The Fast Assembler

Yves Han

Here's a truly amazing machine language assembler for the 64 and 128 (in 64 mode). "Fast Assembler" supports multiple statement lines, labels, and macro-like "include" files. It can assemble to memory or to disk. Written very compactly, it occupies only about 2600 bytes, leaving the rest of memory for your source code. It also adds to the BASIC editor several new features useful to both BASIC and machine language programmers.

Symbolic label-based assemblers are the most convenient way to write machine language (ML) programs. The instructions are entered as *source code* and later assembled into object code (the actual ML program—the numbers in memory). And rather than using memory locations, you can name routines with meaningful labels. It's as if you could enter GOSUB JOYSTICK in BASIC.

Saving Memory By Using The BASIC Editor

You write your ML programs for "The Fast Assembler" (FA) with the 64's BASIC editor. You save to tape or disk as you would a BASIC program, and listing it to a printer is exactly the same as listing BASIC.

The FA is an extension of the BASIC interpreter especially designed for writing programs in machine language. Writing it as a BASIC extension kept the program short (under 2600 bytes) because many subroutines of the BASIC interpreter could be used. Some modifications have been made to BASIC to make writing programs easier. To do this, the BASIC ROM had to be copied to its matching RAM.

Even if you don't write programs in machine language, you can still use the assembler because of the new features added to BASIC and the extra BASIC commands. The assembler will execute a BASIC program just like normal BASIC would.

To start up FA, first load it as if it were a BASIC program (don't use a secondary address of 1, just type LOAD "Fast Assembler",8. Then type RUN. The enabling SYS is built into the first line of the program. The screen will clear, and a message will appear at the top of the screen, indicating FA has been enabled. You can now start programming—in BASIC or machine language.

BASIC Modifications And Enhancements

The following changes have been made to the BASIC interpreter:

• **Structured listings.** Spaces between the line number and the first character on the line are not deleted. This makes it possible to indent lines and make listings easier to read.

• **List pause.** You can freeze a listing by holding down the SHIFT key or pressing SHIFT-LOCK. Listing can be continued by releasing the SHIFT key.

• **ASCII translations and hexadecimal/binary numbers.** In arithmetic expressions, you can use hexadecimal and binary numbers. Hexadecimal numbers should be preceded by "$" and binary numbers by "%". You can also use a character preceded by a single quote ('A is the same as ASC("A")). You can also use this to find the value of a BASIC token. For example, PRINT 'END will print the value 128, which is the BASIC code for END. If you put a space between the quote and the character, the ASCII value of the space will be taken instead of the character.

• **Variable and function names.** The rules for variable and function names have been changed a little bit. Instead of the first two, the first eight characters are recognized. FA recognizes NUMBER1 and NUMBER2 as separate variables, while ordinary BASIC would consider them the same variable (NU). Variables may contain but not be equal to BASIC/assembler commands or mnemonics: LAND is a legitimate variable name, even though it contains the keyword AND. But variable labels starting with TI or ST (reserved keywords) are not automatically set to zero the first time you use them. An exception to the eight character names is that only the first two characters of

array variables are significant.

• **Keywords.** Because variable and function names may contain keywords, FA has to be able to decide whether a keyword is a keyword or part of a variable or function name. So the assembler recognizes a keyword if it's followed by a space or nonalphabetic character. For example, in PRINT "OK" the keyword PRINT will be recognized as a PRINT command, but in A$="OK":PRINTA$, the keyword PRINT is recognized as part of the variable name PRINTA$. You would have to insert a space (PRINT A$) if you wanted to print the variable A$.

• **REM and DATA.** Capital letters in REM and DATA lines are listed as capital letters and not as tokenized BASIC keywords. For example, **10 rem AB** lists as it is entered and not as **10 rem atnpeek** as normal BASIC would do.

New BASIC Commands

AUTO ***step value***

This command turns automatic line numbering on and defines the step value between the line numbers. To enter AUTO mode, type AUTO followed by the step value and press RETURN. Then enter a line with a line number. The next line number prints automatically. To leave auto mode, move the cursor to an empty line and press RETURN. To turn automatic line numbering off altogether, enter AUTO only.

You can also use this command to delete part of a program. Turn automatic line numbering on with a step value of one. Type the number of the first line you want to delete and press RETURN. Keep pressing RETURN until you've reached the end of the section you want to delete. Instead of pressing RETURN again and again, you can enter **POKE 650,128** and hold RETURN down until you've reached the last line to be erased.

OLD

If you accidentally type NEW, you can restore your program with this command. It can also be used if you've installed a reset button. If you've assembled a program and are testing it, sometimes your computer locks up. Use the reset button and then enter SYS 4408 to restart the assembler and type OLD to restore the source program. If your program has not destroyed the assembler or the source program, everything will be there.

Semicolon (;)

This has the same function as the REM statement. It need not be separated with a colon from the preceding command. For example:

```
10 X=0:REM SET X TO ZERO
```

is the same as

```
10 X=0;SET X TO ZERO
```

The semicolon in the commands PRINT and INPUT is not treated as a REM statement but as a separator.

Using Labels As Variables And Addresses

Label names follow the same rules as variable names. They can be used in arithmetic expressions like normal variables. You can define a label in two ways:

You can place the label name just before the command to which you want to refer. If more commands are on the same line, you must separate the label from the commands with a colon.

Or you can label the current program counter: **LABELNAME=***. The asterisk (*) is a special variable which gives the value of the program counter. The counter is the address where the next instruction or datum will be placed. You can only read the variable *. You cannot assign a value to it with the statement *=expr.

Here's an example of using labels to mark routines in a program (don't type this in, it's only a fragment of a program):

```
50 JSR DISPLAY1; JUMP TO LABELED
   SUBROUTINE (LINE 90)
60 LDA $FF: BNE SKIPIT ; CONDI-
   TIONAL BRANCH AHEAD TO
   SKIPIT
70 TYA
80 SKIPIT: LDX #4: STA $8000,X: RTS;
   TARGET OF BRANCH IN 60
90 DISPLAY1=* ; THIS LABELS THE
   CURRENT PROGRAM COUNTER
100 ;
110 LDA #65: JSR $FFD2: RTS
```

Remember that in the lines above, the semicolon marks the beginning of a comment which, like a REM, is ignored by FA. The technique in line 90 is valuable if you think you may be adding some code at the beginning of the routine. As listed, the subroutine called DISPLAY1 starts with LDA #65, but later you could go in and add some lines between 90 and 110.

Three Passes To Assemble

Three passes are required to assemble source code (what you write) into object code (an executable ML program that the computer can follow). But FA doesn't do it by itself. You have to insert a loop that repeats three times with BASIC commands:

```
10 FOR PASS=1 TO 3
   .
   . (Insert source code)
   .
90 NEXT PASS:END
```

If you use an invalid addressing mode such as LSR (expr),y you'll see ILLEGAL ADDRESSING-MODE ERROR. Mnemonics can only be used in program mode—that is, in a program you execute with RUN. If you enter a mnemonic in direct mode, you'll see ILLEGAL DIRECT ERROR.

Also note that for Immediate Addressing, the argument can be an actual number or an arithmetic expression with a value in the range 0–255. Or you can substitute a string expression, in which case the assembler takes the ASCII value of the first character as the argument. If the string length is zero, the argument becomes zero.

Assembler Commands

Assembler commands which write data to the output device can only be used in program mode, otherwise you'll get ILLEGAL DIRECT ERROR. All assembler commands must be included in every pass.

ORG ***address,mode,device,name***

This command must be used at the start of each pass. It does several things. First, it sets the origin (ORG), the memory address for the beginning of the ML program. It assigns an initial value to the program counter. It also sets the assembler mode, which should be zero on the first two passes and one on the third and last. ORG also sets the output device and filename (if necessary).

Not all arguments are necessary. Also permitted are:

ORG
ORG address
ORG address,mode

Default values for the arguments are:

address = 49152 (=$C000)
mode = 0
device = 0 and no name

If you use a mnemonic or assembler command before you've used the command ORG, you'll see UNDEF'D LOCATION COUNTER ERROR.

The address assigns a value to the program counter. Usually, you use more than one pass to assemble the source program. Only during the last pass should the object code be written to memory or to the output device. Mode tells the assembler when the last pass is reached. Zero means it's not the last pass, so no object code should be produced, and there's no range checking for arguments and no checking for too large branches.

On the final pass, you should set the mode to one, which signals the last pass, when object code is written to the output device.

Finally, you set the device number of the output device and a string expression which contains the filename if the object code is not written to memory. Zero means the output device is memory. Be careful not to write to memory locations where the assembler is placed ($0801–$121B) or where the BASIC interpreter is placed ($A000–$BFFF).

A device number in the range 8–11 means the output device is a disk drive. If mode is equal to one, the assembler will open a PRG file with the name specified in the argument name. The logical file number will be eight.

BYTE *expression,expression,...*
This command writes numbers or characters to memory or the selected output device. It can have one or more arithmetic or string expressions separated by commas. Arithmetic expressions must give a positive value less than 256. The value will be placed in one byte. Each character of a string expression will be placed in one byte.

WORD *expression,expression,...*
This has the same function as BYTE except that values of arithmetic expressions must be positive and less than 65536. The value will be placed in two bytes in low/high format.

INCLUDE *name,device*
This command assembles a file from disk and inserts the resulting object code into memory or the output device. The file must be a normal PRG file and may not contain BASIC commands which cause a branch to another line or stop the program. Also not permitted are the BASIC commands DEF, RETURN, CLR, NEW, and the assembler commands SEND and INCLUDE.

The file is opened with a logical file number of nine. The file is closed when the end of the file is reached. The name is the filename you're including, and the device number can be 8–11 (use 8 if you have a single drive). If you have only one disk drive and you assemble to disk, the file(s) for the command INCLUDE must be on the same disk to which you assemble.

All variables and labels are *global*, which means you can pass parameters to INCLUDE files so they can work like macro-instructions. Let's say you're writing a program that needs to access several different disk files, and there are several points in the program that use the Kernal routines SETLFS, SETNAM, and OPEN. You could write the source code that performs these Kernal calls and save it to disk under the program name "OPEN" to be used later. Then, in the main program, use INCLUDE "OPEN" ,8. When the source code is compiled, the series of commands from the OPEN file are automatically inserted in the proper place in the object code.

SEND *stringexpr*
The command SEND may be used only if the object program is written to disk. It's used to link object code to a BASIC program. *Stringexpr* must contain a BASIC line with line number. If you forget the line number, you'll get MISSING LINE NUMBER ERROR. If you want to send more than one line, you must use SEND for each line, and you have to send the lines in the right order. You must send the lines before the actual object code is written to disk. The address in the ORG command must be the start of BASIC RAM (2049).

UNSEND
If you load a program which consists of both BASIC and ML, the interpreter has to know where the BASIC part ends. UNSEND places a mark which the computer recognizes as the end of the BASIC part.

Example Programs

```
100 FOR PASS=1 TO 3:PRINT
    "PASS"PASS
110 ORG $C000
120 IF PAS=3 THEN OFG $C000,1
130 START: LDX #0
140 LOOP: LDA TEXT,X:PRINT TEXT,
150 BEQ EXIT
160 JSR $FFD2
170 INX
180 BNE LOOP
190 EXIT: RTS
200 PRINT *
210 TEXT: BYTE "EXAMPLE 1",0
220 NEXT PASS:END
```

Lines 110 and 120 show how to use the command ORG. In every pass, line 110 sets mode 0. But in pass three, line 120 sets mode 1. The object code will start at 49152 (hexadecimal $C000). Line 200 prints the current value of the location counter (*).

You can assemble the program with the command RUN. The program will give the following output:

```
PASS 1    0        49165
PASS 2    49165    49166
PASS 3    49166    49166
```

The first column is the pass number. The second column is the value of the label TEXT in the instruction LDA TEXT,X in line 140. The third column is the value the label should have when the source code is assembled. You can see that only in pass three are these values equal to each other. This is because the assembler defaults to zero-page addressing. In pass one, TEXT has a value less than 256 so zero-page addressing is assumed. This means a two-byte instruction instead of three. The value assigned to TEXT will be too low, as you can see in pass one. In pass two, this value, which is too low, will be used in assembling line 140. The assembler decides not to use zero-page addressing, so TEXT is assigned the correct value. In pass three, the cor-

rect value replaces the previously incorrect values during assembly.

```
5 ; EXAMPLE PROGRAM 2
6 ;
10 PRINT CHR$(147)
11 DEF FN H(X)=INT(X/256)
12 DEF FN L(X)=X-256*FN H(X)
20 PRINT:PRINT" Loader maker"
30 PRINT:PRINT" Enter the name of the
   program that"
40 PRINT" has to be loaded by the
   loader."
50 INPUT" >";NAME$
60 PRINT:PRINT" Enter the name of the
   loader."
70 INPUT" >";N$
80 PRINT:PRINT" Enter the address to
   execute the"
90 PRINT" program."
100 INPUT" >";ADDRESS:ADDRESS=
    ADDRESS-1
105 ;
110 FOR PASS=1 TO 3
115 ;
120 ORG 2049
130 IF PASS=3 THEN ORG 2049,1,8,N$
135 ;
140 SEND "10 SYS+STR$(LOADER)
150 UNSEND
155 ;
160  LOADER:  LDA #8:TAX:LDY #1
170            JSR $FFBA
180            LDX #FN L(NAME)
190            LDY #FN H(NAME)
200            LDA #LEN(NAME$)
210            JSR $FFBD
220            LDA #FN H(ADDRESS):
               PHA
230            LDA #FN L(ADDRESS):
               PHA
240            LDA #0:JMP $55D5
250  NAME:     BYTE NAME$
255  ;
260  NEXT PASS:CLOSE 8:END
```

The above example program shows how to use the commands SEND and UNSEND to write a program that includes a SYS within a BASIC line.

The main routine at 160–250 illustrates how to load another program from an ML program. Note that the lines up to 100 are BASIC; they prepare the variables and defined functions for use in the source code. If you assemble the program with the command RUN, you'll get a program that can load another ML program from disk and execute it. The object code will be written to disk.

In line 140, the command SEND writes a BASIC line to the output device by which you can load and run the program as if it were a normal BASIC program. Line 150 marks the end of the BASIC part of the object code.

The INPUTs in lines 50, 70, and 100 permit you to enter the parameters for the object program when the source program is assembled. In this way you can make different object programs with one source program.

Another advantage of writing the assembler as a BASIC extension is that you can assemble a program to the top of memory. Use the following construction to do this:

```
100 POKE 56,PEEK(56)-4:CLR
110 TOPOFMEM=PEEK(55)+256*(PEEK
    (56)+4)
120 ADDRESS=0:MODE=0
130 FOR PASS=1 TO 3
140 ORG ADDRESS
150 IF PASS=3 THEN ORG ADDRESS,
    MODE
      .
      .
    . Source code
      .
      .
900 NEXT PASS
910 IF MODE=1 THEN END
920 ADDRESS=TOPOFMEM-*
930 MODE=1:GOTO 130
```

In this program, the source code goes through six passes. During the first three passes the location counter remains at zero. Mode 0 is used so the object program will not be written to the output device. The length of the program is calculated and subtracted from TOPOFMEM. This address is used in the second three passes to assemble to the top of memory. MODE is set to one so the assembler will write the object code to the output device during the sixth pass (actually pass three of the second time around). Line 100 is used to reserve 1K at the top of memory for the object program.

Large Programs

If your source program won't fit into memory, you can split your program and use the command INCLUDE. For example:

```
10 FOR PASS=1 TO 3
20 ORG ADDRESS
30 IF PASS=3 THEN ORG ADDRESS,1
   .
   . Part 1 of source code
   .
90 INCLUDE "PART 2",8
100 INCLUDE "PART 3",8
110 NEXT PASS:END
```

The labels and variables used in the INCLUDE files will be global variables, which means you can use them in arithmetic expressions everywhere in the program.

Another possibility is chaining the programs, but then you can't use a FOR-NEXT loop for the passes. You must use another way to define the passes. For example:

```
FIRSTPROGRAM
10 PASS=PASS+1:IF PASS=4 THEN
   END
20 ORG ADDRESS
30 IF PASS=3 THEN ORG ADDRESS,1
   .
   . Source code
   .
90 LOAD"SECONDPROGRAM",8
SECONDPROGRAM
   .
   . Source code
   .
90 LOAD"FIRSTPROGRAM",8
```

Note that these are just examples. You'd have to insert your own source code as indicated. To chain programs, you would load and execute the first program. It controls the number of passes and loads the next program. The next program loads the following program and so on until the last program, which must load the first again. ©