Volume II Issue 3
MARCH 1984

MEETING NOTICE

The next meeting will be Monday, March 19, at 7:30. The meeting location is Alpha Audio's third floor conference room, at 2049 West Broad Street. The night-time phone number there is 358-3853. The front door has a touch-pad combination lock, and the combination for the night will be 6408 (six four zero eight).

Everyone is welcome!

MINUTES (Meeting of February 20, 1984)

The minutes were not available at press time, and will be presented at the next meeting.

THOUGHTS WHILE HACKING - Ed.

In my recent travails with my new job, I have had the opportunity to use two software programs which have limited application, but are worth mentioning. One of my tasks has been to transfer a COBOL program written on an IBM-XT to a Dual-Density Osborne.
The first program which came to my rescue is FILETRAN, from Business Micro-Products. This program enables the user to accept files on one machine from each of a group of other machines. Of particular interest is that Heath DD disks are supported. I think the documentation is sufficiently well written, but it does presume a certain level of computer experience. I had difficulty in the transfer of files in that the disks I was copying were formatted by the format program of the IBM-XT, which defaults a soft 9-sector format and the Osborne was expecting only eight sectors from the IBM-PC. This did cause some headscratching for a while. But, when all else fails, read the instructions (not theirs, IBM's). I sent the vendor a note commenting on this. The program successfully moved the files to the Osborne with little other trouble and with very good instructions presented on the screens. Note that I am not saying that it will transfer a compiled program that you can then run. The programs I transferred were a compiled data file and ASCII source programs, and an object file without a run time module.

Well, you may ask, why don't you know if they will run? You did ask, didn't you? The programs I had to transfer were COBOL source and object files. Have you ever tried to find a COBOL program for an Osborne? No! of course not.

I talked with two members of the local Osborne Users group, John Barry and Erik Schultz. John gave me a copy of a public domain COBOL program, which was altogether inadequate. Erik gave me all sorts of help including suggestions of several other languages in which to write the programs I need. Both referred me to NEVADA COBOL which is the subject of this discussion. Incidentally, both of these gentlemen are computer consultants, and Erik especially was a fountain of good information.

For $40.00, no language can be all bad. For those of you who know COBOL, it is a less enhanced version of Microsoft COBOL, available from Heath. Its vocabulary is significantly less than that of M-COBOL. To give an example of some of the differences, in COBOL, its source text is expected to have 6 columns of numbers beginning each line. N-COBOL expects only 4! If the 5th column is open, it will take the line as executable, and it expects to see certain things in column 6. If there is nothing there, it will look at column 10. Since I had copied over a file written with six numbered columns assumed, there was a certain amount of confusion on the part of the compiler. N-COBOL will not accept the single quote mark ('). Nor will it accept a WRITE...AFTER command. But it does accept a WRITE...BEFORE command and double quotes ("'). Since the programmer who had written the program I needed to transfer wrote in IBM-COBOL (Also Microsoft COBOL) used both idioms exclusively, there was a lot of trouble getting a good compile.

I mention these troubles because they are symptomatic of the weaknesses of N-COBOL. It is a primitive version of COBOL based upon some of the ANSI-74 standards. It is very difficult to step down (as it were) from something like Microsoft-COBOL to Nevada COBOL.

But here is some good news. The manual is very well written and easy to follow. I would consider N-COBOL an inexpensive primer for COBOL. I compare using it and then M-COBOL to using ZENCALC and then LOTUS 1-2-3. ZENCALC is an excellent program, but wait till you add the bells and whistles! Microsoft COBOL has the bells and whistles, compared to NEVADA COBOL.
If you want to try an inexpensive way to pick up the fundamentals of COBOL, I think NEVADA COBOL is a way to go. As I say, for $40, no language is all bad. NEVADA Cobol is sold by Ellis Computing, of San Francisco.

Ellis also sells NEVADA FORTRAN, NEVADA EDIT, and NEVADA PILOT for $40 each.

I also just bought SUPERCALC-3 from SORCIM Corp. I will have some words about that after I have had a chance to work with it. So far, it appears to be able to do everything that LOTUS 1-2-3 does, but cheaper.

---

ASSEMBLY LANGUAGE PROGRAMMING - PART 8
by Jim Scott

INTRODUCTION

This is the eighth of a series of articles which parallel and summarize the discussions about assembly language at our meetings. The purpose of the discussions and the articles is to present enough information about assembly language programming so that someone who knows how to program in a higher-level language, and is willing to use the proper manuals for reference, will at least have some idea how to get started at programming in assembly language.

This time we will discuss assembler directive instructions, and a few other aspects of assemblers. But first let's review what has happened so far.

In part 1 we learned that, before writing an assembler language program, you have to know three things:

1. What processor (CPU) will you be programming for? These articles cover the 8080 (8080A) CPU.
2. What operating system will your program run under? We are dealing with CP/M and HDOS.
3. Which assembler will you use? Here we consider the assemblers that come free with CP/M and HDOS (both named ASM).

Part 2 described the logical structure of the 8080 CPU: its memory and its registers. It also reviewed some example instructions in the sample assembler listings included in the first article.

Parts 3 through 7 described in detail the five groups of machine language instructions in the 8080:

3. Data Transfer Group
4. Arithmetic Group
5. Logical Group
6. Branch Group
7. Stack, I/O, and Machine Control Group

In summary, the first article presented three questions, and the next six answered the first question in great detail. In other words, now that we know the memory structure, registers, and the machine language instructions, we know just about everything about the 8080 that an assembler language programmer needs to know.
Now we jump to the third question. The assembler directives, unlike the machine operations, are specific not to the CPU, but to the assembler being used to translate the source program into an executable program.

**THE ASSEMBLER DIRECTIVES**

The machine operations described in parts 3-7 are translated by the assembler from their source form into machine language. Each machine operation in the source program will become a one- to three-byte machine language instruction in the executable program (COM or ABS file), also known as the object program. (The terms "machine operation" and "machine language instruction" are similar, and are often used interchangeably. I am attempting to use them in a precise way to distinguish two forms of the same thing. In a source program, or assembler language program, a machine operation is represented as having a mnemonic operation code, or opcode, like LXI or MOV, as well as operands with symbolic names like A or TOTAL or CTR3. In an object program, which is the end result of running the assembler (plus the LOAD program, in the case of CP/M), a machine language instruction is simply one to three bytes, usually represented for semi-human-consumption in hex or octal, such as 3A F1 27 or 066 012.)

But an assembler language source program is a mixture of machine operations and assembler directives. What does the assembler do with an assembler directive?

An assembler directive (sometimes also known as a pseudo operation) is a source instruction that does not get translated into a machine language instruction. It is a way of telling the assembler something about how we want it to do its job. One way of looking at it is this - an assembler directive is executed at assembly time (when the assembler is actually being run to translate the source program into the executable program), whereas a machine operation will not be executed until object time (when the resulting COM or ABS file, or object program, is run, after the process of assembling is finished).

Since the two ASM assemblers (one for CP/M and one for HDOS) are two different programs, they do not have exactly the same set of assembler directives. This article will not cover all of the assembler directives, but will emphasize the ones that are most frequently used, and that are common to, and similar in, the two ASM assemblers. A complete listing and description of assembler directives can be found in your CP/M or HDOS manual, or the manual for whatever assembler you plan to use.

The individual instructions are as follows. Remember that any instruction can have a label, or name, before the mnemonic operation code. If it does, the label can be used elsewhere in the program as referring to the address of that instruction. The possibility of an instruction having a label will only be mentioned below when it is particularly significant.

**ORG (Origin)**

Format: ORG addr

where addr is an address of a memory location (or an expression which is the equivalent of a memory address; "expression" will be defined more precisely later). The ORG statement tells the assembler where in memory the object program will be located at object (execution) time.
Example: ORG 100H
This tells the assembler that the following machine operation will be
located at memory address 100 (hex) at object time, and subsequent
machine operations will be located at successive memory addresses.

This instruction must precede the first machine operation in the
source program. Occasionally there will be a reason for having more
than one ORG statement in a program; in this case, the second ORG
gives the starting address for the machine operations that follow it,
and this may be in an entirely different part of memory from where the
code between the first and second ORGs is to be located.

Ordinarily, a program written to run under CP/M will ORG at 100H,
and a program written to run under HDOS will ORG at 042200A (octal).

END (End of Program)

Format: END addr
where addr is an address of a memory location. Addr may be optional,
depending on the assembler. The END statement marks the end of the
source program, and, if addr is provided, tells it the address which
is to be considered the entry point of the program. The entry point
is the address of the machine instruction which is to be the first one
executed at object time.

Example: END 100H
This tells the assembler that the physical end of the source program
has been reached, and that execution is to begin at address 100H. A
typical program might begin with

START ORG 100H (or 042200A)

and end with

END START

The END instruction must be the last in the source program.
Anything following it will be ignored by the assembler. Since nuances
of assemblers change from one version to another, as well as between
operating systems, check your assembler manual to see if END is
required, and whether addr is optional.

EQU (Equate)

Format: EQU expr
where expr is an integer expression. Expr is often a memory address,
but it need not be. The EQU instruction must have a label, and it
must be a label which is not used on any other instruction in the
program. The EQU instruction simply tells the assembler to treat the
label as the equivalent, or synonym, of the expression expr.

Example: CR EQU 13
This tells the assembler to treat any subsequent reference to CR in
the program as if it were the integer 13. (A byte whose numerical
value is 13 is a carriage return character.) A typical CP/M program
might begin with

BASE EQU 0H ;MEMORY ADDRESS OF BOTTOM OF CP/M.
TPA EQU BASE+100H ;TRANSIENT PROGRAM AREA.
ORG TPA
Thus, BASE represents address 0, and TPA represents address 0+100, or 100 (hex). The origin is at address 100.

The EQU instruction is often used to give an easily-rememberable name to an address or a constant. It can also be used to make a program more easily alterable, in the following way. Suppose your program needs to know how many columns the screen has, and you are writing it for the H19 terminal, which has 80 columns. You could use the constant 80 throughout the program to refer to screen width, but a better way would be to include an equate such as

```
NCOLS EQU 80
```

near the beginning of the program, and then refer to NCOLS instead of 80 in the rest of the program. Ten years now, when the H39 terminal comes out, with 120 columns (is this how rumors get started?), you can update the program simply by changing the EQU and reassembling.

**DB (Define Bytes)**

**Format:** DB expr

where expr is either a character string between apostrophes, or a series of one-byte integer values separated by commas, or a combination of both. The DB instruction tells the assembler to insert the characters or one-byte values into the object program, as data.

**Example:** MSG2 DB 'Enter Y or N',13,10

This causes the assembler to put 14 bytes of data into the object program. The first 12 bytes will be the character string 'Enter Y or N', and the last two bytes will be the one-byte integers 13 and 10 (also known as the carriage return and linefeed characters). The label MSG2 stands for the address of the first byte of this series of data (the 'E').

The DB instruction is unlike the ORG, END, and EQU in that DB actually causes the assembler to generate bytes of the object program, as the machine operations do. The difference is that a machine operation causes the assembler to generate a machine language instruction, while the DB instruction causes the assembler to generate bytes of data.

**DW (Define Words)**

**Format:** DW expr

where expr is either a character string (maximum length of two characters) between apostrophes, or a series of two-byte values separated by commas, or a combination of both. The DW instruction tells the assembler to insert the characters or two-byte values into the object program, as data. DW more or less assumes that each two-byte word is a memory address, so it generates each one to be consistent with the 8080's crazy scheme of storing addresses with the low-order byte first.

**Example:** ADR1 DW 45DAH,'AB'

This causes the assembler to put 4 bytes of data into the object program. The first two bytes will be the hex value 45DA, low-order byte first; the last two bytes will be the character string 'AB', also stored low-order byte first. Thus the result of this DW instruction in hex will be DA 45 42 41. The label ADR1 stands for the address of the first byte of this series of data (the 'DA' byte).
Like DB, the DW instruction causes the assembler to generate data bytes in the object program.

**DS (Define Space)**

Format: DS expr
where expr is an integer, or an expression which is the equivalent of an integer. The DS instruction tells the assembler to reserve space in the object program for expr bytes of data. The assembler reserves the space, but it does not put any particular values into these bytes. It is assumed that when the resulting program runs (object time), the program will put data of some kind into this space.

Example: BUFFER DS 80
This causes the assembler to leave room in the object program for 80 bytes of data. The label BUFFER stands for the address of the first byte of this 80-byte area.

**Other Assembler Directives**

The IF, ENDIF, and SET assembler directives are also common to both the CP/M and the HDOS assemblers, but they are not likely to be used by a beginning programmer. The HDOS assembler (and possible newer versions of the CP/M assembler) also offer the ELSE, XTEXT, TITLE, STL, EJECT, SPACE, LON, LOF, and ERRXX assembler directives. Look these up after you get comfortable with the ones described above.

**OTHER STUFF YOU NEED TO KNOW ABOUT ASSEMBLERS**

**Labels**

A label, you will remember, is a name that appears in the label field of a machine operation or assembler directive. The exact rules for what a label can look like are rather different between CP/M and HDOS, but here is a rule that complies with both compilers: A label may be one to seven characters, of which the first must be alphabetic (A-Z) and the rest may be alphabetic or the digits 0-9.

The value of a label is determined by the instruction where it appears in the label field. In other instructions, it may appear in the operand field.

**Constants**

A constant is either a character string or an integer.

A character string is one or more characters between apostrophes (single quotes). If you want to include an apostrophe character in a string, make it a double apostrophe: e.g., 'Don’t' is a five-character string, consisting of D, o, n, ’, and t.

An integer is a plain old decimal whole number, or its equivalent in hex, octal, or binary. To specify the base of an integer, follow it by one of the following letters:

- **B** (binary, or base 2)
- **D** or **O** (octal, or base 8)
- **A** (split octal) (HDOS only)
- **D** (decimal, or base 10) (assumed if no letter used)
- **H** (hexadecimal, or base 16)
An integer may contain only the digits allowed for that base: 0 or 1 for B; 0-7 for O, Q, or A; 0-9 for D; and 0-9, A-F for H. A hex integer whose first digit is A-F must be preceded by a 0 (zero digit), to keep the assembler from confusing it with a label.

The Instruction Counter Symbol

The special symbol $ (CP/M assembler) or * (HDOS assembler) represents the address of the next instruction to be assembled. This is also known as the current value of the "instruction counter" (defined below). I like to use this to avoid putting labels on machine operations, in case I want to insert more instructions. For example,

CHARIN EQU $ MOV A,M

This has the same effect as if the label CHARIN were in the label field of the MOV instruction. But if I later decide to insert instructions before the MOV instruction, and I want the label CHARIN to apply to the first of the inserted instructions, I just insert them immediately after the EQU $ instruction, without having to move the label from one instruction to another. This may seem like a useless nicety, but it applies often enough to be a significant effort saver.

Expressions

Often the operand of an instruction (machine operation or assembler directive) can be an expression. The specific type of instruction may put limits on what the value of the expression can be for that instruction, but the general rule is that an expression can be any of the following:

- a label
- an integer
- a character string
- the instruction counter symbol
- a combination of any or all of the above, connected by + - * /

The CP/M assembler also allows MOD, NOT, AND, OR, XOR, SHL, and SHR as operators in forming an expression. Look it up if you care.

Here are some examples of expressions using operators:

BASE+100H (the sum of the value of the label BASE, and hex 100)
20/2 (20 divided by 2 is 10)
'Z'+1 (the numeric value of the character just after 'Z' in ASCII sequence)
*+3 (the address of the next instruction to be assembled, plus 3)
SUB*INC+5 (the product of the values of the labels SUB and INC, plus 5)

The CP/M and HDOS assemblers differ in some of the rules for forming expressions. For example, CP/M allows parentheses and HDOS doesn't.

How the Assembler Works

It helps to understand a little about how an assembler goes about its business of translating a source program into an object program. A typical simple assembler would be a two-pass assembler. One of the
few good things about a cassette system is that it illustrates clearly
what is meant by a two-pass assembler. On a cassette system, you
first load the assembler itself from tape. Then you put into the tape
recorder the tape that contains your source program. When the
assembler starts running, it reads the source program from beginning
to end (the end of the file, or the END statement in the program);
then it signals you to rewind the tape to the beginning of the
program; finally, it reads the source program from tape a second time
and produces its results. A disk based assembler can be thought of as
reading the source twice like this, even though it is not as obvious
that it is doing so.

Why two passes? On the first pass, the assembler builds a table
of the exact values of the labels, and the addresses of the
instructions. On the second pass, it completes the assembly by
inserting the addresses into the machine language instructions where
they are needed. Suppose a program contains the instruction JMP STOP
and later has an instruction with STOP in its label field. On the
first pass, the assembler sees JMP STOP and notes that a JMP
instruction is a three-byte instruction, with the first byte being an
opcode and the other two bytes being the address to be jumped to. But
at this point it doesn’t know what address STOP is equivalent to. It
will not know that until it gets down to the instruction with the STOP
label, later in the first pass. On the second pass, it will insert
that address into the machine language JMP instruction.

On the first pass, the assembler keeps track of the address of
each instruction. The first instruction is located at the address
specified by the ORG statement. This address is used as the initial
value of the instruction counter. Each machine operation adds one,
two, or three to the instruction counter. Each DB, DW, or DS adds its
own data length to the instruction counter. (In fact, the only real
purpose of DS is to add a number to the instruction counter.) If an
ORG instruction occurs after the first one, it causes the instruction
counter to be reset to the value specified. The special instruction
counter symbol $ or * represents the current value of the instruction
counter as the assembler reaches the instruction that contains the $
or *.

CONCLUSION

This completes the discussion of the assembler directive
instructions. The next article will continue the discussion of
assembly language programming for the 8080.