MEMOTRON
PRESENTS

E - Z

MACRO
ASSEMBLER

Table of Contents

Memotron

E - Z

Macro - Assembler

C-64 AND C-128 VERSIONS

Available commands and features :

SOU ASSEMBLER Commands :

Name	Purpose	Format
1). Append	Ties 2 or more programs together	Append Filename
2). Assemble	Start or continue assembling	Assemble
3). Clear	Clears all memory off	Clear
4). Commands	A call to view all commands available	Commands
5). Decimal of	Returns Decimal of 4 digit Hex Number	Decimal of $XXXX
6). Disk	Allows you to send commands to Disk (Scratch, Read Error Channel)	Disk (>prompts that you are inside the disk mode)
7). DLoad	Loads files previously DSaved on Disk	DLoad Filename
8). DSave	Save files to disk	DSave Filename
9). Dump	Prints to printer the code as input	Dump 20 30 Filename (prints line 20 to 30 and attach a filename)

10). Erase	Erases Lines of Code	Erase 2 (erase line 2) Erase 6 20 (erase from line 6 to line 20)
11). Exit	Exit back to Basic	Exit
12). Hex of	Returns Hex of decimal number	Hex of #XXXXX
13). Kernal	Display all Kernal call outs and corresponding addresses	Kernal
14). List	List Lines of Code	List 10 30 (list from line 10 to line 30) List 10 (list line 10) List (list all lines) List - 10 (list from line 1 to line 10) List 10 - (list from line 10 to last line)
15). Menu on disk	Displays files on Disk	Menu on Disk
16). Object	Loads 2nd part of Macro Assembler	Object
17). Pseudo Ops	Display all pseudo-ops	Pseudo Ops
18). Revise	Allows update or modification of code lines	Revise 20 (update line 20)
19). Symbolic	Loads & Starts the Symbolic Disassembler	Symbolic

20). Wedge	Places Code Between already written code	Wedge 20 10 (wedge 10 lines starting at line 20)

Pseudo Ops :

Name	Example	Comment
1). ASC	ASC "Message"	Places message on ASCII form on memory
2). BYT	BYT 3,20,$AA	Places 3,20,$AA on memory
3). DST	Buffer DST 100	It will create a 100 byte Table called Buffer
4). EQU	Vector EQU $A000	It will give the label "Vector" a value of $A000
5). ORG	(Space)ORG $C000	It tells assembler where the assembly is going to start from. (code start address)

Extras It Supports :

1). Addition = Numerical values can be added to code.

2). (#>) Gets Hi Byte of Address

3). (#<) Gets Lo Byte of Address

4). (#'A) Gets ASCII Number of "A" or any character

5). (;) and (*) are used for comments

6). Supports either Hex or Decimal Numerical inputs

Plus :

E - Z SYMBOLIC DISASSEMBLER is also included

Features :

1). Creates real Source Files from Object Files

2). Generates Equates for internal & external addresses

3). Disassembles any 6502-6510-8502 machine code program into clear, easy to read source code

4). 5 pass Symbolic Disassembler with automatic label generation

5). Outputs Source Code Files to disk, which are fully compatible with the E - Z SOU.ASSEMBLER module

6). Outputs formatted listings to screen or printer

7). Disassembles programs regardless of load address, in other words, it will handle "AUTOBOOT" programs

8). Recognizes instructions hidden under BIT instructions

9). Helps adapt existing programs to your specific needs

10). Gives you the option to Start and End disassembling anywhere within the program to be disassembled

Error Generation on SOU ASSEMBLER :

1). Command Not Available

2). Danger All Lines Have Been Used

3). File Too Large

4). Wrong Parameters

5). Disk I/O Error (Supports All C-64 Disk Errors)

6). File To Be Appended Is Too Large

Error Generation on OBJ ASSEMBLER :

1). Illegal Opcode

2). Illegal Addressing Mode

3). Label Not Found

4). Too Long Conditional Branch

5). No Data On Pseudo Op

6). No Delimiter On ASC Pseudo Op

7). Illegal Table Length

8). Value Out Of Range (65535 LIMIT)

9). Selected File Too Large

10). EQU Pseudo Op Error In Line # XXXX

11). Error On Source Assembler Line # XXXX

12). Please Scratch = (FILENAME) File From Disk

13). Linking File Is Too Large

14). Disk I/O Error (Supports All C-64 Disk Errors)

Error Generation on SYMBOLIC DISASSEMBLER :

1). Try again, Name for Object File is too long

2). Try again, Name for Source File is too long

3). Error - Out of Range Address

4). Disk I/O Error (Supports All C-64 Disk Errors)

SUMMARY OF FEATURES

1). LOW COST / HIGH-PRICED FEATURES

2). A TRUE THREE PASS ASSEMBLER

3). FULL SCREEN EDITING OF SOURCE PROGRAM

4). SOURCE FILE CHAINING (APPENDING) CAPABILITIES

5). OBJECT FILE CHAINING (LINKING) CAPABILITIES

6). SUPPORTS SPECIAL PSEUDO OPS

7). OUTPUTS LABEL REFERENCE TABLES (BY ADDRESS)

8). A TRUE SYMBOLIC AND LABEL ASSEMBLER

9). FULLY MENU AND PROMPT DRIVEN

10). FAST AND EFFICIENT MACRO ASSEMBLER CAPABILITIES

11). ADVANCED 5 PASS SYMBOLIC DISASSEMBLER IS INCLUDED

INTRODUCTION

What BASIC is to BASIC programming, an ASSEMBLER is to ML programming. The E-Z assembler is a complete language. You write programs (source code) which the E-Z assembler translates into the finished, executable ML (object code). Unlike less advanced assemblers, however, symbolic assemblers such as the E-Z assembler can be as easy to use as higher level languages like BASIC. The source code is very simple to modify. Variables and subroutines have names. The program can be internally commented with REM-like explanations.

This text will not teach you everything there is to know about assembly language programming. It's purpose is to give you some of the vocabulary and general ideas which will help you on your way.

I'm certain that everyone has been introduced to the idea of a bit. A bit has two states: on and off, typically represented with the symbols "1" and "0". In this context, DON'T think of 1 and 0 as numbers. They are merely convenient shorthand labels for the state of a bit.

The memory of your computer consists of a huge collection of bits, each of which could be in either the 1 or 0 (on or off) state.

At the heart of your computer is a microprocessor chip, named the 6510 by MOS, who makes the chip. What this chip can do is manipulate the bits which make up the memory. The 6510 likes to handle bits in chunks, and so we"ll introduce a special name for the size of bit chunks the 6510 is most happy with. A byte will refer to a collection of eight bits.

A collection of bits holds a pattern, determined by the state of it's individual bits.

If you've had a course in probability, it's quite easy to work out that there are 256 possible patterns that a byte could hold.

Without getting too far ahead of myself, I'll just casually mention that there are about 56 fundamental operations that the 6510 microprocessor chip can carry out.

The point of this discussion is that we can use bit patterns to represent anything we want, and by manipulating the patterns in different ways, we can produce results which have significance in terms of what we're choosing to represent.

As stated before, the 6510 chip inside your computer can manipulate the bit patterns which make up the computer's memory. Some of the possible manipulations are copying patterns from one place to another, turning on or turning off certain bits, or interpreting the patterns as numbers and performing arithmetic operations on them. To perform any of these actions, the 6510 has to know what part of memory is to be worked on. A specific location in memory is identified by it's address.

An address is a pointer into memory. Each address points to the beginning of a byte long chunk of memory. The 6510 has the capability to distinguish 65535 different bytes of memory.

The contents of memory may be broken down into two broad classes. The first is data, just raw patterns of bits for the 6510 to work on. The significance of the patterns is determined by what the computer is being used for at any given time.

The second class of memory contents are instructions. The 6510 can look at memory and interpret a pattern it sees there as specifying one of the 50 some fundamental operations it knows how to do. This mapping of patterns onto operations is called the machine language of the 6510. A machine language program consists of a series of patterns located in consecutive memory locations, whose corresponding operations perform some useful process.

Note that there is no way for the 6510 to know whether a given pattern is meant to be an instruction, or a piece of data to operate on. It is quite possible for the chip to accidentally begin reading what was intended to be data, and interpret it as a program. Some pretty bizarre things can occur when this happens. In assembly language programming circles, this is known as "crashing or locking up the system".

WHY AN ASSEMBLER ?

Unless you happen to be a 6510 chip, the patterns which make up a machine language can be pretty incomprehensible. For example, the pattern that tells the 6510 to load the Accumulator with a value of zero is :

A9 00

Which is not very informative. On the other hand, Assembly Language represents each of the many operations that the computer can do with a **MNEMONIC**, a short, easy to remember series of letters. (3 letters) Using the prior example Assembly Language will look like this :

LDA #$00

Which is a lot English like and easy to understand. Therefore, what is needed is a special program to run on the 6510 which converts the string "LDA #$00" into the pattern "A9 00". This program is called an assembler. A good analogy is that an assembler program is like a meat grinder which takes in assembly language and gives out machine language.

Typically, an Advanced assembler reads a file of assembly language and translates it one line at a time, outputting a file of machine language. Often times, the input file is called the Source file and the output file is called the Object file. The machine language patterns produced are called the Object code. The E-Z assembler is such an assembler.

The source code that you build and save in the Sou Assembler module will have a ".SC" tag attached to it's filename. SC stands for Source Code.

The object code that you build and save in the Obj Assembler module will have a ".OC" tag attached to it's filename. OC stands for Object Code.

Also produced during the assembly process is a listing, which summarizes the results of the assembly process. The listing shows each line from the source file. In the event that the assembler was unable to understand any of the source lines, it inserts error messages in the listing, pointing out the problem.

The last part of an assembly language line is a **comment**. The comment is totally ignored by the assembler, but is vital for humans who are attempting to understand the program. Assembly language programs tend to be very hard to follow, and so it's particularly important to put in lots of comments so that you'll remember just what it was you were trying to do with a given piece of code. Professional assembly language programmers put a comment on every line of code, explaining what it does, plus devoting many entire lines for additional explanations.

Since the assembler ignores the comments, they cost you nothing in terms of size or speed of execution in the resulting machine language program. This is in sharp contrast to BASIC, where each remark slows your program down and eats up precious memory.

Generally, a character is set aside to indicate to the assembler the beginning of a comment, so that it knows to skip over. This assembler follows a common convention of reserving the semi-colon (;) and also the asterisk (*) for marking comments.

The E-Z assembler recognizes a series of pseudo-operations which are handled as embedded commands to the assembler itself, not as an instruction in the machine language program being built. Almost invariably, you'll see the phrase pseudo-operation abbreviated down to pseudo-op. Sometimes you'll see assembler directive, which means the same thing, but just doesn't seem to roll off the tongue as well as pseudo-op.

One very common pseudo-op recognized by the E-Z assembler is the Equate, usually given mnemonic EQU. What this allows you to do is assign a name to a frequently used constant. Thereafter, anywhere you use that name, the assembler automatically substitutes the equated constant. This process makes your program easier to read, since in place of the somewhat meaningless looking pattern, you see a name which tells you what the pattern is for. It also makes your program easier to modify, since if you decide to change the constant, you only need to do it once, rather than all over the program.

Examples of an Equate would be :

1. START EQU $FFFF

2. VECTOR EQU $0314

3. IRQ EQU 378

Another pseudo-op supported by the E-Z assembler is the ASC pseudo-op. This pseudo-op is a very handy utility which saves you all the trouble of translating by hand each character in a message that is desired to be stored in memory for a later use such as in a prompt or a message.
Examples of an ASC conversion will be :

1. PROMPT ASC 'THIS IS THE END'

2. MESSAGE ASC .THIS IS THE START.

3. ALARM ASC [DANGER 'NOT ENOUGH MEMORY']

Notice that in the first example we used the ' ' HYPHEN as delimitors (A delimitor is a mark to define the start and the end of an ASC conversion). In other words, the E-Z assembler will take the very first character that it finds and use it as a start delimitor, then it will look for a similar character to use it as the ending delimitor, so anything in between will be considered part of the ASC conversion, this is useful since this way you can use the ' ' HYPHEN inside a message as shown in example 3 of the preceding examples. The only limitation on this pseudo-op, is that you can not use comments on the same line that a ASC conversion has been performed.

The third pseudo-op supported by the E-Z assembler is the ORG pseudo-op, which is used to indicate the desired starting address on the machine language.
Examples of ORG usage would be :

1. (space) ORG $C000

2. (space) ORG 8192

3. (space) ORG 40960

When entering the ORG pseudo-op, always leave an space between it and the beginning of that line, and always place ORG as the first line of the source code.

The fourth pseudo-op is called BYT and, it is used to enter a list of numbers in a consecutive manner, such as vectors, pointers, etc.
Examples of BYT usage would be :

```
1. VECTOR BYT $14,$03,$14,$03

2. POINT BYT 99,00,76,55

3. LIST BYT 01,02,03,04,05,06,08,09
```

Notice that you can enter either hex or decimal numbers, also that each number is separated by a comma, but be careful because no commas are allowed at the end of a line, and also like in the case of the ASC pseudo-op no comments are allowed in a line that has had a BYT pseudo-op.

The last pseudo-op supported by the E-Z assembler is the DST pseudo-op (DST = declare stable table). This pseudo-op allows us to declare or reserve an area of memory that can be used for storage area, for tables, or simply to hold vectors and pointers.
Examples of DST usage would be :

```
1. ENDPTR DST 2

2. STRPTR DST 2

3. SPEECH DST 2000
```

Notice how we are using the DST pseudo-op to open up tables where we can store either pointers (as in # 1 & 2 example) or to hold raw data (as in # 3 example).

The E-Z assembler also supports mini utilities such as :

1). Addition

```
    VECTOR EQU $0314

    ADD LDA VECTOR+1
```

2). (#>) gets hi-byte of label

```
    2000 START LDA VECTOR

  2003 NEW LDA #>START ;get the hi-byte address of the label
called START in this case 20 HEX.
```

3). (#<) gets lo-byte of label

2000 START LDA VECTOR

2003 NEW1 LDA #<START ;get the lo-byte address of the label called START in this case 00 HEX.

4). (#'Z) gets ASCII number of "Z" or any character

2000 LOAD LDA #'A

2002 STORE STA SECOND ;first load the accumulator with the numerical value of the letter "A" then store in the label called second.

5). The asterisk (*) and the semi-colon (;) are use to indicate that a comment is to follow.

2000 DEMO STA $80 ;THIS IS A COMMENT (USE ONLY SEMI-COLON IN THIS TYPE OF COMMENT)

;THIS IS A COMMENT THAT USES THE WHOLE LINE

THIS IS ANOTHER COMMENT THAT USES THE WHOLE LINE

LABELS

Probably the most powerful feature of the E-Z assembler is it's capability of using labels. Labels are words or strings of characters that refer to a certain value, memory location or to a certain part of a program. For example a value of 32 can be assigned to a word called "SPACE" and then, from now on we just refer to it as "SPACE" instead of having to remenber that number 32 is equal to a "SPACE".
In practice , by using labels throughout your programs, you have the capability of assembling programs that are fully relocatable, and can operate in different memory locations.

Also included in the E - Z ASSEMBLER PACKAGE is the E - Z SYMBOLIC DISASSEMBLER.

Why a Symbolic Disassembler ?

The more you get involved in Assembly Language Programming, the more likely that you will acquire machine language object files, for which you don't have any information for, but that you would like to analyze, understand or modify. Or perhaps you would like to relocate the program or investigate a certain programmers' technique. In order to do this, you'll require a program called Disassembler. Now, there are two kinds of disassemblers, Plain Disassemblers & Symbolic Disassemblers.

Plain Disassemblers are utility programs that will scan a given machine code program and will typically display on Screen or on your printer a corresponding disassembled machine language listing. Some disassemblers will go as far as allowing you to modify the assembly as it is being displayed, but they are generally awkward to work with, and very hard to follow if the program being examined is of any reasonable length.

Symbolic Disassemblers like the E-Z Disassembler scan a given machine code (object file) program and generate a corresponding Assembly language Source File that in return can be used by our E-Z SOU ASSEMBLER Module.

Also during Disassembly, our disassembler generates labels to denote addresses (Locations) and values (expressions). These labels are attached to instructions or expressions to denote memory locations, and then all jumps & branches within the code are created by referencing to these labels. All these labels start with the (LB) characters. The main benefit of using labels is that they make the code automatically relocatable since all memory locations are referenced as relative and defined by a label. Also, branching no longer involves complex hexadecimal calculations.

The following pages give you a more detailed explanation of all the features available in the E-Z ASSEMBLER, so read on.

Getting Started

The write protect tab should be removed from disk for the following procedures.

1). Load the program with a Load "MT",8,1 A welcome screen will greet you, then you are asked to press return. (This will Auto Load & Auto Start the SOU ASSEMBLER module)

2). You will have to wait about 60 seconds while the program is being Loaded

3). You then will be presented with the copyright notice and in the left lower corner with a blinking cursor besides a "." PERIOD this indicates that you are inside the SOU ASSEMBLER module.

4). Next, type COMMANDS and press return.

5). This will display all 20 commands available to you while inside the Sou Assembler Module.

6). Now lets walk through some of them to familiarize you with them.

7). Type DECIMAL OF $2000 and hit return.

8). The display will show a decimal 8192 number. Therefore, we use the command DECIMAL OF $XXXX to translate from hexadecimal number to decimal numbers.

9). Next, type HEX OF #49152 and hit return.

10). The display will show a $C000 hexadecimal. Therefore, we use the HEX OF #XXXXX to translate from decimal into hex numbers.

11). Next, type KERNAL and hit return.

12). We then are presented with all possible Kernal calls and their respective addresses.

13). Type PSEUDO-OPS and hit return.

14). You will be presented with the 5 pseudo-ops available to you to make your Assembly language experience a bit easier.

15). Type MENU ON DISK and hit return.

16). The contents of the disk that you have present on your disk drive will be presented on your display without affecting your computer memory at all. (Note you can freeze this Menu listing by pressing the Run/Stop key, or terminate the listing by pressing the Space Bar key).

17). Type DISK and hit return.

18). A ">" prompt will be present indicating that you are inside the disk mode (to abort, just type"?" and hit return). Now you can send Disk Commands to your disk such as Scratch, Validate, New, etc. (Note: you do not need to use "" Quotes anymore). Also, you can read the Disk Drive Error channel by typing besides the > prompt the word ERROR and then hitting return.

19). Type Exit and hit return.

20). Before the program will let you exit, it will ask you whether or not you have DSaved your Source Code.

21). If you answer "Y", the program will let you back into Basic and instruct you on how to get back in into the Assembler.

22). If you answer "N", the program will terminate the Exit Command and will send you back into the SOU ASSEMBLER.

23). Type OBJECT and hit return.

24). Before the program will let you go into the Object Module (OBJ ASSEMBLER), it will ask you whether or not you have DSaved your Source Code.

25). If you answer "Y", the program will ask you to press return to confirm your action; and, it will automatically Auto Load and Auto Start the OBJ ASSEMBLER Module.

26). If you answer "N", the program will terminate the Object Command and will send you back into the SOU ASSEMBLER.

27). Bring out the Menu on your Disk, by typing MENU ON DISK and hitting return.

28). Type DLOAD TEST and hit return.

29). We just loaded the file called "Test.SC" into memory. (Notice the .SC tag attached to the Filename. This is just to keep all files separated. To DSave or DLoad, you don't need to attach this tag since the computer will do it for you).

30). Next, type **LIST** and hit return.

31). The Test file will be scrolling across your screen. You can slow down the Listing by pressing the Control key or freeze the Listing by pressing the Run/Stop key or just terminate the Listing by pressing the Space Bar. (See all format combinations available for the List Command by looking them over on the Macro Assembler Features section of this Manual).

32). Type **WEDGE 1 2** and hit return.

33). A One will be present with a solid cursor at it's right. Then, press "*" twice and hit return. Then, a two will be present. Press "*" twice and hit return. (? can also be used to abort).

34). Now, type **LIST** and hit return.

35). You will notice that Lines 1 and 2 have the asterisks that you just wedged in. this is how you use the Wedge Command to wedge or place code inside code that is already present (Note, you can wedge in any Line of the code).

36). Type **ERASE 1 2** and hit return.

37). Now, type **LIST** and hit return.

38). The listing will show you that we just erased Lines #1 and Lines #2 (Note: same syntax as with List can be used to erase Lines).

39). Type **REVISE 4** and hit return.

40). This will present you with a copy of what you already have on Line #4 but, you want to revise (? can also be used to abort). Type *** TESTING *** and hit return.

41). Now, **LIST** and you will see that you have revised Line #4 with the new message.

42). Type DUMP 1 10 TEST and hit return (Note: a printer should be connected before attempting this).

43). A Listing from Line 1 to Line 10 will be printed and the filename "Test" will be placed at the very top of the printed listing.

44). Type APPEND TEST1 and hit return.

45). Now, LIST and you will see that the Source Code of TEST1 has been added ("APPENDED") to the Source Code of TEST (Note: you can append in any combination or on any sequence as long as the total # of Lines appended together do not exceed 1000 Lines and all EQU pseudo-codes are on the first file).

46). Type DSAVE TEST0+1 and hit return.

47). You have just DSaved a Source file containing both Source codes from TEST and TEST1 (Note: a tag ".SC" is attached to the TEST0+1 filename on disk, but you don't need to add this tag).

48). Type CLEAR and press return.

49). Now LIST and you will find out that nothing lists since we just cleared all memory available.

50). The ASSEMBLE command will be discussed a bit later.

51). Now let me show you some samples of the pseudo-ops available on the assembler.

52). Type DLOAD SAMPLE1 and press return.

53). LIST and see how both ASC and BYT pseudo-ops need to be formatted.

54). Type DLOAD SAMPLE2 and press return.

55). LIST and see how the ORG, DST, and BYT pseudo-ops need to be formatted.

56). Type DLOAD SAMPLE3 and press return.

57). LIST and See how the EQU Pseudo-op needs to be formatted.

58). Type DLOAD SAMPLE4 and press return.

59). LIST and see samples of COMMENTS, #<, #>, +, and #', and their formats.

60). Before we go into the **ASSEMBLE** Mode, let me give you a clue or two on the internal operations of the Assembler.

The E - Z Macro Assembler consist of 3 modules :

Module 1). **SOU ASSEMBLER** .- Load"MT",8,1

This is the module that behaves like a Mini-Word processor, and allows you to input, edit, append, save, etc your Assembly Language Source Code. Also this module can transport you into the **OBJ ASSEMBLER** assembler module by calling the **OBJECT** mode and then answering the questions given by such a call-out.

Module 2). **OBJ ASSEMBLER** .- Load"AE",8,1

This module can be accessed by loading "AE",8,1 (This will Auto Load & Auto Start the **OBJ ASSEMBLER** module), or while in the **SOU ASSEMBLER** module by calling on the **OBJECT** mode and then answering the questions given by the computer.
This module converts the Assembly Language Source Code created by you in the **SOU ASSEMBLER** module into, a runnable machine language program.

61). Now, lets go back to our discussion, type **ASSEMBLE** and hit return.(a ? can be used to abort)

62). A number 1 and a solid cursor should be present. Now, let me explain to you the format of a Source Code line.

```
  1          2          3           4           5
  -          -          -           -           -
LINE#(space)LABEL(space)OPCODE(space)OPERAND(space)COMMENT
```

Number 1 is the **LINE** number field and it is automatically increased for you by the conputer.

Number 2 is the **LABEL** field, you can type-in labels with as many as six characters in them.

Number 3 is the **OPCODE** field, this is the three character wide field used to type-in the 6502/6510 opcodes and the special pseudo-ops.

Number 4 is the **OPERAND** field, this is a ten character wide field used to type-in the operands (numerals or labels).

Number 5 is the COMMENT field, this is the space allocated for you to type REM LIKE comments (always start this field with the character ";").

63). By looking at the Source Code line, you will notice that all 5 fields are separated by a space. This fact is very important, since we use spaces to define the beginning and the ending of each field. There is no need to enter all five fields in a Source line, but the field positions should be defined by their preceding spaces. For Example :

A) To enter an opcode and it's operand type :

(space)LDA(space)#$00 then hit return.

B) To enter an opcode without an operand type :

(space)INC then hit return.

C) To enter an opcode without an operand, but with a comment type :

(space)INC(space)(space);ANY KIND OF REM-LIKE COMMENT then hit return.

D) To enter a label, an opcode, it's operand and no comment type :

LABEL(space)LDA(space)$FFFF then hit return.

E) To enter a starting address using the ORG pseudo-op type :

(space)ORG(space)$C000(space);THIS COULD BE A REMARK then hit return.

VERY IMPORTANT :

The ORG Pseudo op can only be used once in a Source Code program and preferably on the first line of code or it can be completely omitted, since the OBJ ASSEMBLER module will ask you for a start address when it doesn't find a ORG Opcode in your Source Code.

F). To use the ASC Pseudo-Op and a label with it type: (Note: No comments are allowed in the same line that an ASC or a BYT Pseudo-Op have been used).

LABEL(space)ASC(space)'Message' then hit return.

Notice that in front of the "M" on message, we typed a APOSTROPHE mark. In addition, we put the APOSTROPHE mark at the end of the message. These two APOSTROPHES are being used to mark the beginning and the end of a word that is to be converted to ASCII (The ' ' APOSTROPHES are being used as Delimitors). Since sometimes people would like to use ' ' APOSTROPHES inside a message or prompt, we have made this Assembler to be able to recognize other delimitors besides the APOSTROPHES for Example:

LABEL (space)ASC(space)?Message? then hit return.
or
(space)ASC(space).Message. then hit return.

This would be okay as long as you start and end a word or phrase with the same delimitor.

G). To use the BYT Pseudo-Op type:

LABEL(space)BYT(space)20,49,$FF,$09 then hit return.
or
(space)BYT(space)40,$00,7,8 then hit return.

Notice that you can input either hex or decimal numbers, also that each number is separated by a COMMA (No commas are allowed at the end of a line and no COMMENTS are allowed in a line that has had an ASC or BYT Pseudo-Op).

H). To use the EQU Pseudo-Op type:

VARIABLE or LABEL(space)EQU(space)$FF08
or
VARIABLE or LABEL(space)EQU(space)10

Equates (EQU) should be placed at the beginning of the program with the ORG Pseudo-Op Line being the only line of code that can be ahead of them. This is especially critical when you want to append or link several programs. In this case, put all the Equates in the very beginning of the very first program. Do not place Equates in any other one of the program that are to be appended or linked.

I). To enter a line with the DST Pseudo-op type:

LABEL or VARIABLE(space)DST(space)200 then hit return.
or
LABEL(space)DST(space)2 then hit return.

J). Also, we can use the characters (;) and (*) to help us comment and beautify our program. For Example:

Type at the beginning of a Source Line the character (*) and then type after it any kind of comment that you would like to have, then hit return. (The (;) character can be used the same way).

64). Now let's get acquainted with the assembler by examining a program. Type DLOAD TEST3 and hit return. Now list the program and examine it's contents. Take special care on observing how the ORG,EQU,COMMENTS,LABELS, the #>, the #<, and #' were used to form this little program. (Note: Test3 is the same sample listing included within this manual at page # 32).

65). Now let's enter the OBJ ASSEMBLER Module:

Step 1. Type OBJECT and hit return; then, answer the question by pressing "Y", wait for the prompt and press return.

Step 2. 60 seconds later, you should be presented with another copyright screen and a prompt inquiring about the SOURCE FILENAME desired to be converted into an Object Code. At this point, answer by typing TEST and pressing return. (Typing EXIT and hitting return will abort the program). The computer will then display the prompt OBJECT FILENAME:TEST, with the cursor blinking on top of the letter T. At this point, you have the option of changing the name of the Output File, or the option of leaving the Output File name the same as the Input filename. For now, just press return.

Step 3. A prompt will ask you if everything was correct. Type "N" if there is something to be corrected. If not, simply press return when the blinking "Y" is present.

Step 4. The program called TEST.SC will be loaded in memory. The computer will then ask you whether or not you'd like to link any other program to the TEST program. Press "Y" (press "N" if only one program is to be assembled). Now, you will be asked to enter the name of the new file to be linked; so, type TEST1 and hit return. You are asked again if everything is correct. Hit return again.

Step 5. After TEST1.SC is loaded on memory, the computer will ask you again if you like to link another program. For now, type "N" (there is only one limitation when using the linker feature, and it is that the total sum of all lines from all programs to be linked can not be more than 1000 lines).

Step 6. The computer will prompt you, that it is looking for the starting address or origin. If it doesn't find an origin, it will ask you to enter one at this time. Even if it finds one, it will still gives you a chance to change your mind. Type "Y", and then return if you want to change the start address or press return when the blinking "N" is present. Now it will ask you to press return to start executing pass 1 and pass 2. Also, it gives you time to change disks if you want to assemble to a different disk.

Step 7. If you are ready, press return. Now it will prompt you while it is doing pass 1 and pass 2. Then, it will ask you whether to assemble to screen or to the printer (regardless of your choice, it will save the assembled program to disk).

Step 8. For our purposes, press return when the "S" is present. Now, you will see the assembly scrolling across your eyes. These programs: the TEST and TEST1 are supplied for demonstration purposes only, and they are not runnable programs (TEST3 is runnable. For a demonstration, run TEST3 program thru the OBJ.ASSEMBLER and reset the computer by turning it off and on, then LOAD"TEST3.OC",8,1. Once it is loaded, type NEW and Hit return, now type SYS49152 and press return. Load any long basic program, then list that program, and while it is listing, press the "@" key. It should freeze the listing. Pressing the "@" again should continue the listing).

At the ending of the assembly, a prompt with the number of errors is generated. If errors were present, a list of these errors and their addresses will be displayed. Also, a message asking you to scratch that file will be generated.

If no errors were encountered, a reference label table will be generated, letting you know that the assembly was successful.

Now before we examine the SYMBOLIC mode, we like to give you the E-Z ASSEMBLER Rules of Thumb.

General Rules :

1). ORG = Only one ORG is allowed and always at the beginning of a program (But as we talk before it can be omitted).

2). ASC & BYT = A comment is not allowed in a line that has had either of these Pseudo-Ops.

3). EQU = Place all Equates at the very beginning of program. When appending or linking several programs, REMENBER to place all Equates at the very beginning of the very first program.

4). DST = Use DST (Declare Stable Tables) to form tables or buffers.

5). Spaces = Remember to use proper spacing syntax when entering code.

6). Appending & Linking = Do not append or link programs whose total line sum will be more than 1000 lines.

7). BYT is equal to BYTE to avoid confusion.

LABELS

The most useful part of this Assembler is it's capability of using labels.

Labels are words or names that refer to a certain value, memory location or to a certain part of a program.

In practice, by using labels throughout your programs, you would have the capability of assembling programs that can readily be relocated in different parts of your available memory. This is accomplished by simply telling the OBJ ASSEMBLER module to start assembling your program in a different starting address.

EXAMPLES OF LABELS :

```
START LDA #$00
      TYA
LOOP  BNE START
      STA RESET
      LDA $00
      BEQ END
JUMP  JMP START
END   RTS
```

Notice that this labels are referring to each other by their name and not by their physical address location. Therefore the program has the freedom to be relocated into any space of memory that one desires to assemble to, by simply changing the starting address on the program.

We recommend the following reading material to enhance your knowledge on assembly language.

1). 6502 Software Design; Leo Scanlon.

2). Advanced 6502 Interfacing; Leo Scanlon.

3). Programming The 6502, Osborne.

4). C-64 Programmers Reference Guide
Howard W. Sams & co., Inc.

5). MOS Microcomputers Software Manual
Commodore Business Machines.

6). Machine Language for Beginners; Richard Mansfield.

7). What's really inside the Commodore 64;
Milton Bathurst.

8). The Anatomy of the Commodore 64, Abacus Software.

9). Machine Language on the Commodore 64, Abacus Software.

10). Advanced Machine Language on C-64, Abacus Software.

The following pages will show you how to become familiar with our E - Z Symbolic Disassembler which is also included in the E - Z Assembler package.

Module 3). **SYMBOLIC DISASSEMBLER** .- Load "UT",8,1

This module can be accessed by Loading"UT",8,1 (This will Auto-Load & Auto-Start the **SYMBOLIC DISASSEMBLER** Module), or while in the **SOU ASSEMBLER** module, by calling on the **SYMBOLIC** Mode Command and then answering the questions given by the computer. This module scans a given machine code program (object file) and generates a corresponding Assembly Language Source file, that in return can be DLoaded by our **E-Z SOU ASSEMBLER** module in order to be examined or modified.

Now let's get acquainted with the **SYMBOLIC DISASSEMBLER.**

Step 1. Type Load "UT",8,1 and press return.
or
While in the **SOU ASSEMBLER** Module, type SYMBOLIC and hit return; then, answer the questions by pressing "Y", wait for the prompt and press return.

Step 2. Sixty seconds later, you should be presented with another copyright screen and a prompt inquiring about the OBJECT FILENAME desired to be converted into a Source Code (Disassembled).

At this point, answer by typing "TEST3.OC" and pressing return (typing EXIT and hitting return will abort the program). The computer will then display the prompt SOURCE FILENAME:TEST3.OC, with the cursor blinking on top of the letter "T".

Now, you have the option of changing the name of the Output File, or the option of leaving the Output Filename the same as the Input Filename. For now, just press return.

Step 3. A prompt will ask you if everything was correct. Type "N" if there is something to be corrected. If not, simply press return when the blinking "Y" is present. The computer will then open up the object file (Disk Drive red light comes on) and will set some pointers up.

Step 4. At this point you will be asked if you would like to have the Bit operations converted to Byte operations (Read page # 28 for explanation of this feature), Hit return when the blinking "N" is present.

Pass #1 shows you that program is searching for origin of object code file. Then, it will give you the origin and end address and also the total length of the file to be disassembled.

Step 5. The program will ask you to input the starting and ending address of the part of code that you would like to have disassembled. The prompt **START ADDRESS:** contains the default value of **TEST3.OC** which is **49152**. When the blinking cursor is on top of "4" on **49152**, hit return. (You can change this number to a higher number to disassemble a specific area smaller than the complete file that is being disassembled.)

Step 6. The computer will present you with the prompt **END ADDRESS:49183** (this number can also be changed, but to a smaller number). Hit Return again.

Step 7. The **Pass #2** prompt will be displayed indicating that the disassembler is creating a Label Table (the red light on the Disk Drive comes on). In a few seconds, you will have two more prompts present. The first one will inform you of the number of labels that are located within the Source File. The second one will inform you of the number of labels that are located outside the Source File (generally, these will be kernal calls).

Step 8. Now, the computer will ask you where would you like to have the disassembly sent to, the Printer or the Screen. Simply Press Return to select screen for now.

Step 9. The **Pass #3** prompt will be displayed and a few seconds later, the source listing of the **TEST3.OC** file will be scrolled across your screen. After the scrolling is completed, Pass **#4** & **#5** are executed. A prompt indicating that the source file is being saved to disk appears, and a few seconds later (depending on the length of the file), the name of the file as it was saved will appear in reverse character format. Then, prompts appear letting you know that no errors were present, and that the Source File is ready to be used by the E-Z SOU.ASSEMBLER Module. Also, Re-Entry To Program Instructions appear.

Notes on the BIT Instruction

The technique explained in the following lines is supplied to you in case you come across it in someone else's program, since it's a fairly widely used and accepted 6502 - 6510 programing practice. Generally though, programmers who use tricks like this enjoy writing obscure code to save a byte or two of memory, and don't care if anyone else can look at the program and understand it. Many programs, including those printed in computer magazines, are designed to be easily read by people, not computers, and should keep away from such brain-twisting exercises. But giving such advice to a hacker is about as effective as advising a kid not to step in puddles on his way home from school.

If you have ever looked through someone's machine language program and come across a seemingly useless BIT instruction (for example BIT $FFA9), or an inexplicable. BYTE $2C, there is a explanation to this madness.

The BIT instruction sets the Zero, Minus, and Overflow flags based on the contents of the given memory location. In some instances, BIT is used almost like a NOP, but with one major difference: the two operand bytes used to specify the memory location are part of the instruction, and so are not executed as instructions if the BIT is executed. If the first byte of the instruction ($2C) is skipped however, you can execute a 2-byte instruction. For example, consider the following assembler code:

```
START1 BYT $2C
START2 LDA #$FF
```

If a program were to execute the code starting at START1, the CPU would see a $2C which is a BIT instruction, and interpret the next two bytes (LDA #$FF) as the argument for the BIT - in this case, the CPU would see: BIT $FFA9.

If the $2C was skipped over and instructions were executed from START2, the CPU sees the bytes $A9, $FF and interprets the LDA #$FF instruction normally.

Using the above technique allows you to enter a routine with the "A" ACCUMULATOR intact, and later enter the routine one byte past the start and have the register changed to something else before the routine does its thing. Of course, any register may be used instead, or any 1 or 2 byte op code can be executed after the $2C.

Take some time and, study carefully all the instructions in this manual, happy computing, and luck on your Assembly Language endeavor.

6502/6510 MNEMONICS

```
00 BRK              20 JSR              40 RTI
01 ORA (Indirect,X) 21 AND (Indirect,X) 41 EOR (Indirect,X)
02 Future Expansion 22 Future Expansion 42 Future Expansion
03 Future Expansion 23 Future Expansion 43 Future Expansion
04 Future Expansion 24 BIT Zero Page    44 Future Expansion
05 ORA Zero Page    25 AND Zero Page    45 EOR Zero Page
06 ASL Zero Page    26 ROL Zero Page    46 LSR Zero Page
07 Future Expansion 27 Future Expansion 47 Future Expansion
08 PHP              28 PLP              48 PHA
09 ORA Immediate    29 AND Immediate    49 EOR Immediate
0A ASL Accumulator  2A ROL Accumulator  4A LSR Accumulator
0B Future Expansion 2B Future Expansion 4B Future Expansion
0C Future Expansion 2C BIT Absolute     4C JMP Absolute
0D ORA Immediate    2D AND Absolute     4D EOR Absolute
0E ASL Absolute     2E ROL Absolute     4E LSR Absolute
0F Future Expansion 2F Future Expansion 4F Future Expansion
10 BPL              30 BMI              50 BVC
11 ORA (Indirect),Y 31 AND (Indirect),Y 51 EOR (Indirect),Y
12 Future Expansion 32 Future Expansion 52 Future Expansion
13 Future Expansion 33 Future Expansion 53 Future Expansion
14 Future Expansion 34 Future Expansion 54 Future Expansion
15 ORA Zero Page,X  35 AND Zero Page,X  55 EOR Zero Page,X
16 ASL Zero Page,X  36 ROL Zero Page,X  56 LSR Zero Page,X
17 Future Expansion 37 Future Expansion 57 Future Expansion
18 CLC              38 SEC              58 CLI
19 ORA Absolute,Y   39 AND Absolute,Y   59 EOR Absolute,Y
1A Future Expansion 3A Future Expansion 5A Future Expansion
1B Future Expansion 3B Future Expansion 5B Future Expansion
1C Future Expansion 3C Future Expansion 5C Future Expansion
1D ORA Absolute,X   3D AND Absolute,X   5D EOR Absolute,X
1E ASL Absolute,X   3E ROL Absolute,X   5E LSR Absolute,X
1F Future Expansion 3F Future Expansion 5F Future Expansion
```

60	RTS	80	Future Expansion	A0	LDY Immediate
61	ADC (Indirect,X)	81	STA (Indirect,X)	A1	LDA (Indirect,X)
62	Future Expansion	82	Future Expansion	A2	LDX Immediate
63	Future Expansion	83	Future Expansion	A3	Future Expansion
64	Future Expansion	84	STY Zero Page	A4	LDY Zero Page
65	ADC Zero Page	85	STA Zero Page	A5	LDA Zero Page
66	ROR Zero Page	86	STX Zero Page	A6	LDX Zero Page
67	Future Expansion	87	Future Expansion	A7	Future Expansion
68	PLA	88	DEY	A8	TAY
69	ADC Immediate	89	Future Expansion	A9	LDA Immediate
6A	ROR Accumulator	8A	TXA	AA	TAX
6B	Future Expansion	8B	Future Expansion	AB	Future Expansion
6C	JMP Indirect	8C	STY Absolute	AC	LDY Absolute
6D	ADC Absolute	8D	STA Absolute	AD	LDA Absolute
6E	ROR Absolute	8E	STX Absolute	AE	LDX Absolute
6F	Future Expansion	8F	Future Expansion	AF	Future Expansion
70	BVS	90	BCC	B0	BCS
71	ADC (Indirect),Y	91	STA (Indirect),Y	B1	LDA (Indirect),Y
72	Future Expansion	92	Future Expansion	B2	Future Expansion
73	Future Expansion	93	Future Expansion	B3	Future Expansion
74	Future Expansion	94	STY Zero Page,X	B4	LDY Zero Page,X
75	ADC Zero Page,X	95	STA Zero Page,X	B5	LDA Zero Page,X
76	ROR Zero Page,X	96	STX Zero Page,Y	B6	LDX Zero Page,Y
77	Future Expansion	97	Future Expansion	B7	Future Expansion
78	SEI	98	TYA	B8	CLV
79	ADC Absolute,Y	99	STA Absolute,Y	B9	LDA Absolute,Y
7A	Future Expansion	9A	TXS	BA	TSX
7B	Future Expansion	9B	Future Expansion	BB	Future Expansion
7C	Future Expansion	9C	Future Expansion	BC	LDY Absolute,X
7D	ADC Absolute,X	9D	STA Absolute,X	BD	LDA Absolute,X
7E	ROR Absolute,X	9E	Future Expansion	BE	LDX Absolute,Y
7F	Future Expansion	9F	Future Expansion	BF	Future Expansion

C0	CPY Immediate
C1	CMP (Indirect,X)
C2	Future Expansion
C3	Future Expansion
C4	CPY Zero Page
C5	CMP Zero Page
C6	DEC Zero Page
C7	Future Expansion
C8	INY
C9	CMP Immediate
CA	DEX
CB	Future Expansion
CC	CPY Absolute
CD	CMP Absolute
CE	DEC Absolute
CF	Future Expansion
D0	BNE
D1	CMP (Indirect),Y
D2	Future Expansion
D3	Future Expansion
D4	Future Expansion
D5	CMP Zero Page,X
D6	DEC Zero Page,X
D7	Future Expansion
D8	CLD
D9	CMP
DA	Future Expansion
DB	Future Expansion
DC	Future Expansion
DD	CMP Absolute,X
DE	DEC Absolute,X
DF	Future Expansion
E0	CPX Immediate
E1	SBC (Indirect,X)
E2	Future Expansion
E3	Future Expansion
E4	CPX Zero Page
E5	SBC Zero Page
E6	INC Zero Page
E7	Future Expansion
E8	INX
E9	SBC Immediate
EA	NOP
EB	Future Expansion
EC	CPX Absolute
ED	SBC Absolute
EE	INC Absolute
EF	Future Expansion
F0	BEQ
F1	SBC (Indirect),Y
F2	Future Expansion
F3	Future Expansion
F4	Future Expansion
F5	SBC Zero Page,X
F6	INC Zero Page,X
F7	Future Expansion
F8	SED
F9	SBC Absolute,Y
FA	Future Expansion
FB	Future Expansion
FC	Future Expansion
FD	SBC Absolute,X
FE	INC Absolute,X
FF	Future Expansion

```
                         E - Z ASSEMBLER V1.5  PAGE # 32

                 MEMOTRON MACRO - ASSEMBLER REVISION V1.5


FILENAME: TEST3.OC

LINE LOC.   CODE     LABEL  OP. OPERAND         COMMENTS
---- ----   ----     -----  --- -------         --------

0001 C000                   ORG $C000      ;THIS IS THE STARTING ADDRESS
0002 C000            *
0003 C000            *
0004 C000            *
0005 C000            GETCHR EQU $FFE4      ;KERNAL CALL
0006 C000            IRQVEC EQU $0314      ;IRQ VECTOR POINTER
0007 C000            *
0008 C000            *
0009 C000            ******************
0010 C000            * INTERRUPT TEST *
0011 C000            ******************
0012 C000            *
0013 C000            *
0014 C000            ;
0015 C000 78         CHANGE SEI            ;DISABLE IRQ
0016 C001 A9 0D             LDA #<NEWIRQ   ;GET LOW-BYTE OF NEW LOOP
0017 C003 8D 14 03          STA IRQVEC     ;AND PLACE IT ON IRQ VECTOR
0018 C006 A9 C0             LDA #>NEWIRQ+1 ;DO HI-BYTE NOW
0019 C008 8D 15 03          STA IRQVEC+1
0020 C00B 58                CLI            ;ENABLE IRQ
0021 C00C 60                RTS            ;BACK TO BASIC
0022 C00D            ;
0023 C00D 20 E4 FF   NEWIRQ JSR GETCHR     ;SCAN KEYBOARD
0024 C010 C9 40             CMP #'@        ;FOR '@'
0025 C012 D0 07             BNE EXIT       ;IF NOT EXIT
0026 C014 20 E4 FF   WAIT   JSR GETCHR     ;IF HERE WE ARE FREEZED WAITING FOR AN
OTHER '@'
0027 C017 C9 40             CMP #'@        ;SCAN IF '@'
0028 C019 D0 F9             BNE WAIT       ;LOOP AND LOOP
0029 C01B 4C 31 EA   EXIT   JMP $EA31      ;GO BACK TO REGULAR IRQ ROUTINE.
0030 C01E            *
0031 C01E            *
0032 C01E            *********
0033 C01E            * TEST3 *
0034 C01E            *********
0035 C01E            *
0036 C01E            *
0037 C01E            *

OBJECT ASSEMBLY COMPLETED.


  ¥ 0  ERRORS


REFERENCE LABEL TABLE: (BY ADDRESS)

GETCHR-$FFE4   IRQVEC-$0314   CHANGE-$C000   NEWIRQ-$C00D   WAIT---$C014
EXIT---$C01B
```

BACK-UP / REPLACEMENT DISKETTES

Orders for replacement diskettes must be accompanied by the defective diskette. The prices and terms quoted below are subject to change without notice. Contact MEMOTRON for current prices and terms.

Orders for a back-up or replacement diskette must be prepaid or may be charged to your credit card. Purchase orders will not be accepted. Kansas residents must add sales tax or include a sales tax exemption form.

Proof of purchase is necessary on all transaction.

Please send the order form to :

MEMOTRON SOFTWARE
806 NORTH WHEELER
McPHERSON, KANSAS. 67460

ORDER FORM

You may use this form (or copy of this form) to order BACK-UP / REPLACEMENT E - Z Assembler program diskettes.

Check one :

( ) I am enclosing a check or money order for $10.00 to cover the purchase of a BACK-UP diskette of the E - Z Assembler program.

( ) I am enclosing a defective E - Z Assembler diskette for exchange under warranty. It has been 30 or fewer days since I purchased E - Z Assembler.

( ) I need a replacement for my program diskette, but the warranty has expired. I am enclosing a defective E - Z Assembler diskette and a check or money order for $7.50

( ) Instead of check or money order, please charge my :

( ) MASTERCARD
( ) VISA

Account Name ____________________________________

Card Number ____________________________________

Valid Date ____________________________________

Expiration Date ________________________________

Date Purchased _________________________________

Name ____

Company ____

Address ____

City ____

State __________________ Zip ___________________

Telephone ______________________________________

PROPRIETARY NOTICE

The information and product design as disclosed in this manual (as they pertain to the E - Z ASSEMBLER) were originated by and are the property of MEMOTRON SOFTWARE.

Specifications and information herein are subject to change to allow the introduction of design improvements.

The contents of this manual have been reviewed for accuracy; however, no responsability is assumed should any portion not be accurate. Inevitably, some typographical or other errors will be found. The author will be grateful for any comment by alert readers so that future editions may benefit from their experience. Any other suggestions for improvements of the manual will be appreciated.

COPYRIGHT NOTICE

WARRANTY

MEMOTRON SOFTWARE makes no warranties, expressed or implied as to the fitness of this software product for any particular purpose. In no event will MEMOTRON SOFTWARE be liable for consequential damages. MEMOTRON SOFTWARE will replace any copy of the software which is unreedable if returned within 30 days of purchase. Thereafter, there will be a nominal charge for replacement.