# CAPÍTULO 1

## Introdução aos Microcontroladores



## Introdução

As circunstâncias que se nos deparam hoje no campo dos microcontroladores têm os seus primórdios no desenvolvimento da tecnologia dos circuitos integrados. Este desenvolvimento tornou possível armazenar centenas de milhares de transístores num único chip. Isso constituiu um pré-requisito para a produção de microprocessadores e, os primeiros computadores foram construídos adicionando periféricos externos tais como memória, linhas de entrada e saída, temporizadores e outros. Um crescente aumento do nível de integração, permitiu o aparecimento de circuitos integrados contendo simultaneamente processador e periféricos. Foi assim que o primeiro chip contendo um microcomputador e que mais tarde haveria de ser designado por microcontrolador, apareceu.

## História

É no ano de 1969 que uma equipa de engenheiros japoneses pertencentes à companhia BUSICOM chega aos Estados Unidos com a encomenda de alguns circuitos integrados para calculadoras a serem implementados segundo os seus projectos. A proposta foi entregue à INTEL e Marcian Hoff foi o responsável pela sua concretização. Como ele tinha tido experiência de trabalho com um computador (PC) PDP8, lembrou-se de apresentar uma solução substancialmente diferente em vez da construção sugerida. Esta solução pressupunha que a função do circuito integrado seria determinada por um programa nele armazenado. Isso significava que a configuração deveria ser mais simples, mas também era preciso muito mais memória que no caso do projecto proposto pelos engenheiros japoneses. Depois de algum tempo, embora os engenheiros japoneses tenham tentado encontrar uma solução mais fácil, a ideia de Marcian venceu e o primeiro microprocessador nasceu. Ao transformar esta ideia num produto concreto, Frederico Faggin foi de uma grande utilidade para a INTEL. Ele transferiu-se para a INTEL e, em somente 9 meses, teve sucesso na criação de um produto real a partir da sua primeira concepção. Em 1971, a INTEL adquiriu os direitos sobre a venda deste bloco integral. Primeiro eles compraram a licença à companhia BUSICOM que não tinha a mínima percepção do tesouro que possuía. Neste mesmo ano, apareceu no mercado um microprocessador designado por 4004. Este foi o primeiro microprocessador de 4 bits e tinha a velocidade de 6 000 operações por segundo. Não muito tempo depois, a companhia Americana CTC pediu à INTEL e à Texas Instruments um microprocessador de 8 bits para usar em terminais. Mesmo apesar de a CTC acabar por desistir desta ideia, tanto a Intel como a Texas Instruments continuaram a trabalhar no microprocessador e, em Abril de 1972, os primeiros microprocessadores de 8 bits apareceram no mercado com o nome de 8008. Este podia

endereçar 16KB de memória, possuía 45 instruções e tinha a velocidade de 300 000 operações por segundo. Esse microprocessador foi o pioneiro de todos os microprocessadores actuais. A Intel continuou com o desenvolvimento do produto e, em Abril de 1974 pôs cá fora um processador de 8 bits com o nome de 8080 com a capacidade de endereçar 64KB de memória, com 75 instruções e com preços a começarem em $360.

Uma outra companhia Americana, a Motorola, apercebeu-se rapidamente do que estava a acontecer e, assim, pôs no mercado um novo microprocessador de 8 bits, o 6800. O construtor chefe foi Chuck Peddle e além do microprocessador propriamente dito, a Motorola foi a primeira companhia a fabricar outros periféricos como os 6820 e 6850. Nesta altura, muitas companhias já se tinham apercebido da enorme importância dos microprocessadores e começaram a introduzir os seus próprios desenvolvimentos. Chuck Peddle deixa a Motorola para entrar para a MOS Technology e continua a trabalhar intensivamente no desenvolvimento dos microprocessadores.

Em 1975, na exposição WESCON nos Estados Unidos, ocorreu um acontecimento crítico na história dos microprocessadores. A MOS Technology anunciou que ia pôr no mercado microprocessadores 6501 e 6502 ao preço de $25 cada e que podia satisfazer de imediato todas as encomendas. Isto pareceu tão sensacional que muitos pensaram tratar-se de uma espécie de vigarice, considerando que os competidores vendiam o 8080 e o 6800 a $179 cada. Para responder a este competidor, tanto a Intel como a Motorola baixaram os seus preços por microprocessador para $69,95 logo no primeiro dia da exposição. Rapidamente a Motorola pôs uma acção em tribunal contra a MOS Technology e contra Chuck Peddle por violação dos direitos de autor por copiarem ao copiarem o 6800. A MOS Technology deixou de fabricar o 6501, mas continuou com o 6502. O 6502 é um microprocessador de 8 bits com 56 instruções e uma capacidade de endereçamento de 64KB de memória. Devido ao seu baixo custo, o 6502 torna-se muito popular e, assim, é instalado em computadores como KIM-1, Apple I, Apple II, Atari, Comodore, Acorn, Oric, Galeb, Orao, Ultra e muitos outros. Cedo aparecem vários fabricantes do 6502 (Rockwell, Sznertek, GTE, NCR, Ricoh e Comodore adquiriram a MOS Technology) que, no auge da sua prosperidade, chegou a vender microprocessadores à razão de 15 milhões por ano !

Contudo, os outros não baixaram os braços. Frederico Faggin deixa a Intel e funda a Zilog Inc.
Em 1976, a Zilog anuncia o Z80. Durante a concepção deste microprocessador, Faggin toma uma decisão crítica. Sabendo que tinha sido já desenvolvida uma enorme quantidade de programas para o 8080, Faggin conclui que muitos vão permanecer fieis a este microprocessador por causa das grandes despesas que adviriam das alterações a todos estes programas. Assim, ele decide que o novo microprocessador deve ser compatível com o 8080, ou seja, deve ser capaz de executar todos os programas que já tenham sido escritos para o 8080. Além destas características, outras características adicionais foram introduzidas, de tal modo que o Z80 se tornou um microprocessador muito potente no seu tempo. Ele podia endereçar directamente 64KB de memória, tinha 176 instruções, um grande número de registos, uma opção para refrescamento de memória RAM dinâmica, uma única alimentação, maior velocidade de funcionamento, etc. O Z80 tornou-se um grande sucesso e toda a gente se transferiu do 8080 para o Z80.
Pode dizer-se que o Z80 se constituiu sem sombra de dúvida como o microprocessador de 8 bits com maior sucesso no seu tempo. Além da Zilog, outros novos fabricantes como Mostek, NEC, SHARP e SGS apareceram. O Z80 foi o coração de muitos computadores como o Spectrum, Partner, TRS703, Z-3 e Galaxy, que foram aqui usados.

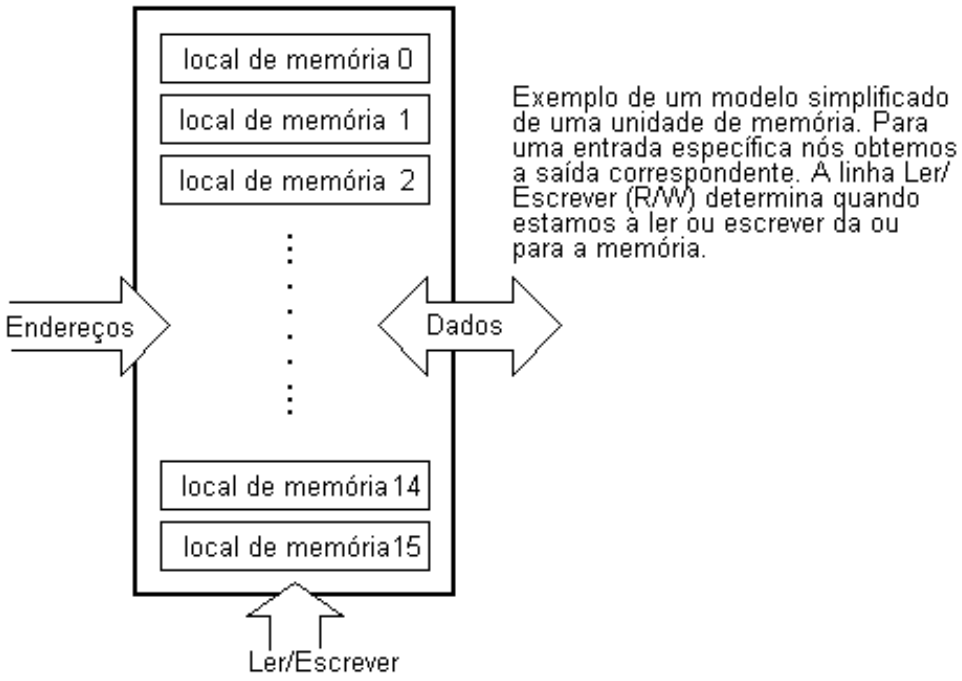
Em 1976, a Intel apareceu com uma versão melhorada do microprocessador de 8 bits e designada por 8085. Contudo, o Z80 era tão superior a este que, bem depressa, a Intel perdeu a batalha. Ainda que mais alguns microprocessadores tenham aparecido no mercado (6809, 2650, SC/MP etc.), já tudo estava então decidido. Já não havia mais grandes melhorias a introduzir pelos fabricantes que fundamentassem a troca por um novo microprocessador, assim, o 6502 e o Z80, acompanhados pelo 6800, mantiveram-se como os mais representativos microprocessadores de 8 bits desse tempo.

## Microcontroladores versus Microprocessadores

Um microcontrolador difere de um microprocessador em vários aspectos. Primeiro e o mais importante, é a sua funcionalidade. Para que um microprocessador possa ser usado, outros componentes devem-lhe ser adicionados, tais como memória e componentes para receber e enviar dados. Em resumo, isso significa que o microprocessador é o verdadeiro coração do computador. Por outro lado, o microcontrolador foi projectado para ter tudo num só. Nenhuns outros componentes externos são necessários nas aplicações, uma vez que todos os periféricos necessários já estão contidos nele. Assim, nós poupamos tempo e espaço na construção dos dispositivos.

## 1.1 Unidade de Memória

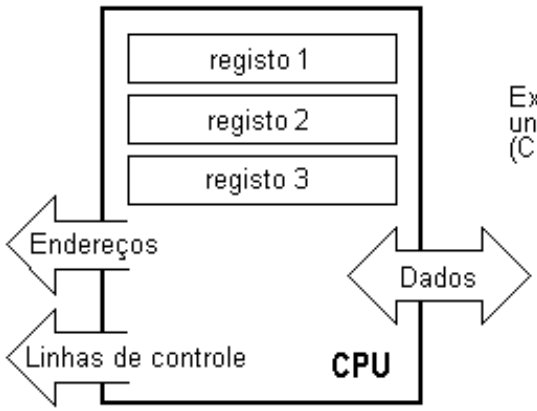
A memória é a parte do microcontrolador cuja função é guardar dados.
A maneira mais fácil de explicar é descrevê-la como uma grande prateleira cheia de gavetas. Se supusermos que marcamos as gavetas de modo a elas não se confundirem umas com as outras, então o seu conteúdo será facilmente acessível. Basta saber a designação da gaveta e o seu conteúdo será conhecido.




Os componentes de memória são exactamente a mesma coisa. Para um determinado endereço, nós obtemos o conteúdo desse endereço. Dois novos conceitos foram apresentados: endereçamento e memória. A memória é o conjunto de todos os locais de memória (gavetas) e endereçamento nada mais é que seleccionar um deles. Isto significa que precisamos de seleccionar o endereço desejado (gaveta) e esperar que o conteúdo desse endereço nos seja apresentado (abrir a gaveta). Além de ler de um local da memória (ler o conteúdo da gaveta), também é possível escrever num endereço da memória (introduzir um conteúdo na gaveta). Isto é feito utilizando uma linha adicional chamada linha de controle. Nós iremos designar esta linha por R/W (read/write) - ler/escrever. A linha de controle é usada do seguinte modo: se r/w=1, é executada uma operação de leitura, caso contrário é executada uma operação de escrita no endereço de memória.
A memória é o primeiro elemento, mas precisamos de mais alguns para que o nosso microcontrolador possa trabalhar.

## 1.2 Unidade Central de Processamento

Vamos agora adicionar mais 3 locais de memória a um bloco específico para que possamos ter as capacidades de multiplicar, dividir, subtrair e mover o seus conteúdos de um local de memória para outro. A parte que vamos acrescentar é chamada "central processing unit" (CPU) ou Unidade Central de Processamento. Os locais de memória nela contidos chamam-se registos.

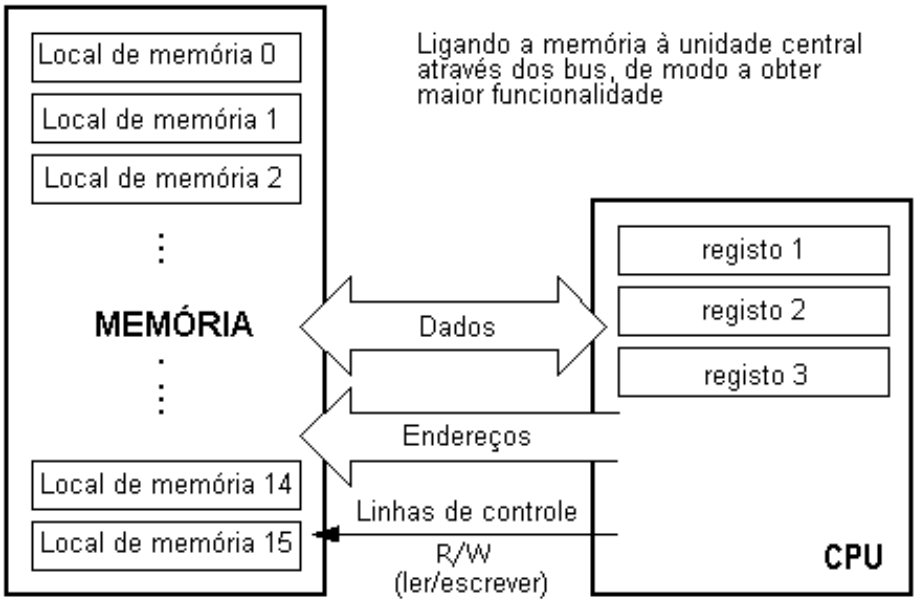



Exemplo simplificado de uma unidade central de processamento (CPU) com três registos.

Os registos são, portanto, locais de memória cujo papel é ajudar a executar várias operações matemáticas ou quaisquer outras operações com dados, quaisquer que sejam os locais em que estes se encontrem.
Vamos olhar para a situação actual. Nós temos duas entidades independentes (memória e CPU) que estão interligadas, deste modo, qualquer troca de dados é retardada bem como a funcionalidade do sistema é diminuída. Se, por exemplo, nós desejarmos adicionar os conteúdos de dois locais de memória e tornar a guardar o resultado na memória, nós necessitamos de uma ligação entre a memória e o CPU. Dito mais simplesmente, nós precisamos de obter um "caminho" através do qual os dados possam passar de um bloco para outro.

## 1.3 Bus

Este "caminho" designa-se por "bus" . Fisicamente ele corresponde a um grupo de 8, 16 ou mais fios.
Existem dois tipos de bus: bus de dados e de endereço. O número de linhas do primeiro depende da quantidade de memória que desejamos endereçar e o número de linhas do outro depende da largura da palavra de dados, no nosso caso é igual a oito. O primeiro bus serve para transmitir endereços do CPU para a memória e o segundo para ligar todos os blocos dentro do microcontrolador.

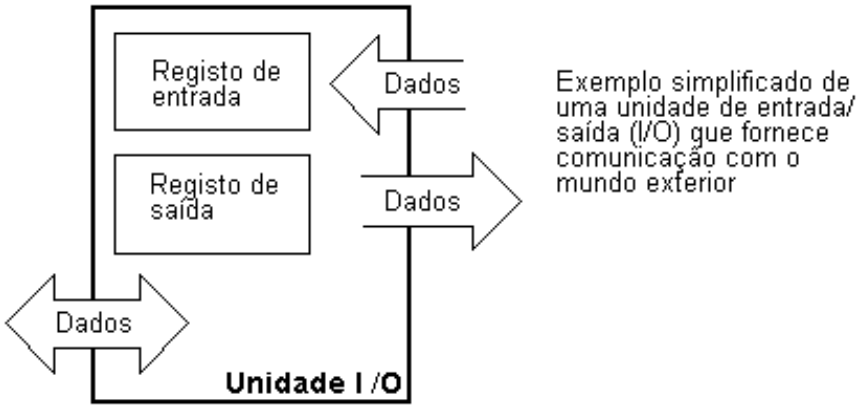



Ligando a memória à unidade central através dos bus, de modo a obter maior funcionalidade

Neste momento, a funcionalidade já aumentou mas um novo problema apareceu: nós temos uma unidade capaz de trabalhar sozinha, mas que não possui nenhum contacto com o mundo exterior, ou seja, conosco! De modo a remover esta deficiência, vamos adicionar um bloco que contém várias localizações de memória e que, de um lado, está ligado ao bus de dados e do outro às linhas de saída do microcontrolador que coincidem com pinos do circuito integrado e que, portanto, nós podemos ver com os nossos próprios olhos.

## 1.4 Unidade de entrada/saída

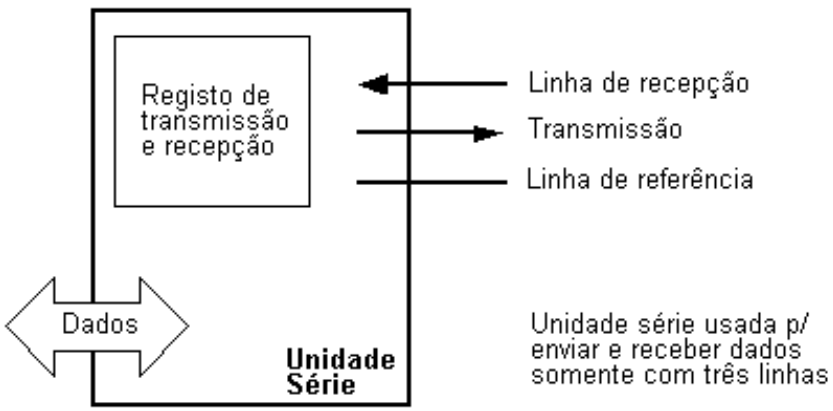
Estas localizações que acabamos de adicionar, chamam-se "portos". Existem vários tipos de portos: de entrada, de saída e de entrada/saída. Quando trabalhamos com portos primeiro de tudo é necessário escolher o porto com que queremos trabalhar e, em seguida, enviar ou receber dados para ou desse porto.




Quando se está a trabalhar com ele, o porto funciona como um local de memória. Qualquer coisa de que se está a ler ou em que se está a escrever e que é possível identificar facilmente nos pinos do microcontrolador.

## 1.5 Comunicação série

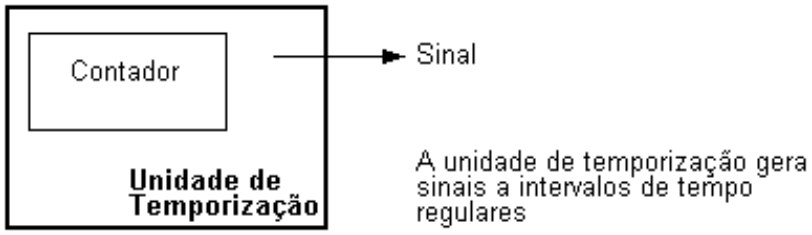
Anteriormente, acrescentámos à unidade já existente a possibilidade de comunicar com o mundo exterior. Contudo, esta maneira de comunicar tem os seus inconvenientes. Um dos inconvenientes básicos é o número de linhas que é necessário usarmos para transferir dados. E se for necessário transferi-los a uma distância de vários quilómetros? O número de linhas vezes o número de quilómetros não atesta a economia do projecto. Isto leva-nos a ter que reduzir o número de linhas de modo a que a funcionalidade se mantenha. Suponha que estamos a trabalhar apenas com três linhas e que uma linha é usada para enviar dados, outra para os receber e a terceira é usada como linha de referência tanto do lado de entrada como do lado da saída. Para que isto trabalhe nós precisamos de definir as regras para a troca de dados. A este conjunto de regras chama-se protocolo. Este protocolo deve ser definido com antecedência de modo que não haja mal entendidos entre as partes que estão a comunicar entre si. Por exemplo, se um homem está a falar em francês e o outro em inglês, é altamente improvável que efectivamente e rapidamente, ambos se entendam. Vamos supor que temos o seguinte protocolo. A unidade lógica "1" é colocada na linha de transmissão até que a transferência se inicie. Assim que isto acontece, a linha passa para nível lógico '0' durante um certo período de tempo (que vamos designar por T), assim, do lado da recepção ficamos a saber que existem dados para receber e, o mecanismo de recepção, vai activar-se. Regressemos agora ao lado da emissão e comecemos a pôr zeros e uns lógicos na linha de transmissão correspondentes aos bits, primeiro o menos significativo e finalmente o mais significativo. Vamos esperar que cada bit permaneça na linha durante um período de tempo igual a T, e, finalmente, depois do oitavo bit, vamos pôr novamente na linha o nível lógico "1" , o que assinala a transmissão de um dado. O protocolo que acabamos de descrever é designado na literatura profissional por NRZ (Não Retorno a Zero).



Como nós temos linhas separadas para receber e enviar, é possível receber e enviar dados (informação) simultaneamente. O bloco que possibilita este tipo de comunicação é designado por bloco de comunicação série. Ao contrário da transmissão em paralelo, aqui os dados movem-se bit após bit em série, daqui provém o nome de comunicação série. Depois de receber dados nós precisamos de os ler e guardar na memória, no caso da transmissão de dados o processo é inverso. Os dados vêm da memória através do bus para o local de transmissão e dali para a unidade de recepção de acordo com o protocolo.

## 1.6 Unidade de temporização

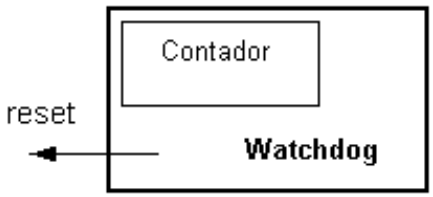
Agora que já temos a unidade de comunicação série implementada, nós podemos receber, enviar e processar dados.




Contudo, para sermos capazes de utilizar isto na indústria precisamos ainda de mais alguns blocos. Um deles é o bloco de temporização que nos interessa bastante porque pode dar-nos informações acerca da hora, duração, protocolo, etc. A unidade básica do temporizador é um contador que é na realidade um registo cujo conteúdo aumenta de uma unidade num intervalo de tempo fixo, assim, anotando o seu valor durante os instantes de tempo T1 e T2 e calculando a sua diferença, nós ficamos a saber a quantidade de tempo decorrida. Esta é uma parte muito importante do microcontrolador, cujo domínio vai requerer muita da nossa atenção.

## 1.7 Watchdog

Uma outra coisa que nos vai interessar é a fluência da execução do programa pelo microcontrolador durante a sua utilização. Suponha que como resultado de qualquer interferência (que ocorre frequentemente num ambiente industrial), o nosso microcontrolador pára de executar o programa ou, ainda pior, desata a trabalhar incorrectamente.



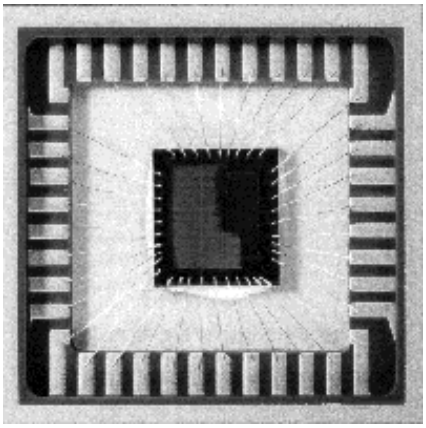

Claro que, quando isto acontece com um computador, nós simplesmente carregamos no botão de reset e continuamos a trabalhar. Contudo, no caso do microcontrolador nós não podemos resolver o nosso problema deste modo, porque não temos botão. Para ultrapassar este obstáculo, precisamos de introduzir no nosso modelo um novo bloco chamado watchdog (cão de guarda). Este bloco é de facto outro contador que está continuamente a contar e que o nosso programa põe a zero sempre que é executado correctamente. No caso de o programa "encravar", o zero não vai ser escrito e o contador, por si só, encarregar-se-á de fazer o reset do microcontrolador quando alcançar o seu valor máximo. Isto vai fazer com que o programa corra de novo e desta vez correctamente. Este é um elemento importante para que qualquer programa se execute fiavelmente, sem precisar da intervenção do ser humano.

## 1.8 Conversor analógico - digital

Como os sinais dos periféricos são substancialmente diferentes daqueles que o microcontrolador pode entender

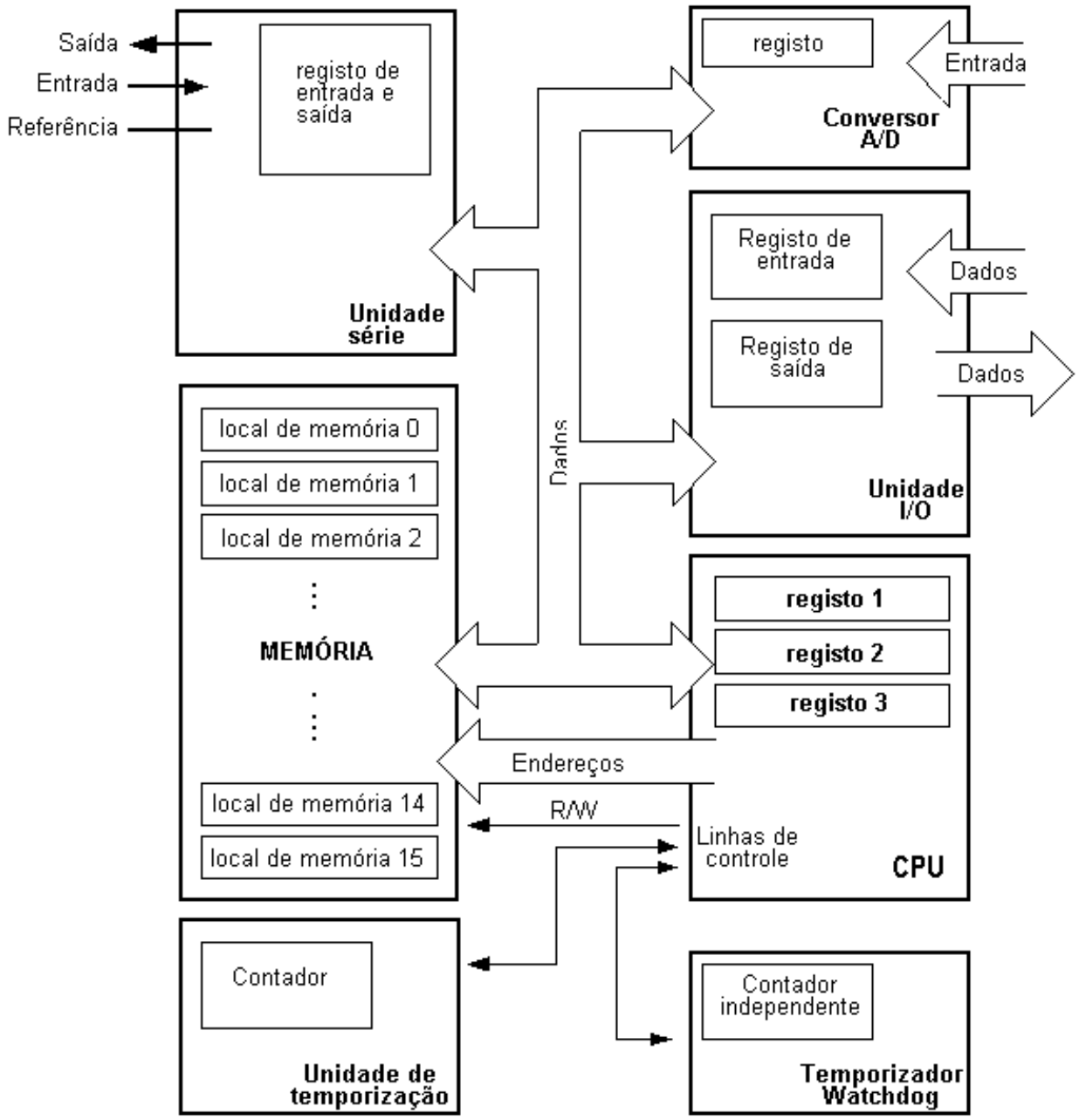
(zero e um), eles devem ser convertidos num formato que possa ser compreendido pelo microcontrolador. Esta tarefa é executada por intermédio de um bloco destinado à conversão analógica-digital ou com um conversor A/D. Este bloco vai ser responsável pela conversão de uma informação de valor analógico para um número binário e pelo seu trajecto através do bloco do CPU, de modo a que este o possa processar de imediato.


Neste momento, a configuração do microcontrolador está já terminada, tudo o que falta é introduzi-la dentro de um aparelho electrónico que poderá aceder aos blocos internos através dos pinos deste componente. A figura a seguir, ilustra o aspecto interno de um microcontrolador.



**Configuração física do interior de um microcontrolador**

As linhas mais finas que partem do centro em direcção à periferia do microcontrolador correspondem aos fios que interligam os blocos interiores aos pinos do envólucro do microcontrolador. O gráfico que se segue representa a parte principal de um microcontrolador.




**Esquema de um microcontrolador com os seus elementos básicos e ligações internas**.

Numa aplicação real, um microcontrolador, por si só, não é suficiente. Além dele, nós necessitamos do programa que vai ser executado e de mais alguns elementos que constituirão um interface lógico para outros elementos (que vamos discutir em capítulos mais à frente).

## 1.9 Programa

Escrever um programa é uma parte especial do trabalho com microcontroladores e é designado por "programação". Vamos tentar escrever um pequeno programa numa linguagem que seremos nós a criar e que toda a gente será capaz de compreender.

```
INICIO
REGISTO1=LOCAL_DE_ MEMORIA_A
REGISTO2=LOCAL_DE_ MEMORIA_B
PORTO_A=REGISTO1+REGISTO2
FIM
```

O programa adiciona os conteúdos de dois locais de memória e coloca a soma destes conteúdos no porto A. A primeira linha do programa manda mover o conteúdo do local de memória "A" para um dos registos da unidade central de processamento. Como necessitamos também de outra parcela, vamos colocar o outro conteúdo noutro registo da unidade central de processamento (CPU). A instrução seguinte pede ao CPU para adicionar os conteúdos dos dois registos e enviar o resultado obtido para o porto A, de modo a que o resultado desta adição seja visível para o mundo exterior. Para um problema mais complexo, naturalmente o programa que o resolve será maior.
A tarefa de programação pode ser executada em várias linguagens tais como o Assembler, C e Basic que são as linguagens normalmente mais usadas. O Assembler pertence ao grupo das linguagens de baixo nível que implicam um trabalho de programação lento, mas que oferece os melhores resultados quando se pretende poupar espaço de memória e aumentar a velocidade de execução do programa. Como se trata da linguagem mais frequentemente usada na programação de microcontroladores, ela será discutida num capítulo mais adiantado. Os programas na linguagem C são mais fáceis de se escrever e compreender, mas, também, são mais lentos a serem executados que os programas assembler. Basic é a mais fácil de todas para se aprender e as suas instruções são semelhantes à maneira de um ser humano se exprimir, mas tal como a linguagem C, é também de execução mais lenta que o assembler. Em qualquer caso, antes que escolha entre uma destas linguagens, precisa de examinar cuidadosamente os requisitos de velocidade de execução, de espaço de memória a ocupar e o tempo que vai demorar a fazer o programa em assembly.
Depois de o programa estar escrito, nós necessitamos de introduzir o microcontrolador num dispositivo e pô-lo a trabalhar. Para que isto aconteça, nós precisamos de adicionar mais alguns componentes externos. Primeiro temos que dar vida ao microcontrolador fornecendo-lhe a tensão (a tensão eléctrica é necessária para que qualquer instrumento electrónico funcione) e o oscilador cujo papel é análogo ao do coração que bate no ser humano. A execução das instruções do programa é regulada pelas pulsações do oscilador. Logo que lhe é aplicada a tensão, o microcontrolador executa uma verificação dele próprio, vai para o princípio do programa e começa a executá-lo. O modo como o dispositivo vai trabalhar depende de muitos parâmetros, os mais importantes dos quais são a competência da pessoa que desenvolve o hardware e do programador que, com o seu programa, deve tirar o máximo do dispositivo.

# CAPÍTULO 2

## Microcontrolador PIC16F84



## Introdução

O **PIC 16F84** pertence a uma classe de microcontroladores de 8 bits, com uma arquitectura RISC. A estrutura genérica é a do mapa que se segue, que nos mostra os seus blocos básicos.

**Memória de programa** (FLASH) - para armazenar o programa que se escreveu.
Como a memória fabricada com tecnologia FLASH pode ser programa e limpa mais que uma vez. ela torna-se adequada para o desenvolvimento de dispositivos.

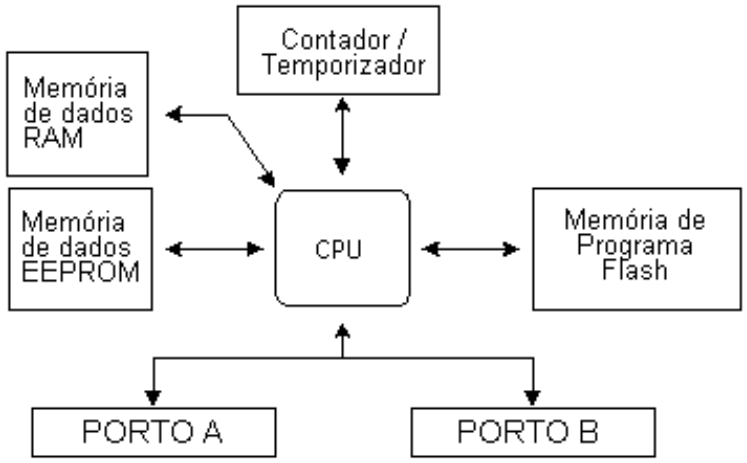
**EEPROM** - memória dos dados que necessitam de ser salvaguardados quaando a alimentação é desligada. Normalmente é usada para guardar dados importantes que não se podem perder quando a alimentação, de repente, “vai abaixo”. Um exemplo deste tipo de dados é a temperatura fixada para os reguladores de temperatura. Se, durante uma quebra de alimentação, se perdessem dados, nós precisaríamos de proceder a um novo ajustamento quando a alimentação fosse restabelecida. Assim, o nosso dispositivo, perderia eficácia.

**RAM** - memória de dados usada por um programa, durante a sua execução.
Na RAM, são guardados todos os resultados intermédios ou dados temporários durante a execução do programa e que não são cruciais para o dispositivo, depois de ocorrer uma falha na alimentação.

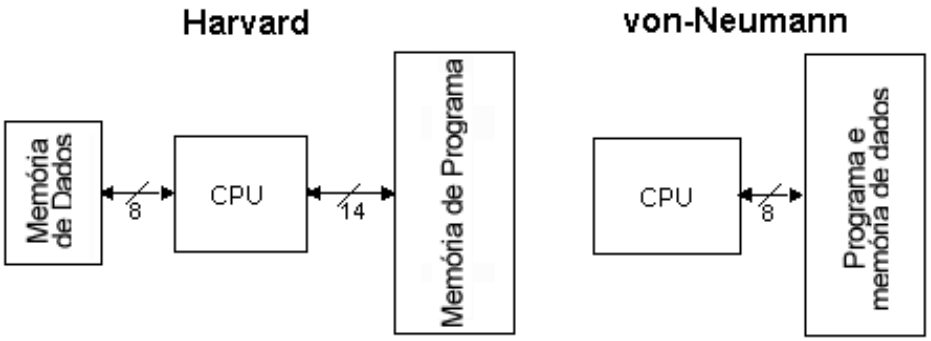
**PORTO A e PORTO B** são ligações fisicas entre o microcontrolador e o mundo exterior. O porto A tem cinco pinos e o porto B oito pinos.

**CONTADOR/TEMPORIZADOR** é um registo de 8 bits no interior do microcontrolador que trabalha independentemente do programa. No fim de cada conjunto de quatro ciclos de relógio do oscilador, ele incrementa o valor armazenado, até atingir o valor máximo (255), nesta altura recomeça a contagem a partir de zero. Como nós sabemos o tempo exacto entre dois incrementos sucessivos do conteúdo do temporizador, podemos utilizar este para medir intervalos de tempo, o que o torna muito útil em vários dispositivos.

**UNIDADE DE PROCESSAMENTO CENTRAL** faz a conexão com todos os outros blocos do microcontrolador. Ele coordena o trabalho dos outros blocos e executa o programa do utilizador.




**Esquema do microcontrolador PIC16F84**




**Arquitecturas Harvard versus von Neumann**

## CISC, RISC

Já foi dito que o PIC16F84 tem uma arquitectura RISC. Este termo é encontrado, muitas vezes, na literatura sobre computadores e necessita de ser explicada aqui, mais detalhadamente. A arquitectura de Harvard é um conceito mais recente que a de von-Neumann. Ela adveio da necessidade de pôr o microcontrolador a trabalhar mais rapidamente. Na arquitectura de Harvard, a memória de dados está separada da memória de programa. Assim, é possível uma maior fluência de dados através da unidade central de processamento e, claro, uma maior velocidade de funcionamento. A separação da memória de dados da memória de programa, faz com que as instruções possam ser representadas por palavras de mais que 8 bits. O PIC16F84, usa 14 bits para cada instrução, o que permite que que todas as instruções ocupem uma só palavra de instrução. É também típico da arquitectura Harvard ter um reportório com menos instruções que a de von-Neumann's, instruções essas, geralmente executadas apenas num único ciclo de relógio.

Os microcontroladores com a arquitectura Harvard, são também designados por "microcontroladores RISC". RISC provém de Computador com um Conjunto Reduzido de Instruções (Reduced Instruction Set Computer). Os microcontroladores com uma arquitectura von-Neumann são designados por 'microcontroladores CISC'. O nome CISC deriva de Computador com um Conjunto Complexo de Instruções (Complex Instruction Set Computer). Como o PIC16F84 é um microcontrolador RISC, disso resulta que possui um número reduzido de instruções, mais precisamente 35 (por exemplo, os microcontroladores da Intel e da Motorola têm mais de cem instruções). Todas estas instruções são executadas num único ciclo, excepto no caso de instruções de salto e de ramificação. De acordo com o que o seu fabricante refere, o PIC16F84 geralmente atinge resultados de 2 para 1 na compressão de código e 4 para 1 na velocidade, em relação aos outros microcontroladores de 8 bits da sua classe.

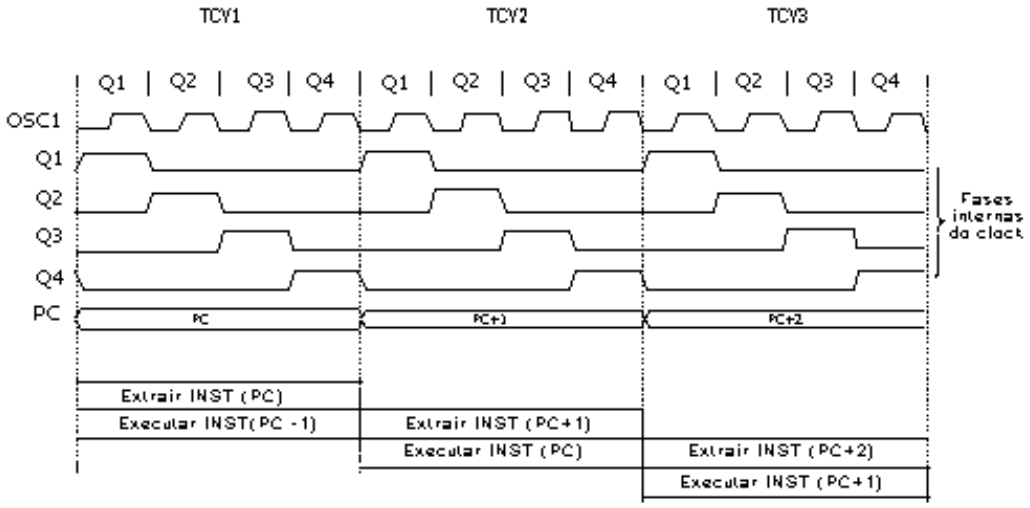
## Aplicações

O PIC16F84, é perfeitamente adequado para muitas variedades de aplicações, como a indústria automóvel, sensores remotos, fechaduras eléctricas e dispositivos de segurança. É também um dispositivo ideal para cartões inteligentes, bem como para dispositivos alimentados por baterias, por causa do seu baixo consumo.
A memória EEPROM, faz com que se torne mais fácil usar microcontroladores em dispositivos onde o armazenamento permanente de vários parâmetros, seja necessário (códigos para transmissores, velocidade de um motor, frequências de recepção, etc.). O baixo custo, baixo consumo, facilidade de manuseamento e flexibilidade fazem com que o PIC16F84 se possa utilizar em áreas em que os microcontroladores não eram anteriormente empregues (exemplo: funções de temporização, substituição de interfaces em sistemas de grande porte, aplicações de coprocessamento, etc.).
A possibilidade deste chip de ser programável no sistema (usando somente dois pinos para a transferência de dados), dão flexibilidade do produto, mesmo depois de a sua montagem e teste estarem completos.
Esta capacidade, pode ser usada para criar linhas de produção e montagem, para armazenar dados de calibragem disponíveis apenas quando se proceder ao teste final ou, ainda, para aperfeiçoar os programas presentes em produtos acabados.

## Relógio / ciclo de instrução

O relógio (clock), é quem dá o sinal de partida para o microcontrolador e é obtido a partir de um componente externo chamado “oscilador”. Se considerasse-mos que um microcontrolador era um relógio de sala, o nosso clock corresponderia ao pêndulo e emitiria um ruído correspondente ao deslocar do pêndulo. Também, a força usada para dar corda ao relógio, podia comparar-se à alimentação eléctrica.

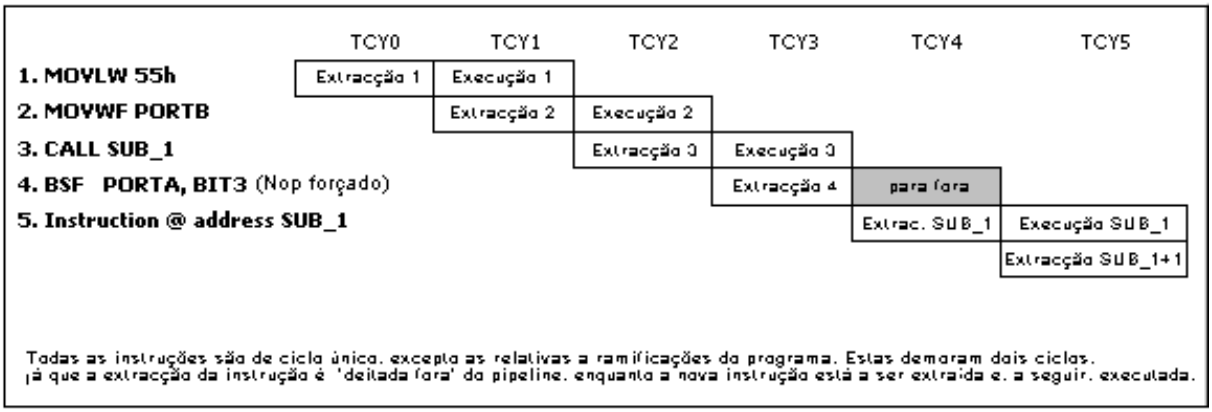
O clock do oscilador, é ligado ao microcontrolador através do pino OSC1, aqui, o circuito interno do microcontrolador divide o sinal de clock em quatro fases, Q1, Q2, Q3 e Q4 que não se sobrepõem. Estas quatro pulsações perfazem um ciclo de instrução (também chamado ciclo de máquina) e durante o qual uma instrução é executada.
A execução de uma instrução, é antecedida pela extracção da instrução que está na linha seguinte. O código da instrução é extraído da memória de programa em Q1 e é escrito no registo de instrução em Q4.
A descodificação e execução dessa mesma instrução, faz-se entre as fases Q1 e Q4 seguintes. No diagrama em baixo, podemos observar a relação entre o ciclo de instrução e o clock do oscilador (OSC1) assim como as fases Q1-Q4.
O contador de programa (Program Counter ou PC) guarda o endereço da próxima instrução a ser executada.




## Pipelining

Cada ciclo de instrução inclui as fases Q1, Q2, Q3 e Q4. A extracção do código de uma instrução da memória de programa, é feita num ciclo de instrução, enquanto que a sua descodificação e execução, são feitos no ciclo de instrução seguinte. Contudo, devido à sobreposição – pipelining (o microcontrolador ao mesmo tempo que executa uma instrução extrai simultaneamente da memória o código da instrução seguinte), podemos considerar que, para efeitos práticos, cada instrução demora um ciclo de instrução a ser executada. No entanto, se a instrução provocar

uma mudança no conteúdo do contador de programa (PC), ou seja, se o PC não tiver que apontar para o endereço seguinte na memória de programa, mas sim para outro (como no caso de saltos ou de chamadas de subrotinas), então deverá considerar-se que a execução desta instrução demora dois ciclos. Isto acontece, porque a instrução vai ter que ser processada de novo, mas, desta vez, a partir do endereço correcto. O ciclo de chamada começa na fase Q1, escrevendo a instrução no registo de instrução (Instruction Register – IR). A descodificação e execução continua nas fases Q2, Q3 e Q4 do clock.

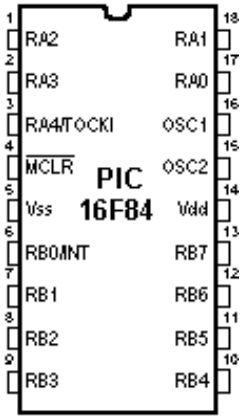



### Fluxograma das Instruções no Pipeline

**TCY0** é lido da memória o código da instrução MOVLW 55h (não nos interessa a instrução que foi executada, por isso não está representada por rectângulo).
**TCY1** é executada a instrução MOVLW 55h e é lida da memória a instrução MOVWF PORTB.
**TCY2** é executada a instrução MOVWF PORTB e lida a instrução CALL SUB_1.
**TCY3** é executada a chamada (call) de um subprograma CALL SUB_1 e é lida a instrução BSF PORTA,BIT3. Como esta instrução não é a que nos interessa, ou seja, não é a primeira instrução do subprograma SUB_1, cuja execução é o que vem a seguir, a leitura de uma instrução tem que ser feita de novo. Este é um bom exemplo de uma instrução a precisar de mais que um ciclo.
**TCY4** este ciclo de instrução é totalmente usado para ler a primeira instrução do subprograma no endereço SUB_1.
**TCY5** é executada a primeira instrução do subprograma SUB_1 e lida a instrução seguinte.

## Significado dos pinos

O PIC16F84 tem um total de 18 pinos. É mais frequentemente encontrado num tipo de encapsulamento DIP18, mas, também pode ser encontrado numa cápsula SMD de menores dimensões que a DIP. DIP é uma abreviatura para Dual In Package (Empacotamento em duas linhas). SMD é uma abreviatura para Surface Mount Devices (Dispositivos de Montagem em Superfície), o que sugere que os pinos não precisam de passar pelos orifícios da placa em que são inseridos, quando se solda este tipo de componente.




Os pinos no microcontrolador PIC16F84, têm o seguinte significado:

Pino nº 1, **RA2** Segundo pino do porto A. Não tem nenhuma função adicional.
Pino nº 2, **RA3** Terceiro pino do porto A. Não tem nenhuma função adicional.
Pino nº 3, **RA4** Quarto pino do porto A. O TOCK1 que funciona como entrada do temporizador, também utiliza este pino.
Pino nº 4, **MCLR** Entrada de reset e entrada da tensão de programação Vpp do microcontrolador .
Pino nº 5, **Vss** massa da alimentação.
Pino nº 6, **RB0**, bit 0 do porto B. Tem uma função adicional que é a de entrada de interrupção.
Pino nº 7, **RB1** bit 1do porto B. Não tem nenhuma função adicional.
Pino nº 8, **RB2** bit 2 do porto B. Não tem nenhuma função adicional.
Pino nº 9, **RB3** bit 3 do porto B. Não tem nenhuma função adicional.
Pino nº 10, **RB4** bit 4 do porto B. Não tem nenhuma função adicional.
Pino nº 11, **RB5** bit 5 do porto B. Não tem nenhuma função adicional.
Pino nº 12, **RB6** bit 6 do porto B. No modo de programa é a linha de clock
Pino nº 13, **RB7** bit 7 do porto B. Linha de dados no modo de programa
Pino nº 14, **Vdd** Pólo positivo da tensão de alimentação.
Pino nº 15, **OSC2** para ser ligado a um oscilador.
Pino nº 16, **OSC1** para ser ligado a um oscilador.
Pino nº 17, **RA0** bit 0 do porto A. Sem função adicional.
Pino nº 18, **RA1** bit 1 do porto A. Sem função adicional.

## 2.1 Gerador de relógio – oscilador

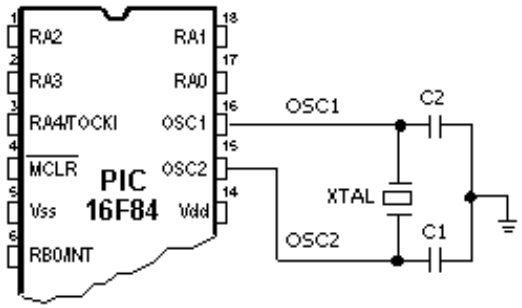
O circuito do oscilador é usado para fornecer um relógio (clock), ao microcontrolador. O clock é necessário para que o microcontrolador possa executar um programa ou as instruções de um programa.

### Tipos de osciladores

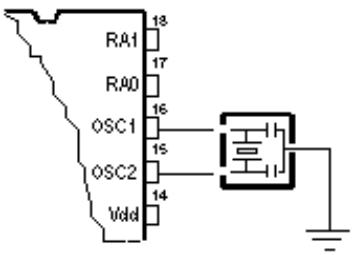
O PIC16F84 pode trabalhar com quatro configurações de oscilador. Uma vez que as configurações com um oscilador de cristal e resistência-condensador (RC) são aquelas mais frequentemente usadas, elas são as únicas que vamos mencionar aqui.
Quando o oscilador é de cristal, a designação da configuração é de XT, se o oscilador for uma resistência em série com um condensador, tem a designação RC. Isto é importante, porque há necessidade de optar entre os diversos tipos de oscilador, quando se escolhe um microcontrolador.

### Oscilador XT

O oscilador de cristal está contido num envólucro de metal com dois pinos onde foi escrita a frequência a que o cristal oscila. Dois condensadores cerâmicos devem ligar cada um dos pinos do cristal à massa. Casos há em que cristal e condensadores estão contidos no mesmo encapsulamento, é também o caso do ressonador cerâmico ao lado representado. Este elemento tem três pinos com o pino central ligado à massa e os outros dois pinos ligados aos pinos OSC1 e OSC2 do microcontrolador. Quando projectamos um dispositivo, a regra é colocar o oscilador tão perto quanto possível do microcontrolador, de modo a evitar qualquer interferência nas linhas que ligam o oscilador ao microcontrolador.



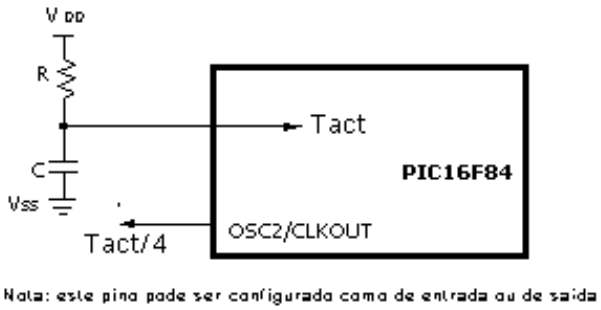
Clock de um microcontrolador a partir de um cristal de quartzo



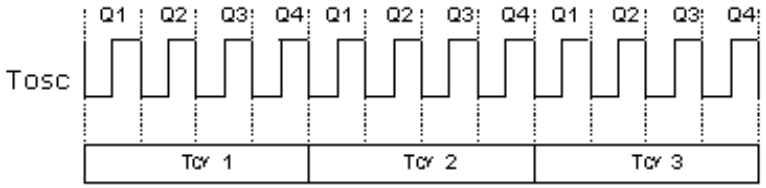
Clock de um microcontrolador com um ressonador

### OSCILADOR RC

Em aplicações em que a precisão da temporização não é um factor crítico, o oscilador RC torna-se mais económico. A frequência de ressonância do oscilador RC depende da tensão de alimentação, da resistência R, capacidade C e da temperatura de funcionamento.

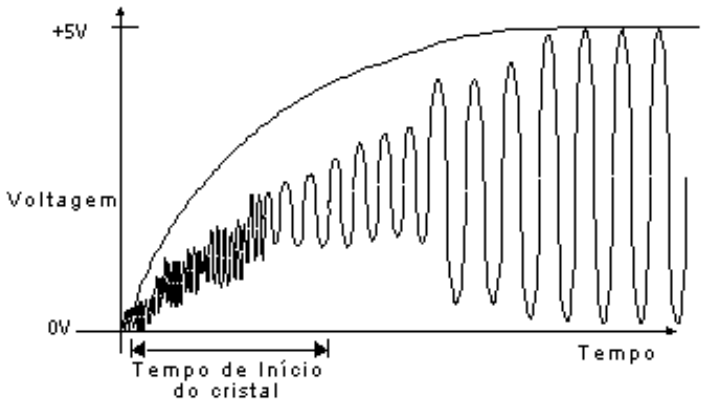



O diagrama acima, mostra como um oscilador RC deve ser ligado a um PIC16F84. Com um valor para a resistência R abaixo de 2,2 K, o oscilador pode tornar-se instável ou pode mesmo parar de oscilar. Para um valor muito grande R (1M por exemplo), o oscilador torna-se muito sensível à humidade e ao ruído. É recomendado que o valor da resistência R esteja compreendido entre 3K e 100K. Apesar de o oscilador poder trabalhar sem condensador externo (C = 0 pF), é conveniente, ainda assim, usar um condensador acima de 20 pF para evitar o ruído e aumentar a estabilidade. Qualquer que seja o oscilador que se está a utilizar, a frequência de trabalho do microcontrolador é a do oscilador dividida por 4. A frequência de oscilação dividida por 4 também é fornecida no pino OSC2/CLKOUT e, pode ser usada, para testar ou sincronizar outros circuitos lógicos pertencentes ao sistema.




Relação entre o sinal de clock e os ciclos de instrução

Ao ligar a alimentação do circuito, o oscilador começa a oscilar. Primeiro com um período de oscilação e uma amplitude instáveis, mas, depois de algum tempo, tudo estabiliza.



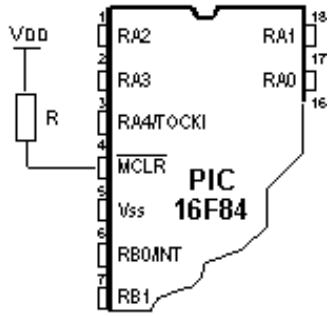

Sinal de clock do oscilador do microcontrolador depois de ser ligada a alimentação

Para evitar que esta instabilidade inicial do clock afecte o funcionamento do microcontrolador, nós necessitamos de manter o microcontrolador no estado de reset enquanto o clock do oscilador não estabiliza. O diagrama em cima, mostra uma forma típica do sinal fornecido por um oscilador de cristal de quartzo ao microcontrolador quando se liga a alimentação.

## 2.2 Reset

O reset é usado para pôr o microcontrolador num estado conhecido. Na prática isto significa que às vezes o microcontrolador pode comportar-se de um modo inadequado em determinadas condições indesejáveis. De modo a que o seu funcionamento normal seja restabelecido, é preciso fazer o reset do microcontrolador, isto significa que todos os seus registos vão conter valores iniciais pré-definidos, correspondentes a uma posição inicial. O reset não é usado somente quando o microcontrolador não se comporta da maneira que nós queremos, mas, também pode ser usado, quando ocorre uma interrupção por parte de outro dispositivo, ou quando se quer que o microcontrolador esteja pronto para executar um programa .

De modo a prevenir a ocorrência de um zero lógico acidental no pino MCLR (a linha por cima de MCLR significa o sinal de reset é activado por nível lógico baixo), o pino MCLR tem que ser ligado através de uma resistência ao lado positivo da alimentação. Esta resistência deve ter um valor entre 5 e 10K. Uma resistência como esta, cuja função é conservar uma determinada linha a nível lógico alto, é chamada “resistência de pull up”.




Utilização do circuito interno de reset

O microcontrolador PIC16F84, admite várias formas de reset:

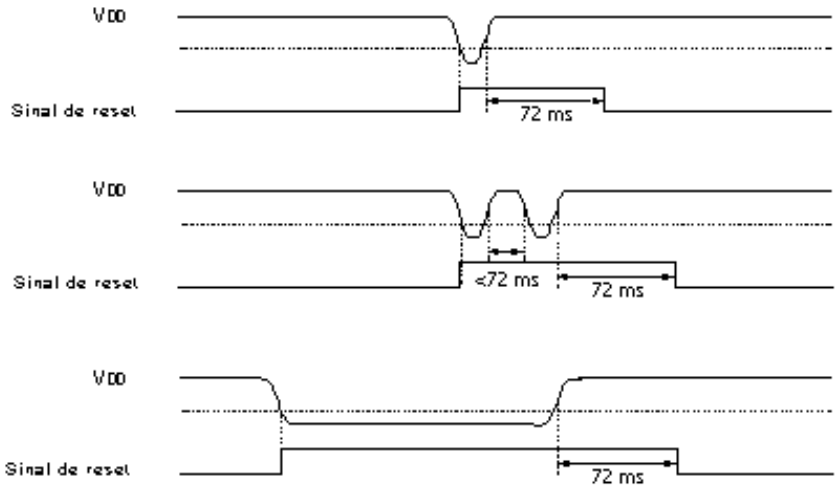
a) Reset quando se liga a alimentação, POR (Power-On Reset)
b) Reset durante o funcionamento normal, quando se põe a nível lógico baixo o pino MCLR do microcontrolador.
c) Reset durante o regime de SLEEP (dormir).
d) Reset quando o temporizador do watchdog (WDT) transborda (passa para 0 depois de atingir o valor máximo).
e) Reset quando o temporizador do watchdog (WDT) transborda estando no regime de SLEEP.

Os reset mais importantes são o a) e o b). O primeiro, ocorre sempre que é ligada a alimentação do microcontrolador e serve para trazer todos os registos para um estado inicial. O segundo que resulta da aplicação de um valor lógico baixo ao pino MCLR durante o funcionamento normal do microcontrolador e, é usado muitas vezes, durante o desenvolvimento de um programa.

Durante um reset, os locais de memória da RAM (registos) não são alterados. Ou seja, os conteúdos destes registos, são desconhecidos durante o restabelecimento da alimentação, mas mantêm-se inalterados durante qualquer outro reset. Ao contrário dos registos normais, os SFR (registos com funções especiais) são reiniciados com um valor inicial pré-definido. Um dos mais importantes efeitos de um reset, é introduzir no contador de programa (PC), o valor zero (0000), o que faz com que o programa comece a ser executado a partir da primeira instrução deste.

**Reset quando o valor da alimentação desce abaixo do limite permitido (Brown-out Reset).**

O impulso que provoca o reset durante o estabelecimento da alimentação (power-up), é gerado pelo próprio microcontrolador quando detecta um aumento na tensão Vdd (numa faixa entre 1,2V e 1,8V). Esse impulso perdura durante 72ms, o que, em princípio, é tempo suficiente para que o oscilador estabilize. Esse intervalo de tempo de 72ms é definido por um temporizador interno PWRT, com um oscilador RC próprio. Enquanto PWRT estiver activo, o microcontrolador mantém-se no estado de reset. Contudo, quando o dispositivo está a trabalhar, pode surgir um problema não resultante de uma queda da tensão para 0 volts, mas sim de uma queda de tensão para um valor abaixo do limite que garante o correcto funcionamento do microcontrolador. Trata-se de um facto muito provável de ocorrer na prática, especialmente em ambientes industriais onde as perturbações e instabilidade da alimentação ocorrem frequentemente. Para resolver este problema, nós precisamos de estar certos de que o microcontrolador entra no estado de reset de cada vez que a alimentação desce abaixo do limite aprovado.

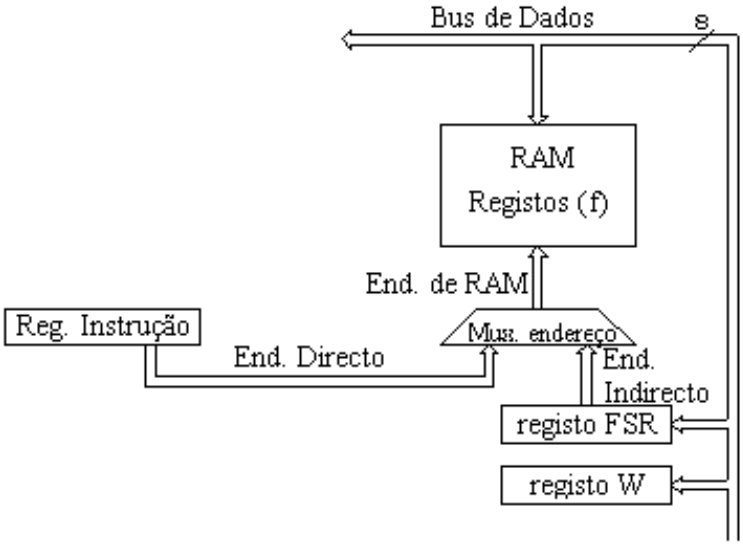



Exemplos de quedas na alimentação abaixo do limite

Se, de acordo com as especificações eléctricas, o circuito interno de reset de um microcontrolador não satisfizer as necessidades, então, deverão ser usados componentes electrónicos especiais, capazes de gerarem o sinal de reset desejado. Além desta função, estes componentes, podem também cumprir o papel de vigiarem as quedas de tensão para um valor abaixo de um nível especificado. Quando isto ocorre, aparece um zero lógico no pino MCLR, que mantém o microcontrolador no estado de reset, enquanto a voltagem não estiver dentro dos limites que garantem um correcto funcionamento.

## 2.3 Unidade Central de Processamento

A unidade central de processamento (CPU) é o cérebro de um microcontrolador. Essa parte é responsável por extrair a instrução, descodificar essa instrução e, finalmente, executá-la.



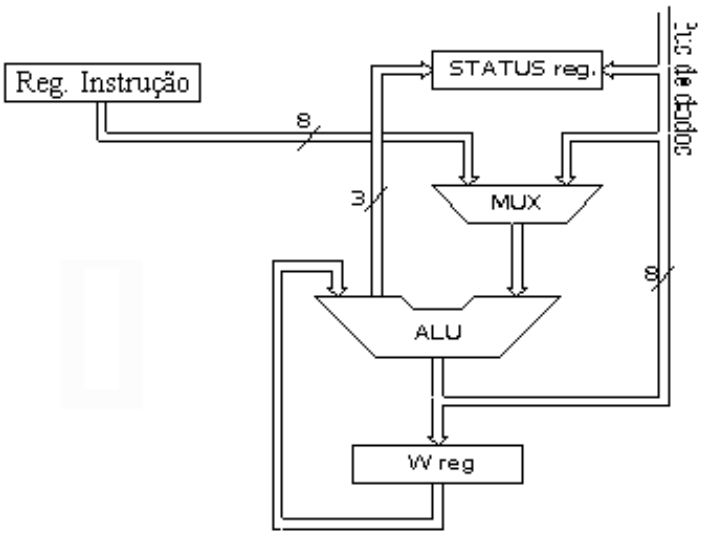

Esquema da unidade central de processamento - CPU

A unidade central de processamento, interliga todas as partes do microcontrolador de modo a que este se comporte como um todo. Uma das sua funções mais importante é, seguramente, descodificar as instruções do programa. Quando o programador escreve um programa, as instruções assumem um claro significado como é o caso por exemplo de MOVLW 0x20. Contudo, para que um microcontrolador possa entendê-las, esta forma escrita de uma instrução tem que ser traduzida numa série de zeros e uns que é o 'opcode' (operation code ou código da operação). Esta passagem de uma palavra escrita para a forma binária é executada por tradutores assembler (ou simplesmente assembler). O código da instrução extraído da memória de programa, tem que ser descodificado pela unidade central de processamento (CPU). A cada uma das instruções do reportório do microcontrolador,

corresponde um conjunto de acções para a concretizar. Estas acções, podem envolver transferências de dados de um local de memória para outro, de um local de memória para os portos, e diversos cálculos, pelo que, se conclui que, o CPU, tem que estar ligado a todas as partes do microcontrolador. Os bus de de dados e o de endereço permitem-nos fazer isso.

**Unidade Lógica Aritmética (ALU)**

A unidade lógica aritmética (ALU – Arithmetic Logic Unit), é responsável pela execução de operações de adição, subtracção, deslocamento (para a esquerda ou para a direita dentro de um registo) e operações lógicas. O PIC16F84 contém uma unidade lógica aritmética de 8 bits e registos de uso genérico também de 8 bits.




Unidade lógica-aritmética e como funciona

Por operando nós designamos o conteúdo sobre o qual uma operação incide. Nas instruções com dois operandos, geralmente um operando está contido no registo de trabalho W (working register) e o outro operando ou é uma constante ou então está contido num dos outros registos. Esses registos podem ser “Registos de Uso Genérico” (General Purpose Registers – GPR) ou “Registos com funções especiais” (Special Function Registers – SFR). Nas instruções só com um operando, um dos operandos é o conteúdo do registo W ou o conteúdo de um dos outros registos. Quando são executadas operações lógicas ou aritméticas como é o caso da adição, a ALU controla o estado dos bits (que constam do registo de estado – STATUS). Dependendo da instrução a ser executada, a ALU, pode modificar os valores bits do Carry (C), Carry de dígito (DC) e Z (zero) no registo de estado - STATUS.

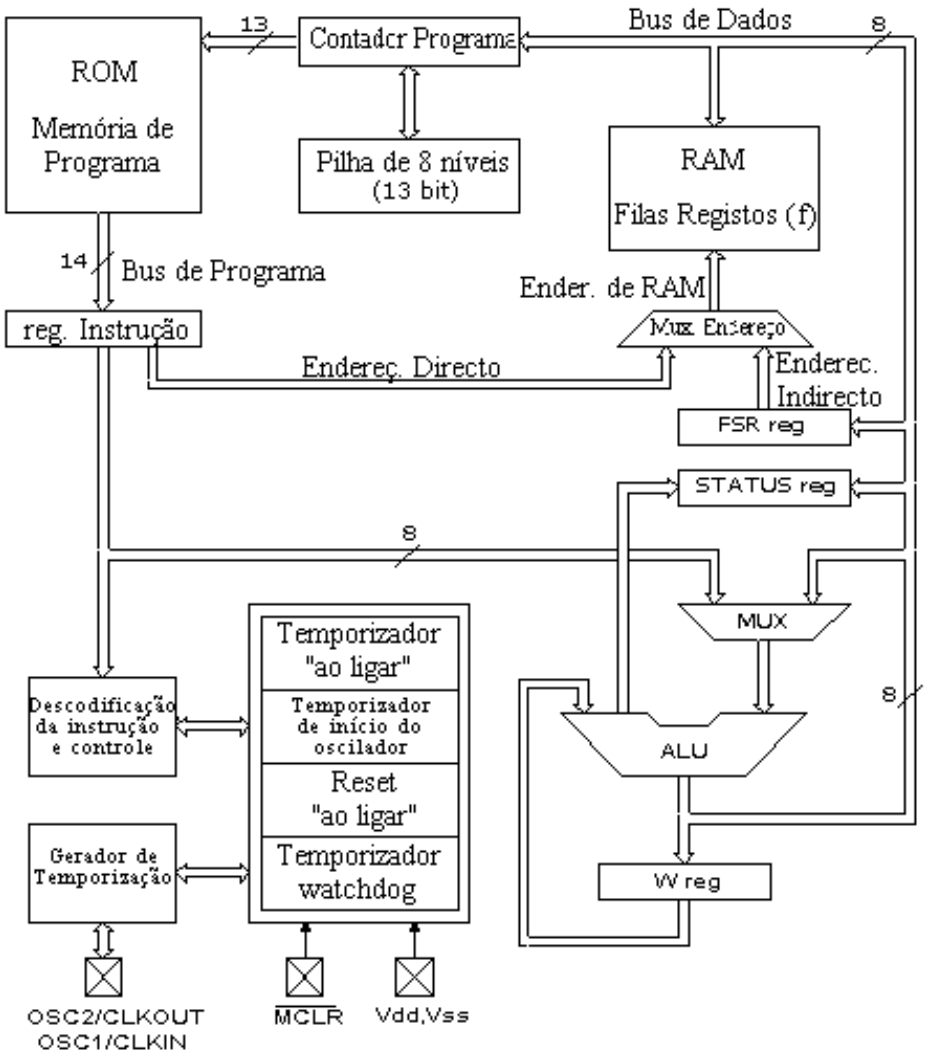



Diagrama bloco mais detalhado do microcontrolador PIC16F84

## Registo STATUS

| R/W-0 | R/W-0 | R/W-0 | R-1 | R-1 | R/W-x | R/W-x | R/W-x |
|---|---|---|---|---|---|---|---|
| IRP | RP1 | RP0 | $\overline{TO}$ | $\overline{PD}$ | Z | DC | C |
| bit 7 | | | | | | | bit 0 |

Legenda:
**R** = bit p/ ler **W** = bit p/escrever
U =bit por implementar, ler como '0' -n =valor p/ reset 'ao ligar'

**bit 0 C (Carry)** Transporte
Este bit é afectado pelas operações de adição, subtracção e deslocamento. Toma o valor ‘1’ (set), quando um valor mais pequeno é subtraído de um valor maior e toma o valor ‘0’ (reset) quando um valor maior é subtraído de um menor.
1= Ocorreu um transporte no bit mais significativo
0= Não ocorreu transporte no bit mais significativo
O bit C é afectado pelas instruções ADDWF, ADDLW, SUBLW e SUBWF.

**bit 1 DC** (Digit Carry) Transporte de dígito
Este bit é afectado pelas operações de adição, subtracção. Ao contrário do anterior, DC assinala um transporte do bit 3 para o bit 4 do resultado. Este bit toma o valor ‘1’, quando um valor mais pequeno é subtraído de um valor

maior e toma o valor ‘0’ quando um valor maior é subtraído de um menor.
1= Ocorreu um transporte no quarto bit mais significativo
0= Não ocorreu transporte nesse bit
O bit DC é afectado pelas instruções ADDWF, ADDLW, SUBLW e SUBWF.

**bit 2 Z** (bit Zero) Indicação de resultado igual a zero.
Este bit toma o valor ‘1’ quando o resultado da operação lógica ou aritmética executada é igual a 0.
1= resultado igual a zero
0= resultado diferente de zero

**bit 3 PD** (Bit de baixa de tensão – Power Down)
Este bit é posto a ‘1’ quando o microcontrolador é alimentado e começa a trabalhar, depois de um reset normal e depois da execução da instrução CLRWDT. A instrução SLEEP põe este bit a ‘0’ ou seja, quando o microcontrolador entra no regime de baixo consumo / pouco trabalho. Este bit pode também ser posto a ‘1’, no caso de ocorrer um impulso no pino RB0/INT, uma variação nos quatro bits mais significativos do porto B, ou quando é completada uma operação de escrita na DATA EEPROM ou ainda pelo watchdog.
1 = depois de ter sido ligada a alimentação
0 = depois da execução de uma instrução SLEEP

**bit 4 TO** Time-out ; transbordo do Watchdog
Este bit é posto a ‘1’, depois de a alimentação ser ligada e depois da execução das instruções CLRWDT e SLEEP. O bit é posto a ‘0’ quando o watchdog consegue chegar ao fim da sua contagem (overflow = transbordar), o que indica que qualquer coisa não esteve bem.
1 = não ocorreu transbordo
0 = ocorreu transbordo

**bits 5 e 6 RP1:RP0** (bits de selecção de banco de registos)
Estes dois bits são a parte mais significativa do endereço utilizado para endereçamento directo. Como as instruções que endereçam directamente a memória, dispõem somente de sete bits para este efeito, é preciso mais um bit para poder endereçar todos os 256 registos do PIC16F84. No caso do PIC16F84, RP1, não é usado, mas pode ser necessário no caso de outros microcontroladores PIC, de maior capacidade.
01 = banco de registos 1
00 = banco de registos 0

**bit 7 IRP** (Bit de selecção de banco de registos)
Este bit é utilizado no endereçamento indirecto da RAM interna, como oitavo bit
1 = bancos 2 e 3
0 = bancos 0 e 1 (endereços de 00h a FFh)

O registo de estado (STATUS), contém o estado da ALU (C, DC, Z), estado de RESET (TO, PD) e os bits para selecção do banco de memória (IRP, RP1, RP0). Considerando que a selecção do banco de memória é controlada através deste registo, ele tem que estar presente em todos os bancos. Os bancos de memória serão discutidos com mais detalhe no capítulo que trata da Organização da Memória. Se o registo STATUS for o registo de destino para instruções que afectem os bits Z, DC ou C, então não é possível escrever nestes três bits.

## Registo OPTION

| R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 |
|---|---|---|---|---|---|---|---|
| $\overline{\text{RBPU}}$ (1) | INTEDG | T0CS | T0SE | PSA | PS2 | PS1 | PS0 |
| bit 7 | | | | | | | bit 0 |

| Legenda: | |
|---|---|
| **R** = bit p/ ler | **W** = bit p/ escrever |
| U =não implementado, ler como '0' | -n = valor p/ reset 'ao ligar' |

**bits 0 a 2 PS0, PS1, PS2** (bits de selecção do divisor Prescaler)
Estes três bits definem o factor de divisão do prescaler. Aquilo que é o prescaler e o modo como o valor destes três bits afectam o funcionamento do microcontrolador será estudado na secção referente a TMR0.

| Bits | 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 |

b**it 3 PSA** (Bit de Atribuição do Prescaler)
Bit que atribui o prescaler ao TMR0 ou ao watchdog.
1 = prescaler atribuído ao watchdog
0 = prescaler atribuído ao temporizador TMR0

**bit 4 T0SE** (bit de selecção de bordo activo em TMR0)
Se for permitido aplicar impulsos em TMR0, a partir do pino RA4/TOCK1, este bit determina se os impulsos activos são os impulsos ascendentes ou os impulsos descendentes.
1 = bordo descendente
0 = bordo ascendente

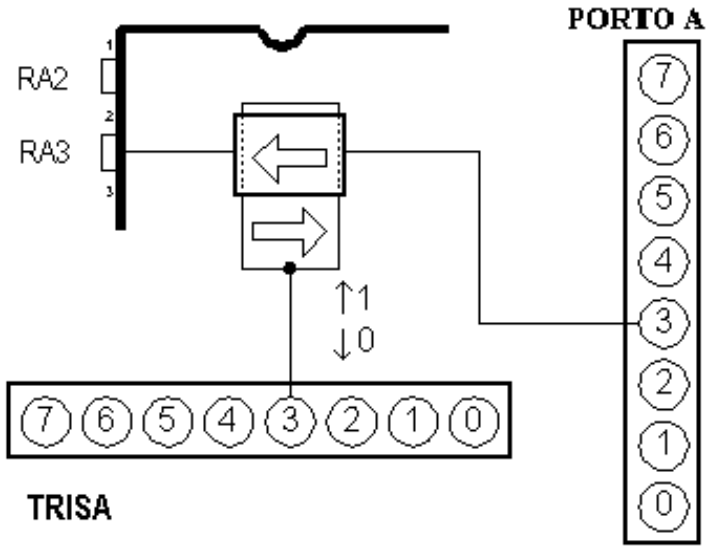
**bit 5 TOCS** (bit de selecção de fonte de clock em TMR0)
Este pino escolhe a fonte de impulsos que vai ligar ao temporizador. Esta fonte pode ser o clock do microcontrolador (frequência de clock a dividir por 4) ou impulsos externos no pino RA4/TOCKI.
1 = impulsos externos
0 = ¼ do clock interno

**bit 6 INDEDG** (bit de selecção de bordo de interrupção)
Se esta interrupção estiver habilitada, é possível definir o bordo que vai activar a interrupção no pino RB0/INT.
1 = bordo ascendente
0 = bordo descendente

**bit 7 RBPU** (Habilitação dos pull-up nos bits do porto B)
Este bit introduz ou retira as resistências internas de pull-up do porto B.
1 = resistências de “pull-up” desligadas
0 = resistências de “pull-up” ligadas

## 2.4 Portos

Porto, é um grupo de pinos num microcontrolador que podem ser acedidos simultaneamente, e, no qual nós podemos colocar uma combinação de zeros e uns ou ler dele o estado existente. Fisicamente, porto é um registo dentro de um microcontrolador que está ligado por fios aos pinos do microcontrolador. Os portos representam a conexão física da Unidade Central de Processamento (CPU) com o mundo exterior. O microcontrolador usa-os para observar ou comandar outros componentes ou dispositivos. Para aumentar a sua funcionalidade, os mesmos pinos podem ter duas aplicações distintas, como, por exemplo, RA4/TOCKI, que é simultaneamente o bit 4 do porto A e uma entrada externa para o contador/temporizador TMR0. A escolha de uma destas duas funções é feita através dos registos de configuração. Um exemplo disto é o TOCS, quinto bit do registo OPTION. Ao seleccionar uma das funções, a outra é automaticamente inibida.




Relação entre os registos TRISA e PORTO A

Todos os pinos dos portos podem ser definidos como de entrada ou de saída, de acordo com as necessidades do dispositivo que se está a projectar. Para definir um pino como entrada ou como saída, é preciso, em primeiro lugar, escrever no registo TRIS, a combinação apropriada de zeros e uns. Se no local apropriado de um registo TRIS for escrito o valor lógico “1”, então o correspondente pino do porto é definido como entrada, se suceder o contrário, o pino é definido como saída. Todos os portos, têm um registo TRIS associado. Assim, para o porto A, existe o registo TRISA no endereço 85h e, para o porto B existe o registo TRISB, no endereço 86h.

## PORTO B

O porto B tem 8 pinos associados a ele. O respectivo registo de direcção de dados chama-se TRISB e tem o endereço 86h. Ao pôr a ‘1’ um bit do registo TRISB, define-se o correspondente pino do porto como entrada e se pusermos a ‘0’ um bit do registo TRISB, o pino correspondente vai ser uma saída. Cada pino do PORTO B possui uma pequena resistência de ‘pull-up’ (resistência que define a linha como tendo o valor lógico ‘1’). As resistências de pull-up são activadas pondo a ‘0’ o bit RBPU, que é o bit 7 do registo OPTION. Estas resistências de ‘pull-up’ são automaticamente desligadas quando os pinos do porto são configurados como saídas. Quando a alimentação do microcontrolador é ligada, as resistências de pull-up são também desactivadas.

Quatro pinos do PORTO B, RB4 a RB7 podem causar uma interrupção, que ocorre quando qualquer deles varia do valor lógico zero para valor lógico um ou o contrário. Esta forma de interrupção só pode ocorrer se estes pinos forem configurados como entradas (se qualquer um destes 4 pinos for configurado como saída, não será gerada uma interrupção quando há variação de estado). Esta modalidade de interrupção, acompanhada da existência de resistências de pull-up internas, torna possível resolver mais facilmente problemas frequentes que podemos encontrar na prática, como por exemplo a ligação de um teclado matricial. Se as linhas de um teclado ficarem ligadas a estes pinos, sempre que se prime uma tecla, ir-se-á provocar uma interrupção. Ao processar a interrupção, o microcontrolador terá que identificar a tecla que a produziu. Não é recomendável utilizar o porto B, ao mesmo tempo que esta interrupção está a ser processada.

```
clrf   STATUS       ;Banco 0
clrf   PORTB        ;Porto B = 0
bsf    STATUS,RP0   ;Banco 1
movlw  0x0F         ;Definir pinos de entrada e saída
movwf  TRISB        ;Escrever no registo TRISB
```

O exemplo de cima mostra como os pinos 0, 1, 2 e 3 são definidos como entradas e 4, 5, 6 e 7 como saídas.

## PORTO A

O porto A (PORTA) está associado a 5 pinos. O registo de direcção de dados correspondente é o TRISA, no

endereço 85h. Tal como no caso do porto B, pôr a ‘1’ um bit do registo TRISA, equivale a definir o correspondente pino do porto A, como entrada e pôr a ‘0’ um bit do mesmo registo, equivale a definir o correspondente pino do porto A, como saída.
O quinto pino do porto A tem uma função dupla. Nesse pino está também situada a entrada externa do temporizador TMR0. Cada uma destas opções é escolhida pondo a ‘1’ ou pondo a ‘0’ o bit TOCS (bit de selecção de fonte de clock de TMR0). Conforme o valor deste bit, assim o temporizador TMR0 incrementa o seu valor por causa de um impulso do oscilador interno ou devido a um impulso externo aplicado ao pino RA4/TOCKI.

```
bcf    STATUS,RPO   ;Banco 0
clrf   PORTA        ;Porto A = 0
bsf    STATUS,RPO   ;Banco 1
movlw  0x1F         ;Definir pinos de entrada e  de saída
movwf  TRISA        ;Escrever no registo TRISA
```

Este exemplo mostra como os pinos 0, 1, 2, 3 e 4 são declarados como entradas e os pinos 5, 6 e 7 como pinos de saída.

## 2.5 Organização da memória

O PIC16F84 tem dois blocos de memória separados, um para dados e o outro para o programa. A memória EEPROM e os registos de uso genérico (GPR) na memória RAM constituem o bloco para dados e a memória FLASH constitui o bloco de programa.

**Memória de programa**

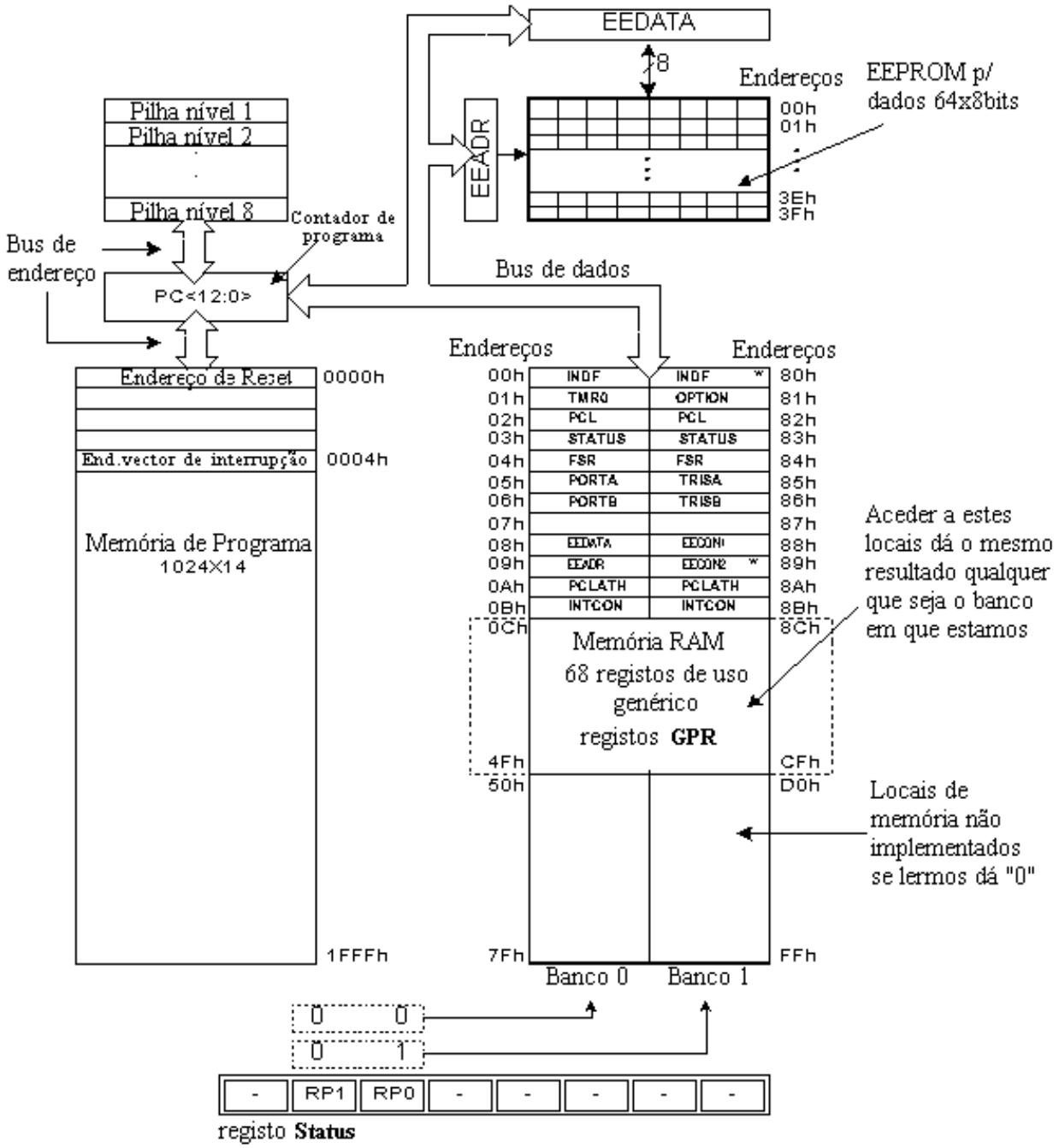
A memória de programa é implementada usando tecnologia FLASH, o que torna possível programar o microcontrolador muitas vezes antes de este ser instalado num dispositivo, e, mesmo depois da sua instalação, podemos alterar o programa e parâmetros contidos. O tamanho da memória de programa é de 1024 endereços de palavras de 14 bits, destes, os endereços zero e quatro estão reservados respectivamente para o reset e para o vector de interrupção.

**Memória de dados**

A memória de dados compreende memória EEPROM e memória RAM. A memória EEPROM consiste em 64 posições para palavras de oito bits e cujos conteúdos não se perdem durante uma falha na alimentação. A memória EEPROM não faz parte directamente do espaço de memória mas é acedida indirectamente através dos registos EEADR e EEDATA. Como a memória EEPROM serve usualmente para guardar parâmetros importantes (por exemplo, de uma dada temperatura em reguladores de temperatura), existe um procedimento estrito para escrever na EEPROM que tem que ser seguido de modo a evitar uma escrita acidental. A memória RAM para dados, ocupa um espaço no mapa de memória desde o endereço 0x0C até 0x4F, o que corresponde a 68 localizações. Os locais da memória RAM são também chamados registos GPR (General Purpose Registers = Registos de uso genérico). Os registos GPR podem ser acedidos sem ter em atenção o banco em que nos encontramos de momento.

**Registos SFR**

Os registos que ocupam as 12 primeiras localizações nos bancos 0 e 1 são registos especiais e têm a ver com a manipulação de certos blocos do microcontrolador. Estes registos são os SFR (Special Function Registers ou Registos de Funções Especiais).




Organização da memória no microcontrolador PIC16F84

**Bancos de Memória**

Além da divisão em 'comprimento' entre registos SFR e GPR, o mapa de memória está também dividido em 'largura' (ver mapa anterior) em duas áreas chamadas 'bancos'. A selecção de um dos bancos é feita por intermédio dos bits RP0 e RP1 do registo STATUS.

Exemplo :

```
bcf STATUS, RP0
```

A instrução BCF "limpa" o bit RP0 (RP0 = 0) do registo STATUS e, assim, coloca-nos no banco 0.

```
bsf STATUS, RP0
```

A instrução BSF põe a um, o bit RP0 (RP0 = 1) do registo STATUS e, assim, coloca-nos no banco 1.

Normalmente, os grupos de instruções muito usados são ligados numa única unidade que pode ser facilmente

invocada por diversas vezes num programa, uma unidade desse tipo chama-se genericamente Macro e, normalmente, essa unidade é designada por um nome especifico facilmente compreensível. Com a sua utilização, a selecção entre os dois bancos torna-se mais clara e o próprio programa fica mais legível.

```
BANK0 macro
Bcf STATUS, RP0 ;Selecionar o banco 0 da memória
Endm

BANK1 macro
Bsf STATUS, RP0 ; Selecionar o banco 1 da memória
Endm
```

Os locais de memória 0Ch – 4Fh são registos de uso genérico (GPR) e são usados como memória RAM. Quando os endereços 8Ch – CFh são acedidos, nós acedemos também às mesmas localizações do banco 0. Por outras palavras, quando estamos a trabalhar com os registos de uso genérico, não precisamos de nos preocupar com o banco em que nos encontramos!

**Contador de Programa**

O contador de programa (PC = Program Counter), é um registo de 13 bits que contém o endereço da instrução que vai ser executada. Ao incrementar ou alterar (por exemplo no caso de saltos) o conteúdo do PC, o microcontrolador consegue executar as todas as instruções do programa, uma após outra.

**Pilha**

O PIC16F84 tem uma pilha (stack) de 13 bits e 8 níveis de profundidade, o que corresponde a 8 locais de memória com 13 bits de largura. O seu papel básico é guardar o valor do contador de programa quando ocorre um salto do programa principal para o endereço de um subprograma a ser executado. Depois de ter executado o subprograma, para que o microcontrolador possa continuar com o programa principal a partir do ponto em que o deixou, ele tem que ir buscar à pilha esse endereço e carregá-lo no contador de programa. Quando nos movemos de um programa para um subprograma, o conteúdo do contador de programa é empurrado para o interior da pilha (um exemplo disto é a instrução CALL). Quando são executadas instruções tais como RETURN, RETLW ou RETFIE no fim de um subprograma, o contador de programa é retirado da pilha, de modo a que o programa possa continuar a partir do ponto em que a sequência foi interrompida. Estas operações de colocar e extrair da pilha o contador de programa, são designadas por PUSH (meter na pilha) e POP (tirar da pilha), estes dois nomes provêm de instruções com estas designações, existentes nalguns microcontroladores de maior porte.

**Programação no Sistema**

Para programar a memória de programa, o microcontrolador tem que entrar num modo especial de funcionamento no qual o pino MCLR é posto a 13,5V e a voltagem da alimentação Vdd deve permanecer estável entre 4,5V e 5,5V. A memória de programa pode ser programada em série, usando dois pinos ‘data/clock’ que devem ser previamente separados do dispositivo em que o microcontrolador está inserido, de modo a que não possam ocorrer erros durante a programação.

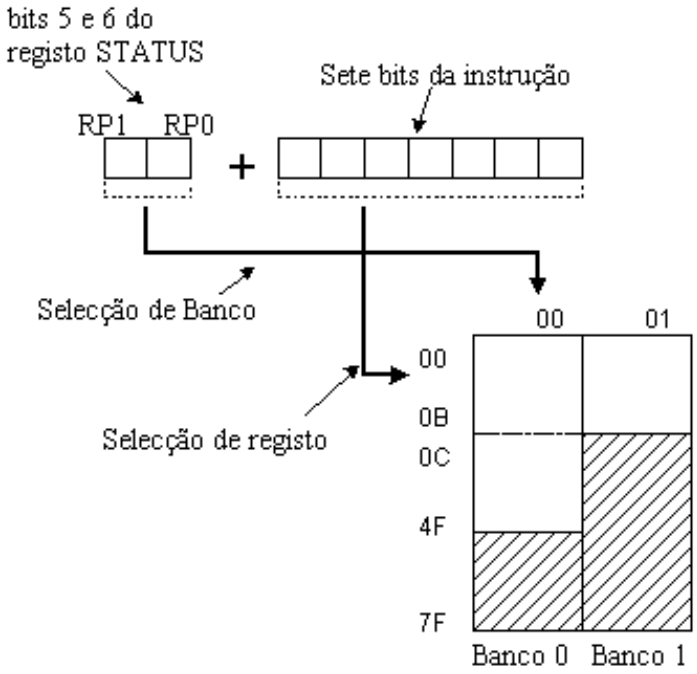
**Modos de endereçamento**

Os locais da memória RAM podem ser acedidos directa ou indirectamente.

**Endereçamento Directo**

O endereçamento directo é feito através de um endereço de 9 bits. Este endereço obtém-se juntando aos sete bits do endereço directo de uma instrução, mais dois bits (RP1 e RP0) do registo STATUS, como se mostra na figura que se segue. Qualquer acesso aos registos especiais (SFR), pode ser um exemplo de endereçamento directo.

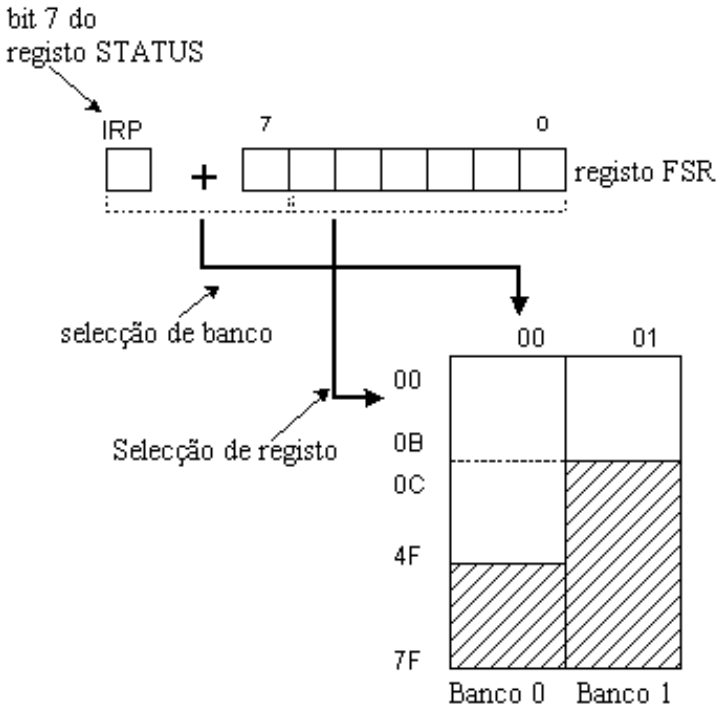
```
Bsf STATUS, RP0   ; Banco 1
movlw 0xFF        ; w = 0xFF
movwf TRISA       ; o endereço do registo TRISA é tirado do código da instrução movwf TRISA
```




Endereçamento Directo

**Endereçamento Indirecto**

O endereçamento indirecto, ao contrário do directo, não tira um endereço do código instrução, mas fá-lo com a ajuda do bit IRP do registo STATUS e do registo FSR. O local endereçado é acedido através do registo INDF e coincide com o endereço contido em FSR. Por outras palavras, qualquer instrução que use INDF como registo, na realidade acede aos dados apontados pelo registo FSR. Vamos supor, por exemplo, que o registo de uso genérico de endereço 0Fh contém o valor 20. Escrevendo o valor de 0Fh no registo FSR, nós vamos obter um ponteiro para o registo 0Fh e, ao ler o registo INDF, nós iremos obter o valor 20, o que significa que lemos o conteúdo do registo 0Fh, sem o mencionar explicitamente (mas através de FSR e INDF). Pode parecer que este tipo de endereçamento não tem quaisquer vantagens sobre o endereçamento directo, mas existem problemas que só podem ser resolvidos de uma forma simples, através do endereçamento indirecto.



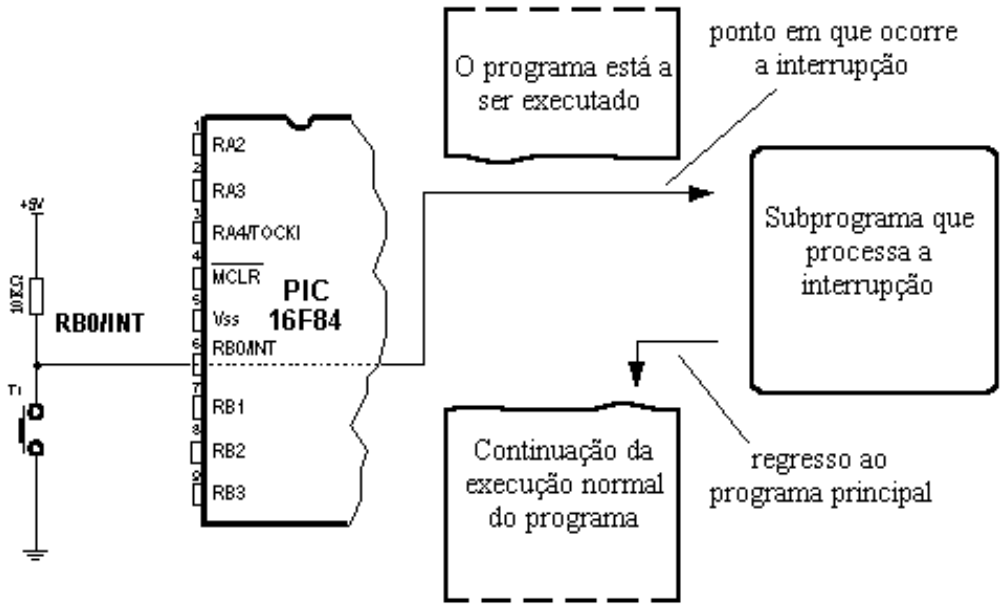
Um exemplo pode ser enviar um conjunto de dados através de uma comunicação série, usando buffers e indicadores (que serão discutidos num capítulo mais à frente, com exemplos), outro exemplo é limpar os registos da memória RAM (16 endereços neste caso) como se pode ver a seguir.

```
        Movlw  0x0C          ;definição do endereço de início
        Movwf  FSR           ;FSR aponta p/ o endereço 0x0C
LOOP    clrf   INDF          ;INDF = 0
        incf   FSR           ;endereço = endereço inicial + 1
        btfss  FSR,4         ;todos os locais de memória limpos?
        goto   loop          ;não, para ' loop' de novo
CONTINUE
        :                    ;sim, continuar com o programa
```

Quando o conteúdo do registo FSR é igual a zero, ler dados do registo INDF resulta no valor 0 e escrever em INDF resulta na instrução NOP (no operation = nenhuma operação).

## 2.6 Interrupções

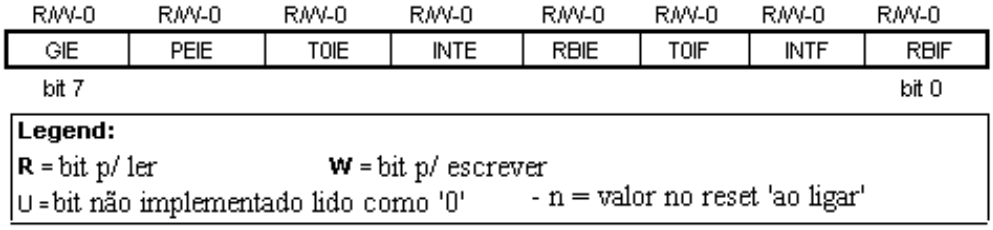
As interrupções são um mecanismo que o microcontrolador possui e que torna possível responder a alguns acontecimentos no momento em que eles ocorrem, qualquer que seja a tarefa que o microcontrolador esteja a executar no momento. Esta é uma parte muito importante, porque fornece a ligação entre um microcontrolador e o mundo real que nos rodeia. Geralmente, cada interrupção muda a direcção de execução do programa, suspendendo a sua execução, enquanto o microcontrolador corre um subprograma que é a rotina de atendimento de interrupção. Depois de este subprograma ter sido executado, o microcontrolador continua com o programa principal, a partir do local em que o tinha abandonado.




Uma das possíveis fontes de interrupção e como afecta o programa principal

O registo que controla as interrupções é chamado INTCON e tem o endereço 0Bh. O papel do INTCON é permitir ou impedir as interrupções e, mesmo no caso de elas não serem permitidas, ele toma nota de pedidos específicos, alterando o nível lógico de alguns dos seus bits.

### Registo INTCON




**bit 0 RBIF** (flag que indica variação no porto B) Bit que informa que houve mudança nos níveis lógicos nos pinos 4, 5, 6 e 7 do porto B.
1= pelo menos um destes pinos mudou de nível lógico
0= não ocorreu nenhuma variação nestes pinos

**bit 1 INTF** (flag de interrupção externa INT) Ocorrência de uma interrupção externa
1= ocorreu uma interrupção externa
0= não ocorreu uma interrupção externa
Se um impulso ascendente ou descendente for detectado no pino RB0/INT, o bit INTF é posto a '1' (o tipo de sensibilidade, ascendente ou descendente é definida através do bit INTEDG do registo OPTION). O subprograma de atendimento desta interrupção, deve repor este bit a '0', afim de que a próxima interrupção possa ser detectada.

**bit 2 TOIF** (Flag de interrupção por transbordo de TMR0) O contador TMR0, transbordou.
1= o contador mudou a contagem de FFh para 00h
0= o contador não transbordou
Para que esta interrupção seja detectada, o programa deve pôr este bit a '0'

**bit 3 RBIE** (bit de habilitação de interrupção por variação no porto B) Permite que a interrupção por variação dos níveis lógicos nos pinos 4, 5, 6 e 7 do porto B, ocorra.
1= habilita a interrupção por variação dos níveis lógicos
0= inibe a interrupção por variação dos níveis lógicos
A interrupção só pode ocorrer se RBIE e RBIF estiverem simultaneamente a '1' lógico.

**bit 4 INTE** (bit de habilitação da interrupção externa INT) bit que permite uma interrupção externa no bit RB0/INT.
1= interrupção externa habilitada
0= interrupção externa impedida
A interrupção só pode ocorrer se INTE e INTF estiverem simultaneamente a '1' lógico.

**bit 5 TOIE** (bit de habilitação de interrupção por transbordo de TMR0) bit que autoriza a interrupção por transbordo do contador TMR0.
1= interrupção autorizada
0= interrupção impedida
A interrupção só pode ocorrer se TOIE e TOIF estiverem simultaneamente a '1' lógico.

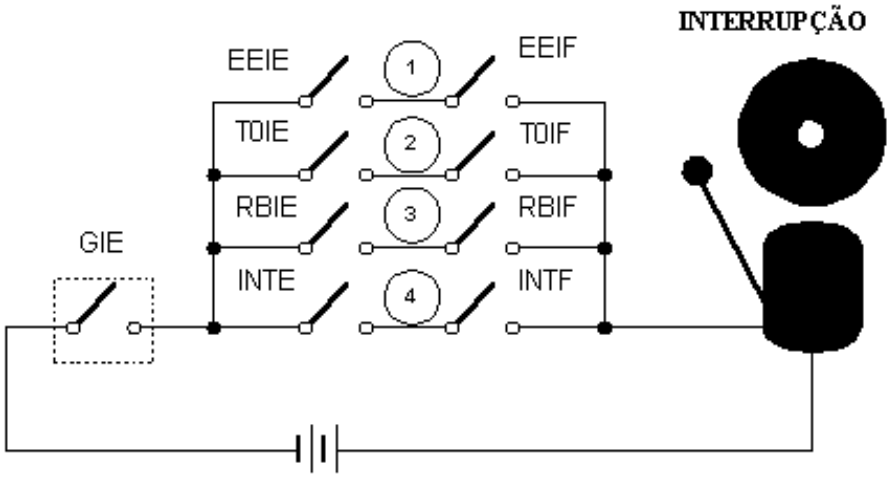
**bit 6 EEIE** (bit de habilitação de interrupção por escrita completa, na EEPROM) bit que habilita uma interrupção quando uma operação de escrita na EEPROM termina.
1= interrupção habilitada
0= interrupção inibida
Se EEIE e EEIF (que pertence ao registo EECON1) estiverem simultaneamente a '1', a interrupção pode ocorrer.

**bit 7 GIE** (bit de habilitação global de interrupção) bit que permite ou impede todas as interrupções
1= todas as interrupções são permitidas
0= todas as interrupções impedidas

O PIC16F84 possui quatro fontes de interrupção:

1. Fim de escrita na EEPROM
2. Interrupção em TMR0 causada por transbordo do temporizador
3. Interrupção por alteração nos pinos RB4, RB5, RB6 e RB7 do porto B.
4. Interrupção externa no pino RB0/INT do microcontrolador

De um modo geral, cada fonte de interrupção tem dois bits associados. Um habilita a interrupção e o outro assinala quando a interrupção ocorre. Existe um bit comum a todas as interrupções chamado GIE que pode ser usado para impedir ou habilitar todas as interrupções, simultaneamente. Este bit é muito útil quando se está a escrever um programa porque permite que todas as interrupções sejam impedidas durante um período de tempo, de tal maneira que a execução de uma parte crítica do programa não possa ser interrompida. Quando a instrução que faz GIE= 0 é executada (GIE= 0 impede todas as interrupções), todas os pedidos de interrupção pendentes, serão ignorados.



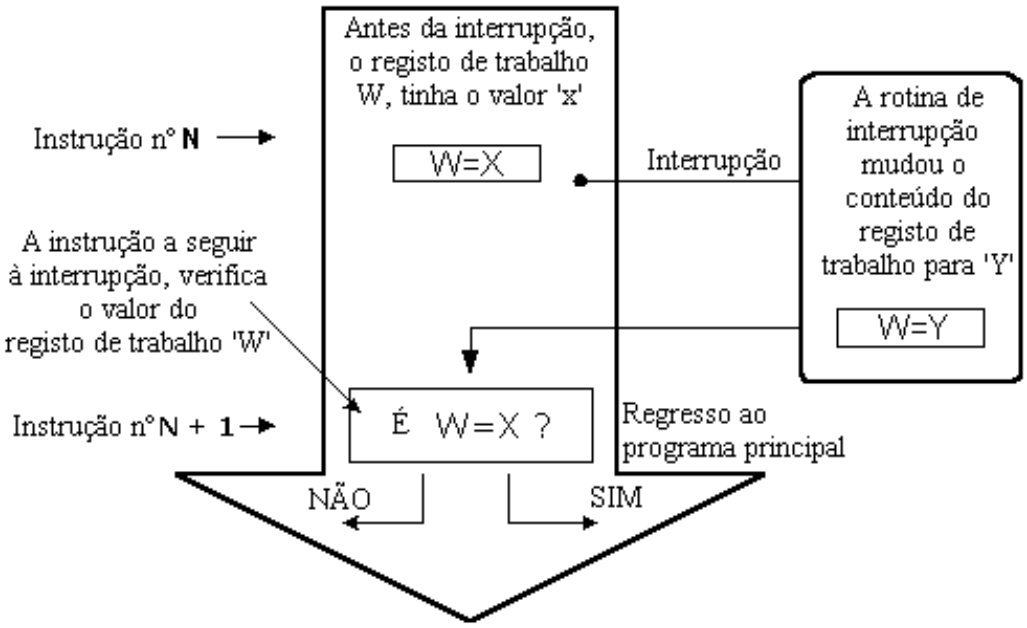

Esquema das interrupções no microcontrolador PIC16F84

As interrupções que estão pendentes e que são ignoradas, são processadas quando o bit GIE é posto a ‘1’ (GIE= 1, todas as interrupções permitidas). Quando a interrupção é atendida, o bit GIE é posto a ‘0’, de tal modo que, quaisquer interrupções adicionais sejam inibidas, o endereço de retorno é guardado na pilha e, no contador de programa, é escrito 0004h – somente depois disto, é que a resposta a uma interrupção começa!
Depois de a interrupção ser processada, o bit que por ter sido posto a ‘1’ permitiu a interrupção, deve agora ser reposto a ‘0’, senão, a rotina de interrupção irá ser automaticamente processada novamente, mal se efectue o regresso ao programa principal.

**Guardando os conteúdos dos registos importantes**

A única coisa que é guardada na pilha durante uma interrupção é o valor de retorno do contador de programa (por valor de retorno do contador de programa entende-se o endereço da instrução que estava para ser executada, mas que não foi, por causa de ter ocorrido a interrupção). Guardar apenas o valor do contador de programa não é, muitas vezes, suficiente. Alguns registos que já foram usados no programa principal, podem também vir a ser usados na rotina de interrupção. Se nós não salvaguardamos os seus valores, quando acontece o regresso da subrotina para o programa principal os conteúdos dos registos podem ser inteiramente diferentes, o que causaria um erro no programa. Um exemplo para este caso é o conteúdo do registo de trabalho W (work register). Se supormos que o programa principal estava a usar o registo de trabalho W nalgumas das suas operações e se ele contiver algum valor que seja importante para a instrução seguinte, então a interrupção que ocorre antes desta instrução vai alterar o valor do registo de trabalho W, indo influenciar directamente o programa principal.

O procedimento para a gravação de registos importantes antes de ir para a subrotina de interrupção, designa-se por ‘PUSH’, enquanto que o procedimento que recupera esses valores, é chamado POP. PUSH e POP são instruções provenientes de outros microcontroladores (da Intel), agora esses nomes são aceites para designar estes dois processos de salvaguarda e recuperação de dados. Como o PIC16F84 não possui instruções comparáveis, elas têm que ser programadas.




Uma das possíveis causas de erros é não salvaguardar dados antes de executar um subprograma de interrupção

Devido à sua simplicidade e uso frequente, estas partes do programa podem ser implementadas com macros. O conceito de Macro é explicado em “Programação em linguagem Assembly”. No exemplo que se segue, os conteúdos de W e do registo STATUS são guardados nas variáveis W_TEMP e STATUS_TEMP antes de correr a rotina de interrupção. No início da rotina PUSH, nós precisamos de verificar qual o banco que está a ser seleccionado porque W_TEMP e STATUS_TEMP estão situados no banco 0. Para troca de dados entre estes dois registos, é usada a instrução SWAPF em vez de MOVF, pois a primeira não afecta os bits do registo STATUS.

Exemplo é um programa assembler com os seguintes passos:

1. Verificar em que banco nos encontramos
2. Guardar o registo W qualquer que seja o banco em que nos encontramos
3. Guardar o registo STATUS no banco 0.
4. Executar a rotina de serviço de interrupção ISR (Interrupt Service Routine)
5. Recuperação do registo STATUS
6. Restaurar o valor do registo W

Se existirem mais variáveis ou registos que necessitem de ser salvaguardados, então, precisamos de os guardar depois de guardar o registo STATUS (passo 3) e recuperá-los depois de restaurar o registo STATUS (passo 5).

```
Push
        BTFSS STATUS, RP0           ; Banco 0?
        GOTO RP0CLEAR               ; Sim
        BCF STATUS, RP0             ; Não, ir p/ Banco 0
        MOVWF W_TEMP                ; Guardar registo W
        SWAPF STATUS, W             ; W <- STATUS
        MOVWF STATUS_TEMP           ; STATUS_TEMP <- W
        BSF STATUS_TEMP, 1          ; RP0(STATUS_TEMP)=1
        GOTO ISR_Code               ; Push completado
RP0CLEAR
        MOVWF W_TEMP                ; Guardar registo W
        SWAPF STATUS, W             ; W <- STATUS
        MOVWF STATUS_TEMP           ; STATUS_TEMP <- W
;
ISR_Code
        :
        ;Subprograma de Interrupção
        :
;
Pop
        SWAPF STATUS_TEMP, W        ; W <- STATUS_TEMP
        MOVWF STATUS                ; STATUS <-W
        BTFSS STATUS, RP0           ; Banco 1?
        GOTO Return_WREG            ; Não
        BCF STATUS, RP0             ; Sim, ir p/ o banco 0
        SWAPF W_TEMP, F             ; Recuperar o conteúdo de W
        SWAPF W_TEMP, W             ;
        BSF STATUS, RP0             ; Regressar ao banco 1
        RETFIE                      ; POP completo
Return_WREG
        SWAPF W_TEMP, F             ; Recuperar o conteúdo de W
        SWAPF W_TEMP, W             ;
        RETFIE                      ; POP completo
```

A mesma operação pode ser realizada usando macros, desta maneira obtemos um programa mais legível. Os macros que já estão definidos podem ser usados para escrever novos macros. Os macros BANK1 e BANK0 que são explicados no capítulo “Organização da memória” são usados nos macros ‘push’ e ‘pop’.

```
push    macro
        movwf  W_Temp               ;W_Temp <- W
        swapf  W_Temp,F             ;trocar a ordem dos bits
        BANK1                       ;Macro p/ aceder ao banco 1
        swapf  OPTION_REG,W         ;W <- OPTION_REG
        movwf  Option_Temp          ;Option_Temp <- W
        BANK0                       ;Macro p/ aceder ao banco 0
        swapf  STATUS,W             ;W <- STATUS
        movwf  Stat_Temp            ;Stat_Temp <-W
        endm                        ;Fim do macro push

pop     macro
        swapf  Stat_Temp,W          ;W <- Stat_Temp
        movwf  STATUS               ;STATUS <- W
        BANK1                       ;Macro p/ aceder ao banco 1
        swapf  Option_Temp,W        ;W <- Option_Temp
        movwf  OPTION_REG           ;OPTION_REG <- W
        BANK0                       ;Macro p/ aceder ao banco 0
        swapf  W_Temp,W             ;W <- W_Temp
        endm                        ;Fim do macro pop
```

**Interrupção externa no pino RB0 / INT do microcontrolador**

A interrupção externa no pino RB0/ INT é desencadeada por um impulso ascendente (se o bit INTEDG = 1 no registo OPTION<6>), ou por um impulso descendente (se INTEDG = 0). Quando o sinal correcto surge no pino

INT, o bit INTF do registo INTCON é posto a ‘1’. O bit INTF (INTCON<1>) tem que ser reposto a ‘0’ na rotina de interrupção, afim de que a interrupção não possa voltar a ocorrer de novo, aquando do regresso ao programa principal. Esta é uma parte importante do programa e que o programador não pode esquecer, caso contrário o programa irá constantemente saltar para a rotina de interrupção. A interrupção pode ser inibida, pondo a ‘0’ o bit de controle INTE (INTCON<4>).

**Interrupção devido ao transbordar (overflow) do contador TMR0**

O transbordar do contador TMR0 (passagem de FFh para 00h) vai pôr a ‘1’ o bit TOIF (INTCON<2>), Esta é uma interrupção muito importante, uma vez que, muitos problemas da vida real podem ser resolvidos utilizando esta interrupção. Um exemplo é o da medição de tempo. Se soubermos de quanto tempo o contador precisa para completar um ciclo de 00h a FFh, então, o número de interrupções multiplicado por esse intervalo de tempo, dá-nos o tempo total decorrido. Na rotina de interrupção uma variável guardada na memória RAM vai sendo incrementada, o valor dessa variável multiplicado pelo tempo que o contador precisa para um ciclo completo de contagem, vai dar o tempo gasto. Esta interrupção pode ser habilitada ou inibida, pondo a ‘1’ ou a ‘0’ o bit TOIE (INTCON<5>).

**Interrupção por variação nos pinos 4, 5, 6 e 7 do porto B**

Uma variação em 4 bits de entrada do Porto B (bits 4 a 7), põe a ‘1’ o bit RBIF (INTCON<0>). A interrupção ocorre, portanto, quando os níveis lógicos em RB7, RB6, RB5 e RB4 do porto B, mudam do valor lógico ‘1’ para o valor lógico ‘0’ ou vice-versa. Para que estes pinos detectem as variações, eles devem ser definidos como entradas. Se qualquer deles for definido como saída, nenhuma interrupção será gerada quando surgir uma variação do nível lógico. Se estes pinos forem definidos como entradas, o seu valor actual é comparado com o valor anterior, que foi guardado quando se fez a leitura anterior do porto B. Esta interrupção pode ser habilitada/ inibida pondo a ‘1’ ou a ‘0’, o bit RBIE do registo INTCON.

**Interrupção por fim de escrita na EEPROM**

Esta interrupção é apenas de natureza prática. Como escrever num endereço da EEPROM leva cerca de 10ms (o que representa muito tempo quando se fala de um microcontrolador), não é recomendável que se deixe o microcontrolador um grande intervalo de tempo sem fazer nada, à espera do fim da operação da escrita. Assim, dispomos de um mecanismo de interrupção que permite ao microcontrolador continuar a executar o programa principal, enquanto, em simultâneo, procede à escrita na EEPROM. Quando esta operação de escrita se completa, uma interrupção informa o microcontrolador deste facto. O bit EEIF, através do qual esta informação é dada, pertence ao registo EECON1. A ocorrência desta interrupção pode ser impedida, pondo a ‘0’ o bit EEIE do registo INTCON.

**Iniciação da interrupção**

Para que num microcontrolador se possa usar um mecanismo de interrupção, é preciso proceder a algumas tarefas preliminares. Estes procedimentos são designados resumidamente por “iniciação”. Na iniciação, nós estabelecemos a que interrupções deve o microcontrolador responder e as que deve ignorar. Se não pusermos a ‘1’ o bit que permite uma certa interrupção, o programa vai ignorar a correspondente subrotina de interrupção. Por este meio, nós podemos controlar a ocorrência das interrupções, o que é muito útil.

```
clrf  INTCON              ;  todas as interrupções impedidas
movlw  B'00010000'        ;  só autorizada a interrupção externa
movwf INTCON
bsf  INTCON, GIE          ;  permitida a ocorrência de interrupçõses
```

O exemplo de cima, mostra a iniciação da interrupção externa no pino RB0 de um microcontrolador. No sítio em que vemos ‘1’, isso significa que essa interrupção está habilitada. A ocorrência de outras interrupções não é permitida, e todas as interrupções em conjunto estão mascaradas até que o bit GIE seja posto a ‘1’.

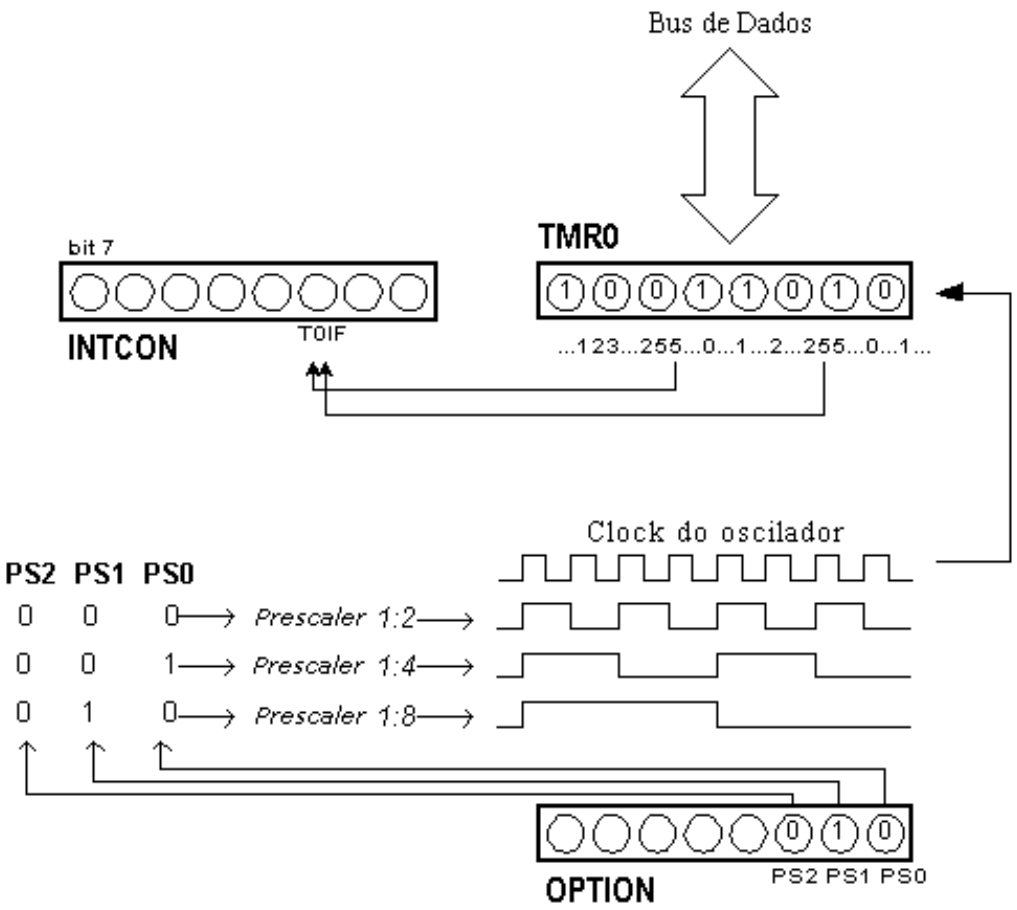
O exemplo que se segue, ilustra uma maneira típica de lidar com as interrupções. O PIC16F84 tem somente um endereço para a rotina de interrupção. Isto significa que, primeiro, é necessário identificar qual a origem da interrupção (se mais que uma fonte de interrupção estiver habilitada), e a seguir deve executar-se apenas a parte da subrotina que se refere à interrupção em causa.

```
org ISR_ADDR            ;ISR_ADDR é o endereço da rotina de interrupção
btfsc INTCON, GIE       ;bit GIE desligado ?
goto ISR_ADR            ;não, voltar ao princípio
PUSH                    ;guardar os conteúdos dos registos importantes
btfsc INTCON, RBIF      ;variação nos pinos 4, 5, 6 e 7 do porto B?
goto ISR_PORTB          ;saltar para a secção correspondente
btfsc INTCON,INTF       ;ocorreu uma interrupção externa em RB0 ?
goto ISR_RB0            ;saltar p/ esse local
btfsc INTCON, TOIF      ;o temporizador TMR0 transbordou ?
goto ISR_TMR0           ;saltar p/ essa secção
BANK1                   ;Banco 1 p/ aceder a EECON1
Btfsc EECON1, EEIF      ;escrita na EEPROM completa?
goto ISR_EEPROM         ;saltar para o endereço correspondente
BANK0                   ;Banco 0

ISR_PORTB
      :                 ;parte do código processado por uma
                        ;interrupção?
      :
      goto END_ISR      ;saltar para a saída da interrupção
ISR_RB0
      :                 ;parte de código processado pela interrupção?
      :
      goto END_ISR      ;saltar para a saída da interrupção
ISR_TMR0
      :                 ;parte de código processado pela interrupção?
      :
      goto END_ISR      ;saltar para a saída da interrupção
ISR_EEPROM
      :                 ;parte de código processado pela interrupção?
      :
      goto END_ISR      ;saltar para a saída da interrupção
END_ISR                 ;
      POP               ;recuperar os conteúdos dos
                        ;registos importantes
      RETFIE            ;regressar e pôr o bit GIE a '1'
```

O regresso de uma rotina de interrupção pode efectuar-se com as instruções RETURN, RETLW e RETFIE. Recomenda-se que seja usada a instrução RETFIE porque, essa instrução é a única que automaticamente põe a ‘1’ o bit GIE, permitindo assim que novas interrupções possam ocorrer.

## 2.7 Temporizador TMR0

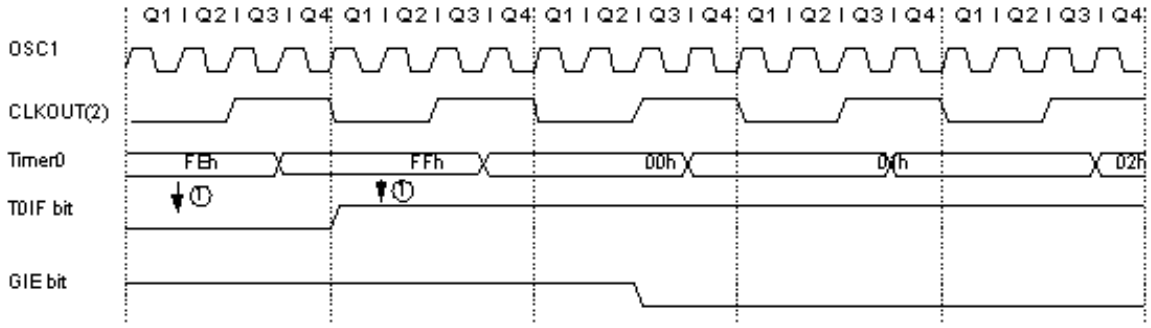
Os temporizadores são normalmente as partes mais complicadas de um microcontrolador, assim, é necessário gastar mais tempo a explicá-los. Servindo-nos deles, é possível relacionar uma dimensão real que é o tempo, com uma variável que representa o estado de um temporizador dentro de um microcontrolador. Físicamente, o temporizador é um registo cujo valor está continuamente a ser incrementado até 255, chegado a este número, ele começa outra vez de novo: 0, 1, 2, 3, 4, ...,255, 0,1, 2, 3,..., etc.




Relação entre o temporizador TMR0 e o prescaler

O incremento do temporizador é feito em simultâneo com tudo o que o microcontrolador faz. Compete ao programador arranjar maneira de tirar partido desta característica. Uma das maneiras é incrementar uma variável sempre que o microcontrolador transvaza (passa de 255 para 0). Se soubermos de quanto tempo um temporizador precisa para perfazer uma contagem completa (de 0 a 255), então, se multiplicarmos o valor da variável por esse tempo, nós obteremos o tempo total decorrido.

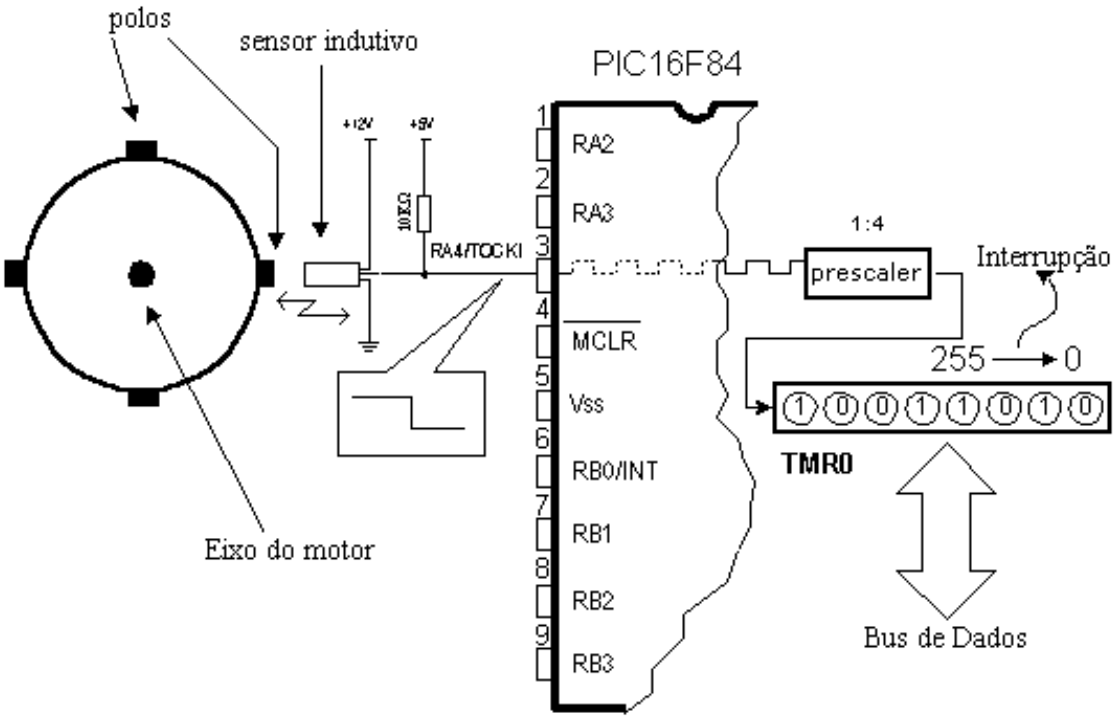
O PIC16F84, possui um temporizador de 8 bits. O número de bits determina a quantidade de valores diferentes que a contagem pode assumir, antes de voltar novamente para zero. No caso de um temporizador de 8 bits esse valor é 256. Um esquema simplificado da relação entre um temporizador e um prescaler está representado no diagrama anterior. Prescaler é a designação para a parte do microcontrolador que divide a frequência de oscilação do clock antes que os respectivos impulsos possam incrementar o temporizador. O número pelo qual a frequência de clock é dividida, está definido nos três primeiros bits do registo OPTION. O maior divisor possível é 256. Neste caso, significa que só após 256 impulsos de clock é que o conteúdo do temporizador é incrementado de uma unidade. Isto permite-nos medir grandes intervalos de tempo.




Notas:1. O estado da flag de interrupção TOIF é verificado em todos os Q1
2. CLKOUT existe só no modo de oscilador RC

Diagrama temporal de uma interrupção causada pelo temporizador TMR0

Quando a contagem ultrapassa 255, o temporizador volta de novo a zero e começa um novo ciclo de contagem até 255. Sempre que ocorre uma transição de 255 para 0, o bit TOIF do registo INTCON é posto a '1'. Se as interrupções estiverem habilitadas, é possível tirar partido das interrupções geradas e da rotina de serviço de interrupção. Cabe ao programador voltar a pôr a '0' o bit TOIF na rotina de interrupção, para que uma nova interrupção possa ser detectada. Além do oscilador de clock do microcontrolador, o conteúdo do temporizador pode também ser incrementado através de um clock externo ligado ao pino RA4/TOCKI. A escolha entre uma destas opções é feita no bit TOCS, pertencente ao registo OPTION. Se for seleccionado o clock externo, é possível definir o bordo activo do sinal (ascendente ou descendente), que vai incrementar o valor do temporizador.




Utilização do temporizador TMR0 na determinação do número de rotações completas do eixo de um motor

Na prática, um exemplo típico que é resolvido através de um clock externo e um temporizador, é a contagem do número de rotações completas do eixo de uma máquina, como por exemplo um enrolador de espiras para transformadores. Vamos considerar que o 'rotor' do motor do enrolador, contém quatro polos ou saliências. Vamos colocar o sensor indutivo à distância de 5mm do topo da saliência. O sensor indutivo irá gerar um impulso descendente sempre que a saliência se encontre alinhada com a cabeça do sensor. Cada sinal vai representar um quarto de uma rotação completa e, a soma de todas as rotações completas, ficará registado no temporizador TMR0. O programa pode ler facilmente estes dados do temporizador através do bus de dados.

O exemplo seguinte mostra como iniciar o temporizador para contar os impulsos descendentes provenientes de uma fonte de clock externa com um prescaler 1:4.

```
      clrf TMR0          ;TMR0=0
      clrf INTCON        ;Interrupções inibidas e TOIF = 0
      bsf STATUS,RP0     ;Banco 1
      movlw B'00110001'  ;prescaler 1:4; interrupção externa no bordo descendente
                         ;fonte de clock externa e resistências de pull-up do
                         ;porto B, activadas.

      movwf OPTION_REG ;OPTION_REG <- W
TO_OVFL
      btfss INTCON, TOIF      ;testando a flag de transbordo
      goto TO_OVFL            ;a interrupção não ocorreu ainda, esperar
;
; (Parte do programa que processa os dados, consoante o número de voltas)
;
goto TO_OVFL                  ; esperar que torne a transbordar
```

O mesmo exemplo pode ser implementado através de uma interrupção do modo seguinte:

```
      org 0x00           ;endereço de reset
      goto Start         ;início do programa

      org 0x04           ;endereço de interrupção
      goto TO_OVFL       ;início da rotina de interrupção

Start clrf TMR0          ;TMR0=0
      clrf INTCON        ;Intrrupções inibidas e TOIF=0
      bsf STATUS,RP0     ;Banco 1
      movlw B'00110001'  ;prescaler 1:4; interrupção externa no bordo descendente
                         ;fonte de clock externa e resistências de pull-up do
                         ;porto B, activadas.

      movwf OPTION_REG   ;OPTION_REG <- W
      bsf INTCON,TOIE    ;interrupção ao transbordar habilitada
      bsf INTCON,GIE     ; interrupções permitidas
TO_OVFL

; (Parte do programa que processa os dados, consoante o número de voltas)
;

bcf INTCON,TOIF    ;a flag de interrupção é limpa, para que a próxima interrupção
                   ;possa ser detectada.

retfie             ;regresso da rotina de interrupção
```

O prescaler tanto pode ser atribuído ao temporizador TMR0, como ao watchdog. O watchdog é um mecanismo que o microcontrolador usa para se defender contra "estouros" do programa. Como qualquer circuito eléctrico, também os microcontroladores podem ter uma falha ou algum percalço no seu funcionamento. Infelizmente, o microcontrolador também pode ter problemas com o seu programa. Quando isto acontece, o microcontrolador pára de trabalhar e mantém-se nesse estado até que alguém faça o reset. Por causa disto, foi introduzido o mecanismo de watchdog (cão de guarda). Depois de um certo período de tempo, o watchdog faz o reset do microcontrolador (o que realmente acontece, é que o microcontrolador executa o reset de si próprio). O watchdog trabalha na base de um princípio simples: se o seu temporizador transbordar, é feito o reset do microcontrolador e este começa a executar de novo o programa a partir do princípio. Deste modo, o reset poderá ocorrer tanto no caso de

funcionamento correcto como no caso de funcionamento incorrecto. O próximo passo é evitar o reset no caso de funcionamento correcto, isso é feito escrevendo zero no registo WDT (instrução CLRWDT) sempre que este está próximo de transbordar. Assim, o programa irá evitar um reset enquanto está a funcionar correctamente. Se ocorrer o "estouro" do programa, este zero não será escrito, haverá transbordo do temporizador WDT e irá ocorrer um reset que vai fazer com que o microcontrolador comece de novo a trabalhar correctamente.

O prescaler pode ser atribuído ao temporizador TMR0, ou ao temporizador do watchdog, isso é feito através do bit PSA no registo OPTION. Fazendo o bit PSA igual a '0', o prescaler é atribuído ao temporizador TMR0. Quando o prescaler é atribuído ao temporizador TMR0, todas as instruções de escrita no registo TMR0 (CLRF TMR0, MOVWF TMR0, BSF TMR0,...) vão limpar o prescaler. Quando o prescaler é atribuído ao temporizador do watchdog, somente a instrução CLRWDT irá limpar o prescaler e o temporizador do watchdog ao mesmo tempo. A mudança do prescaler está completamente sob o controle do programador e pode ser executada enquanto o programa está a correr.

Existe apenas um prescaler com o seu temporizador. Dependendo das necessidades, pode ser atribuído ao temporizador TMR0 ou ao watchdog, mas nunca aos dois em simultâneo.

## Registo de Controle OPTION

| R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 |
|---|---|---|---|---|---|---|---|
| $\overline{\text{RBPU}}$ (1) | INTEDG | T0CS | T0SE | PSA | PS2 | PS1 | PS0 |
| bit 7 | | | | | | | bit 0 |

**Legenda:**
**R** = bit p/ ler **W** = bit p/ escrever
U = Não implementado, ao ler-se dá '0' -n = valor de reset 'ao ligar'

**bit 0:2 PS0, PS1, PS2** (bits de selecção do divisor prescaler)
O prescaler e como estes bits afectam o funcionamento do microcontrolador, são abordados na secção que trata de TMR0.

| Bits | TRM0 | 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 |

**bit 3 PSA** (bit de Atribuição do Prescaler)
Bit que atribui o prescaler ou ao temporizador TMR0 ou ao temporizador do watchdog
1 = o prescaler está atribuído ao temporizador do watchdog.
0 = o prescaler está atribuído ao temporizador TMR0.

**bit 4 T0SE** (selecção de bordo activo em TMR0)
Se o temporizador estiver configurado para contar impulsos externos aplicados ao pino RA4/T0CKI, este bit vai determinar quando a contagem irá incidir sobre os impulsos ascendentes ou descendentes do sinal.
1 = bordo descendente
0 = bordo ascendente

**bit 5 T0CS** (bit de selecção de fonte de clock para TMR0)
Este pino habilita o contador/temporizador TMR0 a incrementar o seu valor ou com os impulsos do oscilador interno, isto é, a 1/4 das oscilações do clock do oscilador, ou através de impulsos externos aplicados ao pino

**RA4/T0CKI.**
1 = impulsos externos
0 = 1/4 do clock interno

**bit 6 INTEDG** (bit de selecção do bordo activo da interrupção)
Se a ocorrência de interrupções estiver habilitada, este bit vai determinar qual o bordo em que a interrupção no pino RB0/INT vai ocorrer.
1 = bordo ascendente
0 = bordo descendente

**bit 7 RBPU** (Bit de habilitação dos pull-up no porto B)
Este bit introduz ou retira as resistências de pull-up internas do porto B.
1 = resistências de 'pull-up' inseridas
0 = resistências de 'pull-up' retiradas

## 2.8 Memória de dados EEPROM

O PIC16F84 tem 64 bytes de localizações de memória EEPROM, correspondentes aos endereços de 00h a 63h e onde podemos ler e escrever. A característica mais importante desta memória é de não perder o seu conteúdo quando a alimentação é desligada. Na prática, isso significa que o que lá foi escrito permanece no microcontrolador, mesmo quando a alimentação é desligada. Sem alimentação, estes dados permanecem no microcontrolador durante mais de 40 anos (especificações do fabricante do microcontrolador PIC16F84), além disso, esta memória suporta até 10000 operações de escrita.

Na prática, a memória EEPROM é usada para guardar dados importantes ou alguns parâmetros de processamento. Um parâmetro deste tipo, é uma dada temperatura, atribuída quando ajustamos um regulador de temperatura para um processo. Se esse valor se perder, seria necessário reintroduzi-lo sempre que houvesse uma falha na alimentação. Como isto é impraticável (e mesmo perigoso), os fabricantes de microcontroladores começaram a instalar nestes uma pequena quantidade de memória EEPROM.

A memória EEPROM é colocada num espaço de memória especial e pode ser acedida através de registos especiais. Estes registos são:

- **EEDATA** no endereço 08h, que contém o dado lido ou aquele que se quer escrever.
- **EEADR** no endereço 09h, que contém o endereço do local da EEPROM que vai ser acedido
- **EECON1** no endereço 88h, que contém os bits de controle.
- **EECON2** no endereço 89h. Este registo não existe fisicamente e serve para proteger a EEPROM de uma escrita acidental.

O registo EECON1 ocupa o endereço 88h e é um registo de controle com cinco bits implementados.
Os bits 5, 6 e 7 não são usados e, se forem lidos, são sempre iguais a zero.
Os bits do registo EECON1, devem ser interpretados do modo que se segue.

### Registo EECON1

| U-0 | U-0 | U-0 | R/W-1 | R/W-1 | R/W-x | R/S-0 | R/S-x |
|---|---|---|---|---|---|---|---|
| — | — | — | EEIF (1) | WRERR | WREN | WR | RD |
| bit 7 | | | | | | | bit 0 |

**Legenda:**
**R** = bit p/ ler **W** = bit p/ escrever
U = Não implementado, ao ler-se dá '0' -n = valor de reset 'ao ligar'

**bit 0 RD** (bit de controle de leitura)
Ao pôr este bit a '1', tem início a transferência do dado do endereço definido em EEADR para o registo EEDATA. Como o tempo não é essencial, tanto na leitura como na escrita, o dado de EEDATA pode já ser usado na instrução seguinte.
1 = inicia a leitura
0 = não inicia a leitura

**bit 1 WR** (bit de controle de escrita)
Pôr este bit a '1' faz iniciar-se a escrita do dadoo a partir do registo EEDATA para o endereço especificado no registo EEADR.
1 = inicia a escrita
0 = não inicia a escrita

**bit 2 WREN** (bit de habilitação de escrita na EEPROM). Permite a escrita na EEPROM.
Se este bit não estiver a um, o microcontrolador não permite a escrita na EEPROM.
1 = a escrita é permitida
0 = não se pode escrever

**bit 3 WRERR** ( Erro de escrita na EEPROM). Erro durante a escrita na EEPROM
Este bit é posto a '1' só em casos em que a escrita na EEPROM tenha sido interrompida por um sinal de reset ou por um transbordo no temporizador do watchdog (no caso de este estar activo).
1 = ocorreu um erro
0 = não houve erros

**bit 4 EEIF** (bit de interrupção por operação de escrita na EEPROM completa) Bit usado para informar que a escrita do dadoo na EEPROM, terminou.
Quando a escrita tiver terminado, este bit é automaticamente posto a '1'. O programador tem que repôr a '0' o bit EEIF no seu programa, para que possa detectar o fim de uma nova operação de escrita.
1 = escrita terminada
0 = a escrita ainda não terminou ou não começou.

**Lendo a Memória EEPROM**

Pondo a '1' o bit RD inicia-se a transferência do dado do endereço guardado no registo EEADR para o registo EEDATA. Como para ler os dados não é preciso tanto tempo como a escrevê-los, os dados extraídos do registo EEDATA podem já ser usados na instrução seguinte.

Uma porção de um programa que leia um dado da EEPROM, pode ser semelhante ao seguinte:

```
bcf   STATUS, RP0       ;banco 0, porque EEADR está em 09h
movlw 0x00              ;endereço do local a ler
movwf EEADR             ;endereço transferido para EEADR
bsf   STATUS, RP0       ;banco 1, porque EECON1 está em 88h
bsf   EECON1, RD        ;ler da EEPROM
bcf   STATUS, RP0       ;banco 0, porque EEDATA está em 08h
movf  EEDATA, W         ;W <-- EEDATA
```

Depois da última instrução do programa, o conteúdo do endereço 0 da EEPROM pode ser encontrado no registo de trabalho w.

**Escrevendo na Memória EEPROM**

Para escrever dados num local da EEPROM, o programador tem primeiro que endereçar o registo EEADR e introduzir a palavra de dados no registo EEDATA. A seguir, deve colocar-se o bit WR a '1', o que faz desencadear o processo. O bit WR deverá ser posto a '0' e o bit EEIF será posto a '1' a seguir à operação de escrita, o que pode ser usado no processamento de interrupções. Os valores 55h e AAh são as primeira e segunda chaves que tornam impossível que ocorra uma escrita acidental na EEPROM. Estes dois valores são escritos em EECON2 que serve apenas para isto, ou seja, para receber estes dois valores e assim prevenir contra uma escrita acidental na memória EEPROM. As linhas do programa marcadas como 1, 2, 3 e 4 têm que ser executadas por esta ordem em intervalos de tempo certos. Portanto, é muito importante desactivar as interrupções que possam interferir com a temporização necessária para executar estas instruções. Depois da operação de escrita, as interrupções podem, finalmente, ser de novo habilitadas.

Exemplo da porção de programa que escreve a palavra 0xEE no primeiro endereço da memória EEPROM:

```
        bcf   STATUS, RP0       ;banco 0, porque EEADR está em 09h
        movlw 0x00              ;endereço do local de memória
                                ;em que se quer escrever
        movwf EEADR             ;endereço transferido para
                                ;EEADR
        movlw 0xEE              ;escrever o valor 0xEE
        movwf EEDATA            ;dado no registo EEDATA
        bsf STATUS, RP0         ;banco 1
        bcf INTCON, GIE         ;todas as interrupções impedidas
        bsf EECON1, WREN        ;permissão de escrita
        movlw 55h
1)      movwf EECON2            ;1ª chave    55h --> EECON2
2)      movlw AAh
3)      movwf EECON2            ;2ª chave     AAh --> EECON2
4)      bsf EECON1,WR           ;iniciar a escrita
        bsf INTCON, GIE         ;interrupções habilitadas
```

Recomenda-se que WREN esteja sempre inactivo, excepto quando se está a escrever uma palavra de dados na EEPROM, deste modo, a possibilidade de uma escrita acidental é mínima.
Todas as operações de escrita na EEPROM 'limpam' automaticamente o local de memória, antes de escrever de novo nele !

# CAPÍTULO 3

## Conjunto de Instruções



## Introdução

Já dissemos que um microcontrolador não é como qualquer outro circuito integrado. Quando saem da cadeia de produção, a maioria dos circuitos integrados, estão prontos para serem introduzidos nos dispositivos, o que não é o caso dos microcontroladores. Para que um microcontrolador cumpra a sua tarefa, nós temos que lhe dizer exactamente o que fazer, ou, por outras palavras, nós temos que escrever o programa que o microcontrolador vai executar. Neste capítulo iremos descrever as instruções que constituem o assembler, ou seja, a linguagem de baixo nível para os microcontroladores PIC.

## Conjunto de Instruções da Família PIC16Cxx de Microcontroladores

O conjunto completo compreende 35 instruções e mostra-se na tabela que se segue. Uma razão para este pequeno número de instruções resulta principalmente do facto de estarmos a falar de um microcontrolador RISC cujas instruções foram optimizadas tendo em vista a rapidez de funcionamento, simplicidade de arquitectura e compacidade de código. O único inconveniente, é que o programador tem que dominar a técnica “desconfortável” de fazer o programa com apenas 35 instruções.

## Transferência de dados

A transferência de dados num microcontrolador, ocorre entre o registo de trabalho (W) e um registo ‘f’ que representa um qualquer local de memória na RAM interna (quer se trate de um registo especial ou de um registo de uso genérico).

As primeiras três instruções (observe a tabela seguinte) referem-se à escrita de uma constante no registo W (MOVLW é uma abreviatura para MOVa Literal para W), à cópia de um dado do registo W na RAM e à cópia de um dado de um registo da RAM no registo W (ou nele próprio, caso em que apenas a flag do zero é afectada) . A instrução CLRF escreve a constante 0 no registo ‘f’ e CLRW escreve a constante 0 no registo W. A instrução SWAPF troca o nibble (conjunto de 4 bits) mais significativo com o nibble menos significativo de um registo, passando o primeiro a ser o menos significativo e o outro o mais significativo do registo.

## Lógicas e aritméticas

De todas as operações aritméticas possíveis, os microcontroladores PIC, tal como a grande maioria dos outros microcontroladores, apenas suportam a subtracção e a adição. Os bits ou flags C, DC e Z, são afectados conforme o resultado da adição ou da subtracção, com uma única excepção: uma vez que a subtracção é executada como uma adição com um número negativo, a flag C (Carry), comporta-se inversamente no que diz respeito à subtracção. Por outras palavras, é posta a '1' se a operação é possível e posta a '0' se um número maior tiver que ser subtraído de outro mais pequeno.

A lógica dentro do PIC tem a capacidade de executar as operações AND, OR, EX-OR, complemento (COMF) e rotações (RLF e RRF).
Estas últimas instruções, rodam o conteúdo do registo através desse registo e da flag C de uma casa para a esquerda (na direcção do bit 7), ou para a direita (na direcção do bit 0). O bit que sai do registo é escrito na flag C e o conteúdo anterior desta flag, é escrito no bit situado do lado oposto no registo.

## Operações sobre bits

As instruções BCF e BSF põem a '0' ou a '1' qualquer bit de qualquer sítio da memória. Apesar de parecer uma operação simples, ela é executada do seguinte modo, o CPU primeiro lê o byte completo, altera o valor de um bit e, a seguir, escreve o byte completo no mesmo sítio.

## Direcção de execução de um programa

As instruções GOTO, CALL e RETURN são executadas do mesmo modo que em todos os outros microcontroladores, a diferença é que a pilha é independente da RAM interna e é limitada a oito níveis. A instrução 'RETLW k' é idêntica à instrução RETURN, excepto que, ao regressar de um subprograma, é escrita no registo W uma constante definida pelo operando da instrução. Esta instrução, permite-nos implementar facilmente listagens (também chamadas tabelas de lookup). A maior parte das vezes, usamo-las determinando a posição do dado na nossa tabela adicionando-a ao endereço em que a tabela começa e, então, é lido o dado nesse local (que está situado normalmente na memória de programa).

A tabela pode apresentar-se como um subprograma que consiste numa série de instruções 'RETLW k' onde as constantes 'k', são membros da tabela.

```
Main    movlw 2
        call Lookup
Lookup  addwf PCL, f
        retlw k
        retlw k1
        retlw k2
          :
          :
        retlw kn
```

Nós escrevemos a posição de um membro da nossa tabela no registo W e, usando a instrução CALL, nós chamamos o subprograma que contém a tabela. A primeira linha do subprograma 'ADDWF PCL, f', adiciona a posição na tabela e que está escrita em W, ao endereço do início da tabela e que está no registo PCL, assim, nós obtemos o endereço real do dado da tabela na memória de programa. Quando regressamos do subprograma, nós vamos ter no registo W o conteúdo do membro da tabela endereçado. No exemplo anterior, a constante 'k2' estará no registo W, após o retorno do subprograma.

RETFIE (RETurn From Interrupt – Interrupt Enable ou regresso da rotina de interrupção com as interrupções habilitadas) é um regresso da rotina de interrupção e difere de RETURN apenas em que, automaticamente, põe a '1' o bit GIE (habilitação global das interrupções). Quando a interrupção começa, este bit é automaticamente reposto a '0'. Também quando a interrupção tem início, somente o valor do contador de programa é posto no cimo da pilha. Não é fornecida uma capacidade automática de armazenamento do registo de estado.

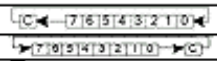
Os saltos condicionais estão sintetizados em duas instruções: BTFSC e BTFSS. Consoante o estado lógico do bit do registo 'f' que está a ser testado, a instrução seguinte no programa é ou não executada.

## Período de execução da instrução

Todas as instruções são executadas num único ciclo, excepto as instruções de ramificação condicional se a condição for verdadeira, ou se o conteúdo do contador de programa for alterado pela instrução. Nestes casos, a execução requer dois ciclos de instrução e o segundo ciclo é executado como sendo um NOP (Nenhuma Operação). Quatro oscilações de clock perfazem um ciclo de instrução. Se estivermos a usar um oscilador com 4MHz de frequência, o tempo normal de execução de uma instrução será de 1μs e, no caso de uma ramificação condicional de 2μs.

## Listagem das palavras

**f** qualquer local de memória num microcontrolador
**W** registo de trabalho
**b** posição de bit no registo 'f'
**d** registo de destino
*label* grupo de oito caracteres que marca o início de uma parte do programa (rótulo)
**TOS** cimo da pilha
**[ ]** opcional
<> grupo de bits num registo

| Menemónica | | Descrição | | Flag | CLK | Notas |
|---|---|---|---|---|---|---|
| | | **Transferência de dados** | | | | |
| MOVLW | k | Mova literal para W | k → W | | 1 | |
| MOVWF | f | Mova W para f | W → f | | 1 | |
| MOVF | f, d | Mova f | f → d | Z | 1 | 1, 2 |
| CLRW | - | Clear W (limpar W) | 0 → W | Z | 1 | |
| CLRF | f | Clear f (limpar f) | 0 → f | Z | 1 | 2 |
| SWAPF | f, d | Swap nibbles in f (trocar) | f(7:4),(3:0)→f(3:0),(7:4) | | 1 | 1, 2 |
| | | **Lógicas e Aritméticas** | | | | |
| ADDLW | k | Adicionar literal a W | W+k → W | C, DC, Z | 1 | |
| ADDWF | f, d | Adicionar W a f | W+f → d | C, DC, Z | 1 | 1, 2 |
| SUBLW | k | Subtrair W de literal | k-W →W | C, DC, Z | 1 | |
| SUBWF | f, d | Subtrair W de f | f-W → d | C, DC, Z | 1 | 1, 2 |
| ANDLW | k | AND literal com W | W .AND. k → W | Z | 1 | |
| ANDWF | f, d | AND W com f | W .AND. f → d | Z | 1 | 1, 2 |
| IORLW | k | Inclusivo OR de literal com W | W .OR. k → W | Z | 1 | |
| IORWF | f, d | Inclusivo OR de W com f | W .OR. f → d | Z | 1 | 1, 2 |
| XORWF | f, d | Exclusivo OR de W com f | W . XOR. f → d | Z | 1 | 1, 2 |
| XORLW | k | Exclusivo OR de literal com W | W .XOR. k → W | Z | 1 | |
| INCF | f, d | Incrementar f | f+1 → f | Z | 1 | 1, 2 |
| DECF | f, d | Decrementar f | f-1 → f | Z | 1 | 1, 2 |
| RLF | f, d | Rode f p/ esquerda com o carry |  | C | 1 | 1, 2 |
| RRF | f, d | Rode f p/ a direita com o carry |  | C | 1 | 1, 2 |
| COMF | f, d | Complementar f | $\overline{f}$ → d | Z | 1 | 1, 2 |
| | | **Operações sobre bits** | | | | |
| BCF | f, b | Bit Clear f (bit de f a '0') | 0 → f(b) | | 1 | 1, 2 |
| BSF | f, b | Bit Set f (bit de f a '1') | 1 → f(b) | | 1 | 1, 2 |
| | | **Direccionamento do programa** | | | | |
| BTFSC | f, b | Bit Test f, Salte se Clear ('0') | salte se f(b)=0 | | 1(2) | 3 |
| BTFSS | f, b | Bit Test f, Salte se Set ('1') | salte se f(b)=1 | | 1(2) | 3 |
| DECFSZ | f, d | Decremente f, salte se der 0 | f-1 → d, salte se der 0 | | 1(2) | 1, 2, 3 |
| INCFSZ | f, d | Incremente f, salte se der 0 | f+1 → d, salte se der 0 | | 1(2) | 1, 2, 3 |
| GOTO | k | Go to address (Ir p/ endereço) | k → PC | | 2 | |
| CALL | k | Chamar subrotina | PC → TOS, k → PC | | 2 | |
| RETURN | - | Retorno de subrotina | TOS → PC | | 2 | |
| RETLW | k | Retorno com literal em W | k → W, TOS → PC | | 2 | |
| RETFIE | - | Retorno de interrupção | TOS → PC, 1 → GIE | | 2 | |
| | | **Outras instruções** | | | | |
| NOP | - | Nenhuma operação | | | 1 | |
| CLRWDT | - | Temporizador do Watchdog=0 | 0 → WDT, 1 → TO, 1 → PD | $\overline{TO}$, $\overline{PD}$ | 1 | |
| SLEEP | - | Entrar no modo 'sleep' | 0 → WDT, 1 → TO, 0 → PD | $\overline{TO}$, $\overline{PD}$ | 1 | |

| | | | | | | |
|---|---|---|---|---|---|---|
| **Outras instruções** | | | | | | |
| NOP | - | Nenhuma operação | | | 1 | |
| CLRWDT | - | Temporizador do Watchdog=0 | 0 → WDT, 1 → TO, 1 → PD | $\overline{TO}$, $\overline{PD}$ | 1 | |
| SLEEP | - | Entrar no modo 'sleep' | 0 → WDT, 1 → TO, 0 → PD | $\overline{TO}$, $\overline{PD}$ | 1 | |

***1** Se o porto de entrada/saída for o operando origem, é lido o estado dos pinos do microcontrolador.
***2** Se esta instrução for executada no registo TMR0 e se d=1, o prescaler atribuído a esse temporizador é automaticamente limpo.
***3** Se o PC for modificado ou se resultado do teste for verdadeiro, a instrução é executada em dois ciclos.

**Página anterior** **Tabela de conteúdos** **Página seguinte**

# CAPÍTULO 4

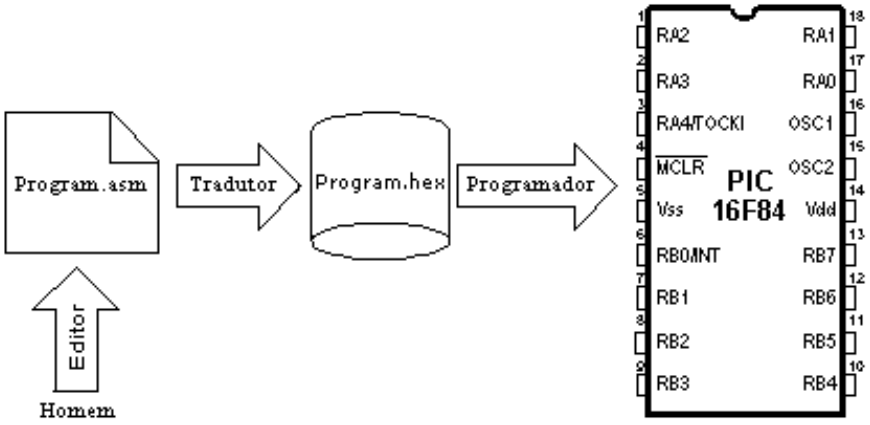
## Programação em Linguagem Assembly



## Introdução

A capacidade de comunicar é da maior importância nesta área. Contudo, isso só é possível se ambas as partes usarem a mesma linguagem, ou seja, se seguirem as mesmas regras para comunicarem. Isto mesmo se aplica à comunicação entre os microcontroladores e o homem. A linguagem que o microcontrolador e o homem usam para comunicar entre si é designada por “linguagem assembly”. O próprio título não tem um significado profundo, trata-se de apenas um nome como por exemplo inglês ou francês. Mais precisamente, “linguagem assembly” é apenas uma solução transitória. Os programas escritos em linguagem assembly devem ser traduzidos para uma “linguagem de zeros e uns” de modo a que um microcontrolador a possa receber. “Linguagem assembly” e “assembler” são coisas diferentes. A primeira, representa um conjunto de regras usadas para escrever um programa para um microcontrolador e a outra, é um programa que corre num computador pessoal que traduz a

linguagem assembly para uma linguagem de zeros e uns. Um programa escrito em “zeros” e “uns” diz-se que está escrito em “linguagem máquina”.




**O processo de comunicação entre o homem e o microcontrolador**

Fisicamente, “**Programa**” representa um ficheiro num disco de computador (ou na memória se estivermos a ler de um microcontrolador) e é escrito de acordo com as regras do assembly ou qualquer outra linguagem de programação de microcontroladores. O homem pode entender a linguagem assembly já que ela é constituída por símbolos alfabéticos e palavras. Ao escrever um programa, certas regras devem ser seguidas para alcançar o efeito desejado. Um **Tradutor** interpreta cada instrução escrita em linguagem assembly como uma série de zeros e uns com significado para a lógica interna do microcontrolador.
Consideremos, por exemplo, a instrução “RETURN” que um microcontrolador utiliza para regressar de um subprograma.
Quando o assembler a traduz, nós obtemos uma série de uns e zeros correspondentes a 14 bits que o microcontrolador sabe como interpretar.

**Exemplo:** RETURN 00 0000 0000 1000

Analogamente ao exemplo anterior, cada instrução assembly é interpretada na série de zeros e uns correspondente.
O resultado desta tradução da linguagem assembly, é designado por um ficheiro de “execução”. Muitas vezes encontramos o nome de ficheiro “HEX”. Este nome provém de uma representação hexadecimal desse ficheiro, bem como o sufixo “hex" no título, por exemplo “correr.hex". Uma vez produzido, o ficheiro de execução é inserido no microcontrolador através de um programador.

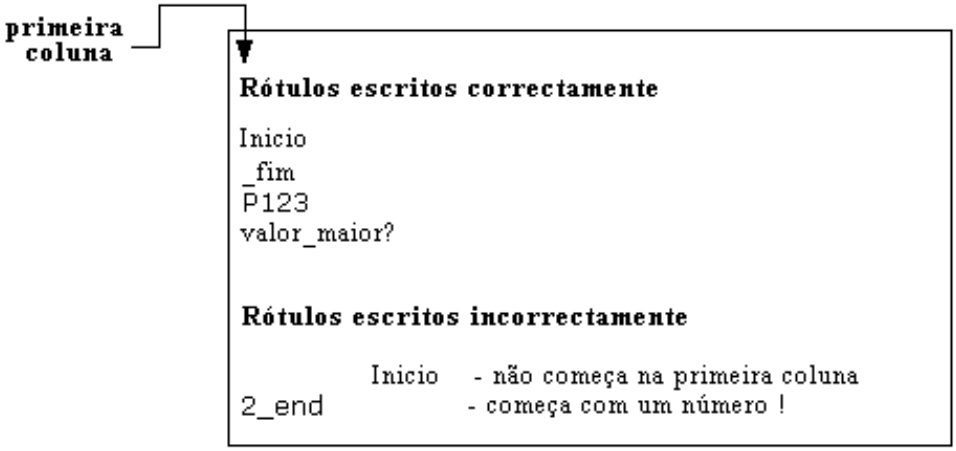
Um programa em **Linguagem Assembly** é escrito por intermédio de um processador de texto (editor) e é capaz de produzir um ficheiro ASCII no disco de um computador ou em ambientes próprios como o MPLAB – que vai ser explicado no próximo capítulo.

## Linguagem Assembly

Os elementos básicos da linguagem assembly são:

- Labels (rótulos)
- Instruções
- Operandos
- Directivas
- Comentários

Um **Label** (rótulo) é uma designação textual (geralmente de fácil leitura) de uma linha num programa ou de uma secção de um programa para onde um microcontrolador deve saltar ou, ainda, o início de um conjunto de linhas de um programa. Também pode ser usado para executar uma ramificação de um programa (tal como Goto....), o programa pode ainda conter uma condição que deve ser satisfeita, para que uma instrução Goto seja executada. É importante que um rótulo (label) seja iniciado com uma letra do alfabeto ou com um traço baixo "_". O comprimento de um rótulo pode ir até 32 caracteres. É também importante que o rótulo comece na primeira coluna.




## Instruções

As instruções são específicas para cada microcontrolador, assim, se quisermos utilizar a linguagem assembly temos que estudar as instruções desse microcontrolador. O modo como se escreve uma instrução é designado por "sintaxe". No exemplo que se segue, é possível reconhecer erros de escrita, dado que as instruções movlp e gotto não existem no microcontrolador PIC16F84.

**Instruções escritas correctamente**

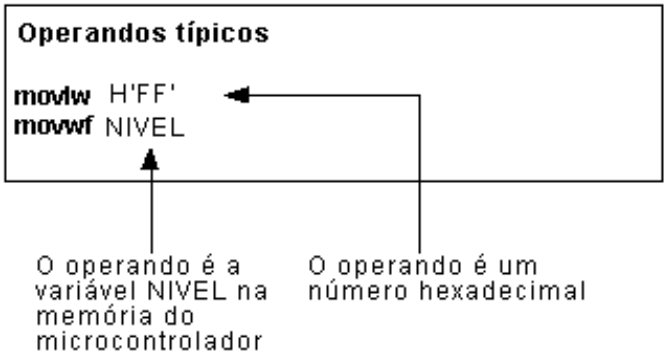
```
movlw   H'FF'
goto    Inicio
```

**Instruções incorrectamente escritas**

```
movlp   H'FF'
gotto   Inicio
```

## Operandos

Operandos são os elementos da instrução necessários para que a instrução possa ser executada. Normalmente são **registos, variáveis e constantes**. As constantes são designadas por "literais". A palavra literal significa "número".




## Comentários

**Comentário** é um texto que o programador escreve no programa afim de tornar este mais claro e legível. É colocado logo a seguir a uma instrução e deve começar com uma semi-vírgula ";".

## Directivas

Uma **directiva** é parecida com uma instrução mas, ao contrário desta, é independente do tipo de microcontrolador e é uma característica inerente à própria linguagem assembly. As directivas servem-se de variáveis ou registos para satisfazer determinados propósitos. Por exemplo, NIVEL, pode ser uma designação para uma variável localizada no endereço 0Dh da memória RAM. Deste modo, a variável que reside nesse endereço, pode ser acedida pela palavra NIVEL. É muito mais fácil a um programador recordar a palavra NIVEL, que lembrar-se que o endereço 0Dh contém informação sobre o nível.

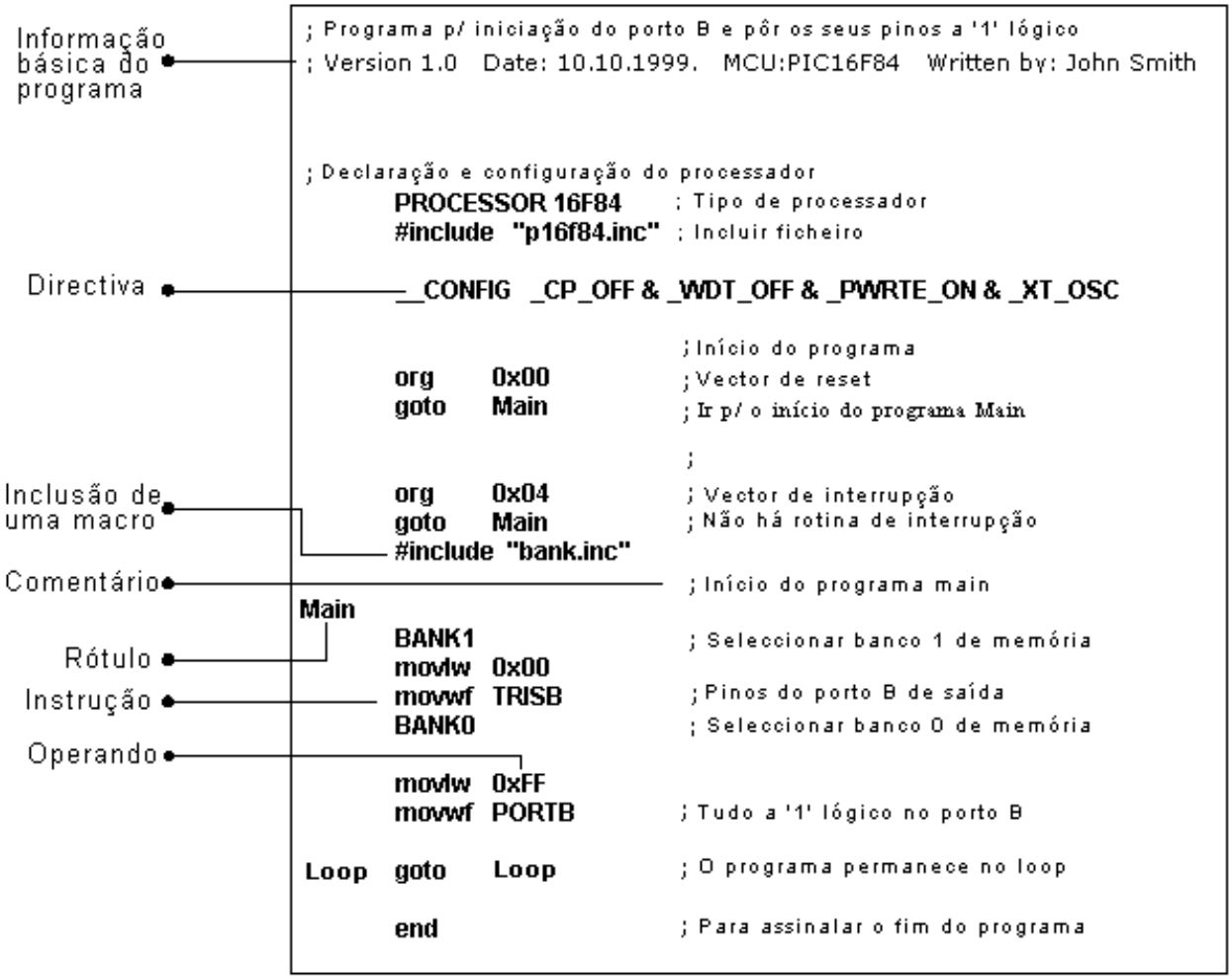
**Algumas directivas usadas frequentemente:**

```
PROCESSOR 16F84
#include    "p16f84.inc"

__CONFIG   _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC
```

## Exemplo de como se escreve um programa

O exemplo que se segue, mostra como um programa simples pode ser escrito em linguagem assembly, respeitando regras básicas.

Quado se escreve um programa, além das regras fundamentais, existem princípios que, embora não obrigatórios é conveniente, serem seguidos. Um deles, é escrever no seu início, o nome do programa, aquilo que o programa faz, a versão deste, a data em que foi escrito, tipo de microcontrolador para o qual foi escrito e o nome do programador.



```
end                    ; Para assinalar o fim do programa
```

Uma vez que estes dados não interessam ao tradutor de assembly, são escritos na forma de **comentários**. Deve ter-se em atenção que um comentário começa sempre com ponto e vírgula e pode ser colocado na linha seguinte ou logo a seguir à instrução.
Depois deste comentário inicial ter sido escrito, devem incluir-se as directivas. Isto mostra-se no exemplo de cima.

Para que o seu funcionamento seja correcto, é preciso definir vários parâmetros para o microcontrolador, tais como:
- tipo de oscilador
- quando o temporizador do watchdog está ligado e
- quando o circuito interno de reset está habilitado.

Tudo isto é definido na directiva seguinte:

__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

Logo que todos os elementos de que precisamos tenham sido definidos, podemos começar a escrever o programa. Primeiro, é necessário definir o endereço para que o microcontrolador deve ir quando se liga a alimentação. É esta a finalidade de (org 0x00).
O endereço para onde um programa salta se ocorrer uma interrupção é (org 0x04).
Como este é um programa simples, é suficiente dirigir o microcontrolador para o início de um programa com uma instrução "**goto Main**" (Main = programa principal).

As instruções encontradas em **Main**, seleccionam o banco 1 (BANK1) de modo a poder aceder-se ao registo TRISB, afim de que o porto B seja definido como uma saída (movlw 0x00, movwf TRISB).

O próximo passo é seleccionar o banco de memória 0 e colocar os bits do porto B no estado lógico '1' e, assim, o programa principal fica terminado.
É preciso, no entanto, um outro ciclo (loop), onde o microcontrolador possa permanecer sem que ocorram erros. Trata-se de um 'loop' infinito que é executado continuamente, enquanto a alimentação não for desligada.
Finalmente, é necessário colocar a palavra "end" no fim de cada programa, de modo a informar o tradutor de assembly de que o programa não contém mais instruções.

## Directivas de controle

### 4.1 #DEFINE Troca de uma porção de texto por outra

**Sintaxe**:
#define<nome> [< texto atribuído a nome > ]

**Descrição:**
De cada vez que a palavra <nome> aparece no programa, vai ser substituída por <texto atribuído a nome>.

**Exemplo:**

#define ligado 1
#define desligado 0

**Directivas similares:** #UNDEFINE, IFDEF, IFNDEF

### 4.2 INCLUDE Incluir um ficheiro adicional num programa

**Sintaxe:**
include <<nome_do_ficheiro>>
include "<nome_do_ficheiro>"

**Descrição:**
A aplicação desta directiva faz com que um ficheiro completo seja copiado para o local em que a directiva "include" se encontra. Se o nome do ficheiro estiver entre aspas, estamos a lidar com um ficheiro do sistema, se não estiver

entre aspas, mas sim entre os sinais < >, trata-se de um ficheiro do utilizador. A directiva “include”, contribui para uma melhor apresentação do programa principal.

**Exemplo:**
include < regs.h >
include “subprog.asm”

### 4.3 CONSTANT Atribui um valor numérico constante a uma designação textual

**Sintaxe:**
constant < nome > = < valor >

**Descrição:**
Cada vez que < nome > aparece no programa, é substituído por < valor > .

**Exemplo:**
constant MAXIMO = 100
constant Comprimento = 30

**Directivas similares:** SET, VARIABLE

### 4.4 VARIABLE Atribui um valor numérico variável à designação textual

**Sintaxe:**
variable < nome > = < valor >

**Descrição:**
Ao utilizar esta directiva, a designação textual muda o seu valor.
Difere da directiva CONSTANT no facto de, depois de a directiva ser aplicada, o valor da designação textual poder variar.

**Exemplo:**
variable nivel = 20
variable tempo = 13

**Directivas similares:** SET, CONSTANT

### 4.5 SET Definir uma variável assembler

**Sintaxe:**

< nome_variavel > set <valor>

**Descrição:**
À variável < nome_variavel > é atribuída a expressão <valor> . A directiva SET é semelhante a EQU, mas com a directiva SET é possível tornar a definir a variável com outro valor.

**Exemplo:**
nivel set 0
comprimento set 12
nivel set 45

**Directivas similares:** EQU, VARIABLE

### 4.6 EQU Definindo uma constante em assembler

**Sintaxe:**
< nome_da_constante > equ < valor >

**Descrição:**
Ao nome de uma constante < nome_de_constante > é atribuído um valor < valor >

**Exemplo:**
cinco equ 5
seis equ 6
sete equ 7

**Instruções similares:** SET

### 4.7 ORG Define o endereço a partir do qual o programa é armazenado na memória do microcontrolador

**Sintaxe:**
<rótulo> org <valor>

**Descrição:**
Esta é a directiva mais frequentemente usada. Com esta directiva nós definimos em que sítio na memória de programa o programa vai começar.

**Exemplo:**
Inicio org 0x00
movlw 0xFF
movwf PORTB

Estas duas instruções a seguir à directiva 'org', são guardadas a partir do endereço 00.

### 4.8 END Fim do programa

**Sintaxe:**
end

**Descrição:**
No fim do programa, é necessário colocar a directiva 'end', para que o tradutor do assembly (assembler), saiba que não existem mais instruções no programa.

**Exemplo:**
.
.
movlw 0xFF
movwf PORTB
end

## Instruções condicionais

### 4.9 IF Ramificação condicional do programa

**Sintaxe:**
if <termo_condicional>

**Descrição:**
Se a condição em <termo_condicional> estiver satisfeita, a parte do programa que se segue à directiva IF, deverá ser executada. Se a condição não for satisfeita, então é executada a parte que se segue às directivas ELSE ou ENDIF.

**Exemplo:**

if nivel = 100
goto ENCHER
else
goto DESPEJAR
endif

**Directivas similares:** ELSE, ENDIF

### 4.10 ELSE Assinala um bloco alternativo se a condição termo_condicional presente em 'IF' não se verificar

**Sintaxe:**
Else

**Descrição:**
Usado com a directiva IF como alternativa no caso de termo_condicional ser falso.

**Exemplo:**
```
if tempo < 50
goto DEPRESSA
else goto DEVAGAR
endif
```

**Instruções similares:** ENDIF, IF

### 4.11 ENDIF Fim de uma secção condicional do programa

**Sintaxe:**
endif

**Descrição:**
Esta directiva é escrita no fim de um bloco condicional, para informar o tradutor do assembly de que o bloco condicional terminou.

**Exemplo:**
```
if nivel = 100
goto METER
else
goto TIRAR
endif
```

**Directivas similares:** ELSE, IF

### 4.12 WHILE A execução da secção do programa prossegue, enquanto a condição se verificar

**Sintaxe:**
```
while <condição>
.
endw
```

**Descrição:**
As linhas do programa situadas entre WHILE e ENDW devem ser executadas, enquanto a condição for verdadeira. Se a condição deixar de se verificar, o programa deverá executar as instruções a partir da linha que sucede a ENDW. O número de instruções compreendidas entre WHILE e ENDW pode ir até 100 e podem ser executadas até 256 vezes.

**Exemplo:**
```
while i < 10
i = i + 1
endw
```

### 4.13 ENDW Fim da parte condicional do programa

**Sintaxe:**
endw

**Descrição:**
Esta directiva é escrita no fim do bloco condicional correspondente a WHILE, assim, o assembler fica a saber que o bloco condicional chegou ao fim.

**Exemplo:**
while i < 10
i = i + 1
.
endw

**Directivas similares:** WHILE

### 4.14 IFDEF Executar uma parte do programa se um símbolo estiver definido

**Sintaxe:**
ifdef < designação >

**Descrição:**
Se a designação <designação> tiver sido previamente definida (normalmente através da directiva #DEFINE), as instruções que se lhe sucedem serão executadas até encontrarmos as directivas ELSE ou ENDIF.

**Exemplo:**
#define teste
.
ifdef teste ; como teste foi definido
.....; as instruções nestas linhas vão ser executadas
endif

**Directivas similares:** #DEFINE, ELSE, ENDIF, IFNDEF, #UNDEFINE

### 4.15 IFNDEF Execução de uma parte do programa se o símbolo não tiver sido definido

**Sintaxe:**
ifndef <designação>

**Descrição:**
Se a designação <designação> não tiver sido previamente definida ou se esta definição tiver sido mandada ignorar através da directiva #UNDEFINE, as instruções que se seguem deverão ser executadas, até que as directivas ELSE ou ENDIF, sejam alcançadas.

**Exemplo:**
#define teste
.....
#undefine teste
.....
ifndef teste ; como teste não está definido
..... ; as instruções nestas linhas são executadas
endif

**Directivas similares:** #DEFINE, ELSE, ENDIF, IFDEF, #UNDEFINE

## Directivas de Dados

### 4.16 CBLOCK Definir um bloco para as constantes nomeadas

**Sintaxe:**
Cblock [< termo >]
<rótulo> [:<incremente>], <rótulo> [:<incremente>].....
endc

**Descrição:**

Esta directiva é usada para atribuir valores às constantes a seguir nomeadas. A cada termo seguinte, é atribuído um valor superior em uma unidade ao anterior. No caso de <incremente> estar preenchido, então é o valor de <incremente> que é adicionado à constante anterior.
O valor do parâmetro <termo>, é o valor inicial. Se não for dado, então, por defeito, é considerado igual a zero.

**Exemplo:**
cblock 0x02
primeiro, segundo ; primeiro = 0x02, segundo = 0x03
terceiro ;terceiro = 0x04
endc

cblock 0x02
primeiro : 4, segundo : 2 ; primeiro = 0x06, segundo = 0x08
terceiro ; terceiro = 0x09
endc

**Directivas similares:** ENDC

### 4.17 ENDC Fim da definição de um bloco de constantes

**Sintaxe:**
endc

**Descrição:**
Esta directiva é utilizada no fim da definição de um bloco de constantes, para que o tradutor de assembly saiba que não há mais constantes.

**Directivas similares:** CBLOCK

### 4.18 DB Definir um byte de dados

**Sintaxe:**
[<termo>] db <termo> [, <termo>,.....,<termo>]

**Descrição:**
Esta directiva reserva um byte na memória de programa. Quando há mais termos a quem é preciso atribuir bytes, eles serão atribuídos um após outro.

**Exemplo:**
db ‘t’, 0x0f, ‘e’, ‘s', 0x12

**Instruções similares:** DE, DT

### 4.19 DE – Definir byte na memória EEPROM

**Sintaxe:**
[<termo>] de <termo> [, <termo>,.....,<termo>]

**Descrição:**
Esta directiva reserva um byte na memória EEPROM. Apesar de ser destinada em primeiro lugar para a memória EEPROM, também pode ser usada em qualquer outro local de memória.

**Exemplo:**
org H’2100’
de “Versão 1.0”, 0

**Directivas similares:** DB, DT

### 4.20 DT Definindo uma tabela de dados

**Sintaxe:**
[<termo>] dt <termo> [, <termo>,.....,<termo>]

**Descrição:**
Esta directiva vai gerar uma série de instruções RETLW, uma instrução para cada termo.

dt “Mensagem” , 0
dt primeiro, segundo, terceiro

**Directivas similares:** DB, DE

## Configurando uma directiva

### 4.21 __CONFIG Estabelecer os bits de configuração

**Sintaxe:**
__config<termo> ou __config <endereço>, <termo>

**Descrição:**
São definidos o tipo de oscilador, e a utilização do watchdog e do circuito de reset interno. Antes de usar esta directiva, tem que declarar-se o processador através da directiva PROCESSOR.

**Exemplo:**
__CONFIG _CP_OFF & _WDT_OFF & PWRTE_ON & _XT_OSC

**Directivas similares:** __IDLOCS, PROCESSOR

### 4.22 PROCESSOR Definindo o modelo de microcontrolador

**Sintaxe:**
processor <tipo_de_microcontrolador>

**Descrição:**
Esta directiva, estabelece o tipo de microcontrolador em que o programa vai correr.

**Exemplo:**
processor 16f84

## Operadores aritméticos de assembler

| Operador | Descrição | Exemplo |
|---|---|---|
| $ | Valor actual do contador de programa | goto $ +3 |
| ( | Parêntesis esquerdo | 1 + ( d * 4 ) |
| ) | Parêntesis direito | ( Length + 1 ) * 256 |
| ! | NOT (complemento lógico) | if ! ( a - b ) |
| - | Complemento | flags = -flags |
| - | Negação (complemento p/ 2) | -1 * Length |
| high | Return o byte mais alto | movlw high CTR_Table |
| low | Return o byte - significativo | movlw low CTR_Table |
| * | Multiplicação | a = b * c |
| / | Divisão | a = b / c |
| % | Módulo | entry_len = tot_len % 16 |
| + | Adição | tot_len = entry_len * 8 + 1 |
| - | Subtracção | entry_len = ( tot - 1 ) / 8 |
| << | Deslocamento p/ a esquerda | val = flags << 1 |
| >> | Deslocamento p/ a direita | val = flags >> 1 |
| >= | Maior ou igual | if entry_idx >= num_entries |
| > | Maior que | if entry_idx > num_entries |
| < | Menor que | if entry_idx < num_entries |
| <= | Menor que ou igual | if entry_idx <= num_entries |
| == | Igual | if entry_idx == num_entries |
| != | Não igual | if entry_idx != num_entries |
| & | 'E' bit a bit | flags = flags & ERROR_BIT |
| ^ | OU exclusivo bit a bit | flags = flags ^ ERROR_BIT |
| \| | OU bit a bit | flags = flags \| ERROR_BIT |
| && | 'E' lógico | if (len == 512) && (b == c) |
| \|\| | 'OU' lógico | if (len == 512) \|\| (b == c) |
| = | Igual a | entry_index = 0 |
| += | Somar e igualar | entry_index += 1 |
| -= | Subtrair e igualar | entry_index -= 1 |
| *= | Multiplicar e igualar | entry_index *= entry_length |
| /= | Dividir e igualar | entry_total /= entry_length |
| %= | Igualar ao módulo | entry_index %= 8 |
| <<= | Mover p/ esquerda e igual | flags <<= 3 |
| >>= | Mover p/ direita e igual | flags >>= 3 |
| &= | 'E' lógico e igual | flags &= ERROR_FLAG |
| \|= | 'OU' bit a bit e igual | flags \|= ERROR_FLAG |
| ^= | OU exclusivo bit a bit e igual | flags^= ERROR_FLAG |
| ++ | Incrementar uma unidade | i ++ |
| - - | Decrementar uma unidade | i -- |

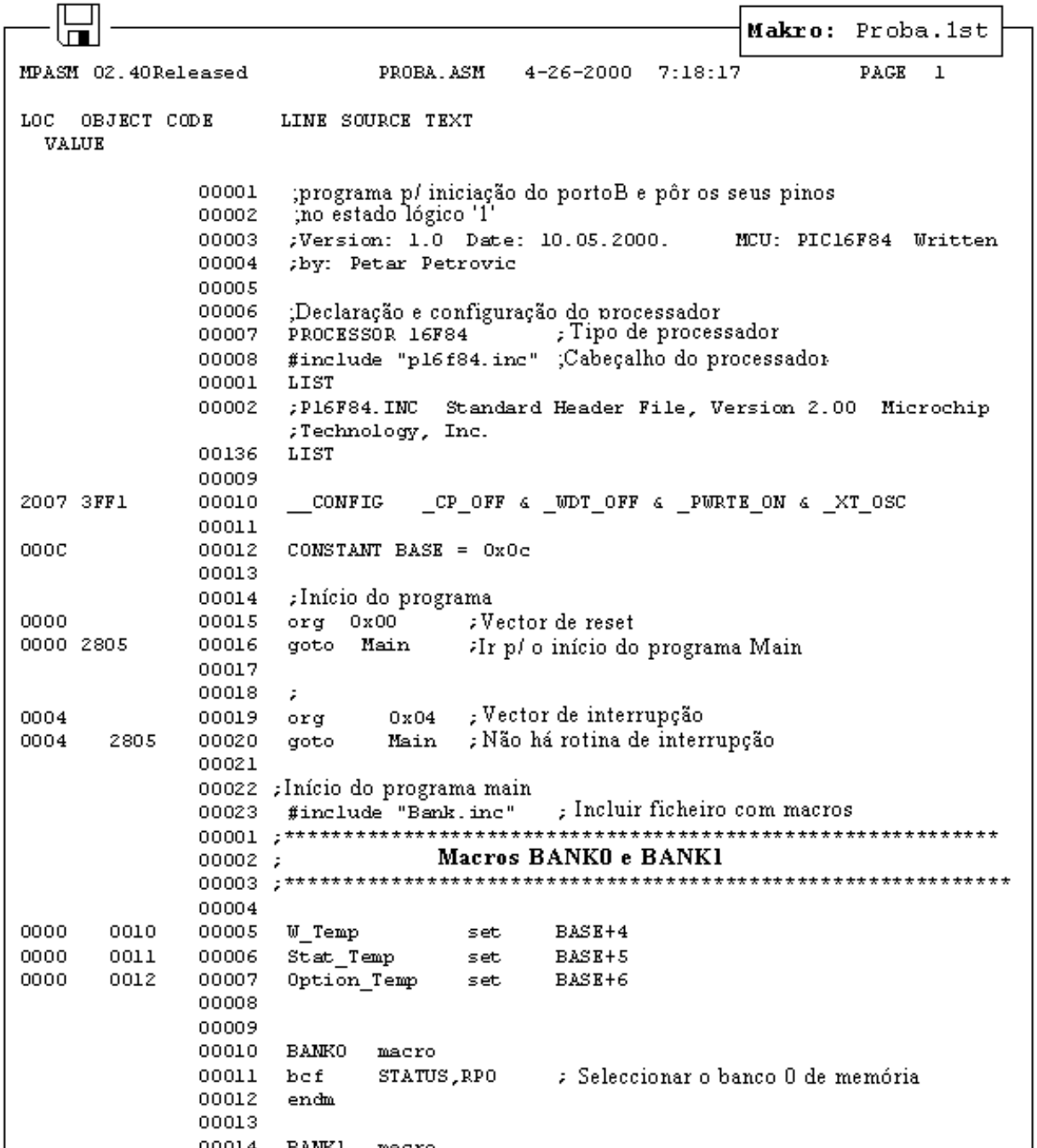
## Ficheiros criados ao compilar um programa

Os ficheiros resultantes da tradução de um programa escrito em linguagem assembly são os seguintes:

• Ficheiro de execução (nome_do_programa.hex)
• Ficheiro de erros no programa (nome_do_programa.err)
• Ficheiro de listagem (nome_do_programa.lst)

O primeiro ficheiro contém o programa traduzido e que vai ser introduzido no microcontrolador quando este é programado. O conteúdo deste ficheiro não dá grande informação ao programador, por isso, não o iremos mais abordar.
O segundo ficheiro contém erros possíveis que foram cometidos no processo de escrita e que foram notificados pelo assembler durante a tradução. Estes erros também são mencionados no ficheiro de listagem “list”. No entanto é preferível utilizar este ficheiro de erros “err”, em casos em que o ficheiro “lst” é muito grande e, portanto, difícil de consultar.
O terceiro ficheiro é o mais útil para o programador. Contém muita informação tal como o posicionamento das instruções e variáveis na memória e a sinalização dos erros.

A seguir, apresenta-se o ficheiro ‘list’ do programa deste capítulo. No início de cada página, encontra-se informação acerca do nome do ficheiro, data em que foi criado e número de página. A primeira coluna, contém o endereço da memória de programa, onde a instrução mencionada nessa linha, é colocada. A segunda coluna, contém os valores de quaisquer símbolos definidos com as directivas: SET, EQU, VARIABLE, CONSTANT ou CBLOCK. A terceira coluna, tem, o código da instrução que o PIC irá executar. A quarta coluna contém instruções assembler e comentários do programador. Possíveis erros são mencionados entre as linhas, a seguir à linha em que o erro ocorreu.



**Makro:** Proba.lst

```
MPASM 02.40Released           PROBA.ASM   4-26-2000  7:18:17          PAGE  1

LOC  OBJECT CODE     LINE SOURCE TEXT
  VALUE

                     00001   ;programa p/ iniciação do portoB e pôr os seus pinos
                     00002   ;no estado lógico '1'
                     00003   ;Version: 1.0  Date: 10.05.2000.      MCU: PIC16F84  Written
                     00004   ;by:  Petar Petrovic
                     00005
                     00006   ;Declaração e configuração do processador
                     00007   PROCESSOR 16F84         ;Tipo de processador
                     00008   #include "p16f84.inc"  ;Cabeçalho do processador
                     00001   LIST
                     00002   ;P16F84.INC  Standard Header File, Version 2.00  Microchip
                             ;Technology, Inc.
                     00136   LIST
                     00009
2007 3FF1            00010   __CONFIG    _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC
                     00011
000C                 00012   CONSTANT BASE = 0x0c
                     00013
                     00014   ;Início do programa
0000                 00015   org  0x00       ;Vector de reset
0000 2805            00016   goto   Main     ;Ir p/ o início do programa Main
                     00017
                     00018   ;
0004                 00019   org      0x04   ;Vector de interrupção
0004    2805         00020   goto     Main   ;Não há rotina de interrupção
                     00021
                     00022 ;Início do programa main
                     00023  #include "Bank.inc"    ; Incluir ficheiro com macros
                     00001 ;*****************************************************
                     00002 ;             Macros BANK0 e BANK1
                     00003 ;*****************************************************
                     00004
0000    0010         00005  W_Temp          set     BASE+4
0000    0011         00006  Stat_Temp       set     BASE+5
0000    0012         00007  Option_Temp     set     BASE+6
                     00008
                     00009
                     00010  BANK0   macro
                     00011  bcf     STATUS,RP0      ; Seleccionar o banco 0 de memória
                     00012  endm
                     00013
                     00014  BANK1   macro
```

```
                  00010   BANK0   macro
                  00011   bcf     STATUS,RP0      ; Seleccionar o banco 0 de memória
                  00012   endm
                  00013
                  00014   BANK1   macro
                  00015   bsf     STATUS,RP0      ; Seleccionar o banco 1 de memória
                  00016   endm
                  00017
0005              00024   Main
                  00025   BANK1                   ; Seleccionar o banco 1 de memória
0005    1683      M       bsf     STATUS,RP0      ; Seleccionar o banco 1 de memória
0006    3000      00026   movlw    0x00
Message[302]: Register in operand not in bank 0. Ensure that bank bits are
correct.
0007    0086      00027   movwf   TRISB           ;Pinos do porto B de saída
                  00028
                  00029   BANK0                   ;Seleccionar banco 0 de memória
0008    1283      M       bcf     STATUS,RP0      ;Seleccionar banco 0 de memória
0009    30FF      00030   movlw   0xFF
000A    0086      00031   movwf   PORTB           ;Tudo a '1' lógico no porto B
                  00032
000B    280B      00033   Loop    goto    Loop    ;o programa permanece no loop
                  00034
                  00035   END             ;Para assinalar o fim do programa

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

0000 : X---XXXXXXXX---- ---------------- ---------------- ----------------
2000 : -------X-------- ---------------- ---------------- ----------------

All other memory blocks unused.

Program Memory Words Used:    9
Program Memory Words Free:    1015

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

No fim do ficheiro de listagem, é apresentada uma tabela dos símbolos usados no programa. Uma característica útil do ficheiro 'list' é a apresentação de um mapa da memória utilizada. Mesmo no fim, existe uma estatística dos erros, bem como a indicação da memória de programa utilizada e da disponível.

## Macros

As macros são elementos muito úteis em linguagem assembly. Uma macro pode ser descrita em poucas palavras como "um grupo de instruções definido pelo utilizador que é acrescentado ao programa pelo assembler, sempre que a macro for invocada". É possível escrever um programa sem usar macros. Mas, se as utilizarmos, o programa torna-se muito mais legível, especialmente se estiverem vários programadores a trabalhar no mesmo programa. As macros têm afinidades com as funções nas linguagens de alto nível.

**Como as escrever:**

<rótulo> macro
[<argumento1>,<argumento2>,.....,<argumentoN>]
.....
.....
endm

Pelo modo como são escritas, vemos que as macros podem aceitar argumentos, o que também é muito útil em programação.
Quando o argumento é invocado no interior de uma macro, ele vai ser substituído pelo valor <argumentoN>.

**Exemplo:**

```
ON_PORTB        macro ARG1
                BANK0           ;Seleccionar banco 0 de memória
                movlw ARG1      ;valor do argumento ARG1
                                ;guardado no registo de trabalho
                movwf PORTB     ;valor do argumento ARG1
                                ;guardado no Porto B
                endm            ;fim de macro
```

O exemplo de cima, mostra uma macro cujo propósito é enviar para o porto B, o argumento ARG1, definido quando a macro foi invocada. Para a utilizarmos num programa, basta escrever uma única linha: ON_PORTB 0xFF e, assim, colocamos o valor 0xFF no porto B. Para utilizar uma macro no programa, é necessário incluir o ficheiro macro no programa principal, por intermédio da instrução #include “nome_da_macro.inc”. O conteúdo da macro é automaticamente copiado para o local em que esta macro está escrita. Isto pode ver-se melhor no ficheiro ‘lst’ visto atrás, onde a macro é copiada por baixo da linha #include “bank.inc”.

# CAPÍTULO 5

## MPLAB



## Introdução

O MPLAB é um pacote de programas que correm no Windows e que tornam mais fácil escrever ou desenvolver um programa. Pode descrever-se ainda melhor como sendo um ambiente de desenvolvimento para uma linguagem de programação standard e destinado a correr num computador pessoal (PC). Anteriormente, as operações incidiam sobre uma linha de instrução e contemplavam um grande número de parâmetros, até que se introduziu o IDE "Integrated Development Environment" (Ambiente Integrado de Desenvolvimento) e as operações tornaram-se mais fáceis, usando o MPLAB. Mesmo agora, as preferências das pessoas divergem e alguns programadores preferem ainda os editores standard e os intérpretes linha a linha. Em qualquer dos casos, o programa escrito é legível e uma ajuda bem documentada está disponível.

## 5.1 Instalando o programa - MPLAB


O MPLAB compreende várias partes:

- Agrupamento de todos os ficheiros do mesmo projecto, num único projecto (Project Manager)
- Escrever e processar um programa (Editor de Texto)
- Simular o funcionamento no microcontrolador do programa que se acabou de escrever (Simulador)

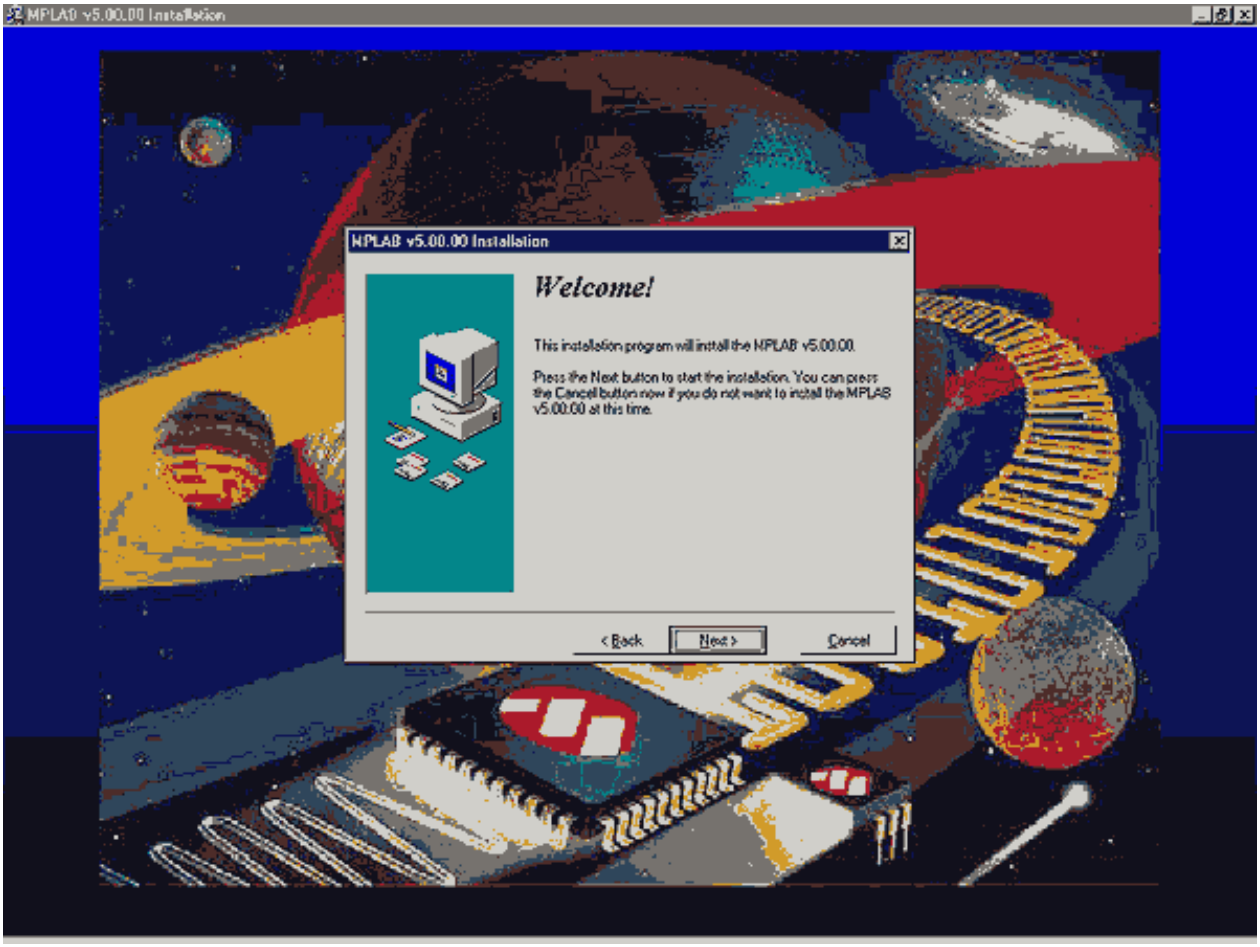
Além destes, existem sistemas de suporte para os produtos da Microchip, tais como o PICSTART Plus e ICD (In Circuit Debugger - Detecção de erros com o microcontrolador a funcionar). Este livro não aborda estes dois dispositivos que são opcionais.

Os requisitos mínimos para um computador que possa correr o MPLAB, são:
• Computado PC com microprocessador 486 ou superior
• Microsoft Windows 3.1x, Windows 95 ou versões mais recente do sistema operativo Windows.
• Placa gráfica VGA
• 8MB de memória (32MB recomendados)
• 20MB de espaço no disco duro
• Rato

Antes de iniciarmos o MPLAB, temos primeiro que o instalar. A instalação é o processo de copiar os ficheiros do MPLAB para o disco duro do computador, a partir do CD respectivo. Existe uma opção em cada nova janela que permite regressar à anterior. Assim, os erros não constituem problema e o trabalho de instalação, torna-se mais fácil. Este modo de instalação é comum à maioria dos programas Windows. Primeiro, aparece uma janela de boas vindas, a seguir pode-se escolher entre as opções indicadas e, no fim do processo, obtém-se uma mensagem que informa de que o programa está instalado e pronto a funcionar.

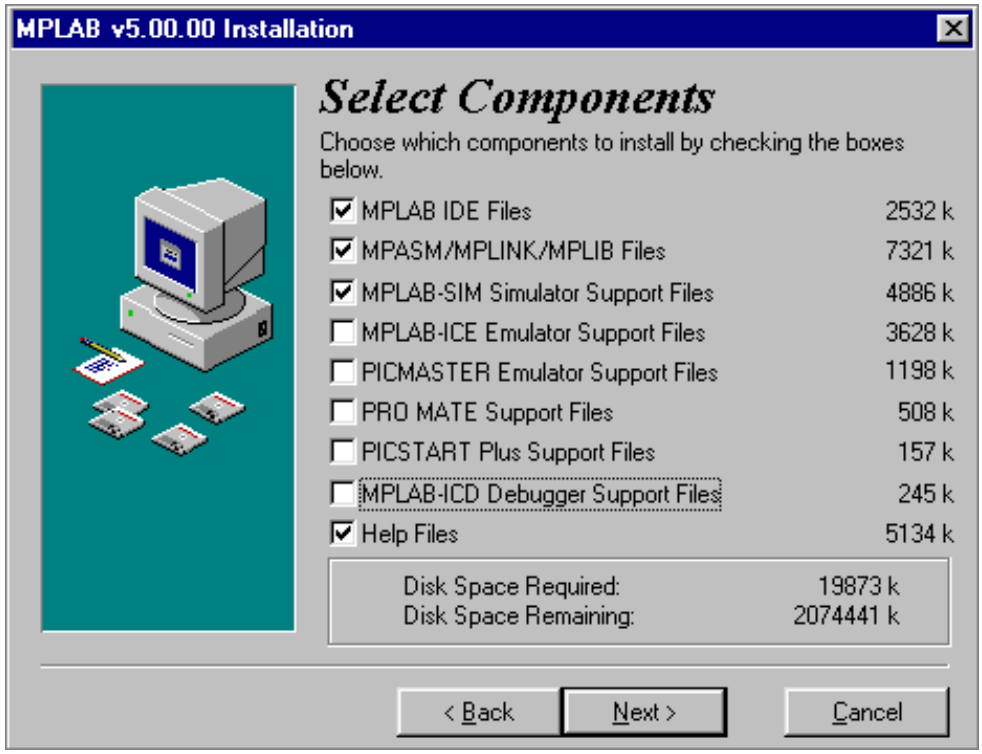
Passos para instalar o MPLAB:
1. Correr o Windows da Microsoft
2. Introduzir o CD da Microchip na drive de CD.
3. ‘Clicar’ no botão ‘INICIAR’, situado no lado esquerdo do écran ao fundo e escolher a opção Executar...
4. Clicar em ‘Procurar’ e seleccionar a drive de CD do seu computador
5. Descobrir o directório MPLAB na CD ROM
6. Clicar em SETUP.EXE e a seguir em OK.
7. Clicar novamente em OK na sua janela de Executar

A instalação propriamente dita, começa depois destes sete passos. As figuras que se seguem explicam o significado de certas etapas dessa instalação.



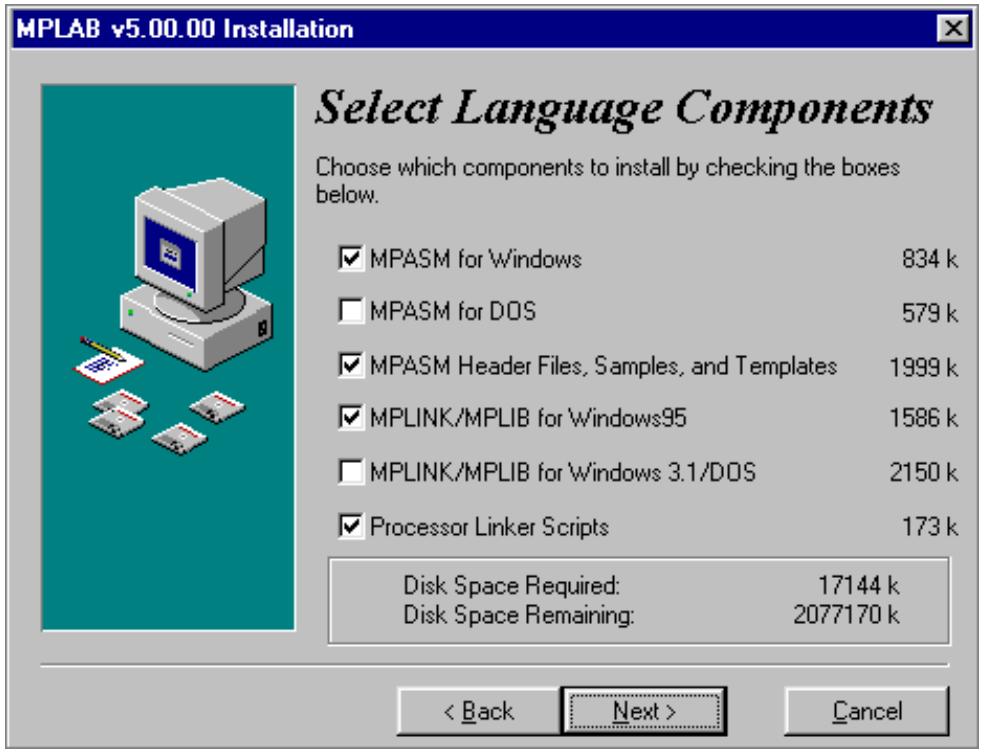

**Écran de boas-vindas no início da instalação do MPLAB**

Logo no início, é necessário seleccionar quais os componentes do MPLAB com que vamos trabalhar. Como, em princípio, não dispomos dos componentes de hardware originais da Microchip tais como programadores ou emuladores, as únicas coisas que vamos instalar é o ambiente MPLAB, Assembler, Simulador e instruções.



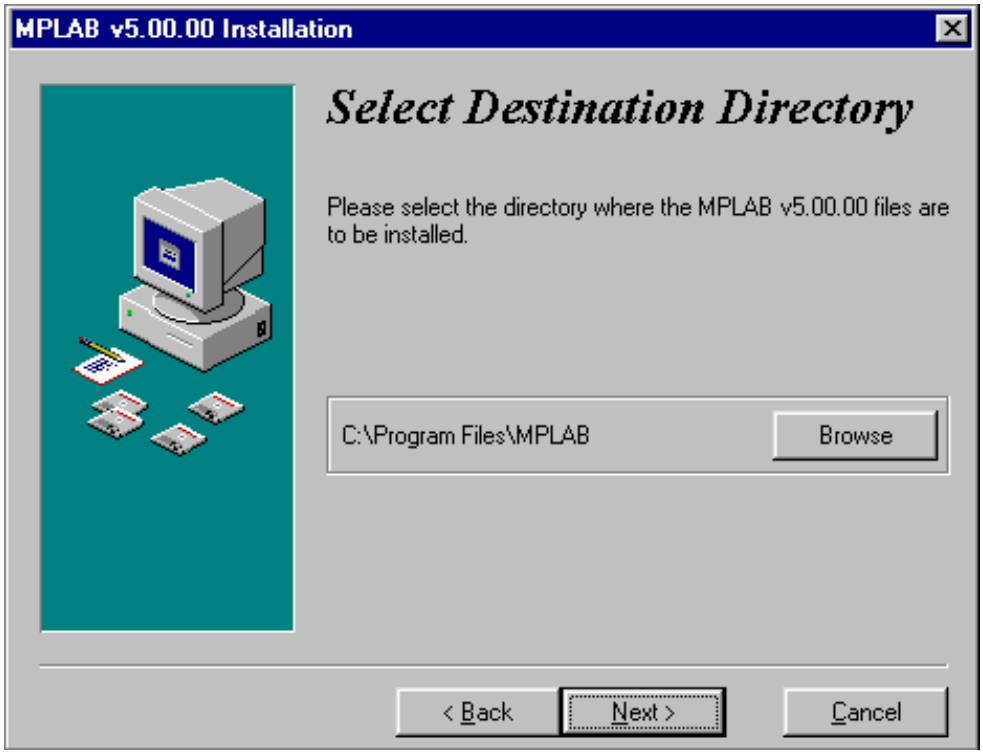
**Seleccionando os componentes do ambiente de desenvolvimento MPLAB**

Como é suposto irmos trabalhar com o Windows 95 (ou um sistema operativo ainda mais moderno), tudo aquilo que diga respeito ao sistema operativo DOS, não deve ser contemplado ao fazer a selecção da linguagem assembler. Contudo, se preferir continuar a trabalhar em DOS, então precisa de desmarcar todas as opções relacionadas com o Windows e seleccionar os componentes apropriados para o DOS.



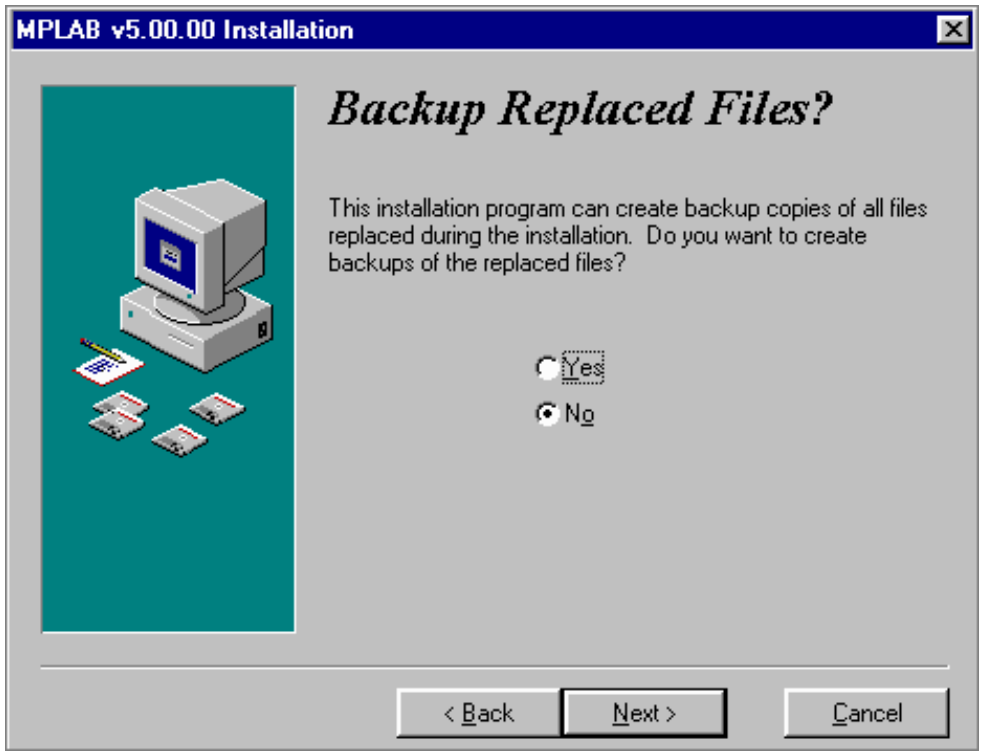

**Seleccionando o Assembler e o sistema operativo**

Como qualquer outro programa, o MPLAB deve ser instalado num directório. O directório escolhido, pode ser qualquer um no disco duro do computador. Se não tiver razões fortes para não o fazer, deve aceitar a escolha indicada por defeito.




**Escolhendo o directório em que o MPLAB vai ficar instalado**

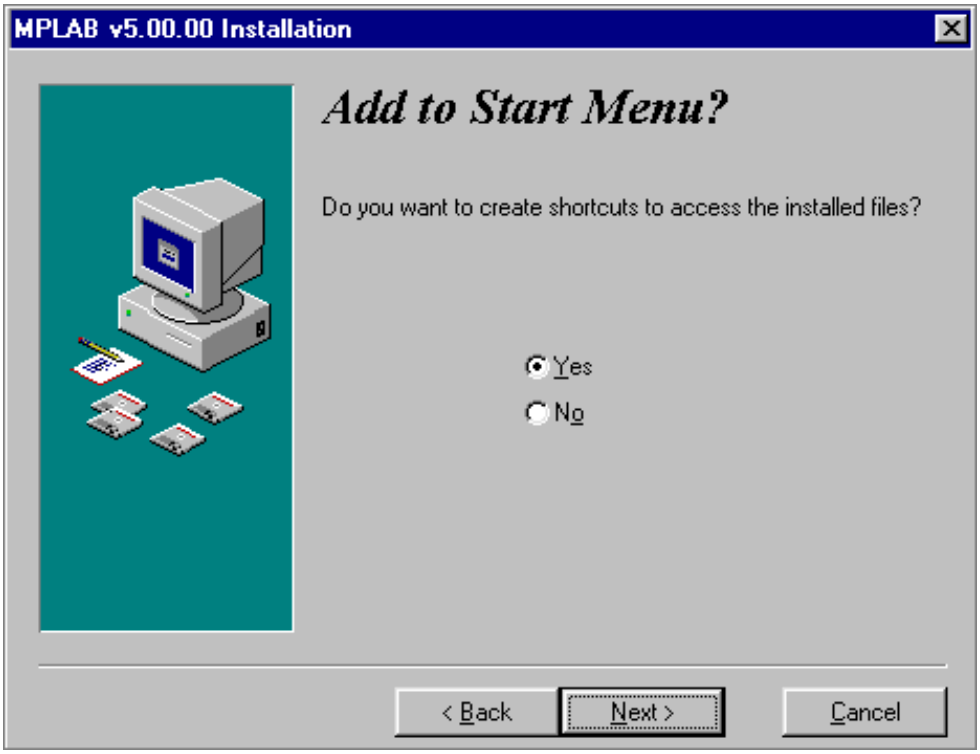
Os utilizadores que já tenham o MPLAB instalado (versão mais antiga que esta), necessitam da opção que se segue. O propósito desta opção é salvaguardar cópias de todos os ficheiros que foram modificados durante o processo de mudança da versão antiga para a versão mais moderna do MPLAB. No nosso caso, deixaremos a opção NO seleccionada, porque se presume que estamos a fazer a primeira instalação do MPLAB no nosso computador.




**Opções para os utilizadores que estão a instalar uma nova versão do MPLAB por cima de outra versão instalada, mais antiga**

O menu Iniciar (START) contém um conjunto de ponteiros para programas e é seleccionado clicando na opção INICIAR ao fundo, no canto esquerdo do écran. Para que o MPLAB também possa ser iniciado a partir daqui, nós

precisamos de deixar esta opção tal ela se nos apresenta.




**Introduzindo o MPLAB no menu iniciar**

A janela que se mostra a seguir, tem a ver com uma parte do MPLAB em cuja explicação não necessitamos de entrar em detalhes. Seleccionando um directório especial, o MPLAB guarda todos os ficheiros relacionados com o 'linker', num directório separado.

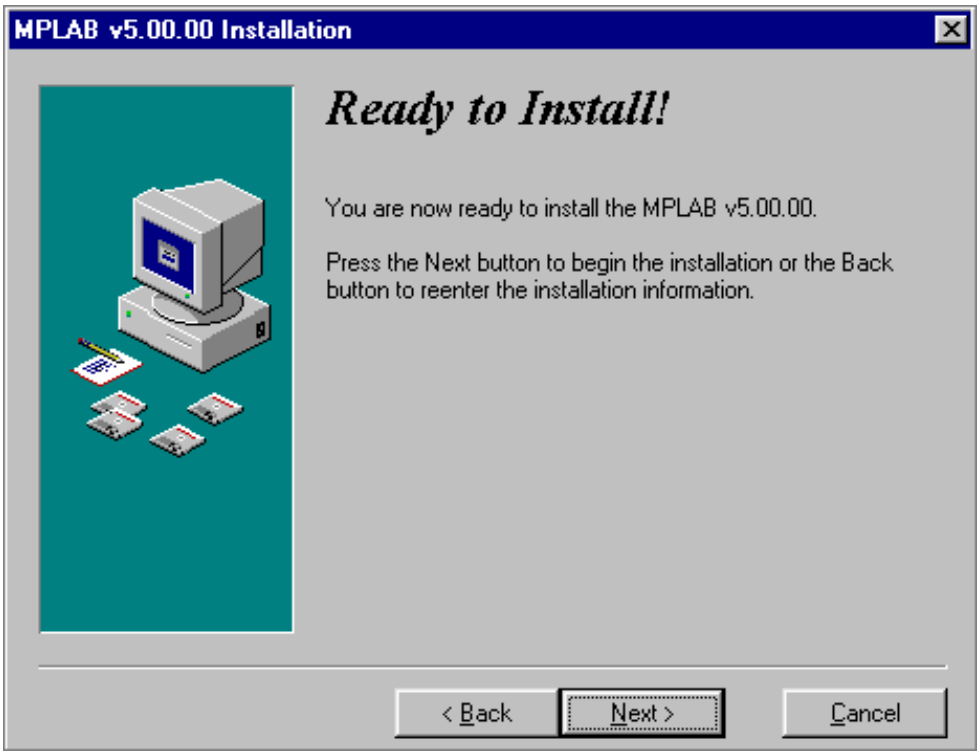

**Definição do directório dos ficheiros linker**

Geralmente, todos os programas que correm no Windows, têm os ficheiros do sistema guardados no directório

Windows. Depois de múltiplas instalações, o directório Windows torna-se demasiado grande e povoado. Assim, alguns programas permitem que os seus ficheiros do sistema fiquem guardados nos mesmos directórios em que estão os programas. O MPLAB é um exemplo disto e, por isso, deve ser seleccionada a opção do fundo.


**Selecção de um directório para os ficheiros do sistema**

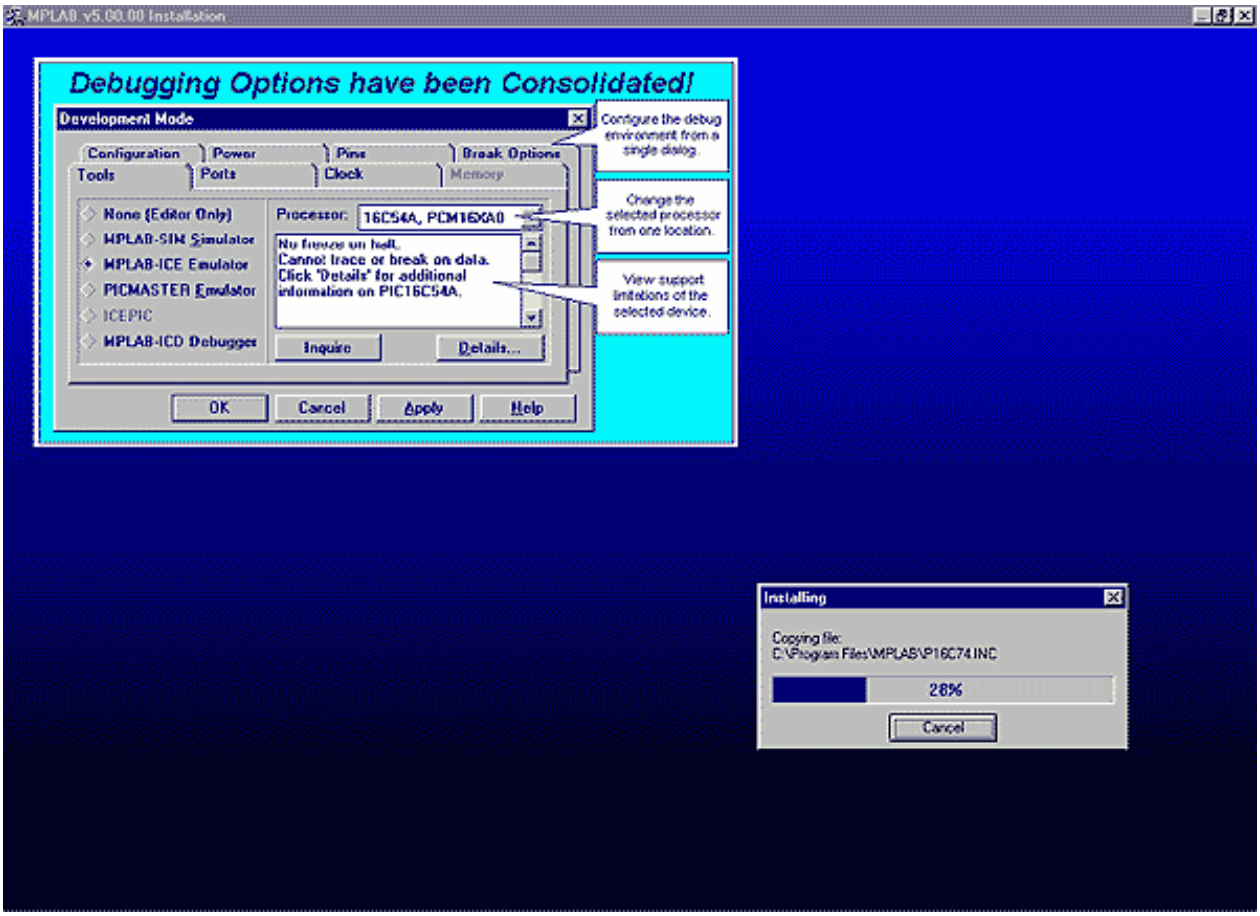
Depois de realizados estes passos, a instalação inicia-se ao clicar em 'Next'.




**Écran antes da instalação**

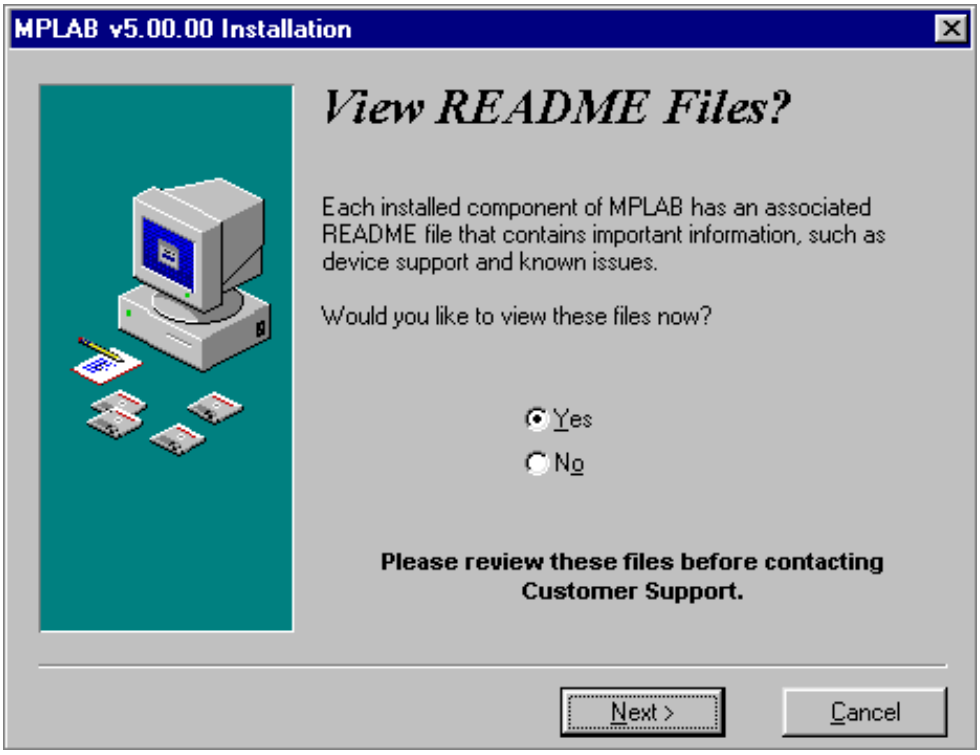
A instalação não demora muito tempo e o processo de copiar os ficheiros pode ser visualizado numa pequena janela no lado direito do

écran.




**A instalação a decorrer**

Depois de concluída a instalação, aparecem dois écrans de diálogo, um que menciona as últimas informações e correcções relativas ao programa e o outro que é um écran de boas vindas.
Se os ficheiros de texto (Readme.txt) forem abertos, é preciso, fechá-los a seguir.

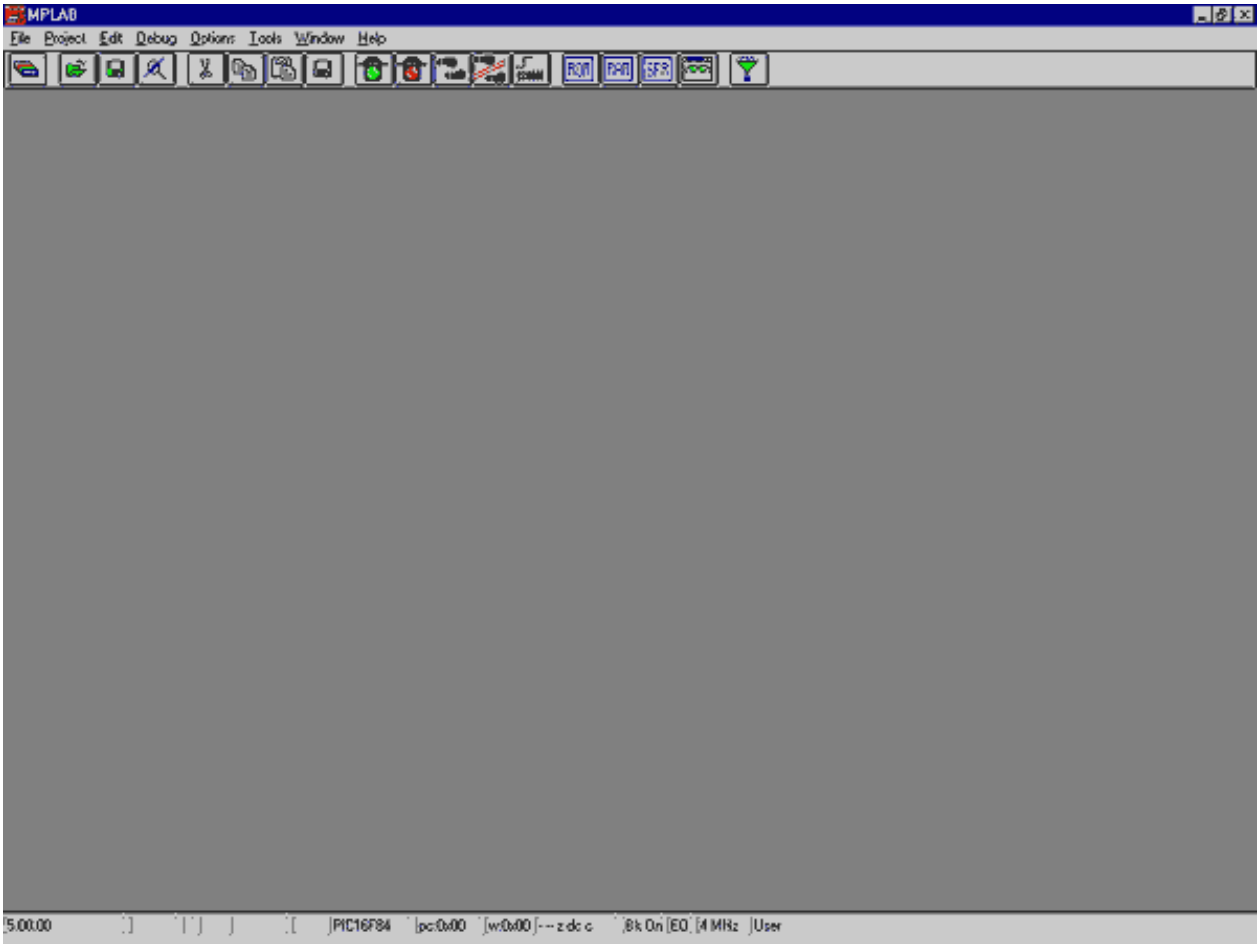



**Informações de “último minuto”, respeitantes às versões do programa e a correcções**

Depois de clicar em ‘Finish’, a instalação do MPLAB está terminada.

## 5.2 MPLAB

Quando terminamos o processo de instalação, aparece-nos no écran o programa propriamente dito. Como pode ver-se, o aspecto do MPLAB é o mesmo da maioria dos programas Windows. Perto da área de trabalho existe um “menu” (faixa azul em cima, com as opções File, Edit, etc.), “toolbar” (barra com figuras que preenchem pequenos quadrados) e a linha de status no fundo da janela. Assim, pretende-se seguir uma regra no Windows que é tornar também acessíveis por baixo do menu, as opções usadas mais frequentemente no programa,. Deste modo, é possível acedê-las de um modo mais fácil e tornar o nosso trabalho mais rápido. Ou seja, aquilo que está disponível na barra de ferramentas, também está disponível no menu.




**O écran depois de o MPLAB ser iniciado**

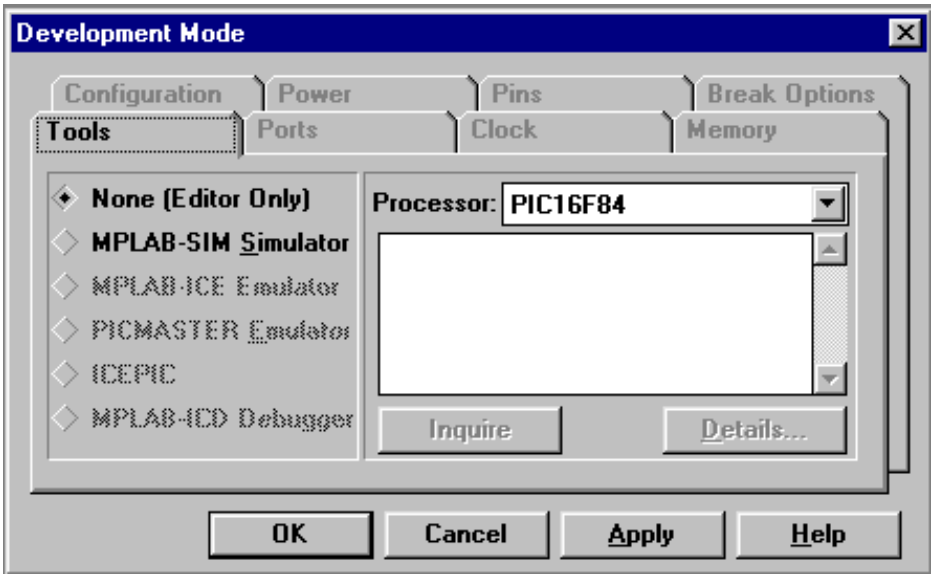
O propósito deste capítulo é familiarizá-lo com o ambiente de desenvolvimento MPLAB e com elementos básicos do MPLAB, tais como:

Escolher um modo de desenvolvimento
Designar um projecto
Designar um ficheiro para o programa original
Escrever um programa elementar na linguagem de programação assembler
Traduzir um programa para linguagem máquina
Iniciar o programa
Abrir uma nova janela de simulação
Abrir uma nova janela para as variáveis cujos valores queremos observar (watch window)
Guardar a janela para as variáveis cujos valores queremos observar (janela anterior)
Definir breakpoints no simulador (pontos de paragem)

A preparação de um programa para ser lido num microcontrolador compreende várias etapas básicas:

## 5.3 Escolhendo o modo de desenvolvimento

Para que o MPLAB possa saber que ferramentas vão ser usadas na execução do programa que se escreveu, é necessário definir o modo de desenvolvimento. No nosso caso, nós precisamos de preparar o simulador como preparamos uma ferramenta que vamos usar. Clicando em OPTIONS---> DEVELOPMENT MODE, uma nova janela idêntica à que se mostra na figura em baixo, irá aparecer:

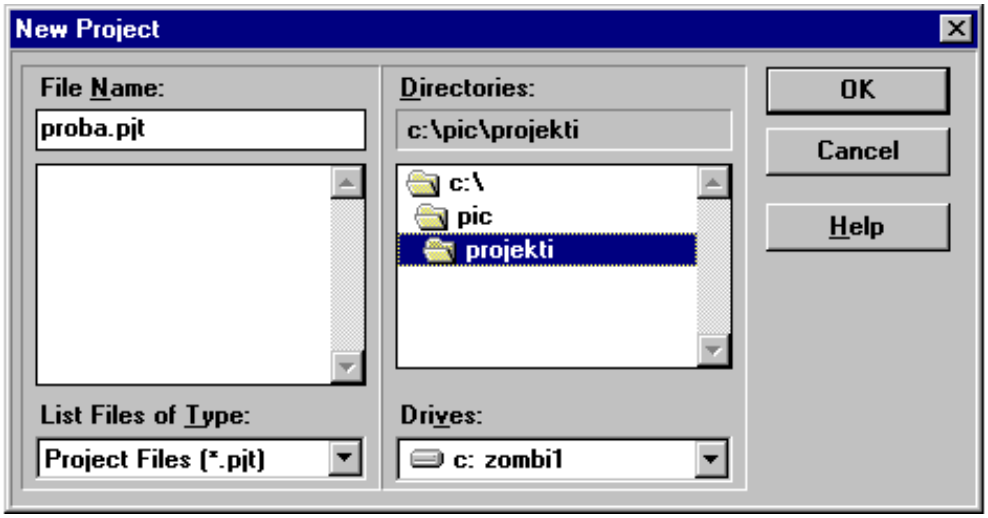



**Definindo um modo de desenvolvimento**

Nós devemos seleccionar a opção 'MPLAB-SIM Simulator', porque é neste ambiente que o nosso programa vai ser experimentado. Além desta opção, está também disponível a opção 'Editor Only' (somente editor). Esta última opção só é usada, se o que desejamos é apenas escrever o programa e usar um programador para transferir um 'ficheiro hex' para o microcontrolador. A selecção do modelo de microcontrolador é feita no lado direito. Como o livro é baseado no PIC16F84, é este o modelo de microcontrolador que deve ser seleccionado. Normalmente, quando começamos a trabalhar com microcontroladores, usamos um simulador. Depois, à medida que o nível dos nossos conhecimentos sobe, podemos escrever o programa no microcontrolador, logo após a sua tradução. O nosso conselho, é que você use sempre o simulador. Embora possa parecer que, assim, o programa demora mais tempo a implementar, no fim vai ver que vale a pena.

## 5.4 Implementando um projecto

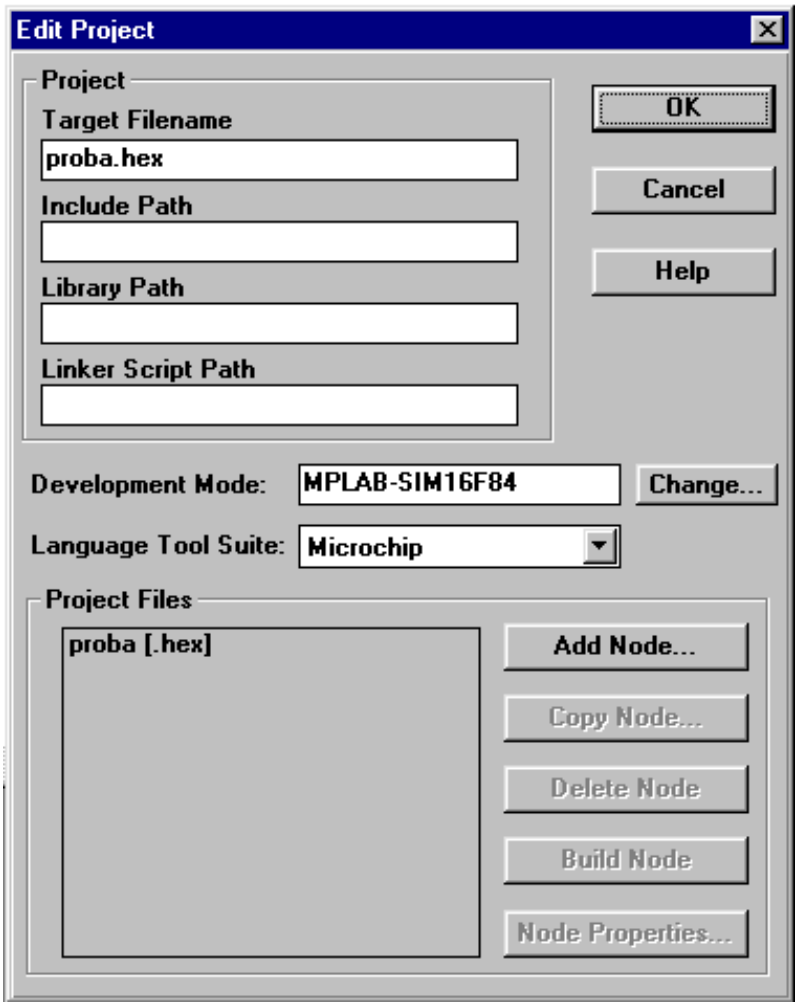
Para começar a escrever um programa é preciso primeiro criar um projecto. Clicando em PROJECT --> NEW PROJECT você pode dar um nome ao seu projecto e guardá-lo num directório à sua escolha. Na figura em baixo, um projecto designado por 'test.pjt' está a ser criado e é guardado no directório c:\PIC\PROJEKTI\ . Escolheu-se este directório porque os autores têm este directório no seu computador. De um modo genérico, escolhe-se um directório que está contido noutro directório maior e cujo nome deve fazer lembrar os ficheiros que contém.




**Abrindo um novo projecto**

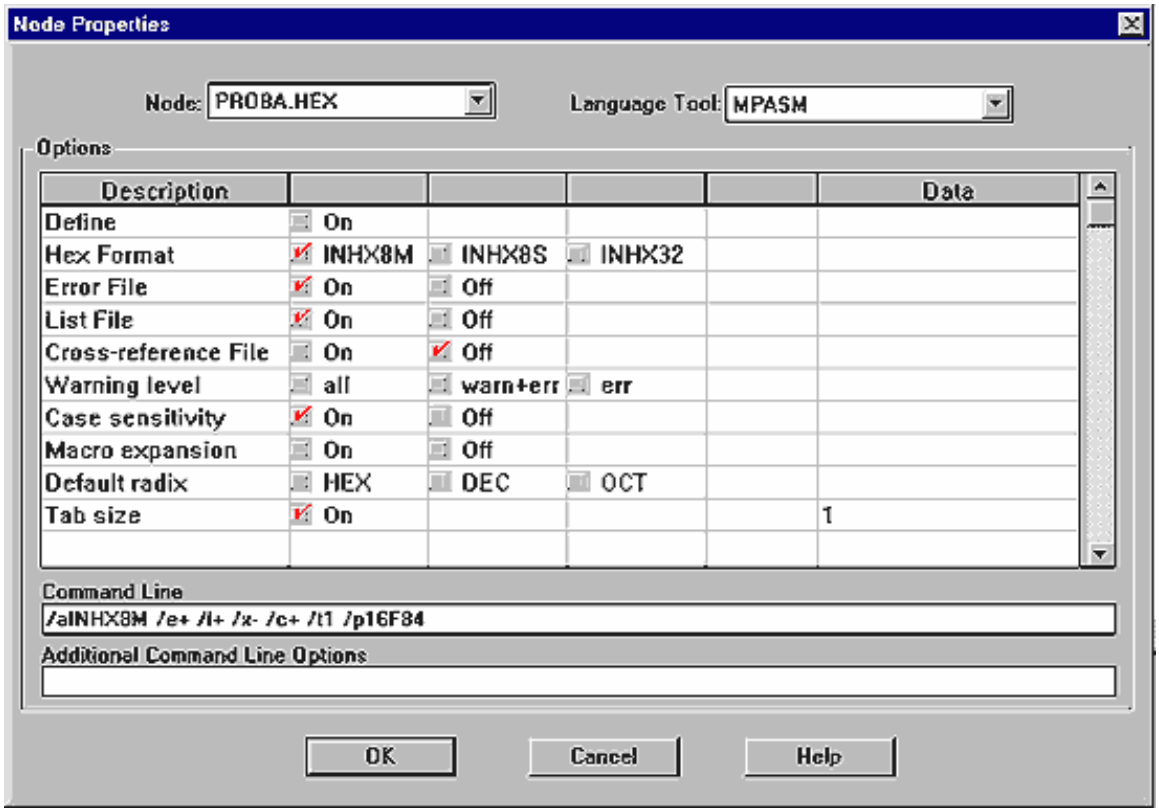
Depois de dar um nome ao projecto clique em OK. Veremos que aparece uma nova janela, idêntica à que se

mostra na figura seguinte.



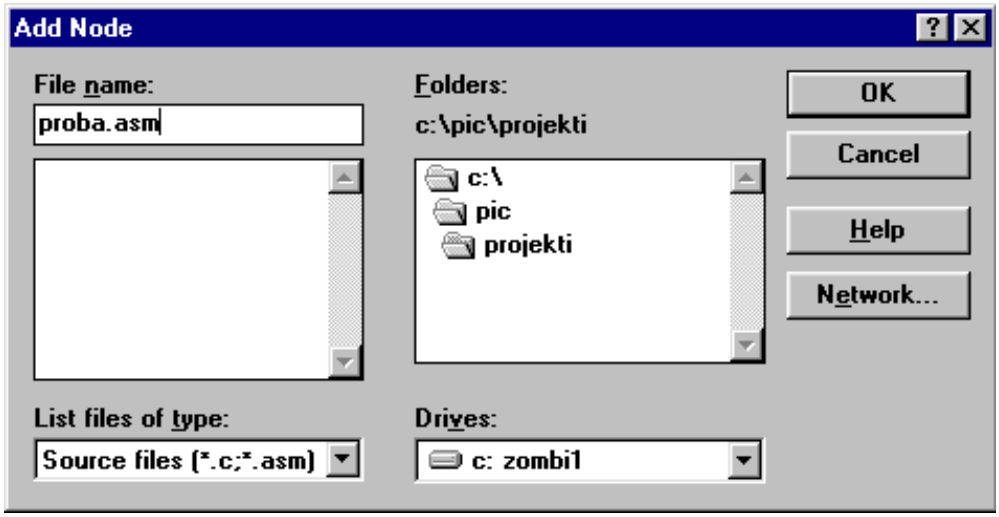

**Ajuste dos elementos do projecto**

Com o rato, clique em "proba [.hex]", o que activa a opção 'Node properties', ao fundo no lado direito. Clicando esta opção, obtém-se a janela seguinte.



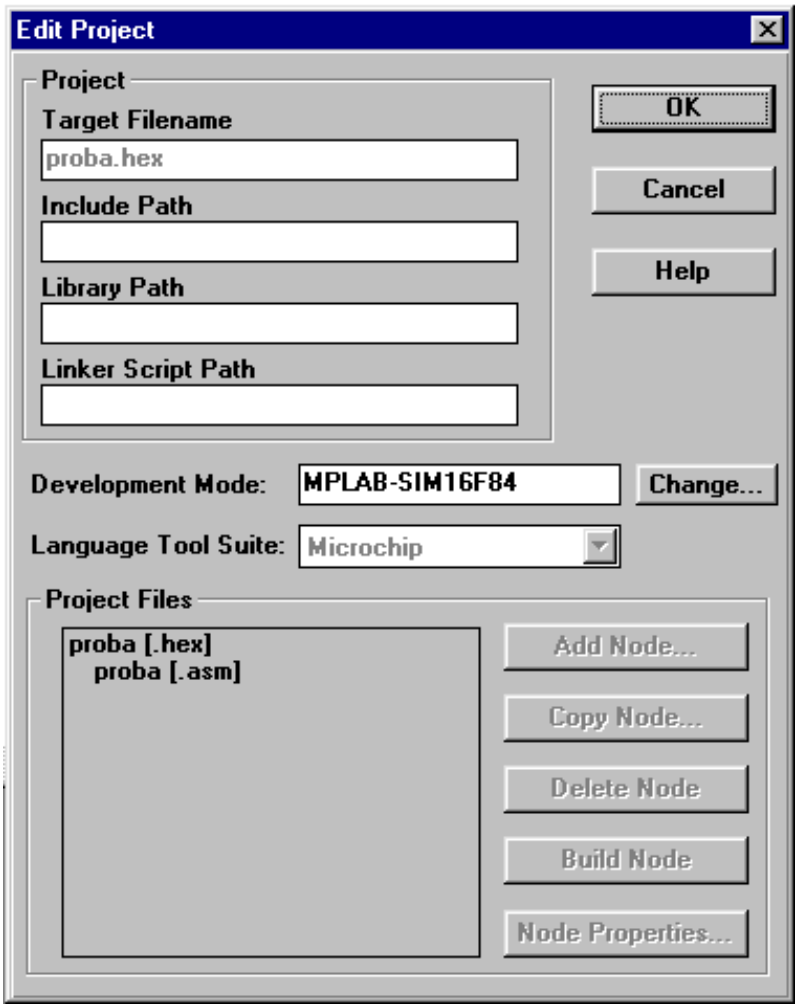

**Definindo os parâmetros do assembler MPASM**

Na figura pode verificar-se que existem muitos parâmetros diferentes. Cada um deles, corresponde a um termo na “linha de comandos”. Como memorizar estes parâmetros é bastante desconfortável ou mesmo proibitivo para principiantes, foi introduzida possibilidade de um ajuste feito graficamente. Observando a figura, verifica-se rapidamente quais as opções que estão seleccionadas. Clicando em OK, voltamos à janela anterior onde "Add node" é agora uma opção activa. Clicando nela, obtemos a seguinte janela onde vamos dar o um nome ao nosso programa assembler. Vamos chamar-lhe "Proba.asm", e vai ser o nosso primeiro programa em MPLAB.




**Abrindo um novo projecto**

Clicando em OK, voltamos à janela de inicial onde vemos adicionado um ficheiro assembler.



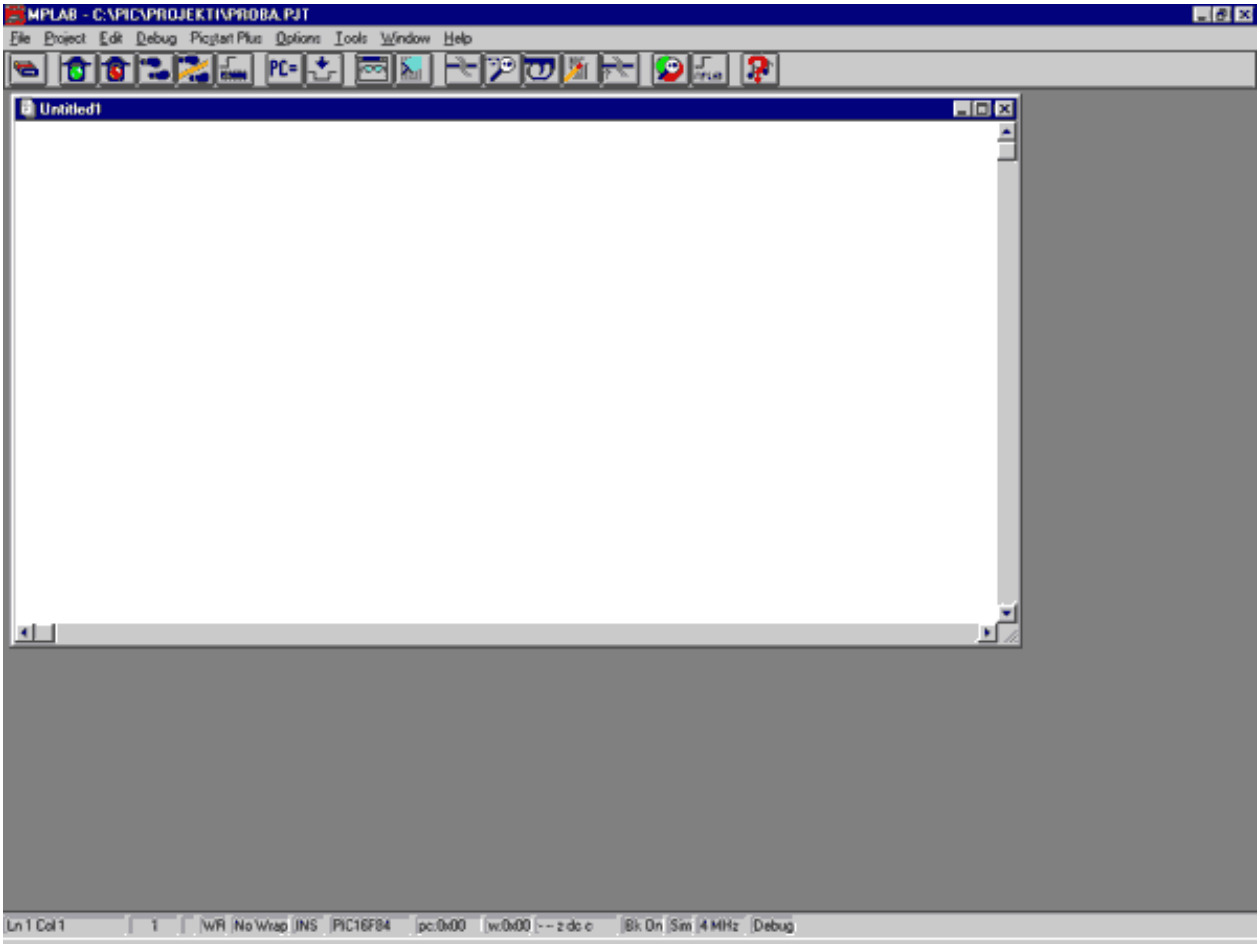

**Um ficheiro assembler foi adicionado**

Clicando em OK voltamos ao ambiente MPLAB.

## 5.5 Criando um novo ficheiro assembler (escrevendo um novo programa)

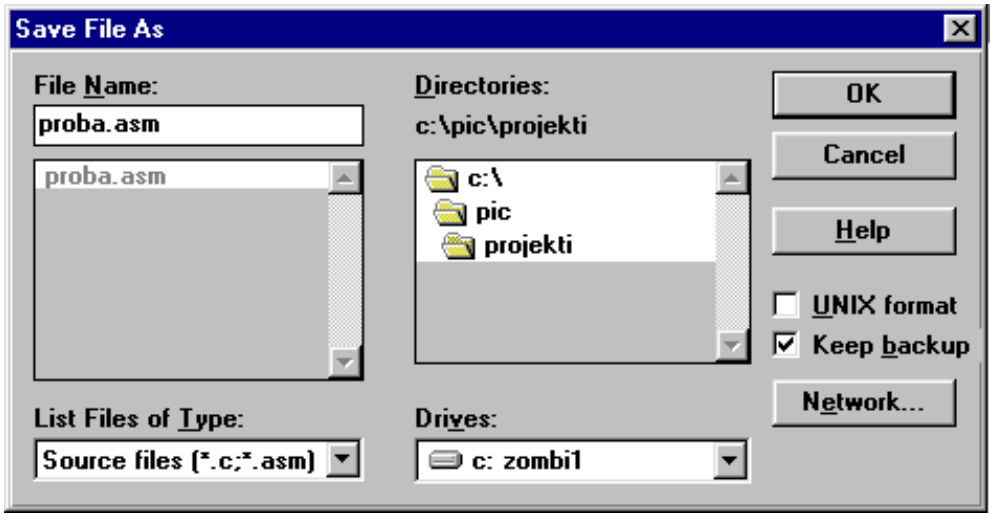
Depois de a parte de criação de "project", ter terminado, é altura de começarmos a escrever um programa. Por outras palavras, um novo ficheiro deve ser aberto e vai ser designado por "proba.asm". No nosso caso, o ficheiro tem que ser designado por "proba.asm" porque, em projectos constituídos por um único ficheiro (como é o caso do nosso), o nome do projecto e o nome do ficheiro fonte tem que ser o mesmo.

Para abrir um novo ficheiro, clica-se em FILE>NEW. Assim, obtemos uma janela de texto dentro do espaço de trabalho do MPLAB.




**Um novo ficheiro assembler foi aberto**

A nova janela representa o ficheiro onde o programa vai ser escrito. Como o nosso ficheiro assembler tem que ser designado por "proba.asm", vamos dar-lhe esse nome. A designação do programa faz-se (como em todos os programas Windows) clicando em FILE>SAVE AS. Deste modo, vamos obter uma janela análoga à que se mostra na figura seguinte.




**Dando um nome e guardando um novo ficheiro assembler**

Quando obtemos esta janela, precisamos de escrever 'proba.asm' por baixo de 'File name:' e clicar em OK. Depois de fazer isto, podemos ver o nome do ficheiro 'proba.asm', no cimo da nossa janela.

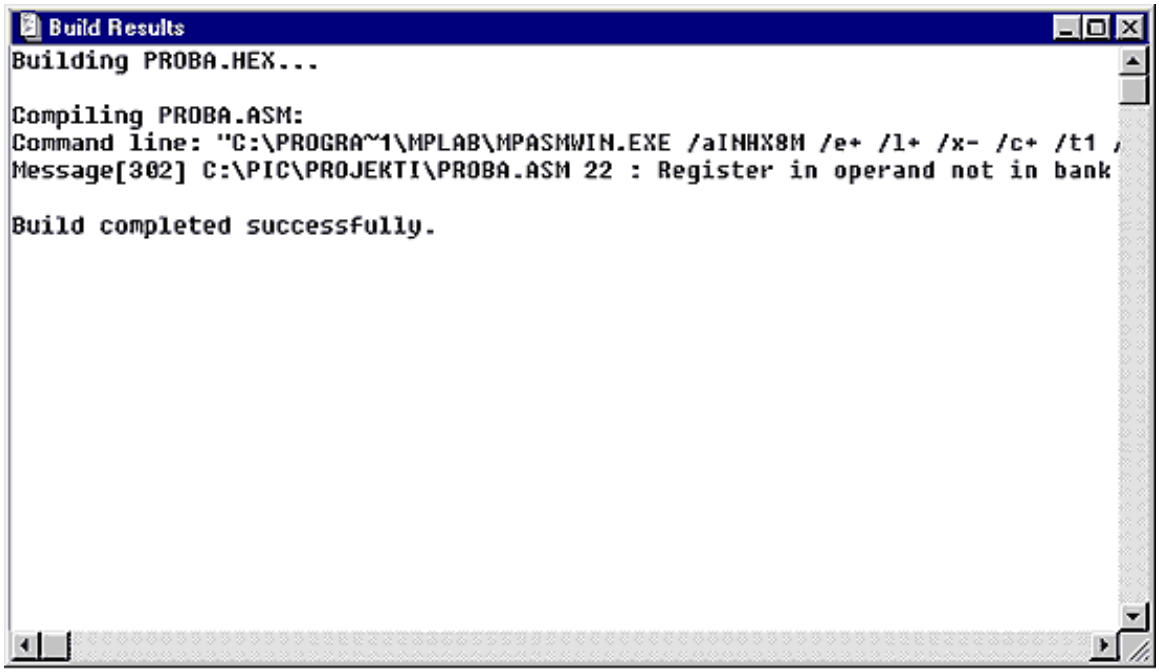
## 5.6 Escrevendo um programa

Só depois de completadas todas as operações precedentes é que nós podemos começar a escrever um programa. Como já dispomos de um programa simples que foi escrito na parte do livro "Programação em Linguagem Assembler", vamos usar esse programa aqui, também.

**Program:** Proba.asm

```
;Programa p/ iniciação do porto B e pôr os seus pinos a '1' lógico
;
;Version: 1.0 Date: 25.04.2000 MCU: PIC16F84 Written by: Petar
;Petrovic

;Declaração e configuração do processador

        PROCESSOR 16F84                     ; Tipo de processador
        #include          "p16f84.inc"      ;

        __CONFIG    _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC


        org     0x00            ; Vector de reset
        goto    Main            ; Ir p/ o início do programa Main
                                ;

        org     0x04            ; Vector de interrupção
        goto    Main            ; Não há rotina de interrupção


        #include "bank.inc" ; Macros BANK0 e BANK1

; Início do programa Main

Main
        BANK1                   ; Seleccionar Banco 1 de memória
        movlw   0x00
        movwf   TRISB           ; Pinos do porto B são saídas
        BANK0                   ; Seleccionar Banco 0 de memória

        movlw   0xFF
        movwf   PORTB           ; Tudo a '1' lógico no porto B

Loop    goto    Loop            ; O programa permanece no loop

        end                     ; Para assinalar o fim do programa
                                ;
```

Este programa tem que ser copiado numa janela que esteja aberta, ou copiado do disco ou tirado da página da internet da MikroElektronika usando os comandos copiar e colar. Quando o programa é copiado para a janela "proba.asm", nós podemos usar o comando PROJECT -> BUILD ALL (se não existirem erros) e, uma nova janela idêntica à representada na figura seguinte, vai aparecer.




**Janela com as mensagens que se sucedem à tradução do programa assembler**

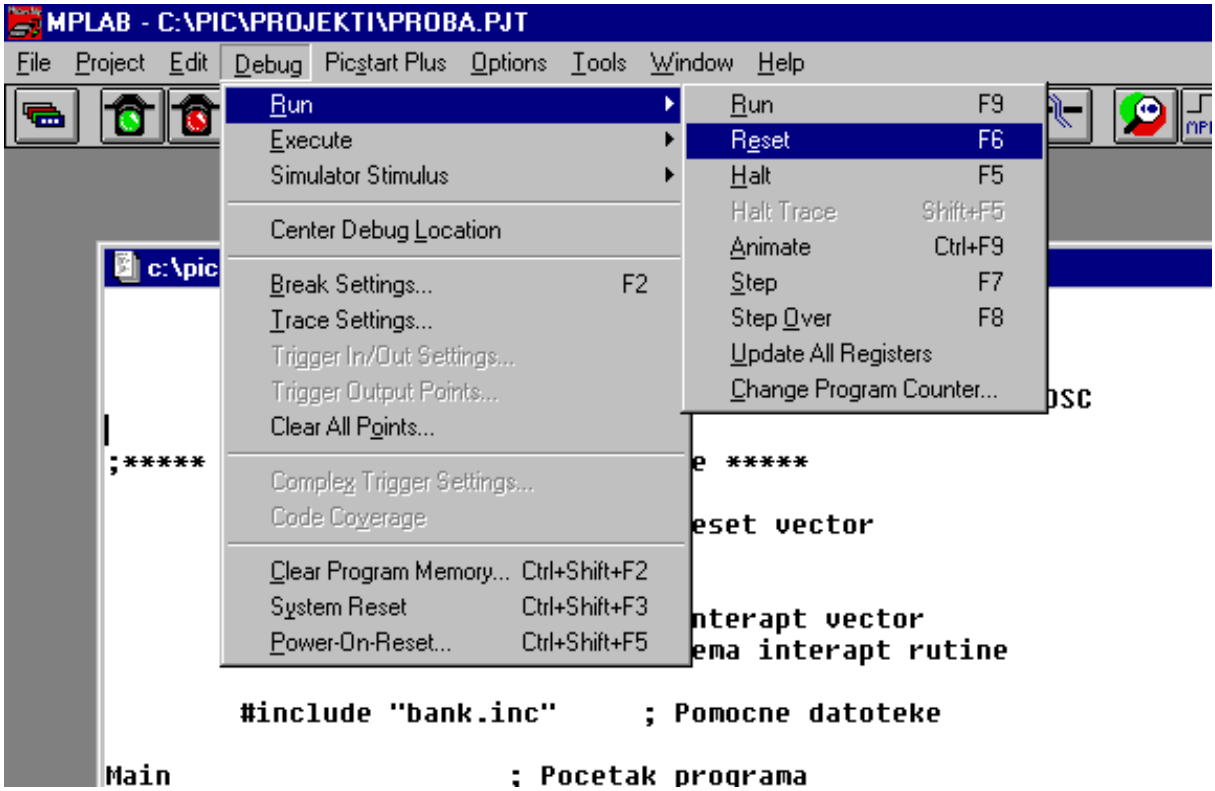
Na figura podemos observar que obtemos o ficheiro "proba.hex" como resultado do processo de tradução, que é usado o programa MPASMWIN para traduzir e que existe uma mensagem. De toda essa informação, a última frase que aparece na janela é a mais importante, já que nos diz se a tradução foi ou não bem sucedida. 'Build completed successfully' é uma mensagem que nos indica que a tradução foi feita com sucesso e que não apareceram erros.

No caso de serem indicados erros, precisamos de clicar duplamente nas mensagens de erros da janela 'Build Results'. Este acto, transfere-nos automaticamente para o programa assembler e para a linha em que o erro se encontra.

## 5.7 Simulador MPSIM

Simulador, é a parte do ambiente MPLAB que fornece uma melhor visão interna do modo como o microcontrolador trabalha. Através de um simulador nós podemos monitorizar os valores actuais das variáveis, os valores dos registos e os estados lógicos dos pinos dos portos. Para falar verdade, o simulador não dá exactamente os mesmos resultados em todos os programas. Se um programa for simples (como aquele que estamos a utilizar como exemplo), a simulação não é de grande importância, porque pôr todos os pinos do porto B a nível lógico um, não é uma tarefa difícil. Contudo, o simulador pode ser uma grande ajuda em programas mais complicados que incluem temporizadores, diferentes condições em que alguma coisa aconteça e outros requisitos semelhantes (especialmente com operações matemáticas). Simulação, como o próprio nome indica, "simula o funcionamento de um microcontrolador". Como o microcontrolador executa as instruções uma a uma, o simulador é concebido para executar o programa passo a passo (linha a linha), mostrando o que acontece aos dados dentro do microcontrolador. Quando o programa está completamente escrito, convém que o programador, em primeiro lugar, verifique o seu programa num simulador e, só a seguir o experimente numa situação real. Infelizmente, muitas vezes as pessoas esquecem-se dos bons hábitos e passam por cima desta etapa. As razões disto passam pela maneira de ser das pessoas e pela falta de bons simuladores.

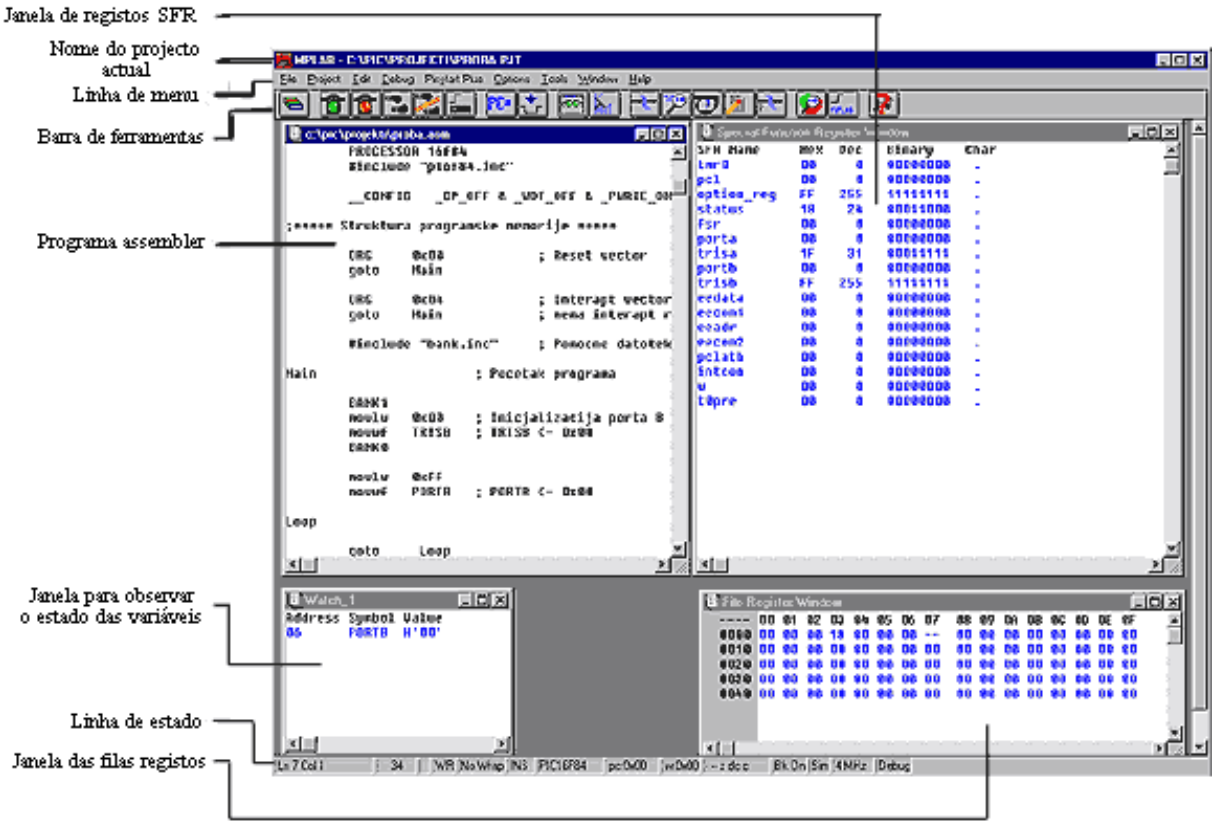
A primeira coisa que precisamos de fazer numa situação real, é o reset do microcontrolador com o comando DEBUG > RUN > RESET. Este comando faz com que surja em negrito a linha de início do programa e que, o contador de programa contenha o valor zero como se pode verificar na linha de estado (pc: 0x00).




**O início da simulação do programa faz-se com o reset do microcontrolador**

Uma das principais características de um simulador, é a possibilidade de observar o estado dos registos dentro do microcontrolador. Principalmente os registos com funções especiais (SFR).
È possível abrir uma janela com os registos SFR, clicando em WINDOW->SPECIAL FUNCTION REGISTERS, ou, no ícone SFR.

Além dos registos SFR, pode ser útil observar os conteúdos dos outros registos. Uma janela com as filas-registos pode ser aberta, clicando em WINDOW->FILE REGISTERS.
Se existirem variáveis no programa, também é conveniente observá-las. A cada variável pode ser atribuída uma janela (Watch Windows) clicando em WINDOW->WATCH WINDOWS.

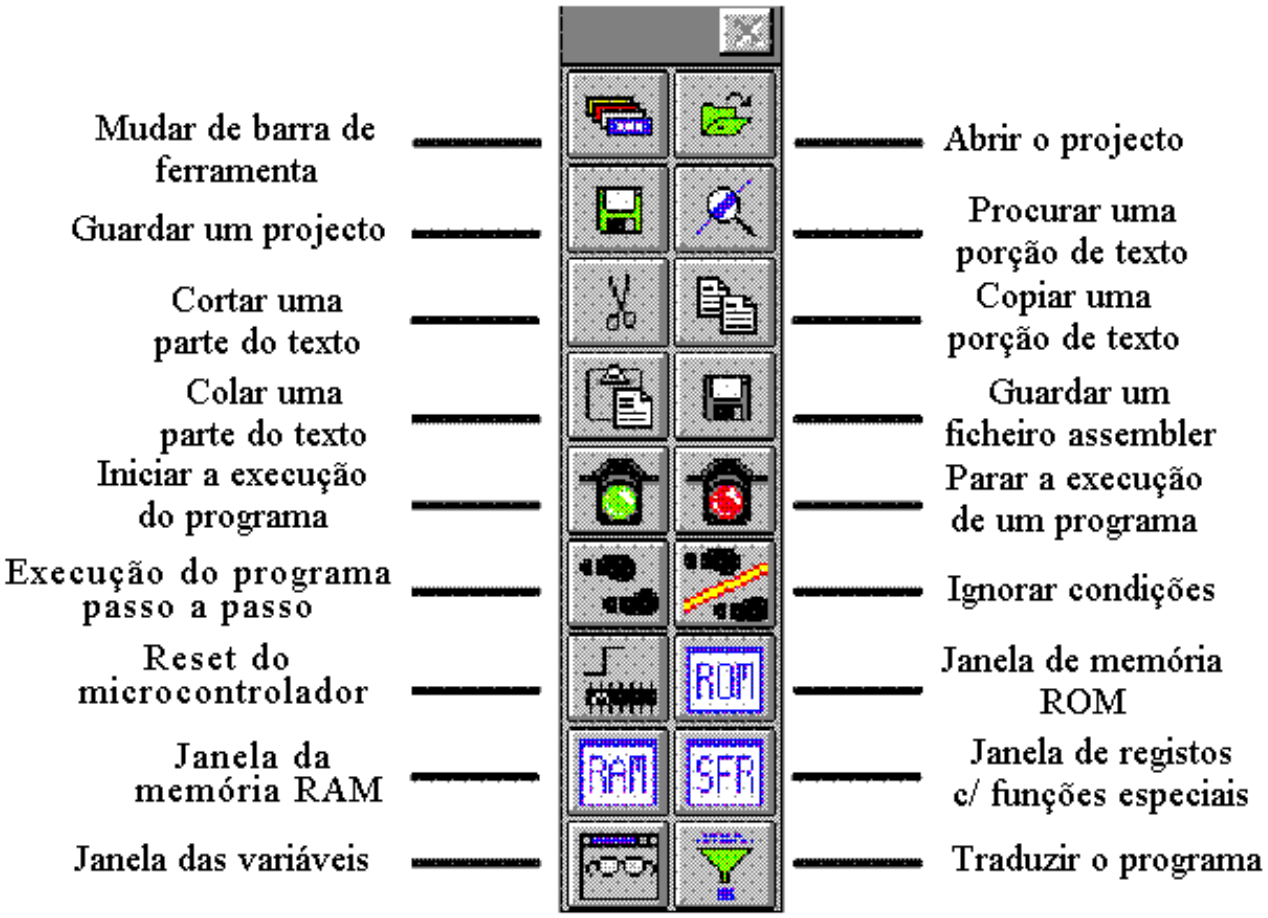



**Simulador com janelas abertas para registos SFR, filas registos e variáveis**

O próximo comando num simulador é DEBUG>RUN>STEP que inicia a simulação passo a passo do programa. O mesmo comando pode ser introduzido através da tecla <F7> do teclado (de um modo geral, todos os comandos mais significativos têm teclas atribuídas no teclado). Utilizando a tecla F7, o programa é executado passo-a-passo. Quando utilizamos uma macro, o ficheiro que contém a macro é aberto (Bank.inc) e podemos prosseguir através da macro. Na janela dos registos SFR, podemos observar como o registo de trabalho W recebe o valor 0xFF e como este valor é transferido para o porto B. Clicando de novo em F7 nós não conseguimos nada porque o programa entra num "loop infinito". Loop infinito é um termo que iremos encontrar muitas vezes. Representa um loop (laço) do qual o microcontrolador não pode sair, a menos que ocorra uma interrupção (se o programa utilizar interrupções) ou, então, quando é executado o reset do microcontrolador.

## 5.8 Barra de ferramentas

Como o MPLAB tem várias componentes, cada uma dessas componentes tem a sua própria barra de ferramentas (toolbar). Contudo, existe uma barra de ferramentas que é uma espécie de resumo de todas as barras de ferramentas. Esta barra de ferramentas é, normalmente suficiente para as nossas necessidades e vai ser explicada com mais detalhe. Na figura debaixo, podemos observar a barra de ferramentas de que precisamos, juntamente com uma breve explicação de cada ícon. Por causa do formato limitado deste livro, esta barra é apresentada como uma barra suspensa. Contudo, normalmente, está colocada horizontalmente por baixo do menu, ao longo do écran.




**Barra de ferramentas universal com uma explicação sumária dos ícones**

## Significado dos ícons na barra de ferramentas

| | |
|---|---|
|  | Sempre que se clica este ícon, uma nova 'toolbar' aparece. Se clicarmos quatro vezes seguidas, a barra actual reaparece. |
|  | Ìcon para abrir um projecto. O projecto aberto desta maneira contém todos os ajustamentos do écran e de todos os elementos que são cruciais para o projecto actual. |
|  | Ícon para guardar um projecto. O projecto guardado conserva todos os ajustamentos de janelas e parâmetros. Quando lermos o programa de novo, tudo regressa ao écran, tal e qual como quando o programa foi fechado. |
|  | Para procurar uma parte do programa ou palavras de que necessitamos, em programas maiores. Usando este ícon podemos encontrar rapidamente uma parte do programa, rótulos, macros, etc. |
|  | Ícon para cortar uma parte do texto. Este e os três icons seguintes são standardisados em todos os programas que lidam com ficheiros de texto. Como cada programa é representado por uma porção de texto, estas operações são muito úteis. |
| | |

| | |
|---|---|
|  | Para copiar uma porção de texto. Existe uma diferença entre este e o ícon anterior. Quando se corta, tira-se uma parte de texto para fora do écran (e do programa), que poderá ser colado a seguir. Mas, tratando-se de uma operação de cópia, o texto é copiado mas não cortado e permanece visualizado no écran. |
|  | Quando uma porção de texto é copiado ou cortado, ela move-se para uma parte da memória que serve para transferir dados no sistema operacional Windows. Mais tarde, clicando neste ícon, ela pode ser 'colada' num texto, no sítio onde se encontra o cursor. |
|  | Guardar um programa (ficheiro assembler). |
|  | Começar a execução do programa a toda a velocidade. Essa execução, é reconhecida pelo aparecimento de uma linha de estado amarela. Nesta modalidade de execução, o simulador executa o programa a toda a velocidade até que é interrompido pelo clicar no ícon de tráfego vermelho. |
|  | Pára a execução do programa a toda a velocidade. Depois de clicar neste ícon, a linha de status torna-se de novo cinzenta e a execução do programa pode continuar passo a passo. |
|  | Execução do programa passo a passo. Clicando neste ícon, nós começamos a executar uma instrução na linha de programa a seguir à linha actual. |
|  | 'Passar por cima'. Como um simulador não é mais que uma simulação da realidade por software, é possível passar por cima de algumas instruções e o programa prosseguir normalmente, depois disso. Normalmente, a parte do programa que interessa ao programador é a que se segue a esse 'salto por cima'. |
|  | Faz o reset do microcontrolador. Clicando neste ícon, o contador de programa é posicionado no início do programa e a simulação pode começar. |
|  | Clicando neste ícon obtemos uma janela com um programa, mas neste caso na memória de programa, onde podemos ver quais as instruções e em que endereços. |
|  | Com a ajuda deste ícon, obtemos uma janela com o conteúdo da memória RAM do microcontrolador . |
|  | Clicando neste ícon, aparece a janela dos registos SFR (registos com funções especiais). Como os registos SFR são usados em todos os programas, recomenda-se que, no simulador, esta janela esteja sempre activa. |
|  | Se uma programa contiver variáveis cujos valores necessitemos de acompanhar (por exemplo um contador), então é necessária uma janela para elas, isso pode ser feito usando este ícon. |
|  | Quando certos erros num programa se manifestam durante o processo de simulação, o programa tem que ser corrigido. Como o simulador usa o ficheiro HEX como entrada, a seguir à correcção dos erros, nós necessitamos de traduzir de novo o programa, de maneira a que estas mudanças também sejam transferidas para o simulador. Clicando neste ícone, o projecto completo é de novo traduzido e, assim, obtemos a nova versão do ficheiro HEX para o simulador. |

# CAPÍTULO 6

## Exemplos



## Introdução

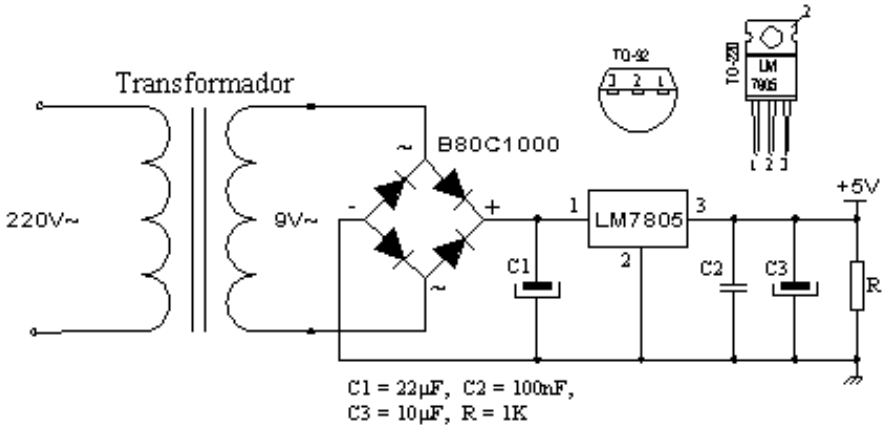
Os exemplos que se mostram neste capítulo, exemplificam como se deve ligar o microcontrolador PIC a periféricos ou a outros dispositivos quando projectamos o nosso próprio sistema de microcontrolador. Cada exemplo contém uma descrição detalhada do hardware com o esquema eléctrico e comentários acerca do programa. Todos os programas podem ser copiados da página da internet da 'MikroElektronika'.

## 6.1 Alimentando o microcontrolador

De um modo geral, uma tensão de alimentação correcta é da maior importância para o bom funcionamento do sistema de microcontrolador. Pode comparar-se este sistema a um homem que precisa de respirar. É provável que um homem que respire ar puro viva mais tempo que um que viva num ambiente poluído.

Para que um microcontrolador funcione convenientemente, é necessário usar uma fonte de alimentação estável, uma função de 'reset ao ligar' fiável e um oscilador. De acordo com as especificações técnicas fornecidas pelo fabricante do microcontrolador PIC, em todas as versões, a tensão de alimentação deve estar compreendida entre 2,0V e 6,0V. A

solução mais simples para a fonte de alimentação é utilizar um regulador de tensão LM7805 que fornece, na sua saída, uma tensão estável de  +5V. Uma fonte com estas características, mostra-se na figura em baixo.




Para que o circuito funcione correctamente, de modo a obter-se 5V estáveis na saída (pino 3), a tensão de entrada no pino 1 do LM7805 deve situar-se entre 7V e 24V. Dependendo do consumo do dispositivo, assim devemos usar o tipo de regulador LM7805 apropriado. Existem várias versões do LM7805. Para um consumo de corrente até 1A, deve usar-se a versão TO-220, com um dissipador de calor apropriado. Se o consumo for somente de 50mA, pode usar-se o 78L05 (regulador com empacotamento TO92 de menores dimensões para correntes até 100mA).

## 6.2 Utilização de macros em programas

Os exemplos que se apresentam nas secções seguintes deste capítulo, vão utilizar frequentemente as macros WAIT, WAITX e PRINT, por isso elas vão ser explicadas com detalhe.

### Macros WAIT, WAITX

O ficheiro Wait.inc contém duas macros WAIT e WAITX. Através destas macros é possível conseguir diferentes intervalos de tempo. Ambas as macros usam o preenchimento do contador TMR0 como intervalo de tempo básico. Modificando o valor do prescaler, nós podemos variar o intervalo de tempo correspondente ao enchimento do contador TMR0.

WAIT.inc

```
;*****Declaração de constantes*****

        CONSTANT PRESCstd = b'00000001' ;Prescaler standard para TMR0

;***** Macros *****

WAIT    macro   timeconst_1

        movlw   timeconst_1
        call    WAITstd
        endm

WAITX   macro   timeconst_2, PRESCext

        movlw   timeconst_2
        mowwf   WCYCLE                  ; Definir espaço de tempo
        movlw   PRESCext                ; Definir valor do prescaler
        call    WAIT_x
        endm

;*****Subprogramas*****

WAITstd
        mowwf   WCYCLE                  ; Definir espaço de tempo
        movlw   PRESCstd                ; Definir valor do prescaler
WAIT_x
        clrf    TMR0
        BANK1
        mowwf   OPTION_REG              ; Atribuir prescaler a TMR0
        BANK0

WAITa
      bcf     INTCON,T0IF       ; pôr a '0' a flag de transbordo de TMR0
WAITb
        btfss   INTCON,T0IF       ; verificar se flag de transbordo igual a '0'
        goto    WAITb             ; continuar a esperar
        decfsz  WCYCLE,1          ; repetir, se esta variável não for zero
        goto    WAITa
        RETURN
```

Se usarmos um oscilador (ressonador) de 4MHz e para valores do prescaler de 0, 1 e 7 a dividir o clock básico do oscilador, os intervalos de tempo causados por transbordo do temporizador TMR0, serão nestes três casos de respectivamente 0,512mS, 1,02mS e 65,3mS. Na prática isso significa que o maior intervalo de tempo possível será de 256x65,3mS = 16,72 segundos.

| Prescaler | Divisor | Enchimento |
|---|---|---|
| b'00000000' | 1:2 | 0.512 ms |
| b'00000001' | 1:4 | 1.02 ms |
| b'00000111' | 1:256 | 65.3 ms |

Para se poderem usar macros no programa principal, é necessário declarar as variáveis wcycle e prescWAIT, como é feito nos exemplos que se seguem neste capítulo.

A Macro WAIT tem um argumento. O valor standard atribuído ao prescaler nesta macro é 1 (1,02mS) e não pode ser alterado.

WAIT timeconst_1

**timeconst_1** é um número de 0 a 255. Multiplicando esse número pelo tempo de enchimento, obtemos o tempo total: TEMPO=timeconst_1 x 1,02mS.

**Exemplo:** WAIT .100

O exemplo mostra como gerar um atraso de 100x1,02mS no total de 102mS.

Ao contrário da macro WAIT, a macro WAITX tem mais um argumento que serve para atribuir um valor ao prescaler. Os dois argumentos da macro WAITX são :

**Timeconst_2** é um número entre 0 e 255. Multiplicando esse número pelo tempo de enchimento, obtemos o tempo total:

TIME=timeconst_1 x 1,02mS x PRESCext

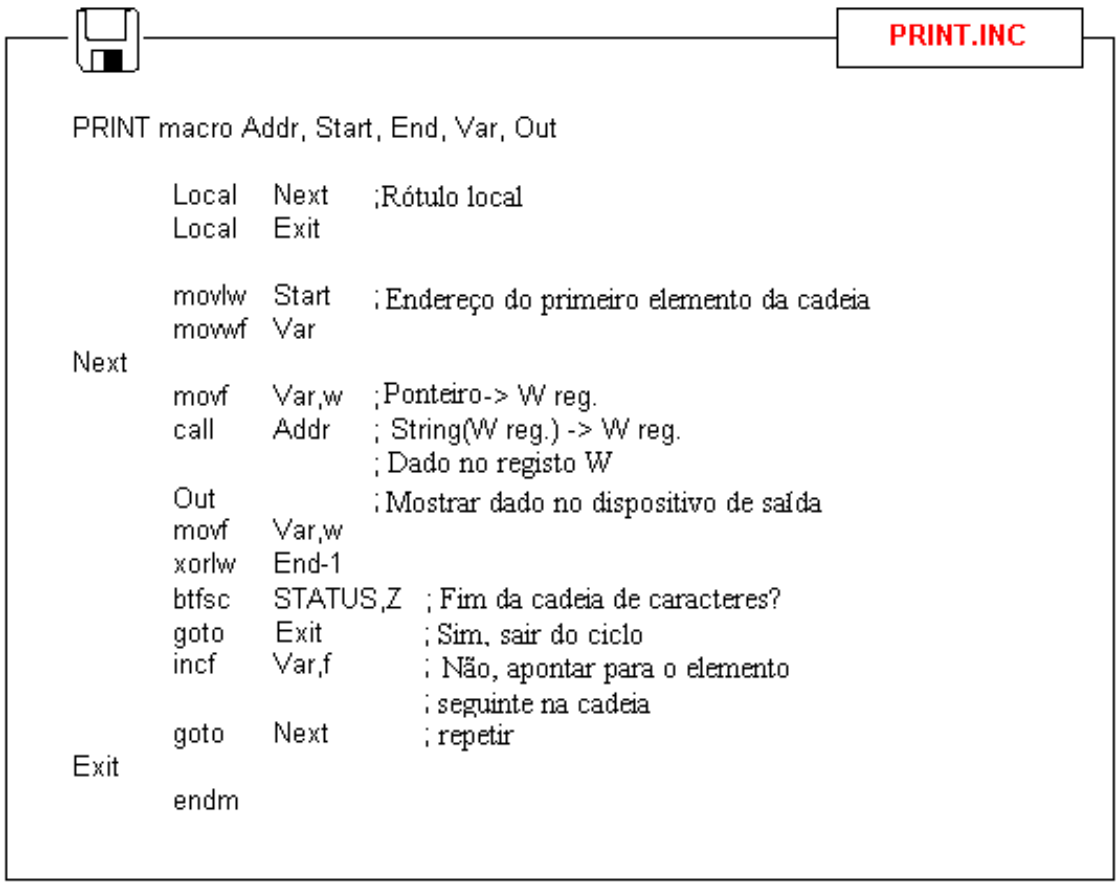
**PRESCext** é um número entre 0 e 7 que estabelece a relação entre o clock e o temporizador TMR0.

**Exemplo:** WAITX .100, 7

O exemplo mostra como gerar um intervalo de tempo de 100x65,3 mS, ou seja, de 6,53S.

MACRO PRINT

A Macro PRINT encontra-se no ficheiro Print.inc. Esta macro facilita o envio de uma série de dados ou caracteres para dispositivos de saída tais como: display LCD, RS232, impressora matricial, ...,etc. A melhor maneira de formar a série, é usar uma directiva dt (definir tabela). Esta instrução guarda uma série de dados na memória de programa, na forma de um grupo de instruções retlw cujos operandos são os caracteres da cadeia de caracteres.



PRINT.INC

```
PRINT macro Addr, Start, End, Var, Out

        Local   Next    ;Rótulo local
        Local   Exit

        movlw   Start   ;Endereço do primeiro elemento da cadeia
        movwf   Var
Next
        movf    Var,w   ;Ponteiro-> W reg.
        call    Addr    ; String(W reg.) -> W reg.
                        ; Dado no registo W
        Out             ;Mostrar dado no dispositivo de saída
        movf    Var,w
        xorlw   End-1
        btfsc   STATUS,Z  ; Fim da cadeia de caracteres?
        goto    Exit      ; Sim, sair do ciclo
        incf    Var,f     ; Não, apontar para o elemento
                          ; seguinte na cadeia
        goto    Next      ; repetir
Exit
        endm
```

O modo como uma sequência é formada usando uma instrução dt, mostra-se no seguinte exemplo:

```
        org     0x00

goto    Main

String  movwf  PCL

String1  dt  “esta é a cadeia ‘ASCII’”

String2  dt  “Segunda série”

        End

Main

movlw   .5

call    String
```

A primeira instrução depois do rótulo Main, escreve a posição de um membro da cadeia (string) no registo w. A seguir, com a instrução call saltamos para o rótulo string, onde a posição de um membro da sequência é adicionada ao valor do contador de programa: PCL=PCL+W. A seguir, teremos no contador de programa um endereço da instrução retlw com o membro da cadeia desejado. Quando esta instrução é executada, o membro da cadeia vai ficar no registo w e o endereço da instrução a executar depois da instrução call estará guardado no contador de programa. O rótulo END é um módulo elegante de marcar o endereço em que a cadeia termina.

A Macro PRINT possui cinco argumentos:

PRINT macro Addr, Start, End, Var, Out

**Addr** é um endereço onde uma ou mais cadeias (que se seguem uma após outra) começam.

**Start** é o endereço do primeiro caracter da cadeia.

**End** é o endereço em que a cadeia termina

**Var** é a variável que tem o papel de mostrar (apontar) os membros da cadeia

**Out** é um argumento que usamos para enviar o endereço do subprograma que trabalha com os dispositivos de saída, tais como: LCD, RS-232 etc.

Exemplo:

```
        org     0x00
        goto    Main

        Series  movwf PCL
        Message dt "mikroElektronika"
        End

Main
        PRINT Series, Message, End, Pointer, LCDw
              :
```

A macro PRINT escreve uma série de caracteres ASCII, correspondentes a 'MikroElektronika' no display LCD. A cadeia ocupa uma parte da memória de programa a começar no endereço 0x03.

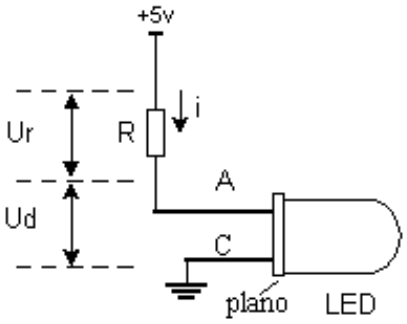
## 6.3 Exemplos

Díodos Emissores de Luz - LEDs

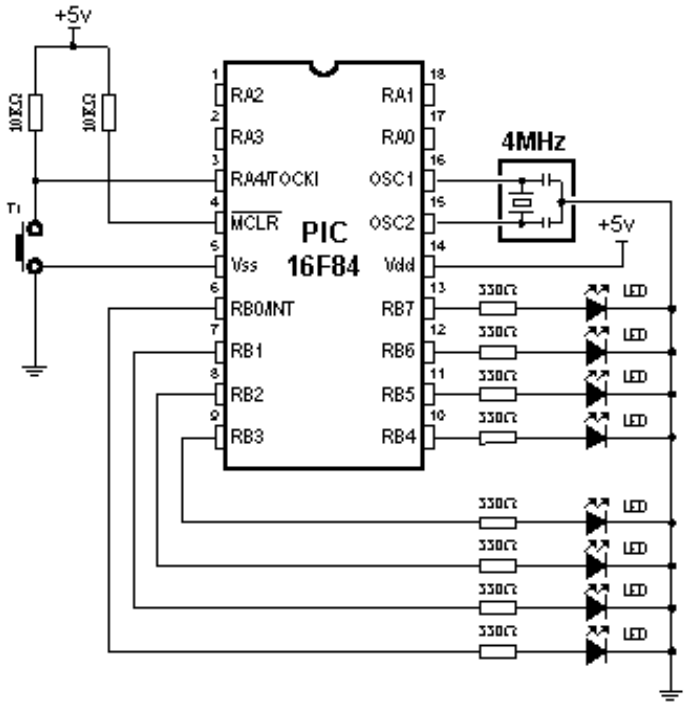
Os LEDs são seguramente uns dos componentes mais usados em electrónica. LED é uma abreviatura para 'Light Emitting Diode' (Díodo emissor de luz). Quando se escolhe um LED, vários parâmetros devem ter-se em atenção: diâmetro, que é usualmente de 3 ou 5mm (milímetros), corrente de funcionamento, habitualmente de cerca de 10mA (pode ser menor que 2mA para LEDs de alta eficiência – alta luminosidade) e, claro, a cor que pode ser essencialmente vermelha ou verde, embora também existam amarelos, laranjas, azuis, etc.

Os LEDs, para emitirem luz, têm que ser ligados com a polaridade correcta e a resistência de limitação de corrente tem também que ter o valor correcto para que o LED não se estrague por sobreaquecimento. O pólo positivo da alimentação deve estar do lado do ânodo e o negativo do lado do cátodo. Para identificar os terminais do led, podemos ter em atenção que, normalmente, o terminal do cátodo é mais curto e, junto deste, a base do LED é plana. Os LED's só emitem luz se a corrente fluir do ânodo para o cátodo. Se for ao contrário, a junção PN fica polarizada inversamente e, a corrente, não passa. Para que o LED funcione correctamente, deve ser adicionada uma resistência em série com este, que vai limitar a corrente através do LED, evitando que este se queime. O valor desta resistência é determinado pelo valor da corrente que se quer que passe através do LED. A corrente máxima que pode atravessar um LED está estabelecida pelo fabricante. Os LEDs de alto rendimento podem produzir uma saída muito satisfatória com uma corrente de 2mA.

Para determinar o valor da resistência em série, nós necessitamos de saber o valor da alimentação. A este valor vamos subtrair a queda de tensão característica no LED. Este valor pode variar entre 1,2v e 1,6v, dependendo da cor do LED. O resultado desta subtracção é a queda de tensão na resistência **Ur**. Sabendo esta tensão e a corrente, determinamos o valor da resistência usando a fórmula **R=Ur/I** .




Os LEDs podem ser ligados ao microcontrolador de duas maneiras. Uma é faze-los acender com o nível lógico zero e a outra com o nível lógico um. O primeiro método é designado por lógica NEGATIVA e o outro por lógica POSITIVA. O diagrama de cima, mostra como se faz a ligação utilizando lógica POSITIVA. Como em lógica POSITIVA se aplica uma voltagem de +5V ao díodo em série com a resistência, ele vai emitir luz sempre que o pino do porto B forneça um valor lógico 1 (1 = saída Alta). A lógica NEGATIVA requer que o LED fique com o ânodo ligado ao terminal positivo da alimentação e o cátodo ligado ao pino porto B, através da resistência. Neste caso, quando uma saída Baixa do microcontrolador é aplicada à resistência em série com o LED, este acende.




**Ligação dos díodos LED ao Porto B do microcontrolador**

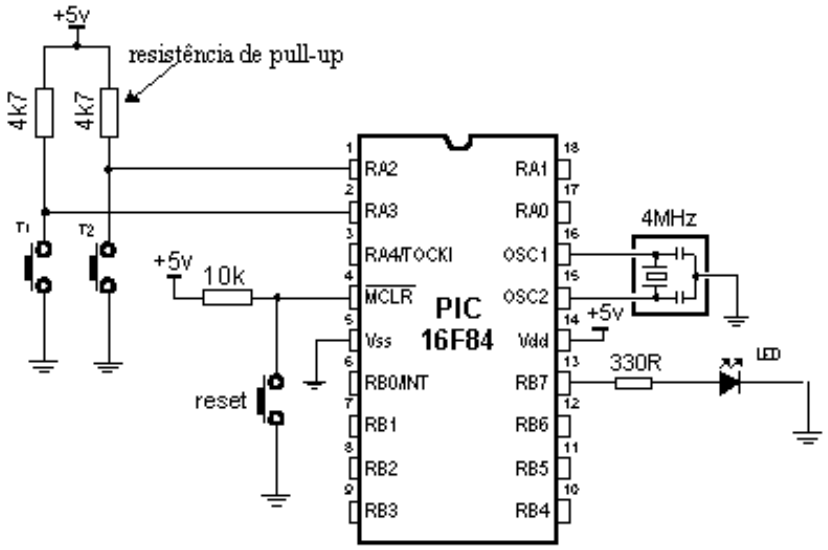
O exemplo que se segue, define o porto B como de saída e põe a nível lógico um todos os pinos deste porto, acendendo os LEDs.

TEST.asm

```
;*****  Declarar e configurar o microcontrolador  *****

      PROCESSOR 16f84
      #include "p16f84.inc"

        __CONFIG   _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;*****  Declarar variáveis  *****

        Cblock 0x0C             ; Princípio da RAM
        WCYCLE                  ; Pertence à macro 'WAITX'
        PRESCwait
        endc

;*****Estrutura da memória de programa*****

        ORG     0x00            ; Vector de reset
        goto    Main

        ORG     0x04            ; Vector de interrupção
        goto    Main            ; Sem rotina de interrupção

        #include "bank.inc"     ; Ficheiros auxiliares


Main                            ; Início do programa

        BANK1
        movlw   0xff            ; Iniciação do Porto A
        movwf   TRISA           ; TRISA <- 0xff  tudo entradas
        movlw   0x00            ; Iniciação do Porto B
        movwf   TRISB           ; TRISB <- 0x00
                                ; Porto B, tudo saídas
        BANK0

        movlw   0xff
        movwf   PORTB           ;acender todos os leds

Loop
        goto    Loop            ; Repetir Loop

   End                          ; Fim de programa
```

## Teclado

As teclas de um teclado, são dispositivos mecânicos usados para desfazer ou estabelecer as ligações entre pares de pontos. As teclas podem aparecer com vários tamanhos e satisfazer vários propósitos. As teclas ou interruptores que vamos usar são também designadas por “teclas-dip”. Elas são muito usadas em electrónica e são soldadas directamente na placa de circuito impresso. Possuem quatro pinos (dois para cada contacto), o que lhes confere uma boa estabilidade mecânica.




**Exemplo de ligação de teclas, aos pinos do microcontrolador**

O modo como funcionam é simples. Quando pressionamos uma tecla, os dois contactos são curto-circuitados e é estabelecida uma ligação. No entanto, isto não é tão simples como parece. O problema reside no facto de a tensão ser uma grandeza eléctrica e na imperfeição dos contactos mecânicos. Quer dizer, antes que o contacto se estabeleça ou seja interrompido, há um curto período de tempo em que pode ocorrer uma vibração (oscilação) como resultado do desajuste dos contactos, ou da velocidade diferente de accionamento das teclas (que depende da pessoa que usa o teclado). O termo associado a este fenómeno é designado por BOUNCE (ressalto) do interruptor. Se não o considerarmos quando estivermos a escrever o programa, pode ocorrer um erro, ou seja, o programa pode detectar vários impulsos apesar de a tecla ter sido pressionada uma única vez. Para evitar isto, um método é introduzir um curto período de espera quando se detecta que um contacto é fechado. Isto assegura que a uma única pressão de tecla, corresponde um único impulso. O tempo de espera (tempo de DEBOUNCING), é produzido por software e o seu valor depende da qualidade da tecla e do serviço que está a efectuar. Este problema pode ser parcialmente resolvido por exemplo, colocando um condensador entre os contactos da tecla, mas, um programa bem feito resolve melhor o problema. Ao escrever este programa vai-se fazendo variar o tempo de debouncing até se verificar que a hipótese de uma detecção fica completamente eliminada.

Nalguns casos, uma simples espera pode ser adequada, mas, se quisermos que o programa execute várias tarefas ao mesmo tempo, uma espera significa, que o microcontrolador "não faz mais nada" durante um longo período de tempo, podendo falhar outras entradas ou, por exemplo, não activar um display no momento adequado.

A melhor solução é ter um programa que detecte quando se pressiona e em seguida se liberta a tecla. A macro em baixo, pode ser usada para fazer o 'debouncing' de uma tecla.

BUTTON.inc

```
Button  macro HiLo, Port, Bit, Delay, Adrress

        Local   Exit      ; Rótulos locais
        Local   Loop

        if HiLo == 0      ; Tecla premida ?
        btfsc Port,Bit    ; Linha de entrada baixa?
        else
        btfss Port,Bit    ; Linha de entrada alta?
        endif
        goto Exit         ; Se tecla não premida, sair da macro

        WAIT Delay        ; Esperar que a tecla estabilize
Loop
        if HiLo == 0
        btfss Port,Bit    ; Tecla libertada ?
        else
        btfsc Port,Bit
        endif
        goto Loop

        WAIT Delay        ; Tempo p/ estabilizar a tecla

        call Adrress      ; Chamar subprograma de atendimento
Exit                      ; Saída da macro
        endm              ; Fim da macro
```

A macro de cima tem vários argumentos que necessitam de serem explicados:

BUTTON macro HiLo, Port, Bit, Delay, Address

**HiLo** pode ser ‘0’ ou ‘1’ e representa o impulso, descendente ou ascendente produzido quando se pressiona uma tecla e que faz com que o subprograma seja executado.

**Port** é o porto do microcontrolador ao qual a tecla está ligada. No caso do PIC16F84 só pode ser o Porto A ou o Porto B.

**Bit** é a linha do porto à qual a tecla está ligada.

**Delay** é um número entre 0 e 255, usado para obter o tempo necessário para que as oscilações nos contactos parem. É calculado pela fórmula TEMPO = Delay x 1ms.

**Address** é o endereço para onde o microcontrolador vai depois de a tecla premida, ter sido solta. A subrotina situada neste endereço contém a resposta a este movimento.

**Exemplo 1**: BUTTON 0, PORTA, 3, .100, Tester1_above

Chave1 está ligada a RA3 (bit 3 do porto A), usa um tempo de ‘debouncing’ de 100 milisegundos e zero é o nível lógico activo. O subprograma que processa o movimento desta tecla encontra-se a partir do endereço com o rótulo Tester1_above.

**Exemplo2:** BUTTON 1, PORTA, 2, .200, Tester2_below

Chave-2 está ligada a RA2 (bit 2 do porto A), usa um tempo de 'debouncing' de 200 milisegundos e '1' é o nível lógico activo. O subprograma que processa o movimento desta tecla encontra-se a partir do endereço com o rótulo Tester2_below.

O exemplo que se segue, mostra como se usa esta macro num programa. O programa TESTER.ASM acende e apaga um LED. O LED está ligado ao bit 7 do porto B. A tecla 1 é usada para acender o LED. A tecla 2 apaga o LED.

**BUTTON.asm**

```
;******Escolher e configurar um microcontrolador*****

        PROCESSOR 16f84
        #include "p16f84.inc"

          __CONFIG   _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;*****    Declarar variáveis    *****

          Cblock 0x0C               ; Início da RAM
          WCYCLE                    ; Pertence à macro  'WAITX'
          PRESCwait
          endc

;***** Estrutura da memória de programa *****

          ORG     0x00              ; Vector de reset
          goto    Main

          ORG     0x04              ; Vector de interrupção
          goto    Main              ; Não há rotina de interrupção

          #include "bank.inc"       ; Ficheiros auxiliares
          #include "button.inc"
          #include "wait.inc"

Main                                ; Início do programa
          BANK1
          movlw   0xff              ; Iniciação do Porto A
          movwf   TRISA             ; TRISA <- 0xff
          movlw   0x00              ; Iniciação do Porto B
          movwf   TRISB             ; TRISB <- 0x00
          BANK0

          clrf    PORTB             ; Porto B <- 0x00

Loop
          Button 0, PORTA, 3 , .100, On     ; Tecla 1
          Button 0, PORTA, 2 , .100, Off    ; Tecla 2
          goto    Loop

On
          bsf     PORTB,7           ; Acender LED
          return

Off
```

```
        bsf     PORTB,7         ; Acender LED
        return

Off
        bcf     PORTB,7         ; Apagar LED
        return

    End             ; Fim de programa
```

## Optoacopladores

Os optoacopladores incluem um LED e um fototransistor juntos no mesmo encapsulamento. O propósito do optoacoplador é manter duas partes do circuito isoladas entre si.

Isto é feito por um certo número de razões:

**Interferência.** Uma parte do circuito pode estar colocada num sítio onde pode captar um bocado de interferência (de motores eléctricos, equipamento de soldadura, motores a gasolina, etc.). Se a saída deste circuito estiver ligada através de um optoacoplador a um outro circuito, somente os sinais desejados passam pelo optoacoplador. Os sinais de interferência não têm "força" suficiente para activar o LED do optoacoplador e assim são eliminados. Exemplos típicos são unidades industriais com muitas interferências que afectam os sinais nas linhas. Se estas interferências afectarem o funcionamento da secção de controle, podem ocorrer erros e a unidade parar de trabalhar.

**Isolamento e amplificação de um sinal em simultâneo.** Um sinal de amplitude baixa, por exemplo de 3V, é capaz de activar um optoacoplador e a saída do optoacoplador pode ser ligada a uma linha de entrada do microcontrolador. O microcontrolador requer uma entrada de 5v e, neste caso, o sinal é amplificado de 3v para 5v. Pode também ser utilizado para amplificar um sinal de corrente. Ver em baixo como se pode usar uma linha de saída de um microcontrolador para amplificar a corrente.

**Tensão de isolamento elevada.** Os optoacopladores possuem intrinsecamente uma grande tensão de isolamento. Como o LED está completamente separado do fototransistor, os optoacopladores podem exibir uma tensão de isolamento de 3kv ou superior.

Os optoacopladores podem ser usados como dispositivos de entrada e de saída. Alguns, fornecem funções adicionais tais como Schmitt trigger (a saída de um Schmitt trigger é 0 ou 1 – ele transforma sinais descendentes ou ascendentes de baixo declive em sinais zero ou um bem definidos). Os optoacopladores são empacotados numa única unidade, ou em grupos de dois ou mais num único encapsulamento. Eles podem também servir como fotointerruptores, uma roda com ranhuras gira entre o LED e o fototransistor, sempre que a luz emitida pelo LED é interrompida, o transistor produz um impulso.

Cada optoacoplador necessita de duas alimentações para funcionar. Ele pode ser usado só com uma alimentação mas, neste caso, a capacidade de isolamento perde-se.

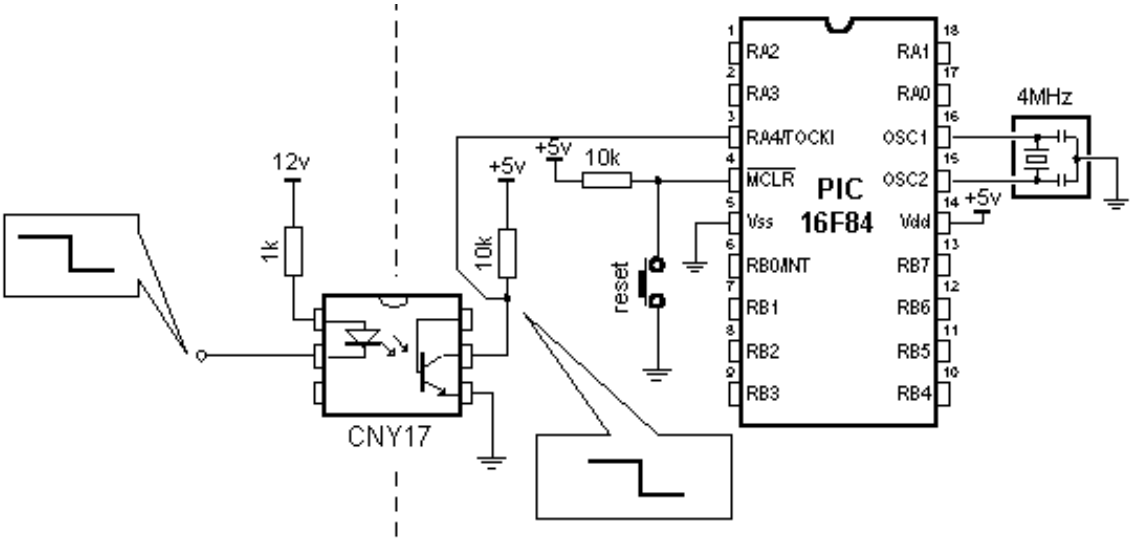
## Optoacoplador numa linha de entrada

O modo como trabalha é simples: quando o sinal chega, o LED do optoacoplador conduz e ilumina o fototransistor que está na mesma cápsula. Quando o transistor começa a conduzir, a tensão entre o colector e o emissor cai para 0,5V ou menos o que o microcontrolador interpreta como nível lógico zero no pino RA4.

O exemplo em baixo, é aplicável em casos como um contador usado para contar items numa linha de produção, determinar a velocidade do motor, contar o número de rotações de um eixo, etc.

Vamos supor que o sensor é um microinterruptor. Cada vez que o interruptor é fechado, o LED é iluminado. O LED 'transfere' o sinal para o fototransistor fazendo este conduzir, assim é produzido um sinal BAIXO na entrada RA4 do microcontrolador.  No microcontrolador deve existir um programa que evite uma falsa contagem e um indicador

ligado às saídas do microcontrolador, que mostre o estado actual da contagem.




**Exemplo de um optoacoplador ligado a uma linha de entrada**

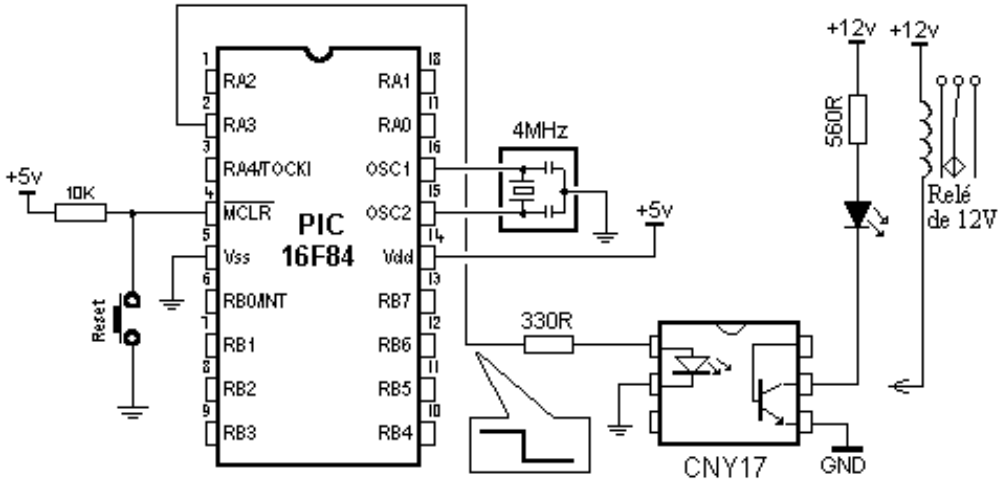
OPTO_IN.asm

```
;*****    Escolhendo e configurando o microcontrolador    *****

        PROCESSOR 16f84
        #include "p16f84.inc"

          __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;*****  Estrutura da memória de programa  *****

          ORG     0x00              ; Vector de reset
          goto    Main

          ORG     0x04              ; Vector de interrupção
          goto    Main              ; Sem rotina de interrupção

          #include "bank.inc"       ; Ficheiro auxiliar

Main                                ; Início do programa
          BANK1
          movlw   0xff              ;Iniciação do Porto A
          movwf   TRISA             ; TRISA <- 0xff (entradas)
          movlw   0x00              ; Iniciação do Porto B
          movwf   TRISB             ; TRISB <- 0x00 (saídas)
          movlw   b'00110000'       ; RA4 -> TMR0,
          movwf   OPTION_REG        ; Incrementar TMR0 no bordo descendente
          BANK0

          clrf    PORTB ; Porto B <- 0
          clrf    TMR0   ; TMR0 <- 0

Loop

          movf    TMR0,w            ; Copiar TMR0 para reg. W
          movwf   PORTB             ; Reg. W para o Porto B
          goto    Loop              ; Repetir o Loop

    End                             ; Fim do programa
```

```
movwf   PORTB           ; Reg. W para o Porto B
goto    Loop            ; Repetir o Loop

End             ; Fim do programa
```

## Optoacoplador numa linha de saída

Um optoacoplador pode ser usado para separar o sinal de saída de um microcontrolador de um dispositivo de saída. Isto pode ser necessário para isolar o circuito de uma tensão alta ou para amplificar a corrente. A saída de alguns microcontroladores está limitada a 25mA. O optoacoplador pode servir-se do sinal de corrente do microcontrolador para alimentar um LED ou relé, como se mostra a seguir:




**Exemplo de um optoacoplador ligado a uma linha de saída**

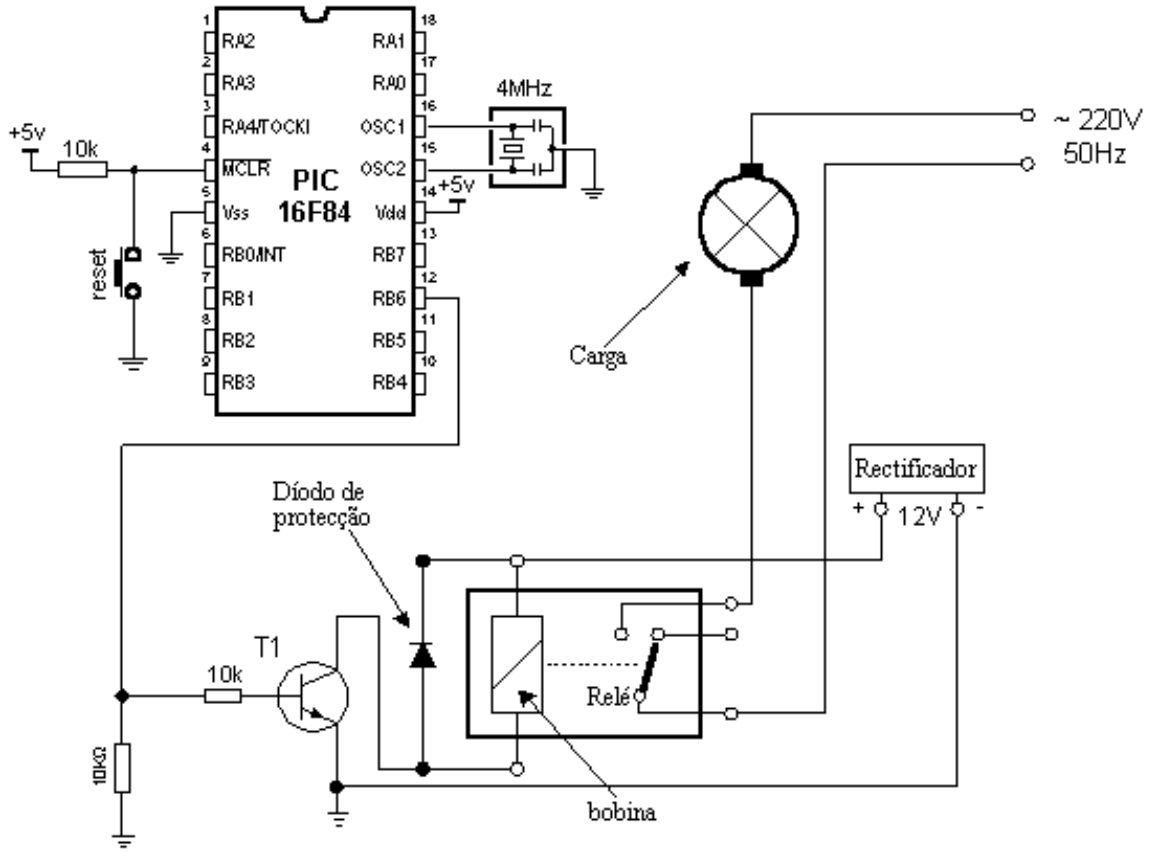
O programa para este exemplo é simples. Fornecendo um nível lógico ‘1’ ao pino 3 do porto A, o LED vai ser activado e o transistor do optoacoplador vai conduzir. A corrente limite para este transistor é de cerca de 250mA.

## O Relé

Um relé é um dispositivo electromecânico que transforma um sinal eléctrico em movimento mecânico. É constituído por uma bobina de fio de cobre isolado, enrolado à volta de um núcleo ferromagnético e por uma armadura metálica com um ou mais contactos.

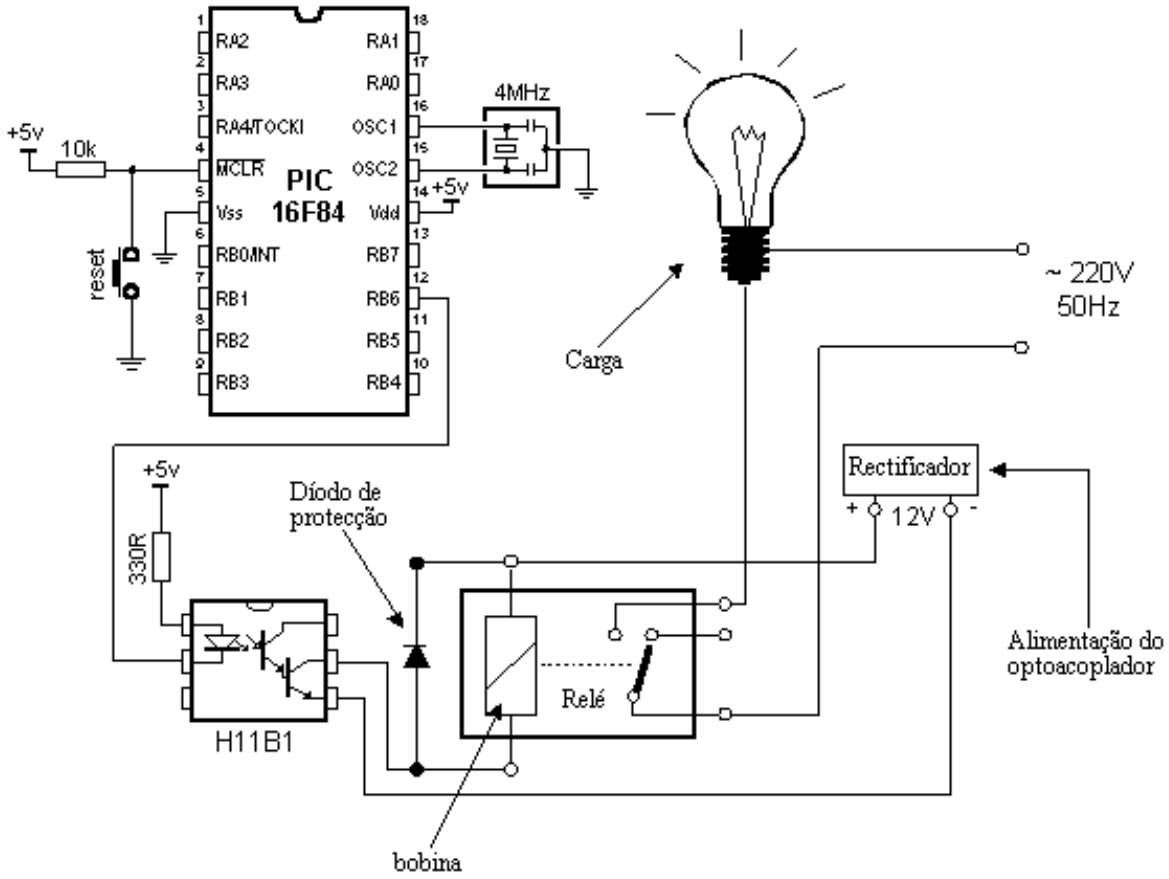
Quando a tensão de alimentação é  ligada à bobina, esta vai ser atravessada por uma corrente e vai produzir um campo magnético que atrai a armadura fechando uns contactos e /ou abrindo outros.

Quando a alimentação do relé é desligada, o fluxo magnético da bobina irá desaparecer e estabelece-se uma corrente por vezes muito intensa em sentido inverso, para se opor à variação do fluxo.  Esta corrente, pode danificar o transistor que está a fornecer a corrente, por isso, um díodo polarizado inversamente deve ser ligado aos terminais da bobina, para curto circuitar a corrente de rotura.




**Ligando um relé a um microcontrolador, através de um transistor**

Muitos microcontroladores não conseguem alimentar um relé directamente e, assim, é necessário acrescentar um transistor ao circuito para obter a corrente necessária. Um nível ALTO na base do transistor, faz este conduzir, activando o relé. O relé pode estar ligado a partir dos seus contactos a qualquer dispositivo eléctrico. Uma resistência de 10k limita a corrente na base do transistor. A outra resistência de 10k entre o pino do microcontrolador e a massa, evita que um ruído na base do transistor faça actuar o relé intempestivamente. Deste modo, só um sinal bem definido proveniente do microcontrolador pode activar o relé.




**Ligando o optoacoplador e um relé a um microcontrolador**

Um relé pode também ser activado através de um optoacoplador que actua como “buffer” de corrente e ao mesmo tempo aumenta a resistência de isolamento. Estes optoacopladores capazes de fornecerem uma corrente muito grande, contêm normalmente um transistor ‘Darlington’ na saída.

A ligação através de um optoacoplador é recomendada especialmente em aplicações de microcontroladores que controlam motores, já que o ruído provocado pela actuação dos comutadores, pode regressar ao microcontrolador através das linhas da alimentação. O optoacoplador faz actuar o relé e este activa o motor.

A figura em baixo, é um exemplo de um programa de activação do relé e inclui algumas macros anteriormente apresentadas.

RELAY.asm

```
;*****   Escolhendo e configurando o microcontrolador    *****

        PROCESSOR 16f84
        #include "p16f84.inc"

          __CONFIG   _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;*****Declarando as variáveis *****

          Cblock 0x0C               ; Início da RAM
          WCYCLE                    ; Pertence à macro 'WAITX'
          PRESCwait
          endc

;*****   Definindo o hardware      *****

          #define RELAY PORTA,3

;***** Estrutura da memória de programa  *****

          ORG     0x00              ; Vector de reset
          goto    Main

          ORG     0x04              ; Vector de interrupção
          goto    Main              ; Sem rotina de interrupção

          #include "bank.inc"       ; Ficheiros auxiliares
          #include "button.inc"
          #include "wait.inc"

Main                                ; Início do programa
          BANK1
          movlw   0x17              ; Iniciação do Porto A
          movwf   TRISA             ; TRISA <- 0x17
          movlw   0x00              ; Iniciação do Porto B
          movwf   TRISB             ; TRISB <- 0x00
          BANK0

          clrf    PORTB             ; Porto B <- 0x00

Loop
          Button 0, PORTA, 0, .100, On     ; Tecla 0
          Button 0, PORTA, 1, .100, Off    ; Tecla 1

          goto    Loop              ; Repetir o Loop

On
          bsf RELAY                 ; Activar relé
          return

Off
          bcf RELAY                 ; Relé inactivo
          return

    End             ; Fim de programa
```

```
End             ; Fim de programa
```

## Produzindo um som

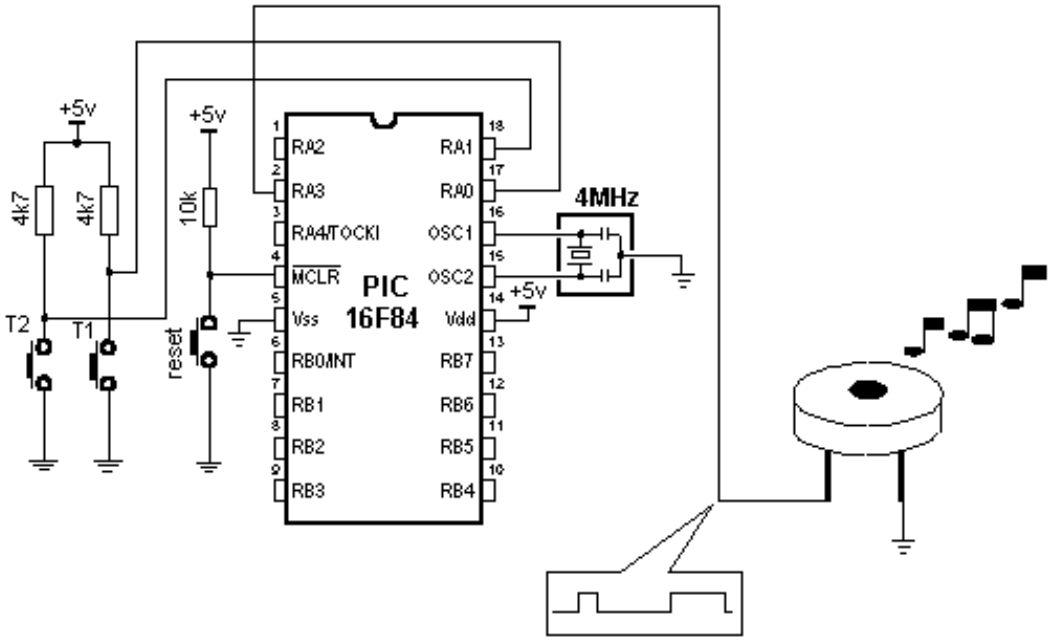
Um diafragma pizoeléctrico pode ser adicionado a uma linha de saída do microcontrolador para se obterem tons, bips e sinais.

É importante saber-se que existem dois tipos de dispositivos pizo emissores de som. Um, contém componentes activos encontram-se dentro do envólucro e só precisam de que lhe seja aplicada uma tensão contínua que emita um tom ou um bip. Geralmente os tons ou bips emitidos por estes dispositivos sonoros não podem mudar, pois são fixados pelos respectivos circuitos internos. Não é este o tipo de dispositivo que vamos discutir neste artigo.

O outro tipo requer, para que possa funcionar, que lhe seja aplicado um sinal. Dependendo da frequência da forma de onda, a saída pode ser um tom, uma melodia, um alarme ou mesmo mensagens de voz.

Para o pormos a funcionar, vamos fornecer-lhe uma forma de onda constituída por níveis Alto e Baixo sucessivos. É a mudança de nível ALTO para BAIXO ou de BAIXO para ALTO que faz com que o diafragma se mova para produzir um pequeno som característico. A forma de onda pode corresponder a uma mudança gradual (onda sinusoidal) ou uma variação rápida (onda rectangular). Um computador é um instrumento ideal para produzir uma onda quadrada. Quando se utiliza a onda quadrada produz-se um som mais áspero.

Ligar um diafragma pizoeléctrico é uma tarefa simples. Um pino é ligado à massa e o outro à saída do microcontrolador, como se mostra na figura em baixo. Deste modo, aplica-se uma forma de onda rectangular de 5v ao pizo. Para produzir um alto nível de saída, a forma de onda aplicada tem que ter uma maior grandeza, o que requer um transistor e uma bobina.




**Ligação de um diafragma pizoeléctrico a um microcontrolador**

Como no caso de uma tecla, podemos utilizar uma macro que forneça uma ROTINA BEEP ao programa, quando for necessário.

BEEP macro freq, duration:

**freq:** frequência do som. Um número maior produz uma frequência mais alta.

**duration:** duração do som. Quanto maior o número, mais longo é o som.

**Exemplo 1:** BEEP 0xFF, 0x02

Nesta caso, a saída do dispositivo pizoeléctrico, tem a maior frequência possível e a duração de 2 ciclos de 65,3mS o que dá 130,6mS.

**Exemplo 2:** BEEP 0x90, 0x05

Aqui, a saída do diafragma pizoeléctrico, tem uma frequência de 0x90 e uma duração de 5 ciclos de 65,3mS. É melhor experimentar diversos argumentos para a macro e seleccionar aquele que melhor se aplica.

A seguir, mostra-se a listagem da Macro BEEP:

**BEEP.inc**

```
;*****  Declarar constantes    *****

        CONSTANT PRESCbeep = b'00000111'          ; 65,3 ms por ciclo

;***** Macros *****

BEEP    macro   freq,duration

        movlw   freq
        movwf   Beep_TEMP1
        movlw   duration
        call    BEEPsub
        endm

BEEPinit macro

        bcf     BEEPport
        BANK1
        bcf     BEEPtris
        BANK0
        endm

;*****  Subrotinas  *****

BEEPsub
        movwf   Beep_TEMP2          ; Guardar duração do som
        clrf    TMR0                ; Iniciar o contador
        bcf    BEEPport
        BANK1
        bcf    BEEPport
        movlw   PRESCbeep           ; Valor do prescaler de TMR0
        movwf   OPTION_REG          ; OPTION <- W
        BANK0
BEEPa
         bcf    INTCON,T0IF    ; Flag de transbordo de TMR0 = 0
BEEPb
        bsf     BEEPport
        call    B_Wait          ; Permanência a "1" lógico
        bcf     BEEPport
        call    B_Wait          ; Permanência a "0" lógico
        btfss   INTCON,T0IF     ; Verificar flag de transbordo de TMR0
        goto    BEEPb           ; Ignorar se for '1'
        decfsz  Beep_TEMP2,1    ; Beep_TEMP2 = 0 ?
        goto    BEEPa           ; Se não, voltar a BEEPa
        RETURN
```

```
        goto    BEEPb           ; Ignorar se for '1'
        decfsz  Beep_TEMP2,1    ; Beep_TEMP2 = 0 ?
        goto    BEEPa           ; Se não, voltar a BEEPa
        RETURN

B_Wait
        movfw   Beep_TEMP1
        movwf   Beep_TEMP3
B_Waita
        decfsz  Beep_TEMP3,1
        goto    B_Waita
        RETURN
```

O exemplo que se segue, mostra o uso de uma macro num programa. O programa produz duas melodias que são obtidas, premindo T1 ou T2. Algumas das macros apresentadas anteriormente são utilizadas no programa.

**BEEP.asm**

```
;*****   Escolhendo e configurando o microcontrolador   *****

      PROCESSOR 16f84
      #include "p16f84.inc"

        __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;*****  Declarar as variáveis   *****

        Cblock  0x0C            ; Início da RAM
        WCYCLE                  ; Pertencem à macro 'WAITX'
        PRESCwait
        Beep_TEMP1              ; Pertencem à macro 'BEEP'
        Beep_TEMP2
        Beep_TEMP3
        endc

;*****     Definir o hardware     *****

        #define  BEEPport PORTA,3 ; Porto e pino do altifalante
        #define  BEEPtris TRISA,3

;***** Estrutura da memória de programa  *****

        ORG     0x00            ; Vector de reset
        goto    Main

        ORG     0x04            ; Vector de interrupção
        goto    Main            ; Sem rotina de interrupção

       #include "bank.inc"      ; Ficheiros auxiliares
       #include "button.inc"
       #include "wait.inc"
       #include "beep.inc"

Main                            ; Início do programa
        BANK1
        movlw   0x17            ; Iniciação do Porto A
        movwf   TRISA           ; TRISA <- 0x17
        movlw   0x00
        movwf   TRISB
```

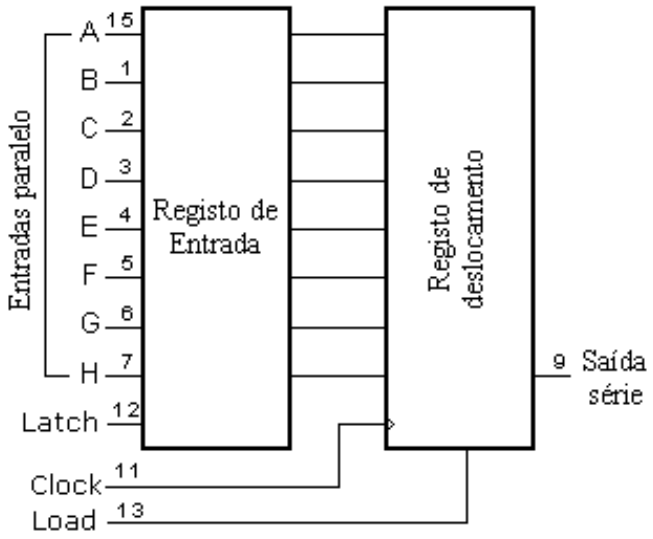
```
        movlw   0x17            ; Iniciação do Porto A
        movwf   TRISA           ; TRISA <- 0x17
        movlw   0x00
        movwf   TRISB
        BANK0

        BEEPinit                ; Iniciar o altifalante

        clrf    PORTB

Loop
        Button 0, PORTA, 0, .100, Play1         ; Tecla 1
        Button 0, PORTA, 1, .100, Play2         ; Tecla 2
        goto    Loop

Play1
        BEEP 0xFF, 0x02
        BEEP 0x90, 0x05
        BEEP 0xC0, 0x03
        BEEP 0xFF, 0x03                         ; Primeira melodia
        return

Play2
        BEEP 0xbb, 0x02
        BEEP 0x87, 0x05
        BEEP 0xa2, 0x03
        BEEP 0x98, 0x03                         ; Segunda melodia
        return

    End             ; Fim de programa
```

## Registos de deslocamento

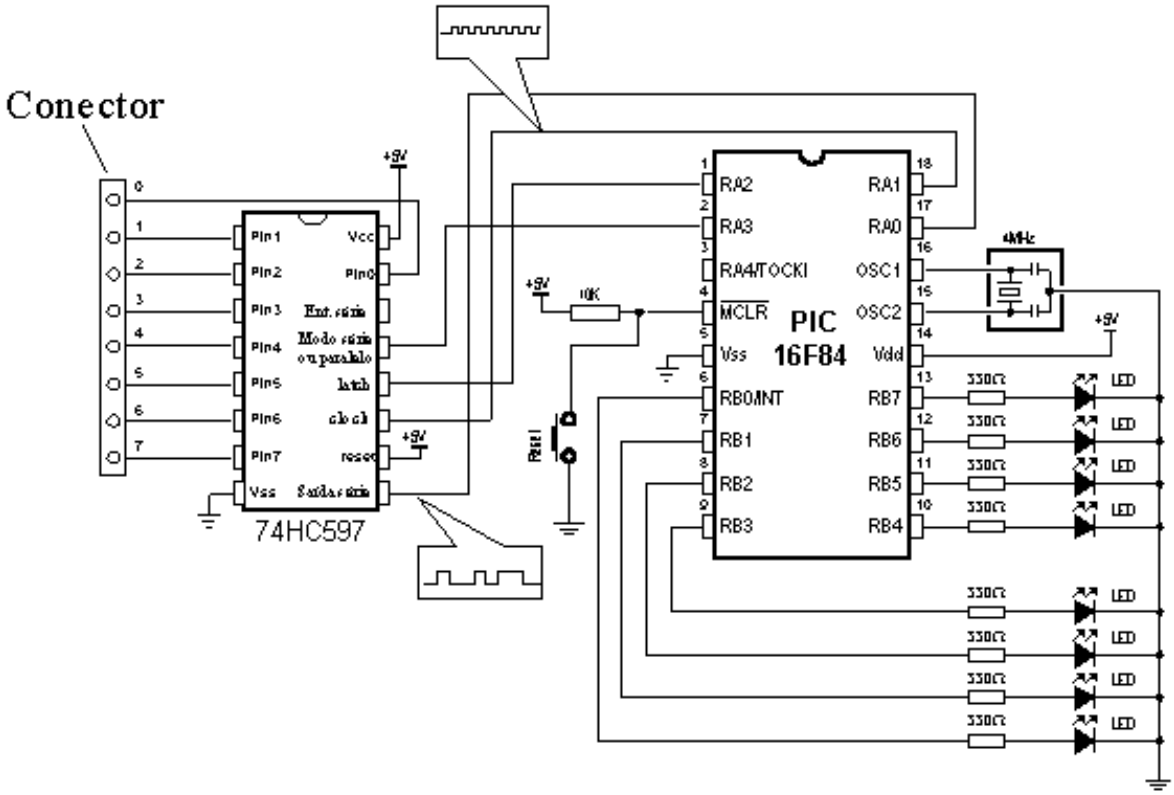
Existem dois tipos de registos de deslocamento: de entrada paralelo e registo de saída paralelo. Os **registos de deslocamento de entrada** recebem os dados em paralelo, através de 8 linhas e enviam-nos em série para o microcontrolador, através de duas linhas. Os **registos de deslocamento de saída** trabalham ao contrário, recebem os dados em série e quando uma linha é habilitada esses dados ficam disponíveis em paralelo em oito linhas. Os registos de deslocamento são normalmente usados para aumentar o número de linhas de entrada e de saída de um microcontrolador. Actualmente não são tão usados, já que os microcontroladores mais modernos dispõem de um grande número de linhas de entrada e de saída. No caso dos microcontroladores PIC16F84, o seu uso pode ser justificado.

## Registo de deslocamento de entrada 74HC597

Os registos de deslocamento de entrada, transformam os dados paralelo em dados série e transferem-nos em série para o microcontrolador. O modo de funcionamento é muito simples. São usadas quatro linhas para transferir os dados: **clock**, **latch**, **load** e **data**. Os dados são lidos primeiro dos pinos de entrada para um registo interno quando uma linha 'latch' é activada. A seguir, com um sinal 'load' activo, os dados passam do registo interno, para o registo de deslocamento e, daqui, são transferidos para o microcontrolador por meio das linhas 'data' (saída série) e 'clock'.




O esquema de ligações do registo de deslocamento 74HC597 ao microcontrolador, mostra-se a seguir:




**Ligação de um registo de deslocamento de entrada paralelo a um microcontrolador**

Para simplificar o programa principal, pode ser usada uma macro para o registo de deslocamento de entrada paralelo. A macro HC597 tem dois argumentos:

HC597 macro Var, Var1

**Var** variável para onde os estados lógicos dos pinos de entrada do registo de deslocamento de entrada paralelo, são transferidos

**Var1** contador de ciclos

**Exemplo:** HC597 dados, contador

Os dados provenientes dos pinos de entrada do registo de deslocamento são guardados na variável dados. O contador/temporizador é usado como contador de ciclos.

Listagem da macro:

HC597.inc

```
HC597   macro   Var,Var1

        Local   Loop            ; Rótulo local

        movlw   .8              ;oito bits a transferir
        mowwf   Var1            ; Iniciação do contador

        bsf     Latch           ;Ler pinos de entrada
        nop
        bcf     Latch           ;Transferir p/ registo de entrada

        bcf     Load            ; p/ o registo de deslocamento
        nop
        bsf     Load

Loop    rlf     Var,f           ; Rodar 'Var' um bit para a esquerda

        btfss   Data            ; '1' na linha de dados?
        bcf     Var,0           ;  Se não, limpar bit 0 da variável 'Var'
        btfsc   Data            ; '0' na linha de dados?
        bsf     Var,0           ;  Se não pôr a '1' o bit 0 de 'Var'

        bsf    Clock            ;Um impulso de clock
       nop
        bcf    Clock

       decfsz  Var1,f           ; Recebidos os oito bits?
       goto    Loop             ; Se não, repetir

      endm
```

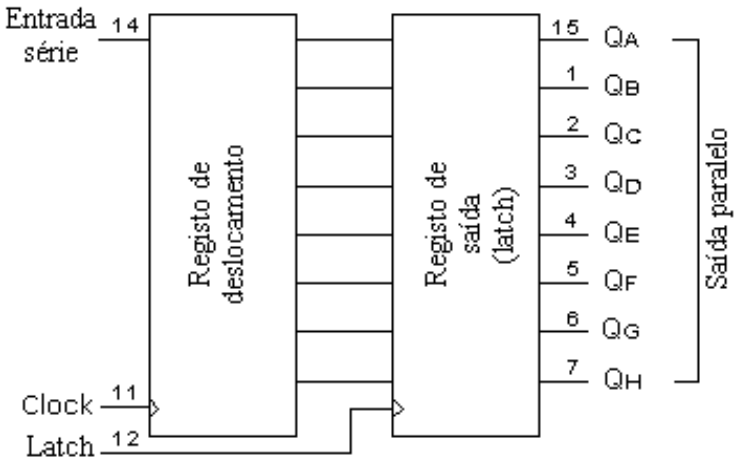
Um exemplo de como usar a macro HC597 mostra-se no programa seguinte. Neste programa, é suposto que o byte de dados é recebido nas entradas paralelo do registo de deslocamento, a partir deste, os bits saem em série e entram no microcontrolador onde são guardados na variável RX. Os LEDs ligados ao Porto B visualizam a palavra de dados.

HC597.asm

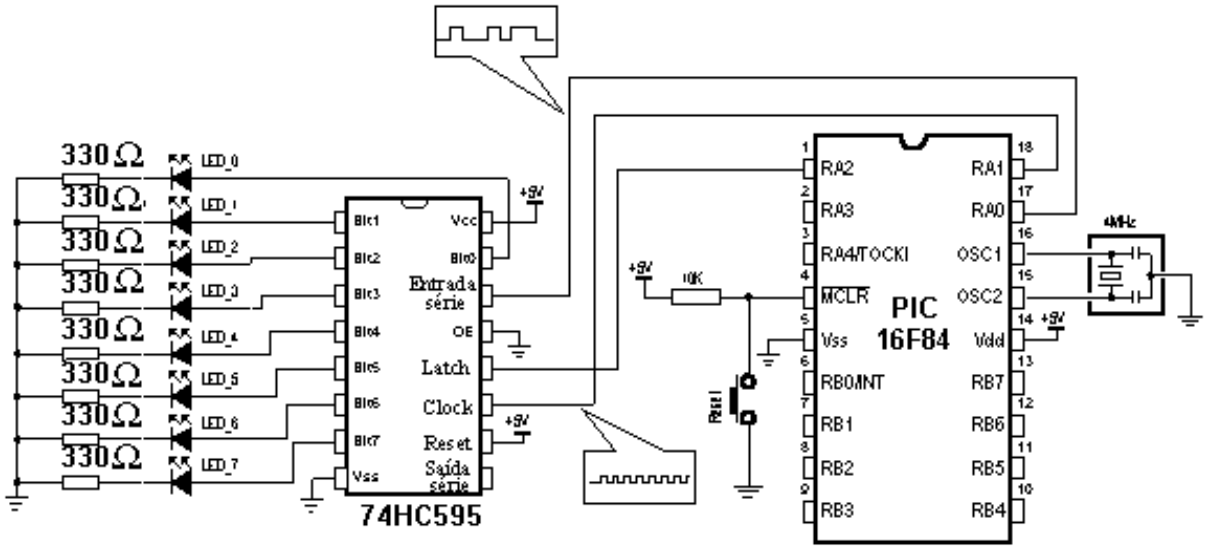
```
;*****  Escolher e configurar o microcontrolador  *****

      PROCESSOR 16f84
      #include "p16f84.inc"

        __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Declarar variáveis  *****

        Cblock  0x0C            ; Início da Ram
        RX
        CountSPI
        endc

;*****  Declarar o hardware   *****

        #define  Data    PORTA,0
        #define  Clock   PORTA,1
        #define  Latch   PORTA,2
        #define  Load    PORTA,3

; ***** Estrutura da memória de programa*****

        ORG    0x00             ; Vector de reset
        goto   Main

        ORG    0x04             ;Vector de interrupção
        goto   Main             ; Sem rotina de interrupção

       #include "bank.inc"      ;Ficheiros auxiliares
       #include "hc597.inc"

Main                            ; Início do programa
        BANK1
        movlw  b'00010001'      ; Iniciação do Porto A
        movwf  TRISA            ; TRISA <- 0x11
        clrf   TRISB
        BANK0

        clrf   PORTA            ; PORTA <- 0x00

        bsf    Load             ;Habilitar reg. de deslocamento

Loop
        HC597   RX, CountSPI

                                ;  Os estados dos pinos de entrada no Reg.
        movf   RX,W             ; de deslocamento estão na variável RX
        movwf  PORTB            ; Variável RX, para o Porto B
        goto   Loop             ; Repetir o loop

        End                     ; Fim de programa
```

## Registo de deslocamento de entrada paralela

Os registos de deslocamento de entrada série e saída paralela, transformam dados série em dados paralelo. Sempre que ocorre um impulso ascendente de clock, o registo de deslocamento lê o estado lógico da linha de dados, guarda-o num registo temporário e repete oito vezes esta operação. Quando a linha 'latch' é activada, os dados são copiados do registo de deslocamento para o registo de saída (registo latch) onde ficam disponíveis em paralelo.




As ligações entre um registo de deslocamento 74HC595 e um microcontrolador, mostram-se no diagrama em baixo.




**Ligação de um registo de deslocamento de saída paralelo a um microcontrolador**

A macro usada neste exemplo é o ficheiro hc595.inc e é designada por HC595.

A macro HC595 tem dois argumentos:

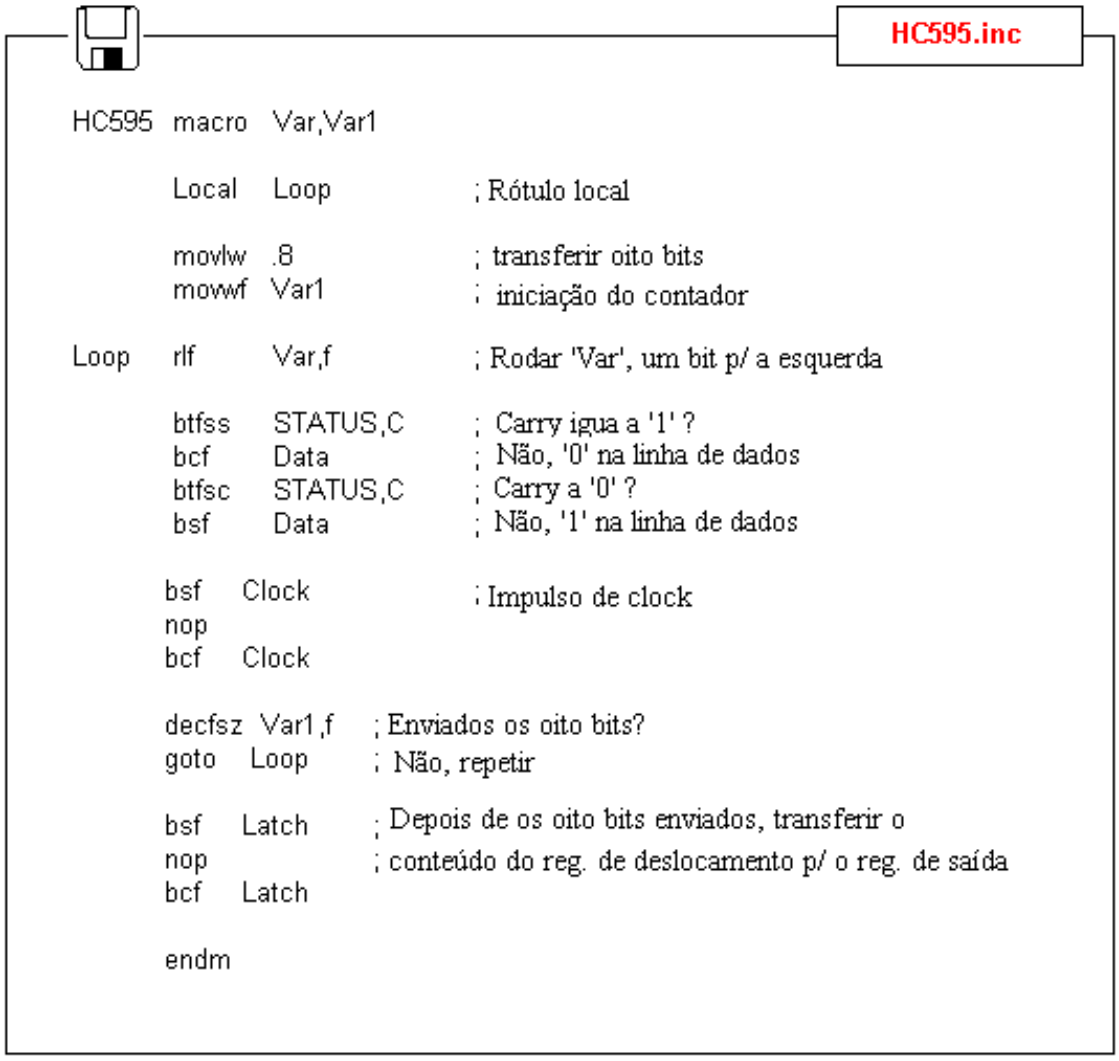
HC595 macro Var, Var1

**Var** variável cujo conteúdo é transferido para as saídas do registo de deslocamento.

**Var1** contador de ciclos (loops)

**Exemplo:** HC595 Dados, contador

O dado que queremos transferir, é guardado na variável dados e a variável contador é usada como contador de ciclos.


HC595.inc

```
HC595  macro  Var,Var1

       Local  Loop          ; Rótulo local

       movlw  .8            ; transferir oito bits
       movwf  Var1          ; iniciação do contador

Loop   rlf    Var,f         ; Rodar 'Var', um bit p/ a esquerda

       btfss  STATUS,C      ; Carry igua a '1' ?
       bcf    Data          ; Não, '0' na linha de dados
       btfsc  STATUS,C      ; Carry a '0' ?
       bsf    Data          ; Não, '1' na linha de dados

       bsf    Clock         ; Impulso de clock
       nop
       bcf    Clock

       decfsz Var1,f    ; Enviados os oito bits?
       goto   Loop      ; Não, repetir

       bsf    Latch     ; Depois de os oito bits enviados, transferir o
       nop              ; conteúdo do reg. de deslocamento p/ o reg. de saída
       bcf    Latch

       endm
```

Um exemplo de como usar a macro HC595 mostra-se no programa que se segue. Os dados provenientes da variável TX são transferidos em série para o registo de deslocamento. Os LEDs ligados às saídas paralelo do registo de deslocamento indicam os níveis lógicos destas linhas. Neste exemplo, é enviado o valor 0xCB (1100 1011) e, portanto, os LEDs ligados aos bits sete, seis, três, um e zero, vão acender.

HC595.asm

```
;*****Seleccionar e configurar o microcontrolador*****

      PROCESSOR 16f84
      #include "p16f84.inc"

        __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;*****  Declarar variáveis  *****

          Cblock  0x0C          ; Início da Ram
          TX                    ; Pertence à macro 'HC595'
          CountSPI
          endc

;*****Declarar o hardware     *****

        #define Data     PORTA,0
        #define Clock    PORTA,1
        #define Latch    PORTA,2

;*****Estrutura da memória de programa *****

        ORG    0x00           ;Vector de reset
        goto   Main

        ORG    0x04           ;Vector de interrupção
        goto   Main           ;Sem rotina de interrupção

        #include "bank.inc"   ;Ficheiros auxiliares
        #include "hc595.inc"

Main                          ; Início do programa

        BANK1
        movlw  0x18           ; Iniciação do Porto A
        movwf  TRISA          ; TRISA <- 0x18
        BANK0

        clrf   PORTA          ; PORTA <- 0x00

        movlw  0xCB           ; Preencher o buffer TX
        movwf  TX             ; TX <- '11001011'

        HC595  TX, CountSPI

Loop    goto   Loop    ;loop infinito

      End              ;Fim de programa
```

# Apêndice A

## Conjunto de Instruções



### A.1 MOVLW Escrever constante no registo W

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* MOVLW **k** |
| **Descrição:** | A constante de 8-bits **k** vai para o registo **W**. |
| **Operação:** | **k** ⇒ ( **W** ) |
| **Operando:** | $0 \leq \mathbf{k} \leq 255$ |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | MOVLW 0x5A |
| | Depois da instrução: W= 0x5A |
| **Exemplo 2:** | MOVLW REGISTAR |
| | Antes da instrução: W = 0x10 e REGISTAR = 0x40 |
| | Depois da instrução: W = 0x40 |

### A.2 MOVWF Copiar W para f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* MOVWF **f** |
| **Descrição:** | O conteúdo do registo **W** é copiado para o registo **f** |
| **Operação:** | **W** ⇒ ( **f** ) |
| **Operando:** | $0 \leq \mathbf{f} \leq 127$ |
| **Flag:** | - |
| **Número de palavras:** | 1 |

| | |
|---|---|
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | MOVWF OPTION_REG |
| | Antes da instrução: OPTION_REG = 0x20 |
| | W = 0x40 |
| | Depois da instrução: OPTION_REG = 0x40 |
| | W = 0x40 |
| **Exemplo 2:** | MOVWF INDF |
| | Antes da instrução: W = 0x17 |
| | FSR = 0xC2 |
| | Conteúdo do endereço 0xC2 = 0x00 |
| | Depois da instrução: W = 0x17 |
| | FSR = 0xC2 |
| | Conteúdo do endereço 0xC2 = 0x17 |

## A.3 MOVF Copiar f para d

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* MOVF **f, d** |
| **Descrição:** | O conteúdo do registo **f** é guardado no local determinado pelo operando **d** |
| | Se **d = 0**, o destino é o registo **W** |
| | Se **d = 1**, o destino é o próprio registo **f** |
| | A opção **d = 1**, é usada para testar o conteúdo do registo **f**, porque a execução desta instrução afecta a flag Z do registo STATUS. |
| **Operação:** | **f** $\Rightarrow$ ( **d** ) |
| **Operando:** | $0 \leq \mathbf{f} \leq 127$, d $\in$ [0, 1] |
| **Flag:** | Z |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | MOVF FSR, 0 |
| | Antes da instrução: FSR = 0xC2 |
| | W = 0x00 |
| | Depois da instrução: W = 0xC2 |
| | Z = 0 |
| **Exemplo 2:** | MOVF INDF, 0 |
| | Antes da instrução: W = 0x17 |
| | FSR = 0xC2 |
| | conteúdo do endereço 0xC2 = 0x00 |
| | Depois da instrução: W = 0x00 |
| | FSR = 0xC2 |
| | conteúdo do endereço 0xC2 = 0x00 |
| | Z = 1 |

## A.4 CLRW Escrever 0 em W

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* CLRW |
| **Descrição:** | O conteúdo do registo **W** passa para 0 e a flag Z do registo STATUS toma o valor 1. |
| **Operação:** | $0 \Rightarrow (\mathbf{W})$ |
| **Operando:** | - |
| **Flag:** | Z |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo:** | CLRW |
| | Antes da instrução: W = 0x55 |
| | Depois da instrução: W = 0x00 |
| | Z = 1 |

## A.5 CLRF Escrever 0 em f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* CLRF **f** |
| **Descrição:** | O conteúdo do registo **'f'** passa para 0 e a flag Z do registo STATUS toma o valor 1. |
| **Operação:** | $0 \Rightarrow f$ |
| **Operando:** | $0 \leq \mathbf{f} \leq 127$ |
| **Flag:** | Z |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | CLRF STATUS |
| | Antes da instrução: STATUS = 0xC2 |
| | Depois da instrução: STATUS = 0x00 |
| | Z = 1 |
| **Exemplo 2:** | CLRF INDF |
| | Antes da instrução: FSR = 0xC2 |
| | conteúdo do endereço 0xC2 = 0x33 |
| | Depois da instrução: FSR = 0xC2 |
| | conteúdo do endereço 0xC2 = 0x00 |
| | Z = 1 |

## A.6 SWAPF Copiar o conteúdo de f para d, trocando a posição dos 4 primeiros bits com a dos 4 últimos

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* SWAPF **f, d** |
| **Descrição:** | Os 4 bits + significativos e os 4 bits – significativos de f, trocam de posições. |
| | Se **d = 0**, o resultado é guardado no registo **W** |
| | Se **d = 1**, o resultado é guardado no registo **f** |

| | |
|---|---|
| **Operação:** | **f** <0:3>⇒ **d** <4:7>, **f** <4:7>⇒ **d** <0:3>, |
| **Operando:** | $0 \leq \mathbf{f} \leq 127$, $d \in [0, 1]$ |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | SWAPF REG, 0 |
| | Antes da instrução: REG = 0xF3 |
| | Depois da instrução: REG = 0xF3 |
| | W = 0x3F |
| **Exemplo 2:** | SWAPF REG, 1 |
| | Antes da instrução: REG = 0xF3 |
| | Depois da instrução: REG = 0x3F |

## A.7 ADDLW Adicionar W a uma constante

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* ADDLW **k** |
| **Descrição:** | O conteúdo do registo **W**, é adicionado à constante de 8-bits **k** e o resultado é guardado no registo **W**. |
| **Operação:** | ( **W** ) + **k** ⇒ **W** |
| **Operando:** | $0 \leq \mathbf{k} \leq 255$ |
| **Flag:** | **C, DC, Z** |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | ADDLW 0x15 |
| | Antes da instrução: W= 0x10 |
| | Depois da instrução: W= 0x25 |
| **Exemplo 2:** | ADDLW REG |
| | Antes da instrução: W = 0x10 |
| | REG = 0x37 |
| | Depois da instrução: W = 0x47 |

## A.8 ADDWF Adicionar W a f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* ADDWF **f**, **d** |
| **Descrição:** | Adicionar os conteúdos dos registos **W** e **f** |
| | Se **d = 0**, o resultado é guardado no registo **W** |
| | Se **d = 1**, o resultado é guardado no registo **f** |
| **Operação:** | **(W)** + ( **f** ) ⇒ **d, d** ∈ [0, 1] |
| **Operando:** | $0 \leq \mathbf{f} \leq 127$ |
| **Flag:** | **C, DC, Z** |
| **Número de palavras:** | 1 |

| | |
|---|---|
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | ADDWF FSR, 0 |
| | Antes da instrução: W = 0x17 |
| | FSR = 0xC2 |
| | Depois da instrução: W = 0xD9 |
| | FSR = 0xC2 |
| **Exemplo 2:** | ADDWF INDF,0 |
| | Antes da instrução: W = 0x17 |
| | FSR = 0xC2 |
| | conteúdo do endereço 0xC2 = 0x20 |
| | Depois da instrução: W = 0x37 |
| | FSR = 0xC2 |
| | Conteúdo do endereço 0xC2 = 0x20 |

## A.9 SUBLW Subtrair W a uma constante

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* SUBLW **k** |
| **Descrição:** | O conteúdo do registo **W**, é subtraído à constante **k** e, o resultado, é guardado no registo **W**. |
| **Operação:** | $\mathbf{k} - (\mathbf{W}) \Rightarrow \mathbf{W}$ |
| **Operando:** | $0 \leq \mathbf{k} \leq 255$ |
| **Flag:** | **C, DC, Z** |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | SUBLW 0x03 |
| | Antes da instrução: W= 0x01, C = x, Z = x |
| | Depois da instrução: W= 0x02, C = 1, Z = 0 Resultado > 0 |
| | Antes da instrução: W= 0x03, C = x, Z = x |
| | Depois da instrução: W= 0x00, C = 1, Z = 1 Resultado = 0 |
| | Antes da instrução: W= 0x04, C = x, Z = x |
| | Depois da instrução: W= 0xFF, C = 0, Z = 0 Resultado < 0 |
| **Exemplo 2:** | SUBLW REG |
| | Antes da instrução: W = 0x10 |
| | REG = 0x37 |
| | Depois da instrução: W = 0x27 |
| | C = 1 Resultado > 0 |

## A.10 SUBWF Subtrair W a f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* SUBWF **f**, **d** |
| **Descrição:** | O conteúdo do registo **W** é subtraído ao conteúdo do registo **f** |
| | Se **d = 0**, o resultado é guardado no registo **W** |

| | |
|---|---|
| | Se **d = 1**, o resultado é guardado no registo **f** |
| **Operação:** | ( **f** ) - (**W**) ⇒ **d** |
| **Operando:** | $0 \leq \mathbf{f} \leq 127$, $\mathbf{d} \in [0, 1]$ |
| **Flag:** | **C, DC, Z** |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo:** | SUBWF REG, 1 |
| | Antes da instrução: REG= 3, W= 2, C = x, Z = x |
| | Depois da instrução: REG= 1, W= 2, C = 1, Z = 0 Resultado > 0 |
| | Antes da instrução: REG= 2, W= 2, C = x, Z = x |
| | Depois da instrução: REG=0, W= 2, C = 1, Z = 1 Resultado = 0 |
| | Antes da instrução: REG=1, W= 2, C = x, Z = x |
| | Depois da instrução: REG= 0xFF, W=2, C = 0, Z = 0 Resultado < 0 |

## A.11 ANDLW Fazer o "E" lógico de W com uma constante

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* ANDLW **k** |
| **Descrição:** | É executado o "E" lógico do conteúdo do registo **W**, com a constante **k** |
| | O resultado é guardado no registo **W**. |
| **Operação:** | ( **W** ) .AND. **k** ⇒ **W** |
| **Operando:** | $0 \leq \mathbf{k} \leq 255$ |
| **Flag:** | Z |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | ANDLW 0x5F |
| | Antes da instrução: W= 0xA3 ; 0101 1111 (0x5F) |
| | ; 1010 0011 (0xA3) |
| | Depois da instrução: W= 0x03; 0000 0011 (0x03) |
| **Exemplo 2:** | ANDLW REG |
| | Antes da instrução: W = 0xA3 ; 1010 0011 (0xA3) |
| | REG = 0x37 ; 0011 0111 (0x37) |
| | Depois da instrução: W = 0x23 ; 0010 0011 (0x23) |

## A.12 ANDWF Fazer o "E" lógico de W com f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* ANDWF **f**, **d** |
| **Descrição:** | Faz o "E" lógico dos conteúdos dos registos **W** e **f** |
| | Se **d = 0**, o resultado é guardado no registo **W** |
| | Se **d = 1**, o resultado é guardado no registo **f** |
| **Operação:** | (**W**) .AND. ( **f** ) ⇒ **d** |

| | |
|---|---|
| **Operando:** | $0 \leq \mathbf{f} \leq 127$, $\mathbf{d} \in [0, 1]$ |
| **Flag:** | Z |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | ANDWF FSR, 1 |
| | Antes da instrução: W= 0x17, FSR= 0xC2; 0001 1111 (0x17) |
| | ; 1100 0010 (0xC2) |
| | Depois da instrução: W= 0x17, FSR= 0x02 ; 0000 0010 (0x02) |
| **Exemplo 2:** | ANDWF FSR, 0 |
| | Antes da instrução: W= 0x17, FSR= 0xC2; 0001 1111 (0x17) |
| | ; 1100 0010 (0xC2) |
| | Depois da instrução: W= 0x02, FSR= 0xC2; 0000 0010 (0x02) |

## A.13 IORLW Fazer o "OU" lógico de W com uma constante

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* IORLW **k** |
| **Descrição:** | É executado o "OU" lógico do conteúdo do registo **W**, com a constante de 8 bits **k**, o resultado é guardado no registo **W**. |
| **Operação:** | ( **W** ) .OR. **k** $\Rightarrow$ **W** |
| **Operando:** | $0 \leq \mathbf{k} \leq 255$ |
| **Flag:** | Z |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | IORLW 0x35 |
| | Antes da instrução: W= 0x9A |
| | Depois da instrução: W= 0xBF |
| | Z= 0 |
| **Exemplo 2:** | IORLW REG |
| | Antes da instrução: W = 0x9A |
| | conteúdo de REG = 0x37 |
| | Depois da instrução: W = 0x9F |
| | Z = 0 |

## A.14 IORWF Fazer o "OU" lógico de W com f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* IORWF **f**, **d** |
| **Descrição:** | Faz o "OU" lógico dos conteúdos dos registos **W** e **f** |
| | Se **d = 0**, o resultado é guardado no registo **W** |
| | Se **d = 1**, o resultado é guardado no registo **f** |
| **Operação:** | (**W**) .OR. ( **f** ) $\Rightarrow$ **d** |
| **Operando:** | $0 \leq \mathbf{f} \leq 127$, $\mathbf{d} \in [0, 1]$ |
| **Flag:** | Z |

| | |
|---|---|
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | IORWF REG, 0 |
| | Antes da instrução: REG= 0x13, W= 0x91 |
| | Depois da instrução: REG= 0x13, W= 0x93 |
| | Z= 0 |
| **Exemplo 2:** | IORWF REG, 1 |
| | Antes da instrução: REG= 0x13, W= 0x91 |
| | Depois da instrução: REG= 0x93, W= 0x91 |
| | Z= 0 |

## A.15 XORLW "OU- EXCLUSIVO" de W com uma constante

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* XORLW **k** |
| **Descrição:** | É executada a operação "OU-Exclusivo" do conteúdo do registo **W**, com a constante **k.** O resultado é guardado no registo **W.** |
| **Operação:** | $(\mathbf{W})$ .XOR. $\mathbf{k} \Rightarrow \mathbf{W}$ |
| **Operando:** | $0 \leq \mathbf{k} \leq 255$ |
| **Flag:** | Z |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | XORLW 0xAF |
| | Antes da instrução: W= 0xB5 ; 1010 1111 (0xAF) |
| | ; 1011 0101 (0xB5) |
| | Depois da instrução: W= 0x1A; 0001 1010 (0x1A) |
| **Exemplo 2:** | XORLW REG |
| | Antes da instrução: W = 0xAF ; 1010 1111 (0xAF) |
| | REG = 0x37 ; 0011 0111 (0x37) |
| | Depois da instrução: W = 0x98 ; 1001 1000 (0x98) |
| | Z = 0 |

## A.16 XORWF "OU-EXCLUSIVO" de W com f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* XORWF **f**, **d** |
| **Descrição:** | Faz o "OU-EXCLUSIVO" dos conteúdos dos registos **W** e **f** |
| | Se **d = 0**, o resultado é guardado no registo **W** |
| | Se **d = 1**, o resultado é guardado no registo **f** |
| **Operação:** | $(\mathbf{W})$ .XOR. $(\mathbf{f}) \Rightarrow \mathbf{d}$ |
| **Operando:** | $0 \leq \mathbf{f} \leq 127$, $\mathbf{d} \in [0, 1]$ |
| **Flag:** | Z |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |

| | |
|---|---|
| **Exemplo 1:** | XORWF REG, 1 |
| | Antes da instrução: REG= 0xAF, W= 0xB5 ; 1010 1111 (0xAF) |
| | ; 1011 0101 (0xB5) |
| | Depois da instrução: REG= 0x1A, W= 0xB5 001 1010 (0x1A) |
| **Exemplo 2:** | XORWF REG, 0 |
| | Antes da instrução: REG= 0xAF, W= 0xB5; 1010 1111 (0xAF) |
| | ; 1011 0101 (0xB5) |
| | Depois da instrução: REG= 0xAF, W= 0x1A ; 0001 1010 (0x1A) |

## A.17 INCF Incrementar f

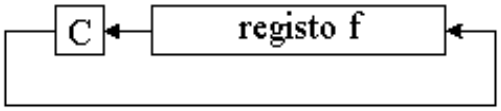
| | |
|---|---|
| **Sintaxe:** | *[rótulo]* INCF **f**, **d** |
| **Descrição:** | Incrementar de uma unidade, o conteúdo do registo f. |
| | Se **d=0**, o resultado é guardado no registo **W** |
| | Se **d=1**, o resultado é guardado no registo **f** |
| **Operação:** | $(\mathbf{f}) + 1 \Rightarrow \mathbf{d}$ |
| **Operando:** | $0 \leq \mathbf{f} \leq 127, \mathbf{d} \in [0, 1]$ |
| **Flag:** | Z |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | INCF REG, 1 |
| | Antes da instrução: REG = 0xFF |
| | Z = 0 |
| | Depois da instrução: REG = 0x00 |
| | Z = 1 |
| **Exemplo 2:** | INCF REG, 0 |
| | Antes da instrução: REG = 0x10 |
| | W = x |
| | Z = 0 |
| | Depois da instrução: REG = 0x10 |
| | W = 0x11 |
| | Z = 0 |

## A.18 DECF Decrementar f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* DECF **f**, **d** |
| **Descrição:** | Decrementar de uma unidade, o conteúdo do registo f. |
| | Se **d=0**, o resultado é guardado no registo **W** |
| | Se **d=1**, o resultado é guardado no registo **f** |
| **Operação:** | $(\mathbf{f}) - 1 \Rightarrow \mathbf{d}$ |
| **Operando:** | $0 \leq \mathbf{f} \leq 127, \mathbf{d} \in [0, 1]$ |
| **Flag:** | Z |

| | |
|---|---|
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | DECF REG, 1 |
| | Antes da instrução: REG = 0x01 |
| | Z = 0 |
| | Depois da instrução: REG = 0x00 |
| | Z = 1 |
| **Exemplo 2:** | DECF REG, 0 |
| | Antes da instrução: REG = 0x13 |
| | W = x |
| | Z = 0 |
| | Depois da instrução: REG = 0x13 |
| | W = 0x12 |
| | Z = 0 |

## A.19 RLF Rodar f para a esquerda através do Carry

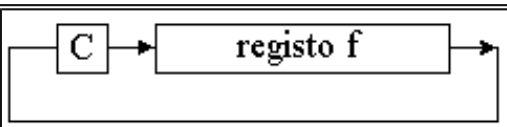
| | | |
|---|---|---|
| **Sintaxe:** | *[rótulo]* RLF **f**, **d** | |
| **Descrição:** | O conteúdo do registo **f** é rodado um espaço para a esquerda, através de C (flag do Carry). | |
| | Se **d = 0**, o resultado é guardado no registo **W** | |
| | Se **d = 1**, o resultado é guardado no registo **f** | |
| **Operação:** | ( **f** <n>) ⇒ **d**<n+1>, f<7> ⇒ C, C ⇒ **d**<0>; | |
| **Operando:** | $0 \le \mathbf{f} \le 127$, $\mathbf{d} \in [0, 1]$ | |
| **Flag:** | C |  |
| **Número de palavras:** | 1 | |
| **Número de ciclos:** | 1 | |
| **Exemplo 1:** | RLF REG, 0 | |
| | Antes da instrução: REG = 1110 0110 | |
| | C = 0 | |
| | Depois da instrução: REG = 1110 0110 | |
| | W = 1100 1100 | |
| | C = 1 | |
| **Exemplo 2:** | RLF REG, 1 | |
| | Antes da instrução: REG = 1110 0110 | |
| | C = 0 | |
| | Depois da instrução: REG = 1100 1100 | |
| | C = 1 | |



## A.20 RRF Rodar f para a direita através do Carry

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* RRF **f**, **d** |

| | |
|---|---|
| **Descrição:** | O conteúdo do registo **f** é rodado um espaço para a direita, através de C (flag do Carry). |
| | Se **d = 0**, o resultado é guardado no registo **W** |
| | Se **d = 1**, o resultado é guardado no registo **f** |
| **Operação:** | ( **f** <n>) ⇒ **d**<n-1>, f<0> ⇒ C, C ⇒ **d**<7>; |
| **Operando:** | $0 \le \mathbf{f} \le 127$, $\mathbf{d} \in [0, 1]$ |
| **Flag:** | C |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | RRF REG, 0 |
| | Antes da instrução: REG = 1110 0110 |
| | W = x |
| | C = 0 |
| | Depois da instrução: REG = 1110 0110 |
| | W = 0111 0011 |
| | C = 0 |
| **Exemplo 2:** | RRF REG, 1 |
| | Antes da instrução: REG = 1110 0110 |
| | C = 0 |
| | Depois da instrução: REG = 0111 0011 |
| | C = 0 |




## A.21 COMF Complementar f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* COMF **f**, **d** |
| **Descrição:** | O conteúdo do registo **f** é complementado. |
| | Se **d = 0**, o resultado é guardado no registo **W** |
| | Se **d = 1**, o resultado é guardado no registo **f** |
| **Operação:** | () ⇒ d |
| **Operando:** | $0 \le \mathbf{f} \le 127$, $\mathbf{d} \in [0, 1]$ |
| **Flag:** | Z |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | COMF REG, 0 |
| | Antes da instrução: REG= 0x13 ; 0001 0011 (0x13) |
| | Depois da instrução: REG= 0x13 ; complementar |
| | W = 0xEC ; 1110 1100 (0xEC) |
| **Exemplo 2:** | COMF INDF, 1 |
| | Antes da instrução: FSR= 0xC2 |
| | conteúdo de FSR = (FSR) = 0xAA |
| | Depois da instrução: FSR= 0xC2 |
| | conteúdo de FSR = (FSR) = 0x55 |

## A.22 BCF Pôr a “0” o bit b de f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* BCF **f**, **b** |
| **Descrição:** | Limpar (pôr a ‘0’), o bit **b** do registo **f** |
| **Operação:** | $0 \Rightarrow$ f<b> |
| **Operando:** | $0 \leq \mathbf{f} \leq 127, 0 \leq \mathbf{b} \leq 7$ |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | BCF REG, 7 |
| | Antes da instrução: REG = 0xC7 ; 1100 0111 (0xC7) |
| | Depois da instrução: REG = 0x47 ; 0100 0111 (0x47) |
| **Exemplo 2:** | BCF INDF, 3 |
| | Antes da instrução: W = 0x17 |
| | FSR = 0xC2 |
| | conteúdo do endereço em FSR (FSR) = 0x2F |
| | Depois da instrução: W = 0x17 |
| | FSR = 0xC2 |
| | conteúdo do endereço em FSR (FSR) = 0x27 |

## A.23 BSF Pôr a “1” o bit b de f

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* BSF **f**, **b** |
| **Descrição:** | Pôr a ‘1’, o bit **b** do registo **f** |
| **Operação:** | $1 \Rightarrow$ f<b> |
| **Operando:** | $0 \leq \mathbf{f} \leq 127, 0 \leq \mathbf{b} \leq 7$ |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | BSF REG, 7 |
| | Antes da instrução: REG = 0x07 ; 0000 0111 (0x07) |
| | Depois da instrução: REG = 0x17 ; 1000 0111 (0x87) |
| **Exemplo 2:** | BSF INDF, 3 |
| | Antes da instrução: W = 0x17 |
| | FSR = 0xC2 |
| | conteúdo do endereço em FSR (FSR) = 0x2F |
| | Depois da instrução: W = 0x17 |
| | FSR = 0xC2 |
| | conteúdo do endereço em FSR (FSR) = 0x28 |

## A.24 BTFSC Testar o bit b de f, saltar por cima se for = 0

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* BTFSC **f**, **b** |
| **Descrição:** | Se o bit **b** do registo **f** for igual a zero, ignorar instrução seguinte. Se este bit **b** for zero, então, durante a execução da instrução actual, a execução da instrução seguinte não se concretiza e é executada, em vez desta, uma instrução NOP, fazendo com que a instrução actual, demore dois ciclos de instrução a ser executada. |
| **Operação:** | Ignorar a instrução seguinte se (**f**<**b**>) = 0 |
| **Operando:** | $0 \leq \mathbf{f} \leq 127, 0 \leq \mathbf{b} \leq 7$ |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 ou 2 dependendo do valor lógico do bit **b** |
| **Exemplo:** | LAB_01 BTFSC REG, 1; Testar o bit 1 do registo REG |
| | LAB_02 ..... ; Ignorar esta linha se for 0 |
| | LAB_03 ..... ; Executar esta linha depois da anterior, se for 1 |
| | Antes da instrução, o contador de programa contém o endereço LAB_01.<br><br>Depois desta instrução, se o bit 1 do registo REG for zero, o contador de programa contém o endereço LAB_03. Se o bit 1 do registo REG for 'um', o contador de programa contém o endereço LAB_02. |

## A.25 BTFSS Testar o bit b de f, saltar por cima se for = 1

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* BTFSS **f**, **b** |
| **Descrição:** | Se o bit **b** do registo **f** for igual a um, ignorar instrução seguinte. Se durante a execução desta instrução este bit **b** for um, então, a execução da instrução seguinte não se concretiza e é executada, em vez desta, uma instrução NOP, assim, a instrução actual demora dois ciclos de instrução a ser executada. |
| **Operação:** | Ignorar a instrução seguinte se (**f**<**b**>) = 1 |
| **Operando:** | $0 \leq \mathbf{f} \leq 127, 0 \leq \mathbf{b} \leq 7$ |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 ou 2 dependendo do valor lógico do bit **b** |
| **Exemplo:** | LAB_01 BTFSS REG, 1; Testar o bit 1 do registo REG |
| | LAB_02 ..... ; Ignorar esta linha se for 1 |
| | LAB_03 ..... ; Executar esta linha depois da anterior, se for 0 |
| | Antes da instrução, o contador de programa contém o endereço LAB_01.<br><br>Depois desta instrução, se o bit 1 do registo REG for 'um', o contador de programa contém o endereço LAB_03. Se o bit 1 do registo REG for zero, o contador de programa contém o endereço LAB_02. |

## A.26 INCFSZ Incrementar f, saltar por cima se der = 0

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* INCFSZ **f**, **d** |

| | |
|---|---|
| **Descrição:** | **Descrição:** O conteúdo do registo **f** é incrementado de uma unidade. |
| | Se **d = 0**, o resultado é guardado no registo **W**. |
| | Se **d = 1**, o resultado é guardado no registo **f**. |
| | Se o resultado do incremento for = 0, a instrução seguinte é substituída por uma instrução NOP, fazendo com que a instrução actual, demore dois ciclos de instrução a ser executada. |
| **Operação:** | (**f**) + 1 ⇒ **d** |
| **Operando:** | $0 \leq \mathbf{f} \leq 127$, $\mathbf{d} \in [0, 1]$ |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 ou 2 dependendo do resultado |
| **Exemplo:** | LAB_01 INCFSZ REG, 1; Incrementar o conteúdo de REG de uma unidade |
| | LAB_02 ..... ; Ignorar esta linha se resultado = 0 |
| | LAB_03 ..... ; Executar esta linha depois da anterior, se der 0 |
| | Conteúdo do contador de programa antes da instrução, PC = endereço LAB_01. Se o conteúdo do registo REG depois de a operação REG = REG + 1 ter sido executada, for REG = 0, o contador de programa aponta para o rótulo de endereço LAB_03. Caso contrário, o contador de programa contém o endereço da instrução seguinte, ou seja, LAB_02. |

## A.27 DECFSZ Decrementar f, saltar por cima se der = 0

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* DECFSZ **f**, **d** |
| **Descrição:** | O conteúdo do registo **f** é decrementado uma unidade. |
| | Se **d = 0**, o resultado é guardado no registo **W**. |
| | Se **d = 1**, o resultado é guardado no registo **f**. |
| | Se o resultado do decremento for = 0, a instrução seguinte é substituída por uma instrução NOP, fazendo assim com que a instrução actual, demore dois ciclos de instrução a ser executada. |
| **Operação:** | (**f**) - 1 ⇒ **d** |
| **Operando:** | $0 \leq \mathbf{f} \leq 127$, $\mathbf{d} \in [0, 1]$ |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 ou 2 dependendo do resultado |
| **Exemplo:** | LAB_01 DECFSZ REG, 1; Decrementar o conteúdo de REG de uma unidade |
| | LAB_02 ..... ; Ignorar esta linha se resultado = 0 |
| | LAB_03 ..... ; Executar esta linha depois da anterior, se der 0 |
| | Conteúdo do contador de programa antes da instrução, PC = endereço LAB_01.<br><br>Se o conteúdo do registo REG depois de a operação REG = REG – 1 ter sido executada, for REG = 0, o contador de programa aponta para o rótulo de endereço LAB_03. Caso contrário, o contador de programa contém o endereço da instrução seguinte, ou seja, LAB_02. |

## A.28 GOTO Saltar para o endereço

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* GOTO **k** |
| **Descrição:** | Salto incondicional para o endereço **k**. |
| **Operação:** | **k** ⇒ PC<10:0>, (PCLATH<4:3>) ⇒ PC<12:11> |
| **Operando:** | 0 ≤ **k** ≤ 2048 |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 2 |
| **Exemplo:** | LAB_00 GOTO LAB_01; Saltar para LAB_01 |
| | : |
| | LAB_01 ..... |
| | Antes da instrução: PC = endereço LAB_00 |
| | Depois da instrução: PC = endereço LAB_01 |

## A.29 CALL Chamar um programa

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* CALL **k** |
| **Descrição:** | Esta instrução, chama um subprograma. Primeiro, o endereço de retorno (PC+1) é guardado na pilha, a seguir, o operando **k** de 11 bits, correspondente ao endereço de início do subprograma, vai para o contador de programa (PC). |
| **Operação:** | PC+1 ⇒ Topo da pilha (TOS – Top Of Stack) |
| **Operando:** | 0 ≤ **k** ≤ 2048 |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 2 |
| **Exemplo:** | LAB_00 CALL LAB_02 ; Chamar a subrotina LAB_02 |
| | LAB_01 : |
| | : |
| | LAB_02 ..... |
| | Antes da instrução: PC = endereço LAB_00 |
| | TOS = x |
| | Depois da instrução: PC = endereço LAB_02 |
| | TOS = LAB_01 |

## A.30 RETURN Retorno de um subprograma

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* RETURN |
| **Descrição:** | O conteúdo do topo da pilha é guardado no contador de programa. |
| **Operação:** | TOS ⇒ Contador de programa PC |
| **Operando:** | - |
| **Flag:** | - |

| | |
|---|---|
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 2 |
| **Exemplo:** | RETURN |
| | Antes da instrução: PC = x |
| | TOS = x |
| | Depois da instrução: PC = TOS |
| | TOS = TOS - 1 |

## A.31 RETLW Retorno de um subprograma com uma constante em W

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* RETLW **k** |
| **Descrição:** | A constante **k** de 8 bits, é guardada no registo **W**. |
| **Operação:** | (**k**) ⇒ **W;** TOS ⇒ PC |
| **Operando:** | $0 \leq k \leq 255$ |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 2 |
| **Exemplo:** | RETLW 0x43 |
| | Antes da instrução: W = x |
| | PC = x |
| | TOS = x |
| | Depois da instrução: W = 0x43 |
| | PC = TOS |
| | TOS = TOS – 1 |

## A.32 RETFIE Retorno de uma rotina de interrupção

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* RETLW **k** |
| **Descrição:** | Retorno de uma subrotina de atendimento de interrupção. O conteúdo do topo de pilha (TOS), é transferido para o contador de programa (PC). Ao mesmo tempo, as interrupções são habilitadas, pois o bit GIE de habilitação global das interrupções, é posto a '1'. |
| **Operação:** | TOS ⇒ PC ; 1⇒ GIE |
| **Operando:** | - |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 2 |
| **Exemplo:** | RETFIE |
| | Antes da instrução: PC = x |
| | GIE = 0 |
| | Depois da instrução: PC = TOS |
| | GIE = 1 |

## A.33 NOP Nenhuma operação

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* NOP |
| **Descrição:** | Nenhuma operação é executada, nem qualquer flag é afectada. |
| **Operação:** | - |
| **Operando:** | - |
| **Flag:** | - |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo:** | NOP |

## A.34 CLRWDT Iniciar o temporizador do watchdog

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* CLRWDT |
| **Descrição:** | O temporizador do watchdog é reposto a zero. O prescaler do temporizador de Watchdog é também reposto a 0 e, também, os bits do registo de estado e são postos a 'um'. |
| **Operação:** | 0 ⇒ WDT |
| | 0 ⇒ prescaler de WDT |
| | 1⇒ |
| | 1⇒ |
| **Operando:** | - |
| **Flag:** | |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo:** | CLRWDT |
| | Antes da instrução: Contador de WDT = x |
| | Prescaler de WDT = 1:128 |
| | Depois da instrução: Contador do WDT = 0x00 |
| | Prescale do WDT = 0 |

## A.35 SLEEP Modo de repouso

| | |
|---|---|
| **Sintaxe:** | *[rótulo]* SLEEP |
| **Descrição:** | O processador entra no modo de baixo consumo. O oscilador pára. O bit (Power Down) do registo Status é reposto a '0'. O bit (Timer Out) é posto a '1'. O temporizador de WDT (Watchdog) e o respectivo prescaler são repostos a '0'. |
| **Operação:** | 0 ⇒ WDT |
| | 0 ⇒ prescaler do WDT |

| | |
|---|---|
| | $1 \Rightarrow$ TO |
| | $0 \Rightarrow PD$ |
| **Operando:** | - |
| **Flag:** | |
| **Número de palavras:** | 1 |
| **Número de ciclos:** | 1 |
| **Exemplo 1:** | SLEEP |
| | Antes da instrução: Contador do WDT = x |
| | Prescaler do WDT = x |
| | Depois da instrução: Contador do WDT = 0x00 |
| | Prescaler do WDT = 0 |

# Apêndice B

## Sistemas Numéricos



### Introdução

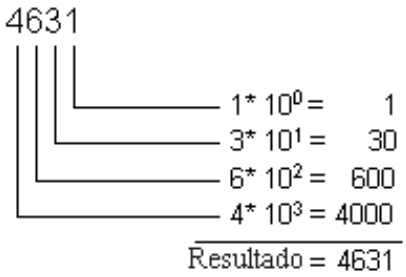
É sempre difícil às pessoas, aceitarem coisas que diferem, em alguma coisa, do seu modo de pensar. Essa é, provavelmente, uma das razões pelas quais os sistemas numéricos diferentes do sistema decimal, ainda são difíceis de entender. No entanto, é necessário aceitar a realidade. O sistema numérico decimal que as pessoas usam no seu dia a dia, foi agora ultrapassado pelo sistema binário, que é usado pelos milhões de computadores de todo o mundo.

Todos os sistemas numéricos possuem uma base. No sistema numérico a base é 10, no sistema binário a base é 2 e, o sistema hexadecimal, tem base 16. O valor representado por cada algarismo no sistema, é determinado pela respectiva posição em relação aos outros algarismos que constituem o número. A soma dos valores representados por cada algarismo dá-nos o número completo. Os sistemas binário e hexadecimal interessam-nos sobremaneira neste livro. Além destes, iremos também abordar o sistema decimal, de modo a compará-lo com os outros dois sistemas. Apesar de o sistema decimal ser um assunto a que já estamos acostumados, iremos discuti-lo de modo a facilitar a compreensão dos outros sistemas.

### B.1 Sistema numérico decimal

A designação de decimal para este sistema numérico, advém de usar a base 10 e usa os algarismos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. À posição de cada um destes algarismos dentro do número está associado um determinado valor. Assim, e caminhando da direita para a esquerda, o algarismo mais à direita, deve ser multiplicado por 1, o algarismo situado imediatamente à esquerda deste, é multiplicado por 10, o que vem a seguir por 100, etc.

**Exemplo:**




As operações de adição, subtracção, divisão e multiplicação no sistema numérico decimal, são realizadas da maneira que todos já conhecemos, portanto, não vamos abordar este assunto.

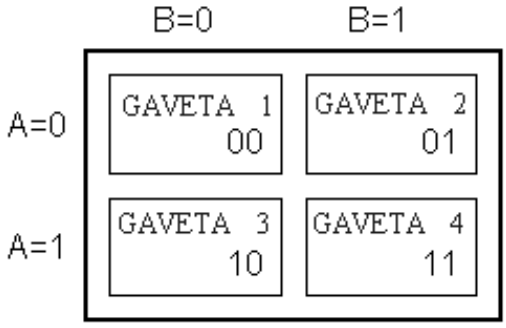
## B.2 Sistema numérico binário

O sistema numérico binário, difere em vários aspectos do sistema decimal que é o que nós utilizamos na vida diária. Este sistema numérico é de base igual a 2 e só contém dois algarismos, que são '1' e '0'. O sistema numérico binário, é o usado nos computadores e nos microcontroladores, porque é, de longe, muito mais adequado ao processamento por parte destes dispositivos, que o sistema decimal. Normalmente, os números binários que iremos usar, contêm 8, 16, ou 32, dígitos binários, não sendo importante, no âmbito deste livro, discutir as razões. De momento, basta-nos aceitar que isto é assim.

**Exemplo:**

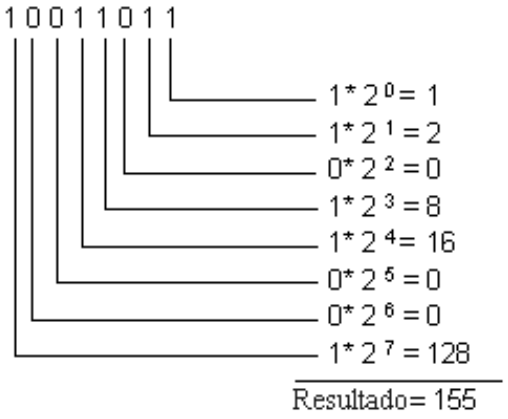
10011011 é um número binário com 8 dígitos

De modo a perceber a lógica dos números binários, vamos considerar um exemplo. Vamos imaginar uma pequena estante com quatro gavetas e, que precisamos de dizer a alguém, para nos trazer qualquer coisa que esteja numa dessas gavetas. Nada mais simples, iremos dizer, (gaveta) em baixo, do lado esquerdo e, a gaveta que pretendemos, fica claramente definida. Contudo, se quisermos dar a indicação sem usarmos instruções tais como esquerda, direita, por baixo, por cima, etc. , nesse caso temos um problema a resolver. Existem muitas soluções para isto, mas vamos escolher uma que seja prática e nos ajude! Vamos designar as linhas por A e as colunas por B. Se A=1, estamos a referir-nos às gavetas de cima e se A=0, estamos a escolher as gavetas em baixo (na linha de baixo). Do mesmo modo, se B=1 estamos a referir-nos às gavetas da esquerda (coluna da esquerda) e se B=0 às gavetas da direita (ver figura seguinte). Agora, só precisamos de escolher uma das quatro combinações possíveis: 00, 01, 10, 11. Este processo de designar individualmente cada gaveta, não é mais que uma representação numérica binária ou a conversão dos números decimais a que estamos habituados para a forma binária. Por outras palavras, referências tais como "primeiro, segundo, terceiro e quarto" são substituídas por "00, 01, 10 e 11".




Aquilo que falta para nos familiarizarmos com a lógica que é usada no sistema numérico binário, é saber extrair um valor numérico decimal de uma série de zeros e uns e, claro, de uma maneira que nós possamos entender. Este procedimento é designado por conversão binário-decimal.

**Exemplo:**




Como se pode ver, a conversão de um número binário para um número decimal é feita, calculando a expressão do lado esquerdo. Consoante a sua posição no número, assim cada algarismo binário traz associado um determinado valor (peso), pelo qual ele vai ser multiplicado, finalmente, adicionando os resultados de todas estas multiplicações, obtemos o tal número decimal que nós já somos capazes de entender. Continuando, vamos agora supor que dentro de cada gaveta existem berlindes: 2 berlindes na primeira gaveta, 4 na segunda gaveta, 7 na terceira e 3 na quarta gaveta. Vamos agora dizer à pessoa que vai abrir as gavetas para usar a representação binária na resposta. Nestas circunstâncias, a pergunta pode ser esta: “Quantos berlindes há na gaveta 01?” e, a resposta, deve ser: “Na gaveta 01 existem 100 berlindes”. Deve notar-se que tanto a pergunta como a resposta são muito precisas, apesar de não estarmos a utilizar a linguagem normal. Deve notar-se também, que, para representarmos todos os números decimais de 0 a 3, apenas precisamos de dois símbolos binários e que, se quisermos números superiores a estes, temos que ir acrescentando mais dígitos binários. Para podermos representar todos os berlindes que estão em qualquer das gavetas, precisamos de 3 algarismo binários. Ou seja, para representarmos os decimais de 0 a 7, bastam-nos três símbolos binários, de 0 a 15, quatro, etc. Generalizando, o maior valor decimal, que pode ser representado por intermédio por um determinado número de símbolos binários, coincide com 2 elevado a um expoente igual ao número de símbolos binários utilizados, subtraído de uma unidade.

**Exemplo:**

$$2^4 - 1 = 16 - 1 = 15$$

Isto significa que é possível representar os números decimais de 0 a 15, apenas com 4 algarismos binários (incluem-se portanto os números ‘0’ e ‘15’), ou seja, 16 valores diferentes. As operações que se executam no sistema decimal, também podem ser executadas no sistema binário. Por razões de clareza e legibilidade, neste apêndice, só iremos abordar a adição e a subtracção.

As regras básicas aplicáveis à adição binária, são:

$$\begin{array}{rrrr} 1 & 0 & 0 & 1 \\ +\ 0 & +\ 1 & +\ 0 & +\ 1 \\ \hline 1 & 1 & 0 & 10 \end{array}$$

A adição é executada de tal modo que, são somados individualmente, os dígitos situados em posições idênticas, em ambos os números. Se ambos os dígitos forem zero, então, a soma é zero, se um deles for igual a ‘0’ e o outro for ‘1’, o resultado é ‘1’. A soma de ‘1’ com ‘1’ dá dois, mas em binário dá ‘0’ e ‘vai um’, este ‘1’ vai ter que ser adicionado ao resultado da soma dos dígitos binários situados imediatamente à esquerda dos anteriores.

**Exemplo:**

```
    1010  Primeiro número
+   1001  Segundo número
-------
   10011  Resultado
```

É possível verificar se o resultado está correcto, convertendo estes dois números binários para o sistema decimal e determinando nós, a soma. Ao fazer a conversão do primeiro número, nós obtemos o decimal 10, e o segundo número, depois de convertido dá 9, a soma correspondente será 19. Deste modo, provámos que o resultado está correcto. Pode, no entanto, surgirem problemas, se o resultado da soma for maior que o maior número binário representável, com o número de dígitos atribuídos. Neste caso, várias soluções podem ser adoptadas, uma das soluções é aumentar o número de posições atribuídas e que foi a seguida no exemplo anterior.

A subtracção, tal como a adição, obedece ao mesmo princípio. O resultado de subtrairmos dois zeros ou dois uns, é zero. Se quisermos subtrair '1' a '0', temos que pedir emprestado '1' ao dígito binário imediatamente à esquerda no número.

**Exemplo:**

```
    1010  Primeiro número
-   1001  Segundo número
-------
    0001  Resultado
```

Para verificar o resultado, tal como fizemos para a adição, convertemos o subtraendo e o subtrator para decimal e, assim, obtemos respectivamente os números 10 e 9. A diferença dá 1, que foi o valor que obtivemos.

## B.3 Sistema numérico hexadecimal

O sistema numérico hexadecimal, tem uma base igual a 16. Se a base é 16, vamos precisar de 16 símbolos diferentes para algarismos. No sistema hexadecimal, os algarismos são: "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F". As letras A, B, C, D, E e F correspondem respectivamente aos decimais 10, 11, 12, 13, 14 e 15. Escolhemos estes símbolos, afim de tornar a escrita dos números mais fácil. Tal como para o caso do sistema binário, também aqui, nós podemos determinar, através da mesma fórmula, qual o maior número decimal que é possível representar com um determinado número de algarismos hexadecimais.

**Exemplo: Com dois algarismos hexadecimais**

$$16^2 - 1 = 256 - 1 = 255$$

Geralmente, os números hexadecimais são escritos com um prefixo "$" ou "0x", ou com o sufixo "h", para realçar o sistema numérico que estamos a utilizar. Assim, o número hexadecimal A37E, pode ainda ser mais correctamente escrito como $A37E, 0xA37E ou A37Eh. Para traduzirmos um número hexadecimal para o sistema numérico binário, não é necessário executar qualquer cálculo mas, simplesmente, substituir cada algarismo do número pelos dígitos binários que o representam. Como o valor máximo representado por um algarismo no sistema hexadecimal é 15, isso significa que são precisos 4 dígitos binários, para cada algarismo hexadecimal.

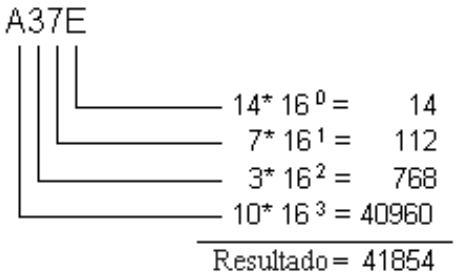
**Exemplo:**

```
$E4 = 1110 0100
        |    |
        E    4
```

Se convertermos ambos os membros da identidade para o sistema numérico decimal, obtemos, em ambos os casos, o número decimal 228, o que comprova que não nos enganamos.

Para obter o equivalente decimal a um número hexadecimal, precisamos de multiplicar cada algarismo do número, por uma potência de 16, cujo expoente, deve corresponder à posição desse algarismo, no número hexadecimal. Em seguida, deve-se adicionar todos os resultados obtidos.

**Exemplo:**




A adição, também é executada, tal como nos dois exemplos precedentes.

**Exemplo:**

| | |
|---|---|
| $3A2B | Primeiro número |
| + $A9C1 | Segundo número |
| $E3EC | Resultado |

Quando adicionamos dois algarismos hexadecimais, se a respectiva soma for igual a 16, escrevemos '0' na posição respectiva e adicionamos uma unidade á soma dos dois algarismos que se seguem. Quer dizer, se a soma dos dois algarismos for, por exemplo, 19 (19 = 16 +3) escrevemos '3' nessa posição e, transferimos o '1' para o algarismo imediatamente a seguir. Se verificarmos, a primeira parcela é o número 14891 e a segunda parcela da soma é 43457. A soma das duas parcelas é 58348, que coincide com o equivalente decimal do número hexadecimal $E3EC. A subtracção, também segue um processo idêntico ao dos dois outros sistemas. Se o algarismo do subtraendo for menor que o do subtrator, é necessário decrementar de uma unidade, o algarismo seguinte no subtraendo.

**Exemplo:**

| | |
|---|---|
| $2D46 | Primeiro número |
| + $1752 | Segundo número |
| $15F4 | Resultado |

Analisando o resultado, verificamos que o subtraendo e o subtrator, correspondem, respectivamente, aos decimais 11590 e 5970, a diferença é 5620, que é o número que obtemos se fizermos a conversão de $15F4, para o sistema numérico decimal.

## Conclusão

O sistema numérico binário é ainda o mais utilizado, o decimal é o mais fácil de perceber e o hexadecimal situa-se entre estes dois sistemas. O sistema hexadecimal é fácil de memorizar e fácil de converter para o sistema binário, o que faz, com que seja, um dos mais importantes sistemas numéricos.

Página anterior | Tabela de conteúdos | Página seguinte

# Apêndice C

## Glossário

Introdução



### Introdução

Como em muitos outros campos da actividade humana, também no caso dos microcontroladores existem termos frequentemente usados e consensualmente adoptados (a partir dos quais outras definições e noções são criadas). Assim, o correcto entendimento de ideias base, permitem apreender, mais facilmente, outras ideias.

### Microcontrolador

É um microprocessador e vários periféricos num único componente electrónico.

### Pino de Entrada/Saída (I/O)

Pino de ligação externa do microcontrolador, que pode ser configurado como entrada ou saída. Na maioria dos casos, o pino de entrada e saída permite ao microcontrolador comunicar, controlar ou ler informação.

## Software

Informação de que o microcontrolador necessita, para poder funcionar. O software não pode apresentar quaisquer erros se quisermos que o programa e o dispositivo funcionem como deve ser. O software pode ser escrito em diversas linguagens tais como: Basic, C, Pascal ou assembler. Fisicamente é um ficheiro guardado no disco do computador.

## Hardware

Microcontrolador, memória, alimentação, circuitos de sinal e todos os componentes ligados ao microcontrolador. Um outro modo de ver isto (especialmente se não estiver a funcionar) é que hardware é aquilo em que se pode dar um pontapé!

## Simulador

Pacote de software para correr num PC que simula o funcionamento interno do microcontrolador. É um instrumento ideal para verificar as rotinas de software e todas as porções de código que não implicam ligação com o mundo exterior. Existem opções para observar o código quando nos deslocamos no programa para trás e para a frente ou passo-a-passo e para detecção de erros.

## ICE

ICE (In Circuit Emulator) ou emulador interno, é um utensílio bastante útil que se liga entre um PC (e não um microcontrolador) e o dispositivo que estamos a desenvolver. Isto permite, ao software, correr no computador PC, mas tudo se passando como se fosse um microcontrolador real que estivesse inserido no dispositivo. O ICE, possibilita que nos desloquemos através do programa, em tempo real, para observar o que se está a passar dentro do microcontrolador e como este comunica com o mundo exterior.

## Emulador de EPROM

Um Emulador de EPROM, é um dispositivo que não emula o microcontrolador completo (como no caso do ICE), mas sim a sua memória. É mais frequentemente usado nos microcontroladores que possuem memória externa. Usando um emulador de Eprom, nós evitamos estar sempre a escrever e a apagar, a memória EPROM.

## Assembler

Pacote de software que traduz código fonte em código que o microcontrolador pode compreender. Uma parte deste software, destina-se também, à detecção dos erros cometidos, ao escrever o programa.

## Ficheiro HEX

Ficheiro criado pelo tradutor assembler, quando traduz um ficheiro fonte e que está num formato que é entendido pelos microcontroladores. Este ficheiro aparece normalmente sob a forma Nome_ficheiro.HEX, daqui deriva a designação de “ficheiro hex".

## Ficheiro LIST

Trata-se de um ficheiro produzido pelo tradutor assembler, que contém todas as instruções do ficheiro fonte, o código destino e os respectivos endereços e, ainda, os comentários que o programador escreveu. É um ficheiro

muito útil para detectar os erros no programa. Este ficheiro tem a extensão LST, daqui provém a sua designação.

### Ficheiro Fonte (Source File)

Ficheiro escrito em linguagem perceptível pelos humanos e pelo tradutor assembler. A tradução do ficheiro fonte produz os ficheiros HEX e LIST.

### Detecção de erros (Debugging)

Ao escrever um programa, nós fazemos erros de que não nos apercebemos. Estes erros podem ser muito simples como é o caso de erros tipográficos, ou erros complexos como os que advêm de um uso incorrecto da linguagem de programção. O assembler é capaz de descobrir a maioria destes erros e mencioná-los no ficheiro '.LST'. Outros erros, só podem ser descobertos, experimentando e observando o funcionamento do dispositivo.

### ROM, EPROM, EEPROM, FLASH, RAM

São tipos de memórias que encontramos quando utilizamos o microcontrolador. A primeira não pode ser limpa, aquilo que se escreve nela, permanece para sempre e nunca mais pode ser apagado. A segunda, pode apagar-se por meio de uma lâmpada de raios ultravioletas. A terceira, pode ser apagada electricamente usando a tensão a que o microcontrolador funciona. A quarta, também é apagável electricamente mas, ao contrário da memória EEPROM, não implica um grande número de ciclos, ao escrever ou apagar os conteúdos dos endereços de memória. Finalmente, o último tipo, é a memória mais rápida, mas não conserva o seu conteúdo quando ocorre uma falha na alimentação. Por isso, esta memória não é usada para guardar o programa, mas sim para guardar os valores das variáveis e resultados intermédios.

### Endereçamento

Determina e designa o local da memória a aceder.

### ASCII

Abreviatura para "American Standard Code for Information Interchange" – código standard americano para troca de dados.  É largamente utilizado e, em particular, atribui a cada caracter alfanumérico (letra ou número) um código de oito bits.

### Carry

Bit de transferência, associado às operações aritméticas.

### Código

Ficheiro ou parte do ficheiro que contém as instruções do programa.

### Byte, Kilobyte, Megabyte

Termos relacionados com quantidades de informação. A unidade básica é o byte que corresponde a 8 bits. Um kilobyte são 1024 bytes e um megabyte tem 1024 kilobytes.

### Flag

Normalmente, refere-se aos bits do registo de estado (status). Quando estes bits são actuados (postos a '1' ou a

‘0’), o programador é notificado de que um determinado acontecimento ocorreu. Se necessário, esta ocorrência pode motivar uma resposta do programa.

## Vector de interrupção ou interrupções

Local na memória do microcontrolador. Desta localização o microcontrolador retira informação sobre uma secção do programa que vai ser executada em resposta a um acontecimento de interesse para o programador e dispositivo.

## Programador

Dispositivo que torna possível escrever software na memória do microcontrolador, possibilitando, assim, que o microcontrolador funcione autonomamente. Compreende uma secção de hardware, normalmente ligada a um dos portos do microcontrolador e uma parte de software sob a forma de um programa que corre num computador PC.

## Produto

O desenvolvimento de um produto, é um misto de sorte e experiência. Em poucas palavras, o estabelecimento de um tempo limite para a produção deve ser evitado, já que, mesmos as requisições mais simples, necessitam de muito tempo para serem desenvolvidas e melhoradas. Para implementar um projecto, nós necessitamos de tempo, calma, raciocínio lógico e, o mais importante de tudo, compreensão das necessidades do consumidor. O percurso típico da criação de um produto deve estar de acordo com o algoritmo seguinte: