (as RB6 and RB7 are used for programming they are unavailable). You could try this out using the relevant bits of the setup shown in test.pcx. To program/reprogram the PIC you should first close the reset switch and select the +12V-14V supply with the other switch; then run mypp with whatever else you need on the command line (mypp walk.hex for Exemplo); when pp asks you to "Insert PIC ..." open the reset switch and hit a key; when pp exits close the reset switch and reselect the +5V supply; finally, open the reset switch at which point the PIC should start running. Sounds complicated but you'll soon get the hang of it: reset closed - select +12V-14V - reset open - reset closed - select +5V - reset open. (Adding a couple of transistors as described in topic02.zip automates these steps.) Build everything on a solderless breadboard and you don't really need the switches at all as you can get the same effect by moving a couple of wires about.

If you don't have power supplies available you'll need a few more components to get you going. There are several options: one that I hesitate to mention is to steal the supplies from your PC via a spare floppy disk power connector - do this at your own risk (and the risk is considerable!); use a regulated 12V supply (12V battery or 8 AA batteries or whatever) and a 7805 regulator plus a couple of decoupling capacitors to get +5V; use the power supply parts of the programmer shown in pp.pcx together with an inexpensive "battery eliminator" on its 12V setting (which will typically produce about 16V-17V so don't be tempted to omit the 78L12 regulator); use a regulated +5V supply plus a couple of small 9V (PP3) batteries in series instead of the +12V-14V supply and connect a 12V zener diode from /MCLR to ground; use a regulated +5V supply and a DC-DC converter (one of the 8-pin Maxim chips for Exemplo) to get +12V; you get the idea ...

Have fun.

V-0.3 20 April 1998

TEST.PCX Simple 16C84 (or 16F84) test circuit (just connect one LED for an even simpler circuit)



WALK.ASM PIC "LED walking" program to exercise the test circuit

```
; WALK.ASM
;
; To use this program connect four LEDs from each of RB0-RB3 to ground
; via four 470 ohm resistors.  The LEDs are illuminated one at time in
; a to-and-fro pattern.
;
; The illumination rate is more or less independent of the PIC clock
; frequency and configuration although this program assumes an RC
; oscillator.  The program includes the __CONFIG, __IDLOCS and DE
; directives (mostly just to show how they can be used).  The program
; can be used unchanged on any 16X8X device.

      LIST          P=16C84
      ERRORLEVEL    -302    ;SUPPRESS BANK SELECTION MESSAGES
      __CONFIG      3FF7H   ;RC OSC, WATCHDOG
      __IDLOCS      1234
```

```
;
PORTB   EQU     6
TRISB   EQU     86H
OPTREG  EQU     81H
STATUS  EQU     3
CARRY   EQU     0
RP0     EQU     5
MSB     EQU     3               ;BIT POSITION OF LEFTMOST LED
;
        CLRF    PORTB           ;ALL LEDS OFF
        BSF     STATUS,RP0      ;SELECT REGISTER BANK 1
        CLRF    TRISB           ;SET PORTB TO ALL OUTPUTS
        MOVLW   0AH
        MOVWF   OPTREG          ;ASSIGN PRESCALER (1:4) TO WDT
        BCF     STATUS,RP0      ;SELECT REGISTER BANK 0
        INCF    PORTB,F         ;TURN ON RIGHTMOST LED
        BCF     STATUS,CARRY    ;CLEAR CARRY
LEFT    SLEEP                   ;WAIT FOR WDT TIMEOUT
        RLF     PORTB,F         ;TURN ON LED TO LEFT
        BTFSS   PORTB,MSB       ;REACHED LEFTMOST?
        GOTO    LEFT            ;LOOP IF NOT
RIGHT   SLEEP                   ;WAIT FOR WDT TIMEOUT
        RRF     PORTB,F         ;TURN ON LED TO RIGHT
        BTFSS   PORTB,0         ;REACHED RIGHTMOST?
        GOTO    RIGHT           ;LOOP IF NOT
        GOTO    LEFT            ;START NEW CYCLE
;
        ORG     2100H
;
        DE      "Copyright (C) 1996 David Tait"
        END
```

WALK.HEX Hex file produced by MPASM from WALK.ASM

```
:10000000860183168601A3081008312860A031056
:100010006300860D861D08286300860C061C0C28CC
:020020000828AE
:0840000010002003000400AE
:02400E00F73F7A
:1042000043006F00700079007200690067006800 69
:1042100074002000280043002900200031003900EC
:1042200039003600200044006100760069006400 17
:0A4230002000540061006900 7400D2
:00000001FF
```

MYPP.BAT Batch file to run PP (customise this for your own setup;

as it stands it can be used with the QANDD.PCX hardware and you can use MYPP WALK.HEX to download WALK.HEX to a

PIC with this hardware)

MYPP.PIF Use this to run the software from Windows (you'll need
to use the PIF editor to customise it for yourself;
as it stands it assumes the software is in the directory
C:\WORK\NEW\PIC84V05 and will try to download WALK.HEX
using MYPP.BAT; the font used can be changed when the
PIF file opens the window)

Note: the *.PCX files can be viewed with the Windows Paintbrush program.

Programming the PIC16F84

PIC MICROCONTROLLERS

:

When studying the PIC series of microcontrollers, the first thing to realize is that the architecture is completely different from anything you are probably used to. This makes understanding the PIC quite confusing at first. You are probably familiar with the spinal cord type of computer with memory, cpu and perpherial chips hooked in parallel to the same data and address bus. The PIC chips have two separate 'data' busses, one for instructions and one for everything else. Instructions are essentially in ROM and dedicate the microcontroller to doing one task. There is very little RAM, a few dozen bytes, and this is reserved for variables operated on by the program. There is also very little 'data' storage, again a few dozen bytes, and this is in EEPROM which is slow and clumsy to change.

TYPES OF PIC's:

There are a dozen or so I/O pins on the PIC which can be configured as inputs or outputs. When outputs, they have to strength to drive LED's directly. A couple of the I/O pins are used to program the internal ROM serially with an external programmer. These are the OTP, (One Time Programmable), chips. There are also similar chips with UV erasable EPROM's used for prototyping at about three times the price. Then there is one series that is of special interest to the hobbyist, the 16F84, (C84,83), chips which have electrically reprogramable EEPROM memory for instructions. These can be reprogrammed hundreds of times. There have been many programmers designed for this series, one of the simplist appeared in the Sept. '98 issue of 'Electronic Now'.

ROM INSTRUCTION MEMORY:

In the 16F84 instructions are 14 bits wide and stored in EEPROM. There is a maximum of 1024 of these. It is impossible to modify these instructions except through external programming. You can't have self modifying code. When the chip is reset, a program counter is set to zero and instructions are executed from there. The program is retained when power is removed from the chip.

RAM MEMORY:

Besides the 14 bit program bus there is another 8 bit data bus in the PIC connected to registers, ports, timer etc. There are 80 RAM locations in the 16F84. RAM is where you put your variables. The only way to change these RAM locations is through instructions. You don't load RAM from outside as in a 'regular' computer. The information in RAM disappears when power is removed. The first 12 RAM locations, ($00 - $0B), have internal registers mapped to them. Changing these locations with instructions changes the corresponding registers. Microchip calls RAM locations 'files' or 'registers' and uses the symbol 'f' when referring to them. The remaining 68 locations can be used for your variables. Microchip calls the first 12 locations special function registers and the remaining 68 general purpose registers.

BANKED RAM MEMORY:

Five special function registers are not among the first twelve addresses, not even among the 80 . Because of something called 'banking' you have to set a bit in the byte at RAM location 3 to reach them. This location is called STATUS and the bit, (bit 5), is called RP0. If RP0 is zero you are in bank 0, if it is 1 you are in bank 1. For your own variables it doesn't matter which bank is in use because they are mapped to both banks. For some of the first 12 locations it does matter. Seven of the 12 are mapped to both banks but five are not; so location 5 for Exemplo has two meanings depending on RP0. If RP0 is clear, (bank 0), location 5 refers to the data on PORT A. IF RP0 is set, (bank 1), location 5 refers to the direction register TRISA that tells which bits of PORTA are inputs and which are outputs.
Much of this complication can be avoided by using two instructions that Microchip indicates it might not support in future products. The TRIS instruction can be used to set the port direction registers and OPTION can be used to set the OPTION register which deals mainly with timer operations. If you port your code to future Microchip processors that don't support these instructions, you will probably want to rewrite the code for some other reason anyway.

EEPROM MEMORY:

There is a third type of memory in the 16F84, 64 bytes of electrically reprogrammable memory, (8 bit). This could be used to hold values you would like to remember when the power is turned off. There are a couple of difficulties. First, the memory is not directly addressable; you have to work indirectly through four of the special function registers. Secondly, it takes a few hundredths of a second to 'burn' the information in so it isn't real fast memory like RAM. This memory can also be burned in when you burn in program memory.

INSTRUCTIONS:

The small instruction set, (37 instructions), and the 14 bit size of instructions lead to a number of compromises. For one thing you can't have two registers specified in a single instruction. Each register takes 7 bits to specify its address, but you also have to specify the instruction number and what to do. The solution is to run almost everything through 'W' or working register which is internal to the processor and doesn't have an address. A register to register transfer would take two instructions. Suppose you had a pattern of LED segments to be lit in the variable PATTERN and want to move it to PORTB to light the segments:

```
     MOVF  PATTERN, W       ; copy the contents of PATTERN into the
working
                            ; register
     MOVWF PORTB            ; copy the contents of W into Port B
```

The first instruction is of the form MOV f,d which moves the register 'f' to the destination 'd', ('W' in this case). The second instruction simply moves whatever is in 'W' into the register 'f', (MOVWF, f). PATTERN remains unchanged in the first instruction and W remains unchanged in the second. Maybe it is more like a copy than a move.

You might think you could get away with one instruction with literals. Literals ,(k), are 8 bits (0-255). Instructions with literals have no room to specify a register, you must use 'W'. Loading a register with a literal also takes two instructions:

```
      MOVLW $AA             ; put the pattern 10101010 into W
      MOVWF PATTERN         ; put W into the register PATTERN
```

The same applies when literals are used in addition, subtraction and the logical functions **AND**, **IOR** ,(inclusive OR) and **XOR** ,(exclusive OR); all involve two instructions:

```
      MOVLW k               ; move literal into 'W'
      SUBWF f,d             ; put the result of subtracting  W from f
into d
                            ; d could be either W or f
                            ; if it's W then f is not changed
                            ; if it's f then W is unaffected
```

Suppose we wanted to zero out the lower nibble of PATTERN:

```
      MOVLW $F0             ; set up mask
      ANDWF PATTERN, f      ; result of PATTERN AND $F0 is placed in
PATTERN
                            ; note that the destination could be W if we
                            ; want the changed pattern to end up there
```

IF the value to be changed is already in W and the destination is W, a single instruction will work: **ADDLW k**, **SUBLW k**, **ANDLW k**, **IORLW k** and **XORLW k**.

Single operand instructions are easy to understand:

```
      CLRF f    ; set all bits in register f to zero, (clear register f)
      CLRW      ; set all bits in register W to zero, (clear working
register)
      BCF f,b   ; set bit b in register f to zero, (bit clear bit b in f)
      BSF f,b   ; set bit b in register f to one, (bit set bit b in f)
```

THINGS TO WATCH:

Small errors are easy to make and can cause hours of wasted time. Here are some that can cause problems: Many instructions in a program are MOV instructions and involve 'W'. It is very easy to confuse loading a register with W and loading W with a register.

```
      MOVWF f               ; W IS MOVED TO THE REGISTER f, ( f is
changed )
```

```
      MOVF f, W             ; THE REGISTER f IS MOVED TO W, ( W is
changed )

      MOVF f, f             ; THE REGISTER f IS MOVED TO ITSELF
                            ;  f is not changed but flags may be set
```
Note that MOVWF is the only 'WF' instruction that doesn't have a choice of destination. It's always 'f'. The other 'WF' instructions are **ADDWF**, **SUBWF**, **ANDWF**, **SWAPWF**, **IORWF** & **XORWF**. In all of these cases one of 'W' or 'f' will be changed and the other not according to the destination. Also remember in SUBWF that 'W' is subtracted from 'f'. Other instructions where the destination is changed include:
```
      INC f,d    ; put the value of register f + 1 in either W or f
      DEC f,d    ; put the value of register f - 1 in either W or f
      COMP f,d   ; put the result of toggling all bits of f in destination
d
      SWAP f,d   ; put the result of swapping nibbles in f into d
      RLF f,d    ; the result of rotating f left thru carry goes into d
      RRF f,d    ; the result of rotating f right thru carry goes into d
```
If the destination is 'W' then only 'W' is affected; the original register remains the same.
IN SUBLW k, 'W' is subtracted from the literal 'k'.
It is easy to code GOTO when you meant to code CALL and vice-versa. You might think that this would cause your program to lock up but many times it just makes it act strangely instead.
Beware of using the same registers in two different routines, especially if one calls the other. For Exemplo if you use TEMP in a timing loop and then use TEMP in a subroutine that calls the timing loop, you might overlook the fact that the timing loop is changing TEMP.
The rotate instructions, (RLF,RRF), are rotates through carry so carry has to be set up prior to the instruction and rotates into the low or high order bit. Likewise the hi or low order bit rotates into carry.

FLOW CONTROL:

Normally a program will start with instruction 0 and skip to the next as each is executed. Instructions which change this involve a program counter. 'GOTO 666' would set the program counter to 666 and the instruction at location 666 would be executed next. 'CALL 666' on the other hand would first push the next location on the stack and then set the program counter to 666. Instructions after 666 would be executed until a RETURN ,RETLW k or RETIE instruction is encountered:
```
      RETURN - return from call, location of next instruction is
               popped from the stack and put into the program counter
      RETLW k - as RETURN but literal k is also placed into W
      RETIE - as RETURN but interrupts are also enabled
```
The stack consists of eight words and is circular, after the eighth word it rolls over to the first. Calls can be nested only eight deep. There is no push or pop instruction to access the stack directly.
There are four other flow instructions ,(beside CALL and GOTO), which might be called 'skip it' instructions:
```
      INCFSZ f,d - put f + 1 into either W or f. skips over the next
                   instruction if the result of the increment is
zero
      DECFSZ f,d - put f - 1 into either W or f. skips over the next
                   instruction if the result of the decrement is
zero
      BTFSC f,b - tests bit b of register f, skip next instruction if
                  the bit is zero (bit test skip clear). doesn't
                  change bit
      BTFSS f,b - tests bit b of register f, skip next instruction if
                  the bit is one (bit test skip set). doesn't
change bit
```

INPUT/OUTPUT PORTS:

The 16F84 has 13 pins that can be individually configured as either inputs or outputs. They a divided into PORTA, (5 bits), and PORTB, (8 bits). The direction of each bit is determined by the bits in the corresponding direction registers TRISA and TRISB. A zero means the bit will be an output, a 1 means input. To set up PORTB with alternating inputs and outputs:

```
MOVLW $AA               ; port pattern '10101010'
TRIS TRISB              ; W is placed into register TRISB
```

Certain port pins are also hooked to other functions of the processor. The high 4 bits of PORTB can be used as interrupt pins when they are programmed as inputs. The high bit of PORTA is also used as an external clock input for the counter/timer. Bit 0 of PORTB (RB0/INT) can be used for an external interrupt.

TIMING:

Often you wish to simply sit in a loop and wait for a specified period of time. Each instruction takes four clock cycles or 1 microsecond for a 4 Mhz crystal unless the program counter has to be changed, (flow control instruction). 2 microseconds are required for program branches. Here is a delay subroutine which will give a 1 millisecond delay for a 4 Mhz clock:

```
MSEC1           MOVLW $F9           ; allow for 4 microsec overhead..
                NOP                 ; (2 for CALL)
MICRO4          ADDLW $FF           ; subtract 1 from W
                BTFSS STATUS,Z      ; skip when you reach zero
                GOTO MICRO4         ; more loops
                RETURN
```

Some comments about the code:

- ? Each loop takes 4 microseconds; ADDLW takes 1, BTFSS takes 1 and GOTO takes 2 microseconds. When the skip is taken it is also 4.
- ? SUBLW 1 subtracts W from one, not the reverse. To subract one from W you add the two's compliment of 1 which is $FF.
- ? You are testing the zero bit, (Z) in the STATUS register which will be set when the subtraction results in zero.
- ? The bit test takes 1 microcsecond unless the skip is taken, in which case it takes 2 microseconds.
- ? You could call MICRO4 directly with the number of 4 microsec loops in W for delays in multiples of 4 microseconds. Remember that the call itself is 2 microseconds. Don't enter with 0 in W or you get a 256 microsec delay.
- ? Notice that the subroutine does not use a register, just 'W'.

For longer time periods, you are going to have to use a register. The following routine is entered with the number of milliseconds delay in 'W'. Up to a quarter of a second delays are possible (1 - 255 msec):

```
NMSEC           MOVWF CNTMSEC       ; W to msec count register
MSECLOOP        MOVLW  $F8          ; allow for 8 microsec overhead
                CALL MICRO4         ; 248 * 4 + 2 = 994 here
                NOP                 ; make rest of loop ...
                NOP                 ; add up to 6 microseconds
                DECFSZ CNTMSEC, f   ; decrement count, skip when zero
                GOTO MSECLOOP       ; more loops
                RETURN
```

COUNTER/TIMER:

There is a internal 8 bit counter/timer that sets a flag when it rolls over from 255 to zero. This can be used as a counter or timer. As a timer, it is connected to the internal clock and increments at the clock frequency divided by four. A flag can be polled to tell when time is up. The timer can also be set up to generate an interrupt when this happens. It wouldn't take long to count all the way up at 1 Mhz so a programmable 'prescaler' can be used. The prescaler can be set to give output pulses at ratios of 1:2,1:4,1:8 etc. up to 1:256, extending the timeout up to the tens of milliseconds range for a 4 Mhz clock.

In counter mode, input comes from pulses applied externally at the high bit of Port A. The prescaler can be inserted to count every second, forth, eighth etc. pulse. The input pin can also be set to count on either rising or falling transitions.

Various bits in the OPTION register are used to set up the counter/timer. The low three bits, (0-2), set the prescaler ratio. Bit 3 determines whether the prescaler is assigned to timer 0 or the watchdog timer. Only one can use the prescaler at any one time. Bit 5 decides if TMR0 register is used as a timer or is used as a counter of pulses from Port A bit 4, (RA4).

INTERRUPTS:

Sometimes you can't afford to just sit and wait for a flag to go high. The solution is to set up the timer to generate an interrupt. When the timer rolls over a flag is set, the address of the next operation is pushed on the stack and the program goes to location 4 and continues from there. This is usually a GOTO to the interrupt routine.
The interrupt routine does whatever you want to happen each time the interrupt occurs. Further interrupts are disabled when the interrupt starts. You are responsible for clearing the flag and re-enabling interrupts. RETIE pulls the saved instruction address off the stack and enables interrupts.
Three other situations can be set up to cause interrupts:

1. PORTB, bit 0 ,(RB0/INT), can be used as an external interrupt pin. A rising or falling edge can be used.
2. A change of state of any of the high 4 bits of PORTB.
3. A write to data EEPROM completion.

The enable and flag bits are the key to interrupts on the PIC. Each of these four situations has an associated flag bit and an enable bit. The flag bit set means the situation has happened. You have to reset these. The enable bit set means the setting of this particular flag can cause an interrupt. If in addition you want the interrupt to cause a jump to location 4, a Global Interrupt Enable, (GIE), flag must be set before the individual flag bit goes high. GIE is cleared at the interrupt condition and prevents further interrupts. RETIE resets GIE.

SLEEP MODE:

Sleep mode is a low current mode used to save battery life. The pic can draw as little as 50 microamps in sleep mode. The mode is started with the SLEEP instruction and can be ended by one of the following:

1. external reset on MCLR pin.
2. Watchdog timeout (if enabled).
3. Interrupt from:
    1. register B port change.
    2. RB0/INT pin.
    3. EEPROM write completion.

While in sleep mode instruction execution is suspended. In particular, timer 0 is not incrementing. Upon wakeup from sleep instruction exection continues from the stopped point or if GIE is set, instruction continues from location 4.

WATCHDOG TIMER:

The watchdog timer is independent of the PIC's internal clock. It times out from a CLRWDT ,(Clear Watchdog Timer), instruction in roughly 18 milliseconds. It is not very accurate. The prescaler can be assigned to the WDT, giving time out periods of up to a couple of seconds. The purpose of the WDT in normal use is to keep the PIC from going off into never-never land without the ability to recover. At timeout, a flag is cleared, (TO), the program counter is reset to 0000 and the program starts again. To prevent the reset, you to build a CLRWDT, (Clear Watch Dog Timer), instructions into your program before the timeout occurs.

EEPROM:

Reading or writing to data memory, (EEPROM) requires using magic series of instructions involving the registers EEADR, EEDATA, EECON1 and EECON2. EECON1 and EECO2 are in bank 1 so you have to do some bank switching.

ASCII MESSAGES:

Suppose you had the message "Hello World!" and wanted send it out as ASCII characters to a PC COM port. Data memory is scarce and hard to use so we put the message in program memory. How do we access it? We use a table. You load 'W' with the offset of the character you want and call MSGTXT. A 0 will return 'H', a 1 will return 'e' etc:

```
MSGTXT     ADDWF PCL, f              ; offset added to PCL
           RETLW $48                 ; 'H'
           RETLW $65                 ; 'e'
           RETLW $6C                 ; 'l'
           RETLW $6C                 ; 'l'
           RETLW $6F                 ; 'o'
```

```
                RETLW $20               ; ' '
                RETLW $57               ; 'W'
                RETLW $6F               ; 'o'
                RETLW $72               ; 'r'
                RETLW $6C               ; 'l'
                RETLW $64               ; 'd'
                RETLW $21               ; '!'
                RETLW $0D               ; carriage return
                RETLW $0A               ; line feed
                RETLW $00               ; indicates end
```

To output a character string you set up a register to point to the initial character, (MSGPTR), and repeatedly call MSGTXT, incrementing the pointer each time. We have reached the end of the string when a zero is returned. The routine is entered with the offset of the first character in 'W':

```
      OUTMSG    MOVWF MSGPTR    ; put 'W' into message pointer
      MSGLOOP   MOVF MSGPTR, W  ; put the offset in 'W'
                CALL MSGTXT     ; returns ASCII character in 'W'
                ADDLW 0         ; sets the zero flag if W = 0
                BTFSC STATUS, Z ; skip if zero bit not set
                RETURN          ; finished if W = 0
                CALL OUTCH      ; output the character
                INCF MSGPTR, f  ; point at next
                GOTO MSGLOOP    ; more characters
```

OUTPUT ASCII CHARACTERS:

Serial output of characters is just a matter of making an output go high or low at the proper times. It is normally high, and going low signals a start bit. At 4800 baud the bit time would be 1/4800 = 208 microseconds. Eight data bits, each one bit period, follow the start bit. A high level for longer than one bit period signifies stop bit/s. The bits are sent Least Significant Bit first. Sampling occurs midway in the bit period to determine if the bit is a 1 or 0.

RS232 high is -3V or lower, low is +3V or greater. You can actually get away with +5V for the low and 0V for the high if you keep the line short. Notice that this is flipped from what you might expect. We can use MICRO4 and 52 X 4 microsec loops for 1 bit time at 4800 baud. Actually 12 microseconds are used in overhead so we use 49 as the count. The subroutine is entered with the character to be output in 'W'. Port A bit 2, (pin 1), is use as output:

```
        OUTCH     MOVWF TXREG       ; put W into transmit register
                  MOVLW 8           ; eight bits of data
                  MOVWF BITS        ; a counter for bits
                  BSF PORTA, 2      ; start bit (flipped remember), RA2
        TXLOOP    MOVLW $31         ; 49 decimal, delay time
                  CALL MICRO4       ; wait 49 x 4 = 196 microseconds
                  RRF TXREG, f      ; roll rightmost bit into carry
                  BTFSC STATUS, C   ; if carry 0 want to set bit, ( a
low )
                  GOTO CLRBIT       ; else clear bit, ( a high )
                  BSF PORTA, 2      ; +5V on pin 1 ( RA2 )
                  GOTO TESTDONE     ; are we finished?
        CLRBIT    BCF PORTA, 2      ; 0V on pin 1 ( RA2 )
                  NOP               ; to make both options 12 micosec
        TESTDONE  DECFSZ BITS, f    ; 1 less data bit, skip when zero
                  GOTO TXLOOP       ; more bits left, delay for this one
                  MOVLW $34         ; full 208 microsec this time
                  CALL MICRO4       ; delay for last data bit
                  BCF PORTA, 2      ; 0V, ( a high ) for stop bits
                  MOVLW $68         ; decimal 104 delay for 2 stop bits
                  CALL MICRO4
                  RETURN
```

A PIC TRANSMITTER:

We now have almost all the code necessary to program a PIC to transmit the message 'Hello Word'. The MAIN routine might look like this:

```
MAIN        MOVLW 0           ; all port bit outputs
            TRIS TRISA        ; on port A
            TRIS TRISB        ; and port B
            CLRF PORTA        ; RA2 is 0 ( RS232 high )
            MOVLW $32         ; delay for 50 msec
            CALL NMSEC        ; so no glitches interfere
            MOVLW 0           ; this is offset of message
            CALL OUTMSG       ; output the message
ENDLESS     GOTO ENDLESS      ; go into an endless loop
```

Some comments about the code:

- All bits of all ports are set to outputs. Unconnected port bits should never be set up as inputs. When floating, CMOS inputs can flip rapidly between states and cause excessive current draw, heating and even damaging the PIC.
- The offset of the message is the address of the first character minus the address of MSGTXT minus 1. Other messages could be added to the table, everything has to stay in the first 256 addresses though to be reached by PCL, (program counter low).
- The order of subroutines doesn't matter; the main routine has to be the first thing reached from address 0000 though. Either it comes first or you have a 'GOTO MAIN' instruction at 0000.
- The GOTO at 0000 is a good idea because it can skip over location 4 which could be reserved for a GOTO to an interrupt subroutine.

HOOKING IT UP:

The actual connections to a PIC16F84 are few:

- pin 1: RA2 - a six foot or less wire to pin 2 (RX) of a DB9 female socket. If a DB25 is used, it is pin 3.
- pin 4: MCLR - pulled high to +5V with a few K resistor. You can also add a short wire that can be touched to ground to reset the PIC.
- pin 5: 0V - (gnd), the negative side of the power supply. Also going to ground is another six foot or less wire going to pin 5 of the DB9 or pin 7 of a DB25 if used instead.
- pin 14: +5V - the positive side of the power supply. A .1 mfd capacitor from here to ground wouldn't hurt but isn't necessary.
- pin 15: OSC1 - one outer pin of a 4Mz ceramic resonator with caps.
- pin 16: OSC2 - the other outer pin of the ceramic resonator. The center pin goes to ground.

CONFIGURATION MEMORY

Your programmer probably inputs information to be programmed into the PIC in Intel Hex Format (INHX8S). Files in this format have a '.HEX' extension. If you look at one of these files you will see lines that start with ':'. The first 9 characters of the line tell the number of bytes of data and where to put it in memory. Various different addresses are assigned to the program memory, the EEPOM memory or what is called the configuration word.

Various bits in the configuration word tell things like what type of oscillator is being used and whether the watchdog timer is enabled. Some programming software allows you to input this information as command line switches when you run the program. Others expect the information to already be in the .HEX file. In the case the PIC transmitter program, the oscillator type should be set to XT.

PROGRAMMING THE PIC

I use either the NOPPP programmer by Michael Covington from 'Electronics Now': http://mindspring.com/~covington/noppp/noppp.zip or the 'TOPIC' programmer designed by David Tait: http://www.man.ac.uk/~mbhstdj/files/topic03.zip. To produce the '.HEX' files, I use an assembler I wrote called 'Picbuild84': http://www.picpoint.com/download/projects/picbuild.zip. (local copy). Since Picbuild doesn't put configuration data in the HEX files and NOPPP software doesn't allow command line switches for this, I use the TOPIC software which does and works for both programmers. I've included the files 'HELLO.HEX' and 'HELLO.F84' for use with PicBuild84. Use the -x switch with TOPIC.

RUNNING THE PROGRAM

Fire up your favorite terminal program. Set it for 4800 baud, 8 bits, no parity and plug the connector from the PIC into the COM port. When you power the PIC, 'Hello World!' should be printed on the screen. Touching the wire from MCLR to ground briefly should print it again. If you don't have a terminal program and are running DOS you could enter the following QuickBasic program and run it instead:

```
CLS
OPEN "COM2:4800,N,8,2,CD0,CS0,DS0,OP0" FOR INPUT AS #1
DO
   PRINT INPUT$(1, #1);
LOOP
```

A program that prints out a single message over and over may be instructive but it isn't really very useful. There are many input devices that could be connected to the PIC. Serial devices are especially convenient. One such device is the Dallas Semiconductor DS1820 1 wire Digital Thermometer http://www.dalsemi.com/DocControl/PDFs/1820.pdf. It is available from Newark http://www.newark.com for about $6.

DS1820

The DS1820 is a computer, temperature sensor and serial port in a three pin package. It looks like a transistor and is only a little larger. There is a power pin, a ground and a single data I/O pin. The I/O pin is connected to a single line bus which can support multiple DS1820's. Only the simple situation with a single unit connected to a PIC will be discussed.

The bus is pulled high to +5V through a 4.7K resistor. Either the PIC or the 1820 can pull this line low. The PIC I/O pin is set to output and the port bit is set to 0 to pull the bus low. Changing the pin to input lets the resistor pull the line high. All timing originates with the PIC. Data bits are transferred in time slots initiated when the PIC pulls the bus down. If the bus is released,(I/O pin made input), and allowed to go high for the rest of the time slot, a 1 bit is sent. If the bus remains low it is a zero bit transfer, PIC to 1820. In either case, the bus has to be high at the end of the time slot and remain high for at least 1 microsecond before the next low. The slot must be at least 60 microseconds long. Bit timing is not critical as it is with RS232. Time slots can be any reasonable length over the minimum.

What about transfers the other way, 1820 to PIC? First, you have to send the 1820 a command to read it's scratchpad memory. The PIC must then generate read time slots to get the data bits out. The PIC pulls the bus low as before to initiate the time slot and quickly releases it. Data from the 1820 will be valid only for the first 15 microseconds. The read should take place toward the end of this interval. If the 1820 is pulling the bus low you receive a 0 bit, if not it is a 1 bit.

Any sequence of commands given to the 1820 by the PIC has to be preceded by a reset pulse and the address protocol, (64 bits). The reset pulse is a low from the PIC lasting about 600 microseconds followed by roughly a 400 microsecond period where the PIC has released the bus and can read a response 'presence pulse' from the 1820, (low), indicating everything is O.K. We will ignore this return pulse an assume everything is O.K. Commands given the 1820 include:

- $BE - Read contents of scratchpad. Up to 9 bytes can be read. We are interested in only the first two which contain the temperature. Reading can be interrupted with a reset pulse.
- $CC - Skip ROM - skip protocol for telling which device is to be addressed. There is only one slave device on the bus, (DS1820). This saves having to send out the 64 bit ROM code after each reset.
- $44 - Begin temperature conversion. Read the 1820 after this until a $FF is returned, indicating conversion is complete or wait 500 msec which is the maximum conversion time.

The temperature value is held in the first two bytes of the scratchpad memory. First to come out is the magnitude, the second is either $00 or $FF and represents the sign, ($FF is negative). The top 7 bits of the magnitude hold an integer number in binary, (0-125 degrees C). The lowest bit indicates if an additional 0.5 C is to be tacked on, (1), or not, (0).

An additional complication is negative numbers. The unit goes to -55 degrees C, but negative numbers are expressed in twos compliment form. You have to change these by taking the compliment, (flipping all the bits), and adding one to get the magnitude in normal form.

A 1 WIRE PROGRAM

An outline of a program to read and send temperature would look like this:

1. Send reset pulse, $CC & $44 to start a conversion.
2. Wait 500 milliseconds for conversion to take place.

3. Send reset pulse, $CC & $BE to begin memory read.
4. Receive value and sign bytes and save them.
5. Send reset pulse to terminate read.
6. If sign is $FF then send '-' out RS232.
7. Convert integer magnitude to decimal and send out RS232.
8. Send a '.' out RS232.
9. If LSB of value is 0 send a '0' else send '5' out RS232.
10. Send carriage return out RS232.
11. Delay until next reading.
12. Go to first step and repeat measurement.

The I/O bit used is Port A bit 3. The lowest level 1 wire routines look like this:

```
M1HI        BSF  STATUS, RP0       ; go to page 1
            BSF  TRISA, 3          ; make RA3 an input
            BCF  STATUS, RP0       ; back to page 0
            RETURN                 ; master 1 wire HI

M1LO        BCF  PORTA, 3          ; port A, bit 3 low
            BSF  STATUS, RP0       ; go to page 1
            BCF  TRISA, 3          ; make port A, bit 3 an output
            BCF  STATUS, RP0       ; back to page 0
            RETURN                 ; master 1 wire LO

SLOTLO     CALL M1LO          ; begin by taking RA3 low
           GOTO DLY80         ; and remain low
SLOTHI     CALL M1LO          ; a quick low on RA3
           CALL M1HI          ; return high right away
DLY80      MOVLW  $14         ; decimal 20 x 4 = 80 microsec
           CALL MICRO4        ; delay
           CALL M1HI          ; always end on high
           RETURN
```

The routine for transmission of characters looks much like the routine used for RS232. In fact, we can use the same registers:

```
TXBYTE1    MOVWF TXREG        ; put W into transmit register
           MOVLW 8            ; eight bits of data
           MOVWF BITS         ; a counter for bits
TX1LOOP    RRF TXREG, f       ; roll rightmost bit into carry
           BTFSC STATUS, C    ; if carry 0 want to send 0
           GOTO HIBIT1        ; else send 1 bit
           CALL SLOTLO        ; output low bit ( RA3 )
           GOTO DONE1         ; are we finished?
HIBIT1     CALL SLOTHI        ; output a high bit ( RA3 )
DONE1      DECFSZ BITS, f     ; 1 less data bit, skip when zero
           GOTO TXBYTE1       ; more bits left
           RETURN
```

The receive routine includes the read time slot generated by the PIC:

```
RXBYTE1    MOVLW 8            ; eight bits of data
           MOVWF BITS         ; a counter for bits
RX1LOOP    CALL  M1LO         ; a quick low pulse
           CALL  M1HI         ; and back high
           NOP                ; waiting for about ...
           NOP                ; 14 microseconds ...
           NOP                ; to elapse ...
           NOP                ; before reading RA3
           BSF STATUS, C      ; make carry a 1
           BTFSS PORTA, 3     ; read RA3 skip if received a 1
           BCF STATUS, C      ; make carry a 0
           RLF  RXREG, f      ; roll carry into RXREG LSB
           DECFSZ BITS, f     ; 1 less data bit, skip when zero
           GOTO RXLOOP1       ; more bits left
```

```
                RETURN
```
The reset pulse is straightforward:
```
        RESET   CALL  M1LO        ; make I/O line low
                MOVLW $96         ; 150 x 4 = 600 microseconds
                CALL MICRO4       ; delay
                CALL  M1HI        ; and back high
                MOVLW $64         ; 100 x 4 = 400 microseconds
                CALL MICRO4       ; delay
                RETURN
```
Except for the binary to decimal conversion routine, we now have almost all the code necessary to write a program to use the DS1820 to read temperature and send it to the PC COM port. The files 'DEGREES.HEX' and 'DEGREES.F86' can be entered into PicBuild. A LED on RB1 is use to give a quick blink at each reading indicating the unit is working. A normally open push button is connected between RB2 and ground to set up the delay between readings.

SETTING THE READING INTERVAL

Numbers which represent the interval between readings are kept in EEPROM data memory locations 1-13. They are the total delay in seconds - 1. The measurement and transmit cycle is set to take 1 second by itself. I usually program in the numbers 0,1,2,3,4,9,19,29,44,59,119,179 & 239. EEPROM location 0 holds a number 1 to 13 which points to one of these locations. To change this number you hold the button down while turning the power on. This causes a GOTO to a routine that puts a 1 in a location 0 and then increments it each second. The LED blinks once for each number. You release the button when the desired number is reached. The program then burns the pointer number into EEPROM location 0 and goes into an endless loop. You power down and when you power up again collection uses the delay interval pointed to. The push button is connected to RB2. Port B can have weak pullups activated by clearing bit 7 of the OPTION register. This saves having to add an external pullup resistor for the pin. All bits in the OPTION register ar 1's at startup.

WIRING THE DS1820

The center pin of the DS1820 is the I/O pin and is connected by a wire to RA3, (pin 2), of the PIC. A 4.7K resistor should be connected from pin 2 to +5 Volts. Looking down on the flat side of the DS1820, the ground pin on the left is wired to ground. The pin on the right is connected to +5 volts. The cathode of a LED is connected to ground and the anode is connected through a 1k resistor to RB1, (pin 7), of the PIC. Pin 8 of the PIC, (RB2), is connected to a normally open push button, the other side is connected to ground.

RUNNING THE DEGREES PROGRAM

Connect the 9 pin socket to the COM port on the PC. Run a terminal program setting 4800 baud, 8 bits, no parity and 2 stop bits. When the PIC is powered the temperature should appear, being updated according to the delay time set in EEPROM. Put the DS1820 between your fingers and the temperature should go up.
If the time interval between points is not correct, you turn the unit off and hold the button down when you power up. Hold the button until the proper number of flashes has occurred for the desired interval. Release the button, turn the unit off and power up again to start collecting data.
Here is a QuickBasic program to store readings in a file. The file is a text file with one reading per line. You should be able to read the file directly into a spreadsheet and graph the data: The program 'GrafData' will give you a quick graph of the data on the screen.
```
' *** Take N readings, change to Deg F, and store in file ***
CLS
INPUT "How many readings? ", readings%
INPUT "Filename for readings? ", filename$
OPEN "COM2:4800,N,8,2,CD0,CS0,DS0,OP0" FOR INPUT AS #1
IF filename$ <> "" THEN OPEN filename$ FOR OUTPUT AS #2
count% = 1
DO
   INPUT #1, temp!
   GOSUB ShowTemp
LOOP UNTIL count% > readings%
IF filename$ <> "" THEN CLOSE #2
END
```

```
ShowTemp:
   temp! = 9 * temp! / 5 + 32
   PRINT count%; " - ";
   PRINT USING "###.# Deg F"; temp!
   IF filename$ <> "" THEN PRINT #2, USING "###.#"; temp!
   count% = count% + 1
RETURN
```

I've also included a Pascal program to take data, (NTEMP.EXE) if you don't want to go to the trouble of loading QBasic. Notice how much extra code is necessary just to set up the COM port. COM2 is used by the program but can be changed to COM1 by making the changes noted and recompiling.

Now we have a useful device, but not as useful as it could be. After all, you don't want to lug around a computer just to measure temperature. We could use the EEPROM data memory in the PIC to collect data and carry the unit to the PC and dump them, but data memory is pretty small. The answer lies in adding more memory to the PIC and serial memory is definately the way to go. Adding a 8 pin 24C65 serial EEPROM from Microchip gives us 8K bytes of additional memory. These cost about $3 from Digi-Key. Using serial memory we can construct a reasonable temperature data logger.

24C65 MEMORY

The 24C65 memory communicates with the PIC over a 2 wire I2C bus. The two lines are SDA, (serial data), and SCL, (serial clock). The PIC will act as the master and the memory a slave. The master provides the clock signal and begins and terminates all transfers with start and stop signals. PIC and memory will both transmit and recieve information over SDA. SDA has to be pulled high with a resistor and the PIC pin has to be set to input to allow slaves to pull the line low, (like the DS1820 I/O line). The clock line would normally have to be the same but in this case there is only one master and one source of clock signal so it isn't necessary. Information transfer occurs only when the clock is high:

- ? The status of data bits is determined when the clock is high.
- ? A start condition occurs when SDA is taken low while SCL is high.
- ? A stop condition occurs when SDA is taken high while SCL is high.

Special care should be taken that SCL is low before any changes are made to SDA. Otherwise, it would be interpreted as a start or stop condition. Assume SDA is connected to RA1 and SCL is connected to RA0. The four most basic routines you need are:

- ? HIGH_SDA - Make Port A, bit 1 an input. SDA should not be made high by making the PIC an output and setting it high. If a slave then brought SDA low, it would be a direct short.
- ? LOW_SDA - Make Port A, bit 1 an output and put a 0 in PortA, 1.
- ? HIGH_SCL - make Port A, bit 0 a 1, (previously set to output).
- ? LOW_SCL - make Port A, bit 0 a 0.

Now routines using these four can be written. A start condition would look like this:

```
        START       CALL LOW_SCL    ; bring SCL low before a change to
SDA
                    CALL HIGH_SDA   ; SDA will start high
                    CALL HIGH_SCL   ; now set up for information
transfer
                    CALL LOW_SDA    ; start: SDA goes low while SCL high
                    CALL LOW_SCL    ; clock back to where SDA can change
                    RETURN
```

Notice that the clock is left in a condition where changes to SDA won't be seen as start or stop conditions. The code for a stop condition would be exactly the same except with the calls that change SDA reversed. The stop condition leaves SDA high and the bus released.

Once the slave sees the start condition it expects data bits to come from the PIC. The PIC transfers data by raising the clock for 5 microseconds or more and lowering it again. If SDA is high during this period a 1 is transferred if it is low, a 0. The PIC must also put out this clock pulse to receive data. It releases the bus (HIGH_SDA), brings SCL high and examines SDA. A high SDA means a 1 bit from memory, a low means a 0 bit. Clock lows should also last 5 microseconds or longer. This gives a maximum bit transfer rate of 100Kbits/s. Bit timing isn't important as long as it isn't too fast. The critical factor is what takes place on the SDA line when the SCL line is high .

The basic unit of transfer is the byte with an acknowledgement required after each byte is transferred. The PIC sends 8 bits, releases the bus and sends out a 9th clock pulse. If memory responds by pulling SDA low it means everything is O.K. The acknowledgement doesn't have to be acted upon but the clock pulse must be

sent. We call this response from memory NACK. Likewise when the PIC is receiving bytes from memory, memory expects an acknowledgement pulse, (SDA high & clock pulse), back after each byte. We call this pulse from the PIC ACK.
Unlike the case with the DS1820, there is no way to get around sending out a slave address after each start condition. The device address is contained in the high 7 bits of a control byte. The lowest bit of the byte tells if the operation is to be a read, (1), of the slave or a write to the slave, (0).
Transmission and reception of bytes of data over the 2 wire I2C is done by the routines 'TXBYTE2' and 'RXBYTE2' in the program 'TLOGGER'. Notice that RLF is used to transfer bytes into carry because the most significant bit comes first in this case.
Writing a byte to a random address in memory requires sending a control byte and address before the actual byte to be saved. It goes like this:

- ? Create start condition
- ? Send contol byte with LSB = 0
- ? Do NACK
- ? Send high byte of address
- ? Do NACK
- ? Send low byte of address
- ? Do NACK
- ? Send data byte to be written
- ? Do NACK
- ? Create stop condition
- ? Wait 25 milliseconds for byte to 'burn in'

We use this procedure to send both the value byte and sign byte from the DS1820 to memory. We also send the 7th and 8th byte ouput by the 1820 which will be used to break down the tenths of a degree of the reading. Four bytes are saved for each temperature reading. We start at memory location zero and increment the address after each is sent.
Reading a byte from memory also requires sending a start condition, control byte and address bytes to memory. The first 7 steps would be the same as for a write. Then a new start condition is created and a control byte with LSB =1 is sent. Only then can the PIC receive a byte by sending out clock pulses and testing SDA while the clock is high. The PIC must then switch into output mode and send an acknowledge pulse indicating the byte has been received, (ACK). Fortunately a sequential read can follow by continuing to read another byte after each ACK. When you are finished you produce a NACK instead of ACK and then produce a stop condition.
We either have to transfer a fixed amount of data or somehow remember how much data was saved. The latter approach is taken by requiring that a button be pushed before the power is turned off. At the button press, the program jumps to the code at FINISH which saves the current memory address to data EEPROM locations $0E and $0F. These two locations provide the first two bytes sent to the PC COM port when the 24C65 memory is dumped.

ADDING 24C65 MEMORY

An eight pin DIP socket is need with pins connected as follows:

- ? 1,2,3,4,7 - Ground
- ? 8 - +5 Volts
- ? 5 - SDA line goes to PIC pin 18, (RA1). There must also be a 10K resistor from pin 18 to +5 volts to pull SDA high.
- ? 6 - SCL line goes to PIC pin 17, (RA0).
- ? show a schematic [Note, pin numbers are shown looking at the chips from below].

In addition, some provision has to be made to signal a dump of 24C65 memory to the PC. A simple way is to add a PC boardjumper to ground on RB3, (pin 9). If the jumper is in place, control on start-up is transferred to 'MEMDUMP'. Note that if the jumper is not in place, collection of data will begin on power-up and memory will be overwritten. You may prefer a slide switch rather than a jumper.

USING THE DATALOGGER

To use the datalogger you simply turn it on and press the button when you want to start collecting data. The LED will blink as each four bytes for a data point is collected. If the time interval between readings is not correct, turn the unit off and reset it as described in the 'DEGREES' program.

After collecting data and before you turn the unit off, be sure to press and hold the button to save the stopping address. Hold the button down for at least a second or two. Put the jumper on to prevent loss of data if the unit is accidently turned on again.
While the jumper is on, connect to a PC COM2 port. Run the program 'READUMP' and turn the power on. Press the button to start data transfer. The datalogger doesn't convert the binary data to decimal like the 'DEGREES' program. This is done in the 'READUMP' program. Two other collected bytes are used to break the reading down to tenths of a degree. This is done more easily in the higher level language. It makes the graphs less 'blocky'. The file created by 'READUMP' can be loaded into a spreadsheet program or graphed with the program 'GRAFDATA'.

IMPROVEMENTS

The ceramic resonator seems to be accurate to only a percent or so. A quartz crystal would be more accurate but much larger. If you use a quartz crystal consider trimming the overall main loop to make it closer to one second. I've added coarse and fine loops just before getting the EEPROM delay. I set these roughly by seeing how much the blinking lagged over long periods of time and inserting appropriate numbers.
One thing I've thought of adding but haven't worked out yet is multiple collections of data. The idea would be to hold starting locations in data EEPROM and cycle through them, indicating which collection you want to dump. It would probably mean adding a line to transmit data to the PIC from the PC. A 22k resistor would be used to limit current from the RS232 TX line.
It would be nice to reduce the amount of current drawn by the unit to extend battery life. One way to do this would be to use SLEEP mode, but you would have to supply a low current external clock to do this because the PIC clock shuts down during SLEEP. A simpler way to save current is to put a switch in series with the LED to take it out of the circuit when not needed. The PIC itself seems to draw only a milliamp or so. It is also possible to run the circuit at 3 volts which reduces the current. A lower clock rate would help too but you might have to reduce the RS232 transmit baud rate.
A readout of temperature on the unit itself would be nice. The problem is that we only have 6 port bits left, (7 if you count RB1). It would probably mean quite a bit of extra hardware for miminimal advantage.
The conversion to tenths of a degree and decimal output as in 'DEGREES' could have been done in the PIC. I just thought it would be easier in a higher level language.

THANKS

I would like to thank all people that have posted PIC information on the Internet. After all, that is where I got all the material I learned from. I would especially like to thank:

- David Tait - TOPIC and simple demo programs. Links at http://www.man.ac.uk/~mbhstdj/piclinks.html.
- P.H. Anderson and his students at http://www.phanderson.com. Many code Exemplos including ones for DS1820 and 24C65. He also sells parts and kits at very reasonable prices.
- Microchip - Lots of application notes and spec sheets. Among the most useful for this project are DS30430B, (PIC16F8X) & DS21189B, (24LC64, I couldn't find a '65). http://www.microchip.com.
- Steve Marchant at http://www.nottingham.ac.uk/~cczsteve/pic/ds1820.asm has DS1820 support for the PIC16C84.
- Parallax Inc. - For their 'Pic Applications Handbook'. The RS232 sections were very helpful for this project. ftp://ftp.parallaxinc.com/pub/acrobat/picapps.pdf.

Written by Stan Ockers.
WWW space provided by Manchester University.
Links

http://www.nexuscomputing.com/~picarchive