DISK EDTASM

COLOR COMPUTER
DISK EDITOR ASSEMBLER WITH ZBUG

CUSTOM MANUFACTURED
IN USA BY RADIO SHACK
A DIVISION OF TANDY CORPORATION

EDTASM6309 64K Edition

NO WARRANTY

EDTASM6309 V3.03.03 - Robert Gault

https://colorcomputerarchive.com/repo/Disks/Programming/EDTASM6309 (Robert Gault).zip

EDTASM6309 is now in the CoCo archive.

I did not want this very powerful package to be lost and I have not sold it in years.

In the .ZIP are versions for most CoCo versions. If you are using a modern flat screen instead of an old TV monitor, you can work with an 80 character line.

To include some of the 6309 code into the original disk EDTASM, there are some instructions that don't match normal 6309 code but documentation is included in the package.

There is a version that works nicely with VCC. There is also one for SDC modified CoCos.

Have fun!

Robert Gault

This is a major patch to Tandy's Disk EDTASM (editor/assembler) and offers many improvements over the original version and earlier patch versions. The program comes in two versions, one for the Coco 1 or 2 and the second for the Coco 3. The only user noticeable difference is that the Coco3 version uses 40/80 column screens and runs in the fast 2MHz mode.

Use RGBCNFIG.BAS to select whether the 40 or 80 column screen will be used.

Changes to original EDTASM (TANDY VERSION)

1) Tape is no longer supported; code has been removed.

2) Buffer size increased to over 42K bytes.

3) Directory obtainable from both Editor and ZBUG; V command.

4) Multiple FCB and FDB data per line.

5) FCS supported.

6) SET command now works properly.

7) Screen colors remain as set from Basic before starting EDTASM.

8) Symbol table printed in five names per line on Coco3.

9) On assembly with /NL option, actual errors are printed.

10) Warning error on long branch where short is possible.

11) ZBUG now defaults to numeric instead of symbolic mode.

12) RGB DOS users now have support for drive numbers higher than 3.

13) Hitachi 6309 opcodes are now supported for both assembly and disassembly including latest discoveries.

14) HD6309 detection is included and if present incorporates a ZBUG error trap

for illegal opcodes and enables monitoring and changing the E,F,V registers from ZBUG.

15) Coco 3 users can now safely exit to Basic and use their RESET button from

EDTASM.

16) Keyboard now has auto repeat keys when keys are held down.

17) Lower case is now supported for commands, opcodes, options, and symbols. Take care when loading or saving files or using symbols, ex. NAME does not equal name, \.A not= \.a, etc.

18) Local names are now supported. is A@-Z@ and a@-z@ for 52 local symbols. New sets of locals are

started after each blank line in the source code. Local symbols do not appear in or clutter symbol table.

19) Local symbols can only be accessed from ZBUG in expanded form: ex. A@00023 not A@.

20) Now reads source code files that don't have line numbers. Writes normal source files with line numbers (W filename) or without line numbers (W# filename).

21) Macro parameters now function correctly from INCLUDED files.

22) While in the Editor, the U key will backup one screen in your source file.

23) DOS.BAS can be used to program the F1 and F2 keys on a Coco3. See below.

24) Coco3 WIDTH80 now uses 28 lines of text.

NOTE:

Coco 1&2 versions do require 64K RAM, the Coco 3 version will work with 128K of RAM. You can assemble 6309 code even if your Coco has a 6809 cpu.

NEW COMMANDS

V - obtains a directory from either Editor or ZBUG modes.

U - scrolls backwards through source code.

FCS - is used exactly like FCC but automatically add $80 to the last character in the string.

FCB, FDB - for multiple entries per line entries should be seperated by a comma. Make sure that the comment field for that line DOES NOT CONTAIN ANY

COMMAS or an error will result.

6309 OPCODES - since there is no endorsed memnonic convention for these code, the syntax of the Hirotsugo Kakugawa paper "A Memo on the Secret Features of the 6309" has been followed with one exception.

TFM (transfer block) have the following format:

H.K format	EDTASM format
TFR r1+,r2+	TFRP r1,r2 ie. plus
TFR r1-,r2-	TFRM r1,r2 ie. minus
TFR r1+,r2	TFRS r1,r2 ie. send
TFR r1, r2+	TFRR r1,r2 ie. receive

SUPPORT FOR 6309 OPCODES

The product disk contains several text files describing the 6309 and the new opcodes. These can be accessed with any word processing program or may be printed by a simple READER program also included on the disk

GENERAL INSTRUCTIONS AND CAVEATS

The version of DOS which is used with EDTASM6309 is patched specifically for the new program. Do not use it as a general purpose DOS as explained in the EDTASM manual, instead use the old version.

Do not attempt to test any HSCREEN functions from ZBUG as changes made to the system MMU will cause program failure. For the same reason do not change bytes in the region of $400-$600.

Stand alone programs may do whatever they like.

If you are testing a routine from ZBUG which accesses the ROMs you must conform to the following practices. Make sure that your stack register points below $8000 and execute your routine as shown below:

```
Coco3 users          Coco 1&2 users

PSHS A,CC     STA $FFDE
ORCC #$50
CLR $FF91
go do the routine    go do the routine

LDA #1        STA $FFDF
STA $FF91
PULS A,CC
```

This is required because the extended buffer may cause your code to extend over and corrupt the RAM image of the ROMS.

EDTASM6309 64K Edition

INSTALLATION

With Tandy out of business, your EDTASM is ready to use. Just select the appropriate version from the zip directories and mount it in an emulator. With a hardware Coco, you will need to either mount your disk on Drivewire to use or for transfer to a real Coco disk.

IF HELP IS NEEDED

Every effort has been made to create an error free program but errors can happen. The best place to get help in 2023 and beyond is probably the COCO DISCORD “Assembly” Channel.

*******RGBDOS USERS ONLY******

You may now boot EDTASM6309 from any hard drive excluding drive255. To use this option, you must first copy DOS.BAS, DOS.BIN, RGBCNFIG.BAS, and EDTASM.BIN to the selected drive number. Run RGBCNFIG.BAS (RGB-Config) to patch DOS.BIN for booting from the selected drive.

!!!!!WARNING!!!!!

There is an unavoidable problem with hard drive use. There is only room for 4 FAT buffers. To access multiple drives, DOS now uses modulo math. However, there is no way to check for a drive in use under this system without a major rewrite of DOS.BIN. If this situation occurs during assembly through the use of INCLUDE, you can and will damage the directory track. It is left to the user to avoid "overlapping" drive numbers.

The use of "overlapping" is explained below. This situation is easily avoided by limiting the range of any one assembly to 4, ex. 3,4,5,6 37,38,39,40 etc. If you must use a wider range of drive numbers,

YOU MUST PRE-CALCULATE TO AVOID OVERLAP.

```
0  1  2  3     FAT BUFFER NUMBER
0  1  2  3     ACTUAL DRIVE IN USE
4  5  6  7
8  9 10 11
12 13 14 15
```
etc.

```
FAT BUFFER FORMULA: FAT = 4 * ((drive number / 4) - INT(drive number / 4))
```

!!!DO NOT USE OVERLAPPING DRIVE NUMBERS DURING ASSEMBLY!!!

There is also a bug in DOS.BIN for which a correction has not been found. Files with 256 bytes in the last sector can not be written or loaded correctly.

If you get strange errors related to bad line number or line too long, add or subtract a one byte character from the file; the error will probably vanish.

USEFUL HINTS

Coco3 users now have a line in DOS.BAS: 0 WIDTH32.

If you didn't do this and started DOS.BAS from a high res. text screen, you wouldn't be able to read any of the DOS.BAS instructions.

You can make DOS.BAS autostart EDTASM (that's what EDTASM6309 is called on the disk) with the line:

```
1 REMEDTASM      IS NOW LOADING
```

That also has been added to the DOS.BAS program.

Coco3 users can set the F keys to perform often repeated tasks. Check DOS.BAS after installation for examples of Assembly and Write. You will need to use the file name of your project and change lines 3 & 4.

[Lets just ignore this part]

To Our Customers ...

The heart of the Color Computer is a 6809E, and in some cases a 6309 "processor." It controls all other parts of the Color Computer.

The processor understands only a code of Os and 1 s, not at all intelligible to the human mind. This code is called "6809 machine code."

When you run a BASIC program, a system called the "BASIC Interpreter" translates each statement, one at a time, into 6809 machine code. This is an easy way to
program, but inefficient.

The Disk EDTASM program lets you program using an intelligible representation of 6809 machine code, called "assembly language," that talks directly to the processor. You then assemble the entire program into 6809 machine code before running it.

Programming with the Disk EDTASM tools gives you these benefits:

• You have direct and complete control of the Color Computer. You can use its features, such as high resolution graphics, in ways that are impossible with BASIC.

• Your program runs faster. This is because it is already translated into 6809 machine code when you run it.

To Use the Disk EDTASM You Need ...

A Color Computer Disk System that has at least 64K of RAM, preferably MORE.

The Disk EDTASM Contains:

• `EDTASM/BIN`, a system for creating 6809 programs. **EDTASM** contains:

1. An editor, for writing and editing 6809 assembly language programs.
2. An assembler, for assembling the programs into 6809 machine code.
3. **ZBUG**, for examining and debugging 6809 machine-code programs.

• `DOS/BIN`, a disk operating system. **`DOS`** contains disk access routines that you can call from an assembly language program. (*You cannot call **BASIC**'s disk access routines with any program other than **BASIC**.*)

`EDTASM/BIN`, useS **`DOS`** routines and must be run with the **`DOS`** `program`.

The Disk EDTASM also contains:

• `DOS/BAS`. A **BASIC** program that loads **`DOS/BIN`**.

How to Use this Manual

This manual is organized for both beginning and advanced assembly language programmers. Sections I-IV are tutorials; Section V is reference.

Beginning Programmers:

Read Section I first. It shows how the entire system works and explains enough about assembly language to get you started.

Then, read Sections II, III, and IV in any order you want. Use Section V, "Reference," as a summary.

This manual does not try to teach you 6809/6309 mnemonics.

To learn this, read the following:

Radio Shack Catalog #62-2077
by William Barden Jr.

6809 Assembly Language Programming
by Lance A. Leventhal

Nor does it teach you disk programming concepts. To learn these, read:

Color Computer Disk System Manual
(Radio Shack Catalog #26-3022)

Advanced Programmers:

First, read Chapters 1 and 2 to get started and see how the entire system works. Then, read Section V, "Reference. "

You can use the DOS program listing to obtain information on routines and addresses not explained in this manual. Please note the following:

Radio Shack supports only these DOS routines: OPEN, CLOSE, READ, and WRITE. Additional DOS routines are listed in Reference H. However, Radio Shack does not promise to support them.

Even more DOS routines and addresses can be found in the program listing. However, Radio Shack does not promise to support them nor even provide them in the future.

This manual uses these terms and notations:

[KEY]:	To denote a key you must press.
Italics:	To denote a value you must supply.
filespec:	To denote·a DOS file name

A DOS filespec is in one of these formats:

filename/ext:drive

filename.ext:drive

filename has one to eight characters.

extension has one to three characters.

drive is the drive number. If the drive number is omitted, DOS uses the first available drive.

When specifying a hexadecimal (Base 16) number, for example $0F, it represents hexadecimal value 0F, which is equal to 15 in decimal (Base 10) notation.

NOTE:

Either a 40 or 80 column screen can be selected if you have a Coco3. The CTRL key toggles between the two screens. This feature should be used only at the Editor or ZBug prompts.

Table of Contents

SECTION I xiii

GETTING STARTED xiii

Chapter 1/ Preparing Diskettes 3

Chapter 2/ Running a Sample Program 5

Chapter 3/ Overview 9

SECTION II 12

COMMANDS 12

Chapter 4/ Using the DOS Menu (DOS Commands) 15

Chapter 5/ Examining Memory ZBUG Commands 17

Chapter 6/ Editing the Program Editor Commands 21

Chapter 7/ Assembling a Program (ASM Commands) 25

SECTION I

GETTING STARTED

SECTION I

GETTING STARTED

This section gets you started using the Disk EDTASM and explains some concepts you need to know.

Chapter 1/ Preparing Diskettes

Before using the Disk EDTASM, you need to format blank diskettes and back up the master Disk EDTASM diskette.

Formatting Blank Diskettes

1. Power up your disk system and insert a blank diskette in Drive 0. (See the Color Computer Disk System Manual for help.)

2. At the OK prompt, type:

`DSKINI0` [ENTER]

BASIC formats the diskette. When finished, it again shows the OK prompt.

Making Backups of Disk EDTASM

Single-Drive Systems

1. Insert the master Disk EDTASM diskette, your "source" diskette, in Drive0.

2. At the BASIC OK prompt, type:

`BACKUP 0 TO 0` [ENTER]

3. BASIC then prompts you to insert the "destination" diskette. Remove the source diskette and insert a formatted diskette. Press [ENTER]

4. BASIC prompts you to alternatively insert the source, then destination diskettes. When the backup is finished, the OK prompt appears.

The destination diskette is now a duplicate of the master Disk EDTASM diskette.

Multi-Drive Systems

• Insert the master Disk EDTASM diskette in Drive0
• Insert a formatted diskette in Drive 1
• At the BASIC OK prompt, type:
`BACKUP 0 TO 1` [ENTER]

BASIC makes the backup. When the backup is finished, the OK prompt appears.

The diskette in Drive1 is now a duplicate of the master Disk EDTASM diskette.

Chapter 2/ Running a Sample Program

This "sample session" gets you started writing programs and shows how to use the Disk EDTASM. The next chapters explain why the program works the way it does.

Load and Run DOS

Insert the Disk EDTASM diskette in Drive 0. At the OK prompt, type:

```
RUN "DOS"   [ENTER]
```

DOS then loads and puts you directly into EDTASM "command mode."

Type the Source Program

Notice the asterisk * prompt. This means you are in the editor program of EDTASM . The editor lets you type and edit an assembly language "source" program.

At the * prompt, type:

```
I   [ENTER]
```

This puts you in the editor's insert mode. The editor responds with line number 00100. Type:

```
START [→] LDA [→] #$F9  [ENTER]
```

The right arrow tabs to the next column. [ENTER] inserts the line in the editor's "edit buffer."

The $ means that F9 is a hexadecimal (Base 16) number.

Your screen should show:

```
00100  START LDA   #$F9
00110
```

meaning that you inserted line 100 and can now insert line 110.

If you make a mistake, press [BREAK]. Then, at the * prompt, delete Line 100 by typing:

```
D100 [ENTER]
```

Now, insert Line 100 correctly in the same manner described above.

Insert the entire assembly language program listed below.

Note that line 150 uses brackets. Do not substitute parentheses for the brackets.

To produce the left bracket, press [SHIFT] and [DOWN ARROW] at the same time.

To produce the right bracket, press [SHIFT] and [→] at the same time.

Program Entry (continued)

```
00100  START   LDA     #$F9
00110          LDX     #$400
00120  SCREEN  STA     .X+
00130CMPX     #$600
00140          BNE     SCREEN
00150  WAIT    JSR     [$A000]
00180          BEQ     WAIT
00170          CLR     $71
00180          JMP     [$FFFE]
00180  DONE    EQU     *
00200          END
```

If you make a mistake, press [BREAK]. Then, at the * prompt, delete the program by typing:

```
D#:*
```

After deletion, you can now type in the program again.

When finished, press [BREAK]. The program you have inserted is an assembly language "source" program, which we'll explain in the next chapter.

Assemble the Source Program in Memory

At the * prompt, type:

```
A/IM/WE  [ENTER]
```

which loads the assembler program. The assembler then assembles your source program into 6809/6309 machine code into the memory area just above the EDTASM program.

To let you know what has been done, EDTASM prints the resulting listing.

If the assembler does not print this entire listing, but stops and shows an error message instead, you have an error in the source program. Repeat Steps 3 and 4.

The assembler listing is explained in Figure 1 of Chapter 7.

Prepare the Program for DOS

Before saving the program, you need to prepare it so that you can load and run it from DOS.

First, you must give it an "origination address" for DOS to use in loading the program back into memory. (We recommend you use Address $7000, the first address available after the

```
4B28 88   F9      00100 START
  LDA   #$F9
4B2A 8E   0400   00110
  LDX   #$400
4B2D A7   80      00120 SCREEN
  STA   ,X+
4B2F 8C   0600   00130
  CMPX  #$600
4B32 26   F9      00140
  BNE   SCREEN
4B34 AD   9F A000  00150 WAIT
  JSR   []
4B38 27   FA      00160
  BEQ   WAIT
4B3A 0F   71      00170
  CLR   $71
4B3C 6E   9F FFFE  00180
  JMP   [$FFFE]
4B40            00190 DONE
  EQU   *
       0000   00200
  END

00000 TOTAL ERRORS
DONE    4B40
SCREEN  4B2D
START     4B28
WAIT    4B34
```

the DOS system.) To do so, type:

```
I50   [ENTER]
```

and insert this line:

```
50          ORG         $1200
```

Next, you need to add two lines to your program to tell DOS how long the program is. Insert these lines:

```
60 BEGIN    JMP   START
70          FDB   DONE-BEGIN
```

When finished, press [BREAK]. To see the entire program, type:

```
P#:*    [ENTER]
```

It should look like this:

```
00050          ORG      $1200
00060 BEGIN    JMP      START
00070          FDB      DONE-BEGIN
00100 START    LDA      #$F9
00110          LDX      #$400
00120 SCREEN   STA      ,X+
00130          CMPX     #$600
00140          BNE      SCREEN
00150 WAIT     JSR      [$A000]
00180          BEQ      WAIT
00170          CLR      $71
00180          JMP      [$FFFE]
00180 DONE     EQU      *
00200          END
```

If you make a mistake, delete the line with the error and insert it again.

Save the Source Program on Disk

To save the source program, type (at the * prompt):

```
WD SAMPLE [ENTER]
```

This saves the source program on disk as

```
SAMPLE/ASM
```

Save the Assembled Program on Disk

At the * prompt, type:

```
AD SAMPLE /SR  [ENTER]
```

Be sure you have a blank space between SAMPLE and /SR.

This causes the assembler to again assemble the source program into 6809/6309 code. This time, the Assembler saves the assembled program on disk as

```
SAMPLE/BIN
```

(You must use the /SR "switch" to assemble any program that you want to load and run from DOS.)

Run the Assembled Program from DOS

To run the assembled program, you need to be in the DOS command mode. At the * prompt, type:

```
K [ENTER]
```

which causes the Editor to return you to the DOS command menu. Press [2] to execute a program. Then type SAMPLE, the name of the assembled program. (The assembler assumes you mean SAMPLE/BIN.)

The screen shows:

```
EXECUTE A PROGRAM
PROGRAM NAME: [SAMPLE ]/BIN
```

Press [ENTER].

The SAMPLE program executes, filling your entire screen with a graphics checkerboard.

Press any key to exit the program. The program returns to BASIC startup message.

Debug the Program (if necessary)

ZBUG lets you to look at memory.

At the * prompt, type:

```
Z  [ENTER]
```

EDTASM loads its ZBUG program and displays ZBUG's # prompt. You can now examine any memory address.

Type:

```
4000/
```

and ZBUG shows you what is in memory at this address. Press [DOWN ARROW] a few times to look at more memory addresses. When finished, press [BREAK].

In Chapter 8, we'll show you how to use ZBUG to examine and test your program. To return to EDTASM's editor, type:

```
E  [ENTER]
```

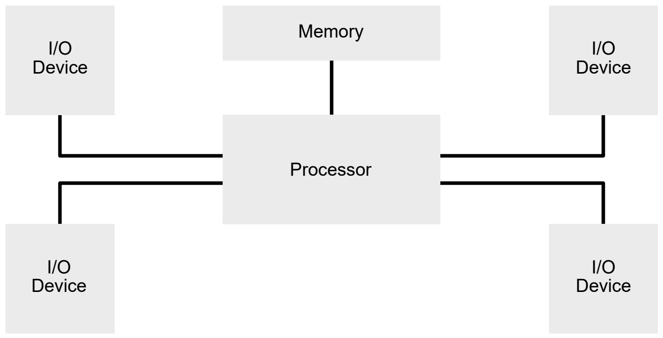
Chapter 3/ Overview

This chapter is for beginning assembly language programmers. It explains some concepts you need. If you're not a beginner, use this chapter as a refresher or skip it.

The Color Computer Hardware

The Color Computer consists of:

• The 6809E Processor
• Memory
• Input/Output Devices
This shows how they relate to each other:



The Processor

The processor processes all data going to each memory address and device. It contains:

• Registers - for temporarily storing 1- or 2-byte values.

• Buses - for transferring data to or from the processor.

All instructions to the processor must be in 6809

machine code: a code of Os and 1 s containing

"opcodes" and data. "Opcodes" are instructions that tell the processor to manipulate data in some way.

For example, the machine-code instruction "10000110 11111001" contains:

The opcode "10000110" (decimal 134 or hexadecimal 86)

The data "11111001" (decimal 249 or hexadecimal F9)

This instruction tells the processor to load Register A with 11111001.

Memory

Memory is a storage area for programs and data. There are two kinds of memory:

• Random access memory (RAM) - for temporary storage of programs or data. When you load a program ROM disk, you load it Into RAM. Many opcodes store data in RAM temporarily.

• Read only memory (ROM) - for permanent storage of programs. BASIC, as well as any program pack you use, is stored in ROM. The Color Computer contains several "ROM routines" that you can use to access the keyboard, screen, or tape recorder.

When writing an assembly language program, you must constantly be aware of What's happening in memory. For this reason, this manual provides a memory map. (See Reference J.)

Devices

All other parts of the hardware are called devices. A device expects the processor to input or output data to it in a certain format. To input or output data in this format, you can use these pre-programmed subroutines:

• Routines stored in ROM (ROM routines) - for inputting or outputting to the keyboard, screen, printer, or tape recorder.

• Routines stored in DOS (DOS routines) - for inputting or outputting to disk.

The Disk EDTASM Assembler

The Disk EDT ASM looks for three fields in your instructions: label, command, and operand. For example, in this instruction:

```
BEGIN  JMP START
```

BEGIN is the label. JMP is the command. START is the operand.

In the label field, it looks for:

- Symbols (symbolic names)

In the command field, it looks for:
- Mnemonics
- Pseudo Ops

In the operand field, it looks for:
- Symbols
- Operators
- Addressing-Mode Characters
- Data

Symbols

A symbol is similar to a variable. It can represent a value or a location. BEGIN (in the sample session) is a symbol that represents the location of the instruction JMP START. START is also a symbol that represents the location of LDA #$F9.

Mnemonics

A mnemonic is a symbolic representation of an opcode.

It is a command to the processor. "LDA" is a mnemonic. Depending on which "addressing-mode character" you use, LDA represents one of these opcodes:

10000110
10010110
10110110
10100110

(Addressing-mode characters are discussed below.)

Mnemonics are specific to a particular processor. For example, Radio Shack's Model 4 uses the Z80 processor, which understands Z80 mnemonics, rather than the 6809/6309 mnemonics.

Pseudo Ops

A pseudo op is a command to the assembler. END (in the sample session) is a pseudo op. It tells the assembler to quit assembling the program.

Data

Data is numbers or characters. Many of the mnemonics and pseudo ops call for data. Unless you use an operator (described next), the assembler interprets your data as a decimal (Base 10) number.

Operators

An operator tells the assembler to perform a certain operation on the data. In the value $1200, the $ sign is an operator. It tells the assembler that 1200 is a hexadecimal (Base 16) number, rather than a decimal (Base 10) number.

The more commonly used operators are arithmetic and relational. Addition (+) and equation (=) are examples of these operators.

In the value %10110101, the % sign is an operator. It tells the assembler that 10110101 is a binary (Base 2) number. This notation may now be used in both data and equations.

Addressing-Mode Characters

An addressing mode character tells the assembler how it should interpret the mnemonic. The assembler then assembles the mnemonic into the appropriate opcode.

The sample session uses the # character with the LDA mnemonic to denote the "immediate" addressing mode. This causes the assembler to assemble LDA into the opcode 10000110.

The immediate mode means that the number following the mnemonic (in this case, $F9) is data rather than an address where the data is stored.

Pseudo ops, symbols, operators, and addressing-mode characters vary from one assembler to another. Section 11/ explains them in detail.

Sample Program

This is how each line in the sample program works:

```
 50             ORG     $1200
```

ORG is a pseudo op for "originate." It tells the assembler to begin loading the program at Location $1200 (Hexadecimal 1200). This means that when you load and run the program from DOS, the program starts at Memory Address $1200.

```
 60      BEGIN   JMP     START
```

BEGIN is a symbol. It equals the location where the JMP START instruction is stored.

JMP is a mnemonic for "jump to an address." It causes the processor to jump to the location of the program labeled by the symbol START, which is the LDA #$F9 instruction.

You must use JMP or LBRA as the first instruction in a DOS program.

```
 70             FDB     DONE-BEGIN
```

FDB is a pseudo op for "store a 2-byte value in memory." It stores the value of DONE-BEGIN (the length of the program) in the next two bytes of memory.

You must store this value at the beginning of the program to tell DOS how much of the program to load.

```
 00100 START LDA    #$F8
```

START is a symbol. It equals the location where LDA #$F9 is stored.

LDA is a mnemonic for "Load Register A". It loads Register A with $F9, which is the hexadecimal ASCII code for a graphics character. The ASCII characters are listed in Reference K.

```
 00110          LDX     #$400
```

LDX is a mnemonic for "load Register X." It loads Register X with $400, the first address of video memory. Reference J shows where video memory begins and ends.

```
 00120 SCREEN STA .X+
```

SCREEN is a symbol. It equals the location where STA ,X+ is stored.

STA is a mnemonic for "store Register A" It stores the contents of Register A ($F9) in the address contained in Register X ($400). This puts the $F9 graphics character at the upper left corner of your screen.

The "," and "+" are addressing-mode characters. The “,” causes the processor to store $F9 in the address contained in Register X. The “+” causes the processor to then increment the contents of Register X to $401.

```
 00130       CMPX  #$600
```

CMPX is a mnemonic for "compare Register X." It compares the contents of Register X with $600. If Register X contains $600, the processor sets the "Z" bit in the Register CC to 1.

```
 00140       BNE     SCREEN
```

BNE is a mnemonic for "branch if not equal." It tells the processor return to SCREEN (the STA,X + instruction) until the Z bit is set.

The BNE SCREEN instruction creates a loop. The program branches back to SCREEN, filling all video memory addresses with $F9, until it fills Address $600. At that time, Register X contains $600, Bit Z is set, and program control continues to the next instruction.

```
 00150 WAIT   JSR     [$A000]
```

JSR is a mnemonic for "jump to a subroutine." $A000 is a memory address that stores the address of a ROM routine called POLCAT. (See Reference F.)

POLCAT scans the keyboard to see if you press a key. When you do, it clears the Z bit.

The "[ ]" are addressing-mode characters. They tell the processor to use an address contained in an address, rather than the address itself. Always use the "[ ]" signs when calling ROM routines.

```
 00160       BEQ     WAIT
```

BEQ is a mnemonic for "branch if equal." It branches to the JSR [$A000] instruction until the Z bit is clear. This causes the program to loop until you press a key, at which time POLCAT clears the Z bit.

```
 00170          CLR     $71
 00180       JMP     [$FFFEJ
```

CLR is a mnemonic for "clear," and JMP is a mnemonic for "jump to memory address." These two instructions end the program and return to BASIC's startup message.

(CLR inserts a zero in Address $71 ; this signals that the system is at its original "uninitialized" condition. JMP goes to the address contained in Address $FFFE; this is where BASIC initialization begins.)

```
 00180 DONE   EQU    *
```

EQU is a pseudo op. It equates the symbol DONE with an asterisk (*), which represents the last line in the program.

```
 00180       END
```

END is a pseudo op. It tells the assembler to quit assembling the program.

SECTION II

COMMANDS

SECTION il

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 Drive0.

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 [UP ARROW] to move the cursor left or [DOWN ARROW] to move right.

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

```
[SAMPLE**] [***] :[0] <FILE SPEC
```

Use the [SPACE BAR] 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] :[0] <FILE SPEC
```

Press [ENTER] when finished. Return to the main menu.

To see all directory entries on the disk in Drive0, simply press [ENTER] without specifying a filename or extension:

```
[********] [***] : [0] <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 CD and CD to position the cursor. Answer the prompts as shown:

```
Source File Name       [SAMPLE ]
Extension              [ASM]
Drive                  [0]

Destination File Name  [COPY ]
Extension              [ASM]
Drive                  [0]

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                  [0]

Destination File Name  [COPY ]
Extension              [ASM]
Drive                  [0]

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 Drive0 to SAMPLE/ASM in Drive1. Answer the prompts as shown:

```
Source File Name       [SAMPLE ]
Extension              [ASM]
Drive                  [0]

Destination File Name  [COPY ]
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

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.

At the editor's * prompt, type:

```
Z [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 [DOWN ARROW]. Use [UP ARROW] to scroll to the preceding address.

Continue pressing [UP ARROW] or [DOWN ARROW]. Notice that as you use the [UP ARROW] 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 m increments the address by one. The m 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 [DOWN ARROW] key a few times. In this mode, the [DOWN ARROW] increments the address by two. The numbers contained in each address are the same, but you are seeing them two bytes or oneword at a time.

Press the [UP ARROW] a few times. The CD 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 [DOWN ARROW] 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 [DOWN ARROW] a few times. In the mnemonic mode,

ZBUG assumes you're examining an assembly language program. The [DOWN ARROW] 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 [DOWN ARROW]; 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:

```
015AI
```

Then type:

```
DD
[DOWN ARROW]
```

This changes the contents to DD and lets you change the next address. (Press the [DOWN ARROW] 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,8,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:

```
015AI
```

To change its contents to a C, type:

```
   'C
[DOWN ARROW]
```

Pressing the [UP ARROW] 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]

```
015A/
```

To change this instruction, type:

```
86
```
[ENTER]

Now Address $015A contains the opcode for the LDA

mnemonic. Open location 0158:

```
015B/
```

and insert $06, the operand:

```
06
```
[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:

From EDTASM ZBUG, return to the editor by typing E [ENTER]

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] + [@] KEYS

To continue, press any key.

In addition to the P range command you now also have U. This command will have the effect of scrolling backwards through your source code. The current line will be moved backwards one screen and a new screen will be printed from that point onward.

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#:*
```
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:

```
E100
```
[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
Drange

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

```
0110: 140
```
[ENTER]

Lines 110 through 140 are gone.

Insert Command
Istartllne, increment

Type:

```
I152,2
```
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 accidentally 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:

```
I200
```
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
Nstartline,increment

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

```
N100,50
```
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
```
The current line is now the first line number. Renumber now so you will be ready for the next instruction.

Type:

```
N100,10
```

Replace Command
Rstartline,increment

The replace command is a variation of the insert command.

Type:

```
R100,3
```
You may now replace line 100 with a new line and begin inserting lines using an increment of three.

Copy Command
Cstartline,range,increment

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

To copy lines, type:

```
C500,100:150,10
```
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 includes the ZBUG debugger program. This allows you to enter ZBUG while your program is still in memory.

To enter ZBUG, type:

```
Z
```
The # prompt tells you that you are now in ZBUG. To re-enter the editor from ZBUG, type the ZBUG command:

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

```
K
```
Entering DOS or BASIC empties your edit buffer. Reentering the editor empties your BASIC buffer.

Write Command
WD filespec

W 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:

filenamelext: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 3 with a floppy disk system or from 0 to driveMAX on an RGB hard drive system.

If the drive number is omitted, the editor uses the last drive accessed. At startup this will normally be drive #0 unless you patch DOS. A LOAD, WRITE, or V (directory) will change the default drive. RGB systems are limited to 254 as the highest accessable drive.

Write has a new option: save source code files without line numbers. This was done to complement the new Load option.

Examples:

```
WD TEST
```

saves source file currently in memory as TESTI ASM.

```
WD TEST/PR1
```

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

```
W# filename
```

save source without line numbers

```
W filename
```

normal format; saves source with line numbers

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.

It is now possible to load source code files that do not have line numbers. This function, transparent to the user, allows source files from other editor/assemblers, such as Macro-80C, to be loaded.

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 11/).

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 EDT ASM 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 a Program (ASM Commands)

To load the assembler program and assemble the source program into 6809 machine code, EDTASM 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 /switch1/switch2/...
```

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.

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).

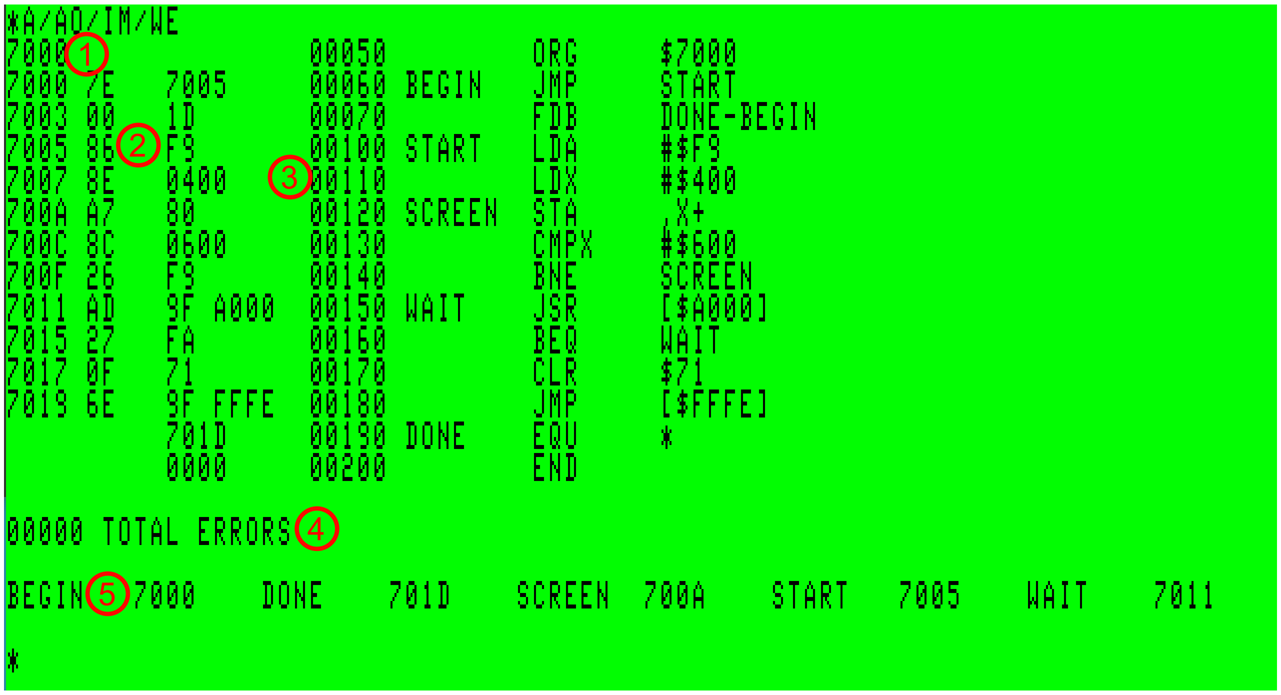
```
A TEST /LP
```

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.

Assembly Listing



```
*A/AO/IM/WE
7000                00050         ORG     $7000
7000 7E   7005      00060 BEGIN   JMP     START
7003 00   1D        00070         FDB     DONE-BEGIN
7005 86   F9        00100 START   LDA     #$F9
7007 8E   0400      00110         LDX     #$400
700A A7   80        00120 SCREEN  STA     ,X+
700C 8C   0600      00130         CMPX    #$600
700F 26   F9        00140         BNE     SCREEN
7011 AD   9F A000   00150 WAIT    JSR     [$A000]
7015 27   FA        00160         BEQ     WAIT
7017 0F   71        00170         CLR     $71
7019 6E   9F FFFE   00180         JMP     [$FFFE]
          701D      00190 DONE    EQU     *
          0000      00200         END

00000 TOTAL ERRORS

BEGIN  7000    DONE    701D    SCREEN  700A    START   7005    WAIT    7011

*
```

Figure 1. Assembly Display Listing

1. The location in memory where the assembled code will be stored. In this example, the origin for the code will be $7000
2. The assembled code for the program line. $86F9 is the assembled code for LDA #$F9.
3. The Column dedicated to showing the program line numbers.
4. The number or errors produced while being assembled. Use the /WE flag to stop assembly on errors.
5. The symbols you used in your program and the memory locations that they refer to are list here.

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 "sting
/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
```

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 ILP 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 program, the edit buffer, and the symbol table.

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:

```
0050 ORG $6000
```

Then, at the * prompt, type:

```
A /IM
```

The START address is now the first available address.

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
```

Your program will now start at the address specified in the ORG statement.

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.

MEMORY OVERVIEW

Values for the end of EDTASM and "TOP OF RAM" have been changed from DISK EDTASM. You can determine the end of EDTASM by using the `O`command as explained on the next pages. Top of RAM is now $FDFF for both 64K Coco 1 & 2 or the Coco 3.

Initially, the RAM from $8000 through $FDFF will be filled with a copy of the system ROMs. If you wish to examine this code with ZBUG, do so before loading or creating source code that over writes this area of RAM.

NOTE: From the diagrams the way the EDIT BUFFER, MACRO TABLE, and SYMBOL TABLE all use the same RAM space. Very large source code files can fill RAM to the point where there is no longer any room for the macro and symbol tables. Large programs require use of the following precautions:

- limit program segments to blocks of 1000 lines
- have one short master source code file which calls all of the segments with the INCLUDE pseudo opcode

Example:

```
1 * Large Program Mainline
2   INCLUDE PART1:0
3   INCLUDE PART2:0
4   INCLUDE PART3:3
5   INCLUDE PART4:1
            etc.

10  END
```

none of the PART segments should have an END

If necessary, shorten the length of the segments and increase the number of PARTS to accommodate very large source files. This method allocates most of RAM to the symbol table reserving a small edit space for each PART as it temporarily loads from disk.

It is possible using this method to assemble to disk source files which are larger than a full disk worth of code.

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
```

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

```
D50
```

Then assemble the program in memory by typing:

```
A /IM
```

Since this sample program uses START to label the beginning of the program, you can find its originating address from the assembler listing. You will see the default starting address location in memory.

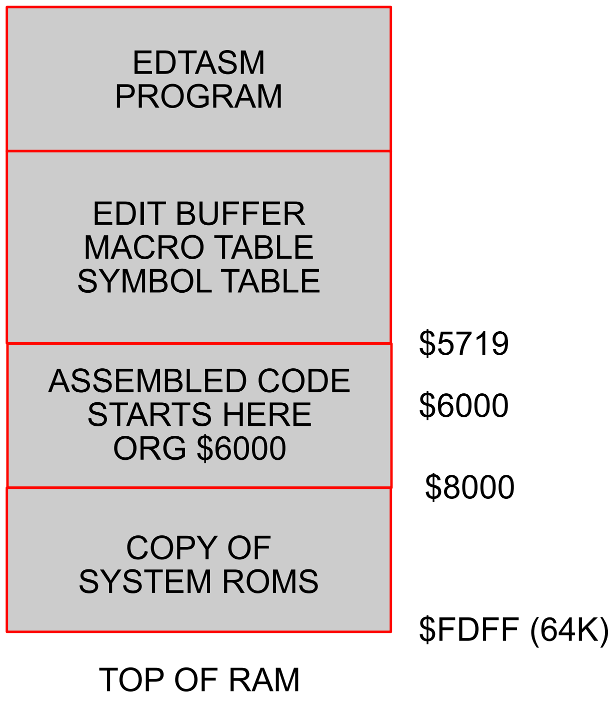

EDTASM PROGRAM
EDIT BUFFER MACRO TABLE SYMBOL TABLE
ASSEMBLED CODE STARTS HERE ORG $6000
COPY OF SYSTEM ROMS
$5719
$6000
$8000
$FDFF (64K)
TOP OF RAM

FIGURE 2. /AO In-Memory Assembly

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
```

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.

```
USERORG=6050
```

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

```
A /IM /MO
```

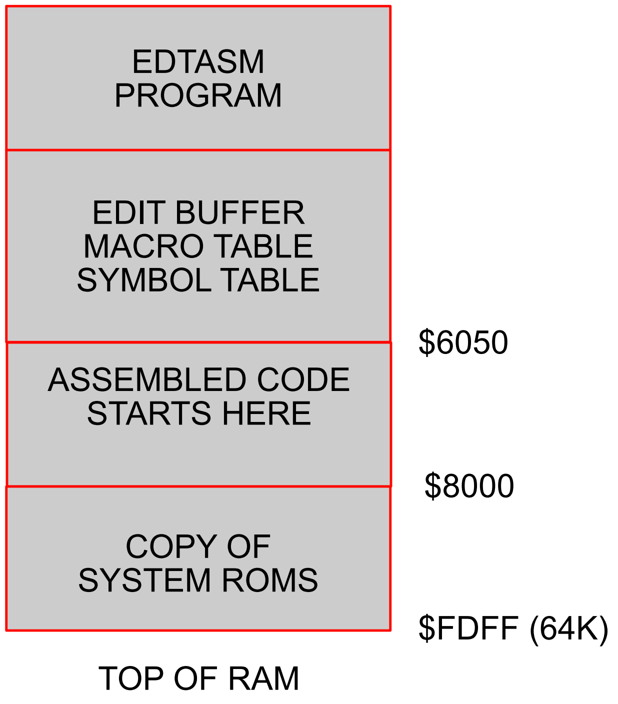


FIGURE 2. /MO In-Memory Assembly

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 $6000 instruction:

```
50 ORG $1200
```

and assemble it to disk by typing:

```
A SAMPLE /SR
```

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.

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 a Program (ZBUG Commands)

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 EDT ASM's ZBUG program or the stand-alone 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
```

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
```

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

```
Z
```

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

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
```

and examine the memory addresses that contain your program: $5800-$5817. 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, Address $580F shows the instruction BNE 580A rather than BNE SCREEN.

Symbolic Mode

From the command level, type:

```
S
```

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
```

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 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:

Type either of the following:

```
GBEGIN
G5800
```

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 Break Points

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 break point at the first address that contains the symbol SCREEN: $580A You can type either of the following:

```
XSCREEN
X580A
```

Now type:

```
GBEGIN
```

to execute the program. Each time execution breaks, type:

```
C
```

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
```

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

```
C10
```

and the tenth time ZBUG encounters that break point, it halts execution. Type:

```
Y
```

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

```
Y0
```

This deletes the first break point (Break point 0). You may not set a break point in a ROM routine. If you set a break point at the point where you are calling a ROM routine, the C command will not let you continue.

Examining Registers and Flags

Type:

```
R
```

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 con□tents of memory. Type:

```
0
```

and the Register X now contains a zero.

Stepping Through the Program

Type: *(Note the comma!)*

```
BEGIN,
```

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

```
,
```

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

```
R
```

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 [$AOOO] 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

Type:

```
U 5800 5000 6
```

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:

```
PS TEST/BUG 5800 5817 5800
```

This saves your program on disk, beginning at Address 5800 and ending at Address 5817.

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
```

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 ZBUG Calculator (ZBUG Commands)

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.

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:

```
D10
```

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

```
D8
```

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

```
D16
```

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:

```
I10
```

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

```
T 48152 48182
```

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 VALUE	AFTER VALUE
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 48152 $C010
```

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

```
T 48152T C010
```

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 COOO (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 cal□culation you wan~ 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 010 mode and try the above examples again.

Operators

You may use arithmetic, relational, or logical operators. (Get into the 016 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=
```

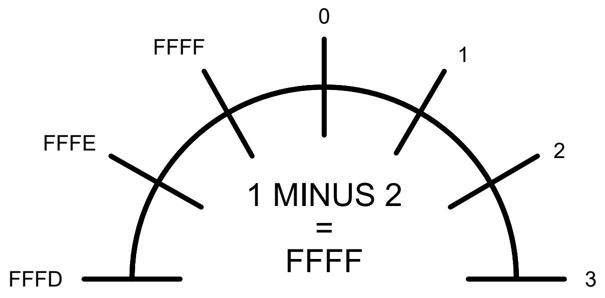
prints 4. (ZBUG can divide integers only.)

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

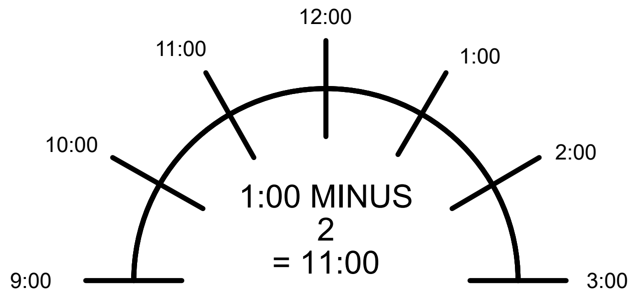
prints 1, the remainder of 9 divided by 2.

```
1-2=
```

prints 0FFFF,65535T, or 1777770, 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):



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



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 1777770 in the O8 mode.)

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

prints 0, since the relationship is false.

Logical Operators

Shift	<
Logical AND	.AND.
Inclusive OR	.OR.
Exclusive OR	.XOR.
Complement	.NOT.

Logical operators perform bit manipulation on binary numbers. To understand bit manipulation, see the6809 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:

1. +
2. .DIV.
3. .MOD.
4. <
5. .AND.
6. .OR.
7. .XOR.
8. +
9. .EQU.
10. .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.

SECTION III

ASSEMBLY LANGUAGE

SECTION III

ASSEMBLY LANGUAGE

This section gives details on the Disk EDT ASM assembly language. It does not explain the 6809 mnemonics, however, since there are many books available on the 6809. To learn about 6809 mnemonics, read one of the books listed in "About This Manual." If you need more technical information on the 6809, read:

MC6809-MC6809E
8-Bit Microprocessor Programming Manual
Motorola, Inc.

Chapter 10/ Writing the Program

The Hitachi 6309 Registers

If you have an Hitachi 6309 CPU resident in your Coco, then you have extra registers present: E,F one byte; W (E+F), V two bytes, MD one byte.

If the 6309 is resident then the new registers E,F, and V will be recognized by ZBUG and the contents can be displayed and altered directly. All new registers can be accessed from your own ml. program.

You can not single step through code which has set reg.MD for native 6309 mode. If you need to test this mode, you must enclose the questionable code with: ORCC #$50; LDMD #1; code section; LDMD #0; ANDCC #$AF.

Test the code section with the G command, from ORCC to ANDCC. Under these conditions, ZBUG can be used with native 6309 mode.

The 6809 Registers

The 6809 contains nine temporary storage areas that you may use in your program:

REGISTER	SIZE	DESCRIPTION
A	1 BYTE	ACCUMULATOR
B	1 BYTE	ACCUMULATOR
D	2 BYTES	ACCUMULATOR
		(A combination
		of A and B)
DP	1 BYTE	DIRECT PAGE
CC	1 BYTE	CONDITION CODE
PC	2 BYTES	PROGRAM COUNTER
X	2 BYTES	INDEX
Y	2 BYTES	INDEX
U	2 BYTES	STACK POINTER
S	2 BYTES	STACK POINTER

Table 2. 6809 Registers

Registers A and B can manipulate data and perform arithmetic calculations. They each hold one byte of data. If you like, you can address them as D, a single 2-byte register.

Register DP is for direct addressing. It stores the most significant byte of an address. This lets the processor directly access an address with the single, least significant byte.

Registers X and V can each hold two bytes of data. They are mainly for indexed addressing. Register PC stores the address of the next instruction to be executed.

Registers U and S each hold a 2-byte address that points to an entire "stack" of memory. This address is the top of the stack + 1. For example, if Register U contains 0155, the stack begins with Address 154 and continues downwards.

The processor automatically points Register S to a stack of memory during subroutine calls and interrupts. Register U is solely for your own use. You can access either stack with the PSH and PUL mnemonics or with indexed addressing.

Register CC is for testing conditions and setting interrupts. It consists of eight "flags." Many mnemonics "set" or "clear" one or more of these flags. Others test to see if a certain flag is set or clear.

C (Carry), Bit 0 - an 8-bit arithmetic operation caused a carry or borrow from the most significant bit.

V (Overflow), Bit 1 - an arithmetic operation caused a signed overflow.

Z (Zero), Bit 2 - the result of the previous operation is zero.

N (Negative), Bit 3 - the result of the previous operation is a negative number.

I (Interrupt Request Mask), Bit 4 - any requests for interrupts are disabled.

H (Half Carry), Bit 5 - an 8-bit addition operation caused a carry from Bit 3.

F (Fast Interrupt Request Mask), Bit 6 - any requests for fast interrupts are disabled.

E (Entire Flag), Bit 7 - all the registers were stacked during the last interrupt stacking operation. (If not set, only Registers PC and CC were stacked)

Assembly Language Fields

You may use four fields in an assembly language instruction: label, command, operand, comment. In this instruction:

```
START LDA #$F9 GETS CHAR
```

START is the label. LDA is the command. #$F9 + 1 is the operand. GETS CHAR is the comment. The comment is solely for your convenience. The assembler ignores it.

The Label

You can use a symbol in the label field to define a memory address or data. The above instruction uses START to define its memory address. Once the address is defined, you can use START as an operand in other instructions. For example:

```
BNE START
```

branches to the memory address defined by START. The assembler stores all the symbols, with the addresses or data they define, in a "symbol table," rather than as part of the "executable program." The symbol can be up to six characters.

The Command

The command can be either a pseudo op or a mnemonic. Pseudo ops are commands to the assembler. The assembler does not translate them into opcodes and does not store them with the executable program. For example:

```
NAME        EQU $43
```

defines the symbol NAME as $43. The assembler stores this in its symbol table.

```
            ORG $3000
```

tells the assembler to begin the executable program at Address $3000.

```
SYMBOL      FCB $6
```

stores 6 in the current memory address and labels this address SYMBOL. The assembler stores this information in its symbol table. Mnemonics are commands to the processor. The assembler translates them into opcodes and stores them with the executable program. For example:

```
            CLRA
```

tells the processor to clear Register A. The assembler *assembles this into opcode number $4F and stores it with the executable program.*

The next chapter shows how to use pseudo ops. Reference L lists the 6809 mnemonics.

The Operand

The operand is either a memory address or data. For example:

```
            LDD #3000+COUNT
```

loads Register D with $3000 plus the value of COUNT. The operand, #$3000 + COUNT, specifies a data constant.

The assembler stores the operand with its opcode. Both are stored with the executable program.

Operators

The plus sign (+) in the above operand (#3000 + COUNT) is called an operator.

You can use any of the operators described in Chapter g, "Using the ZBUG Calculator," as part of the operand.

Addressing Modes

The above example uses the # sign to tell the assem□bler and the processor that $3000 is data. When you omit the # sign, they interpret $3000 in a different "addressing mode."

Example:

```
            LDD $3000
```

tells the assembler and processor that $3000 is an address. The processor loads D with the data contained in Address $3000 and $3001.

Each of the 6809 mnemonics lets you use one to six addressing modes. These addressing modes tell you:

- *If the processor requires an operand to execute the opcode*
- *How the assembler and processor will interpret the operand*

1. Inherent Addressing

There is no operand, since the instruction doesn't require one. For example:

```
     SWI
```

interrupts software. No operand is required.

```
     CLRA
```

clears Register A. Again, no operand is required. Register A is part of the instruction.

2. Immediate Addressing

The operand is data. You must use the # sign to specify this mode. For example:

```
     ADDA #$30
```

adds the value $30 to the contents of Register A.

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

loads the value $8004 into Register X.

```
           CMPX #$1234
```

compares the contents of Register X with the value 1234.

3. Extended Addressing

The operand is an address. This is the default mode of all operands.

(Exception: If the first byte of the operand is identical to the direct page, which is 00 on startup, it is directly addressed. This is an automatic function of the assembler and the processor. You need not be concerned with it if you're a beginner.)

For example:

```
          JSR  #$1234
```

jumps to Address $1234.

```
     SPOT EQU  #$1234
          STA  SPOT
```

stores the contents of Register A in Address $1234. If the instruction calls for data, the operand contains the address where the data is stored.

```
          LDA  $1234
```

does not load Register A with $1234. The processor loads A with whatever data is in Address $1234. If $06 is stored in Address $1234, Register A is loaded with $06.

```
          ADDA  $1234
```
adds whatever data is stored in Address $1234 to the contents of Register A.
```
          LDD   $1234
```
loads D, a 2-byte register, with the data stored in memory addresses $1234 and $1235. You can use the > ssymbol, which is the sign for extended addressing, to force this mode. (See "Direct Addressing.")

Extended Indirect Addressing.

The operand is the address of an address. This is a variation of the extended addressing mode. The [ ] symbols specify it. (Use (SHIFT DOWN ARROW) to produce the [symbol and [SHIFT →] to produce the] symbol.)

In understanding this mode, think of a treasure hunt game. The first instruction is "Look in the clock." The clock contains the second instruction, "Look in the refrigerator. "

Examples:
```
          JSR   [$1234]
```
jumps to the address contained in Addresses $1234 and $1235. If $1234 contains $06 and $1235 contains $11, the effective address is $0611. The program jumps to $0611.
```
     SPOT EQU   $1234
          STA   [SPOT]
```
stores the contents of Register A in the address contained in Addresses $1234 and $1235.
```
          LDD   [$1234]
```
loads D with the data stored in the address that is stored in Addresses $1234 and $1235.

This is a good mode of addressing to use when calling ROM. routines. For example, the entry address of the POLCAT routine is contained in Address $A000. Therefore, you can call it with these instructions:
```
   POLCAT EQU   $A000
          JSR   [POLCAT]
```
If a new version of ROM puts the entry point in a different address, your program still works without changes.

4. Indexed Addressing

The operand is an index register which points to an address. The index register can be any of the 2-byte registers, including PC. You can augment it with:

• A constant or register offset
• An auto-increment or auto-decrement of 1 or 2

The comma (,) indicates indexed addressing.

As an example, load X, a 2-byte register, with $1234:
```
          LDX   #$1234
```
You can now access Address $1234 through indexed addressing. This instruction:
```
          STA   ,X
```
stores the contents of A in Address $1234
```
          STA   3,X
```
stores the contents of A in Address $1237, which is $1234 + 3. (The number 3 is a constant offset.)
```
   SYMBOL EQU   $4
          STA   SYMBOL,X
```
stores the contents of A in Address $1238, which is $1234 + SYMBOL. (SYMBOL is a constant offset.)
```
          LDB   #$5
          STA   B,X
```
stores the contents of A in Address $1239 which is $1234 + the contents of B. (B is a register offset. You can use either of the accumulator registers as a register offset.)
```
          STA   ,X+
```
This instruction does two tasks: (1) stores A's contents in Address $1234 (the contents of X) and then. (2) increments X's contents by one, so that X contains $1235.
```
          STA   ,X++
```
(1) stores A's contents in Address $1235 (the current contents of X) and then (2) increments X's contents by two to equal $1237.
```
          STA   ,--X
```
(1) decrements the current contents of X by two to equal $1235 ($1237 - 2) and then (2) stores A's contents in Address $1235.

As we said above, you can use PC as an index register.

In this form of addressing, called program counter relative, the offset is interpreted differently. For example:
```
   SYMBOL   FCB   0
            LDA   SYMBOL,PCR
```
While assembling the program, the assembler subtracts the contents of Register PC from the offset:
```
            LDA   SYMBOL-PC,PCR
```
While running the program, the processor adds the contents of Register PC to the offset. This causes A to be loaded with SYMBOL.

This seems to be the same as extended addressing. But, by using program counter relative addressing, you can relocate the program without having to reassemble it.

Indexed Indirect Addressing.

The operand is an index register which points to the address of an address. This is a variation of indexed addressing.

For example, assume that:

• Register X contains $1234
• Address $1234 contains $11
• Address $1235 contains $23
• Address $1123 contains $64

This instruction:
```
            LDA   [ ,X]
```
loads A with 64. (Register X pOints to the addresses of the address. This address is storing 6, the required data.)
```
            STA   [ ,X]
```
stores the contents of A in Address $1123. (Register X points to the addresses, $1234 and $1235, of the effective address, $1123.)

5. Relative Addressing

The assembler interprets the operand as a relative address. There is no sign to indicate this mode. The assembler automatically uses it for all branching instructions.

For example, if this instruction is located at Address $0580:

```
        BRA     $0585
```

The assembler converts $0585 to a relative branch of + 3 (0585-0582).

This mode is invisible to you unless you get a BYTE OVERFLOW error, which we discuss below. Because the processor uses this mode, you can relocate your program in memory without changing any of the branching instructions.

The BYTE OVERFLOW error means that the relative branch is outside the range of -128 to + 127. You must use a long branching instruction instead. For example:

```
        LBRA    $0600
```

allows a relative branching range of -32768 to + 32767.

6. Direct Addressing

In this mode, the operand is half of an address. The other half of the address is in Register DP:

ADDRESS	=	DP REGISTER Most significant bit	OPERAND Least significant bit

Figure 7. Direct Addressing

The assembler and the processor use this mode automatically whenever they approach an operand whose first byte is what they assume to be a "direct page" (the contents of Register DP). Until you change the direct page, the assembler and the processor assume it is 00. For example, both of these instructions:

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

cause a jump to Address $0015. In both cases, the assembler uses only 15 as the operand, not 00. When the processor executes them, it gets the 00 portion from Register DP and combines it with $15. (On startup, DP contains 0, as do all the other registers.)

Because of direct addressing, all operands beginning with 00, the direct page, consume less room in memory and run quicker. If most of your operands begin with $12, you might want to make $12 the direct page.

To do this, you first need to tell the assembler what you are doing, by putting a SETDP pseudo-operation in your program:

```
      SETDP $12
```

This tells the assembler to drop the $12 from all operands that begin with $12. That is, the assembler assembles the operand "1234" as simply "34".

Then, you must load Register DP with $12. Since you can use LD only with the accumulator registers, you have to load DP in a round-about manner:

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

Now the direct page is $12, rather than 00. The processor executes all operands that begin with $12 (rather than 00) in an efficient, direct manner.

The assembler uses direct addressing on all operands whose first byte is the same as the direct page. You can denote direct addressing with the < sign if you want to document or be sure that direct addressing is being used.

For example, if the direct page is $12:

```
      JSR  <$15
```

jumps to Address $1215. This instruction documents that the processor uses direct addressing.

Similarly, you might want to use the> sign to force extended addressing. For example:

```
      JSR  >$1215
```

jumps to Address $1215. The assembler and processor use both bytes of the operand.

To learn more about 6809 addressing modes, read one of the books listed at the beginning of this manual.

Chapter 11/ Using Pseudo Ops