IBM Personal Computer Assembly Language Tutorial

Joshua Auerbach
Yale University
Yale Computer Center
175 Whitney Avenue
P. O. Box 2112
New Haven, Connecticut 06520

Installation Code YU

Integrated Personal Computers Project
Communications Group
Communications and Data Base Division

Session C316

This talk is for people who are just getting started with the PC MACRO Assembler. Maybe you are just contemplating doing some coding in assembler, maybe you have tried it with mixed success. If you are here to get aimed in the right direction, to get off to a good start with the assembler, then you have come for the right reason. I can't promise you'll get what you want, but I'll do my best.

On the other hand, if you have already turned out some working assembler code, then this talk is likely to be on the elementary side for you. If you want to review a few basics and have no where else pressing to go, then by all means stay.

Why Learn Assembler?

The reasons for LEARNING assembler are not the same as the reasons for USING it in a particular application. But, we have to start with some of the reasons for using it and then I think the reasons for learning it will become clear.

First, let's dispose of a bad reason for using it. Don't use it just because you think it is going to execute faster. A particular sequence of ordinary bread-and-butter computations written in PASCAL, C, FORTRAN, or compiled BASIC can do the job just about as fast as the same algorithm coded in assembler. Of course, interpretive BASIC is slower, but if you have a BASIC application which runs too slow you probably want to try compiling it before you think too much about translating parts of it to another language.

On the other hand, high level languages do tend to isolate you from the machine. That is both their strength and their weakness. Usually, when implemented on a micro, a high level language provides an escape mechanism to the underlying operating system or to the bare machine. So, for example, BASIC has its PEEK and POKE. But, the route to the bare machine is often a circuitous one, leading to tricky programming which is hard to follow.

For those of us working on PC's connected to SHARE-class mainframes, we are generally concerned with three interfaces: the keyboard, the screen, and the communication line or lines. All three of these entities raise machine dependent issues which are imperfectly addressed by the underlying operating system or by high level languages.

Sometimes, the system or the language does too little for you. For example, with the asynch adapter, the system provides no interrupt handler, no buffer, and no flow control. The application is stuck with the responsibility for monitoring that port and not missing any characters, then deciding what to do with all errors. BASIC does a reasonable job on some of this, but that is only BASIC. Most other languages do less.

Sometimes, the system may do too much for you. System support for the key-board is an example. At the hardware level, all 83 keys on the keyboard send unique codes when they are pressed, held down, and released. But, someone has decided that certain keys, like Num Lock and Scroll Lock are going to do certain things before the application even sees them and can't therefore be used as ordinary keys.

Sometimes, the system does about the right amount of stuff but does it less efficiently then it should. System support for the screen is in this class. If you use only the official interface to the screen you sometimes slow your application down unacceptably. I said before, don't use assembler just to speed things up, but there I was talking about mainline code, which generally can't be speeded up much by assembler coding. A critical system interface is a different matter: sometimes we may have to use assembler to bypass a hopelessly inefficient implementation. We don't want to do this if we can avoid it, but sometimes we can't.

Assembly language code can overcome these deficiencies. In some cases, you can also overcome these deficiencies by judicious use of the escape valves which your high level language provides. In BASIC, you can PEEK and POKE and INP and OUT your way around a great many issues. In many other languages you can issue system calls and interrupts and usually manage, one way or other, to modify system memory. Writing handlers to take real-time hardware interrupts from the keyboard or asynch port, though, is still going to be a problem in most languages. Some languages claim to let you do it but I have yet to see an acceptably clean implementation done that way.

The real reason while assembler is better than "tricky POKEs" for writing machine-dependent code, though, is the same reason why PASCAL is better than assembler for writing a payroll package: it is easier to maintain.

Let the high level language do what it does best, but recognize that there are some things which are best done in assembler code. The assembler, unlike the tricky POKE, can make judicious use of equates, macros, labels, and appropriately placed comments to show what is really going on in this machine-dependent realm where it thrives.

So, there are times when it becomes appropriate to write in assembler; given that, if you are a responsible programmer or manager, you will want to be "assembler-literate" so you can decide when assembler code should be written.

What do I mean by "assembler-literate?" I don't just mean understanding the 8086 architecture; I think, even if you don't write much assembler code yourself, you ought to understand the actual process of turning out assembler code and the various ways to incorporate it into an application. You ought to be able to tell good assembler code from bad, and appropriate assembler code from inappropriate.

Steps to becoming ASSEMBLER LITERATE

1. Learn the 8086 architecture and most of the instruction set. Learn what you need to know and ignore what you don't. Reading: The 8086 Primer by Stephen Morse, published by Hayden. You need to read only two chapters, the one on machine organization and the one on the instruction set.

2. Learn about a few simple DOS function calls. Know what services the operating system provides. If appropriate, learn a little about other systems too. It will aid portability later on. Reading: appendices D and E of the PC DOS manual.

3. Learn enough about the MACRO assembler and the LINKer to write some simple things that really work. Here, too, the main thing is figuring out what you don't need to know. Whatever you do, don't study the sample programs distributed with the assembler unless you have nothing better!

4. At the same time as you are learning the assembler itself, you will need to learn a few tools and concepts to properly combine your assembler code with the other things you do. If you plan to call assembler subroutines from a high level language, you will need to study the interface notes provided in your language manual. Usually, this forms an appendix of some sort. If you plan to package your assembler routines as .COM programs you will need to learn to do this. You should also learn to use DEBUG.

5. Read the Technical Reference, but very selectively. The most important things to know are the header comments in the BIOS listing. Next, you will want to learn about the RS 232 port and maybe about the video adapters.

Notice that the key thing in all five phases is being selective. It is easy to conclude that there is too much to learn unless you can throw away what you don't need. Most of the rest of this talk is going to deal with this very important question of what you need and don't need to learn in each phase. In some cases, I will have to leave you to do almost all of the learning, in others, I will teach a few salient points, enough, I hope, to get you started. I hope you understand that all I can do in an hour is get you started on the way.

Phase 1: Learn the architecture and instruction set.

The Morse book might seem like a lot of book to buy for just two really important chapters; other books devote a lot more space to the instruction set and give you a big beautiful reference page on each instruction. And, some of the other things in the Morse book, although interesting, really aren't very vital and are covered too sketchily to be of any real help. The reason I like the Morse book is that you can just read it; it has a very conversational style, it is very lucid, it tells you what you really need to know, and a little bit more which is by way of background; because nothing really gets belabored to much, you can gracefully forget the things you don't use. And, I very much recommend READING Morse rather than studying it. Get the big picture at this point.

Now, you want to concentrate on those things which are worth fixing in memory. After you read Morse, you should relate what you have learned to this outline.

1. You want to fix in your mind the idea of the four segment registers CODE, DATA, STACK, and EXTRA. This part is pretty easy to grasp. The 8086 and the 8088 use 20 bit addresses for memory, meaning that they can address up to 1 megabyte of memory. But, the registers and the address fields in all the instructions are no more that 16 bits long. So, how to address all of that memory? Their solution is to put together two 16 bit quantities like this:

    calculation SSSS0 - value in the relevant segment register SHL 4 depicted in AAAA - apparent address from register or instruction hexadecimal - RRRRR - real address placed on address bus. In other words, any time memory is accessed, your program will supply a sixteen bit address. Another sixteen bit address is acquired from a segment register, left shifted four bits (one nibble) and added to it to form the real address. You can control the values in the segment registers and thus access any part of memory you want. But the segment registers are specialized: one for code, one for most data accesses, one for the stack (which we'll mention again) and one "extra" one for additional data accesses.

Most people, when they first learn about this addressing scheme become obsessed with converting everything to real 20 bit addresses. After a while, though, you get use to thinking in segment/offset form. You tend to get your segment registers set up at the beginning of the program, change them as little as possible, and think just in terms of symbolic locations in your program, as with any assembly language.

```
    EXAMPLE:
        MOV  AX,DATASEG
        MOV  DS,AX          ;Set value of Data segment
        ASSUME DS:DATASEG   ;Tell assembler DS is usable
        MOV  AX,PLACE       ;Access storage symbolically by 16 bit
;address
```

In the above example, the assembler knows that no special issues are involved because the machine generally uses the DS register to complete a normal data reference.

If you had used ES instead of DS in the above example, the assembler would have known what to do, also. In front of the MOV instruction which accessed the location PLACE, it would have placed the ES segment prefix. This would tell the machine that ES should be used, instead of DS, to complete the address.

Some conventions make it especially easy to forget about segment registers. For example, any program of the COM type gets control with all four segment registers containing the same value. This program executes in a simplified 64K address space. You can go outside this address space if you want but you don't have to.

2. You will want to learn what other registers are available and learn their personalities:

AX and DX are general purpose registers. They become special only when accessing machine and system interfaces.

CX is a general purpose register which is slightly specialized for counting.

BX is a general purpose register which is slightly specialized for forming base-displacement addresses.

AX-DX can be divided in half, forming AH, AL, BH, BL, CH, CL, DH, DL.

SI and DI are strictly 16 bit. They can be used to form indexed addresses (like BX) and they are also used to point to strings.
SP is hardly ever manipulated. It is there to provide a stack.
BP is a manipulable cousin to SP. Use it to access data which has been pushed onto the stack.

Most sixteen bit operations are legal (even if unusual) when performed in SI, DI, SP, or BP.

3. You will want to learn the classifications of operations available WITHOUT getting hung up in the details of how 8086 opcodes are constructed.

8086 opcodes are complex. Fortunately, the assembler opcodes used to assemble them are simple. When you read a book like Morse, you will learn some things which are worth knowing but NOT worth dwelling on.

a. 8086 and 8088 instructions can be broken up into subfields and bits with names like R/M, MOD, S and W. These parts of the instruction modify the basic operation in such ways as whether it is 8 bit or 16 bit, if 16 bit, whether all 16 bits of the data are given, whether the instruction is register to register, register to memory, or memory to register, for operands which are registers, which register, for operands which are memory, what base and index registers should be used in finding the data.

b. Also, some instructions are actually represented by several different machine opcodes depending on whether they deal with immediate data or not, or on other issues, and there are some expedited forms which assume that one of the arguments is the most commonly used operand, like AX in the case of arithmetic.

There is no point in memorizing any of this detail; just distill the bottom line, which is, what kinds of operand combinations EXIST in the instruction set and what kinds don't. If you ask the assembler to ADD two things and the two things are things for which there is a legal ADD instruction somewhere in the instruction set, the assembler will find the right instruction and fill in all the modifier fields for you.

I guess if you memorized all the opcode construction rules you might have a crack at being able to disassemble hex dumps by eye, like you may have learned to do somewhat with 370 assembler. I submit to you that this feat, if ever mastered by anyone, would be in the same class as playing the "Minute Waltz" in a minute; a curiosity only.

Here is the basic matrix you should remember:

Two operands:	One operand:
R <-- M	R
M <-- R	M
R <-- R	S *
R|M <-- I	
R|M <-- S *	
S <-- R|M *	

* -- data moving instructions (MOV, PUSH, POP) only
S -- segment register (CS, DS, ES, SS)
R -- ordinary register (AX, BX, CX, DX, SI, DI, BP, SP, AH, AL, BH, BL, CH, CL, DH, DL)
M -- one of the following
pure address
[BX]+offset
[BP]+offset
any of the above indexed by SI
any of the first three indexed by DI

4. Of course, you want to learn the operations themselves. As I've suggested, you want to learn the op codes as the assembler presents them, not as the CPU machine language presents them. So, even though there are many MOV op codes you don't need to learn them. Basically, here is the instruction set:

a. Ordinary two operand instructions. These instructions perform an operation and leave the result in place of one of the operands. They are:

1) ADD and ADC -- addition, with or without including a carry from a previous addition
2) SUB and SBB -- subtraction, with or without including a borrow from a previous subtraction
3) CMP -- compare. It is useful to think of this as a subtraction with the answer being thrown away and neither operand actually changed
4) AND, OR, XOR -- typical boolean operations
5) TEST -- like an AND, except the answer is thrown away and neither operand is changed.
6) MOV -- move data from source to target
7) LDS, LES, LEA -- some specialized forms of MOV with side effects

b. Ordinary one operand instructions. These can take any of the operand forms described above. Usually, the perform the operation and leave the result in the stated place:

1) INC -- increment contents
2) DEC -- decrement contents
3) NEG -- twos complement
4) NOT -- ones complement
5) PUSH -- value goes on stack (operand location itself unchanged)
6) POP -- value taken from stack, replaces current value

c. Now you touch on some instructions which do not follow the general operand rules but which require the use of certain registers. The important ones are:

1) The multiply and divide instructions
2) The "adjust" instructions which help in performing arithmetic on ASCII or packed decimal data
3) The shift and rotate instructions. These have a restriction on the second operand: it must either be the immediate value 1 or the contents of the CL register.
4) IN and OUT which send or receive data from one of the 1024 hardware ports.
5) CBW and CWD -- convert byte to word or word to doubleword by sign extension

d. Flow of control instructions. These deserve study in themselves and we will discuss them a little more. They include:

1) CALL, RET -- call and return
2) INT, IRET -- interrupt and return-from-interrupt
3) JMP -- jump or "branch"
4) LOOP, LOOPNZ, LOOPZ -- special (and useful) instructions which implement a counted loop similar to the 370 BCT instruction
5) various conditional jump instructions

e. String instructions. These implement a limited storage-to-storage instruction subset and are quite powerful. All of them have the property that:

1) The source of data is described by the combination DS and SI.
2) The destination of data is described by the combination ES and DI.
3) As part of the operation, the SI and/or DI register(s) is (are) incremented or decremented so the operation can be repeated.

They include:

1) CMPSB/CMPSW -- compare byte or word
2) LODSB/LODSW -- load byte or word into AL or AX
3) STOSB/STOSW -- store byte or word from AL or AX
4) MOVSB/MOVSW -- move byte or word
5) SCASB/SCASW -- compare byte or word with contents of AL or AX
6) REP/REPE/REPNE -- a prefix which can be combined with any of the above instructions to make them execute repeatedly across a string of data whose length is held in CX.

f. Flag instructions: CLI, STI, CLD, STD, CLC, STC. These can set or clear the interrupt (enabled), direction (for string operations) or carry flags.

The addressing summary and the instruction summary given above masks a lot of annoying little exceptions. For example, you can't POP CS, and although the R <-- M form of LES is legal, the M <-- R form isn't etc., etc. My advice is:

a. Go for the general rules

b. Don't try to memorize the exceptions

c. Rely on common sense and the assembler to teach you about exceptions over time. A lot of the exceptions cover things you wouldn't want to do anyway.

5. A few instructions are rich enough and useful enough to warrant careful study. Here are a few final study guidelines:

a. It is well worth the time learning to use the string instruction set effectively. Among the most useful are

```
REP MOVSB          ;moves a string
REP STOSB          ;initialize memory
REPNE SCASB        ;look up occurrence of character in
                   ;string
REPE CMPSB         ;compare two strings
```

b. Similarly, if you have never written for a stack machine before, you will need to exercise PUSH and POP and get very comfortable with them because they are going to be good friends. If you are used to the 370, with lots of general purpose registers, you may find yourself feeling cramped at first, with many fewer registers and many instructions having register restrictions. But, you have a hidden ally: you need a register and you don't want to throw away what's in it? Just PUSH it, and when you are done, POP it back. This can lead to abuse.

Never have more than two "expedient" PUSHes in effect and never leave something PUSHed across a major header comment or for more than 15 instructions or so. An exception is the saving and restoring of registers at entrance to and exit from a subroutine; here, if the subroutine is long, you should probably PUSH everything which the caller may need saved, whether you will use the register or not, and POP it in reverse order at the end.

Be aware that CALL and INT push return address information on the stack and RET and IRET pop it off. It is a good idea to become familiar with the structure of the stack.

c. In practice, to invoke system services you will use the INT instruction. It is quite possible to use this instruction effectively in a cookbook fashion without knowing precisely how it works.

d. The transfer of control instructions (CALL, RET, JMP) deserve careful study to avoid confusion. You will learn that these can be classified as follows:

1) all three have the capability of being either NEAR (CS register unchanged) or FAR (CS register changed)

2) JMPs and CALLs can be DIRECT (target is assembled into instruction) or INDIRECT (target fetched from memory or register)

3) if NEAR and DIRECT, a JMP can be SHORT (less than 128 bytes away) or LONG

In general, the third issue is not worth worrying about. On a forward jump which is clearly VERY short, you can tell the assembler it is short and save one byte of code:

```
JMP SHORT  CLOSEBY
```

On a backward jump, the assembler can figure it out for you. On a forward jump of dubious length, let the assembler default to a LONG form; at worst you waste one byte.

Also leave the assembler to worry about how the target address is to be represented, in absolute form or relative form.

e. The conditional jump set is rather confusing when studied apart from the assembler, but you do need to get a feeling for it. The interactions of the sign, carry, and overflow flags can get your mind stuttering pretty fast if you worry about it too much. What it boils down to, though, is:

```
JZ  means what it says
JNZ means what it says
JG reater - this means "if the SIGNED difference is positive"
JA bove - this means "if the UNSIGNED difference is positive"
JL ess - this means "if the SIGNED difference is negative"
JB elow - this means "if the UNSIGNED difference is negative"
JC arry - assembles the same as JB; it's an aesthetic choice
```

You should understand that all conditional jumps are inherently DIRECT, NEAR, and "short"; the "short" part means that they can't go more than 128 bytes in either direction. Again, this is something you could easily imagine to be more of a problem than it is. I follow this simple approach:

1) When taking an abnormal exit from a block of code, I always use an unconditional jump. Who knows how far you are going to end up jumping by the time the program is finished. For example, I wouldn't code this:

```
TEST     AL,IDIBIT       ;Is the idiot bit on?
JNZ      OYVEY           ;Yes.  Go to general cleanup
```

Rather, I would probably code this:

```
TEST     AL,IDIBIT       ;Is the idiot bit on?
JZ       NOIDIOCY        ;No.  I am saved.
JMP      OYVEY           ;Yes.  What can we say...            NOIDIOCY:
```

The latter, of course, is a jump around a jump. Some would say it is evil, but I submit it is hard to avoid in this language.

2) Otherwise, within a block of code, I use conditional jumps freely. If the block eventually grows so long that the assembler starts complaining that my conditional jumps are too long here's what I do:

a) consider reorganizing the block but

b) also consider changing some conditional jumps to their opposite and use the "jump around a jump" approach as shown above.

Enough about specific instructions!

6. Finally, in order to use the assembler effectively, you need to know the default rules for which segment registers are used to complete addresses in which situations.

a. CS is used to complete an address which is the target of a NEAR DIRECT jump. On an NEAR INDIRECT jump, DS is used to fetch the address from memory but then CS is used to complete the address thus fetched. On FAR jumps, of course, CS is itself altered. The instruction counter is always implicitly pointing in the code segment.

b. SS is used to complete an address if BP is used in its formation. Otherwise, DS is always used to complete a data address.

c. On the string instructions, the target is always formed from ES and DI. The source is normally formed from DS and SI. If there is a segment prefix, it overrides the source not the target.

Learning about DOS.

I think the best way to learn about DOS internals is to read the technical appendices in the manual. These are not as complete as we might wish, but they really aren't bad; I certainly have learned a lot from them. What you don't learn from them you might eventually learn via judicious disassembly of parts of DOS, but that shouldn't really be necessary.

From reading the technical appendices, you learn that interrupts 20H through 27H are used to communicate with DOS. Mostly, you will use interrupt 21H, the DOS function manager.

The function manager implements a great many services. You request the individual services by means of a function code in the AH register. For example, by putting a nine in the AH register and issuing interrupt 21H you tell DOS to print a message on the console screen.

Usually, but by no means always, the DX register is used to pass data for the service being requested. For example, on the print message service just mentioned, you would put the 16 bit address of the message in the DX register. The DS register is also implicitly part of this argument, in keeping with the universal segmentation rules.

In understanding DOS functions, it is useful to understand some history and also some of the philosophy of MS-DOS with regard to portability. Generally, you will find, once you read the technical information on DOS and also the IBM technical reference, you will know more than one way to do almost anything. Which is best? For example, to do asynch adapter I/O, you can use the DOS calls (pretty incomplete), you can use BIOS, or you can go directly to the hardware. The same thing is true for most of the other primitive I/O (keyboard or screen) although DOS is more likely to give you added value in these areas. When it comes to file I/O, DOS itself offers more than one interface. For example, there are four calls which read data from a file.

The way to decide rationally among these alternatives is by understanding the tradeoffs of functionality versus portability. Three kinds of portability need to be considered: machine portability, operating system portability (for example, the ability to assemble and run code under CP/M 86) and DOS version portability (the ability for a program to run under older versions of DOS).

Most of the functions originally offered in DOS 1.0 were direct descendants of CP/M functions; there is even a compatibility interface so that programs which have been translated instruction for instruction from 8080 assembler to 8086 assembler might have a reasonable chance of running if they use only the core CP/M function set. Among the most generally useful in this original compatibility set are

09 -- print a full message on the screen
0A -- get a console input line with full DOS editing
0F -- open a file
10 -- close a file (really needed only when writing)
11 -- find first file matching a pattern
12 -- find next file matching a pattern
13 -- erase a file
16 -- create a file
17 -- rename a file
1A -- set disk transfer address

The next set provide no function above what you can get with BIOS calls or more specialized DOS calls. However, they are preferable to BIOS calls when portability is an issue.

00 -- terminate execution
01 -- read keyboard character
02 -- write screen character
03 -- read COM port character
04 -- write COM port character
05 -- print a character
06 -- read keyboard or write screen with no editing

The standard file I/O calls are inferior to the specialized DOS calls but have the advantage of making the program easier to port to CP/M style systems. Thus they are worth mentioning:

14 -- sequential read from file
15 -- sequential write to file
21 -- random read from file
22 -- random write to file
23 -- determine file size
24 -- set random record

In addition to the CP/M compatible services, DOS also offers some specialized services which have been available in all releases of DOS. These include

27 -- multi-record random read.
28 -- multi-record random write.
29 -- parse filename
2A-2D -- get and set date and time

All of the calls mentioned above which have anything to do with files make use of a data area called the "FILE CONTROL BLOCK" (FCB). The FCB is any-where from 33 to 37 bytes long depending on how it is used. You are responsible for creating an FCB and filling in the first 12 bytes, which contain a drive code, a file name, and an extension.

When you open the FCB, the system fills in the next 20 bytes, which includes a logical record length. The initial lrecl is always 128 bytes, to achieve CP/M compatibility. The system also provides other useful information such as the file size.

After you have opened the FCB, you can change the logical record length. If you do this, your program is no longer CP/M compatible, but that doesn't make it a bad thing to do. DOS documentation suggests you use a logical record length of one for maximum flexibility. This is usually a good recommendation.

To perform actual I/O to a file, you eventually need to fill in byte 33 or possibly bytes 34-37 of the FCB. Here you supply information about the record you are interested in reading or writing. For the most part, this part of the interface is compatible with CP/M.

In general, you do not need to (and should not) modify other parts of the FCB.

The FCB is pretty well described in appendix E of the DOS manual. Beginning with DOS 2.0, there is a whole new system of calls for managing files which don't require that you build an FCB at all. These calls are quite incompatible with CP/M and also mean that your program cannot run under older releases of DOS. However, these calls are very nice and easy to use. They have these characteristics

1. To open, create, delete, or rename a file, you need only a character string representing its name.

2. The open and create calls return a 16 bit value which is simply placed in the BX register on subsequent calls to refer to the file.

3. There is not a separate call required to specify the data buffer.

4. Any number of bytes can be transfered on a single call; no data area must be manipulated to do this.

The "new" DOS calls also include comprehensive functions to manipulate the new chained directory structure and to allocate and free memory.

Learning the assembler.

It is my feeling that many people can teach themselves to use the assembler by reading the MACRO Assembler manual if:

1. You have read and understood a book like Morse and thus have a feeling for the instruction set

2. You know something about DOS services and so can communicate with the keyboard and screen and do something marginally useful with files. In the absence of this kind of knowledge, you can't write meaningful practice programs and so will not progress.

3. You have access to some good examples (the ones supplied with the assembler are not good, in my opinion. I will try to supply you with some more relevant ones.

4. You ignore the things which are most confusing and least useful. Some of the most confusing aspects of the assembler include the facilities combining segments. But, you can avoid using all but the simplest of these facilities in many cases, even while writing quite substantial applications.

5. The easiest kind of assembler program to write is a COM program. They might seem harder, at first, then EXE programs because there is an extra step involved in creating the executable file, but COM programs are structurally very much simpler.

At this point, it is necessary to talk about COM programs and EXE programs. As you probably know, DOS supports two kinds of executable files. EXE pro-grams are much more general, can contain many segments, and are generally built by compilers and sometimes by the assembler. If you follow the lead given by the samples distributed with the assembler, you will end up with EXE programs. A COM program, in contrast, always contains just one segment, and receives control with all four segment registers containing the same value. A COM program, thus, executes in a simplified environment, a 64K address space. You can go outside this address space simply by temporarily changing one segment register, but you don't have to, and that is the thing which makes COM programs nice and simple. Let's look at a very simple one.

The classic text on writing programs for the C language says that the first thing you should write is a program which says

    HELLO, WORLD.

when invoked. What's sauce for C is sauce for assembler, so let's start with a HELLO program of our own. My first presentation of this will be bare bones, not stylistically complete, but just an illustration of what an assembler program absolutely has to have:

```
        HELLO  SEGMENT                  ;Set up HELLO code and data
                                ;section
    ASSUME CS:HELLO,DS:HELLO              ;Tell assembler about conditions at ;entry
    ORG  100H                             ;A .COM program begins with 100H ;byte
prefix
MAIN:
    JMP  BEGIN                          ;Control must start here
    MSG  DB   'Hello, world.$'   ;But it is generally useful to put ;data first
BEGIN:
    MOV  DX,OFFSET MSG                    ;Let DX --> message.
    MOV  AH,9                           ;Set DOS function code for printing ;a message
    INT  21H                        ;Invoke DOS
    RET                      ;Return to system
HELLO  ENDS                  ;End of code and data section
END  MAIN                                 ;Terminate assembler and specify ;entry
point
```

First, let's attend to some obvious points. The macro assembler uses the general form

```
  name    opcode    operands
```

Unlike the 370 assembler, though, comments are NOT set off from operands by blanks. The syntax uses blanks as delimiters within the operand field (see line 6 of the example) and so all comments must be set off by semi-colons.

Line comments are frequently set off with a semi-colon in column 1. I use this approach for block comments too, although there is a COMMENT statement which can be used to introduce a block comment.

Being an old 370 type, I like to see assembler code in upper case, although my comments are mixed case. Actually, the assembler is quite happy with mixed case anywhere.

As with any assembler, the core of the opcode set consists of opcodes which generate machine instructions but there are also opcodes which generate data and ones which function as instructions to the assembler itself, some-times called pseudo-ops. In the example, there are five lines which generate machine code (JMP, MOV, MOV, INT, RET), one line which generates data (DB) and five pseudo-ops (SEGMENT, ASSUME, ORG, ENDS, and END).

We will discuss all of them.

Now, about labels. You will see that some labels in the example end in a colon and some don't. This is just a bit confusing at first, but no real mystery. If a label is attached to a piece of code (as opposed to data), then the assembler needs to know what to do when you JMP to or CALL that label. By convention, if the label ends in a colon, the assembler will use the NEAR form of JMP or CALL. If the label does not end in a colon, it will use the FAR form. In practice, you will always use the colon on any label you are jumping to inside your program because such jumps are always NEAR; there is no reason to use a FAR jump within a single code section. I mention this, though, because leaving off the colon isn't usually trapped as a syntax error, it will generally cause something more abstruse to go wrong.

On the other hand, a label attached to a piece of data or a pseudo-op never ends in a colon.

Machine instructions will generally take zero, one or two operands. Where there are two operands, the one which receives the result goes on the left as in 370 assembler.

I tried to explain this before, now maybe it will be even clearer: there are many more 8086 machine opcodes then there are assembler opcodes to rep-resent them. For example, there are five kinds of JMP, four kinds of CALL, two kinds of RET, and at least five kinds of MOV depending on how you count them. The macro assembler makes a lot of decisions for you based on the form taken by the operands or on attributes assigned to symbols elsewhere in your program. In the example above, the assembler will generate the NEAR DIRECT form of JMP because the target label BEGIN labels a piece of code instead of a piece of data (this makes the JMP DIRECT) and ends in a colon (this makes the JMP NEAR). The assembler will generate the immediate forms of MOV because the form OFFSET MSG refers to immediate data and because 9 is a constant. The assembler will generate the NEAR form of RET because that is the default and you have not told it otherwise.
The DB (define byte) pseudo-op is an easy one: it is used to put one or more bytes of data into storage. There is also a DW (define word) pseudo-op and a DD (define doubleword) pseudo-op; in the PC MACRO assembler, the fact that a label refers to a byte of storage, a word of storage, or a doubleword of storage can be very significant in ways which we will see presently.

About that OFFSET operator, I guess this is the best way to make the point about how the assembler decides what instruction to assemble: an analogy with 370 assembler:

```
 PLACE    DC   ......
          ...
          LA   R1,PLACE
          L    R1,PLACE
```

In 370 assembler, the first instruction puts the address of label PLACE in register 1, the second instruction puts the contents of storage at label PLACE in register 1. Notice that two different opcodes are used. In the PC assembler, the analogous instructions would be

```
 PLACE    DW   ......
          ...
          MOV  DX,OFFSET PLACE
          MOV  DX,PLACE
```

If PLACE is the label of a word of storage, then the second instruction will be understood as a desire to fetch that data into DX. If X is a label, then "OFFSET X" means "the ordinary number which represents X's off-set from the start of the segment." And, if the assembler sees an ordinary number, as opposed to a label, it uses the instruction which is equivalent to LA.

If PLACE were the label of a DB pseudo-op, instead of a DW, then

```
          MOV  DX,PLACE
```

would be illegal. The assembler worries about length attributes of its operands.

Next, numbers and constants in general. The assembler's default radix is decimal. You can change this, but I don't recommend it. If you want to represent numbers in other forms of notation such as hex or bit, you generally use a trailing letter. For example,

```
          21H
```

is hexidecimal 21,

```
          00010000B
```

is the eight bit binary number pictured.

The next elements we should point to are the SEGMENT...ENDS pair and the END instruction. Every assembler program has to have these elements.

SEGMENT tells the assembler you are starting a section of contiguous material (code and/or data). The symmetrically named ENDS statement tells the assembler you are finished with a section of contiguous material. I wish they didn't use the word SEGMENT in this context. To me, a "segment" is a hardware construct: it is the 64K of real storage which becomes address-able by virtue of having a particular value in a segment register. Now, it is true that the "segments" you make with the assembler often correspond to real hardware "segments" at execution time. But, if you look at things like the GROUP and CLASS options supported by the linker, you will discover that this correspondence is by no means exact. So, at risk of maybe con-fusing you even more, I am going to use the more informal term "section" to refer to the area set off by means of the SEGMENT and ENDS instructions.

The sections delimited by SEGMENT...ENDS pairs are really a lot like CSECTs and DSECTs in the 370 world.

I strongly recommend that you be selective in your study of the SEGMENT pseudo-op as described in the manual. Let me just touch on it here.

```
 name     SEGMENT
 name     SEGMENT  PUBLIC
 name     SEGMENT  AT  nnn
```

Basically, you can get away with just the three forms given above. The first form is what you use when you are writing a single section of assembler code which will not be combined with other pieces of code at link time.   The second form says that this assembly only contains part of the section; other parts might be assembled separately and combined later by the linker.

I have found that one can construct reasonably large modular applications in assembler by simply making every assembly use the same segment name and declaring the name to be PUBLIC each time. If you read the assembler and linker documentation, you will also be bombarded by information about more complex options such as the GROUP statement and the use of other "combine types" and "classes." I don't recommend getting into any of that. I will talk more about the linker and modular construction of programs a little later. The assembler manual also implies that a STACK segment is required. This is not really true. There are numerous ways to assure that you have a valid stack at execution time.

Of course, if you plan to write applications in assembler which are more than 64K in size, you will need more than what I have told you; but who is really going to do that? Any application that large is likely to be coded in a higher level language.

The third form of the SEGMENT statement makes the delineated section into something like a "DSECT;" that is, it doesn't generate any code, it just describes what is present somewhere already in the computer's memory. Sometimes the AT value you give is meaningful. For example, the BIOS work area is located at location 40 hex. So, you might see

```
 BIOSAREA  SEGMENT AT 40H         ;Map BIOS work area
          ORG  BIOSAREA+10H
 EQUIP     DB   ?                  ;Location of equipment flags, first ;byte
 BIOSAREA  ENDS
```

in a program which was interested in mucking around in the BIOS work area.

At other times, the AT value you give may be arbitrary, as when you are mapping a repeated control block:

```
PROGPREF SEGMENT   AT 0                    ;Really a DSECT mapping the
program ;prefix
              ORG   PROGPREF+6
MEMSIZE  DW   ?                         ;Size of available memory
PROGPREF ENDS
```

Really, no matter whether the AT value represents truth or fiction, it is your responsibility, not the assembler's, to get set up a segment register so that you can really reach the storage in question.   So, you can't say

```
      MOV  AL,EQUIP
```

unless you first say something like

```
      MOV  AX,BIOSAREA               ;BIOSAREA becomes a symbol with ;value 40H
      MOV  ES,AX
      ASSUME ES:BIOSAREA
```

Enough about SEGMENT. The END statement is simple. It goes at the end of every assembly. When you are assembling a subroutine, you just say

```
      END
```

but when you are assembling the main routine of a program you say

```
      END label
```

where 'label' is the place where execution is to begin.

Another pseudo-op illustrated in the program is ASSUME. ASSUME is like the USING statement in 370 assembler. However, ASSUME can ONLY refer to segment registers. The assembler uses ASSUME information to decide whether to assemble segment override prefixes and to check that the data you are trying to access is really accessible. In this case, we can reassure the assembler that both the CS and DS registers will address the section called HELLO at execution time. Actually, the SS and ES registers will too, but the assembler never needs to make use of this information.

I guess I have explained everything in the program except that ORG pseudo-op. ORG means the same thing as it does in many assembly languages. It tells the assembler to move its location counter to some particular address. In this case, we have asked the assembler to start assembling code hex 100 bytes from the start of the section called HELLO instead of at the very beginning. This simply reflects the way COM programs are loaded. When a COM program is loaded by the system, the system sets up all four segment registers to address the same 64K of storage. The first 100 hex bytes of that storage contains what is called the program prefix; this area is described in appendix E of the DOS manual. Your COM program physically begins after this. Execution begins with the first physical byte of your program; that is why the JMP instruction is there.

Wait a minute, you say, why the JMP instruction at all? Why not put the data at the end? Well, in a simple program like this I probably could have gotten away with that. However, I have the habit of putting data first and would encourage you to do the same because of the way the assembler has of assembling different instructions depending on the nature of the operand.

Unfortunately, sometimes the different choices of instruction which can assemble from a single opcode have different lengths. If the assembler has already seen the data when it gets to the instructions it has a good chance of reserving the right number of bytes on the first pass. If the data is at the end, the assembler may not have enough information on the first pass to reserve the right number of bytes for the instruction. Sometimes the assembler will complain about this, something like "Forward reference is illegal" but at other times, it will make some default assumption. On the second pass, if the assumption turned out to be wrong, it will report what is called a "Phase error," a very nasty error to track down. So get in the habit of putting data and equated symbols ahead of code.

OK. Maybe you understand the program now. Let's walk through the steps involved in making it into a real COM file.

1. The file should be created with the name HELLO.ASM (actually the name is arbitrary but the extension .ASM is conventional and useful)

2. ASM HELLO,,;

> (this is just one example of invoking the assembler; it uses the small assembler ASM, it produces an object file and a listing file with the same name as the source file. I am not going exhaustively into how to invoke the assembler, which the manual goes into pretty well. I guess this is the first time I mentioned that there are really two assemblers; the small assembler ASM will run in a 64K machine and doesn't support macros. I used to use it all the time; now that I have a bigger machine and a lot of macro libraries I use the full function assembler MASM. You get both when you buy the package).

3. If you issue DIR at this point, you will discover that you have acquired HELLO.OBJ (the object code resulting from the assembly) and HELLO.LST (a listing file). I guess I can digress for a second here concerning the listing file. It contains TAB characters. I have found there are two good ways to get it printed and one bad way. The bad way is to use LPT1: as the direct target of the listing file or to try copying the LST file to LPT1 without first setting the tabs on the printer. The two good ways are to either

   a. direct it to the console and activate the printer with CTRL-PRTSC. In this case, DOS will expand the tabs for you.

   b. direct to LPT1: but first send the right escape sequence to LPT1 to set the tabs every eight columns. I have found that on some early serial numbers of the IBM PC printer, tabs don't work quite right, which forces you to the first option.

4. LINK HELLO;

   (again, there are lots of linker options but this is the simplest. It takes HELLO.OBJ and makes HELLO.EXE). HELLO.EXE? I thought we were making a COM program, not an EXE program. Right. HELLO.EXE isn't really executable; its just that the linker doesn't know about COM programs. That requires another utility. You don't have this utility if you are using DOS 1.0; you have it if you are using DOS 1.1 or DOS 2.0. Oh, by the way, the linker will warn you that you have no stack segment. Don't worry about it.

5. EXE2BIN HELLO HELLO.COM

   This is the final step. It produces the actual program you will execute. Note that you have to spell out HELLO.COM; for a nominally rational but actually perverse reason, EXE2BIN uses the default extension BIN instead of COM for its output file. At this point, you might want to erase HELLO.EXE; it looks a lot more useful than it is. Chances are you won't need to recreate HELLO.COM unless you change the source and then you are going to have to redo the whole thing.

6. HELLO

You type hello, that invokes the program, it says

```
HELLO YOURSELF!!!
```

(oops, what did I do wrong....?)

What about subroutines?

I started with a simple COM program because I actually think they are easier to create than subroutines to be called from high level languages, but maybe its really the latter you are interested in. Here, I think you should get comfortable with the assembler FIRST with little exercises like the one above and also another one which I will finish up with.

Next you are ready to look at the interface information for your particular language. You usually find this in some sort of an appendix. For example, the BASIC manual has Appendix C on Machine Language Subroutines. The PASCAL manual buries the information a little more deeply: the interface to a separately compiled routine can be found in the Chapter on Procedures and Functions, in a subsection called Internal Calling Conventions.

Each language is slightly different, but here are what I think are some common issues in subroutine construction.

1. NEAR versus FAR? Most of the time, your language will probably call your assembler routine as a FAR routine. In this case, you need to make sure the assembler will generate the right kind of return. You do this with a PROC...ENDP statement pair. The PROC statement is probably a good idea for a NEAR routine too even though it is not strictly required:

```
            FAR linkage:        |           NEAR linkage:                                    |
            ARBITRARY SEGMENT   |  SPECIFIC  SEGMENT  PUBLIC                        PUBLIC
THENAME      |  PUBLIC THENAME
            ASSUME CS:ARBITRARY |  ASSUME CS:SPECIFIC,DS:SPECIFIC
THENAME   PROC FAR  |  ASSUME ES:SPECIFIC,SS:SPECIFIC                   ..... code and
data |  THENAME   PROC NEAR THENAME
                     ENDP      |
                    ..... code and data ....
            ARBITRARY ENDS      |  THENAME   ENDP
            END                 |  SPECIFIC  ENDS
                            |           END
```

With FAR linkage, it doesn't really matter what you call the segment, you must declare the name by which you will be called in a PUBLIC pseudo-op and also show that it is a FAR procedure. Only CS will be initialized to your segment when you are called. Generally, the other segment registers will continue to point to the caller's segments.

With NEAR linkage, you are executing in the same segment as the caller. Therefore, you must give the segment a specific name as instructed by the language manual. However, you may be able to count on all segment registers pointing to your own segment (sometimes the situation can be more complicated but I cannot really go into all of the details). You should be aware that the code you write will not be the only thing in the segment and will be physically relocated within the segment by the linker. However, all OFFSET references will be relocated and will be correct at execution time.

2. Parameters passed on the stack. Usually, high level languages pass parameters to subroutines by pushing words onto the stack prior to calling you. What may differ from language to language is the nature of what is pushed (OFFSET only or OFFSET and SEGMENT) and the order in which it is pushed (left to right, right to left within the CALL statement). However, you will need to study the examples to figure out how to retrieve the parameters from the stack. A useful fact to exploit is the fact that a reference involving the BP register defaults to a reference to the stack segment. So, the following strategy can work:

```
ARGS    STRUC
        DW   3 DUP(?)  ;Saved BP and return address
ARG3    DW   ?
ARG2    DW   ?
ARG1    DW   ?
ARGS    ENDS
      ...........
        PUSH BP              ;save BP register
        MOV  BP,SP            ;Use BP to address stack
        MOV   ...,[BP].ARG2          ;retrieve second argument ;(etc.)
```

This example uses something called a structure, which is only available in the large assembler; furthermore, it uses it without allocating it, which is not a well-documented option. However, I find the above approach generally pleasing. The STRUC is like a DSECT in that it establishes labels as being offset a certain distance from an arbitrary point; these labels are then used in the body of code by beginning them with a period; the construction ".ARG2" means, basically, " + (ARG2-ARGS)."

What you are doing here is using BP to address the stack, accounting for the word where you saved the caller's BP and also for the two words which were pushed by the CALL instruction.

3.   How big is the stack? BASIC only gives you an eight word stack to play with. On the other hand, it doesn't require you to save any registers except the segment registers. Other languages give you a liberal stack, which makes things a lot easier. If you have to create a new stack segment for yourself, the easiest thing is to place the stack at the end of your program and:

```
       CLI                              ;suppress interrupts while changing the ;stack
       MOV  SSAVE,SS                    ;save old SS in local storage (old SP ;already saved in
BP)
       MOV  SP,CS           ;switch
       MOV  SS,SP           ;the
       MOV  SP,OFFSET STACKTOP  ;stack
       STI                  ;(maybe)
```

Later, you can reverse these steps before returning to the caller. At the end of your program, you place the stack itself:

```
       DW   128 DUP(?)      ;stack of 128 words (liberal)
       STACKTOP LABEL WORD
```

4.   Make sure you save and restore those registers required by the caller.

5.   Be sure to get the right kind of addressability. In the FAR call example, only CS addresses your segment. If you are careful with your ASSUME statements the assembler will keep track of this fact and generate CS prefixes when you make data references; however, you might want to do something like:

```
         MOV AX,CS      ;get current segment address
         MOV DS,AX      ;To DS
         ASSUME DS:THISSEG
```

Be sure you keep your ASSUMEs in synch with reality.

Learning about BIOS and the hardware

You can't do everything with DOS calls. You may need to learn something about the BIOS and about the hardware itself. In this, the Technical Reference is a very good thing to look at.

The first thing you look at in the Technical Reference, unless you are really determined to master the whole ball of wax, is the BIOS listings presented in Appendix A. Glory be: here is the whole 8K of ROM which deals with low level hardware support layed out with comments and everything.

In fact, if you are just interested in learning what BIOS can do for you, you just need to read the header comments at the beginning of each section of the listing.

BIOS services are invoked by means of the INT instruction; the BIOS occupies interrupts 10H through 1FH and also interrupt 5H; actually, of these seventeen interrupts, five are used for user exit points or data pointers, leaving twelve actual services.

In most cases, a service deals with a particular hardware interface; for example, BIOS interrupt 10H deals with the screen. As with DOS function calls, many BIOS services can be passed a function code in the AH register and possible other arguments.

I am not going to summarize the most useful BIOS features here; you will see some examples in the next sample program we will look at.
The other thing you might want to get into with the Tech reference is the description of some hardware options, particularly the asynch adapter, which are not well supported in the BIOS. The writeup on the asynch adapter is pretty complete.

Actually, the Tech reference itself is pretty complete and very nice as far as it goes. One thing which is missing from the Tech reference is information on the programmable peripheral chips on the system board. These include

the 8259 interrupt controller
the 8253 timer
the 8237 DMA controller and
the 8255 peripheral interface

To make your library absolutely complete, you should order the INTEL data sheets for these beasts.

I should say, though, that the only I ever found I needed to know about was the interrupt controller. If you happen to have the 8086 Family User's Manual, the big book put out by INTEL, which is one of the things people sometimes buy to learn about 8086 architecture, there is an appendix there which gives an adequate description of the 8259.

A final example

I leave you with a more substantial example of code which illustrates some good elementary techniques; I won't claim its style is perfect, but I think it is adequate. I think this is a much more useful example than what you will get with the assembler:

```
        PAGE 61,132
        TITLE SETSCRN -- Establish correct monitor use at boot time ;
; This program is a variation on many which toggle the equipment flags ; to support
the use of either video option (monochrome or color).
; The thing about this one is it prompts the user in such a way that he ; can select
the use of the monitor he is currently looking at (or
; which is currently connected or turned on) without really having to
; know which is which.  SETSCRN is a good program to put first in an
; AUTOEXEC.BAT file.
;
; This program is highly dependent on the hardware and BIOS of the
; IBMPC and is hardly portable, except to very exact clones.  For this
; reason, BIOS calls are used in lieu of DOS function calls where both ; provide equal
function.
;
```

OK. That's the first page of the program. Notice the PAGE statement, which you can use to tell the assembler how to format the listing. You give it lines per page and characters per line. I have mine setup to print on the host line printer; I routinely upload my listings at 9600 baud and print them on the host; it is faster than using the PC printer.

There is also a TITLE statement. This simply provides a nice title for each page of your listing. Now for the second page:

```
        SUBTTL -- Provide .COM type environment and Data
        PAGE
;
; First, describe the one BIOS byte we are interested in
;
 BIOSDATA  SEGMENT   AT 40H    ;Describe where BIOS keeps his data
;
; Skip parts we are not interested in
;
 EQUIP     DB        ?         ;Equipment flag location
 MONO      EQU       00110000B ;These bits on if monochrome
 COLOR     EQU       11101111B ;Mask to make BIOS think of the color
;board
 BIOSDATA  ENDS                ;End of interesting part
```

```
;
;        Next, describe some values for interrupts and functions
;
 DOS      EQU      21H      ;DOS Function Handler INT code PRTMSG
 EQU      09H               ;Function code to print a message
 KBD      EQU      16H      ;BIOS keyboard services INT code GETKEY
 EQU       00H                 ;Function code to read a character
 SCREEN    EQU       10H       ;BIOS Screen services INT code MONOINIT
 EQU       02H       ;Value to initialize monochrome screen
;
 COLORINIT EQU       03H       ;Value to initialize color screen
                              ;(80x25)
 COLORINIT EQU      01H        ;Value to initialize color screen
;(40X25)
;
;         Now, describe our own segment
;
SETSCRN   SEGMENT              ;Set operating segment for CODE and
;DATA
;
ASSUME CS:SETSCRN,DS:SETSCRN,ES:SETSCRN,SS:SETSCRN    ;All segments
;
         ORG       100H      ;Begin assembly at standard .COM offset ;
MAIN      PROC      NEAR      ;COM files use NEAR linkage
         JMP       BEGIN     ;And, it is helpful to put the data
                             ;first, but
;                             ;then you must branch around it.
;
;         Data used in SETSCRN
;
CHANGELOC   DD      EQUIP     ;Location of the EQUIP, recorded as far
;pointer
MONOPROMPT  DB      'Please press the plus ( + ) key.$'    ;User sees
;on mono
COLORPROMPT DB      'Please press the minus ( - ) key.$'   ;User sees
                             ;on color
```

Several things are illustrated on this page. First, in addition to titles, the assembler supports subtitles: hence the SUBTTL pseudo-op. Second, the PAGE pseudo-op can be used to go to a new page in the listing. You see an example here of the DSECT-style segment in the "SEGMENT AT 40H". Here, our interest is in correctly describing the location of some data in the BIOS work area which really is located at segment 40H.

You will also see illustrated the EQU instruction, which just gives a symbolic name to a number. I don't make a fetish of giving a name to every single number in a program. I do feel strongly, though, that interrupts and function codes, where the number is arbitrary and the function being performed is the thing of interest, should always be given symbolic names.

One last new element in this section is the define doubleword (DD) instruction. A doubleword constant can refer, as in this case, to a location in another segment. The assembler will be happy to use information at its disposal to properly assemble it. In this case, the assembler knows that EQUIP is offset 10 in the segment

BIOSDATA which is at 40H.

```
        SUBTTL -- Perform function
        PAGE
BEGIN:    CALL      MONOON                  ;Turn on mono display                   MOV
DX,OFFSET MONOPROMPT    ;GET MONO PROMPT
        MOV       AH,PRTMSG               ;ISSUE
        INT       DOS                     ;IT
        CALL      COLORON                 ;Turn on color display
        MOV       DX,OFFSET COLORPROMPT   ;GET COLOR PROMPT
        MOV       AH,PRTMSG               ;ISSUE
        INT       DOS                     ;IT
        MOV       AH,GETKEY               ;Obtain user response
        INT       KBD
        CMP       AL,'+'                  ;Does he want MONO?
        JNZ       NOMONO
        CALL      MONOON                  ;yes.  give it to him NOMONO:   RET
MAIN      ENDP
```

The main code section makes use of subroutines to keep the basic flow simple. About all that's new to you in this section is the use of the BIOS interrupt KBD to read a character from the keyboard.

Now for the subroutines, MONOON and COLORON:

```
        SUBTTL -- Routines to turn monitors on
        PAGE
MONOON    PROC      NEAR                ;Turn mono on
        LES       DI,CHANGELOC        ;Get location to change          ASSUME
ES:BIOSDATA                   ;TELL ASSEMBLER ABOUT CHANGE TO
;ES
        OR        EQUIP,MONO
        MOV       AX,MONOINIT         ;Get screen initialization
;value
        INT       SCREEN              ;Initialize screen
        RET
MONOON    ENDP
COLORON   PROC      NEAR                ;Turn color on
        LES       DI,CHANGELOC        ;Get location to change          ASSUME
ES:BIOSDATA                   ;TELL ASSEMBLER ABOUT CHANGE TO
;ES
        AND       EQUIP,COLOR
        MOV       AX,COLORINIT        ;Get screen initialization
;value
        INT       SCREEN              ;Initialize screen
        RET
COLORON   ENDP
SETSCRN   ENDS                          ;End of segment
        END       MAIN                ;End of assembly; execution at MAIN
```

The instructions LES and LDS are useful ones for dealing with doubleword addresses. The offset is loaded into the operand register and the segment into ES (for LES) or DS (for LDS). By telling the assembler, with an ASSUME, that ES now addresses the BIOSDATA segment, it is able to correctly assemble the OR and AND instructions which refer to the EQUIP byte. An ES segment prefix is added.

To understand the action here, you simply need to know that flags in that particular byte control how the BIOS screen service initialize the adapters. BIOS will only work with one adapter at a time; by setting the equipment flags to show one or the other as installed and calling BIOS screen initialization, we achieve the desired effect.

The rest is up to you.