Section II

COMMANDS

Section II

COMMANDS

This section shows how to use the many Disk EDTASM commands. Knowing these commands will help you edit and test your program.

Chapter 4/ Using the DOS Menu (DOS Commands)

When you first enter DOS, a menu of six DOS commands appear on the screen. *Chapter 2* shows how to use the first two DOS commands. This chapter shows how to use the remaining commands:

- Start Clock Display
- Disk Allocation Map
- Copy Files
- Directory

To use the examples in this chapter, you need to have the SAMPLE disk files, which you created in *Chapter 2*, on the diskette in Drive 0.

Directory

The DOS "directory" command lets you select the directory entries you want to see, using three fields: filename, extension, and drive number.

To select the directory entries, press (6) at the DOS Menu. Then, press the (←) to move the cursor left or (→) to move right.

Type this line to select all directory entries that have the filename SAMPLE.

```
[SAMPLE**] [***] :[Ø] <FILE SPEC
```

Use the (SPACEBAR) to erase characters. Press (ENTER) when finished. Then, press any key to return to the DOS menu, and press (6) to return to the directory.

Type this line to select all directory entries with the extension /BIN:

```
[********] [BIN] :[Ø] <FILE SPEC
```

Press (ENTER) when finished. Return to the main menu.

To see all directory entries on the disk in Drive 0, simply press (ENTER) without specifying a filename or extension:

```
[********] [***] :[Ø] <FILE SPEC
```

Disk Allocation Map

The "disk allocation map" command tells you how much free space you have on your diskettes. To see the map, press (4) at the DOS menu.

DOS shows a map of the diskettes that are in each drive. The map shows how each of the diskette's 68 granules is allocated:

- A period (.) means the granule is free.
- An X means all the sectors in the granule are currently allocated to a file.
- A number indicates how many sectors in the granule are currently allocated to a file.

Press any key to return to the DOS menu.

Copy Files

The "Copy Files" command makes a duplicate of a disk file. To use it, press (5) at the DOS menu. DOS then prompts you for the names of the files.

Single-Drive Copy

The first example copies SAMPLE/ASM to another file named COPY/ASM. Use the (←) and (→) to position the cursor. Answer the prompts as shown:

```
Source File Name        [SAMPLE ]
       Extension        [ASM]
       Drive            [Ø]

Destination File Name   [COPY    ]
       Extension        [ASM]
       Drive            [Ø]

If Drives are the same are you
using different diskettes?
      ( Y  or  N )?   [N]
```

When finished, press ENTER. DOS copies SAMPLE/ASM to a new file named COPY/ASM and then returns to the DOS menu. Check the directory (by pressing 6) and you'll see that both SAMPLE/ASM and COPY/ASM are on your diskette.

The next example copies SAMPLE/ASM to another diskette. Answer the prompts as shown:

```
Source File Name       [SAMPLE ]
     Extension         [ASM]
     Drive             [Ø]

Destination File Name  [COPY   ]
     Extension         [ASM]
     Drive             [Ø]

If Drives are the same are you
using different diskettes?
( Y  or  N )?   [Y]
```

Press ENTER. DOS then prompts you to insert the source diskette. Press ENTER again.

DOS then prompts you for a destination diskette. Insert the destination diskette and press ENTER. After copying the file, DOS prompts you for a system diskette. If you press ENTER without inserting a system diskette, you will get a SYSTEM FAILURE error.

When finished, it returns to the DOS menu.

Multi-Drive Copy

This example copies SAMPLE/ASM in Drive 0 to SAMPLE/ASM in Drive 1. Answer the prompts as shown:

```
Source File Name       [SAMPLE ]
     Extension         [ASM]
     Drive             [Ø]

Destination File Name  [SAMPLE ]
     Extension         [ASM]
     Drive             [1]

If Drives are the same are you
using different diskettes?
( Y  or  N )?   [N]
```

Start Clock Display

The Color Computer has a clock that runs on 60-cycle interrupts. Since the clock skips a second or more when the computer accesses tape or disk, we recommend that you not use it while executing a program.

To use the clock, press 3, "Start Clock Display." Six digits appear at the upper right corner of your screen. The first two are hours, the next are minutes, and the next are seconds. This clock counts the time until you exit DOS.

Chapter 5/ Examining Memory ZBUG Commands — Part I

To use the Disk EDTASM, you must understand the Color Computer's memory. You need to know about memory to write the program, assemble it, debug it, and execute it.

In this chapter, we'll explore memory and see some of the many ways you can get the information you want. To do this, we'll use ZBUG.

If you are not "in" ZBUG, with the ZBUG # prompt displayed, you need to get in it now.

> **EDTASM:** Load and run DOS, then execute the EDTASM program. At the editor's * prompt, type
>
> ```
> Z (ENTER)
> ```
>
> **EDTASMOV:** Load and run DOS, then execute the ZBUG program.

You should now have a # prompt on your screen. This means you are in ZBUG and you may enter a ZBUG command. All ZBUG commands must be entered at this command level. You can return to the command level by pressing (BREAK) or (ENTER).

Examining a Memory Location

The 6809 can address 65,536 one-byte memory addresses, numbered 0-65535 ($0000-$FFFF). We'll examine Address $A000. At the # prompt, type:

```
B (ENTER)
```

to get into the "byte mode." Then type:

```
A000/
```

and ZBUG shows the contents of Address $A000. To see the contents of the next bytes, press (↓). Use (↑) to scroll to the preceding address.

Continue pressing (↓) or (↑). Notice that as you use the (↑) the screen continues to scroll down. The smaller addresses are on the lower part of the screen.

All the numbers you see are hexadecimal (Base 16). You see not only the 10 numeric digits, but also the 6 alpha characters needed for Base 16 (A-F). Unless you specify another base (which we do in Chapter 9), ZBUG assumes you want to see Base 16 numbers.

Notice that a zero precedes all the hexadecimal numbers that begin with an alphabetic character. This is done to avoid any confusion between hexadecimal numbers and registers.

Examination Modes

To help you interpret the contents of memory, ZBUG offers four ways of examining it:

- Byte Mode
- Word Mode
- ASCII Mode
- Mnemonic Mode

Byte Mode

Until now, you've been using the byte mode. Typing B (ENTER), at the # prompt got you into this mode.

The byte mode displays every byte of memory as a number, whether it is part of a machine-language program or data.

In this examination mode, the (↓) increments the address by one. The (↑) decrements the address by one.

Word Mode

Type ENTER to get back to the # prompt. To enter the word mode, type:

```
W ENTER
```

Look at the same memory address again. Press the ↓ key a few times. In this mode, the ↓ increments the address by two. The numbers contained in each address are the same, but you are seeing them two bytes or one word at a time.

Press the ↑ a few times. The ↑ always decrements the address by one, regardless of the examination mode.

Look at Address $A000 again by typing:

```
A000/
```

Note the contents of this address "word." This is the address where POLCAT, a ROM routine, is stored.

Examine the POLCAT routine. For example, if $A000 contains A1C1, type:

```
A1C1/
```

and you'll see the contents of the first two bytes in the POLCAT routine. We'll examine this routine later in this chapter using the "mnemonic mode."

ASCII Mode

Return to the command level. To enter the ASCII mode, type:

```
A ENTER
```

ZBUG now assumes the content of each memory address is an ASCII code. If the "code" is between $21 and $7F, ZBUG displays the character it represents. Otherwise, it displays meaningless characters or "garbage."

Here, the ↓ increments the address by one.

Mnemonic Mode

This is the default mode. Unless you ask for some other mode, you will be in the default mode.

Return to the # prompt. To enter the mnemonic mode from another mode, type:

```
M ENTER
```

Look at the addresses where the POLCAT routine is stored. For example, if you found that POLCAT is at address $A1C1, type:

```
A1C1/
```

Press the ↓ a few times. In the mnemonic mode, ZBUG assumes you're examining an assembly language program. The ↓ increments memory one to five bytes at a time by "disassembling" the numbers into the mnemonics they represent.

For example, assume the first two addresses in POLCAT contain $3454. $3454 is an opcode for the PSHS U,X,B mnemonic. Therefore, ZBUG disassembles $3454 into PSHS U,X,B.

Begin the disassembly at a different byte. Press BREAK and then examine the address of POLCAT plus one. For example, if POLCAT starts at address $A1C1, type:

```
A1C2/
```

You now see a different disassembly. The contents of memory have not changed. ZBUG has, however, interpreted them differently.

For example, assume $A1C2 contains a $54. This is the opcode for the LSRB mnemonic. Therefore, ZBUG disassembles $54 into LSRB.

To see the program correctly, you must be sure you are beginning at the correct byte. Sometimes, several bytes will contain the symbol "??". This means ZBUG can't figure out which instruction is in that byte and is possibly disassembling from the wrong point. The only way of knowing you're on the right byte is to know where the program starts.

Changing Memory

As you look at the contents of memory addresses, notice that the cursor is to the right. This allows you to change the contents of that address. After typing the new contents, press ENTER or ↓; the change will be made.

To show how to change memory, we'll open an address in video memory. Get into the byte mode and open Address $015A by typing:

```
BREAK B ENTER
015A/
```

Note that the cursor is to the right. To put a 1 in that address, type:

```
1 ENTER
```

If you want to change the contents of more than one address, type:

```
Ø15A/
```

Then type:

```
DD ↓
```

This changes the contents to DD and lets you change the next address. (Press the ↓ to see that the change has been made.)

The size of the changes you make depends on the examination mode you are in. In the byte mode, you will change one byte only and can type one or two digits.

In the word mode, you will change one word at a time. Any 1-, 2-, 3-, or 4-digit number you type will be the new value of the word.

If you type a hexadecimal number that is also the name of a 6809 registers (A,B,D,CC,DP,X,Y,U,S,PC), ZBUG assumes it's a register and gives you an "EXPRESSION ERROR." To avoid this confusion, include a leading zero (0A,0B, etc.)

To change memory in the ASCII mode, use an apostrophe before the new letter. For example, here's how to write the letter C in memory at Address $015A. To get into the ASCII examination mode, type:

```
A (ENTER)
```

To open Address $015A,type:

```
Ø15A/
```

To change its contents to a C, type:

```
'C ↓
```

Pressing the ↓ will assure you that the address contains the letter C.

If you are in mnemonic mode, you must change one to five bytes of memory depending on the length of the opcode. Changing memory is complex in mnemonic mode because you must type the opcodes rather than the mnemonic.

For example, get into the mnemonic mode and open Address $015A. Type:

```
M (ENTER)
Ø15A/
```

To change this instruction, type:

```
86 (ENTER)
```

Now Address $015A contains the opcode for the LDA mnemonic. Open location 015B:

```
Ø15B/
```

and insert $06, the operand:

```
Ø6 (ENTER)
```

Upon examining Address $015A again, you'll see it now contains an LDA #6 instruction.

Exploring the Computer's Memory

You are now invited to examine each section of memory using ZBUG commands to change examination modes. Use the Memory Map in *Reference J.*

Don't hesitate to try commands or change memory. You can restore anything you alter simply by removing the diskette and turning the computer off and then on again.

Chapter 6/ Editing the Program Editor Commands

The editor has many commands to help you edit your source program. *Chapter 2* shows how to enter a source program. This chapter shows how to edit it.

To use the edit commands you must return to the editor from ZBUG:

> EDTASM: From EDTASM ZBUG, return to the editor by typing E (ENTER)
>
> EDTASMOV: From Stand-Alone ZBUG, return to the DOS menu by typing K (ENTER). Then, execute the EDTASMOV program.

The screen now shows the editor's * prompt. While in the editor, you can return to the * prompt at any time by pressing (BREAK).

This chapter uses SAMPLE/ASM from *Chapter 2* as an example. To load SAMPLE/ASM into the editor, type:

```
L SAMPLE/ASM (ENTER)
```

Print Command
Prange

To print a line of the program on the screen, type:

```
P100 (ENTER)
```

To print more than one line, type:

```
P100:130 (ENTER)
```

You will often refer to the first line, last line, and current line (the last line you printed or inserted). To make this easier, you can refer to each with a single character:

#	first line
*	last line
•	current line (the last line you printed or inserted.)

To print the current line, type:

```
P. (ENTER)
```

To print the entire text of the sample program, type:

```
P#:* (ENTER)
```

This is the same as P050:200 (ENTER).

The colon separates the beginning and ending lines in a range of lines. Another way to specify a range of lines is with !. Type:

```
P#!5 (ENTER)
```

and five lines of your program, beginning with the first one, are printed on the screen.

To stop the listing while it is scrolling, quickly type:

```
(SHIFT) @
```

To continue, press any key.

Printer Commands
Hrange
Trange

If you have a printer, you can print your program with the H and T commands. The H command prints the editor-supplied line numbers. The T command does not.

To print every line of the edit buffer to the printer, type:

```
H#:* (ENTER)
```

You are prompted with:

```
PRINTER READY
```

Respond with (ENTER) when ready.

The next example prints six lines, beginning with line 100, but without the editor-supplied line numbers. Type:

```
T100!6 (ENTER)
```

Edit Command
Eline

You can edit lines in the same way you edit Extended

COLOR BASIC lines. For example, to edit line 100, type:

```
E1ØØ (ENTER)
```

The new line 100 is displayed below the old line 100 and is ready to be changed.

Press the (SPACEBAR) to position the cursor just after START. Type this insert subcommand:

```
IED (ENTER)
```

which inserts ED in the line.

The edit subcommands are listed in *Reference A.*

Delete Command
D*range*

If you are using the sample program, be sure you have written it on disk before you experiment with this command. Type:

```
D11Ø:14Ø (ENTER)
```

Lines 110 through 140 are gone.

Insert Command
I*startline, increment*

Type:

```
I152,2 (ENTER)
```

You may now insert lines (up to 127 characters long) beginning with line 152. Each line is incremented by two. (The editor does not allow you to accidently overwrite an existing line. When you get to line 160, it gives you an error message.)

Press (BREAK) to return to the command level. Then type:

```
I2ØØ (ENTER)
```

This lets you begin inserting lines at the end of the program. Each line is incremented by two, the last increment you used.

Type:

```
(BREAK) I (ENTER)
```

The editor begins inserting at the current line.

On startup, the editor sets the current line to 100 and the increment to 10. You may use any line numbers between 0 and 63999.

Renumber Command
N*startline,increment*

Another command that helps with inserting lines between the lines is N (for renumber). From the command level, type:

```
N1ØØ,5Ø (ENTER)
```

The first line is now Line 100 and each line is incremented by 50. This allows much more room for inserting between lines.

Type:

```
N (ENTER)
```

The current line is now the first line number.

Renumber now so you will be ready for the next instruction. Type:

```
N1ØØ,1Ø (ENTER)
```

Replace Command
R*startline,increment*

The replace command is a variation of the insert command. Type:

```
R1ØØ,3 (ENTER)
```

You may now replace line 100 with a new line and begin inserting lines using an increment of three.

Copy Command
C*startline,range,increment*

The copy command saves typing by duplicating any part of your program to another location in the program.

To copy lines, type:

```
C5ØØ,1ØØ:15Ø,1Ø (ENTER)
```

This copies lines 100 to 150 to a new location beginning at Line 500, with an increment of 10. An attempt to copy lines over each other will fail.

ZBUG Command

The EDTASM system contains a copy of the stand-alone ZBUG program. This allows you to enter ZBUG while your program is still in memory.

EDTASMOV Users: You need to use the Stand-Alone ZBUG program, as shown in *Chapter 2.*

To enter ZBUG, type:

```
Z (ENTER)
```

The # prompt tells you that you are now in ZBUG.

To re-enter the editor from ZBUG, type the ZBUG command:

```
E (ENTER)
```

If you print your program, you'll see that entering and exiting ZBUG did not change it.

BASIC Command

To enter BASIC from the editor, type:

```
Q (ENTER)
```

If you want to enter DOS from the editor, type:

```
K (ENTER)
```

Entering DOS or BASIC empties your edit buffer. Re-entering the editor empties your BASIC buffer.

Write Command

WD *filespec*

This command is the same one you used in *Chapter 2* to write the source program to disk. It saves the program in a disk file named *filespec. Filespec* can be in one of these forms:

filename/ext:drive
filename.ext:drive

The *filename* can be one to eight characters. It is required.

The *extension* can be one to three characters. It is optional. If the extension is omitted, the editor assigns the file the extension /ASM.

The *drive* can be a number from 0 to 4. It is also optional. If the drive number is omitted, the editor uses the first available drive.

Examples:

```
WD TEST (ENTER)
```

saves source file currently in memory as TEST/ASM.

```
WD TEST/PR1
```

saves the source file currently in memory as TEST/PR1.

Load Command

LD *filespec*
LDA *filespec*

This command loads a source *filespec* from disk into the edit buffer. If the source *filespec* you specify does not have an extension, the editor uses /ASM.

If you don't specify the A option, the editor empties the edit buffer before loading the file.

If you specify the A option, the editor appends the file to the current contents of the edit buffer.

Appending files can be useful for chaining long programs. When the second file is loaded, simply renumber the file with the renumber command.

Examples:

```
LD SAMPLE:1
```

empties the edit buffer, then loads a file named SAMPLE/ASM from Drive 1.

```
LDA SAMPLE/PRO
```

loads a file named SAMPLE/PRO from the first available drive, then appends to the current contents of the edit buffer.

The editor has several other commands. These are listed in *Reference A.*

Hints on Writing Your Program

- Copy short programs from any legal source available to you. Then modify them one step at a time to learn how different commands and addressing modes work. Try to make the program relocatable by using indexed, relative, and indirect addressing (described in *Section III*).
- Try to write a long program as a series of short routines that use the same symbols. They will be easier to understand and debug. They can later be combined into longer routines.

  Note: You can use the editor to edit your BASIC programs, as well as assembly language programs. You might find this very useful since the EDTASM editor is much more powerful than the BASIC editor. You need to first save the BASIC program in ASCII format:

  SAVE *filespec*, A

  Then, load the program into the editor.

Chapter 7/ Assembling the Program (Assembler Commands)

To load the assembler program and assemble the source program into 6809 machine code, EDTASM (or EDTASMOV) has an "assembly command." Depending on how you enter the command, the assembler:

- Shows an "assembly listing" giving information on how the assembler is assembling the program.
- Stores the assembled program in memory.
- Stores the assembled program on disk.
- Stores the assembled program on tape.

This chapter shows the different ways you can control the assembly listing, the in-memory assembly, and the disk assembly. Knowing this will help you understand and debug a program.

The Assembly Command

The command to assemble your source program into 6809 machine code is:

Assembling in memory:

A /IM */switch2/switch3/ . . .*

The /IM (in memory) switch is required.

Assembling to disk:

A *filespec /switch1/switch2/ . . .*

The assembled program is stored on disk as *filespec.* If *filespec* does not include an *extension*, the assembler uses /BIN.

Assembling to tape:

A *filename /switch1/switch2/ . . .*

The assembled program is stored on tape as *filename*.

The *switch* options are as follows:

/AO	Absolute origin
/IM	Assemble into memory
/LP	Assembler listing on the line printer
/MO	Manual origin
/NL	No listing
/NO	No object code in memory or disk
/NS	No symbol table in the listing
/SR	Single record
/SS	Short screen listing
/WE	Wait on assembly errors
/WS	With symbols

You may use any combination of the switch options. Be sure to include a blank space before the first switch. If you omit *filespec*, you must use the in-memory switch (/IM).

Examples:

```
A/IM/WE
```

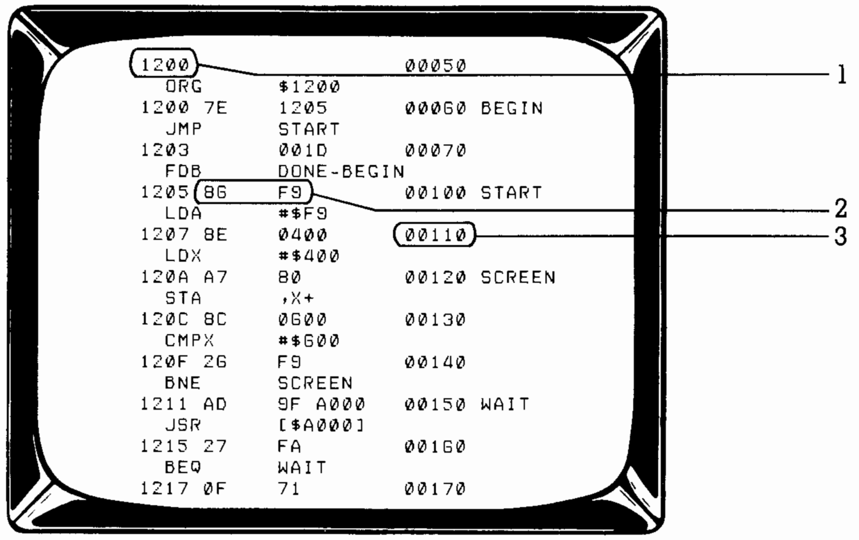
assembles the source program in memory (/IM) and stops at each error (/WE).

```
A TEST /LP
```

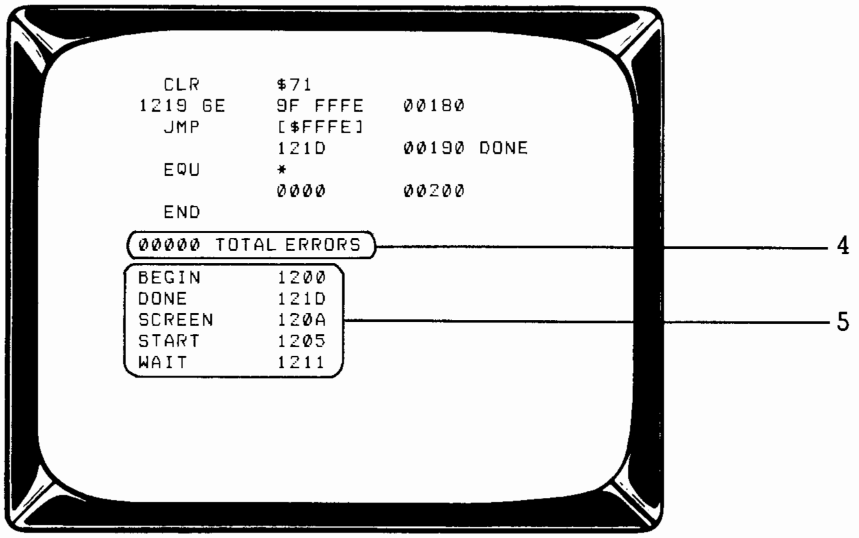
assembles the source program and saves it on disk as TEST/BIN. The listing is printed on the printer (/LP). Note that there must be a space between the *filespec* and the switch.

```
A TEST/PRO
```

assembles the source program and saves it on disk as TEST/PRO.



```
1200                    00050
  ORG      $1200
1200 7E    1205         00060 BEGIN
  JMP      START
1203       001D         00070
  FDB      DONE-BEGIN
1205 86    F9           00100 START
  LDA      #$F9
1207 8E    0400         00110
  LDX      #$400
120A A7    80           00120 SCREEN
  STA      ,X+
120C 8C    0600         00130
  CMPX     #$600
120F 26    F9           00140
  BNE      SCREEN
1211 AD    9F A000      00150 WAIT
  JSR      [$A000]
1215 27    FA           00160
  BEQ      WAIT
1217 0F    71           00170
```



```
  CLR      $71
1219 6E    9F FFFE      00180
  JMP      [$FFFE]
           121D         00190 DONE
  EQU      *
           0000         00200
  END
00000 TOTAL ERRORS
BEGIN      1200
DONE       121D
SCREEN     120A
START      1205
WAIT       1211
```

1. The location in memory where the assembled code will be stored. In this example, the assembled code for LDA#$F9 will be stored at hexadecimal location #1200.
2. The assembled code for the program line. $86F9 is the assembled code for LDA #$F9.
3. The program line.
4. The number of errors. If you have errors, you will want to assemble the program again with the /WE switch.
5. The symbols you used in your program and the memory locations they refer to.

Figure 1. Assembly Display Listing

Controlling the Assembly Listing

The assembler normally displays an assembly listing similar to the one in *Figure 1*. You can alter this listing with one of these switches:

/SS	Short screen listing
/NS	No symbol table in the listing
/NL	No listing
/LP	Listing printed on the printer

For example:

```
A SAMPLE /NS
```

assembles SAMPLE and shows a listing without the symbol table.

If you are printing the listing on the printer, you might want to set different parameters. You can do this with the editor's "set line printer parameters" command:

To use this command, type (at the * prompt):

```
S (ENTER)
```

The editor shows you the current values for:

- LINCNT — the number of lines printed on each page. ("line count")
- PAGLEN — the number of lines on a page. ("page length")
- PAGWID — the number of columns on a page. ("page width")
- FLDFLG — the "fold flag" (This flag should contain 1 if your printer does not "wrap around." Otherwise, the flag should contain 0.)

It then prompts you for different values. Check your printer manual for the appropriate parameters. If you want the value to remain the same, simply press (ENTER). For example:

```
LINCNT=58
PAGLEN=66
PAGWID=80
FLDFLG=0
```

sets the number of lines to 58, the page length to 66, and the page width to 80 columns. You can then assemble the program with the /LP switch:

```
A SAMPLE /LP
```

and the assembler prints the listing on the line printer using the parameters just set.

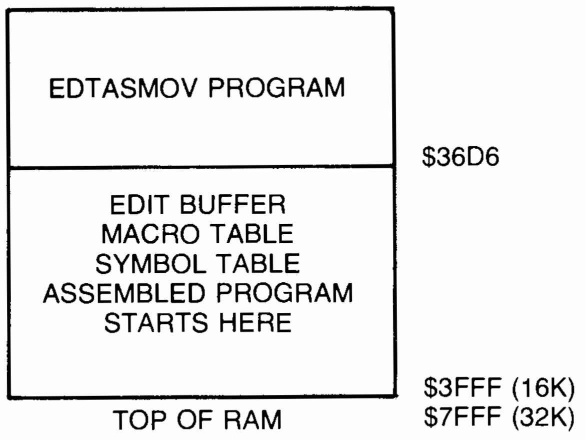
In-Memory Assembly The /IM Switch

The /IM switch causes the program to be assembled in memory, not on disk or tape. This is a good way to find errors in a program.

Where in memory? This depends on whether you use the /IM switch alone or accompany it with an ORG instruction, an /AO switch, or an /MO switch.

Using the /IM Switch Alone

This is the most efficient use of memory. The assembler stores your program at the first available address after the EDTASM (or EDTASMOV) program, the edit buffer, and the symbol table:



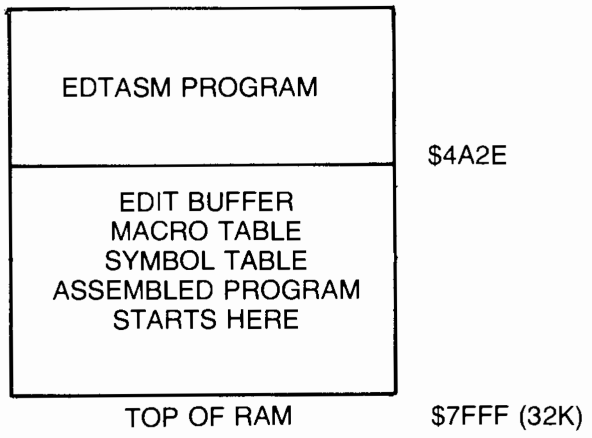


Figure 2. In-Memory Assembly

The EDTASM program ends at Address $4A2D. The EDTASMOV program ends at $36D5.

The edit buffer contains the source program. It begins at Address $4A2E or $36D6 and varies in size depending on your program's length.

The macro table references all the macro symbols in your program and their corresponding values. (Macros are described in Chapter 12.) Its size varies depending on how many macros your program contains.

The symbol table references all your program's symbols and their corresponding values. Its size varies depending on how many symbols your program contains.

Example:

Load the SAMPLE/ASM back into the edit buffer. At the * prompt, type:

```
L SAMPLE/ASM (ENTER)
```

Delete the ORG line. At the * prompt, type:

```
D5Ø (ENTER)
```

Then assemble the program in memory by typing:

```
A/IM (ENTER)
```

(If you want another look, type A/IM again. You can pause the display by pressing (SHIFT) (@) and continue by pressing any key.)

Since this sample program uses START to label the beginning of the program, you can find its originating address from the assembler listing. If you are using EDTASM, it should begin at Address $4B1E. If you are using EDTASMOV, it should begin at $37C6.

Using ORG with /IM for Origination Offset

If you have an ORG instruction in your program and do not use the AO switch, the assembler stores your program at:

the first available address + the value of ORG

Example:

Insert this line at the beginning of the sample program:

EDTASM Systems:

```
ØØ5Ø        ORG $6ØØØ
```

EDTASMOV Systems:

```
ØØ5Ø        ORG $38ØØ
```

Then, at the * prompt, type:

```
A/IM (ENTER)
```

The START address is now the first available address + $6000 or $3800. This means that if you have less than 32K (with EDTASM) or less than 16K (with EDTASMOV), the program extends past the top of RAM and you will get a BAD MEMORY error.

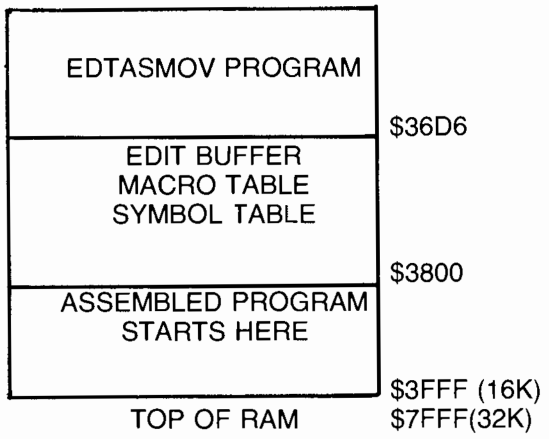
Using IM with /AO for Absolute Origin

The AO switch causes the assembler to store your program "absolutely" at the address specified by ORG.

With the ORG instruction inserted, type (at the * prompt):

```
A/IM/AO (ENTER)
```

Your program now starts at address $6000 or $3800:



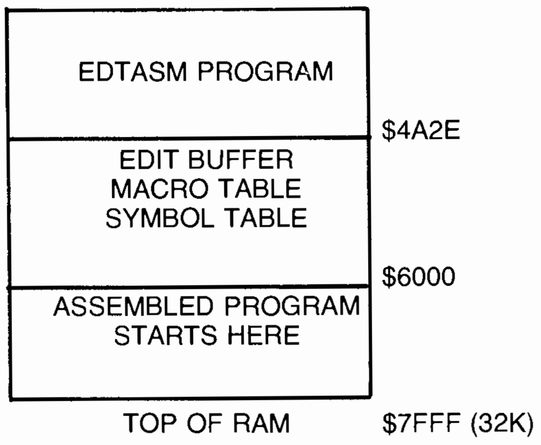


Figure 3. /AO In-Memory Assembly.

As you can see, the AO switch set the location of the assembled program only. It did not set the location of the edit buffer or the symbol table.

If your ORG instruction does not allow enough memory for your program, you will get a BAD MEMORY error. The assembler cannot store your program beyond the top of RAM.

Using /MO with /IM for Manual Origin

The /MO switch causes your program to be assembled at the address set by USRORG (plus the value set in your ORG instruction, if you use one). To set USRORG, use the editor's "origin" command.

Before setting USRORG, remove the ORG instruction from your program. Then, at the * prompt, type:

```
O (ENTER)
```

The editor shows you the current values for:

- FIRST — the first hexadecimal address available
- LAST — the last hexadecimal address available
- USRORG — the current hexadecimal value of USRORG. (On startup, USRORG is set to the top of RAM.)

It then prompts you for a new value for USRORG. If you want USRORG to remain the same, press (ENTER).

If you want to enter a new value, it must be between the FIRST address and LAST address. Otherwise, you will get a BAD MEMORY error.

EDTASM Systems: Set USRORG to $6050:

```
USRORG=6050 (ENTER)
```

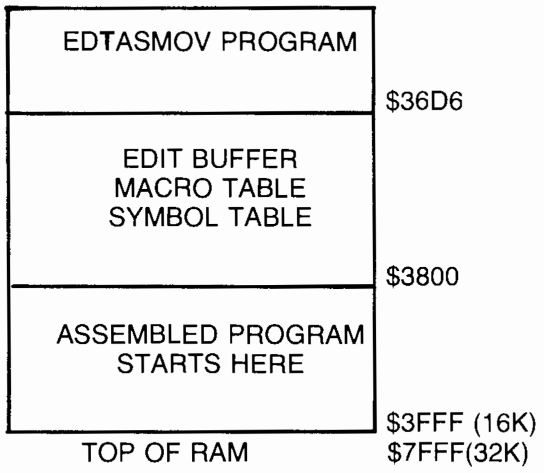
EDTASMOV Systems: Set USRORG to $3800:

```
USRORG=3800 (ENTER)
```

After setting USRORG, you can assemble the program at the USRORG address. Type:

```
A/IM/MO (ENTER)
```

Your assembled program now starts at Address $6050 or $3800:



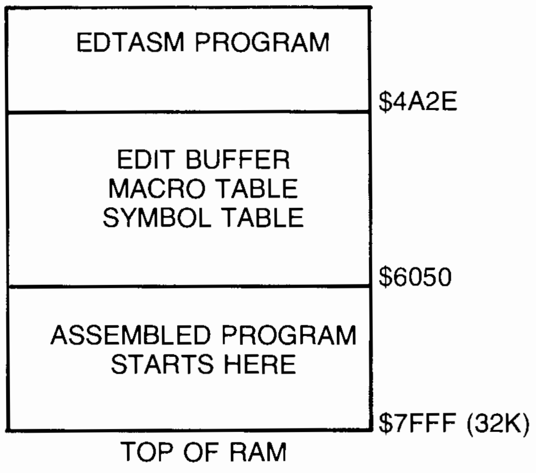


Figure 4. /MO In-Memory Assembly.

Disk Assembly

When you specify a filespec in the assembler command, the assembler saves the assembled program on disk. You can then load the program from one of these systems:

- DOS (to run as a stand-alone program)
- ZBUG (to debug with the stand-alone ZBUG program)
- BASIC (to call from a BASIC program)

The program originates at the address you specify in the ORG instruction.

What address you should use as the originating address depends upon which of the three systems you will be loading it into.

Assembling for DOS

Reference J shows the memory map that is in effect when DOS is loaded. As you can see, DOS consumes all the memory up to Address $1200. This means you must originate the program after $1200 or you will overwrite DOS.

In the sample program, reinsert the ORG $1200 instruction:

```
50      ORG  $1200
```

and assemble it to disk by typing:

```
A SAMPLE /SR (ENTER)
```

Note the /SR switch. You must use /SR when assembling to disk a program that you plan to load back into DOS. This puts the program in the format expected by DOS.

The assembler saves SAMPLE/BIN to disk with a starting address of $1200. You can now load and execute SAMPLE/BIN from the DOS menu.

Assembling for Stand-Alone ZBUG (EDTASMOV Users)

If you plan to use the stand-alone ZBUG for debugging your program, you need to save the program on disk so that you can load it into ZBUG.

Reference J also shows the memory map that is in effect when ZBUG is loaded. As you can see, you must use an originating address of at least $3800 or you will overwrite ZBUG. Change the ORG instruction to:

```
50      ORG  $3800
```

So that you can test this from ZBUG, without the program returning to BASIC, you need to change the ending of it. First, delete the CLR instruction in Line 170:

```
D170 (ENTER)
```

Then, change the JMP instruction in Line 180 to this:

```
180 SWI
```

After making the changes to the program, assemble it to disk by typing:

```
A SAMPLE/BUG /WS (ENTER)
```

The assembler saves SAMPLE/BUG on disk with a starting address of $3800. The /WS switch causes the assembler to save the symbol table also.

Hints On Assembly

- Use a symbol to label the beginning of your program.
- When doing an in-memory assembly on a program with an ORG instruction, you may want to use the /AO switch. Otherwise, the assembler will not use ORG as the program's originating address. It will use it to offset (add to) the loading address.
- The /WE switch is an excellent debugging tool. Use it to detect assembly errors before debugging the program.
- If you would like to examine the edit buffer and symbol table after an in-memory assembly, use ZBUG to examine the appropriate memory locations.

Chapter 8/ Debugging the Program (ZBUG Commands — Part II)

ZBUG has some powerful tools for a trial run of your assembled program. You can use them to look at each register, every flag, and every memory address during every step of running the program.

Before reading any further, you might want to review the ZBUG commands you learned in *Chapter 5*. We will be using these commands here.

Preparing the Program for ZBUG

In this chapter, we'll use the sample program from *Chapter 2* to show how to test a program. How you load the program into ZBUG depends on whether you are using EDTASM's ZBUG program or the stand-alone ZBUG program.

EDTASM ZBUG:

If you are using EDTASM, you can use EDTASM's ZBUG program.

1. Load SAMPLE/ASM into EDTASM (if it's not already loaded).
2. So that your program will be in the same area of memory as ours, change the ORG instruction to:

```
50        ORG       $5800
```

3. So that you can test the program properly from ZBUG (without the program returning to BASIC), you need to change the program's ending. First, delete the CLR instruction in Line 170:

```
D170 (ENTER)
```

   Then, change the JMP instruction in Line 180 to this:

```
180       SWI
```

4. Assemble the program in memory using the /IM and /AO switches. At the * prompt, type:

```
A/IM/AO (ENTER)
```

5. Enter ZBUG. At the * prompt, type:

```
Z (ENTER)
```

   When the # prompt appears, you're in ZBUG and can test the sample program.

Stand-Alone ZBUG:

If you are using EDTASMOV, you should use the Stand-Alone ZBUG.

1. Assemble SAMPLE/BUG to disk as instructed in the last chapter ("Assembling for Stand-Alone ZBUG").
2. Return to DOS and execute the stand-alone ZBUG program:

```
EXECUTE A PROGRAM
PROGRAM NAME [ZBUG    ]/BIN
```

   ZBUG loads and displays its # prompt.

3. Load SAMPLE/BUG, along with its symbol table, into ZBUG. Type:

```
LDS SAMPLE/BUG (ENTER)
```

   When the # prompt appears, you're ready to test the sample program with ZBUG.

Display Modes

In *Chapter 5*, we discussed four examination modes. ZBUG also has three display modes.

We'll examine each of these display modes from the mnemonic examination mode. If you're not in this mode, type M (ENTER) to get into it.

Numeric Mode

Type:

```
N (ENTER)
```

and examine the memory addresses that contain your program: $5800-$5817 for EDTASM's ZBUG or $3800-$3817 for Stand-Alone ZBUG.

In the numeric mode, you do not see any of the symbols in your program (BEGIN, START, SCREEN, WAIT, and DONE). All you see are numbers. For example, with EDTASM's ZBUG, Address $580F shows the instruction BNE 580A rather than BNE SCREEN.

Symbolic Mode

From the command level, type:

```
S (ENTER)
```

and examine your program again. ZBUG displays your entire program in terms of its symbols (BEGIN, START, SCREEN, WAIT, and DONE). Examine the memory address containing the BNE SCREEN instruction and type:

```
;
```

The semicolon causes ZBUG to display the operand (SCREEN) as a number (580A or 380A).

Half-Symbolic Mode

From the command level, type:

```
H (ENTER)
```

and examine the program. Now all the memory addresses (on the left) are shown as symbols, but the operands (on the right) are shown as numbers.

Using Symbols to Examine Memory

Since ZBUG understands symbols, you can use them in your commands. For example, with EDTASM's ZBUG, both these commands open the same memory address no matter which display mode you are in:

```
BEGIN/
5800/
```

Both of these commands get ZBUG to display your entire program:

```
T  BEGIN DONE
T  5800 5817
```

You can print this same listing on your printer by substituting TH for T.

Executing the Program

You can run your program from ZBUG using the G (Go) command followed by the program's start address:

EDTASM ZBUG: Type either of the following:

```
GBEGIN (ENTER)
G5800 (ENTER)
```

Stand-Alone ZBUG: Type either of the following:

```
GBEGIN (ENTER)
G3800 (ENTER)
```

The program executes, filling all of your screen with a pattern made up of F9 graphics characters. If you don't get this pattern, the program probably has a "bug." The rest of the chapter discusses program bugs.

After executing the program, ZBUG displays 8 BRK @ 5817, 8 BRK @ 3817, or 8 BRK @ DONE. This tells you the program stopped executing at the SWI instruction located at Address DONE. ZBUG interprets your closing SWI instruction as the eighth or final "breakpoint" (discussed below).

Setting Breakpoints

If your program doesn't work properly, you might find it easier to debug it if you break it up into small units and run each unit separately. From the command level, type X followed by the address where you want execution to break.

We'll set a breakpoint at the first address that contains the symbol SCREEN: $580A for EDTASM's ZBUG or 380A for Stand-Alone ZBUG.

EDTASM ZBUG: Type either of the following:

```
XSCREEN (ENTER)
X580A (ENTER)
```

Stand-Alone ZBUG: Type either of the following:

```
XSCREEN (ENTER)
X38ØA (ENTER)
```

Now type GBEGIN (ENTER) to execute the program. Each time execution breaks, type:

```
C (ENTER)
```

to continue. A graphics character appears on the screen each time ZBUG executes the SCREEN loop. (The characters appear to be in different positions because of scrolling.You will not see the first 32 characters because they scroll off the screen.)

Type:

```
D (ENTER)
```

to display all the breakpoints you have set. (You may set up to eight breakpoints numbered 0 through 7.)

Type:

```
C1Ø (ENTER)
```

and the tenth time ZBUG encounters that breakpoint, it halts execution.

Type:

```
Y (ENTER)
```

This is the command to "yank" (delete) all breakpoints. You can also delete a specific breakpoint. For example:

```
YØ (ENTER)
```

This deletes the first breakpoint (Breakpoint 0).

You may not set a breakpoint in a ROM routine. If you set a breakpoint at the point where you are calling a ROM routine, the C command will not let you continue.

Examining Registers and Flags

Type:

```
R (ENTER)
```

What you see are the contents of every register during this stage of program execution. (See *Chapter 10* for definition of all the 6809 registers and flags.)

Look at Register CC (the Condition Code). Notice the letters to the right of it. These are the flags that are set in Register CC. The E, for example, means the E flag is set.

Type:

```
X/
```

and ZBUG displays only the contents of Register X. You can change this in the same way you change the contents of memory. Type:

```
Ø (ENTER)
```

and the Register X now contains a zero.

Stepping Through the Program

Type:

```
BEGIN,      Note the comma!
```

LDA #$F9 is the next instruction to be executed. The first instruction, JMP START, has just been executed. To see the next instruction, type:

```
,      Simply a comma
```

Now, LDA #$F9 has been executed and LDX #$500 is the next. Type:

```
R (ENTER)
```

and you'll see this instruction has loaded Register A with $F9.

Use the comma and R command to continue single-stepping through the program examining the registers at will. If you manage to reach the JSR [$A000] instruction, ZBUG prints:

```
CAN'T CONTINUE
```

ZBUG cannot single-step through a ROM routine or through some of the DOS routines.

Transferring a Block of Memory

EDTASM ZBUG: Type:

```
U 58ØØ 5ØØØ 6 (ENTER)
```

Stand-Alone ZBUG: Type:

```
U 38ØØ 385Ø 6 (ENTER)
```

Now the first six bytes of your program have been copied to memory addresses beginning at 5000 or 3850.

Saving Memory to Disk

To save a block of memory from ZBUG, including the symbol table, type:

EDTASM ZBUG: `PS TEST/BUG 5800 5817 5800` (ENTER)

Stand-Alone ZBUG: `PS TEST/BUG 3800 3817 3800` (ENTER)

This saves your program on disk, beginning at Address 5800 (or 3800) and ending at Address 5817 (or 3817). The last address is where your program begins execution when you load it back into memory. In this case, this address is the same as the start address.

To load TEST/BUG and its symbol table back into ZBUG, type:

```
LDS TEST/BUG (ENTER)
```

Hints on Debugging

- Don't expect your first program to work the first time. Have patience. Most new programs have bugs. Debugging is a fact of life for all programmers, not just beginners.
- Be sure to make a copy of what you have in the edit buffer before executing the program. The edit buffer is not protected from machine language programs.

Chapter 9/ Using the ZBUG Calculator (ZBUG Commands — Part III)

ZBUG has a built-in calculator that performs arithmetic, relational, and logical operations. Also, it lets you use three different numbering systems, ASCII characters, and symbols.

This chapter contains many examples of how to use the calculator. Some of these examples use the same assembled program that we used in the last chapter.

> **Stand-Alone ZBUG:** Some of the memory addresses we use in the examples are too high for your system. Subtract $1000 from all the hexadecimal addresses and 4096 from all the decimal numbers.

Numbering System Modes

ZBUG recognizes numbers in three numbering systems: hexadecimal (Base 16), decimal (Base 10), and octal (Base 8).

Output Mode

The output mode determines which numbering system ZBUG uses to output (display) numbers. From the ZBUG command level, type:

```
O1Ø (ENTER)
```

Examine memory. The T at the end of each number stands for Base 10. Type:

```
O8 (ENTER)
```

Examine memory. The Q at the end of each number stands for Base 8. Type:

```
O16 (ENTER)
```

You're now back in Base 16, the default output mode.

Input Mode

You can change input modes in the same way you change output modes. For example, type:

```
I1Ø (ENTER)
```

Now, ZBUG interprets any number you input as a Base 10 number. For example, if you are in this mode and type:

```
T 49152 49162 (ENTER)
```

ZSBUG shows you memory addresses 49152 (Base 10) through 49162 (Base 10). Note that what is printed on the screen is determined by the output mode, not the input mode.

You can use these special characters to "override" your input mode:

BASE	BEFORE NUMBER	AFTER NUMBER
Base 10	&	T
Base 16	$	H
Base 8	@	Q

Table 1. Special Input Mode Characters

For example, while still in the I10 mode, type:

```
T 49152  $CØ1Ø (ENTER)
```

The "$" overrides the I10 mode. ZBUG, therefore, interprets C010 as a hexadecimal number. As another example, get into the I16 mode and type:

```
T 49152T  CØ1Ø (ENTER)
```

Here, the "T" overrides the I16 mode. ZBUG interprets 49152 as decimal.

Operations

ZBUG performs many kinds of operations for you. For example, type:

```
C000+25T/
```

and ZBUG goes to memory address C019 (Base 16), the sum of C000 (Base 16) and 25 (Base 10). If you simply want ZBUG to print the results of this calculation, type:

```
C000+25T=
```

On the following pages, we'll use the terms "operands," "operators," and "operation." An operation is any calculation you want ZBUG to solve. In this operation:

```
1 + 2 =
```

"1" and "2" are the operands. "+" is the operator.

Operands

You may use any of these as operands:.

1. ASCII characters
2. Symbols
3. Numbers (in either Base 8, 10, or 16) — Please note that ZBUG recognizes integers (whole numbers) only

Examples (Get into the 016 mode):

```
'A=
```

prints 41, the ASCII hexadecimal code for "A".

```
START=
```

prints the START address of the sample program. (It will print UNDEFINDED SYMBOL if you don't have the sample program assembled in memory.)

```
15Q=
```

prints the hexadecimal equivalent of octal 15.

If you want your results printed in a different numbering system, use a different output mode. For example, get into the O10 mode and try the above examples again.

Operators

You may use arithmetic, relational, or logical operators. (Get into the O16 mode for the following examples.)

Arithmetic Operators

Addition	+
Subtraction	–
Multiplication	*
Division	.DIV.
Modulus	.MOD.
Positive	+
Negative	–

Examples:

```
DONE-START=
```

prints the length of the sample program (not including the SWI at the end).

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

prints 4. (ZBUG can divide integers only.)

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

prints 1, the remainder of 9 divided by 2.

```
1-2=
```

prints OFFFF,65535T, or 177777Q, depending on which output mode you are in. ZBUG does not use negative numbers. Instead, it uses a "number circle" which operates on modulus 10000 (hexadecimal):

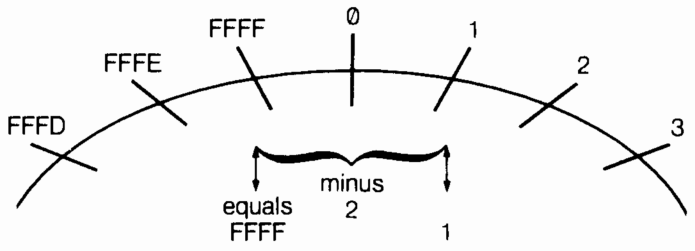


Figure 5. Number Circle Illustration of Memory.

To understand this number circle, you can use the clock as an analogy. A clock operates on modulus 12 in the same way the ZBUG operates on modulus 10000. Therefore, on a clock, 1:00 minus 2 equals 11:00:

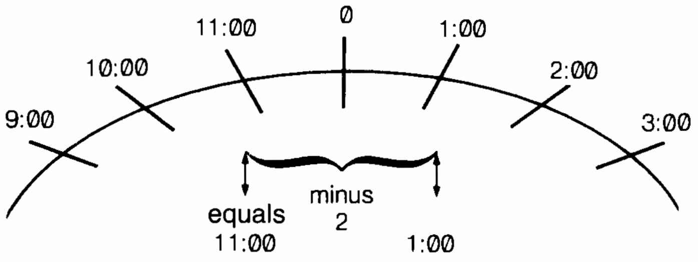


Figure 6. Number Circle Illustration of Clock.

Relational Operators

Equal to	`.EQU.`
Not Equal to	`.NEQ.`

These operators determine whether a relationship is true or false.

Examples:

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

prints 0FFFF, since the relationship is true. (ZBUG prints 65535T in the O10 mode or 177777Q in the O8 mode.)

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

prints 0, since the relationship is false.

Logical Operators

Shift	`<`
LogicalAND	`.AND.`
InclusiveOR	`.OR.`
ExclusiveOR	`.XOR.`
Complement	`.NOT.`

Logical operators perform bit manipulation on binary numbers. To understand bit manipulation, see the 6809 assembly language book we referred to in the introduction.

Examples:

```
10<2=
```

shifts 10 two bits to the left to equal 40. The 6809 SL instruction also performs this operation.

```
10<-2=
```

shifts 10 two bits to the right to equal 4. The 6809 ASR instruction also performs this operation.

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

prints 3, the exclusive or of 6 and 5. The 6809 EOR instruction also performs this operation.

Complex Operations

ZBUG calculates complex operations in this order:

```
+    .DIV.    .MOD.    <
          .AND.
      .OR.     .XOR
          +    -
     .EQU.    .NEQ.
```

You may use parentheses to change this order.

Examples:

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

The division is performed first.

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

The addition is performed first.

```
4*4.DIV.4=
```

The multiplication is performed first.