COLOR EDTASM

Editor Assembler para CP400

INDICE

PARTE I
USANDO COLOR EDTASM

PARTE II
LINGUAGEM ASSEMBLER

PARTE III
RESUMOS

1 - Introdução

BASIC X ASSEMBLER

Os microcomputadores TRS-80 COLOR e seus compatíveis vêm equipados com o Microprocessador 6809. Ele tem um conjunto próprio de instruções em linguagem de máquina, que é a única linguagem que ele pode entender. Naturalmente você pode se comunicar com ele ou programá-lo usando a linguagem BASIC. Isto acontece porque na ROM do equipamento há um interpretador BASIC, que traduz todas as instruções e comandos BASIC para a linguagem "nativa" do processador. Para cada instrução Basic muitas tarefas são executadas pelo interpretador. Isto implica em muito tempo gasto ineficientemente. Em muitos casos onde velocidade e eficiência são fatores críticos para a funcionalidade do programa surge a necessidade de se programar em linguagem de máquina, escapando assim das limitações da quantidade restrita de instruções BASIC e de sua ineficiência em termos de velocidade.

COLOR EDTASM

COLOR EDTASM é uma ferramenta poderosa para o desenvolvimento, montagem e depuração de programas em linguagem de máquina. Nele você conta com três sub-sistemas interligados: Editor, Assembler e ZBUG. O primeiro passo é escrever o programa em linguagem Assembler usando mnemônicos. Para tanto você usará o Editor e o resultado será o código-fonte do programa.
A segunda etapa é "assemblar" ou montar o programa. Esta é a função do Assembler, que transforma o texto do programa-fonte em linguagem de máquina. Este programa já pode ser armazenado na memória ou em cassette.
Na última etapa, que infelizmente é quase sempre necessária, você deverá testar e corrigir seu programa até que ele esteja perfeito. Para isso você conta com ZBUG, um monitor que lhe dará acesso total à memória e facilitará muito sua tarefa.

O Manual

Este manual não se destina a ensinar a programação Assembler. Ele simplesmente explica como usar COLOR EDTASM e fornece algumas informações básicas sobre as instruções 6809.

Para os principiantes será necessário o estudo em outras fontes para se tornar um programador Assembler 6809. Podemos recomendar estes livros:

TRS-80 COLOR COMPUTER
Assembly Language Programming
William Barden Jr.
Radio Shack

Assembly Language Graphics for
the TRS-80 Color Computer
Don Inman - Kurt Inman with Dyeax
Reward Books

6309 Assembly Language Programming
Lance Leventhal
Osborne / McGraw - Hill

Carregamento

Para computadores com 64k de memória. Coloque a fita no gravador e pressione a tecla PLAY. Tecle o comando CLOADM. Após o carregamento tecle o comando EXEC. A seguinte mensagem surgirá na tela:

```
COLOR EDTASM
PEEK & POKE

*
```

O asterisco indica que você está no Editor.

2- MEMORIA

Para usar COLOR EDTASM você deve compreender como funciona a memória de seu computador e como EDTASM aloca seus buffers e informações internas. Há um mapa da memória no manual do computador e um incluído neste manual que explica as áreas reservadas e onde você pode colocar seus programas.
Com ZBUG podemos examinar a memória "por dentro". Para passar ao sub-sistema ZBUG tecle:

Z <ENTER>

Agora você verá o símbolo #, indicando que você está no nível de comando de ZBUG. Todos os comandos devem ser teclados após este símbolo e para voltar ao nível de comando você deve teclar (BREAK) ou (ENTER).

Exame de posições

O micro processador pode acessar 65536 endereços de memória, numerados de 0 a 65535 (base decimal) ou 0000-FFFF (hexadecimal). Ao entrar em ZBUG estaremos na base hexadecimal e iremos examinar a posição C000, o começo do COLOR EDTASM. Tecle:

C000/

LDA 86 é a instrução mnemônica presente neste endereço. Para examinar as próximas instruções use a seta para baixo. Use a seta para cima para voltar à posição imediatamente anterior. Note que com a seta para baixo você avança mais de um byte de cada vez. Isto ocorre por que neste modo de exame estamos analisando as instruções, que podem ter de um a cinco bytes. Já a seta para cima retrocede apenas um byte, independente do modo de exame utilizado.

Modos de Exame

Há quatro modos de exame para se interpretar a memória:

Modo Byte

Tecle (BREAK) para voltar ao nível de comando e depois tecle:

B <ENTER>

Examine o conteudo do endereço C000 novamente. A instrução LDA 86 está agora representada pelo número 86, que é o código de LDA e o operando 6 está na posição C001.
O conteúdo de todos os bytes da memória é representado por um número no modo Byte de exame, independente do endereço conter instruções, dados ou texto. A seta para baixo incrementa o endereço de um em um.

Modo Word

Volte ao nível de comando e tecle:

W <ENTER>

Uma "word" é o mesmo que um par de bytes. Pressionando-se a seta para baixo pode-se examinar os pares de bytes seguintes. Note que o incremento é igual a dois.

Modo ASCII

Agora tecle:

A <ENTER>

Neste modo ZBUG tentará interpretar as informações como código ASCII. Se o conteúdo do endereço for um número entre 21 e 7F (base hexadecimal) você verá o caracter que o código representa. Caso contrário nada será exibido. Examine os endereços a partir de C056, que contém o título do programa.
Os números de A1 até FF terão uma representação de tela que deve ser ignorada pois não corresponde ao verdadeiro código ASCII.

Modo Mnemônico

Este é o modo "default" de exame, ou seja a menos que se determine outro modo todo exame será realizado desta maneira. Para voltar a este modo tecle:

M <ENTER>

Quando este modo é utilizado espera-se encontrar instruções em linguagem de máquina. ZBUG examina de 1 a 5 bytes de cada vez, "disassemblando" os números e mostrando as instruções mnemônicas que eles representam. O número 8606 dos endereços C000 e C001 foi disassemblado para LDA #6; B70CFF (dos endereços C002 a C004) representa STA >FF e assim por diante.

Os mnemonicos são usados para tornar mais fácil a memorização e o entendimento das instruções, que em sua forma numérica são extremamente difíceis de interpretar e utilizar.
Comece o disassembly em um ponto diferente agora. Tecle:

```
<BREAK>
C001/
```

Use a seta para baixo várias vezes. Você verá um disassembly diferente desta vez. Embora o conteúdo da memória não tenha mudado a interpretação mudou por que não começou no byte correto. Algumas vezes um byte pode ser representado por dois pontos de interrogação, significando que ZBUG não conseguiu interpretar aquela instrução. O disassembly deve ter começado no lugar errado ou aquele endereço simplesmente não contem instrução nenhuma e sim dados ou texto.
Assim, para se obter um disassembly correto deve-se conhecer previamente o ponto de entrada correto da rotina ou programa em linguagem de máquina.

MUDAR A MEMORIA

Enquanto se está examinando a memória, o cursor está sempre à direita da interpretação do conteúdo. Para se mudar o conteúdo do endereço basta teclar o novo conteúdo. Após teclar o novo dado tecle <ENTER> ou seta para baixo e a mudança será efetuada.
Para não prejudicar o funcionamento do próprio EDTASM não devemos modificar memória acima do endereço 7EE0 (hexadecimal) nem abaixo de E00. Passe ao modo Byte e abra o endereço 1000 teclando:

```
<BREAK>
B <ENTER>
1000/
```

Note que o cursor está a direita do conteúdo atual da posição. Agora tecle o novo conteúdo: 1 e <ENTER>. O endereço agora contem o número 1.
Você também pode mudar o conteúdo de uma posição usando a seta para baixo ao invés de <ENTER> e assim você já está pronto para modificar a próxima posição. Use a seta para cima para verificar que a mudança foi realmente efetuada.
O tamanho da mudança vai variar de acordo com o modo de exame que está sendo utilizado.
No modo byte você estará modificando um byte de cada vez e pode teclar 1 ou 2 dígitos, que podem ser precedidos de zero.

Alguns números hexadecimais podem ser também o nome de um registrador do 6809 (A,B,D,CC) e ZBUG não os aceitará, avisando-o com uma mensagem de êrro. Para evitar este problema tecle esses números precedidos de zero.
No modo word você estará mudando 2 bytes (1 word) de cada vez. Você pode teclar de 1 a 4 dígitos e este número será o novo valor da word.
No modo ASCII você pode teclar um número ou se preferir teclar diretamente a letra ou símbolo precedê-lo de apóstrofe (<SHIFT> 7). Por exemplo, para escrever a letra D na posição 1000 escolha o modo ASCII de exame (teclando A e <ENTER>) e tecle:

```
1000/
'D
```

Efetue a mudança com seta para baixo. Retroceda uma posição (usando a seta para cima) e você poderá verificar que o novo conteúdo do endereço 1000 é a letra D.
Se você estiver no modo mnemônico ZBUG espera que você mude de 1 a 5 bytes, dependendo da instrução que você deseja colocar no endereço. Contudo a mudança deve ser feita byte a byte numericamente. Este procedimento é mais complexo por que você não pode entrar com mnemônicos, devendo usar os códigos correspondentes, o que é bastante trabalhoso.
Por exemplo, entre no modo mnemônico e abra o endereço 1000 teclando:

```
<BREAK>
M <ENTER>
1000/
86 <ENTER>
1001/
26 <ENTER>
```

Assim você abriu o endereço 1000 e teclou o novo conteúdo (86), que é o código da instrução LDA e depois no endereço seguinte o operando 26. Examinando agora o endereço 1000 você verá que a instrução contida será LDA #26.
Agora que você já viu os vários modos de exame e modificação da memória pode fazer alguns exercícios, usando o Mapa de Memória e lembrando-se de não modificar as posições acima de 7EE0 e abaixo de 800. Caso isso ocorra você poderá perder o controle do programa e ser obrigado a desligar o computador e carregá-lo novamente.

3 - PROGRAMANDO

Para escrever programas em Assembler você usará o Editor. Após carregar e executar EDTASM você já está no Editor. Para passar do ZBUG para o Editor tecle E <ENTER>. O asterisco mostra que você já está no subsistema Editor.
O Editor dispõe de uma série de comandos para auxiliá-lo na tarefa de escrever e editar seu programa. Para acioná-los você deve estar no nível de comando (indicado pelo asterisco). Para voltar ao nível de comando tecle <BREAK>.

Exemplo de Programa

Vamos começar com um pequeno programa de exemplo que será usado em todo o manual para explicar os vários comandos.
Para começar tecle:

I <ENTER>

I é o comando de inserção. Mesmo que você ainda não tenha teclado nenhuma linha use-se este comando para começar o programa.
O Editor responderá com um número de linha, que serve apenas para organizar melhor seu programa enquanto você estiver no Editor. Estes números não são usados no seu programa em linguagem de máquina nem podem ser usados como referência em outras partes de seu programa.
Para fazer linhas de comentário tecle um asterisco (vale como REM em um programa BASIC). Tecle esta linha por exemplo:

00100 *LINHA DE COMENTARIO <ENTER>

O Assembler ignorará as linhas de comentário, que não afetarão o programa em linguagem de máquina. Elas servem somente melhorar o entendimento por parte do programador das linhas ou partes do programa.
Agora volte ao nível de comando teclando <BREAK> e delete esta linha teclando:

D100 <ENTER>

Nas linhas de programa há quatro campos distintos: símbolo, comando, operando e comentário. Para passar de um campo ao próximo você pode usar a seta à direita, que funciona como tabulador.
Tecle a seguinte linha, usando a seta para passar de um campo ao próximo:

00100 SIMBOL CMD OPERAND COMENTARIO

O símbolo, comando e operando devem ser terminados com a seta, espaço ou <ENTER>. O comentário é opcional. A linha não deve exceder 128 caracteres. Note que linhas longas irão exceder a largura do vídeo e continuarão na linha de baixo.
O símbolo e o operando não aparecem em todas as linhas, dependendo da instrução usada e não é raro encontrar linhas que só tenham o comando. Contudo cada campo tem sua função específica e não se pode colocar uma instrução no campo dos símbolos mesmo que este esteja desocupado.
Você deve agora deletar todas as linhas que tenha teclado para poder entrar com nosso programa de exemplo. Caso queira poderá omitir os trechos de comentário:

```
00100 START LDA  #$0F9  PEGA CHR ASCII
00110       LDX  #$500  METADE VIDEO
00120 TELA  STA  ,X+    POE CHR NA TELA
00130       CMPX #$5FF  FIM DA TELA?
00140       BNE  TELA   SE NAO REPETE
00150 FIM   SWI
00160       END
```

Este programa preencherá a metade inferior do vídeo (500 a 5FF) com o caracter gráfico número F9. O cifrão ($) indica um número hexadecimal. Sem este símbolo o Editor pressupõe que o número seja decimal. (Note que o default do Editor é decimal enquanto o default do ZBUG é hexadecimal).
Estude a explicação de todos os símbolos na Parte 2

Comando Write
W nome

Para gravar o programa em fita cassette antes de fazer quaisquer mudanças experimentais tecle:

W EXEMPLO <ENTER>

Surgirá a mensagem "LIGUE CASSETTE". Quando o gravador estiver pronto para gravar (fita limpa posicionada, tecla RECORD pressionada) tecle <ENTER> e o programa será gravado.
Se você não fornecer nenhum nome no comando o programa será gravado com o nome "NONAME". Os nomes podem ter até 8 caracteres e devem começar com uma letra.
Recomenda-se sempre fazer uma cópia do programa antes de executá-lo. Como o uso da memória com a linguagem de máquina é muito mais dinâmico e flexível do que em BASIC um pequeno defeito em seu programa pode apagar todo o buffer de edição

(onde está armazenado o texto do programa) e até mesmo destruir o próprio EDTASM. Em menos de um segundo o trabalho de horas pode ter sido destruído.
Após gravar o programa é uma boa idéia verificar se a gravação foi bem sucedida usando-se o comando V. Este comando verifica o checksum da gravação e assegura que não há erros de I/O.

Comando Load
L nome

Para carregar o programa fonte tecle:

L EXEMPLO <ENTER>

Você deverá ligar o cassette com a fita na posição correta e teclar <ENTER>. Assim começará a pesquisa até encontrar o arquivo EXEMPLO. Para carregar o primeiro arquivo encontrado você pode teclar simplesmente L, omitindo o nome.
Atenção: Este comando é usado para carregar programas fonte gravados em forma de texto. (Os programas já assemblados ou programas objeto gravados em linguagem de máquina devem ser carregados pelo comando CLOADM do BASIC, ou através de Z80D).
O Editor não esvazia automáticamente o buffer de edição antes de executar o load. Se já houver um programa na memória, o programa a ser carregado será acrescentado ao fim do que estiver na memória. Isto pode ser muito útil no caso de programas longos divididos em partes.
Caso você não deseje encadear os programas deverá deletar todas as linhas do programa na memória antes de usar o comando Load. Pode-se fazê-lo com o comando D#:*, onde # representa a primeira linha e * a última.

Comando Print
P faixa

Para imprimir na tela uma linha do programa tecle:

P100 <ENTER>

Para imprimir mais de uma linha tecle:

P100:130 <ENTER>

Como a linha atual, a última e a primeira são frequentemente chamadas, você pode referir-se a elas com um só caracter:

primeira linha

* última linha
. linha atual (a última linha que você inseriu ou imprimiu)

Para imprimir todo o programa tecle:

P#:* <ENTER>

Assim como você usa : (dois pontos) entre a primeira e última linhas de uma faixa, você pode usar ! (ponto de exclamação) para determinar quantas linhas deverão ser impressas a partir de uma determinada linha. Tecle:

P#!5 <ENTER>

e cinco linhas a partir da primeira serão impressas.
Para interromper uma listagem no meio tecle:

<SHIFT> @

Para continuar a listagem pressione qualquer tecla.

Comandos de Impressora

H faixa
T faixa

Se você tiver uma impressora poderá usar estes comandos para obter listagens impressas. Estes comandos funcionam com os mesmos parâmetros do comando P. A diferença entre H e T é que com o comando T os números de linha não serão impressos.

Comando Edit

E linha

Você pode editar linhas da mesma maneira que edita linhas de programas BASIC. Por exemplo para editar a linha 100, tecle:

E100 <ENTER>

A nova linha 100 está abaixo da antiga linha 100, pronta para ser modificada. Tecle a barra de espaços para posicionar o cursor imediatamente após a palavra START e tecle este subcomando de inserção:

I0 <ENTER>

que insere a letra O ao fim de START.
Há vários subcomandos de edição que você poderá estudar no Apêndice A.

Comando Delete
D faixa

Se você estiver usando o programa de exemplo certifique-se que ele já está gravado antes de experimentar este comando. Tecle:

D110:140 <ENTER>

As linhas de 110 a 140 foram eliminadas.

Comando Insere
I linha inicial,inc

Tecle:

I 152,2 <ENTER>

Você pode agora inserir novas linhas a partir da linha 152. Os números de linha serão incrementados de dois em dois. O Editor não permitirá que acidentalmente você escreva por cima de linhas existentes. Ao chegar à linha 160 você será alertado por uma mensagem de êrro.
Tecle <BREAK> para voltar ao nível de comando e então tecle:

I 170 <ENTER>

Assim você poderá inserir novas linhas no fim do programa. O incremento será igual a dois pois este foi o último incremento utilizado.
Tecle:

<BREAK>
I <ENTER>

e o Editor começará a inserir linhas a partir da linha atual.
Ao se entrar no Editor a linha atual será a linha 100 e o incremento igual a 10.
Você pode usar qualquer número de linha entre 0 e 63999.

Comando Renumera
N linha inicial, inc

Este comando também ajuda na inserção de novas linhas. Tecle:

N 100,50 <ENTER>

Agora as linhas começam com 100 e vão incrementando de 50 em 50. Assim cria-se espaço para a inserção de linhas.
Omitindo-se os parâmetros, teclando-se apenas N, o número da linha atual passa a ser o número de linha inicial. Tecle:

N100,10 <ENTER>

Comando Replace
R linha inicial, inc

Este comando é uma variação do comando de inserção. Tecle:

R 100,3 <ENTER>

Você agora pode substituir a linha 100 por uma nova linha e já pode seguir inserindo linhas cujos números incrementarão numa razão de 3.

Comando Copiar
C linha inicial, faixa, inc

O comando de cópia pode economizar muito trabalho duplicando qualquer parte de seu programa em outro local. Tecle:

C 500,100:150,10 <ENTER>

Este comando copiará a faixa de linhas entre 100 e 150 para um novo lugar começando na linha 500 com um incremento de 10. Qualquer tentativa de cópia sobre linhas existentes será interrompida.

Comando ZBUG

Para sair do Editor e entrar para ZBUG tecle:

Z <ENTER>

Ao invés do asterisco (*) você verá o sinal # que indica que você está no nível de comando de ZBUG. Para voltar ao Editor tecle o comando: E <ENTER>
Note que você não perde o programa fonte, que continua no buffer de edição.

Comando BASIC

Para passar ao BASIC a partir do EDITOR tecle:

Q <ENTER>

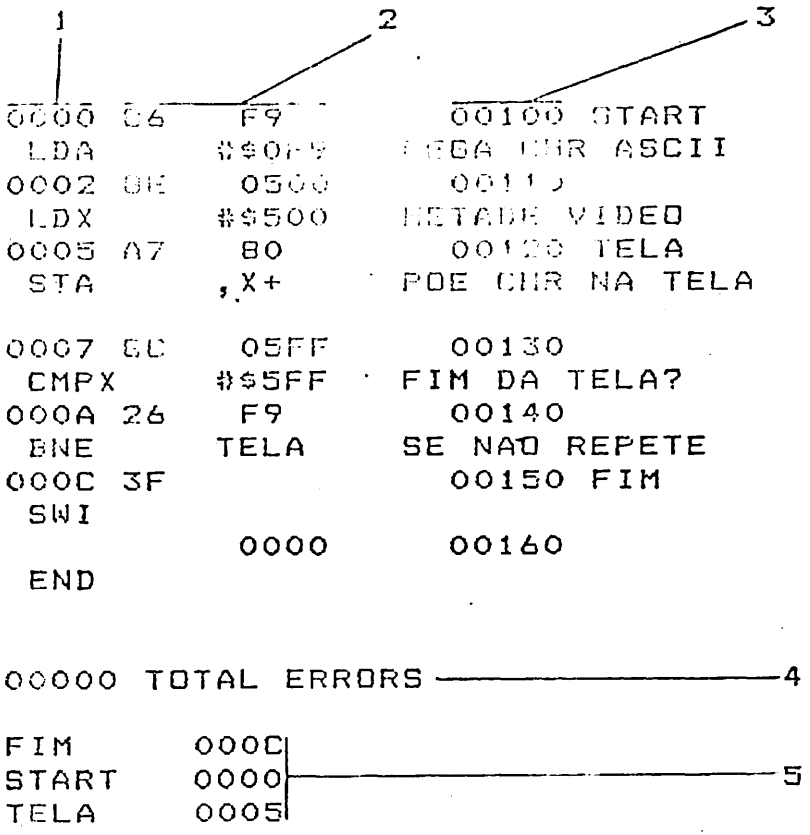
Para voltar ao Editor estando em BASIC tecle um dos comandos:

EXEC &H[illegible] 0800
EXEC [illegible] 2048

Este é o endereço inicializador de EDTASM.
Ao voltar para o Editor você terá perdido seu programa fonte pois ao passar ao BASIC perde-se o buffer de edição. Similarmente, se você tiver algum programa BASIC na memória, ao entrar para o Editor este programa se perderá.

Dicas de Programação

Copie alguns programas de revistas ou livros e vá mudando algumas instruções (uma de cada vez) para aprender como as várias instruções e modos de endereçamento funcionam.
Ao escrever programas longos tente estruturá-lo de maneira que haja muitas subrotinas, o que facilitará o entendimento e a correção de eventuais erros.
Nota: Você pode usar o Editor para escrever também seus programas BASIC pois o formato usado pelo comando WRITE é reconhecido pelo comando CLOAD do BASIC. Há uma série de vantagens como a numeração automática de linhas, o comando Copiar e as facilidades de edição. Experimente!



1 A posição de memória onde o código assemblado será guardado. Neste caso o código relativo à instrução LDA #$0F9 será guardado no endereço 0000.

2 O código assemblado da instrução. 86F9 é o código da instrução LDA #$0F9.

3 O número da linha.

4 Número de erros. Se houver algum erro vale a pena fazer o assembly novamente usando a opção /WE.

5 Os símbolos usados no programa e os endereços correspondentes.

FIGURA 1

4 - ASSEMBLAR

O comando para transformar o código fonte (o texto de seu programa) em linguagem de máquina é muito simples. Estando no nível de comando do Editor basta teclar:

A NOME <ENTER>

Se seu programa fonte estiver na memória surgirá a mensagem para ligar o cassette e quando você teclar <ENTER> o programa em linguagem de máquina será gravado na fita para posteriormente ser carregado através do comando CLOADM. O Assembler produzirá uma listagem na tela explicando o que ele está fazendo. (ver figura 1)
Esta é a forma mais simples de assemblar o programa, mas não será a primeira a ser usada. Antes de gravar o programa deve-se ter certeza absoluta de que ele está totalmente livre de erros.
Há uma série de opções a serem usadas em conjunto com o comando A que facilitam muito os assemblies experimentais de programas. Veja alguns exemplos:

A/IM/WE
A/WE/LP/NS
A TESTE/LP

/WE (Wait on Error)
Pausa em erro

Esta é uma opção muito útil que faz uma pausa na listagem em casos de erros de sintaxe ou construção detectados pelo Assembler. Use qualquer tecla para continuar.
Note que um assembly livre de erros não significa que o programa esteja perfeito. Pode haver erros de lógica que, embora não sejam detectados pelo Assembler, serão fatais na hora da execução.

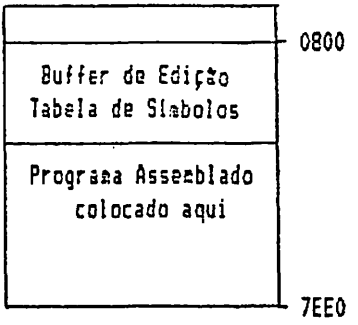
/SS /NO /NS /NL /LP

Estas opções modificam a forma de listagem.

/SS	Listagem compacta
/NO	Não produz código objeto
/NS	Não lista os símbolos
/NL	Não faz a listagem
/LP	Listagem passa à impressora

/IM (In Memory)
Assembly na Memória

O programa será assemblado na memória ao invés de passar ao cassette. Normalmente usado em assemblies experimentais.
Se usado não combinado com outras opções, o Assembler irá colocar o programa logo após a tabela de símbolos, que por sua vez está alocada logo após o buffer de edição.



Topo da Memória

O buffer de edição é a área onde está o texto de seu programa. Começa no endereço hexadecimal 0800. Não tem endereço fixo para terminar, que depende do tamanho do programa.
Logo após o buffer vem a tabela de símbolos que lista todos os símbolos do programa e seus respectivos valores ou endereços. O tamanho da tabela também varia de acordo com o programa e com o número de símbolos utilizados.
Faça agora o assembly do programa de exemplo "in memory". Certifique-se que o programa está em sua forma original e então tecle:

```
A/IM
```

Para examinar a listagem mais atentamente repita a operação usando <SHIFT> @ para fazer uma pausa na listagem. Use qualquer tecla para continuar.
Passando para o subsistema ZBUG, você poderá ver o buffer de edição e a tabela de símbolos (usando o modo ASCII de exame) e o programa em si (pelo modo mnemônico de exame). Como o programa de exemplo começa com o símbolo START você poderá encontrá-lo facilmente.

/AO (Absolute Origin)
Origem Absoluta

Esta opção permite que você determine exatamente onde o programa deve residir na

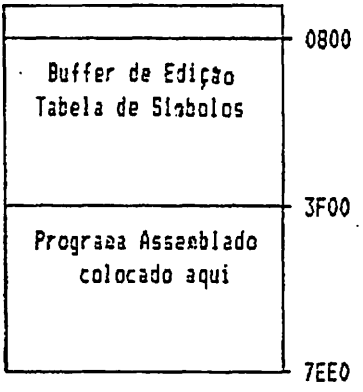
memória. Para poder usar esta opção você deve começar seu programa com uma instrução ORG.
Insira esta linha no início do programa:

```
00050          ORG      $3F00
```

Agora tecle:

```
A/IM/AO
```

Usando o monitor ZBUG você poderá verificar que agora o programa assemblado começa em 3F00:



Topo da Memória

Como você pode verificar a opção AO modificou o endereço de origem do programa mas não o local do buffer de edição ou da tabela de símbolos.

/MO (Manual Origin)
Origem Manual

A opção de origem manual oferece o máximo de controle sobre assemblies na memória, contudo não deve ser tentada por principiantes por implicar em procedimentos mais complexos.
Esta operação envolve o conteúdo de dois endereços de memória:

USRORG que contem o endereço inicial do programa assemblado

BEGTEMP que contem 0600, endereço base a partir do qual é calculada a origem do buffer de edição e da tabela de símbolos. (800-200)

Ao se alterar manualmente o conteúdo destes endereços você poderá fixar os endereços do programa, do buffer e da tabela de símbolos. Esta operação se mostra útil quando seu programa tiver que alterar endereços na área alocada

normalmente ao buffer e à tabela, por exemplo, quando ali houver uma página de gráficos em alta resolução.

Para mudar o conteúdo destas posições você deve passar ao ZBUG. Esta alteração fará com que se perca o conteúdo do buffer de edição, portanto o primeiro passo é gravar o programa. Depois você pode passar para ZBUG e determinar USRORG e BEGTEMP.

Ajustar USRORG

Normalmente a posição 0FD indica o topo da memória RAM, 7FFF. Neste exemplo vamos mudá-la para 2F00. Tecle:

```
FD/
2F00 <ENTER>
```

Agora as posições acima de 2F00 estão protegidas contra a ação do Editor e podem ser usadas por seu programa.

Ajustar BEGTEMP

Originalmente, a posição 0FF contem 600. Neste exemplo vamos modificar para 2000, assim criamos espaço na área abaixo deste endereço, que poderá ser usado para dados ou gráficos de alta resolução. Tecle:

```
FF/
2000 <ENTER>
```

O endereço colocado em BEGTEMP deve ser:
- Um limite de página de memória (hexadecimal terminado em 00)
- Maior que 600
- Pelo menos 300 bytes menor que USRORG

Assemblar o Programa:

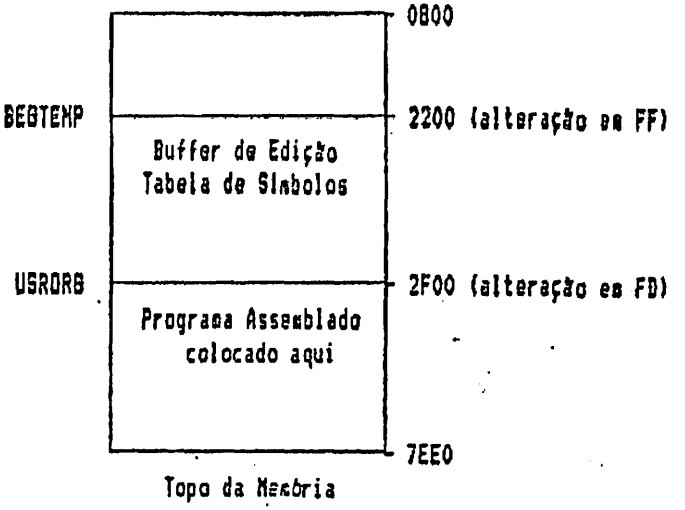
Para voltar ao Editor tecle:

```
G C006  <ENTER>
```

Carregue o programa de exemplo e se você inseriu a instrução ORG delete-a. Agora tecle:

```
A/IM/MO
```

Assim você estará assemblando o programa conforme os endereços que você determinou em USRORG e BEGTEMP. Caso você tenha seguido as instruções corretamente a imagem da memória neste momento será esta:



/NO (No Object Code)
Sem código objeto

Use esta opção quando não desejar mandar código objeto nem para a memória nem para o cassette.

Dicas para Assemblar

Use algum simbolo para indicar o inicio de seu programa.
Use a instrução ORG apenas com a opção /AO. Se usado com outras opções ORG não será o endereço de origem, mas sim um offset a ser somado ao endereço de carregamento.
Use e abuse da opção /NE para verificar êrros em seus programas antes de assemblá-los definitivamente.

5. USANDO ZBUG

ZBUG dispõe de alguns recursos muito práticos e eficientes para executar experimentalmente seus programas em linguagem de máquina. Com ZBUG você poderá examinar cada registrador, flag e conteúdo da memória a cada passo de seu programa.
Revise os comandos explicados no capítulo 2 pois estes serão utilizados neste capítulo.
Usaremos o programa de exemplo do capítulo 3 em nossos exercícios. Insira a instrução ORG $3F00 no início do programa e faça o assembly usando as opções /IM/AO. Depois passe para ZBUG (tecle Z e <ENTER>).

MODOS DE DISPLAY

No capítulo 2 estudamos os modos de exame. Além daqueles modos ZBUG dispõe de 3 modos de display. Nós estudaremos os modos de display juntamente com o modo de exame mnemônico. Se você não estiver neste modo tecle M e <ENTER>.

Modo Numérico

Tecle:

N <ENTER>

e examine as posições de memória de 3F00 a 3F0C onde reside seu programa. No modo numérico de display você não verá nenhum dos símbolos usados em seu programa (START, TELA e FIM). Por exemplo no endereço 3F0A você verá BNE 3F05 ao invés de BNE TELA.

Modo Simbólico

Tecle:

S <ENTER>

e examine o programa novamente. Agora todas as referências são feitas em termos de símbolos. Examine a posição que contem a instrução BNE TELA e tecle ; (ponto e vírgula). Usando-se este comando obtem-se a tradução numérica do operando simbólico TELA.

Modo Semi-simbólico

Tecle:

```
H <ENTER>
```

e examine o programa. Agora todas os endereços (a esquerda) são expressos simbólicamente, enquanto os operandos (a direita) são números.

Usando símbolos para examinar a memória

Como ZBUG entende os símbolos usados para o Assembly você pode usá-los em seus comandos. Por exemplo você pode usar indiferentemente os seguintes comandos:

```
START/
3F00/
```

Para ver todo o programa você pode usar:

```
T START FIM
T 3F00 3F0C
```

Você pode ainda obter uma cópia impressa desta listagem usando TH ao invés de T.

Executando o Programa

Antes de tentar uma execução experimental do programa certifique-se de ter uma cópia gravada do programa-fonte. Lembre-se que uma pequena falha em seu programa pode destruir todo o conteúdo da memória.
Você pode executá-lo com o comando G seguido do ponto de entrada do programa. Tecle um dos dois comandos que se seguem:

```
G START <ENTER>
G 3F00 <ENTER>
```

e o programa irá rodar enchendo metade da tela com o caracter gráfico número F9. Se isto não acontecer é por que o programa está com um ou mais defeitos (bugs). A mensagem 8 BRK @ 3F0C significa que a execução do programa terminou naquele endereço e que ZBUG interpretou a instrução SWI (Software Interrupt) como um "breakpoint" final.

BREAKPOINTS

Se seu programa tiver algum defeito será muito mais fácil encontrá-lo se você dividir o programa em várias partes e executá-las separadamente. No nível de comando tecle X seguido do endereço onde você quer interromper a execução. Vamos começar marcando o breakpoint no endereço 3F05, designado pelo símbolo TELA. Tecle um dos comandos abaixo:

```
XTELA <ENTER>
X3F05 <ENTER>
```

Agora tecle GSTART <ENTER> para executar o programa. Assim que a instrução 3F05 for ser executada o breakpoint irá interromper a execução e o contrôle retornará a ZBUG. Tecle C para continuar a cada vez que a execução for interrompida. A cada vez irá aparecer um caracter gráfico na tela, pois estamos executando o "loop" da instrução TELA, que põe o caracter na tela e avança uma posição de vídeo. Os caracteres parecem formar uma diagonal por causa do scroll de tela de ZBUG.
Tecle:

```
D <ENTER>
```

para ver os breakpoints que você determinou.
Tecle:

```
C10 <ENTER>
```

e ZBUG só interromperá a execução do programa quando passar pelo breakpoint pela décima vez.
Tecle:

```
Y <ENTER>
```

Este é o comando que cancela todos os breakpoints. Usando Y seguido de um endereço somente o breakpoint daquele endereço será cancelado. Se omitimos o endereço todos os breakpoints são cancelados.
Você pode determinar até 8 breakpoints (numerados de 0 a 7).

Registradores e Flags

Tecle:

R/

Assim você verá o conteúdo de todos os registradores do processador 6809 naquele estágio do programa. (Veja a definição de registradores e flags na Parte II). Repare no registrador CC e nas letras a sua direita. CC é o Condition Code, o registrador que mostra várias condições causadas por operações anteriores ou por determinação do programador. As letras indicam as flags que estão ativas no momento (set).A letra E por exemplo indicaria que a flag E está ligada (set).
Tecle:

X/

ZBUG mostrará o conteudo do registrador X. Você pode modificar seu conteúdo como se modifica o conteúdo de qualquer endereço de memória. Tecle:

0 <ENTER>

Agora o registrador X contem 0.

Executando Passo a Passo

Tecle:

3F00, (repare na vírgula)

Desta maneira a instrução 3F00 (LDA #$F9) foi executada (confira com o comando R) e a próxima instrução a ser executada (LDX #$500) é exibida na tela. Para executá-la basta teclar , (vírgula).
Você pode continuar executando as instruções uma a uma examinando os registradores. Este é um bom método para se descobrir algum érro difícil de ser detectado.
Se você chegar a última instrução (SWI) ZBUG irá avisá-lo que é impossivel continuar. Caso esta instrução não fosse SWI você poderia continuar seguindo as instruções contidas na memória.

Transferir Blocos de Memória

Tecle:

```
U 3F00 3000 6
```

Desta maneira os primeiros seis bytes de seu programa foram copiados para os endereços a partir de 3000.

Gravar em Fita

Para gravar um bloco de memória em fita tecle:

```
P EXEMPLO 3F00 3F0C 3F00 <ENTER>
```

Tecle <ENTER> quando o cassette estiver pronto para gravar. Este comando irá gravar seu programa começando no endereço 3F00 e terminando em 3F0C. O último número é o ponto de entrada do programa, onde está a primeira instrução a ser executada. Neste caso o ponto de entrada é igual ao ponto inicial.
Você também pode usar este comando para gravar dados que estejam contidos na memória. Neste caso deve-se fornecer um ponto de entrada fictício, que nunca será usado.
Para carregar de volta o programa EXEMPLO, tecle:

```
L EXEMPLO <ENTER>
```

ou apenas:

```
L <ENTER>
```

Dicas para Debug (Correção)

Seja paciente. Nenhum programa funciona na primeira vez que é testado. Debug é uma etapa necessária no desenvolvimento de programas tanto para principiantes quanto para programadores profissionais.
Faça uma cópia de seu programa fonte antes de executar seu programa objeto. Não há qualquer proteção possível contra a ação de seu programa, que pode destruir o buffer de edição e até mesmo o próprio EDTASM.

6. CALCULADOR ZBUG

ZBUG tem um calculador interno capaz de realizar operações aritmeticas, lógicas e relacionais. Além disso você pode usar tres sistemas numéricos diferentes, caracteres ASCII e simbolos.
Estudaremos aqui vários exemplos de aplicação. Alguns deles requerem que você tenha o programa EXEMPLO (Capitulo 3) assemblado na memória.

Sistemas Numéricos

ZBUG reconhece números em 3 sistemas diferentes: hexadecimal (base 16), decimal (base 10) e octal (base 8).

Modo de Output

O modo de output determina qual sistema numérico ZBUG usará para exibir números na tela. Use a letra O seguida do número base. Tecle:

O10 <ENTER>

e examine a memória. A letra T que segue cada número indica que o sistema decimal está sendo usado. Tecle:

O8 <ENTER>

e você passará a ver os número terminados com a letra Q, indicando que o sistema octal está em função. Tecle:

O16 <ENTER>

e você voltou ao sistema hexadecimal, que é o modo normal de output (default).

Modo de Input

Assim como os modos de output você pode mudar os modos de Input. Para tanto, use a letra I seguida da base desejada. Normalmente se você teclar um número ZBUG irá interpretá-lo como hexadecimal. Tecle:

I10 <ENTER>

e ZBUG passará a interpretar os números que você teclar como decimais. Por

exemplo se você teclar agora.

```
T 49152 49162 <ENTER>
```

você obterá uma listagem das posições de memória 49152 até 49162 (decimal) ou C000-C00A (hexadecimal). Note que a base dos números exibidos na tela é determinada pelo modo de output, independente da base escolhida para input.
Você pode ainda ignorar o modo de input teclando certos símbolos especiais. Em outras palavras, sem modificar o modo de input você pode determinar que um número seja interpretado como pertencente a uma base numérica diferente da que está em vigor.

BASE	Antes do Número	Depois do Número
10	&	T
16	$	H
8	@	Q

Por exemplo, estando ainda no modo de Input I10, tecle:

```
T 49152 $C010 <ENTER>
```

O símbolo "$" tem precedência sobre o modo I10. Portanto ZBUG interpretará C010 como um número hexadecimal. Passe agora para o modo I16 e tecle:

```
T 49152T C010 <ENTER>
```

Aqui o símbolo "T" colocado após 49152 terá precedência sobre o modo I16 e o número será interpretado como decimal.

Operações

ZBUG realiza vários tipos de operações. Tecle por exemplo:

```
C000 +25T/
```

e ZBUG abrirá o endereço C019 (hexa), que é a soma de C000 (hexa) e 25 (decimal). Se você deseja somente conhecer o resultado tecle:

```
C000 + 25T=
```

Nós iremos usar frequentemente os termos "operando", "operador" e "operação".

Uma operação é qualquer cálculo que você deseje resolver com ZBUG. Nesta operação:

```
1+2=
```

1 e 2 são os operandos e + é o operador.

Operandos

Os seguintes podem ser usados como operandos:

1. Caracteres ASCII
2. Símbolos
3. Números inteiros (bases 8, 10 ou 16)

Exemplos:

```
'A=
```

mostrará 41, o código ASCII (hexadecimal) da letra A.

```
START=
```

mostrará o endereço de START do programa de exemplo. Se você não tiver assemblado o programa na memória o resultado será uma mensagem de erro.

```
15Q=
```

mostrará o valor hexadecimal de 15 na base octal.
Se você quiser resultados em outro sistema numérico use um modo de output diferente.

Operadores

Você pode usar operadores aritméticos, lógicos ou relacionais. Passe para o modo de output O16 para estudar os exemplos seguintes.

Operadores Aritméticos

Adição	+	Modulus	.MOD.
Subtração	-	Positivo	+
Multiplicação	*	Negativo	-
Divisão	.DIV.		

Exemplos:

```
FIM-START=
```

mostrará o tamnho em bytes do programa EXEMPLO (excluindo a instrução final)

```
9.DIV.2=
```

mostrará 4 (ZBUG só opera com números inteiros)

```
9.MOD.2=
```

mostrará 1, que é o resto da divisão de 9 por 2.

```
1-2=
```

mostrará 0FFFF, 65535T ou 177777Q, dependendo do modo de output que você estiver usando. ZBUG nunca mostrará um número negativo como resultado. Ele usa um sistema circular que opera com módulo 10000 (hexadecimal). Para entender melhor este sistema você pode compará-lo com um relógio. Um relógio opera com módulo 12 da mesma maneira que ZBUG trabalha com módulo 10000. Para o relógio 1:00 - 2 é igual a 11:00.

Operadores Relacionais

Igual	.EQU.
Não Igual	.NEQ.

Estes operadores mostram se uma relação é falsa ou verdadeira.
Exemplos:

```
5.EQU.5=
```

exibirá 0FFFF (ou 65535T ou 177777Q) uma vez que a relação é verdadeira.

```
5.NEQ.5=
```

exibirá 0, pois a relação é falsa.

SHIFT	<
AND	.AND.
OR	.OR.
XOR	.XOR.
COMPLEMENTO	.NOT.

Operadores lógicos realizam manipulação de bits em números binários. Para melhor entender estas operações consulte os livros de referência citados na Introdução. Exemplos:

```
10<2=
```

Faz o shift de 2 bits à esquerda do número binário 10, resultando em 40. Trata-se da mesma operação realizada pela instrução ASL.

```
10<-2=
```

Faz o shift de 2 bits à direita resultando em 4. Mesma operação realizada pela instrução ASR.

```
6.XOR.5=
```

Resulta em 3, o produto de OR exclusivo entre 6 e 5. A instrução EOR realiza a mesma operação.

Operações Complexas

No caso de operações complexas ZBUG respeita uma hierarquia de operadores. As operações são realizadas nesta ordem:

1. * .DIV. .MOD. <
2. .AND.
3. .OR. .XOR.
4. + -
5. .EQU. .NEQ.

Exemplos:

```
4+4.DIV.2=
```

A divisão é realizada em primeiro lugar. O resultado é 6.

```
(4+4).DIV.2=
```

A adição é realizada em primeiro lugar e o resultado é 4.

7. ASSEMBLER + BASIC

O produto final de seu trabalho é um programa em linguagem de máquina assemblado e testado. Você pode rodar este programa diretamente do BASIC como programa independente ou como rotina de seu programa BASIC. Os procedimentos são os seguintes:

Dentro do Editor:
1. Revisar o programa para que ele funcione como rotina e volte ao BASIC.
2. Fazer o assembly em fita.

Em BASIC:
3. Carregar o programa com CLOADM.
4. Executar o programa:
   - como programa independente usando o comando EXEC.
   - como subrotina de seu prograe BASIC usando CLEAR e USR.

1. Revisar o Programa

Antes de usar seu programa a partir do BASIC você deve fazer uma pequena mudança. Será necessário transformá-lo numa rotina que após a execução retorne ao BASIC.
Em nosso programa fonte a penúltima instrução é SWI. Carregue o programa e altere esta instrução para RTS (Return de sub-rotina).
Agora o programa é realmente uma rotina que poderá ser executada em BASIC. (Se você quiser executá-la com IBUG outra vez você deverá trocar RTS por SWI novamente ou fixar um breakpoint na instrução RTS.)
Veja o programa devidamente revisado:

```
        ORG     $3F00
START   LDA     #$0F9
        LDX     #$500
TELA    STA     ,X+
        CMPX    #$5FF
        BNE     TELA
FIM     RTS
        END
```

2. Fazer o Assembly

Agora que o programa já está em sua versão final, faça o Assembly em fita com o comando:

A EXEMPLO <ENTER>

Aqui treina a função de EDTASM, portanto você pode abandoná-lo com o comando Q ou até mesmo desligar o computador e voltar a ligá-lo.

3. Carregar o Programa

Para carregar o programa prepare o gravador e tecle:

CLOADM <ENTER>

Como o programa continha a instrução ORG $3F00 você não precisa especificar nenhum endereço de carregamento. O programa entrará na memória a partir do endereço 3F00 (ou 16128 decimal).
Se seu programa não tem uma instrução ORG o comando CLOADM deverá especificar um endereço de carregamento. Por exemplo, CLOADM 16000 <ENTER> carregaria o programa a partir do endereço 16000.

4. Executar

Você pode executá-lo independentemente ou como subrotina.

Execução Independente

Tecle um dos comandos abaixo:

EXEC 16128 <ENTER>
EXEC &H3F00 <ENTER>
EXEC <ENTER>

O programa rodará e voltará ao BASIC OK.

Execução como Rotina

Esta é a maneira mais comum de se usar rotinas em linguagem de máquina. Quando seu programa BASIC precisar executar uma tarefa que em BASIC é impossível ou muito demorada você pode chamar uma subrotina em linguagem de máquina. Quando a tarefa tiver sido executada o controle voltará ao programa BASIC.
Digite e rode este programa BASIC:

```
10 CLEAR 200,16128
20 DEFUSR0=16128
30 CLS
40 INPUT"TECLE <ENTER> PARA COMECAR";A$
50 A=USR(0)
60 INPUT"QUER RODAR OUTRA VEZ";A$
70 IF A$="SIM" THEN 20

RUN <ENTER>
```

Normalmente o BASIC pode usar toda a memória a partir de 1536 (decimal), ou seja o programa BASIC poderia destruir sua rotina em linguagem de máquina. A linha 10 faz o CLEAR da área de memória a partir de 16128 (3F00 hexadecimal) e assim o BASIC não poderá usar esta área.
A linha 20 determina o ponto de entrada da rotina (USR). A linha 50 chama a subrotina.

Passando Parâmetros

Se você quiser passar algum dado a ser processado pela rotina você pode substituir o 0 pelo parâmetro. Por exemplo:

```
50 A=USR(5)
```

estará passando o parâmetro 5 para a rotina em linguagem de máquina ao chamá-la. Para usar este parâmetro sua rotina deverá ter estas duas instruções:

```
INTCNV  EQU   $B3ED
        JSR   [INTCNV]    (ver nota)
```

o que irá chamar uma rotina de ROM chamada INTCNV. Esta rotina irá colocar o parâmetro 5 no registrador D.
O programa em linguagem de máquina por sua vez pode devolver um parâmetro ao programa BASIC (por exemplo, um dado já processado). Basta colocá-lo no

registrador D e então executar estas instruções:

```
GIVABF  EQU   $B4F4
        JSR   [GIVABF]
```

GIVABF é uma rotina em ROM que vai atribuir à variável de sua instrução USR (neste caso A) o valor do registrador D.

Você pode obter mais informações sôbre a interação BASIC X Linguagem de Máquina nos livros citados na Introdução e no capítulo 19 do Manual de Operação e Linguagem CP-400 COLOR, onde você encontrará também uma série de rotinas em ROM que você poderá usar em seus programas em linguagem de Máquina.

Nota: Para gerar o caracter "[", tecle SHIFT e seta para baixo simultâneamente e para gerar o caracter "]" tecle SHIFT e seta a direita.

LINGUAGEM ASSEMBLER DO PROCESSADOR 6809

O microprocessador 6809, produzido pela MOTOROLA é o coração da unidade central de processamento dos computadores TRS-80 COLOR e compatíveis.
Este capítulo se destina a dar uma idéia de sua programação. Não se trata de um curso completo de programação Assembler 6809. Aos interessados em aprender seriamente a programação Assembler recomendamos o estudo dos livros citados na Introdução.

Registadores

O processador 6809 contem 9 áreas de armazenamento temporário que você pode usar em seus programas:

Registrador	Tamanho	Descrição
A	1 byte	Acumulador
B	1 byte	Acumulador
D	2 bytes	Acumulador (A+B)
DP	1 byte	Direct Page (Pagina Direta)
CC	1 byte	Condition Code (Código de Condição)
PC	2 bytes	Program Counter (Contador do Programa)
X	2 bytes	Indice
Y	2 bytes	Indice
U	2 bytes	Stack Pointer
S	2 bytes	Stack Pointer

A e B são registradores para manipular dados e fazer cálculos aritméticos. Cada um pode armazenar um byte. Combinados formam o registrador D.

DP é usado para endereçamento direto. Ele armazena o byte mais significativo de um endereço. Isso permite que o processador acesse diretamente um endereço a partir de apenas um byte (menos significativo).

X e Y são registradores com capacidade de armazenar dois bytes cada. Você usará estes registradores principalmente para endereçamento indexado.

PC é o registrador destinado a armazenar o endereco da instrução seguinte à que está sendo executada.

U e S são registradores de dois bytes que apontam para uma área de "stack" na

memória. Um stack é como uma "pilha" de informações na memória. A primeira informação a ser retirada é sempre a última que foi "empilhada". O endereço contido nestes registradores é igual ao topo do stack mais um. Por exemplo se o registrador U contem 0155 o stack começa em 154 e continua para baixo.
O processador automaticamente usa o registrador S em chamadas de rotinas e interrupts. O registrador U é de uso exclusivo do programador. Você pode acessar qualquer um deles através das instruções PSH e PUL ou com endereçamento indexado.

CC é o registrador de código de condição e interrupts. Ele está dividido em oito flags. Muitos operações vão ligar (set) ou desligar (reset ou clear) uma ou mais destas flags. Outras operações vão verificar CC para ver se uma determinada flag está "set" ou "clear". Este é o significado de cada flag quando está na condição "set".

C (carry), bit 0: Uma operação aritmética de 8 bits causou um carry ou borrow do bit 7. Ou seja haveria necessidade de um bit extra no resultado ou nos operandos para que o resultado fosse fiel.

V (overflow), bit 1: uma operação aritmética causou um overflow.

Z (zero), bit 2: o resultado da operação foi igual a zero.

N (negativo), bit 3: o resultado da operação foi negativo

I (interrupt request mask), bit 4: qualquer solicitação de interrupt será ignorada

H (half carry), bit 5: uma operação de adição de 8 bits causou um carry do bit 3

F (fast interrupt request mask), bit 6: qualquer solicitação de fast interrupt será ignorada.

E (entire flag), bit 7: todos os registradores foram para o Stack na ultima operação de stacking de interrupt. (Se E não estiver set apenas PC e CC foram para o stack)

Programas em Assembler

Você pode usar quatro campos numa linha de instrução em linguagem Assembler: símbolo, comando, operando e comentário. Nesta instrução:

```
START   LDA    #$0F9    PEGA CHR ASCII
```

START é o símbolo. LDA é o comando. #$0F9 é o operando e PEGA CHR ASCII é o comentário. Os comentários só servem para auxiliar o usuário, sendo ignorados pelo Assembler. Os símbolos # e $ do Operando serão estudados em seguida.

O símbolo

Você pode usar símbolos (também conhecidos como labels) para definir endereços na memória. START está definindo seu endereço. Uma vez definido você pode usar START como operando em outras instruções. Por exemplo:

```
        BNE     START
```

irá desviar a execução do programa (sob certas condições) para o endereço definido por START.
O Assembler guarda todos os símbolos, juntamente com os endereços ou valores a ele atribuídos, na tabela de símbolos. Esta tabela não faz parte do programa final.

O comando

O comando pode ser uma pseudo-operação (pseudo-op) ou uma instrução 6809.
As pseudo-ops controlam várias funções do próprio Assembler, tais como definir símbolos, determinar em que posição da memória armazenar o programa, etc. Elas não são traduzidas para linguagem de máquina 6809 e não fazem parte do programa final. Elas apenas fornecem informações para o Assembler. Por exemplo:

```
NOME    EQU     $43
```

define o símbolo NOME como igual a $43. Esta informação vai para a tabela de símbolos e permite que se use NOME no campo de operandos ao invés de $43.

```
        ORG     $3000
```

determina ao Assembler que monte o programa a partir do endereço 3000.

```
VALOR   FCB     $6
```

guarda o número 6 no endereço atual da memória e iguala VALOR a este endereço. VALOR e o endereço correspondente são armazenados na tabela de símbolos.

As instruções 6809 instruem ao processador que operação executar. Elas são traduzidas para o código em linguagem de máquina e armazenadas no programa final. Por exemplo:

```
        CLRA
```

indica ao processador que zere o registrador A. O Assembler traduz esta instrução para o código 4F e o coloca no programa. Todas as pseudo-operações e instruções 6809 estão listadas no fim desta parte.

O operando

O operando permite que você especifique um endereço de memória ou um dado. Por exemplo:

```
        LDD     #$3000
```

coloca o valor $3000 no registrador D. O operando, #$3000 especifica uma constante. O símbolo $ indica que 3000 é hexadecimal. O default do Editor-Assembler é decimal portanto você precisa indicar quando o número é hexadecimal ou octal desta maneira:

BASE	Antes do Número	Depois do Número
HEXADECIMAL	$	H
OCTAL	@	Q

O Assembler trata o operando como parte da instrução 6809 e guarda o operando junto com a instrução no programa final.

Modos de Endereçamento

No exemplo acima usamos o símbolo # para determinar a interpretação de 3000 como um dado. Nós podemos especificar uma interpretação diferente omitindo o símbolo #:

```
        LDD     $3000
```

Nesta instrução $3000 será interpretado como endereço. O processador colocará em D os dados contidos nos endereços 3000 e 3001.

As instruções 6809 permitem que você use de um a seis modos de endereçamento. Estes modos de endereçamento determinam se a instrução requer um operando e de que modo este operando deve ser interpretado.

Os vários modos de endereçamento são um dos recursos mais modernos e eficientes do processador 6809 e devem ser entendidos perfeitamente para se conseguir programas compactos e eficientes.

1. Endereçamento Inerente

Não há operando neste modo pois a instrução é auto-suficiente. Por exemplo

```
SWI
```

interrompe o processamento do software (Não há função para operando)

```
CLRA
```

zera o registrador A. O registrador é parte da própria instrução.

2. Endereçamento Imediato

O operando é um dado. Você deve usar o símbolo # para especificar este modo. Por exemplo:

```
ADDA    #$30
```

soma 30 ao valor do registrador A.

```
DATA    EQU     $8004
        LDX     #DATA
```

coloca o valor 8004 no registrador X.

```
CMPX    #$1234
```

compara o conteúdo do registrador X com o valor 1234.

3. Endereçamento Extendido

O operando é um endereço. Este é o modo default de todos os operandos. Se você

não especificar outro modo o Assembler interpretará o operando como endereço.
Exceção: Se o primeiro byte do operando for idêntico ao conteúdo de DP (página direta) o modo de endereçamento será direto. DP começa com conteúdo 00 e assim continua se não for mudada. O endereçamento direto nos casos de coincidência de DP e do primeiro byte do operando é uma função automática do Assembler e do Processador. Este modo de endereçamento será explicado adiante. (Aviso aos principiantes: não se preocupe com esta exceção pois ela não oferece maiores complicações.)
Exemplos de endereçamento extendido:

```
        JSR     $1234
```

desvia a execução do programa para a subrotina no endereço $1234.

```
PONTO   EQU     $1234
        STA     PONTO
```

guarda o conteudo no endereço 1234.
Se a instrução está pedindo dados, o operando contem o endereço onde os dados se encontram. Por exemplo:

```
        LDA     $1234
```

Não irá colocar 1234 em A. O processador colocará em A o dado contido no endereço 1234. Se o conteúdo do endereço 1234 fôr 6 o novo conteudo do registrador A será 6.

```
        ADDA    $1234
```

soma o valor que estiver no endereço 1234 ao conteúdo de A.

```
        LDD     $1234
```

passa para D,um registrador de dois bytes, o conteudo dos endereços 1234 e 1235. Você pode usar o símbolo > (sinal de maior) para forçar este modo de endereçamento extendido, podendo assim evitar possíveis confusões criadas pela exceção acima citada.

3.1 Extendido Indireto
Este é um caso particular de endereçamento extendido. O operando é um endereço de outro endereço. Use os símbolos "[" e "]" para especificá-lo. (Tecle SHIFT seta para baixo para obter "[" e SHIFT seta a direita para obter "]".)

Exemplos:

```
JSR     [$1234]
```

desvia a execução não para o endereço 1234, mas para o endereço contido em 1234 e 1235. Se 1234 contiver 06 e 1235 contiver 11 o endereço efetivo é 0611. A sub rotina a ser executada é a que começa em 0611.

```
PONTO   EQU     $1234
        STA     [PONTO]
```

guarda o conteúdo de A no endereço contido em 1234 e 1235.
Este modo de endereçamento é utilizado frequentemente em chamadas de subrotinas em ROM. Por exemplo, o endereço da rotina POLCAT (que verifica se há alguma tecla sendo pressionada) está contido no endereço A000. A subrotina POLCAT poderia ser acessada através das instruções:

```
POLCAT  EQU     $A000
        JSR     [POLCAT]
```

4. Endereçamento Indexado

O operando é um registrador índice, que aponta um endereço. O registrador índice pode ser qualquer dos registradores de dois bytes, inclusive PC.
O registrador pode ser acrescido de uma constante ou outro registrador e pode também ser incrementado ou decrementado nas razões de 1 ou 2.
O símbolo que indica este modo de endereçamento é a vírgula.
Como exemplo, começaremos carregando X um registrador de dois bytes, com o valor 1234:

```
LDX     #$1234
```

Agora podemos acessar o endereço 1234 através de endereçamento indexado. Esta instrução:

```
STA     ,X
```

coloca o conteúdo de A no endereço 1234.

```
STA     3,X
```

coloca o conteúdo de A no endereço 1237, que é igual a 1234 + 3. (3 é uma

constante funcionando como offset. Offset é um fator a ser acrescentado a uma base.)

```
LOCAL   EQU   $4
        STA   LOCAL,X
```

coloca o conteúdo de A no endereço 1238, que é igual a 1234 + LOCAL. (LOCAL age como offset constante)

```
        LDB   #$5
        STA   B,X
```

coloca o conteúdo de A no endereço 1239, que é igual a 1234 + conteúdo de B no momento da instrução ser executada.

```
        STA   ,X+
```

Esta instrução realiza duas operações: 1- coloca o conteúdo de A no endereço 1234 (indicado por X) e 2- incrementa o valor de X que passa a conter 1235.

```
        STA   ,X++
```

1- coloca o conteúdo de A em 1235 (conteúdo atual de X) e 2- incrementa em 2 o valor de X, que passa a conter 1237

```
        STA   ,--X
```

1- decrementa em 2 o conteúdo de X, que passa a conter 1235 e 2- coloca o valor de A em 1235 (conteúdo atual de X).
Note que quando o incremento é negativo (decremento), o decremento é realizado antes da operação em si. Quando o incremento é positivo a operação é realizada antes de se efetuar o incremento..

4.1 Como dissemos acima, pode-se usar o registrador PC como registrador Índice. Nesta forma de endereçamento, chamada PCR (Program Counter Relative), o offset é interpretado de maneira especial. Por exemplo:

```
VALOR   FCB   0
        LDA   VALOR,PCR
```

Quando este programa for assemblado, o Assembler SUBTRAI o conteúdo do registrador PC do offset:

```
LDA     VALOR-PCR,PCR
```

No momento da execução o processador SOMA o conteúdo do registrador PC ao offset. Isto faz com que A seja carregado com VALOR. Este método se assemelha com endereçamento extendido e sua complexidade não parece justificar seu uso. Contudo, o uso de endereçamento PCR resulta em programas totalmente relocatáveis, ou seja, que podem rodar em qualquer lugar da memória sem necessitar ser modificado.

4.2 Endereçamento Indexado Indireto

O Operando é um registrador índice que aponta para um endereço de um endereço. Esta é uma variação do endereçamento indexado. Para ver um exemplo vamos assumir que:
O registrador X contem 1234
O endereço 1234 contem 11
O endereço 1235 contem 23
O endereço 1123 contem 64

A instrução:

```
LDA     [,X]
```

coloca o valor 64 em A. (O registrador X aponta para o endereço do endereço que contem o valor 64.)

```
STA     [,X]
```

coloca o valor de A no endereço 1123 (X aponta para o endereço 1234 que contem o endereço efetivo 1123.)

5. Endereçamento Relativo

O processador interpreta o operando como um endereço relativo à posição atual (PC). Não há símbolo para indicar este modo. O processador usa este modo automaticamente para todas as operações de BRANCH (desvio de execução).
Por exemplo, se esta instrução estiver no endereço 0580:

```
BRA     $0585
```

O Assembler converterá $0585 para um valor relativo +5 (585-580). Esta

transformação será invisível para você pois normalmente você estará fazendo um Branch para um símbolo. Por exemplo:

```
BRA     LOOP
```

O Assembler verificará o endereço correspondente a LOOP e fará todas as conversões necessárias.
O operando (endereço para onde este tipo de instrução desvia a execução) deve estar a menos de 128 bytes para trás ou para frente da instrução de Branch. Se estiver fora desta faixa você deverá usar a instrução LBRA (Long Branch). Você não deve se preocupar muito com isso, pois caso você use BRA para desviar para um endereço muito distante você será avisado de seu êrro e basta trocar por LBRA que permite uma faixa relativa de +32767 a -32768.

6. Endereçamento Direto

Neste modo o operando é formado pela metade do endereço. A outra metade é o conteúdo do Registrador DP:

```
Endereço  =  Registrador DP   +      Operando
             byte + significativo  byte - significativo
```

O Assembler e o processador usam este modo automaticamente sempre que o operando tiver como primeiro byte o mesmo valor que o conteúdo do registrador DP (direct page). Se você não mudar o registrador DP ele terá o valor 00.
Por exemplo, ambas as instruções:

```
JSR     $0015
JSR     $15
```

desviarão o fluxo do programa para o endereço 0015. Em ambos os casos o Assembler usa apenas 15 como operando, excluindo o 00. Quando o processador for executar a instrução ele pegará o byte 00 do registrador DP e vai combiná-lo com 15. (No começo das operações DP contem 0, assim como todos os outros registradores.)
Com endereçamento direto, todos os operandos começando com 00 ocuparão apenas um byte na memória e o programa pode ficar mais compacto e rápido. Assim se a maioria de seus operandos começa com 12 você pode mudar o registrador DP para 12.
Para fazê-lo você precisa avisar o Assembler que está realizando esta alteração com a instrução:

```
SETDP   $12
```

Assim o Assembler deixará de lado o byte inicial de todos os operandos cujo primeiro byte for 12. Ou seja, o Assembler irá assemblar o operando 1234 simplesmente como 34.
Depois você precisa colocar 12 no registrador DP, para que o processador faça a combinação correta de DP e operando e possa acessar o endereço efetivo correto. Como você não pode usar LD com este registrador, você terá que usar um truque do tipo:

```
LDB     #$12
TFR     B,DP
```

Agora a página direta é $12 para o Assembler na hora de montar o programa e será 12 também para o processador na hora de executar o programa. E assim o processador executará todos os operandos que começem com 12 de maneira direta e eficiente.
Embora o Assembler use endereçamento direto em todos os operandos cujo primeiro byte for igual ao conteúdo do registrador DP, você poderá forçar o uso deste modo de endereçamento e ao mesmo tempo melhorar a documentação de seu programa usando o símbolo < (sinal de menor), que é o símbolo de endereçamento direto. Por exemplo se DP contem 12:

```
JSR     <$15
```

desvia a execução para a subrotina localizada no endereço 1215. A instrução deixa claro que o modo de endereçamento é direto.
Da mesma maneira você pode usar > para determinar o modo de endereçamento extendido. Por exemplo:

```
JSR     >$1215
```

desvia a execução para o endereço 1215, independente do valor de DP. O Assembler e o Processador usarão os dois bytes do operando.

Pseudo Operações Assembler

Aqui estudaremos uma listagem de todas as pseudo operações e a sintaxe a ser observada no uso das mesmas. Modos de endereçamento são irrelevantes nestas operações. Usaremos o termo SIMB (símbolo) como uma string de 1 a 6 caracteres colocada no campo dos símbolos. EXPR é uma expressão qualquer usada no campo do operando.

END

```
     END   EXPR
```

Determina o fim da operação de assembly. Você pode usar uma expressão para especificar o ponto de entrada do programa. Por exemplo:

```
          END    $3F00
```

faz com que o Assembler finalize a operação de Assembly e caso esteja gravando o programa em fita grave $3F00 como ponto de entrada do programa. Assim, quando o programa for carregado através de CLOADM não haverá necessidade de se determinar endereço para a instrução EXEC.

EQU

```
     SIMB   EQU   EXPR
```

Atribui o valor de EXPR ao SIMB. Por exemplo:

```
     LOOP1    EQU    $3F00
```

atribui o valor $3F00 ao símbolo LOOP1. Você pode usar LOOP1 como dado ou endereço. EQU é útil para determinar valores de constantes. Você pode usar EQU em qualquer lugar do programa embora para efeito de organização recomende-se colocar todos os EQUs numa tabela no começo do programa.

FCB

```
     SIMB   FCB   EXPR
```

Coloca o valor da expressão no endereço atual da memória. O símbolo é opcional. A expressão deve ter apenas um byte. Por exemplo:

```
DATA    FCB     $33
```

guarda 33 no endereço DATA

```
DATA2   FCB     $33+VALOR
```

guarda 33+VALOR no endereço DATA2

FCC

```
SIMB   FCC   LIMITE string LIMITE
```

Guarda uma string ASCII na memória começando no endereço atual. O símbolo é opcional. O limite pode ser qualquer caracter. Por exemplo:

```
TABELA FCC    /NOSSA STRING/
```

escreve os codigos ASCII da expressão "NOSSA STRING" nas posições de memória a partir do endereço TABELA. A barra (/) funciona limite.

FDB

```
SIMB   FDB   EXPR
```

Guarda uma expressão na memória começando no endereço atual. O símbolo é opcional. A expressão pode ter dois bytes. Exemplo:

```
DADO    FDB    $3322
```

guarda o valor 3322 em DADO e DADO+1.

ORG

```
ORG   EXPR
```

Determina ao Assembler que o ponto inicial do programa deve ser igual a EXPR. Por exemplo:

```
            ORG    $3F00
```

faz com que o Assembler comece a montar o programa no endereço $3F00. Você pode colocar mais de um ORG no mesmo programa. Quando o Assembler chega ao novo ORG ele começará a colocar o código das instruções a seguir nos endereços que se seguem à nova EXPR.

RMB

```
RMB   EXPR
```

Reserva EXPR bytes de memória para dados. Por exemplo:

```
DATA   RMB    $6
```

Reserva 6 bytes de memória para dados a partir do endereço DATA.

SET

```
SIMB   SET   EXPR
```

Iguala SIMB a EXPR. Você pode usar SET para alterar o valor de SIMB em outros lugares do programa. Por exemplo:

```
VALOR  SET    $3500
```

define VALOR como sendo igual a 3500. Agora você pode usar VALOR como operando de instruções. Mais adiante no programa você pode redefinir VALOR:

```
VALOR   SET    $4300
```

e VALOR passa a ser igual a 4300.

SETDP

```
SETDP   EXPR
```

Determina ao Assembler que o registrador DP será igual a EXPR. Exemplo:

```
SETDP   $20
```

instrui o Assembler a igualar DP a 20. Seu programa deve também incluir uma instrução igualando DP a 20 para não haver discrepâncias entre os procedimentos do Assembler ao montar o programa e do Processador ao executá-lo

APENDICE A
COMANDOS DO EDITOR

Definição de termos

linha
Qualquer número de linha do programa. Qualquer número entre 0 e 63999 pode ser usado. Estes símbolos podem ser usados:

$ primeira linha do programa
* última linha do programa
. linha atual

linha atual
A última linha inserida, editada ou exibida na tela. Valor default: 100.

linha inicial
A linha onde uma operação vai começar. Na maior parte dos comandos a linha inicial é opcional. Quando omitida será usada a linha atual.

faixa
A linha ou linhas a serem usadas em uma operação. Em faixas com mais de uma linha deve-se especificá-las com os símbolos:

: separando a linha inicial da final.
! separando a linha inicial do número de linhas.

incremento
A diferença entre os números de linhas. Caso omitido em comandos o último incremento definido será usado. Valor default: 10.

nome
Nome de um arquivo em fita, de 1 a 8 caracteres começando por letra.

COMANDOS

C linha inic,faixa,inc
Copia a faixa para um novo local a partir de linha inicial usando incremento especificado. Todos os parâmetros devem ser especificados.

C500,100:150,10

D faixa

Deleta. Se a faixa for omitida a linha atual é deletada.

```
D100        D100:150        D
```

E linha

Exibe linha a ser editada. Se a linha não for especificada usa-se a linha atual.

```
E100        E
```

Estes são os subcomandos de edição:

A	Cancela todas as mudanças e recomeça a edição.
nCstring	Muda n caracteres para "string". Se n for omitido troca apenas o caracter onde está posicionado o cursor.
nD	Deleta n caracteres. Se n for omitido deleta apenas o caracter onde está o cursor.
E	Acaba a edição efetuando todas as mudanças sem mostrar o resto da linha.
H	Deleta o resto da linha e entra no modo de inserção.
Istring	Insere "string" na posição atual do cursor. Seta para a esquerda deleta caracteres. Para sair da inserção tecle SHIFT seta para cima.
nKcaracter	Deleta todos os caracters desde a posição atual do cursor até a enésima ocorrência do caracter especificado. Se n for omitido, deleta tudo até a primeira ocorrência.
L	Lista a linha atual e continua a edição.
Q	Abandona a edição e ignora todas as mudanças.
nScaracter	Procura a enésima ocorrência do caracter. Se n for omitido, procura a primeira ocorrência.

X	Leva cursor ao fim da linha e entra no modo de inserção
ENTER	Finaliza a edição, efetuando as mudanças e mostra o final da linha.
SHIFT	Finaliza sub-comando.
nEspaço	Move o cursor n posições para a direita. Se telar somente espaço avança uma posição.
n ←	Move o cursor n posições a esquerda. Quando se omite n volta uma posição.

F string

Procura a string especificada. A pesquisa começa na linha atual e acaba a cada vez que a string for encontrada. Se "string" for omitida a última string definida será utilizada.

```
FSTART      F
```

H faixa

Imprime faixa na impressora. Se a faixa for omitida, a linha atual será impressa.

```
H100      H100:200      H
```

I linha inicial, inc

Insere linhas começando com linha inicial, usando o incremento especificado. Ambos os parâmetros são opcionais.

```
I150,5          I200          I,10
```

L nome

Carrega o texto do programa fonte da fita cassette. Se o nome for omitido o primeiro arquivo encontrado será carregado. Não limpa o buffer de edição antes de efetuar o Load.

```
L EXEMPLO    L
```

M linha inic, faixa, inc
Comando Mover. Funciona como o comando C (copiar), mas as linhas originais são deletadas.

N linha inicial, inc
Renumera o programa começando com o número de linha inicial e usando o incremento especificado. Ambos os parâmetros são opcionais.

N100,50 N100 N

P faixa
Exibe as linhas determinadas pela faixa na tela.

P100:200 P100!5 P# P

Q
Volta ao BASIC. Para voltar ao Editor a partir do BASIC tecle EXEC &H7EE0.

R linha inicial, inc
Substitui linha inicial e depois insere linhas usando o incremento especificado. Ambos os parâmetros são opcionais.

R100,10 R100 R

T faixa
Imprime faixa na impressora sem incluir números de linha

T100 T100:150

V nome
Verifica o arquivo especificado por nome para assegurar que não há erros de checksum. Semelhante ao comando BASIC SKIPF. Se o nome for omitido, verifica o primeiro arquivo encontrado.

VEXEMPLO V

Z -
Passa para sub-sistema ZBUG. Para voltar ao Editor a partir de ZBUG tecle o comando E.

APENDICE B
Opções para Assembler

A NOME/opções...

Faz o assembly do programa fonte, transformando-o em programa em linguagem de máquina. Pode-se usar as seguintes opções:

/AO Origem Absoluta (usada em conjunto com /IM)

/IM Assembly na memória

/LP Listagem na Impressora

/MO Manual Origin (usada em conjunto com /IM)

/NL Não mostra a listagem

/NO Não produz código objeto

/NS Não mostra tabela de símbolos

/SS Listagem compacta

/WE Pausa nos erros

A não ser que IM ou NO sejam usados, o assembly será gravado em cassette usando o NOME especificado. Se o nome for omitido, o programa será gravado com o nome NONAME.
Exemplos:

```
A EXEMPLO/WE
A
A/IM/AO
```

APENDICE C
COMANDOS ZBUG

Definição de termos

expressão
Um ou mais números, símbolos ou caracteres ASCII. Se mais de um número for utilizado, você deve separá-los com estes operadores:

Multiplicação	*	Adição	+
Divisão	.DIV.	Subtração	-
Modulus	.MOD.	Igual	.EQU.
Shift	<	Diferente	.NEQ.
AND	.AND.	Positivo	+
XOR	.XOR.	Negativo	-
OR	.OR.	Complemento	.NOT.

endereço
Uma posição de memória. Pode ser especificado por expressão usando números ou símbolos.

nome
Nome de um arquivo em fita tendo de um a oito caracteres.

COMANDOS

C
Continua a execução de um programa após interrupção em breakpoint.

D
Display de todos os breakpoints em função.

E
Saída para o subsistema Editor.

G endereço
Executa programa começando no endereço especificado.

L nome
Carrega arquivo em linguagem de máquina da fita cassette. Se "nome" for omitido carrega o primeiro arquivo encontrado.

P nome end1 end2 end3

Onde end1 é o primeiro endereço do programa, end2 é o último endereço do programa e end3 é o ponto de entrada.
Este comando grava o conteúdo da memória de end1 até end2 como um programa em linguagem de máquina usando o nome especificado ou NONAME, caso não haja nome especificado. Para gravar dados que não formem um programa deve-se fornecer um ponto de entrada ficticio.

R

Mostra o conteúdo de todos os registradores.

T end1 end2

Mostra na tela todas as posições de memória de endereço 1 até endereço2.

TH end1 end2

Passa para a impressora todas as posições entre end1 e end2.

U end fonte end destino bytes

Transfere o número especificado de bytes contidos nas posições começando em endereço fonte para outra parte da memória começando em endereço de destino.

V nome

Verifica se o arquivo especificado está livre de erros de checksum. Se o nome fôr omitido verifica o primeiro arquivo encontrado.

X endereço

Estabelece um breakpoint no endereço. Se o endereço fôr omitido a posição atual será utilizada.

Y endereço

Deleta o breakpoint do endereço. Se o endereço fôr omitido todos os breakpoints serão cancelados.

COMANDOS DE MODO DE EXAME

A Modo ASCII
B Modo Byte
M Modo Mnemonico
W Modo Word
(o default é M)

COMANDOS DE MODO DE DISPLAY

H Semi-simbólico
N Numérico
S Simbólico
(o default é S)

COMANDOS DE SISTEMA NUMERICO

O base Output
I base Input
(base pode ser 8, 10 ou 16. O default é 16)

SIMBOLOS ESPECIAIS

endereço/
registrador/

Abre o endereço ou registrador e exibe seu conteúdo. Se se usar somente / omitindo endereço ou registrador abre o último endereço que foi examinado. Depois que o conteúdo fôr exibido você poderá teclar:

Novo conteúdo	para modificar o conteúdo
ENTER	para fechar e efetuar qualquer alteração
BREAK	para fechar sem fazer nenhuma alteração
↓	para passar ao próximo endereço e efetuar qualquer alteração
↑	para passar ao endereço anterior
→	para desviar para o endereço apontado pela instrução atual
;	para obter display numérico
=	para forçar o modo byte ou numérico
:	para interpretar o conteúdo em termos de flags

endereço,

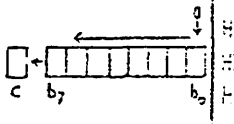
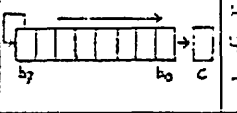
Executa a instrução do endereço. Se o endereço fôr omitido, teclando-se apenas a vírgula a próxima instrução (apontada pelo registrador PC) será executada.

expressão =

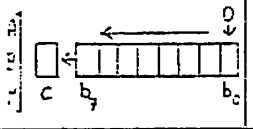
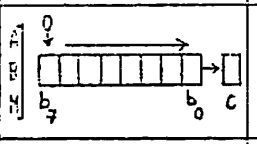
Calcula a expressão e mostra o resultado.

Instruções 6809

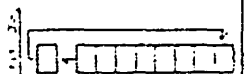
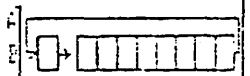
Fornecemos aqui uma listagem de todas as instruções 6809 com suas formas, modos possíveis de endereçamento, uma breve descrição de sua função e as flags que ela vai afetar.
Veja as notas ao fim da listagem para entender os símbolos.

Instrução/Formas		Modos de Endereçamento	Descrição	FLAGS
ABX		INERENTE	B+X → X	-
ADC	ADCA	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	A+M+C → A	HNZVC
	ADCB	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	B+M+C → B	HNZVC
ADD	ADDA	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	A+M → A	HNZVC
	ADDB	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	B+M → B	HNZVC
	ADDD*	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	D+M,M+1 → D	NZVC
AND	ANDA	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	A∧M → A	NZV
	ANDB	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	B∧M → B	NZV
	ANDCC	IMEDIATO	CC∧IM → CC	HNZVC
ASL	ASLA	INERENTE		HNZVC
	ASLB	INERENTE		HNZVC
	ASL	DIRETO,EXTENDIDO,INDEXADO		HNZVC
ASR	ASRA	INERENTE		HNZC
	ASRB	INERENTE		HNZC
	ASR	DIRETO,EXTENDIDO,INDEXADO		HNZC
BCC	BCC	RELATIVO	BRANCH C=0	-
	LBCC*	RELATIVO	LONG BRANCH C=0	-
BCS	BCS	RELATIVO	BRANCH C=1	-
	LBCS*	RELATIVO	LONG BRANCH C=1	-
BEQ	BEQ	RELATIVO	BRANCH Z=0	-
	LBEQ*	RELATIVO	LONG BRANCH Z=0	-

BGE	BGE	RELATIVO	BRANCH >=0	-
	LBGE	RELATIVO	LONG BRANCH >=0	-
BGT	BGT	RELATIVO	BRANCH >0	-
	LBGT	RELATIVO	LONG BRANCH >0	-
BHI	BHI	RELATIVO	BRANCH MAIOR	-
	LBHI	RELATIVO	LONG BRANCH MAIOR .	-
BHS	BHS	RELATIVO	BRANCH MAIOR =	-
	LBHS	RELATIVO	LONG BRANCH MAIOR =	-
BIT	BITA	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	TESTA BIT EM A	NZV
	BITB	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	TESTA BIT EM B	NZV
BLE	BLE	RELATIVO	BRANCH <=0	-
	LBLE	RELATIVO	LONG BRANCH <=0	-
BLO	BLO	RELATIVO	BRANCH MENOR	-
	LBLO	RELATIVO	LONG BRANCH MENOR	-
BLS	BLS	RELATIVO	BRANCH MENOR =	-
	LBCC	RELATIVO	LONG BRANCH MENOR =	-
BLT	BLT	RELATIVO	BRANCH <0	-
	LBLT	RELATIVO	LONG BRANCH <0	-
BMI	BMI	RELATIVO	BRANCH MENOS	-
	LBMI	RELATIVO	LONG BRANCH MENOS	-
BNE	BNE	RELATIVO	BRANCH NAO =	-
	LBNE	RELATIVO	LONG BRANCH NAO =	-
BPL	BPL	RELATIVO	BRANCH MAIS	-
	LBPL	RELATIVO	LONG BRANCH MAIS	-
BRA	BRA	RELATIVO	BRANCH SEMPRE	-
	LBRA	RELATIVO	LONG BRANCH SEMPRE	-
BRN	BRN	RELATIVO	BRANCH NUNCA	-
	LBRN	RELATIVO	LONG BRANCH NUNCA	-

JMP		DIRETO,EXTENDIDO,INDEXADO	ENDERECO → PC	-
JSR		DIRETO,EXTENDIDO,INDEXADO	JUMP SUBROTINA	-
LD	LDA	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	M → A	NZV
	LDB	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	M → B	NZV
	LDD	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	M,M+1 → D	NZV
	LDS	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	M,M+1 → S	NZV
	LDU	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	M,M+1 → U	NZV
	LDX	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	M,M+1 → X	NZV
	LDY	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	M,M+1 → Y	NZV
LEA	LEAS	INDEXADO	END. EFETIVO → S	-
	LEAU	INDEXADO	END. EFETIVO → U	-
	LEAX	INDEXADO	END. EFETIVO → X	Z
	LEAY	INDEXADO	END. EFETIVO → Y	Z
LSL	LSLA	INERENTE		NZVC
	LSLB	INERENTE		NZVC
	LSL	DIRETO,EXTENDIDO,INDEXADO		NZVC
LSR	LSRA	INERENTE		NZC
	LSRB	INERENTE		NZC
	LSR	DIRETO,EXTENDIDO,INDEXADO		NZC
MUL		INERENTE	A*B → D	ZC
NEG	NEGA	INERENTE	$\overline{A}$+1 → A	NNZVC
	NEGB	INERENTE	$\overline{B}$+1 → B	NNZVC
	NEG	DIRETO,EXTENDIDO,INDEXADO	$\overline{M}$+1 → M	NNZVC
NOP		INERENTE	OPERACAO NULA	-
OR	ORA	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	A∨M → A	NZV
	ORB	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	B∨M → B	NZV
	ORCC	IMEDIATO	CC∨IM → CC	NNZVC
PSH	PSHS	IMEDIATO	REGs para stack S	-
	PSHU	IMEDIATO	REGs para stack U	-
PUL	PULS	IMEDIATO	REGs do stack S	-
	PULU	IMEDIATO	REGs do stack U	-

BSR	BSR	RELATIVO	BRANCH SUBROTINA	-
	LBSR	RELATIVO	LONG BRANCH SUBROT	-
BVC	BVC	RELATIVO	BRANCH V=0	-
	LBVC	RELATIVO	LONG BRANCH V=0	-
BVS	BVS	RELATIVO	BRANCH V=1	-
	LBVS	RELATIVO	LONG BRANCH V=1	-
CLR	CLRA	INERENTE	0 → A	NZVC
	CLRB	INERENTE	0 → B	NZVC
	CLR	DIRETO,EXTENDIDO,INDEXADO	0 → M	NZVC
CMP	CMPA	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	A-M	MNZVC
	CMPB	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	B-M	MNZVC
	CMPD	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	D-M,M+1	NZVC
	CMPS	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	S-M,M+1	NZVC
	CMPU	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	U-M,M+1	NZVC
	CMPX	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	X-M,M+1	NZVC
	CMPY	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	Y-M,M+1	NZVC
COM	COMA	INERENTE	A→A	NZVC
	COMB	INERENTE	B→B	NZVC
	COM	DIRETO,EXTENDIDO,IMEDIATO	M→M	NZVC
CWAI		INERENTE,IMEDIATO	CCAIM → CC	ANZVC
DAA		INERENTE.	AJUSTE DECIMAL A	NZVC
DEC	DECA	INERENTE	A-1 → A	NZV
	DECB	INERENTE	B-1 → B	NZV
	DEC	DIRETO,EXTENDIDO,INDEXADO	M-1 → M	NZV
EOR	EORA	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	A⊻M → A	NZV
	EORB	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	B⊻M → A	NZV
EXG	R1,R2	INERENTE	R1 ↔ R2	-
INC	INCA	INERENTE	A+1 → A	NZV
	INCB	INERENTE	B+1 → B	NZV
	INC	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	M+1 → M	NZV

ROL	ROLA	INERENTE		NZVC
	ROLB	INERENTE		NZVC
	ROL	DIRETO,EXTENDIDO,INDEXADO		NZVC
ROR	RORA	INERENTE		NZC
	RORB	INERENTE		NZC
	ROR	DIRETO,EXTENDIDO,INDEXADO		NZC
RTI		INERENTE	VOLTA DE INTERRUPT	?NZVC
RTS		INERENTE	VOLTA DE SUBROTINA	-
SBC	SBCA	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	A-M-C → A	?NZVC
	SBCB	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	B-M-C → A	?NZVC
SEX		INERENTE	SINAL B → A	NZV
ST	STA	DIRETO,EXTENDIDO,INDEXADO	A → M	NZV
	STB	DIRETO,EXTENDIDO,INDEXADO	B → M	NZV
	STD	DIRETO,EXTENDIDO,INDEXADO	D → M,M+1	NZV
	STS	DIRETO,EXTENDIDO,INDEXADO	S → M,M+1	NZV
	STU	DIRETO,EXTENDIDO,INDEXADO	U → M,M+1	NZV
	STX	DIRETO,EXTENDIDO,INDEXADO	X → M,M+1	NZV
	STY	DIRETO,EXTENDIDO,INDEXADO	Y → M,M+1	NZV
SUB	SUBA	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	A-M → A	?NZVC
	SUBB	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	B-M → B	?NZVC
	SUBD	DIRETO,EXTENDIDO,IMEDIATO,INDEXADO	D-M,M+1 → D	NZVC
SWI	SWI	INERENTE	SOFTWARE INTERRUPT 1	-
	SWI2	INERENTE	SOFTWARE INTERRUPT 2	-
	SWI3	INERENTE	SOFTWARE INTERRUPT 3	-
SYNC		INERENTE	SINC. PARA INT.	-
TFR	R1,R2	INERENTE	R1 → R2	-
TST	TSTA	INERENTE	TESTE A	NZV
	TSTB	INERENTE	TESTE B	NZV
	TST	DIRETO,EXTENDIDO,INDEXADO	TESTE M	NZV

NOTAS:

R (R1,R2) é um registrador
S é o stack S
U é o stack U
M é o conteúdo de uma posição de memória
A é o acumulador A
B é o acumulador B
X é o registrador X
Y é o registrador Y
CC é o código de condição
IM é o operando da instrução
PC é o contador do programa
$\wedge$ é a operação lógica AND
$\vee$ é a operação lógica OR
$\veebar$ é a operação lógica OR exclusivo

Instruções 6809
DESCRITIVO

Segue um resumo das funções das instruções 6809. Vários mnemônicos são baseados em descrições em Inglês da função da instrução, que estão entre parênteses para auxiliar a memorização.

ABX Soma B e X e coloca o resultado em X.

ADC Soma operando imediato ou da memória ao conteúdo de A, B ou D e coloca o resultado em A, B ou D.

AND Operação lógica AND entre operando imediato ou de memória e um dos registradores A, B ou CC, com resultado em A, B ou CC.

ASL (Arithmetic Shift Left). Operação de shift à esquerda de todos os bits entrando 0 no bit menos significativo. Usado com A, B ou operando de memória.

ASR (Arithmetic Shift Right). Move todos os bits uma posição à direita e mantem bit 7. Usado com A, B e memória.

BCC (Branch on Carry Clear). Branch (desvie) se a flag C for igual a zero. Use LBCC se o destino estiver fora do alcance.

BCS (Branch on Carry Set). Branch se C=1. Use LBCS se fora do alcance.

BEQ (Branch if Equal). Branch se Z=1. Use LBEQ se fora do alcance.

BGE (Branch if Greater or Equal). Use em comparações entre operandos com sinal. Use LBGE se fora do alcance.

BGT (Branch if Greater Than). Use em comparações entre operandos com sinal. Use LBGT se fora do alcance.

BHI (Branch if High). O mesmo que branch se A>B usado com números sem sinal. Use LBHI se fora do alcance.

BHS (Branch if High or Same). O mesmo que branch se A>=B usado com números sem sinal. Use LBHS se fora do alcance.

BIT Testa qualquer bit ou bits de A ou B fazendo uma operação AND com operando

imediato ou de memória e A ou B. Resultado em CC.

BLE (Branch if Less than or Equal). Uso em comparações entre números com sinal. Use LBLE se fora do alcance.

BLO (Branch if LOwer). Branch se A<B. Uso com números sem sinal. Use LBLO se fora do alcance.

BLS (Branch if Lower or Same). Uso em comparações sem sinal. Use LBLS se fora do alcance.

BLT (Branch on Less Than). Uso em comparações com sinal. Use LBLT se fora do alcance.

BMI (Branch on MInus). Branch se o sinal do resultado for negativo. Use LBMI se fora do alcance.

BNE (Branch on Not Equal). Branch se Z=0. Use LBNE se fora do alcance.

BPL (Branch on PLus). Branch se o sinal do resultado for +. Use LBPL se fora do alcance.

BRA (BRanch Always). Branch incondicionalmente. Use LBRA se fora do alcance.

BRN (BRanch Never). Uma operação nula (NOP).

BSR (Branch to SubRoutine). Chamada de subrotina. Use LBSR se fora do alcance.

BVC Branch se a flag V estiver Clear, ou seja quando não houver overflow. Use LBVC se fora do alcance.

BVS Branch se a flag V estiver Set, em condição de overflow. Use LBVS se fora do alcance.

CLR (CLEAR). Zera A, B ou posição de memória.

CMP Compara o conteudo de A, B, D, S, U, X ou Y com um operando imediato ou de memória. Resultado em CC. Maneira normal de comparar dois operandos.

COM Complemento de A, B ou posição de memória. Complemento binário (muda os zeros para uns e os uns para zeros).

CWAI (WAIT) Espera por interrupt.

DAA (Decical Adjust A). Muda resultado binário para formato BCD. Operação prévia deve ter usado operandos BCD.

DEC Decrementa em 1 A, B ou posição de memória.

EOR (Exclusive OR). Operação de OR exclusivo de A ou B e operando imediato ou de memória. Cada bit ficará Set (=1) se um mas não ambos os bits dos dois operandos forem iguais a 1.

EXG (Exchange) Troca os valores de doiis registradores do mesmo tamanho.

INC Incrementa em 1 A, B ou posição de memória.

JMP (JUMP) Desvia a execução do programa para uma posição.

JSR (Jump SubRoutine) Chamada de subrotina.

LD (LOAD) Carrega em A, B, D, S, U, X, ou Y o valor de operando imediato ou de memória.

LEA (Load Effective Address) Pode carregar o conteudo de um registrador em outro ou incrementar ou decrementar X ou Y.

LSL (Logical Shift Left) O mesmo que ASL.

LSR (Logical Shift Right) Move todos os bits de A, B ou posição de memória para a direita, colocando 0 no bit 7.

MUL Multiplica A por B colocando o resultado em D.

NEG Negar. Complemento de dois em A, B ou posição de memória.

NOP Não faz nada. Operação nula.

OR OR lógico de operando imediato ou de memória com A, B, ou CC. ORCC é usado para ligar uma ou mais flags.

PSH (PUSH) Coloca um ou mais registradores no Stack. Os registradores podem estar em qualquer ordem. U ou S podem ser usados.

PUL (PULL) Tira um ou mais registradores do Stack.

ROL (Rotate Left) Rotação para a esquerda de A, B, ou posição de memória usando também a condição Carry, formando 9 bits.

ROR (Rotate Right) Rotação para a direita de A, B, ou posição de memória usando também a condição Carry, formando 9 bits.

RTI (Return from Interrupt) Volta de rotina de interrupt. Comparável a RTS.

RTS (Return from Subroutine) Volta de subrotina. Tira o endereço de volta do Stack S e volta à instrução seguinte ao BSR ou JSR.

SBC (Subtract with Carry) Subtrai um operando imediato ou de memória (+ conteudo de Carry) dos registradores A ou B. Resultado colocado em A ou B.

SEX (Sign EXtend) Se o sinal de B for - põe $FF em A, do contrário põe 0 em A.

ST (Store) Coloca o conteúdo dos registradores A, B, D, S, U, X ou Y em uma ou duas posições de memória.

SUB Subtrai um operando imediato ou de memória dos registradores A, B ou D. Resultado em A, B ou D.

SWI Software Interrupt.

SYNC Sincronismo para interrupt.

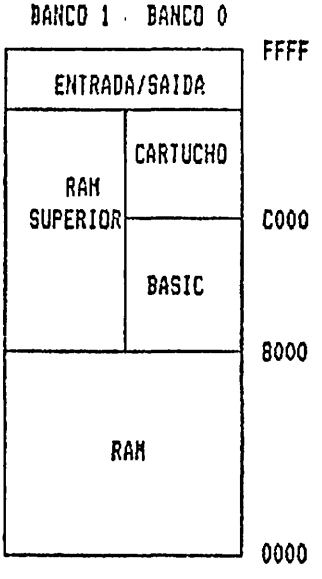
TFR Transfere (copia) um registrador para outro. Os registradores devem ser do mesmo tamanho.

TST Testa o contudo de A, B ou posição de memória, alterando as flags N ou Z de acordo com o resultado.

MAPA DA MEMORIA

O micro processador 6809 pode acessar 65536 posições de memória (FFFF hexadecimal) ou seja 64k bytes. Seu computador tem as posições de 8000 a BFFF gravadas em ROM (Read Only Memory) dedicadas ao interpretador BASIC e as de C000 a FEFF reservadas para a entrada de cartuchos. As posições superiores a FF00 são dedicadas ao controle de Entrada e Saida. Somando-se estas posições obtem-se um total de 32k em ROM.

A primeira vista parece impossivel que o computador possa ter 64k de RAM (Random Access Memory), mas na realidade há 32k de RAM que usam os mesmos endereços que a ROM, formando dois "bancos" distintos de memória que compartilham os mesmos endereços. Acessando-se determinadas posições de memória podemos determinar qual dos bancos devemos acessar num determinado momento. Podemos descrever esta situação com o esquema:



Color Edtasm

Edtasm foi originalmente desenvolvido para rodar em cartucho nos computadores da linha COLOR. Com algumas modificações que não acarretam perda de eficiência nem diminuição da memória acessada, a Peek & Poke fez com que o programa passasse a rodar no banco alternativo de memória dos computadores de 64k, permitindo assim sua gravação em fita cassette.

O efeito geral é o de transportar o banco 0 para o banco 1. Desta maneira todos

os 32k superiores estão em RAM, o que requer certas medidas de precaução por parte do usuário, que nunca deve alterar a memória acima da posição 7EE0, onde começa o módulo introdutório do programa COLOR EDTASM. Vejamos um esquema da memória, mostrando as áreas em que o programa atua e as áreas de acesso livre para o usuário.