



The NanoAssembler

by Roger Wood
HCM Staff

This companion to NanoProcessor shows you how an assembler can provide easy access to machine language—by translating simple instructions into the computer's native tongue.

In the last issue (HCM Vol. 5, No. 5), we presented *NanoProcessor*, a program that introduced the concepts of machine-language programming. This program demonstrated how a microprocessor works at its most fundamental level. Although entering and running simple programs on the *NanoProcessor* can be fun, longer and more complicated machine-language routines are another story. Even with short programs, you probably discovered what a time-consuming and error-prone process it can be to enter machine language one bit at a time.

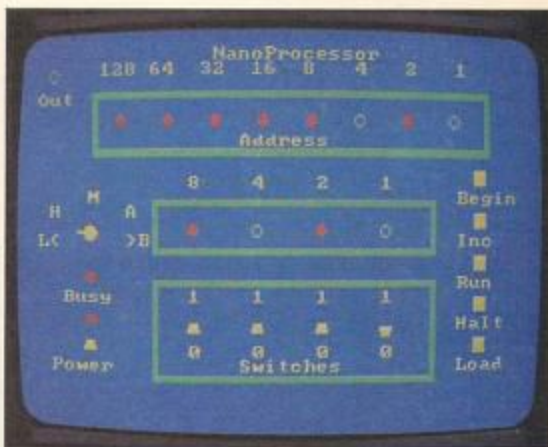
To alleviate the difficulties involved in working with machine language, early computer users created programs called "assemblers." An assembler is a human-to-machine translator. It operates from a "dictionary" of mnemonics (a combination of letters that humans can understand), translating these mnemonics into the numbers of machine code. Using assemblers, you can write a program with the more easily remembered mnemonics, and let the computer create the actual machine language (the ones and zeros).

Thus, we present the *NanoAssembler*: a program that will teach you how to use assemblers. With the *NanoAssembler*, you will be able to write long, complicated programs for the *NanoProcessor* much more easily than you would using machine language.

Source Code To Object Code

You may find that many people refer to "assembly-language programs" and "machine-language programs" interchangeably, as though they were the same thing. Actually, an assembly-language program is a text file—known as a "source file"—that the computer cannot execute directly. It is simply a series of text lines comprising mnemonics, numbers, and labels. Before the computer can run such a program, the source file must be "assembled" or translated into a machine-language file—also known as an "object file."

Take a look at Sample Program 1, which you can load and run on the *NanoProcessor*. You may recognize this program, as it is identical to Sample Program 1 in the last issue. The two left-most columns, entitled *Addr* and *Code*, contain the machine language, (object code), which makes up the program. You can enter this object code bit by bit, or you can enter the more easily read and (with some training) understood assembly language (source code), contained in the *Line*, *Label*, *Mnemonic*, and *Remark* columns. The *Remark* column is like a *REM* statement in BASIC. It makes the program much easier to read and understand.



Our *NanoAssembler* package consists of two BASIC programs: the *NanoEditor* and the *NanoAssembler*. The *NanoEditor* is a simple text editor that lets you enter your program as source code and save it to disk (or alternatively tape on the Atari, Commodore, and TI computers). *NanoAssembler* can then read and translate that file into a corresponding file of object code, which you can save to disk or tape. You can then load the object code into the *NanoProcessor* and run it.

Creating A Program

We will use Sample Program 1 to demonstrate how the *NanoEditor* and the *NanoAssembler* work. To start, Load and RUN The *NanoEditor*. You begin with this menu:

- 1) EDIT
- 2) FILES
- 3) PRINT
- 4) EXIT

Choose the Edit option, which allows you to create and modify files. The *Editor* now displays the command prompt: *CMD*. You may enter one of 5 single-letter commands:

Command	Function
A	Add a line of text
E	Edit a line of text
D	Delete a line of text
I	Insert a line of text
L	List

To begin creating a new file—in this case Sample Program 1—press A. In response, the *Editor* displays line 001, with a flashing cursor waiting for your input. For each line of source code, the *Editor* provides a line number ranging from 001 to 200. When you enter the *Add-a-line* mode, the program always displays the cursor on a new line of source code—one line past the last line in memory. You can automatically advance to the next line by pressing [ENTER] or [RETURN]. To exit the *Add-a-line* mode, press the [ESCAPE] key (see your computer's Control Capsule if your machine does not have an Escape key).

Now enter the contents of the Label, Mnemonic, and Remark columns. Because our *Editor* is in BASIC, your text input will be slower than with a full-blown word processor. The Label column is empty in line 001 of our sample program, so press the proper key or key combination (see your Control Capsule) to tab into the Mnemonic field. (We will explain labels below.) Now type in the first instruction: LDA# 3. You must enter the text exactly as it appears in the listing, or the *NanoAssembler* program will not interpret the code properly. Make sure there is no space between the A and the #. You must, however, place a space between the # and the 3.

This spacing is critical because the Mnemonic field actually consists of two sub-fields; and the space acts as a separator for these sub-fields. The left sub-field is the "op-code," or instruction field, which defines the actual instruction. In line 001, the op-code is LDA#. The right field contains the "operand." The operand is either a two-nibble address or a single-nibble quantity to be loaded or stored in a register or memory location. It defines the number that the op-code is to operate on. In line 001, the number 3 (%0011) is the operand.

After you have entered the first instruction, you may tab into the Remark field. On a program as short as this one, however, you may choose to save time by omitting the remarks. Continue entering lines 002, 003, and 004 in a similar fashion.

Once you've entered part or all of the program into memory using the Add command, you can use the other editing commands. Each of these commands prompts you for a particular line number. E lets you Edit an already-existing line in memory. D allows you to Delete a line, and I lets you Insert a line. The L command lets you List up to 10 lines of a program to inspect what is in memory. If the program extends more than 10 lines beyond the beginning line number that you specify, you have the option to either continue listing more lines or quit and return to the command line.

Labels As Labor Savers

In line 005 (HERE JMP HERE), you encounter an important assembly-language tool—the "label." In the *NanoAssembler*, we define a label as a group of up to 6 alpha-numeric characters, beginning with a letter—in our example, the word HERE. Assembler programs use labels in place of numeric quantities. In this case, HERE represents the address to be JuMPed to. One major advantage of labels is that you do not have to know the actual numeric addresses used in a program. Instead, the assembler uses the labels to assign the correct address to a particular instruction for you.

Before continuing, let's clear up an area that sometimes confuses a beginner at assembly language: the difference between line numbers of a source file and addresses of an object file. Each line in a source file contains only one op-code. But when you


assemble the source file into object code, the op-code may require as many as three addresses (see Figure 1 for the number of nibbles each instruction requires). Thus, a source file's line numbers and the actual addresses of the object code almost always differ. When the *Assembler* prints out its listing, the addresses and codes are located on the line just below the source code, representing the order of events during assembly.


By inspecting the two left-hand columns of Sample Program 1, you can see that the address to be JuMPed to is 6. You know this only because we have already assembled (or translated) the source code on the right into the object code on the left. If we hadn't provided the machine code, however, you would have to assemble all of the instructions to discover what address you wanted to JuMP to. The use of labels saves you from this tedious task and is one of the primary advantages of assemblers.


When you finish entering line 005 and press [RETURN] or [ENTER], a prompt tells you to enter line 6. This program has no line 006, so press the [ESCAPE] key for your machine (see your Control Capsule), and the program returns you to the command line. Now you can use the List command to see if you have entered everything correctly. If you find any errors, you can Edit the line or lines that they occur in. If you change a line, then decide that you don't want those changes, you can press the [ESCAPE] key instead of [RETURN] or [ENTER] to revert back to the original version of the line. This option is also available if you select Insert, but change your mind before finally entering the line.

From Editor To Assembler


After you are sure that you've correctly entered the program, save it to disk (or tape on Atari, C-64, or TI). To save your file, select option (2) Files. Then select the appropriate menu options, and enter the file name. If your operating system does not normally support extensions to file names (all but Atari and IBM), the name must be at least two characters shorter than a normal legal file name. The program will automatically append a .S (_S on the TI), for Source, so that you can use the same name for both source and object files without any confusion. If you have a printer, you may also wish to get a hardcopy of your program. This is helpful when you are tracking down errors during assembly. To use

CONTROL CAPSULE 	
NanoEditor	
KEY	FUNCTION
ESC	Escape
Edit Mode:	
BACKSPACE	Backspace
CONTROL D	Erase line
TAB	Tab
-	Cursor left
-	Cursor right
RETURN	Enter Line

CONTROL CAPSULE 	
NanoEditor	
KEY	FUNCTION
ESC	Escape
Edit Mode:	
DELETE	Backspace
SHIFT DELETE	Erase line
TAB	Tab
CONTROL -	Cursor left
CONTROL -	Cursor right
RETURN	Enter Line

CONTROL CAPSULE 	
NanoEditor	
KEY	FUNCTION
F1	Escape
Edit Mode:	
DEL	Backspace
F3	Erase line
F5	Tab
CRSR -	Cursor left
CRSR -	Cursor right
RETURN	Enter line

CONTROL CAPSULE 	
NanoEditor	
KEY	FUNCTION
ESCAPE	Escape
Edit Mode:	
BACKSPACE	Backspace
DELETE	Delete character
TAB	Tab
-	Cursor left
-	Cursor right
ENTER	Enter line

CONTROL CAPSULE 	
NanoEditor	
KEY	FUNCTION
FTCN 9	Escape
Edit Mode:	
FTCN 1	Delete
FTCN 3	Erase line
FTCN 7	Tab
FTCN 5	Cursor left
FTCN D	Cursor right
ENTER	Enter line

the Print option, just select it from the main menu (3). After you save (and print) the source file, select the Exit option from the main menu. The program gives you a chance to change your mind before ending, so you don't need to worry about losing the program in memory due to an erroneous keypress.

Now it is time to load and RUN the NanoAssembler. The program prompts you to load your source file for assembly. As the program translates your source code into machine code, it lists the source file, the addresses, and object code to either the screen or a printer (if you have one).

Passing Through

The actual assembly of the program occurs in two steps, or "passes." Thus, the NanoAssembler is a "two-pass" assembler. The first pass does most of the work, determining the correct machine-language instructions and the instruction addresses. However, sorting out labels requires a second pass because, until it identifies all address labels, the program may not know the exact address of each instruction.

Try assembling Sample Program 1. If you have entered it correctly, the NanoAssembler should output the assembled version, as shown in Figure 1, to the screen or printer. If you have made an error in entering the program into the NanoEditor, the NanoAssembler informs you of the line number in the source code that contains the error, and states the type of error. For example, if in line 1 you enter LDA #3 instead of LDA \$3, when you try to assemble the program the computer displays the error: ILLEGAL USE OF LABEL IN LINE 1. Here, the computer interprets the code as a Load A *addr* instruction (object code = 2), instead of a Load A immediate instruction (object code = 1). Then, when the computer evaluates the "label" #3, it finds that the label is illegal because it does not begin with a letter.

Figure 1: Instruction Set

Dec.	Binary	Nibbles per instr.	Mnemonic	Flags* affected C Z	Function
0	%0000	1	ADD	Y Y	Add the contents of B register to the contents of A register—result in A.
1	%0001	2	LDA#	N Y	Load A with number following instruction.
2	%0010	3	LDA addr	N Y	Load A with number at location specified by addr.
3	%0011	3	STA addr	N N	Store the contents of A at location specified by addr.
4	%0100	1	TAB	N N	Transfer contents of A to B.
5	%0101	1	TBA	N Y	Transfer contents of B to A.
6	%0110	1	RRC	Y Y	Rotate A right one bit through carry.
7	%0111	1	SLC	Y Y	Rotate A left one bit through carry.
8	%1000	1	AND	Y Y	Logically AND A and B—Result in A.
9	%1001	1	OR	Y Y	Logically OR A and B—Result in A.
10	%1010	1	XOR	Y Y	Logically XOR A and B—Result in A.
11	%1011	3	BZ addr	N N	Branch to addr if Zero flag is set.
12	%1100	3	BNZ addr	N N	Branch to addr if Zero flag is not set.
13	%1101	3	BYS addr	N N	Branch to addr if Carry flag is set.
14	%1110	3	BCC addr	N N	Branch to addr if Carry flag is not set.
15	%1111	3	JMP addr	N N	Branch to addr unconditionally.

Assembler Directives:

n/a	n/a	0	ORG	n/a	Use to specify a particular address (e.g., specify starting address of program).
n/a	n/a	0	EQU	n/a	Equate label with value—assigns the value to the right of the EQU statement to the label to the left.
n/a	n/a	1	DN	n/a	Define Nibble—assigns the value to the right of the DN statement to the label at the left.

*Flags affected refers to whether or not the instruction has any effect on the flags in the status register. The C column stands for the Carry flag (did the operation result in a carry being generated?), and the Z stands for the Zero flag (did the operation result in a zero?). A Y appears in the column if the flag is affected by the instruction. An N indicates the flag is not changed by the instruction.

After displaying the program, NanoAssembler prompts you to save the object file. The saved file is identical in format to the ones you loaded and saved with the NanoProcessor last issue; that is, the file contains the contents of all addresses from 0 through 255. To see that your program works properly, load and RUN the NanoProcessor. You can then load and run the program you've just created according to the instructions detailed in Vol. 5, No. 5.

For a short program such as Sample Program 1, this process may seem a bit time consuming. For longer and more complex programs, however, the ease of writing and debugging provided by an assembler more than makes up for the added steps.

Assembler Directives

Figure 1 displays the 16 instructions that we detailed in the NanoProcessor. You may specify any of these instructions when writing an assembly-language program with the NanoEditor. The NanoAssembler, in turn, converts these instructions into their machine codes. There are three additional commands, known as assembler directives, that the Assembler understands:

Directive	Purpose
ORG	Start object code here
DN	Define a nibble
EQU	Define a label

The ORG command directs the NanoAssembler to assemble the program at a specified address between 0 and 255. For an example of this instruction, see line 1 of Sample Program 2. This program is a slightly modified version of Sample Program 2 that we presented in last issue's NanoProcessor. It performs a two nibble addition of numbers located at addresses 240 and 241, placing the answer in addresses 248 and 249. The ORG statement makes the starting address %1010.

The DN instruction allows you to include a particular value at any address. Just specify the address using the ORG directive, and then define the value to be placed at that address with the DN directive. Lines 22 through 24 of Sample Program 2 define the two nibbles that the program adds.

Figure 2

Decimal	Binary	Hexadecimal
0	%0000	\$0
1	%0001	\$1
2	%0010	\$2
3	%0011	\$3
4	%0100	\$4
5	%0101	\$5
6	%0110	\$6
7	%0111	\$7
8	%1000	\$8
9	%1001	\$9
10	%1010	\$A
11	%1011	\$B
12	%1100	\$C
13	%1101	\$D
14	%1110	\$E
15	%1111	\$F

Sample Program 1

Addr	Code	Line	Label	Mnemonic	Remark
0	%0001	001		LDA# 3	:Get first number
1	%0011				
2	%0100	002		TAB	:Move to B
3	%0001	003		LDA# 7	:Get second number
4	%0111				
5	%0000	004		ADD	:Figure sum
6	%1111	005	HERE	JMP HERE	:Jump self to stop
7	%0110				
8	%0000				

The EQU command lets you identify any address with a particular label. Lines 2 through 6 of Sample Program 2 use this directive. These statements make Sample Program 2 more readable by assigning descriptive labels to the 5 data addresses: NIB1 and NIB2 for the two numbers to be added; LONIB and HINIB for the low and high nibbles of the answer; and OUT for the OUT light. (See last issue's *NanoProcessor* for a complete explanation of how Program 2 uses these 4 locations.)

The other change to Program 2 in this issue is the use of the OUT light located at the upper-left of the *NanoProcessor* screen. When you assemble Sample Program 2 and run it, you will find that the OUT light is off when the program begins, but it turns on when the program is complete. Thus, you do not need to know what address the program will end on. Instead, the OUT light signals that the program is finished.

Sample Program 3 accesses the *NanoProcessor*'s "sound chip." Any time you store a number at either location 254 or 255, the *NanoProcessor* responds with a tone. With 16 different values possible at each of these locations, you can make a total of 32 different tones. Sample Program 3 plays a C scale.

We hope that you have found these *Nano* programs instructive and enjoyable. With what you have learned, you should be able to create your own "machine-language" routines. Feel free to let us know in "Letters to the Editor" of any programs you create, so we may share them with our readers.

HCM Glossary Terms: assembler, label, object code, op-code, operand, pass, source code.

For your type-in listings, see HCM PROGRAM LISTING CONTENTS.

HCM

Three Number Systems Supported

Machine language on the *NanoProcessor* can be entered only in binary. The *NanoAssembler*, however, understands decimal and hexadecimal in addition to binary. Last issue we explained how to convert between decimal and binary—this issue we introduce you to hexadecimal.

As we explained in the previous issue, decimal is a base 10 system. It uses ten digits (0 through 9) to represent numbers. Similarly, binary is a base 2 system and uses two digits (0 and 1). Hexadecimal is a base 16 number system and uses 16 different digits—0 through 9 plus the letters A through F. [See Figure 2 for a conversion chart.] As the conversion chart shows, we can express the number 11 decimal as either the binary number %1010 or the hexadecimal number 8B. [Note that the % symbol denotes a binary number, and the \$ symbol a hexadecimal number.]

To convert a two-digit hexadecimal number (say \$C8) to decimal you simply find the decimal equivalent of the left-most digit (i.e., \$C = 12), and multiply it by 16. Then simply add the decimal equivalent of the right-most digit (12 * 16 + 8 = 200). Hexadecimal is a particularly useful system in assembly language because it can express any nibble as a single character or any byte as two characters.

Sample Program 2

Addr	Code	Line	Label	Mnemonic	Remark
		001		ORG 10	
		002	NIB1	EQU \$F0	
		003	NIB2	EQU \$F1	
		004	LONIB	EQU \$F8	
		005	HINIB	EQU \$F9	
		006	OUT	EQU \$FD	
		007		LDA# 0	:Turn OUT light off
10	%0001				
11	%0000				
12	%0011				
13	%1101				
14	%1111				
15	%0010	009		LDA NIB1	:Get first number
16	%0000				
17	%1111				
18	%0100	010		TAB	:Move to B
19	%0010	011		LDA NIB2	:Get second number
20	%0001				
21	%1111				
22	%0000	012		ADD	:Figure sum
23	%0011	013		STA LONIB	:Low to memory
24	%1000				
25	%1111				
26	%1110	014		BCC NIB	:One nibble answer
27	%0010				
28	%0010				
29	%0001	015		LDA# 1	
30	%0001				
31	%1111	016		JMP \$TH	:All done
32	%0100				
33	%0010				
34	%0001	017	NIB	LDA# 0	:Zero A
35	%0000				
36	%0011	018	STH	STA HINIB	:High to memory
37	%1001				
38	%1111				
39	%0001	019		LDA# ON	:Set OUT light
40	%0001				
41	%0011	020		STA OUT	
42	%1101				
43	%1111				
44	%1111	021	HERE	JMP HERE	:Jump self to end
45	%1100				
46	%0010				
		022		ORG \$FO	
		023		DN \$A	
		024		DN \$C	

Sample Program 3

Addr	Code	Line	Label	Mnemonic	Remark
		001		EQU 254	
		002	SOUND	LDA# 2	
0	%0001				
1	%0010				
2	%0100	003		TAB	
3	%1000	004		AND	
4	%0110	005		RRC	
5	%0011	006		STA SOUND	
6	%1110				
7	%1111				
8	%0000	007		ADD	
9	%0011	008		STA SOUND	
10	%1110				
11	%1111				
12	%0000	009		ADD	
13	%0011	010		STA SOUND	
14	%1110				
15	%1111				
16	%0001	011		LDA# 8	
17	%0110				
18	%0011	012		STA SOUND	
19	%1110				
20	%1111				
21	%0000	013		ADD	
22	%0000	014		STA SOUND	
23	%0011				
24	%1111				
25	%0000	015		ADD	
26	%0011	016		STA SOUND	
27	%1110				
28	%1111				
29	%0000	017		ADD	
30	%0011	018		STA SOUND	
31	%1110				
32	%1111				
33	%0001	019		LDA# \$D	
34	%1101				
35	%1101	020		STA SOUND	
36	%1110				
37	%1111				
38	%1111	021	HERE	JMP HERE	
39	%0110				
40	%0010				