Texas Instruments Home Computer

Line-by-Line Assembler

This cassette contains the LINES graphics demonstration program (operational only on the TI-99/4A computer) and a line-by-line symbolic assembler program that lets you create your own TMS9900 assembly language programs from the keyboard of your TI Home Computer.

These programs require the use of a cassette recorder/player (not included).

TABLE OF CONTENTS

This program was developed by the staff of Texas Instruments Personal Computer Division, with contributions by:

Granville Ott
Paul Urbanus

This manual was developed by the staff of the Texas Instruments Learning Center, with contributions by:

Ira McComic
Jacquelyn Quiram
Jan Stevens

Line-by-Line Assembler

INTRODUCTION

The cassette tape enclosed with the Mini Memory *Solid State Software*™ Command Module contains a line-by-line symbolic assembler and a graphics demonstration program named LINES. The Line-by-Line Assembler allows you to enter TMS9900 assembly language source code, one line at a time, directly from the computer keyboard. The LINES demonstration program (operational only on the TI-99/4A Home Computer) automatically draws colorful lines on the screen.

When the Assembler program is loaded in the Mini Memory module, each source statement you enter is immediately assembled into object code and stored in the memory locations specified by your source code. Therefore, as soon as you complete the entry of your program and store its name and address in the REF/DEF table, it is ready to be run.

IMPORTANT NOTE: Since code is assembled and stored directly, as you enter each line, be sure that the memory addresses specified in the program are available. Otherwise, no code will be generated or stored.

Although the Assembler converts each instruction into machine code as it is entered, some source code is retained in a nine-page text buffer. You can scroll the screen to review previously entered lines of source code by pressing the up- and down-arrow keys.

This manual discusses the features of the Line-by-Line Assembler, assuming that you already know assembly language programming. For a complete reference guide to the TMS9900 assembly language, see the *Editor/Assembler* owner's manual. Instructions for running the LINES demonstration program are also given in this manual (see "Loading the Line-by-Line Assembler").

Note: The LINES program is operational only on the TI-99/4A Home Computer, which has the enhanced graphics processor required by the program.

LOADING THE LINE-BY-LINE ASSEMBLER

Both the Line-by-Line Assembler and the LINES graphics program are loaded at the same time from the cassette tape by the L (LOAD) command of the EASY BUG debugger option. The steps below describe the loading procedure.

1. With the Mini Memory module installed in the computer console, attach your cassette recorder/player to the console, as described in the *User's Reference Guide*.
2. Press any key to make the master selection list appear, and select the MINI MEMORY option from the list. When the Mini Memory selection list appears, press **3** for REINITIALIZE to prepare the memory for loading a new program. Then press **QUIT** to return to the computer's master title screen.
3. Insert the Assembler cassette into the recorder, and rewind the cassette tape.
4. Press any key to make the master selection list appear, and then select the EASY BUG option from the list.
5. When the EASY BUG command description screen appears, press any key to clear the screen. Then type **L** and press **ENTER** to start the loading process. From this point on, the screen displays instructions to help you through the procedure. Follow these directions to load the program.
6. After the Assembler program is loaded, press **QUIT** to return to the master title screen. Then press any key to go on to the master selection list, and select the MINI MEMORY option.
7. When the Mini Memory selection list appears, press **2** to select the RUN option. The screen clears, and the program prompts you for the name of the program you want to run.

   A. If you want to run the LINES program (available only on the TI-99/4A), type LINES and press **ENTER**. The program draws a colorful line design on the screen. If you press the **C** key, you can "freeze" the color of the current line, and all subsequent lines will be that color. Pressing **C** again cancels the single-color effect. To stop the program, press **QUIT**.

B. If you want to create a new assembly language program, type NEW and press **ENTER**. The program enters the Assembler, clears the Symbol Table (more on this later), and waits for your first program line entry.

C. If you want to continue writing a program that you began previously, type OLD and press **ENTER**. The old Symbol Table is retained, and the screen displays the next memory location, ready for you to continue your program.

Note: Your Mini Memory module may already be loaded with the Assembler and LINES programs. To check, select the Mini Memory RUN option, and enter the appropriate program name when the prompt appears. If present, the program immediately begins to run. If the program has not been loaded into the module, the screen reports "PROGRAM NOT FOUND."

You may also review the LINES code by means of the M command in EASY BUG. Just enter M7CD6, and keep pressing **ENTER** to review each line of code without changing any data.

IMPORTANT NOTE:

When you enter and assemble a program, the Symbol Table may overwrite part of the LINES program stored in the Mini Memory module. When you want to run LINES again, simply load the program again from the cassette tape into the module.

ASSEMBLER SYNTAX

Each line (or record) of your source program consists of four sections known as fields. These fields, if present (some are optional), must appear in the order and format (syntax) required by the Assembler. In this manual, the following conventions are used in the syntax definitions for machine instructions and directives.

- Items in capital letters, including special characters, must be entered exactly as shown.
- Items within brackets ([ ]) are optional.
- Items within angle brackets (< >) are required fields.
- A lower-case *b* indicates a single space.
- A lower-case *b* followed by three dots (*b*...) indicates one or more spaces.

The general syntax for an Assembler directive is as follows:

[label]*b*...<opcode>*b*[operand][,operand][*b*...comment]

The label field requires either a space (when there is no label) or one or two characters. The first character must be alphabetic; the second character, if present, may be alphanumeric. The label is followed by one or more spaces. If you do not type a label, pressing the **SPACE BAR** moves the cursor automatically to the beginning of the opcode field.

The opcode field contains the operation code of the task to be performed by the source statement. The field consists of one to four alphabetic characters, such as A for Add or AORG for the Absolute Origin directive, followed by a single space.

The operand field contains one or two operands, as required by the particular instruction. Note that the operand field has no spaces within it, and multiple operands are separated by commas. The operand field is concluded by pressing the **SPACE BAR** (cursor advances to the comment field) or **ENTER** (signifies the end of the line). If an instruction has no operand, the operand field is omitted.

The comment field, if used, may include any character, and it continues until you press **ENTER** to end the line.

Examples:

```
XY      MOV R1,@VP   Save R1 in VP.
Z       S R1, R2     Calculate difference.
```

The Line-by-Line Assembler predefines certain symbols. When an operand includes a dollar sign ($) as an initial character, it is considered to refer to the contents of the location counter. For example, at location >7D00, the statements

```
        JMP $+8
```

and

```
        JMP >7D08
```

are considered to be equivalent. When specifying register operands, you can use the symbol R, followed by a decimal number. Thus,

```
        MOV R1,R15
```

and

```
        MOV 1,15
```

are equivalent.

Note: The default number system for the Line-by-Line Assembler is decimal; hexadecimal numbers are indicated by the greater than (>) prefix.

ASSEMBLER DIRECTIVES

This section discusses the seven *directives* recognized by the Line-by-Line Assembler. An assembler directive should not be confused with an assembly language instruction, which tells the microprocessor to perform a single function only, such as Add or Move. Directives are programming-aid commands which direct the Assembler to perform certain operations at assembly time, and the Assembler may execute many instructions to satisfy one directive. (For a discussion of the TMS9900 assembly language instructions, see the *Editor/Assembler* owner's manual.)

The directives described here are:

AORG	Absolute Origin
BSS	Block Starting with Symbol
DATA	Word Initialization
END	End Program
EQU	Assembly-Time Constant Definition
SYM	Symbol Table Display
TEXT	String Constant Initialization

Absolute Origin—AORG

Format: [*label*] AORG <*address*>

This directive can be used to set the location counter to a specific value (always an even address) during Assembler operation. Generally, it is used as the first program entry to set the starting location of the assembled code; however, it can be used at any time during program entry.

Example:

`AORG >7D80`	Results in the next instruction assembled being stored beginning at memory address >7D80.

Block Starting with Symbol—BSS

Format: [*label*] BSS <*number of bytes to be reserved*>

The BSS directive reserves a block of memory (for variable storage or workspace registers) without initializing the space. Starting from the address specified by the label, the Assembler increases the location counter by the number of bytes specified in the directive.

The number of bytes must be zero or positive. The resulting value in the location counter is rounded down to an even number, if necessary. In other words, the least significant bit of the address is truncated if the resulting value is odd.

Example:

`WS`	`BSS 32`	Assuming that WS refers to memory location >7D00, increases the location counter to >7D20, reserving a 32-byte block as a workspace.

Word Initialization—DATA

Format: [*label*] DATA <*value*>

The DATA directive allows you to initialize a word or words of memory to a particular value. This directive is particularly useful for entering a table of data as part of your program. At any point in the assembly of your program, you can insert a DATA directive in the opcode field, followed by a constant or symbol as an operand.

The operand for a DATA directive may consist of an unresolved forward reference, a numeric constant, a defined symbol, or a string of numeric constants and defined symbols connected by plus (+) and minus (−) signs. If the operand is the latter (a string of sums and differences), no regard is given to carry or overflow.

Examples:

`DATA >1234`	Causes location to be initialized to >1234.
`DATA AX`	If AX = >3456, causes location to be initialized to >3456.
`DATA GH`	If GH is an undefined forward reference, location will be initialized to the value corresponding to GH when GH is resolved.
`DATA 1+5-3`	Initializes location to 3 (equivalent to DATA 3).

The DATA directive also supports a sequence of constants separated by commas.

DATA [*constant* (*defined or undefined*), *constant*,...,*constant*]

Note that an unresolved constant is acceptable only as the first operand of the statement.

The BSS and DATA directives are similar in function; however, the BSS directive simply sets aside memory space without initializing it, while the DATA directive both sets aside memory space and initializes it to a specific value (or values).

End Program—END

Format: END

The Assembler can be exited at any time by entering the END directive in the opcode field. When the END directive is entered, the Assembler displays the number of unresolved references, if any. If unresolved references exist, return to the Assembler and resolve them before exiting from the Assembler. Failure to do so may result in invalid opcodes in your program. The SYM directive (described below) is helpful in identifying unresolved references before ENDing your program.

When all references have been resolved, the Assembler displays the report

```
0000 UNRESOLVED REFERENCES
```

Pressing **ENTER** at this point exits from the Assembler and returns to the Mini Memory selection list. Pressing any other key except **ENTER** causes the Assembler to wait for your next instruction.

Assembly-Time Constant Definition—EQU

Format: <*label*> EQU <*defined constant*>

The EQU (equate) directive is used to define a value for a symbolic constant and to assign the value of one defined symbol to another symbol.

Note: No directive is provided to change the value of a symbol once it is defined.

Examples:

CD	EQU >A55A	Sets CD to the value of >A55A.
FG	EQU 15	Sets FG to the value of 15.
A	EQU FG	Sets A equal to FG.

Symbol Table Display—SYM

Format: SYM

The SYM directive allows you to review, at any time during the entry of a program, the reference symbols and their associated values, if any, which you have used in the program to that point.

The Symbol Table is displayed in three categories:

RESOLVED REFERENCES	Any label which has been defined (given a value).
UNRESOLVED REFERENCES (WORD)	Any label which has been referenced in an instruction other than a jump instruction.
UNRESOLVED REFERENCES (JUMP)	Any label which has been referenced in a jump instruction.

If a category has no symbols associated with it, that category is not displayed. If a symbol is unresolved (has been referenced but not defined), the address of the symbol is also displayed. If the symbol table is empty, the SYM directive is erased, and the Assembler waits for your next instruction.

Example:

```
 Location           Instruction           Comments
                    AORG >7D00     Set starting address of program.
7D00 0000   WS      BSS 32         Reserve workspace.
7D20 0201           LWPI WS        Load the workspace.
7D22 7D00
7D24 0201           LI R1,W1       Load Register 1 with undefined data.
7D26R0000
7D28R10FF           JMP J1         Jump to undefined address.
7D2A XXXX           SYM            Display Symbol Table.

RESOLVED REFERENCES
WS-7D00
UNRESOLVED REFERENCES (WORD)
W1-7D26
UNRESOLVED REFERENCES (JUMP)
J1-7D28
7D2A XXXX                          (XXXX is existing data in
                                   memory.) The Assembler waits
                                   for the next instruction.
```

If an unresolved symbol is referenced at more than one location, the symbol and the address of each reference, up to a maximum of 32 references, is displayed.

String Constant Initialization—TEXT

Format: [*label*] TEXT '<*character string*>'

The TEXT directive allows you to enter a string of characters and have them translated to ASCII code and stored as part of your program. Any displayable character, except the single quote character ('), may be entered as part of the TEXT statement, and the ASCII code for each character is stored in memory as you enter the character. Note also that the control and special function keys (**AID**, **REDO**, etc.) generate valid ASCII codes (in the range of >0 through >F) which are stored but not displayed on the screen.

The TEXT string may be as long as desired, and it must be preceded and terminated by a single quote (') character. If an odd number of ASCII characters is entered, a null byte (>00) is added to the string to force the location counter to an even boundary.

Example:

`TEXT 'ABCD'`	Stores the values >4142 and >4344 in the corresponding memory locations.

Note: The **ERASE** function does not affect characters in memory that have already been entered as part of the TEXT string.

THE SYMBOL TABLE

The Assembler allows unresolved forward word and jump references; that is, references to symbols that are defined later in the program. The Assembler keeps track of all the symbols defined or referenced in a program and stores this information in a Symbol Table.

The Symbol Table is actually a combination of three tables: defined symbol references, unresolved word references, and unresolved jump references. The number of entries in the table is also stored; since each entry is stored as four bytes, the physical length of the table is four times the number-of-entries value.

The Symbol Table starts at memory location >7CD8. Since each Symbol Table entry is four bytes long, be sure that the beginning address of your object code allows adequate room for the number of Symbol Table entries required for your program. Otherwise, when your program is assembled, the Symbol Table may overwrite the beginning of your object code.

Defined Symbol References

If an entry is a resolved label or other defined symbol, the label word stored in the Symbol Table is simply the ASCII equivalent of the symbol. A single character symbol is stored in its ASCII form preceded by the ASCII code for 1; for example, the symbol A is stored as >3141. The second word (the address word) contains the address, constant equivalent, or memory location corresponding to the label or symbol. For example, if label AC (ASCII >4143) is defined to be >8375, the Symbol Table entry is

4143	Defined symbol reference entry.
8375	Value.

Unresolved Word References

The Symbol Table entry for an unresolved word reference is similar to that described above, except that the most significant bit of the label word is set, and the address word points to the last location in which this label was used as a forward reference. For example, if the label AC is used as an unresolved forward reference in location >7E00, and no further references to this label have been made in your program, the Symbol Table entry is

C143	Unresolved word reference entry.
7E00	Address.

Unresolved Jump References

The Symbol Table entry for an unresolved jump reference is also similar, except that the most significant bit of the *least significant byte* of the label word is set, and the address word points to the location of the last jump instruction that uses this unresolved label. For example, if location >7D00 has an unresolved jump reference to label AC and no other references to this label have been made, the Symbol Table entry is

41C3	Unresolved jump reference entry.
7D00	Address.

The least significant byte of the unresolved jump instruction indicates the word distance to the most recent previous unresolved jump reference to the same label. If there is no previous reference, the byte is assigned the value of −1 (>FF).

Maximum Number of Displayed Unresolved References

The first time the Assembler places an unresolved symbol in the Symbol Table, the characters of the symbol are placed in the table followed by the address where the symbol was referenced. This address is called a pointer. The content of that address is then set to zero, indicating that this is the first reference to the unresolved symbol.

For example, consider the following program segment as it is assembled.

```
7D00 02E0          LWPI WS
7D02R0000
7D04 C820          MOV @WS,@DG
7D06R7D02
7D08R0000
                   SYM
UNRESOLVED REFERENCES (WORD)
WS-7D06 WS-7D02 DG-7D08
```

In this example, the content of the first unresolved reference to WS (at location >7D02) is set to zero to indicate the first reference to that symbol. The content of the next unresolved reference to WS (at location >7D06) is set to >7D02 (the address of the previous reference to WS.)

As the result of a typing error during program entry, an unresolved reference may appear to have an unlimited number of pointers. The following program segment gives an example.

```
7D00          W1   EQU >1234
7D00 0201          L1 R1,WS
7D02R000
7D04               AORG >7D00
7D00 0201          L1 R1,W1
7D02 1234
```

In this segment, WS appears in the Symbol Table as an unresolved word reference with a pointer of >7D02. The subsequent AORG directive causes location >7D02 to contain >1234. Therefore, there are an indefinite number of pointers to WS, since the Assembler considers the value >1234 in location >7D02 to be the pointer to a prior reference, and so on.

To prevent the possibility of an indefinite display of unresolved references in such a case, the SYM directive displays a maximum of 32 references to an unresolved symbol.

EDITING TECHNIQUES

As mentioned earlier, the Assembler retains some source code in a nine-page buffer, allowing you to review previously entered program lines. When you reach the end of the buffer, the Line-by-Line Assembler title scrolls back onto the screen to signify that you have filled the buffer and have returned to its beginning. Any new source code you enter will overwrite the previously entered source code. Therefore, it's a good idea to review your source code at that point, by using the up- and down-arrow keys to scroll the data on and off the screen, while the old source code is retained in the buffer.

If you find an error in your source code, make a note of the address of the incorrect line. Then use the AORG directive to return to that address, and retype the line correctly.

You can also correct typing errors as you are typing a line, by pressing **ERASE** or by using one of the methods discussed below.

A label, whether in the label field or in the operand field, can be corrected by simply continuing to type the correct symbols before pressing the **SPACE BAR** to exit from the field. For example, if you type VF instead of CD as a label, just type CD before you press the **SPACE BAR** to go on to the next field. The Assembler accepts the last two characters entered in the field as the correct label. If you need to correct a single-character label, type 1 to indicate that a single-character label is being used, and then type the correct alphabetic character before pressing the **SPACE BAR**.

A similar method can be used to correct a hexadecimal entry in the operand field. For example, if you type >1234 but meant to type >2234, simply continue to type the correct entry. In other words, your entry would be >12342234. The Assembler considers the last four characters as the correct hexadecimal entry. Corrections to decimal entries are somewhat more difficult, since the Assembler considers the last 16 decimal digits entered to be the correct entry. If your decimal value is less than 16 digits long, you must enter enough leading zeros to make the value 16 digits long. For best results, it is probably better to cancel the line by pressing **ERASE** and then retype it correctly.

An opcode entry cannot be corrected except by pressing **ERASE** to cancel the entire line and then retyping the line correctly.

ERROR CONDITIONS

During program entry and assembly, there are three conditions that result in the display of an error message.

Condition	*Displayed Message*
1. Attempting to redefine a previously defined label.	*ERROR*
2. Entering an undefined opcode or directive.	*ERROR*
3. Exceeding the reach of a jump instruction.	*R-ERROR*

Each error display is accompanied by a bad-response tone and is printed on the same line as the instruction which produced the error. Press any key to erase the line. (The location counter is unchanged.)

If you attempt to jump to an undefined label and subsequently find that the definition of the label causes previous jump instructions referring to it to be out of range, the address displayed to the left of the *R-ERROR* message is the address of an out-of-range jump instruction. Continue to press **ENTER** to see other addresses of out-of-range jump references to the same level.

If you anticipate that a jump instruction to an as yet undefined label might be out of range, it's a good idea to follow the jump instruction with two NOP (no operation) instructions, to allow patching your program if a reach error occurs. The following program segment illustrates this procedure.

```
7D00 XXXX         JNE J1
7D00R16FF
7D02 1000         NOP
7D04 1000         NOP
7D06 C081         MOV R1,R2
  .
  .
  .
7E10 XXXX    J1   EQU $
7D00 *R-ERROR*                (Press ENTER to see additional out-of-
                              range jump references to J1; then
                              press any key to clear the error
                              condition.)
7E10 XXXX         AORG >7000
7D00 1302         JEQ $+6     Note that the logic of the
                              replacement jump instruction is
                              opposite to that of the original.
7D02 0460         B @J1
7D04 7E10
```

RUNNING YOUR PROGRAM

After a program is assembled, the name and address of the program must be added to the REF/DEF table so that the Mini Memory module can find the program and run it. One way to add the name and address is to use the LOAD subprogram from TI BASIC (see "Additional TI BASIC Subprograms" in the *Mini Memory* owner's manual).

Another way to add the program's name and address is to use Assembler directives. First, you must determine that there is enough memory space left for you to add your program name to the REF/DEF table. After you have entered the last line of your program, use the AORG directive to read from memory the First Free Address in the module (FFAM) and the Last Free Address in the module (LFAM). The addresses of these two variables are >701C and >701E, respectively. Subtract the value in >701C from the value in >701E; if the difference is greater than 7 bytes, you have enough room to store your program name.

After insuring that there is enough room (8 bytes) to add the program name to the REF/DEF table, subtract 8 from the old LFAM, and poke the new LFAM value to >701E using the DATA directive. This reserves room in the REF/DEF table for your program name and address.

A program name may be from one to six characters in length; however, the program name in the REF/DEF table must be exactly six characters long. If the name of your program is less than six characters long, you must "pad" the name with trailing spaces when you enter it in the table.

Use the AORG directive again to get to the new entry point in the table, and then enter the program name by means of the TEXT directive. When you have entered the program name, the location counter advances to the next word boundary, where the program address (two bytes) is to be stored. Use a DATA directive again to enter the starting address of your program.

Example: Determining Remaining Memory Space

The following example assumes that you have just entered the "Sample Program" given in the *Mini Memory* owner's manual and have not exited from the Assembler.

Screen Shows	*You Enter*	*Comments*
7F04 XXXX	AORG >701C	>7F04 represents the current location counter address, and XXXX represents any data at that address.
701C XXXX		XXXX represents the address of the old FFAM.
701C 7F04	DATA >7F04	The DATA statement supplies the new FFAM, which is the first address immediately following the program.
701E 7FE8		>7FE8 represents the current address of the LFAM. Subtract the FFAM from this value; if the result is 7 bytes or greater, you have enough room for your program name.
701E 7FE0	DATA >7FE0	Subtract 8 bytes from the old LFAM, and store the result as the new LFAM by means of the DATA directive (>7FE0 represents the new LFAM).
7020 XXXX		The counter advances to the next location and displays any assembled data.

Thanks to
Fabio Brianese
for the Scan

Example: Adding Program Name and Address

The following example, starting from the end of the example above, shows you how to add the name and address of the DISP$ program (see "Sample Program" in the *Mini Memory* owner's manual) to the REF/DEF table.

Screen Shows	*You Enter*	*Comments*
7020 XXXX	AORG >7FE0	Gets to the new entry point in the table.
7FE0 XXXX		XXXX represents any value currently stored at location >7FE0.
7FE0 4449 7FE2 5350 7FE4 2420	TEXT 'DISP$ '	Enters the program name DISP$. (Note that one blank space is added to "pad" the name to six characters.) The characters of the name, including the blank space, are stored in the six bytes beginning at location >7FE0.
7FE6 XXXX		The counter advances to the next location, where the entry point of the program will be stored, and displays any value currently stored there.
7FE6 7E20	DATA DS	The label DS is equated to the address which is the entry point in the program.

You are now ready to exit from the Assembler and run your program.

STORING YOUR PROGRAM ON CASSETTE TAPE

To store a program on cassette tape, first press **QUIT** to leave the Mini Memory option; then select EASY BUG and use the S command. For best results, enter a starting address of >7000 and an ending address of >7FFF to be sure that the REF/DEF table and the pointers in low memory are saved. Otherwise, it will be necessary to enter your program name in the REF/DEF table every time you load the program.

For additional details, see the "EASY BUG Debugger" section of the *Mini Memory* owner's manual.

TEXAS INSTRUMENTS
INCORPORATED
DALLAS, TEXAS

Printed in U.S.A.

1036061-2