LESSON 4

.COM AND .EXE FILES

Many computer systems have only one kind of executable file type. The IBM PC has two, the .COM and the .EXE file types. Actually, there are three types if you include the .BAT file type, but the .BAT file is just a list of DOS commands which are executed one after the other by DOS. Here we are interested only in files of machine commands, and this limits us to the .COM and .EXE types. A comparison of these two types gives some useful insights into how the machine calls and runs a program.

.COM

Files of this type can be at most one segment (65536 bytes) long. The code, data, and stack, as well as any other segments are all the same segment.

All JMP and CALL commands and references to data are to addresses within the segment and are assembled as offsets from the start of the segment. These offsets are calculated at Link time. There is nothing more of this type to calculate at the point when the program is loaded into memory.

Each .COM program has exactly one FAR procedure which is the routine to which DOS passes control. All routines called from the main procedure must be NEAR procedures.

Usually a .COM file is created by running the program EXE2BIN with a .EXE file as input. Only .EXE files created with a single common segment as discussed above can be transformed to a .COM file. The EXE2BIN default output file extension is .BIN, and must be renamed to .COM before execution.

.EXE

Files of this type can be as large as can be fit into memory. They can consist of many segments. These segments may be contiguous, overlapping, and/or disjoint.

The offsets involved in references of one segment from another can not be fully coded at link time because it is not known where each segment will be loaded into memory when the program is run. The offsets must be calculated at the time the program is loaded, therefore .EXE programs will take longer to load than .COM programs.

.EXE programs can call NEAR or FAR procedures at will.

.EXE programs are the natural result of the LINK process. If the .EXE program is later to be converted to a .COM file the source will have no Segment Stack directive, and as a result the linker will generate a "No Stack Segment" warning which can be ignored.

What happens when DOS loads a program for execution is both interesting and educational. We will only touch on the subject at this point. Further details can be found in references f) and g). DOS first creates a block of 100H bytes called the "Program Segment Prefix" (PSP). Most of this block is used by DOS for things like the address to jump to if the Control-Break key combination is pressed. One area of the PSP is of interest to us for insight, and possible use. The area from 80H to 100H is used to store any text which
was entered following the program name on the command line, that is, in
response to the DOS prompt. The byte at 80H contains the number of bytes of
text, and the following bytes are that text in ASCII. Thus, if you want to
debug your program GOODCODE.EXE, you type after the prompt:

B>DEBUG GOODCODE.EXE

The byte at 80H will contain C for the 12d bytes of GOODCODE.EXE. Byte 81H
will contain G, and so on. Debug is written with this addressing in mind,
and finds from 80H how many bytes to read. It then reads these bytes and
interprets them as the name of a file it is to fetch from the disk. The
existence of this facility for passing information to the program that is
being called effects how DOS loads the segment registers when the program is
called. You may have (should have?) wondered why we always have to establish
Data Segment addressability with:

```
MOV AX,DSEG
MOV DS,AX
```

The reason is that DS is initially pointing into the PSP in case we want to
retrieve some information that was entered with the program call.

Let us now look at how all the important registers are loaded when a
program is called from DOS.

```
FOR A .COM FILE

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>-00</td>
</tr>
<tr>
<td>DS</td>
<td>-80</td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>-100</td>
</tr>
<tr>
<td>SP</td>
<td>-65535D</td>
</tr>
</tbody>
</table>

APPLICATION
CODE
OF
.COM
FILE

PSP
```

```
FOR A .EXE FILE

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS</td>
<td>&gt;</td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>CODE SEGMENT</td>
</tr>
<tr>
<td>IP</td>
<td>LABEL:(as set by END directive)</td>
</tr>
<tr>
<td>SS</td>
<td>DATA SEGMENT</td>
</tr>
<tr>
<td>SP</td>
<td>STACK SEGMENT</td>
</tr>
</tbody>
</table>
```

Note that in the .COM case the IP is set at 100H into the code segment.
It is the responsibility of the programmer to see that this is where the
program should actually start. Normally this task is accomplished with the
ORG assembly directive (see chapter 9). When we assemble a program in DEBUG
The result is a .COM file. You will recall from the DEBUG instructions of lesson 2 that we started the assembly with an "al00". The 100 is to get us past the PSP. In the case of the .EXE program the IP is set to the start point we pick in the END statement of the program or to the first instruction in the code segment if the END statement does not indicate a start address.

For .EXE programs the SP is set to the top (high address) of the STACK SEGMENT, and it grows down into the segment. In .COM programs the SP is set to the highest address in the segment, at offset 65536 if sufficient memory is available. Again the stack grows to lower addresses.

Note that for both .COM and .EXE files DS points to the beginning of the PSP area. The first thing in the PSP is an INT 20H instruction which is the RETURN TO DOS instruction. We have been using the following "magic" code in our programs which are to be run from DOS:

```
PUSH DS
MOV AX, 0
PUSH AX
```

When the program ends its main FAR procedure the RET pops the 0 into IP and the old DS value into CS. The next instruction executed then is the first one of the PSP, the INT 20H. We can not simply end our program with an INT 20H without arranging for CS to point to the beginning of the PSP. INT 20H, wherever it is found, needs to have CS pointing to the beginning of the PSP since one of the things stored in the PSP is the terminate address, the address to which the machine is to jump on leaving the program.

For simplicity and agreement with the text we will always assume we are going to use .EXE files. It is interesting to note that DOS treats calls to executable programs in manner which may be surprising. One does not need to type the file extension. Thus, one can type Q to invoke QEdit rather than typing Q.EXE. What if on the disk there are files PROG.COM, PROG.EXE, AND PROG.BAT? What gets executed when you type PROG? DOS will execute PROG.COM. If only the .EXE and the .BAT files exist then the .EXE file is executed. The priority goes from .COM to .EXE to .BAT. The surprising part is that if all three types of files exist and you type PROG.BAT or PROG.EXE it is still the .COM file which is executed. When running programs DOS does not even read the file extension, it just looks for files in the priority order.
LESSON 5

PORTS, GENERAL INTERRUPTS, AND INTERRUPT SERVICE Routines

Ports

If a computer is to talk with the outside world it must have an "address" to which it can send data, or from which it can receive data. These addresses are 16 bit numbers just like memory addresses. In some computers the only difference between memory and input/output, I/O, is that different addresses are reserved for each use. In the IBM PC the address numbers are the same but the associated instructions are different; IN and OUT are the two instructions which relate to I/O. The I/O addresses are called "ports" in the IBM PC. Though the 8088 microprocessor is capable of decoding full 16 bit port addresses the IBM PC only decodes (looks at) the lower ten bits. Thus the IBM PC has only 1024 port addresses. We will discuss the addresses available to us shortly. In some other computers ports are referred to as registers (I have used the latter nomenclature for a long time, and am likely to revert to it at any time). The use of IN and OUT to exchange data between a port and AX or AL is covered in the text starting on p.291.

Unfortunately, Thorne uses an atrocious non-standard notation for numbering the various bits in a port.

Thorne

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

The rest of the world

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Your first reaction might be that you like Thorne's way better, but there is a good reason why the rest of the world does it differently. Suppose I want to set to 1 the #2 bit in each style of labeling. With the ROTW (rest of the world) scheme each bit's number is just the exponent of 2 needed to generate the number which, if loaded into the port, sets the correct bit (got that!). Look at our example. 2 to the 2 equals 4 so

MOV AL, 4H
OUT PORT_NUMBER, AL

sets the 2 bit (and clears all the others) in our port.

Now look at the same case with the Thorne notation. We first must translate to ROTW notation to find that his 2 is our 6. After that it is easy. Two to the 6 is 64 so either

MOV AL, 64
OUT PORT_NUMBER, AL

or

MOV AL, 40H
OUT PORT_NUMBER, AL

will do the job.
I will use "ROTW" notation in the lessons. Sixteen bit ports are, of course, numbered from 15 for the left most bit to 0 for the right most bit.

Thorne includes a discussion of the printer ports. We have included the IBM data sheets for the printer adapter in the Appendix to this lesson. These sheets have some additional information, including which pins on the connectors are connected to which bits in the ports. Notice that Thorne and IBM differ on the "BUSY" line. The IBM sheet should list the busy line with +Busy (bus when low) rather than +8Busy. Note also that Thorne's program on p.294 missing a OUT DX,AL right after the 50 microsecond WAIT loop. Further, Thorne's program will put out the characters as advertised, the final result depends on the printer attached to the port. My printer, for example, stores characters in its buffer until a carriage return (CR) is typed, and then prints out all the line at once.

Our final note on the printer is that the addresses given by Thorne are those for the printer port on the video interface card. Stand alone printer cards have the base address (data port address) of 378H or, if properly modified, 278H. Rather than introduce port addresses as we encounter them with various tasks, let us give the whole list here.

### Port Assignments

<table>
<thead>
<tr>
<th>Port Number (Hex)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000-1FF</td>
<td>Reserved for System Board Use (See Thorne p.305)</td>
</tr>
<tr>
<td>200-20F</td>
<td>Game Control</td>
</tr>
<tr>
<td>210-217</td>
<td>Expansion Unit</td>
</tr>
<tr>
<td>220-24F</td>
<td>Reserved</td>
</tr>
<tr>
<td>278-27F</td>
<td>Reserved</td>
</tr>
<tr>
<td>2F0-2F7</td>
<td>Reserved</td>
</tr>
<tr>
<td>2F8-2FF</td>
<td>Asynchronous Communication (Serial Port) (Secondary)</td>
</tr>
<tr>
<td>300-31F</td>
<td>Prototype Card</td>
</tr>
<tr>
<td>320-32F</td>
<td>Fixed (Hard) Disk</td>
</tr>
<tr>
<td>378-37F</td>
<td>Printer</td>
</tr>
<tr>
<td>380-38C</td>
<td>Synchronous Data Line Control (SDLC) Communications</td>
</tr>
<tr>
<td>380-38F</td>
<td>Binary Synchronous Communication (Secondary)</td>
</tr>
<tr>
<td>3A0-3A9</td>
<td>Binary Synchronous Communication (Primary)</td>
</tr>
<tr>
<td>3B0-3BF</td>
<td>IBM Monochrome Display and Printer (Thorne p.291,318)</td>
</tr>
<tr>
<td>3C0-3CF</td>
<td>Reserved</td>
</tr>
<tr>
<td>3D0-3DF</td>
<td>Color/Graphics</td>
</tr>
<tr>
<td>3E0-3F7</td>
<td>Diskette</td>
</tr>
<tr>
<td>3F8-3FF</td>
<td>Asynchronous Communications (Primary)</td>
</tr>
</tbody>
</table>

Often the Prototype Card area is the first choice for add-on interfaces, but any unused ports can be commandeered; the Fixed Disk area, or the SDLC Communications area are common choices. If you want to connect your interface directly to the I/O Channel (the card slots in computer) you will have to read about port address decoding in some book such as that by Royer. If you adopt our time saving method of interfacing to a general purpose I/O board you will find the board has switches that are set to select the port address. In the next lesson we will have an example of such addressing.
Debug and Ports

We can examine the contents of ports and set values in ports from Debug if we wish. (That is how I checked that the printer Busy line really was active low.) The instructions are IXXX which reads and displays the byte in port number XXX, and OXXX YY which writes the byte YY to the port number XXX. Unless you like to crash the computer and restart it, I would not recommend randomly writing to or reading from ports. Yes, you can even hang things up reading from some addresses.

INT XX versus IN XXX or OUT XXX

Often one has a choice of using an INT instruction or an IN or OUT instruction to do the same task. Which should you use? It is a question of priorities. The advantage of IN and OUT is speed; the bare bones action can be done faster. The advantages of INT are safety, portability, and coding speed. The INT instructions are set up to do things in the safest possible way, so you will crash less often using INT. Further, if your software is to run on different computers, you should be aware that the compatibility of machines increases as you go to higher levels. When different machines are put together their BIOSs and DOSs are sure to be designed to do the same thing with INT, but they may not be the same at the port level. Finally, the higher the level of the "language" in which you code, the more you can get written in a given time. For example, in the printer control program of p.294 of Thorne, a three line setup and INT 5H block of code can replace all the code between READ: and JMP READ. INT 17H can be used to send a byte to the printer, to initialize the printer, or to get the printer status, depending on the whether AH contains 0, 1, or 2.

General Interrupts

So far when we have talked of an interrupt we have meant the INT XX type of instruction. The fact that these are interrupts is another oddity of the IBM system. A good case can be made for calling these instructions "System Subroutine Calls". At a fixed point in your program you are branching to some subroutine that just happens to be part of the system software. Your program is not being "interrupted" in the sense of being taken away from its usual operation; it is doing just what it always does at that point. This behavior is the very nature of a subroutine.

The IBM system is unusual in that these "system subroutines" are accessed by essentially the same hardware as is used by a real asynchronous interrupt by some external instrument that wants the attention of the computer. In fact, you can simulate the call of your hardware with an INT instruction. Let us go through just what an interrupt is and how it is "serviced".

The 8088 processor has two interrupt pins. The NonMaskable Interrupt (NMI) pin is positive edge triggered, and is used for internal things such as requests from the 8087 Math Coprocessor chip. We will not worry about NMI interrupts in this introduction. The other pin is INTR, and is high level triggered. Interrupt pulses on this pin will cause an interrupt if the Interrupt Enable (IF) flag is set. This flag can be set by the STI
the interrupts are not latched on the INTR line. Thus if the line is both raised and lowered while the flag is cleared the interrupt will be missed. This behavior of the INTR is not really a problem for us since in the IBM PC we do not have access to this line.

The 8259A

In the IBM PC the INTR line comes from an 8259A Programmable Interrupt Controller (PIC). The 8088 can be interrupted by both internal and external sources. The internal source of interest to us is the INT instruction. For external interrupts there are eight input lines to the 8259A that come from the I/O Channel (remember that is what the circuit cards are plugged into). These lines are labeled IR0-IR7. There is a priority order for interrupts so that if two interrupts appear at the same time the machine can decide which of them to service first. The internal interrupts have higher priority than the external ones. Among the external lines IR0 has the highest priority and IR7 the lowest as the 8259A is initialized by the BIOS when the system is booted. The IR# lines are initialized to be edge (rather than level) triggered. Thus, if the interrupt line remains high past the end of the Interrupt Service Routine (ISR) (see below) a second interrupt is not generated.

It should be noted that all the 8000 series chips in the IBM PC are very versatile devices. When we discuss these chips we will usually be discussing their operation as constrained by the wiring of the IBM PC or the BIOS initialization. We simply do not have the time to discuss all their potential uses. For the most part we will even forgo the options that are available to change the configuration established by the BIOS initialization.

When an Interrupt Occurs

Now let us assume that an interrupt occurs; either an INT instruction is executed or one of the IR# lines is brought high. The hardware responds as follows. The current contents of the FLAG register, CS, and IP are placed in that order on the stack (Note: Royer has this wrong. Thorne p.296 is correct). This procedure is like putting a bookmark in a book when the telephone rings. It lets us find where we were before the interruption. The 8259A determines which interrupt has the higher priority if there is more than one, and for that interrupt determines an appropriate address in low memory. For the INT instruction the address is four times the interrupt number. Thus for INT 21H the address is 84H. For an external interrupt on line IR# the address is 20H + 4(#).H. If the interrupt were on line IR4 the address would be 20H + 4(4)H = 20H + 10H = 30H. We can see that because of the four multiplicative factor each different interrupt type has four bytes of its own in low memory.

The system knows that this address, which is called a "vector address", contains in its first word the IP value of the code which is now to be executed, the "Interrupt Service Routine". The second word of the vector address space contains the CS value for the interrupt service routine. So after an interrupt the return point for the program is safely stored on the stack and the machine is set loose running the interrupt service routine.
after an interrupt the return point for the program is safely stored on the stack and the machine is set loose running the interrupt service routine.

A How-to-do-it Guide to Interrupts

This interrupt procedure suggests several fairly obvious problems and some not so obvious problems. How did the correct memory locations get loaded in at the vector memory locations? For system interrupts DOS and the BIOS do this job when you boot the system. Since DOS does not know about your pet project it is up to you to load the vectors as part of your program. Clearly this loading must take place before there are any enabled interrupts. As usual there are many ways to do the job. On the next page is shown one way to load the vectors. Enough comments are given to give you a chance of understanding the loading process. We will assume, as an example, that we are using vector 140H, that is INT 50H. Only the parts of code having to do with the vectors are shown along with those “landmarks” which show where the code goes into the usual framework. Other parts of the code are indicated by dots.....
DSEG SEGMENT

OLD_VEC DW ?,? ;SET UP PLACE TO STORE THE OLD VECTOR

DSEG ENDS

CSEG SEGMENT

PROG PROC FAR

MOV AX, DSEG ;POINT DS TO DSEG FOR ACCESS TO OLD_VECs.
MOV DS, AX
MOV AX, 0 ;SET ES TO ZERO FOR ACCESS TO VECTOR SPACE.
MOV ES, AX

MOV BX, OFFSET OLD_VEC ;SAVE OLD VECTOR CONTENTS.
MOV AX, ES:[140H]
MOV [BX], AX
MOV AX, ES:[142H]
MOV [BX+2], AX

CLI ;DISABLE INTERRUPTS WHILE CHANGING VECTORS.
MOV WORD PTR ES:140H, OFFSET ISR ;LOAD VECTORS WITH THE ADDRESS OF
MOV ES:142H, SEG ISR ;ROUTINE WE HAVE LABELED "ISR".
STI ;REENABLE INTERRUPTS.

;PROGRAM GOES HERE.

MOV BX, OFFSET OLD_VEC ;NOW RETURN THE OLD INTERRUPT VECTORS.
CLI
MOV AX, [BX]
MOV ES:140H, AX
MOV AX, [BX+2]
MOV ES:142H, AX
STI

PROG ENDP

ISR: ;OUR INTERRUPT SERVICE ROUTINE.

CSEG ENDS
END

A different way to do the same thing would be to use DOS function (INT 21) number 35 to get the old values in the vector and DOS function number 25 to load the new vector contents. Function 25H takes care of the CLI and STI instructions.
DSEG SEGMENT

OLD_VEC_OFF DW ?
OLD_VEC_SEG DW ?
DSEG ENDS

CSEG SEGMENT

PROG PROC FAR

MOV AH,35H ;GET-VECTOR FUNCTION (#35H)
MOV AL,50H ; FOR INTERRUPT 50H.
INT 21H ;LET DOS DO IT.
MOV OLD_VEC_OFF,BX ;FUNCTION 35 RETURNS THE OFFSET IN BX AND
MOV OLD_VEC_SEG,ES ; THE SEGMENT VALUE IN ES. STORE AT OLD_VEC_OFF
 ; AND OLD_VEC_SEG. NOTE TWO DATA LABELS USED.

MOV AH,25H ;PUT-VECTOR FUNCTION (#25H)
MOV AL,50H ; FOR LOADING YOUR VECTOR FOR INTERRUPT 50H.
MOV DX,OFFSET ISR
MOV BX,CSEG ; THE ISR OFFSET IS NEEDED IN DX BY FUNCTION
MOV DS,BX ; 25H.
INT 21H ;LET DOS DO IT.
MOV AX,DSEG ;REMEMBER TO RESET DS TO DSEG
MOV DS,AX ; AFTER THE INT 21H.

;your program

LDS DX,DWORD PTR OLD_VEC_OFF ;FANCY WAY TO LOAD DS AND DX WITH THE
 ; TWO WORDS AT AND FOLLOWING THE
 ; ADDRESS OF OLD_VEC_OFF.

MOV AX,2550H ;LET DOS RESTORE THE ORIGINAL VECTORS
INT 21H ; TO AVOID A SYSTEM CRASH WHEN EXITING.

Restoring the old vector is very important. Even vectors that are not
being used by other programs or hardware need to be restored to avoid locking
up the system on the return to DOS.

Which Vector?

How do we pick the vectors for our interrupts? The answer depends on
whether we are concerned with a hardware or software interrupt. For software
interrupts we can take our pick of interrupts 40-7F with vectors 100-1FF, but
be careful! Are you already using one of these vectors? You may be. You may
have created your program with QEdit (or some similar program), and then
"shelled" out to DOS to run MASM and LINK and to test your program. QEdit is
still sitting in memory waiting for you to type EXIT, and thus return to
Q.EXE. Q.EXE may well have defined some software interrupts. If so, are they
being used at all while that program sits dormant? I do not know, but I would
worry a bit about it. Even more likely a problem is Blankany. We know it is
busy even when other programs are running. One way to proceed is to load all
the memory resident programs who intend to use, and then use DEBUG or RAMVIEW
to display memory from 0:100 to 0:17F with d0:100 (using DEBUG) and then look
at the rest of the allowed vectors with d0:180. I usually find three or four
interrupts (12 or 16 addresses) that are occupied with other than zeros.
Naturally, you want to pick a vector location that is not already occupied.

Which Vector for Hardware?

In the case of hardware interrupts your vector choices are very limited.
Hardware has available on the I/O Channel only the following interrupts and
associated IR# lines: A,IR2; B,IR3; C,IR4; D,IR5; E,IR6; and F,IR7. I will
assume that we, as beginners, do not want to get involved with the tricky
business of redirecting vectors so they point to our software, then having our
software do what we want done, and then passing the program back to the
original vector software. This procedure is often done, particularly with the
keyboard interrupts (which are not on the I/O Channel).

With the above exemption we programmers seem to lose the use of A
(color/graphics), B (Second Serial Interface), C (Primary Serial Interface),
D (Fixed Disk Storage), E (Diskette Storage), and F (Printer). That leaves
us with nothing! Note that Thorne's list on p.306 does not include the B and
D assignments. Of course, things really are not that bad. Chances are that
you do not have all the named devices connected to your computer so you can
use the freed interrupt. Remember on P.3 we mentioned that the vector address
of line IR# is 20H+4(#)H.

On commercial boards the vector number and associated interrupt priority
are typically selected by a jumper placed between various pins on the board.
(Example in the next chapter).

The Hardware Interrupt Unmasked

One of the unobvious problems with doing hardware interrupts results from
the structure of the 8259A and how the BIOS reacts to it. The 8259A contains
a one byte mask register. If bit number X of this register is set then
interrupts on line IRX are disabled (masked). When the system is booted the
BIOS checks to see what is attached to the I/O channel that is standard. If
the BIOS finds a printer it clears bit 7, if there is a floppy disk (diskette)
drive attached it clears bit 6, and so on. Unused (unrecognized) lines are
masked. The system was not built with your elegant interface specifically in
mind, therefore your interface will not be recognized, and your interrupt line
will be masked. It is your job to unmask it. Typically you will not want to
mask other lines with your instruction, so the example given by Thorne on
p.306 is not appropriate. Assume we wish to unmask IR2 (as we will in Lesson
6). Since the mask register is accessed through Port 21H the appropriate
instructions are

IN     AL,21H   ;GET THE CURRENT MASK.
AND    AL,1111011B ;CLEAR BIT 2 IN AL LEAVING OTHER BITS UNCHANGED.
OUT    21H,AL   ;SET NEW MASK.
Some Special Interrupt Presents from DOS

There are several invitations from DOS to use their special interrupts. These offers are in addition to the familiar INT XX. We will accept one of these offers in this week's lab. These interrupts vector to an interrupt service routine which is one instruction long, IRET, the return from interrupt instruction. We can easily redirect these interrupts to our interrupt service routines that first do our job and then end with the IRET instruction.

Interrupt number 4 is generated if an assembly language INTO (INTerrupt on Overflow) instruction is executed when the overflow flag is set to one. We can use this instruction after some calculation in our program if we are worried about a resulting overflow. The ISR for this interrupt, of course, does what we want done if there is an overflow.

Interrupt number 13H is generated if Ctrl-Break (the Break key is depressed while the Ctrl key is depressed) is entered. This interrupt gives us a way of interrupting our running program from the keyboard.

Interrupt number 1CH is generated every 55 msec by the 8253 timer chip. This interrupt is indirect in that the timer actually generates a number 8 interrupt which does several things, including keeping the DOS DATE and TIME values correct. At the end of the interrupt 8 interrupt service routine is the instruction INT 1CH. By directing interrupt 1CH away from its single IRET we can obtain an accurate timing signal for our own use. It is this interrupt that we will redirect in the fourth lab. NOTE THAT THIS CALLING OF ONE INTERRUPT FROM INSIDE THE ISR OF ANOTHER INTERRUPT IS NOT USUALLY POSSIBLE, SEE BELOW.

Another consideration with interrupt 1CH concerns references to the data segment from your interrupt service routine. Remember that interrupt number 8 called interrupt number 1CH, so while you are in your ISR interrupt number 8 has not yet been completed. Well, it turns out that interrupt number 8 uses DS. This means that when you get to your ISR DS no longer points to your data segment. You need to reestablish the address of the data segment in DS if you plan to write to or read from your data segment. Since we are not absolutely sure that interrupt number 8 is done with DS at the time it calls interrupt 1CH, we really should also PUSH DS at the beginning of the ISR and POP DS at the end.

Some Interrupt Service Routine Considerations

Push and Pop

Real (hardware) interrupts are characterized by the fact that they can occur at any point in our program. We can not count on being able to use any register in the computer without messing things up unless we take proper precautions. One of the first things we must do in an interrupt service routine is PUSH all the registers we are going to change. At the end of the service routine the last thing we do is POP those registers.
Do Not use INT in an ISR

One important difference between the IBM PC BIOS (the part of the operating system which is in ROM in the machine) and the IBM PC DOS (the part of the operating system which is loaded from disk into RAM) is that the BIOS is "reentrant", that is, it can call itself. For example, INT 8 can call INT 1CH as we saw above. DOS is not reentrant. It does not do its calling in a manner which saves and cleanly returns everything that it uses. There is a great temptation to use INT 21H to print things out to the screen from inside an ISR, but don't do it. Your will crash the system.

Do an STI Early in Your ISR

When an interrupt vectors to an ISR it does so with interrupts disabled. The 8259A disables further interrupts so that a higher level interrupt can not do things that are difficult to handle, like coming in when only half of the lower level address has been obtained from the vector address. Since the system may (indeed probably does) have urgent tasks to perform during your ISR, it is important to quickly reenable interrupts during your ISR. There is no problem with hardware interrupting other hardware such as there is with DOS interrupting DOS.

Allow for another Occurrence of Your Interrupt

The above STI allows the 8088 to be interrupted, but until you release the 8259A it can not respond to another interrupt at the priority of your interrupt or lower. The IRET (Interrupt Return) statement with which your ISR ends does not perform this release function. To enable the 8259A for your level of interrupt you must issue the proper Operational Command Word (OCW) to it, probably near the end of your ISR. The subject of OCWs (and Initialization Command Words, ICWs) is quite complicated; if you really want to master the 8259A read Royer and Eggebrecht. For our quick-and-dirty approach just issue the following pair of commands.

```
MOV AL, 20H ; 20H = BIT 5 SET = NONSPECIFIC END-OF-INTERRUPT
OUT 20H, AL ; OUTPUT OCW2 TO PORT 20H OF THE 8259A
```
dseg segment
dseg ends

sseg segment stack
dw 0100H dup(?)
sseg ends

cseg segment
assume ds:dseg, ss:sseg, cs:cseg

main proc far
mov ax, 0f000h
push ax
mov ax, 0fff0h
push ax
ret
main endp
cseg ends
end
dseg segment
old_int_seg dw 0
old_int_off dw 0
dseg ends

sseg segment stack
dw 0100H dup(?)
sseg ends

cseg segment
assume ds:dseg, ss:sseg, cs:cseg

main proc far
push ds
mov ax,0
push ax
mov ax,dseg
mov ds,ax
mov ax,0
mov es,ax
mov ax,es:[084h]
mov old_int_off,ax
mov ax, es:[086h]
mov old_int_seg, ax
cli
mov ax, old_int_off ; move int21h to int 0feh
mov es:[03f8h], ax
mov ax, old_int_seg
mov es:[03fa], ax
mov ax, offset myisr ; move our subroutine to int21h
mov es:[084h], ax
mov ax, cseg
mov es:[086h],ax
sti
mov bx,0

myloop: mov ah, 01
;just a simple ro
outine to
int 21h
inc bx
cmp bx, 100D
jne myloop
ret
main endp

myisr: sti
cmp ah,01
JNE call_isr
mov ah, 07
Trick.asm

int 0fleh
inc al
push dx
mov ah, 02
mov dl,al
int 0fleh
pop dx
mov ah, 01
iret

call_isr: int 0fleh ; use old int21h routine
iret

cseg ends
end