CAPÍTULO 11

PROGRAMAÇÃO

CRIANDO UM NOVO PROGRAMA

Finalmente, chegou o momento de falarmos da programação do PIC. Até aqui já foram vistas praticamente todas as teorias e conceitos necessários para que você tenha um bom rendimento durante o aprendizado da programação. Nossa intenção é que este capítulo lhe passe as informações necessárias para que você crie seus próprios programas e projetos. Vamos começar então com algumas dicas sobre a montagem do arquivo de código fonte.

ESTRUTURANDO O CÓDIGO FONTE

Para que um programa seja escrito e funcione corretamente, basta que as instruções certas sejam colocadas na ordem correta. Esta expressão é totalmente verdadeira, mas se só tivermos isto no código do programa, apesar de ele ser funcional, não será eficiente. Isto porque ele não estará devidamente estruturado e padronizado, dificultando futuras alterações e/ou o entendimento por outros programadores. Pode ser que num futuro próximo, você também não consiga mais entender seu próprio programa. Por isso, recomendamos que desde o primeiro código fonte você tenha sempre uma preocupação com a estruturação e organização desse arquivo.

Os anos de trabalho nos laboratórios da Mosaico Engenharia, dedicados à programação do PIC, nos possibilitaram a elaboração de uma estrutura eficiente, o que não significa que ela seja a ideal. Isto depende muito do próprio estilo do programador. Com base na nossa experiência, mostraremos então um exemplo dessa estruturação, que poderá ser adaptada ou modificada de acordo com as suas necessidades. Reforçamos somente a idéia de que a subdivisão e identificação de todas as tarefas (definição de variáveis, entradas, saídas, rotinas, sub-rotinas, programa principal, etc.) é a maneira mais utilizada pela grande maioria dos programadores.

- *A IMPORTÂNCIA DOS COMENTÁRIOS*

Outro recurso tão importante quanto a estruturação do arquivo são os comentários inseridos pelo programador. Esses comentários nada mais são do que textos explicativos que serão desconsiderados pelo compilador na criação do arquivo executável. Isto faz, então, com que os comentários ocupem espaço somente no código fonte, não afetando o tamanho do arquivo final. Por isso, quanto mais comentado estiver um programa, mais fácil será para você (ou qualquer outra pessoa) entendê-lo posteriormente.

No compilador do PIC, para escrever um comentário, basta iniciá-lo com um ponto-e-vírgula (;) em qualquer local do seu programa. Recomendamos que esses comentários estejam escritos sempre na mesma indentação, ao lado direito das instruções. O correto é que praticamente todas as linhas sejam comentadas, e recomendamos ainda que isto seja feito ao mesmo tempo que a programação. Escrever um programa inteiro e só depois comentá-lo pode não ser uma técnica muito eficiente, embora seja possível. Com o tempo você descobrirá qual a melhor maneira de escrever comentários eficientes. Todos os exemplos dados neste livro estão devidamente comentados. Estes comentários dão uma boa base para que você crie os seus próprios.

É importante informá-lo também que o compilador do MPLab pode diferenciar as letras maiúsculas das minúsculas. Para evitar problemas com isso, recomendamos que todo o código seja escrito com somente um tipo de letra. Na Mosaico nos acostumamos a escrever nossos códigos somente em letras maiúsculas.

Para a correta indentação dos comandos e comentários, recomendamos a utilização de tabulação no lugar de espaços, apesar de o compilador ignorar ambos.

- *ARQUIVOS DE DEFINIÇÃO: INCLUDES*

A fim de padronizar e agilizar ainda mais a programação, existe a possibilidade de criarmos e utilizarmos arquivos de definições, que foram chamados pela Microchip de arquivos "Includes". Esses arquivos nada mais são do que arquivos de texto, ou mesmo código fonte, que serão inclusos no seu programa. Desta forma, a própria Microchip já criou um arquivo include para cada tipo de microcontrolador, em que estão definidos os nomes e endereços de todos os SFRs e uma série de outras definições necessárias para a utilização dos microcontroladores. Com esses arquivos evita-se a redigitação de todas essas informações na hora de começar um novo programa.

Os arquivos de includes devem ser gravados com a extensão INC. Futuramente, você também poderá criar seus próprios arquivos de definições. Inicialmente, recomendamos que sejam utilizados os arquivos padrão da Microchip, pois eles são disponibilizados durante a instalação do MPLab (arquivos includes

personalizados devem ser sempre copiados com os arquivos fontes, no caso de transferir o projeto para outro computador).

Para utilização de um arquivo de definições, a seguinte sintaxe deve ser utilizada:

```
#INCLUDE    <nome_do_arquivo.inc>
```

Do modo como foi demonstrado acima, o arquivo deve estar localizado no mesmo diretório de instalação do MPLab. Esta é a melhor forma de referenciar-se aos arquivos fornecidos pela Microchip.

Para os arquivos personalizados, deve ser escrita, além do nome do arquivo, a sua localização completa. Neste caso, os símbolos < e > são substituídos por aspas (").

```
#INCLUDE    "Drive:diretório\nome_do_arquivo.inc"
```

Veja agora o arquivo de include da Microchip para o PIC 16F84 (P16F84.INC).

• *CONSTANTES E DEFINIÇÕES: EQU E DEFINES*

Ao observarmos o arquivo mostrado anteriormente, encontramos, além dos comentários, a palavra EQU. Na verdade, EQU não é um comando do PIC, mas sim uma diretriz para o compilador. Essa diretriz associa um nome a um número. Desta maneira, fica muito mais fácil nos referenciarmos a uma variável pelo seu nome, no lugar do seu endereço na memória. Isto também é muito utilizado para definirmos constantes que serão utilizadas no decorrer da programação. Quando for necessário alterar o valor dessa constante, bastar modificarmos o número relacionado ao seu nome. Simples, não é mesmo?

A sintaxe para a utilização do **EQU** é a seguinte:

```
nome_da_variável   EQU    Endereço_da_memória
```

ou

```
nome_da_constante  EQU    Valor_da_constante
```

Aproveitamos o momento também para informá-lo que um número qualquer pode ser representado de várias formas dentro do assember do PIC:

Decimal: D'??' ou .??

Hexadecimal: H'??' ou 0X??

Binário: B'????????'

ASCII: A'?'

Vejamos então um exemplo utilizado no arquivo P16F84.INC:

```
STATUS      EQU     H'0003'
FSR         EQU     H'0004'
PORTA       EQU     H'0005'
PORTB       EQU     H'0006'
```

Na hora de compilar o código fonte, toda vez que aparecer a palavra PORTA, ela será automaticamente substituída pelo número 05 (em hexa), que é o endereço de memória para a porta A. Desta maneira, é muito mais fácil programar utilizando o nome PORTA no lugar do número 05, não é mesmo? Isto torna as coisas mais fáceis ainda quando nos referimos às variáveis de usuário ou constantes, que devem ser posteriormente alteradas. Neste caso, é muito mais simples alterar o valor em uma definição do que no programa inteiro.

Para este mesmo tipo de aplicação existe também a diretriz #DEFINE. Esta, no entanto, é um pouco mais poderosa que a **EQU**, pelo fato de que ela não substitui nomes somente por números, mas sim por expressões inteiras (substituição de texto).

Essa diretriz é comumente utilizada para definirmos os pinos de entrada e saída do sistema. Para nos referenciarmos a um pino, devemos especificar o bit correto dentro do registrador da porta em questão. Podemos então dar um nome ao conjunto registrador/bit para facilitar o entendimento do programa. Por exemplo:

```
#DEFINE     LED    PORTB,1
```

LED é o nome da definição, e PORTB,1 é o que será considerado toda vez que o nome for utilizado durante o programa. Desta forma, após a utilização do define, podemos nos referir ao pino RB1 com a palavra LED. Isto também facilita muito as alterações futuras no hardware. Caso esse LED seja alterado para o pino RB2, basta alterarmos o #DEFINE, sem precisarmos modificar o código inteiro. Futuramente, veremos que o #DEFINE pode ser utilizado também para a criação de pequenos comandos.

EXEMPLO 0 - ESTRUTURAÇÃO

Vejamos agora um exemplo da estruturação completa. Este modelo será utilizado, no decorrer deste livro, como ponto de partida para todos os programas.

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                        NOME DO PROJETO                                  *
;*                           CLIENTE                                       *
;*          DESENVOLVIDO PELA MOSAICO ENGENHARIA E CONSULTORIA             *
;*   VERSÃO: 1.0                                 DATA: 11/06/98           *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                      DESCRIÇÃO DO ARQUIVO                               *
;*-------------------------------------------------------------------------*
;*   MODELO PARA PIC 16F84                                                 *
;*                                                                         *
;*                                                                         *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                      ARQUIVOS DE DEFINIÇÕES                             *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
#INCLUDE <P16F84.INC>     ;ARQUIVO PADRÃO MICROCHIP PARA 16F84

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                     PAGINAÇÃO DE MEMÓRIA                                *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;DEFINIÇÃO DE COMANDOS DE USUÁRIO PARA ALTERAÇÃO DA PÁGINA DE MEMÓRIA
#DEFINE     BANK0  BCF STATUS,RP0      ;SETA BANK 0 DE MEMÓRIA
#DEFINE     BANK1  BSF STATUS,RP0      ;SETA BANK 1 DE MAMÓRIA
```

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

      ;CORPO DA ROTINA PRINCIPAL

      GOTO MAIN

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

      END
```

Algumas novas diretrizes de compilação apareceram no exemplo e devem ser explicadas:

ORG: Trata-se de um direcionamento para a posição de memória de programação. Só devemos nos preocupar com esse endereçamento no início do programa (vetor de reset), no início das interrupções (vetor de interrupção) e em alguns casos específicos de paginação da área de programa, que não é muito importante no caso do 16F84. Observe que como o vetor de reset do PIC 16F84 é o endereço 0x00, antes da escrita da primeira instrução é dada a diretriz ORG 0x00, para que o programa comece neste ponto. Logo em seguida, um pulo é dado para uma rotina localizada bem mais à frente (GOTO INICIO). Isto é necessário porque no endereço 0x04 deve ser iniciada a rotina de interrupção. Para tal, uma nova diretriz é especificada: ORG 0x04.

END: Essa diretriz deve ser sempre colocada ao final do programa, pois quando o compilador encontrá-la, a compilação será terminada. Para uma melhor estruturação, sempre deixamos a rotina principal no final do arquivo, seguida somente pelo END.

- **CBLOCK e ENDC**: É uma maneira simplificada de definirmos vários EQUs com endereços seqüenciais. Observe que utilizamos este recurso na definição das variáveis do sistema, mas precisamos informar somente o endereço da primeira variável (CBLOCK 0x20). As demais são definidas na seqüência. Isto possibilita mudarmos facilmente o local de todo o bloco de variáveis. Este recurso é utilizado, por exemplo, quando estamos convertermos o programa de um modelo de PIC para outro, cuja RAM de usuário inicia-se em um endereço diferente.

Os demais itens apresentados neste exemplo e que ainda não são conhecidos referem-se a comandos que serão vistos nas próximas seções. O apêndice B explica todas as diretrizes existentes.

TRABALHANDO COM A MEMÓRIA

O objetivo desta seção é a apresentação da operação e dos comandos relacionados à memória do PIC. Como guardar e recuperar valores e também movê-los de um lugar para outro.

- ***O REGISTRADOR WORK (W OU ACUMULADOR)***

Para reforçar os conceitos já vistos, vale lembrá-lo que o PIC possui um registrador temporário utilizado nas operações da ULA e não faz parte direta da memória RAM do sistema. Esse registrador é o Work (W) e será extremamente utilizado de agora em diante, já que não podemos ler ou escrever diretamente na memória sem o uso dele.

- ***LIDANDO COM DADOS (MOVLW, MOVWF, MOVF, CLRF E CLRW)***

Vamos aproveitar o exemplo acima para conhecermos também os três comandos mais utilizados nos programas para PIC. Trata-se do MOVLW, MOVWF e MOVF. Vamos utilizar aqueles termos já estudados para conhecermos a verdadeira função desses comandos?

MOVLW = Move (MOV) uma literal (L) para o registrador work (W).

MOVWF = Move (MOV) o valor de work (W) para um registrador (F).

MOVF = Move (MOV) o valor de um registrador (F) para um local de destino passado como argumento (f ou w).

As sintaxes destas instruções são:

```
MOVLW   k      ;em que k é o número que será colocado em W.
MOVWF   f      ;em que f é o endereço da memória onde será guardado o valor de W.
MOVF    f,d    ;em que f é o registrador que será movido para o destino d
               ;Lembre-se que existem dois destinos possíveis: W e F.
```

Assim sendo, para guardar um número em uma posição de memória, devemos primeiro movê-lo para o work (MOVLW) e depois movê-lo do work para o registrador propriamente dito (MOVWF).

Por exemplo, para definirmos as condições de operação da máquina, devemos ajustar os bits do registrador OPTION_REG. A maneira mais rápida de fazermos isto é movendo um número para o endereço desse registrador. O número é representado na forma binária para compreendermos melhor o estado de cada bit. Lembre-se que pela sintaxe do comando MOVWF deveríamos especificar o endereço do registrador. No entanto, no arquivo de include, este endereço foi sobrecarregado no nome OPTION_REG.

```
MOVLW     B'10000100'
MOVWF     OPTION_REG        ;DEFINE OPÇÕES DE OPERAÇÃO
```

Já a instrução MOVF é utilizada para movermos o conteúdo de um registrador para outro registrador. Por exemplo, veja como fica o código para escrevermos em TEMPO2 o mesmo valor existente em TEMPO1:

```
MOVF  TEMPO1,W      ;MOVE O VALOR DE TEMPO 1 PARA O WORK
MOVWF TEMPO2        ;MOVE O VALOR DE W (TEMPO1) PARA TEMPO2
```

Uma dúvida que pode surgir agora é em relação ao outro destino possível que pode ser utilizado com a instrução **MOVF**. Para que moveríamos o valor de um registrador para ele mesmo? Por mais idiota que possa parecer esta ação, ela tem alguma aplicação que será vista mais adiante.

Para limparmos um registrador, poderíamos mover 0 (zero) para ele, mas como você já deve ter observado, uma operação de movimento de um número para um registrador exige duas instruções (**MOVLW** e **MOVWF**), e portanto, dois ciclos de máquina. A fim de agilizar este caso específico de atualização do registrador, existe o comando **CLRF**:

CLRF = Limpa (**CLR**) o registrador (F)

A sintaxe desta instrução é:

```
CLRF  f     ;em que f é o endereço da memória que se deseja limpar
```

Desta forma, as instruções:

```
MOVLW    B'00000000'
MOVWF    TRISB          ;DEFINE ENTRADAS E SAÍDAS DO PORTB
```

Poderiam ser substituídas por:

```
CLRF  TRISB         ;DEFINE ENTRADAS E SAÍDAS DO PORTB
```

De maneira análoga, existe um comando similar para a limpeza do work, apesar do fato de que para mover 0 (zero) para o work também só é necessária uma instrução. Entretando, as duas instruções que executam esta ação são ligeiramente diferentes, pois uma afeta o **STATUS** de zero e a outra não.

```
MOVLW 0x00          ;LIMPA O WORK SEM AFETAR O STATUS
CLRW                ;TAMBÉM LIMPA O WORK E AFETA O STATUS DE ZERO
```

• *INICIALIZANDO O SISTEMA*

Esta seção é destinada a uma melhor explicação e exemplificação da estrutura apresentada no arquivo-modelo.

DEFININDO LOCAIS PARA AS VARIÁVEIS

Seguindo a estrutura existente no nosso modelo, o primeiro bloco refere-se à definição das variáveis. Isto porque os SFRs já estão definidos no arquivo de include (vide P16F84.INC).

Desta forma, devemos criar espaço e nomes para todas as variáveis que utilizaremos no programa. Recomendamos também comentar para que serve cada uma delas. Vejamos um exemplo:

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

      CBLOCK 0x0C          ;ENDEREÇO INICIAL DA MEMÓRIA DE USUÁRIO

            W_TEMP         ;REGISTRADORES TEMPORÁRIOS PARA
            STATUS_TEMP    ; USO JUNTO ÀS INTERRUPÇÕES
            CONTADOR       ;CONTADOR PARA O NÚMERO DE TRANSMISSÕES
            BYTE_TX        ;BYTE QUE SERÁ TRANSMITIDO
            BYTE_RX        ;BYTE QUE SERÁ RECEBIDO

      ENDC                 ;FIM DO BLOCO DE MEMÓRIA
```

Neste exemplo, estamos usando as diretrizes CBLOCK e ENDC. Sem a utilização delas, o mesmo código seria escrito da seguinte maneira:

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

W_TEMP       EQU   0X0C   ;REGISTRADORES TEMPORÁRIOS PARA
STATUS_TEMP  EQU   0X0D   ;USO JUNTO ÀS INTERRUPÇÕES
CONTADOR     EQU   0X0E   ;CONTADOR PARA O NÚMERO DE TRANSMISSÕES
BYTE_TX      EQU   0X0F   ;BYTE QUE SERÁ TRANSMITIDO
BYTE_RX      EQU   0X10   ;BYTE QUE SERÁ RECEBIDO
```

Lembre-se sempre que a memória disponível ao usuário para o PIC 16F84 vai de 0x0C até 0x4F.

RESERVANDO ESPAÇO PARA FLAGS

Flags são bits que definimos dentro de um byte para serem utilizados como chaves on/off. Desta forma, em um único endereço de memória (registrador) poderemos guardar até 8 flags que registrarão 8 estados diferentes. Por exemplo, um flag pode marcar se um byte já foi transmitido ou não, outro pode marcar se existe algum dado recebido ou não, e assim por diante.

A primeira ação para podermos trabalhar com flags é a definição de um registrador onde eles serão armazenados. Se muitos flags forem necessários, mais de um registrador pode ser utilizado.

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

     CBLOCK 0x0C          ;ENDEREÇO INICIAL DA MEMÓRIA DE USUÁRIO

            W_TEMP        ;REGISTRADORES TEMPORÁRIOS PARA
            STATUS_TEMP   ;USO JUNTO ÀS INTERRUPÇÕES
            FLAG          ;REGISTRADOR PARA FLAGS
            CONTADOR      ;CONTADOR PARA O NÚMERO DE TRANSMISSÕES
            BYTE_TX       ;BYTE QUE SERÁ TRANSMITIDO
            BYTE_RX       ;BYTE QUE SERÁ RECEBIDO

     ENDC                 ;FIM DO BLOCO DE MEMÓRIA
```

Depois, fica mais fácil se definirmos nomes específicos para cada um dos flags por meio da diretriz **#DEFINE**:

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

#DEFINE    TRANSMITIDO   FLAG,0  ;FLAG PARA INFORMAR QUE O DADO
                                 ;FOI TRANSMITIDO:
                                 ; 1 -> TRANSMITIDO
                                 ; 0 -> NÃO TRANSMITIDO

#DEFINE    RECEBIDO      FLAG,1  ;FLAG PARA INFORMAR QUE O DADO
                                 ;FOI RECEBIDO:
                                 ; 1 -> RECEBIDO
                                 ; 0 -> NÃO RECEBIDO
```

Desta forma, o flag TRANSMITIDO fica armazenado no bit 0 do registrador FLAG, e o RECEBIDO fica no bit 1.

CRIANDO CONSTANTES

As constantes são muito úteis para simplificar alterações em valores do sistema. Por exemplo, imagine que seu sistema deva possuir vários delays de atraso durante a execução. Vamos supor que você crie todos esses delays baseados em uma constante de tempo. Se ao final do projeto você descobrir que esses delays devem ser alterados, não fica muito mais fácil modificar somente a constante, ao invés de editar todo o código?

Com a criação de constantes, você poderá substituir um número (literal) por um nome qualquer:

INICIALIZANDO AS VARIÁVEIS

É muito importante que as variáveis também sejam inicializadas, mesmo que seus valores devam ser iguais a zero, pois nunca se sabe como a memória do PIC acordará. Inicializando as variáveis corretamente, muitos problemas podem ser evitados.

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                    INICIALIZAÇÃO DAS VARIÁVEIS                        *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      CLRF   PORTA          ;DESLIGA TODAS AS SAÍDAS DO PORT A
      CLRF   PORTB          ;DESLIGA TODAS AS SAÍDAS DO PORT B
      MOVLW  .10
      MOVWF  CONTADOR       ;INICIA CONTADOR COM 10 (DECIMAL)
```

• *TRABALHANDO COM ROTINAS*

Como em muitas outras linguagens de programação, no assembler do PIC existem dois tipos de rotinas: as rotinas de desvio e as rotinas de chamada, que também podem ser consideradas funções. As rotinas de desvio nada mais são que "pulos" no programa por meio da instrução **GOTO**, exatamente como na linguagem BASIC. Acontece que no assembler do PIC o número da linha é substituído por um nome (label). Já as rotinas de chamadas são acessadas através da instrução **CALL**. Essa instrução possibilita que o próximo ponto do programa (PC+1) seja guardado na pilha (stack), para que o sistema possa retornar a ele mais tarde, por meio da instrução **RETURN** (ou similar) que é utilizada para encerrar a rotina.

Para tornar mais fácil a visualização desses dois tipos de rotina, veja o esquema seguinte:

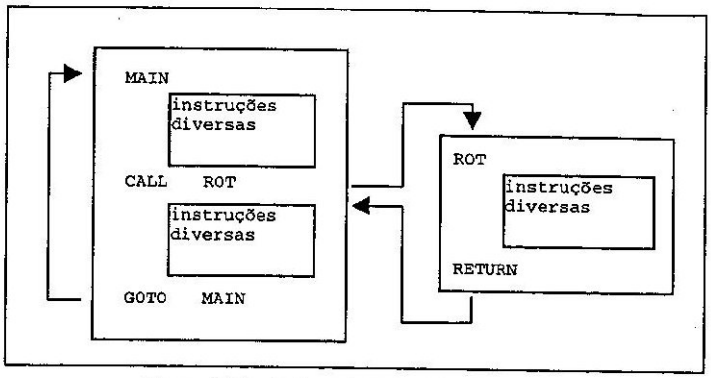

Figura 11.1 – Rotinas de Desvio e de Chamada.

ROTINAS DE DESVIO

As rotinas de desvio são utilizadas geralmente para deixar o programa mais estruturado e organizado. No entanto, a instrução **GOTO** é freqüentemente utilizada em qualquer programa para possibilitar "pulos" necessários à lógica do sistema. Na próxima seção (Tomando decisões e fazendo desvios), isto ficará muito mais claro.

A sintaxe dessa instrução é:

```
GOTO  nome    ;onde nome é a identificação do local para onde se deseja
              ;pular
```

Uma identificação de local nada mais é do que um nome localizado na primeira coluna de texto do arquivo fonte. Para facilitar sua vida, recomendamos que os nomes sejam escritos logo na primeira coluna, e as instruções sejam indentadas à direita, por meio de uma tabulação, como pode ser observado no arquivo-modelo. Os nomes de rotinas não podem possuir espaços. Como é muito importante que esses nomes tenham algum sentido em relação a sua aplicação, recomendamos que o sublinhado (_) seja utilizado para ajudar na sua composição. Por exemplo:

```
GOTO  BT_PRESS      ;pula para o local onde tratará o botão pressionado
GOTO  BT_LIB        ;pula para o local onde tratará o botão liberado
```

Vejamos agora uma dica muito interessante para quando precisamos executar "pulos" muito curtos. Por exemplo, imagine uma situação em que é necessário pular somente 2 ou 3 linhas, para cima ou para baixo. Poderíamos criar um nome para o local do pulo e utilizar o comando GOTO, como foi visto até agora, mas às vezes isso começa a se tornar um problema, pois você já não consegue mais inventar tantos nomes. Para estas, e para muitas outras situações que serão vistas adiante, é que o compilador possibilita o uso do $. Ele é utilizado para

representar a linha atual do programa. Veja então como fica o comando GOTO em conjunto com o $:

```
GOTO  $+3          ;pula 3 linhas para baixo
GOTO  $-2          ;pula 2 linhas para cima
```

Não recomendamos esta técnica para "pulos" muito grandes para não complicar o entendimento e a manutenção do programa.

- ***ROTINAS DE CHAMADA***

Já as rotinas de chamada são utilizadas quando uma tarefa deve ser repetida várias vezes e não se deseja rescrevê-la para não gastar memória de programa. Desta forma, a rotina pode ser usada como uma função, que é chamada de diversos pontos do programa sem gerar problemas. Por exemplo, se seu sistema precisa de um delay de atraso em diversos pontos do programa para funcionar corretamente, podemos criar uma rotina denominada DELAY e chamá-la quantas vezes forem necessárias, em quantos pontos precisarmos.

A instrução utilizada para chamar uma rotina desse tipo é a CALL.

A sintaxe dessa instrução é:

```
CALL  nome    ;onde nome é a identificação da rotina
```

As observações feitas aos nomes empregados com a instrução GOTO também são válidas para a instrução CALL.

A grande vantagem de utilizar uma rotina de chamada é que o endereço seguinte ao ponto de chamada (linha abaixo da instrução CALL) é armazenado na pilha (Stack). Quando terminarmos a rotina com a instrução RETURN, o sistema voltará exatamente para o endereço armazenado na pilha. Simples, não é mesmo? No entanto, vale lembrá-lo que outras rotinas podem ser chamadas de dentro da rotina atual, antes de darmos um RETURN. Isto irá gerar outros níveis de pilha com os respectivos endereços de retorno. Acontece que o PIC 16F84 possui uma pilha de no máximo 8 níveis. Se este limite for ultrapassado, o primeiro nível será sobrescrito, impossibilitando que o sistema retorne a todos os pontos de chamada, podendo gerar um grave erro no programa. Para os sistemas apresentados neste livro, e para muitos outros mais complexos, 8 níveis de pilha é mais que o suficiente, mas é sempre bom ficar alerta, principalmente porque existem PICs que só possuem 2 níveis de pilha.

Para retornar de uma rotina, devem ser utilizadas as instruções RETURN e RETLW.

As sintaxes dessas instruções são:

```
RETURN      ;finaliza a rotina voltando ao último endereço da pilha.
RETLW   k   ;finaliza a rotina voltando ao último endereço da pilha com
            ;o valor k (literal) em W.
```

Vale realçar que alguns modelos de PIC só possuem a instrução **RETLW**.

TOMANDO DECISÕES E FAZENDO DESVIOS

Esta seção irá apresentá-lo às instruções capazes de executar testes e tomar decisões dentro do PIC. Devido à filosofia RISC, o set de instruções é bem resumido, como já explicamos, e você verá que existem pouquíssimas instruções voltadas a esta finalidade. Aproveitaremos a ordem natural das coisas para mostrarmos também como alteramos diretamente bits e flags.

TESTANDO BITS E FLAGS (BTFSC E BTFSS)

As instruções empregadas para testar diretamente bits, que podem ser portas ou flags, são: **BTFSC** e **BTFSS**. Vamos utilizar o sistema de termos predefinidos para entendermos melhor suas funções:

BTFSC = Testa (**T**) o bit (**B**) do registrador (**F**) e pula (**S**) a próxima linha se a resposta for 0 (**C**).

BTFSS = Testa (**T**) o bit (**B**) do registrador (**F**) e pula (**S**) a próxima linha se a resposta for 1 (**S**).

As sintaxes dessas instruções são:

```
BTFSC f,b     ;em que f é o registrador e b o bit a ser testado
Linha1        ;passa por esta linha se o bit testado for 1.
Linha2        ;pula direto para esta linha se o bit testado for 0.

BTFSS f,b     ;em que f é o registrador e b o bit a ser testado
Linha1        ;passa por esta linha se o bit testado for 0.
Linha2        ;pula direto para esta linha se o bit testado for 1.
```

Normalmente, as especificações do registrador e do bit podem ser substituídas por um **#DEFINE**, como no exemplo abaixo:

```
#DEFINE    BOTAO PORTA,0    ;DEFINE O BOTÃO NO PINO RA0
                            ; 0 -> LIBERADO
                            ; 1 -> PRESSIONADO
BTFSS      BOTAO            ;O BOTÃO ESTÁ PRESSIONADO?
GOTO       BT_LIB           ;NÃO,  VAI TRATAR BOTÃO LIBERADO
GOTO       BT_PRES          ;SIM,  VAI TRATAR BOTÃO PRESSIONADO
```

Os mesmos testes podem ser executados com flags para checar um certo estado do sistema.

MUDANDO BITS E FLAGS (BSF E BCF)

Agora que você já sabe como testar se um dado bit está em 1 (um) ou em 0 (zero), e também já sabe como executar uma ação específica conforme o resultado obtido, então está na hora de aprender como alterar o valor dos bits.

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

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

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

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

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

     ORG    0x00   ;ENDEREÇO INICIAL DE PROCESSAMENTO
     GOTO   INICIO

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

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

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                    INÍCIO DO PROGRAMA                                 *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
INICIO
     BANK1                ;ALTERA PARA O BANCO 1
     MOVLW  B'00000100'
     MOVWF  TRISA         ;DEFINE RA2 COMO ENTRADA E DEMAIS
                          ;COMO SAÍDAS
     MOVLW  B'00000000'
     MOVWF  TRISB         ;DEFINE TODO O PORTB COMO SAÍDA
     MOVLW  B'10000000'
     MOVWF  OPTION_REG    ;PRESCALER 1:2 NO TMR0
                          ;PULL-UPS DESABILITADOS
```

```
                        ;AS DEMAIS CONFG. SÃO IRRELEVANTES
      MOVLW  B'00000000'
      MOVWF  INTCON     ;TODAS AS INTERRUPÇÕES DESLIGADAS
      BANK0             ;RETORNA PARA O BANCO 0
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*                    INICIALIZAÇÃO DAS VARIÁVEIS                      *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

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

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

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

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

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

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

      END               ;OBRIGATÓRIO
```

• *FAZENDO OPERAÇÕES ARITMÉTICAS BÁSICAS*

O próximo passo deve ser dado na direção dos cálculos. Basicamente, todos os programas implementados nos microcontroladores necessitam de algum tipo de conta para que sua lógica funcione corretamente. Neste capítulo, veremos que a matemática não é o ponto forte das instruções do PIC 16F84. Não diretamente, mas veremos também que ele possui todos os recursos necessários para que possamos implementar as nossas próprias funções para cálculos muito mais avançados; basta um pouco de conhecimento e muita criatividade. Bem, mas vamos começar pelos cálculos mais simples possíveis, as contas básicas de adição e subtração.

• SOMANDO (INCF, INCFSZ, ADDWF E ADDLW)

Para operações de adição, o assembler do PIC possui dois grupos de instruções, sendo um usado para adições unitárias e o outro para adições diversas. Dentro desses grupos, possuímos um total de quatro instruções as quais serão mostradas juntamente com suas descrições por meio da técnica de termos predefinidos.

INCF: Incremento unitário (INC) do registrador (F).

INCFSZ: Incremento (INC) do registrador (F) pulando a próxima linha (S) se o resultado for 0 (Z)

ADDWF: Soma (ADD) o valor de work (W) ao registrador (F).

ADDLW: Soma (ADD) um número (L) ao valor de work (W).

A sintaxe correta para cada uma dessas instruções pode ser vista em seguida:

```
INCF      f,d    ;em que f é o registrador e d o destino onde será
                 ;guardado o resultado da conta (f + 1 -> d)
INCFSZ    f,b    ;em que f é o registrador e d o destino onde será
                 ;guardado o resultado da conta (f + 1 -> d)
ADDWF     f,d    ;em que f é o registrador e d o destino onde será
                 ;guardado o resultado da conta (f + W -> d)
ADDLW     k      ;em que k é o número que será somado ao W
                 ;o resultado é mantido em W (W + k -> W)
```

A instrução INCF é utilizada normalmente para controlar contadores unitários dentro do sistema ou para funções específicas. Por exemplo, vamos olhar um código que chama uma rotina denominada BIP por oito vezes seguidas:

```
        CRLF    CONTA       ;ZERA O CONTADOR
LOOP
        CALL    BIP         ;CHAMA A ROTINA BIP QUE EMITIRÁ UM SOM
        INCF    CONTA,F     ;INCREMENTA O CONTADOR (CONTA = CONTA + 1)
        BTFSS   CONTA,3     ;TESTA O BIT NÚMERO 3 DO CONTADOR. QUANDO
                            ;ESTE BIT  FOR
                            ;IGUAL A 1, SIGNIFICA QUE CONTADOR = 8.
        GOTO    LOOP        ;CONTADOR AINDA NÃO É 8, RETORNA PARA LOOP
FIM                         ;CONTADOR = 8, ACABOU O EXEMPLO
```

Observe que nós checamos o valor do contador por meio da análise de um de seus bits. Está é uma maneira muito utilizada pelos programadores, mas também muito restrita, pois só permite comparações com os valores relacionados a cada um dos bits pela regra da potência de 2 (1, 2, 4, 8, 16, 32, 64 e 128). Na verdade, a comparação poderia ser feita em relação a qualquer valor, mas teríamos de utilizar instruções ainda não estudadas, por isso esta técnica foi empregada neste exemplo.

Já a instrução INCFSZ é bem mais poderosa, pois, além do incremento, ela faz também uma comparação para poder tomar uma decisão. Após cada incremento, ela verifica se o resultado é igual a zero. Bem, mas quando uma soma de um número

Observe que para esse tipo de contador a utilização da instrução **DECFSZ** no lugar da **INCFSZ** torna o programa mais fácil de entender.

As instruções **SUB** são utilizadas para subtrações diversas:

```
      MOVLW   .10
      MOVWF   NUM_1       ;INICIA NUM_1 COM 10
      MOVLW   .20
      MOVWF   NUM_2       ;INICIA NUM_2 COM 20
      CLRF    RESULTADO   ;INICIA RESULTADO COM 0
SUB1                      ;RESPOSTA = 30 - NUM_1
      MOVF    NUM_1,W     ;COLOCA O VALOR DE NUM_1 (10) E W
      SUBLW   .30         ;SUBTRAI W DE 30 (30 - 10 = 20)
      MOVWF   RESULTADO   ;COLOCA A RESPOSTA EM RESULTADO = 20
SUB2                      ;RESPOSTA = NUM_2 - NUM_1
      MOVF    NUM_1,W     ;COLOCA O VALOR DE NUM_1 (10) EM W
      SUBWF   NUM_2,W     ;SUBTRAI O VALOR DE W (10) DE NUM_2 (20) => W
      MOVWF   RESULTADO   ;COLOCA A RESPOSTA EM RESULTADO = 10
FIM                       ;FIM DO EXEMPLO
```

Aqui também podemos observar que, ao contrário das instruções de adição, a ordem dos fatores para a subtração afeta diretamente o resultado. Em uma soma, o resultado pode ser zero ou positivo, mas na subtração ele pode ser zero, positivo ou negativo. Por meio da análise do flag de carry podemos concluir qual o resultado correto da subtração:

Negativo: Sempre que o resultado da subtração for negativo, o carry será zero (0). Neste caso, o valor da resposta não será diretamente o número negativo, e sim a sua diferença para 256.

Positivo: Sempre que o resultado for positivo, o carry será um (1).

Zero: Sempre que o resultado for zero, o carry será um (1). Neste caso, o flag de zero também será um (1).

Vejamos então um exemplo para a subtração de dois números quaisquer: NUM_1 e NUM_2. O módulo do resultado será colocado em RESP e o flag NEG será setado sempre que o resultado for negativo:

```
      ;IMAGINEMOS QUE NUM_1 E NUM_2 ESTÃO COM VALORES QUAISQUER
      CLRF    RESP        ;LIMPA O REGISTRADOR ONDE SERÁ COLOCADA A RESPOSTA
SUB1                      ;NUM_2 - NUM_1 = RESP
      MOVF    NUM_1,W     ;MOVE O VALOR DE NUM_1 PARA W
      SUBWF   NUM_2,W     ;SUBTRAI O VALOR DE W (NUM_1) DE
                          ;NUM_2 GUARDANDO EM W
      BTFSS   STATUS,C    ;TESTA CARRY. RESULTADO NEGATIVO?
      GOTO    TRATA_NEG   ;SIM, PULA PARA TRATAR RESULTADO NEGATIVO
                          ;NÃO, RESULTADO POSITIVO OU ZERO
      MOVWF   RESP        ;COLOCA O RESULTADO DIRETAMENTE EM RESP
      BCF     NEG         ;LIMPA FLAG DE NÚMERO NEGATIVO
      GOTO    FIM         ;FINALIZA
```

```
TRATA_NEG               ;COMO O RESULTADO FOI NEGATIVO, ENTÃO RESP = 256 - W
                        ;COMO O NÚMERO MÁXIMO PARA 8 BITS É 255, ENTÃO
                        ;256 -> 0
     SUBLW  .0          ;0 - W -> W
     MOVWF  RESP        ;COLOCA O RESULTADO EM RESP
     BSF    NEG         ;SETA O FLAG DE NÚMERO NEGATIVO
FIM                     ;FIM DO EXEMPLO, COM O MÓDULO DO RESULTADO EM RESP
```

As instruções SUB afetam também os flags DC e Z do registrador STATUS.

- ***AS COMPARAÇÕES MAIOR QUE, MENOR QUE E IGUAL:***

Como acabamos de ver, as instruções SUB afetam diretamente o flag de carry, e por meio dele podemos identificar se o resultado foi negativo, positivo ou zero. Desta forma podemos identificar também se um número é maior, menor ou igual a outro. Para isso, utilizaremos a instruções SUBWF conforme o exemplo:

```
     ;IMAGINEMOS QUE NUM_1 E NUM_2 SÃO DOIS REGISTRADORES COM VALORES QUAISQUER

COMPARA1
     MOVF   NUM_1,W     ;MOVE O VALOR DE NUM_1 PARA W
     SUBWF  NUM_2,W     ;SUBTRAI O VALOR DE W (NUM_1) DE
                        ;NUM_2 GUARD. EM W
     BTFSS  STATUS,C    ;TESTA CARRY. RESULTADO NEGATIVO?
     GOTO   RESP1       ;SIM, ENTÃO NUM_2 < NUM_1 (NUM_1 > NUM_2)
     GOTO   RESP2       ;NÃO, ENTÃO NUM_2 >= NUM_1 (NUM_1 <= NUM_2)
FIM

     ;IMAGINEMOS AGORA QUE NUM_1 É UM NÚMERO QUALQUER (LITERAL)
     ;E NUM_2 É UM REGISTRADOR COM VALOR QUALQUER

COMPARA2
     MOVLW  NUM_1       ;MOVE O NÚMERO NUM_1 PARA W
     SUBWF  NUM_2,W     ;SUBTRAI O VALOR DE W (NUM_1) DE NUM_2
                        ;GUARD. EM W
     BTFSS  STATUS,C    ;TESTA CARRY. RESULTADO NEGATIVO?
     GOTO   RESP1       ;SIM, ENTÃO NUM_2 < NUM_1 (NUM_1 > NUM_2)
     GOTO   RESP2       ;NÃO, ENTÃO NUM_2 >= NUM_1 (NUM_1 <= NUM_2)
FIM
```

MULTIPLICANDO (RLF)

Não existem instruções específicas de multiplicação dentro da linguagem do PIC 16F84. No entanto, elas podem ser "construídas" utilizando as instruções já vistas. Além disso, uma nova instrução, a RLF, ajudará muito na hora de multiplicar um número. Vamos dar uma olhada na descrição desta instrução:

RLF: Rotaciona (R) um bit para a esquerda (L) do registrador (F).

A sintaxe dessa instrução é a seguinte:

```
RLF         f,d     ;em que f é o registrador e d o destino onde será
                    ;guardado o resultado da rotação
```

Puxa, mas o que a rotação tem a ver com a multiplicação? O que esta instrução faz realmente? Na verdade, ela simplesmente desloca todos os bits do registrador para a esquerda, colocando o valor de carry no bit zero e depois "jogando" o valor do bit 7 em carry. Veja o próximo esquema para uma melhor compreensão:

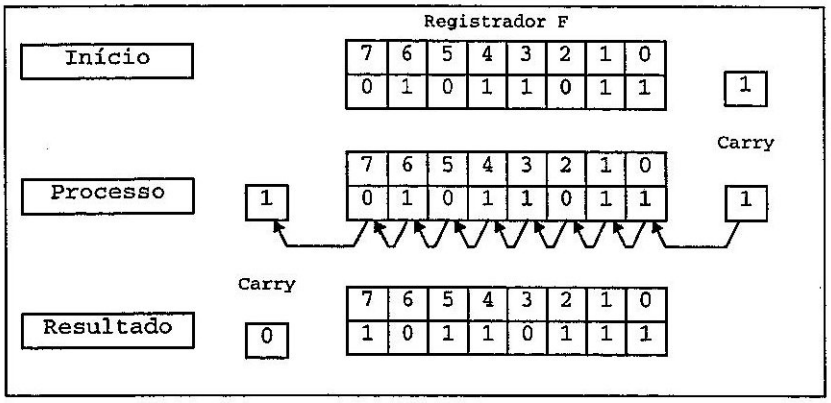

Figura 11.3 – Rotação de 1 Bit para a Esquerda.

Acontece que, matematicamente falando, se garantirmos que o bit que entra na posição menos significativa (bit0) seja zero, então cada vez que rotacionamos o byte para a esquerda, estamos multiplicando seu valor por 2. Desta forma, se o byte for rotacionado três vezes, seu valor será multiplicado por 8 (2x2x2). Esta é uma forma simples e rápida de multiplicação, mas só podemos efetuar contas com potências de 2. Neste caso, assim como nas contas de adição, o resultado da multiplicação pode precisar de mais de 8 bits.

O próximo exemplo mostra um registrador sendo multiplicado por oito:

```
;IMAGINEMOS QUE NUM_1 É UM REGISTRADOR COM VALOR QUALQUER

      CLRF   BYTE_BAIXO    ;LIMPA O REGISTRADOR BYTE_BAIXO
      CLRF   BYTE_ALTO     ;LIMPA O REGISTRADOR BYTE_ALTO
MULT8
      BCF    STATUS,C      ;LIMPA O CARRY
      RLF    NUM_1,F       ;MULTIPLCA POR 2 (NUM_1 = NUM_1 X 2)
      RLF    BYTE_ALTO,F   ;ROTACIONA BYTE_ALTO PARA PEGAR O BYTE "PERDIDO"
                           ;NA CONTA
      RLF    NUM_1,F       ;MULTIPLCA POR 2 (NUM_1 = NUM_1 X 4)
      RLF    BYTE_ALTO,F   ;ROTACIONA BYTE_ALTO PARA PEGAR O BYTE "PERDIDO"
                           ;NA CONTA
      RLF    NUM_1,F       ;MULTIPLCA POR 2 (NUM_1 = NUM_1 X 8)
```