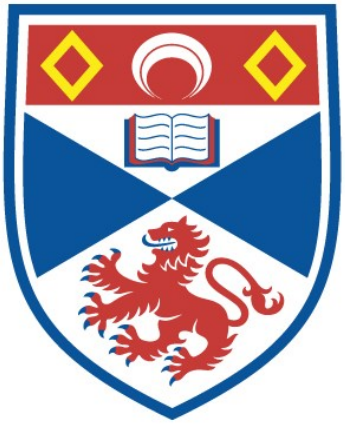
THE THEORY AND IMPLEMENTATION OF A SECURE SYSTEM

David S. S. Robb

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1992

THE
THEORY AND IMPLEMENTATION
OF A
SECURE SYSTEM

A thesis submitted for the degree of Doctor of Philosophy to

the University of St. Andrews

by

David S. S. Robb, BSc

December 1991

DECLARATIONS

I David Shepherd Stewart Robb hereby certify that this thesis has been composed by myself, that it is a record of my own work, and that it has not been accepted in partial or complete fulfilment of any other degree of professional qualification. I was admitted to the Faculty of Science of the University of St. Andrews under Ordinance General No 12 on October 1988 and as a candidate for the degree of PhD. on September 1989.

In submitting this thesis to the University of St. Andrews I wish access to it be subject to the following conditions : for a period of 5 years from the date of submission, the thesis shall be withheld for use.

I understand, however, that the title and abstract of the thesis will be published during this period of restricted access; and that after the expiry of this period the thesis will be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright in the work not being affected thereby, and a copy of the work may be made and supplied to any bona fide library or research worker.

D. S. S. Robb
December 1991

I hereby certify that the candidate has fullfilled the conditions of the Resolution and Regulations appropriate to the degree of Ph.D.

R. C. G. Killean
Research Supervisor
December 1991

Dedicated
to my Grandfather
A special man who I sadly miss

"It could be a science fiction nightmare come to life"

Vin McLellan
New York Times January 31 1988

"Be not overcome of evil, but overcome evil with good."
Romans 12:21

ABSTRACT

Computer viruses pose a very real threat to this technological age. As our dependence on computers increases so does the incidence of computer virus infection. Like their biological counterparts, complete eradication is virtually impossible. Thus all computer viruses which have been injected into the public domain still exist. This coupled with the fact that new viruses are being discovered every day is resulting in a massive escalation of computer virus incidence.

Computer viruses covertly enter the system and systematically take control, corrupt and destroy. New viruses appear each day that circumvent current means of detection, entering the most secure of systems. Anti-Virus software writers find themselves fighting a battle they cannot win : for every hole that is plugged, another leak appears.

Presented in this thesis is both method and apparatus for an Anti-Virus System which provides a solution to this serious problem. It prevents the corruption, or destruction of data, by a computer virus or other hostile program, within a computer system. The Anti-Virus System explained in this thesis will guarantee system integrity and virus containment for any given system. Unlike other anti-virus techniques, security can be guaranteed, as at no point can a virus circumvent, or corrupt the action of the Anti-Virus System presented. It requires no hardware modification of the computer or the hard disk, nor software modification of the computer's operating system. Whilst being largely transparent to the user, the System guarantees total protection against the spread of current and future viruses.

CONTENTS

CHAPTER 1

COMPUTER VIRUSES
AN
INTRODUCTION

1.0 : DEFINING A COMPUTER VIRUS

A Computer Virus is a self replicating section of code which attaches itself to a program or an operating system. When the infected program is subsequently executed, the virus within it activates. Once activated it either continues in its course of self-replication or is triggered by some means into executing its main functional routine. These routines vary from the benign, such as a message displayed on the screen asking for world peace, to vicious attacks on either specific files or the computer system in general [1].

Adleman [2] provides us with a more rigourous definition :

A virus is a map from programs to 'infected' programs

where an infected program on each 'input' makes one of three choices :

[1] Injure : Ignore the intended task and compute another function chosen by the virus.

[2] Infect : Infect other programs, then perform the 'intended' task.

[3] Imitate: Neither injure nor infect : preform the 'intended' task without modification.

where :

'input' is considered as all accessible information such as : user input,
system clock or files, and

'intended' task is the one the uninfected program would have performed for that input.

1.1 : CATEGORISING VIRUSES

Al-Dossary [3] provides us with a series of definitions that can be used to categorise computer viruses, and these are presented in the following sections.

1.1.0 GENERAL CONTAGION AGENT MECHANISM (GCAM)

A virus in this category attaches itself to any file which is capable of perpetuating the existence of the virus.

Most viruses mentioned in this work are of this type.

1.1.1 SPECIFIC CONTAGION AGENT MECHANISM (SCAM)

A virus in this category attaches itself to a specific program, set of programs, or type of program.

This type of virus will spread slower than a GCAM virus as less files will be infected. It can, however, be made less detectable as the virus programmer will use his intimate knowledge of the target program to further limit the chance of detection.

1.1.2 GENERAL TARGET ACTION OR RESULT (GTAR)

A virus of this category will, when 'triggered', attack indiscriminately. The action it takes and the result of this action will not be dependent on the type of data, drive or terminal. Examples of this include : reformatting the hard disk ; corrupting the FAT tables (tables which DOS uses to keep track of which disk sectors belong to which file) ; or scrambling ascii characters on the screen.

Since the actions which can be taken by this category are limited, and it is likely that these actions can be generalised, detection and removal is more likely than a STAR virus.

1.1.3 SPECIFIC TARGET ACTION OR RESULT (STAR)

A virus in this category attacks a specific target. It is also likely that a virus of this type will have a specific purpose. Examples might include alteration/corruption of accounts data or attacks on anti-virus programs.

This type of virus may prove hard to detect : it may complete its task and destroy itself upon completion. Anti-virus programs checking for general virus behaviour are unlikely to detect a virus of this type unless it checks for it specifically.

An example of a STAR virus is the 2000 byte variant of the Dark Avenger DOS virus written in Bulgaria [4]. If the string "(c) 1989 by Vasselin Bontchev" is found in any executed file, the virus hangs the machine making it seem as if the Bontchev program was the cause. Vasselin Bontchev is a Bulgarian Anti-virus software writer. This is a deliberate attack on a specific piece of software hence it comes into the STAR category.

Using these categories we can split viruses into one of the following combinations :

GCAM-GTAR, GCAM-STAR, SCAM-GTAR, or SCAM-STAR.

In actual fact some more complex viruses may be categorised as both GTAR and STAR as the action the virus takes may be largely general but may take a more specific action if certain criteria are met.

1.2 : ANATOMY OF A VIRUS

1.2.0 PROPAGATION MECHANISMS

Viruses can be further categorised by the techniques they use to enter and infect the system. There are two main types of DOS virus:

[1] Boot Sector virus

Within the Boot Sector of a disk resides a small section of code known as bootstrap code, which is executed when the machine attempts to boot from the disk, and is responsible for loading the operating system into memory and executing it.

A Boot Sector virus [5] modifies the code within the boot sector of a disk. The virus usually replaces the bootstrap code with virus code which will load the virus and then run the original bootstrap code which the virus has relocated. The virus is therefore memory resident and active before DOS is even loaded.

On a formatted floppy disk a single boot sector exists at logical sector 0. This sector will contain code whether the disk has been formatted as a boot disk or not. If it has, a bootstrap loader for the operating system will be present within this sector. If not, then it will contain code to print 'non-system disk' on the screen to inform the user of his error. Therefore both boot and non-boot disks are susceptible to infection as both contain code that would be executed if an attempt was made to boot from the floppy disk.

A hard disk contains two types of boot sector. The DISK Boot Sector exists within the first physical sector of the disk and contains the partition table which includes information as to the location and size of all partitions on the disk. It also contains all the code necessary to check the partition table for a bootable partition, load the PARTITION boot sector and transfer control to it. The PARTITION Boot Sector exists within the first logical sector of a partition and contains all code required to bootstrap the operating system files with that partition. Either of these boot sectors are susceptible to infection.

During the Power On Self Test (POST) sequence which is executed before the operating system is loaded, a table is created at the base of RAM, and filled, which contains the JUMP addresses to the start of all software interrupt routines. This is referred to as the Interrupt Vector Table. An infected boot sector will load the virus into memory and alter an address in the interrupt vector table so that it addresses the start of the virus in memory. This means that the virus will be activated for every CALL to that interrupt. The original address, replaced within the Interrupt Vector Table, is transferred to the virus so that the virus can transfer control to the requested interrupt routine when it is finished. Most viruses intercept INT 13H so that they become active when any application attempts disk access. Once the virus is loaded the original boot sector is usually loaded into memory and executed.

When the virus intercepts an INT 13H request, it will check the disk to which the request is made for infection. If the disk is not infected then the virus will infect the disk. Thus infection will occur during the first access of a new floppy disk.

One of the very first DOS viruses was of this type, the BRAIN virus [6,7,8], which infects the floppy disk bootstrap sector. Another such virus is the ITALIAN 'Bouncing Ball' virus [9] which infects both the floppy disk bootstrap sector and the hard disk partition bootstrap sector.

This type of virus should be considered as a special incidence of the SCAM category of viruses as it infects only one specific piece of code, the bootstrap sector, or sectors. No other executable code is infected within the system. A virus of this type may be either GTAR or STAR.

[2] Parasitic virus

A Parasitic virus [5] infects executable files within the system, usually by prepending or appending their code to the code section of the program being infected.

The most common way that a virus integrates itself into the file being infected is to alter a Jump instruction, usually the initial JMP, within the original code to jump to the beginning of the virus code.The virus, once executed, in turn jumps back to what would have been the destination of the jump it altered.

Clearly there are other means but the effect is usually the same. One brute force method involves the overwriting of the beginning of the file thus rendering the original program inoperable. This, although simple to implement, ensures that it is detected early on and reduces the spread.

This type of virus can be further divided into two distinct groups:

[i] Resident

When a virus of the Resident Parasitic type is activated it loads itself as a memory resident piece of code. Once resident it will remain there until the machine is switched off. It is common for a virus of this type to infect the COMMAND.COM which is executed as part of DOS loading sequence. This would cause the virus to be resident once the machine has booted. Advantages of this method include the ability to introduce STEALTH techniques, which are explained later. Being resident also means that any further calls to OPEN, CLOSE, CREATE or EXECUTE a file can be a source of infection. For example the command :

COPY A.COM B.COM

could result in both files being infected even if neither were before the command's first execution on the computer.

[ii] Non-resident

A Non-resident Parasitic virus will spread during the execution of the program to which it is attached. It will infect a file, or set of files, by searching the directory for suitable candidates. Once a candidate is found the virus will infect it and mark it in some way so that it does not attempt to infect it again. If a virus does not mark infected files, then it may repeatedly infect a file, thus alerting a user due to significant increases in file size and increased likelihood of malfunction of the file when executed.

An example of this type is the DOS VIENNA virus [9] : each time it is activated it looks for a single file to infect in either the current directory or any directory mentioned in the PATH. The majority of files are simply infected, but an occasional file is overwritten with code at the start which causes a jump to the BIOS initialisation routine (a warm start).

A third type of virus has emerged recently which fail to fall into the Parasitic virus category because they do not alter the original file in any way :

[3] The Companion Virus

A Companion virus [10] may be considered by some as a special case of the Parasitic type. It makes use of a special feature of the MS-DOS operating system : namely, if two executable files with the same filename exist, but one has a .EXE extension and the other a .COM, then the .COM program executes first. An infecting virus will search for an .EXE file and create a .COM file of the same name with its own virus code within. It places the new .COM file in the same directory as the .EXE and makes it hidden. When the user tries to execute the .EXE, the .COM file executes first : the virus then activates, runs to completion, and then runs the .EXE. The user is entirely unaware that this has happened and the virus has been activated without any changes to the .EXE file. As the .COM file is hidden and the size of the .EXE has clearly not been changed, the virus can not be detected by the DIR command. Utilities such as CHKDSK would however identify an increased number of hidden files.

1.2.1 A FUNCTIONAL DESCRIPTION

(i) Non-resident Parasitic Virus

The functional behaviour of such a virus can best be outlined by the following pseudo-code :

```
program virus

function continue.to.infect : boolean
begin
{Dependent on result will attempt infect one or many files}
end

procedure infect
begin
  find.a.file
  if no.file exit_procedure
  if marked.infected  then infect
                      else begin
                             mark.infected
                             append.virus.to.file
                           end
end

function trigger : boolean
begin
{Check if time to Trigger & return result}
end

procedure trigger.event
begin
{Execute the trigger sequence}
end


MAIN
begin
  while continue.to.infect do infect
  if trigger do trigger.event
  jump_to file.start
end.
```

This pseudo-code applies to a parasitic virus which does not go memory resident. The contents of the function `continue.to.infect` will define whether the virus infects only one, many, or all uninfected files. The procedure `infect` will search for a file and, if it is not infected, it will infect it, then exit.

If it is infected it will search for another file and repeat. If no file is found then the procedure exits. The search for a suitable file is done by `find.a.file` and the criteria it uses will decide whether the virus is a SCAM or GCAM virus. The function `trigger` decides whether it is time to execute the viruses action sequence. The criteria for triggering is covered later. The code placed in the procedure `trigger.event` will determine whether the virus is a STAR or GTAR virus. Once all this is done the virus will then jump to the start of the code for the program it is attached to.

(ii) Resident Parasitic Virus

The functional behaviour of such a virus can best be outlined by the following pseudo-code :

```
program virus2

procedure membit

     function trigger : boolean
     begin
     {Check if time to Trigger & return result}
     end

     procedure trigger.event
     begin
     {Do the trigger sequence}
     end

     procedure infect.file
     begin
       if not(marked.infected) do
       begin
         mark.infected
         append.virus.to.file
       end
     end

begin
  if trigger then trigger.event
             else infect.file
   return.control
end
```

```
procedure make.memory.resident
begin
  generate.space
  place.procedure.membit.in.space
  re-route.interrupts
end

MAIN
begin
  if memory.resident
    then do.nothing
    else make.memory.resident
  jump_to file.start
end
```

This type of virus, when executed from within an executable file, checks if it exists in memory with the function `memory.resident`, and if it does not, then loads the procedure `membit` into memory as a TSR (terminate and stay resident) code using the procedure `make.memory.resident`. Once this task is complete the virus will return control to the executable file to which it is attached.

The procedure `re-route.interrupts` alters the addresses within the software interrupt table to point to the virus code which has been placed in memory. The original address is stored within the virus code so that the virus can pass back control to the function that was originally being called using `return.control`. Therefore every time the software interrupt is made, the virus in memory will be activated.

Once activated it will test for a trigger event using the function `trigger` and execute the trigger sequence in `trigger.event` if the test is positive. As for non-memory resident viruses the code placed in the procedure `trigger.event` will determine whether the virus is of type STAR or GTAR. If the virus does not trigger then it attempts its other function, that of propagation with `infect.file`.

(iii) Boot Sector Virus

The functional behaviour of such a virus can best be outlined by the following pseudo-code :

```
program virus3

procedure membit
    function trigger : boolean
    begin {Check if time to Trigger & return result}
    end

    procedure trigger.event
    begin {Do the trigger sequence}
    end

    procedure infect.floppy
    begin
      if not(marked.infected) do
      begin
         mark.infected
         add.virus.to.boot
      end
    end

    procedure infect.hard.drive
    begin
       if not(marked.infected.hard) do
       begin
          mark.infected.hard
          add.virus.to.boot.hard
       end
    end

begin
  if trigger
      then trigger.event
      else if hard.drive then infect.hard.drive
                         else infect.floppy
   return.control
end

procedure make.memory.resident
begin
  generate.space
  place.procedure.membit.in.space
  re-route.interrupts
end

MAIN
begin
  if memory.resident    then do.nothing
                        else make.memory.resident
  jump_to file.start
end
```

This type of virus is similar, functionally, to the memory resident parasitic type of virus. It differs in that it searches for disks to infect, not files. When executed, during a boot, it goes TSR (terminate and stay ready), as explained in the previous section. Once in memory, the virus will be activated on disk access software interrupts. If the disk to which the request refers is not infected, the virus will infect it using `infect.hard.drive` or `infect.floppy` before passing control to the interrupt that was originally called by the application with `return.control`.

1.2.2 A STRUCTURAL DESCRIPTION

In this section we will consider the structure of infection within programs. In MS-DOS there are two distinct types of executable file and therefore they are covered separately. In MACOS this is not the case and therefore the structure can be generalised. Finally boot sector viruses shall be treated as a special case [7].

[1] Parasitic Virus - MS-DOS (.COM) executable files [11]

A .COM file contains just binary code and is therefore the simplest to infect. This type of executable whilst being compact and fast to load, is limited to 64K Bytes for the combination of program code, stack and data.

Three groups can be distinguished : overwriting, appending and prepending.

The first viruses to overwrite used a primitive version of this method due to its simplicity [12]. The virus would simply overwrite the file from the start. This rendered the original .COM file entirely inoperable. The rate of infection was

exceedingly low as detection was immediate. More advanced viruses which use this method store the section of the original file, which they overwrite, at the end of the file. The virus can then restore the original code after the infection routine has been executed.

Appending viruses are the most common. They overwrite the first few bytes of the file with a jump to the start of the virus code. The virus code is then appended to the end of the original file. The original values of the bytes overwritten at the start are placed somewhere within the body of the virus.

Prepending viruses place their code at the start of the file and shunt the original code along to make room. As this is slightly more complex than appending, viruses of this type are less common.

[2] Parasitic Virus - MS-DOS (.EXE) executable files [13]

Executable files of type .EXE are not pure binary code like .COM files. The code within a .EXE is prepended by a header which contains information which permits the program to use the full capabilities of the 80x86 series of microprocessors. Unlike .COM files, .EXE files may contain multiple segments, each 64K Bytes in length. The first two bytes of the header contain a signature which identifies the program, to MS-DOS, as an .EXE file. The following entries in the header include : the number of 64K Byte pages the file occupies ; the number of bytes occupied in the last page ; size of the header ; minimum space required in addition to the space required for the program image in order to run ; the maximum space required - the value the program requires in order to run most efficiently ; the initial values to be loaded into the 80x86 registers ; a checksum field (unused) ; and finally a relocation

table to facilitate relocation (entries are pointers to addresses requiring the segment start address added to them). Infection of .EXE files requires that this header be modified. Some modify more than others but they all replace the code segment instruction pointer, CS:IP, initial register values to pass control to the virus, and store the original so that control can be passed back when the virus has completed its task. Some viruses also alter the system segment stack pointer, SS:SP, initial register values to provide an internal stack whilst the virus executes. This prevents the original stack from possibly overwriting the virus code.

[3] Parasitic Virus - MACOS executable files [14]

The MACintosh Operating System structures its executables as a series of resources which exist as objects pointed to from the resource fork within the application. When first executed the program uses the jump table in the resource CODE 0 to find the address of the initialisation routine for the application. Introducing a virus is simple : a new code resource is added and the jump table in CODE 0 altered to point to the new virus resource. The original jump value is stored so that control can be returned.

Infection of the System File can often be made more potent by the inserting the virus as an INIT resource, as each INIT segment is executed at boot time.

[4] Boot Sector Virus [15]

A Boot Sector Virus replaces the original boot sector with virus code which first loads the virus into memory, and then loads and executes the original boot

sector. Viruses of this type rarely fit within the single sector which they are replacing. As most viruses continue normal operation after the virus has executed and been loaded into memory, the original boot sector must be stored somewhere as well as any virus code 'overspill'. Many different solutions have appeared to this problem :

(i) Mark desired clusters as 'BAD' so that the operating system will not use them. With this method the partition remains the size desired, but the available space is reduced by the number of sectors thus marked [9].

(ii) Move the partition boundaries to accommodate the extra code at the end of the partition. With this method the partition effectively shrinks by the number of required sectors. This method is only usually adopted for floppy disks as hard disks may have multiple contiguous partitions where making boundary changes would be more complex.

(iii) Use sectors near the end of the disk, without allocation or moving of partition boundaries, as the end of the disk will only be used when the disk is nearly full. As space is not allocated locating, and therefore detecting, the virus is made more difficult. The drawback is that the virus may be overwritten.

(iv) Use the remainder of track zero. This space is rarely used for anything but the boot sector. The remaining sectors are usually free. Again allocation is not required so the virus will not be detected by CHKDSK, for example. However it is a common place for viruses to reside and such a small area of the disk can easily be checked.

(v) Small viruses may use the last sector of the root directory, if only one extra sector is required. This will only be overwritten if the directory is nearly full which, for floppy disks in particular, is unlikely.

(vi) It is possible to format an extra track on a Floppy disk above those formatted by the operating system. This is ideal in that no allocation is required, and no overwriting is possible. Detection is also unlikely [16].

(vii) On many hard disks the last physical track, the outer one, is used for parking the heads and, although available, is never used. A virus could also be placed here without allocation.

All the methods outlined above are fallible. It is possible to detect a virus which uses any one of the methods above. However, no anti-virus programs to date check all these locations, and if they did virus writers would find new 'hide-outs' for their code.

1.2.3 TRIGGER MECHANISMS

The event which will cause a virus to trigger is usually based on one of two criteria.The virus may be programmed to trigger at or after a specific time or date. For example it may activate on April 1st and display a message. Alternatively it may trigger when a count is reached, for example after 10 executions of the infected file, or after the infected file has infected 20 other files.

Viruses of the STAR category may however trigger on a more specific basis. For example if the virus is designed to corrupt data files of a certain database, it may trigger on the existence of the database application.

1.2.4 TRIGGER ACTION SEQUENCES

The action a virus may take on triggering cannot be defined. As mentioned before the action may be categorised into either specific or general : GTAR or STAR. The action itself could be anything ranging from the innocuous to the malicious. Listed below are some examples that have been found in real viruses :

8Tunes [17]	Chooses a tune from 8 possible tunes and plays it.
AIDS [17]	Prints "Your computer now has AIDS" on the display and halts machine.
Armagedon [17]	Attempts to dial the speaking clock in Crete if a Hayes-Compatible modem is installed.
Datacrime [7,9,18]	First 9 tracks of the hard disk is low level formatted. If the drive signature is within this area then the drive will not even be recognised by the system and may have to be returned to the manufacturer. Also if partitions exist these will disappear.
Dyslexia [17]	Occasionally transposes two adjacent characters on the screen.
Fu Manchu [9,18]	Checks keyboard buffer for occurrence of politicians names and appends derogatory comments to them.
Jerusalem [7,9,18]	A section of the screen is scrolled up two lines leaving a black window. A time wasting loop is also inserted.

1.2.5 MARKING AN INFECTED FILE

There are many ways by which a virus can ascertain whether a file is already infected. The most obvious method would be for the virus to check for the occurrence of a certain byte sequence within the sector of a file where the virus would exist if the file was infected. If this sequence is found then the file is infected. Other methods have appeared, two of which are listed below :

[1] The addition of 100 to the Year stamp of the file. As the DIR command only displays the last two digits 2091 is displayed the same as 1991 and therefore the user is unaware of the alteration [19].

[2] Similarly the placement of a number greater than 60 in the seconds field of the time stamp [20]. This requires that stealth functions (explained later) return this value to less than 60 so that for example the DIR command does not alert the user.

1.2.6 TRANSMISSION MEDIUMS

[1] Boot Sector Viruses

This type of virus requires the interchange of physical disks to propagate. Under MS-DOS, these discs need not be formatted as system disks, as explained earlier in section 1.2.0, infections may exist within the 'boot' sector of non-system disks. Infection occurs when attempting a bootstrap from an infected floppy or removable hard disk.

[2] Parasitic Viruses

This type of virus is not constrained to spreading via physical medium. Although propagation via infected disc is common, so is downloading software from bulletin boards via a modem link. Networks are also an ideal medium for propagation. Many viruses are capable of infection from file server to individual PC or vice-versa.

1.2.7 FINDING SPACE

Memory Space has to be found for all viruses which go memory resident. These include Memory Resident Parasitic Viruses and Boot Sector Viruses. There are various means of doing this :

[1] Locate at the top of memory and reduce the space available to MS-DOS. This is done by altering the value held at location 0040:0013 in memory. This means that DOS has 640K bytes available to it minus the length of the virus. This is the favoured mechanism for Boot sector viruses. This allocation can be detected by utilities such as CHKDSK or AntiVirus programs [16,19].

[2] Allocation above 640K in memory. This will involve the virus searching for suitable free RAM space. The likelihood of detection is not high.

[3] Use the MS-DOS TSR functions to place the virus code as resident so that it is not overwritten. This is the simplest method but is easily detected by AntiVirus monitors [21].

[4] Use the first Disk I/O buffer used by MS-DOS and remove it from the I/O chain. These buffers are 512 bytes long which is often enough for a virus. Being an unusual approach this is unlikely to be detected [20,22].

1.3 : FIRST GENERATION COMPUTER VIRUSES

The first, purely experimental virus was injected into a contained environment at the University of Southern California in October 1983. It was written by Fred Cohen, to be run on a VAX 11/750 running UNIX. It was capable of taking over the entire system in less than 30 minutes. File to file infection took much less than a second and was therefore not detectable by a user. It was written purely as a demonstration and never allowed to spread beyond the system mentioned above.

Early examples of computer viruses released into the public domain were probably written as irresponsible jokes. It was late 1987 when the first DOS viruses appeared. First was the BRAIN virus, closely followed by the LEHIGH and JERUSALEM viruses.

BRAIN [6,7,8] is a boot sector virus which attached itself to 360K floppy drives only. It claims both the boot sector, and a further 3 clusters (6 sectors) which it marks as 'BAD' in the FAT. It stores the original boot sector within the extra sectors it allocates itself, which it calls once the virus is installed. It occupies 7k of memory when resident. It gets its name because each infected disk has its label changed to '(c) Brain'. This was the first recorded DOS virus and was also the first to exhibit 'stealth' capabilities (which are explained later) : The virus intercepted any requests to read the boot sector and redirected them to the original sector if they occurred. Thus the virus hid itself from the user. Its source was Lahore in India and its range went as far as Delaware and Washington Universities in the USA, a Midlands University, a Leicester consultancy, and a major Insurance company on the South Coast. This virus is a SCAM type as it infects only boot sectors.

LEHIGH [1,7,23] is a parasitic virus which becomes memory resident. Unlike BRAIN, this virus is far from benign, it trashes a disk after it has been infected four times by overwriting the first thirty two sectors of the disk. It infected several hundred machines causing severe loss of data before it was detected, and several hundred more were infected before a cure was found. It is estimated that it infected 600 disks within just two days. It originated in Lehigh University in the USA, hence its name, and the damage it caused was fairly centralised. The reason for this was the virus triggered after infecting only four files. This meant that detection was quick : it is estimated that if this number was much higher, then tens of thousands of machines world-wide would have been affected. This is a SCAM-GTAR virus as it only infected a specific file, the COMMAND.COM, yet its target action is general.

JERUSALEM [7,24] is a memory resident, parasitic virus. It infects all .EXE and .COM files in the system upon execution. After thirty minutes the virus triggers and the system slows down by a factor of five and an area of the screen from row 5 column 5 to row 16 column 16 scrolls up two lines. All programs executed on Friday the thirteenth are also deleted. It is estimated that at its peak 10 000 to 20 000 machines were infected. This virus is of the GCAM-GTAR type.

These viruses were simple in design and required only limited skill to write. They were simple to disassemble and analyse. Detection and removal of these viruses was also reasonably simple. However this was not the case for very long.

1.4 : NEW GENERATION COMPUTER VIRUSES

Over 400 DOS viruses exist in the world today and this is increasing at the rate of over 1 per day [25]. Other types of machines such as those running MACOS are similarly affected, although the number in existence and the rate of growth are proportionably smaller than for the PC. Reasons for this probably include the simple fact that there are far more PCs in the world today and that they have been in existence longer. DOS is also a far more open and documented operating system than its Macintosh counterpart. Mainframe viruses such as VAX and SUN have also been suffering from infections but these machines have proved to be more immune from attack, at least for the moment.

Analysis of the viruses that are currently appearing indicate that virus infection is, and is going to continue to be, a very real problem within the computing community. As a steady flow of new viruses are released day by day it becomes obvious that a group of highly motivated individuals, or even companies, from all around the world are actively developing virus code.

Eradication of these infections is becoming more and more necessary as they increase in complexity, sophistication, and of course number.

1.4.0 FEATURES FOUND IN NEW GENERATION VIRUSES

New viruses are appearing with code which is self-protecting, self-encrypting, self-modifying and self-correcting. This makes discovery and removal increasingly hard for the user, and indeed the anti-virus software programmer. One virus programmer stated the following about his virus MOTHERFISH or WHALE :

> *".... the Motherfish is not just a virus, it is a virtual living, breathing entity that is capable of teaching itself its pursuers techniques and then increasing its code level sophistication as its environment becomes increasingly hostile..."* [26]

These claims are largely exaggerated but they clearly indicate the arrogance and pride that the programmer has for his creation. It should be noted however that as yet there is no scanning program which is capable of identifying this virus, giving considerable concern to the anti-virus software developers.

Below are descriptions of techniques which have appeared in the more recent viruses :

[1] Self-Encryption

Viruses are appearing which have a pseudo-random encryption routine [26,27] so that each new copy of the virus would consist of an entirely different series of bytes. This makes the code unrecognisable. Such viruses have been coined "Armoured" viruses.

Obviously the decryption routine would have to be present for the virus to be decrypted before execution, and this routine could not itself be encrypted. Other techniques deal with this problem, but even without them encryption complicates disassembly and analysis for the anti-virus software developers.

[2] Self-Modifying code

Viruses such as the 1260 [28] (named after its size in bytes) use self-encryption and they also make the decryption routine self-modifying such that the largest sequence of unchanging bytes is 3. This makes searching for recognisable patterns impossible. This type of virus is often referred to as a 'Chameleon' virus.

[3] Self-Correcting code

Viruses such as the VACSINA [4] (Bulgarian for vaccine) have, incorporated within them, self-correcting Hamming code such that any alteration to the virus of up to 16 bytes will have no effect as the virus will 'repair' itself [22]. This reduces the probability of mutations appearing due to hacking by second parties.

[4] Clandestine Memory Allocation

Viruses of the 'parasitic-resident' type require a copy of themselves to remain in memory. New viruses are appearing which avoid the obvious methods of doing this so that they are more difficult to detect [22]. Examples include :

[i] EDDIE II [4] directly manipulates the memory control blocks.

[ii] 666 [20] (The Number of The Beast virus) uses the first DOS disk buffer (after removing it from the 'available' list).

[5] Surviving the High Level Format

Boot sector viruses such as the NEW ZEALAND [17] are not destroyed by the DOS FORMAT instruction. In order to remove this virus one must do a low level format of the disk in question. This format process overwrites the Master Boot Sector which contains the virus.

[6] Surviving a Warm Boot

The JOSHI virus [16] re-vectors the INT 9H DOS software interrupt to trap the <ctrl><alt><del> (warm boot) key sequence to ensure that it manages to remain intact during the subsequent boot sequence.

[7] The Anti-Virus Virus

The YANKEE virus [29] contains code to search for the viruses ITALIAN [9] and CASCADE [9]. On finding either of these, YANKEE will remove it : Write a virus to catch a virus!

The programmer's reason for this insertion seems to be to remove older generation viruses which are 'clogging' up the system and replace them with a new 'improved' one. As virus infection spreads this may prove to be a required feature of a virus : they will be so prolific that they literally have to fight each other for control.

[8] The Anti- Anti-Virus Virus

As mentioned previously the 2000 byte variant of the DARK AVENGER virus [4] specifically targets Anti-Virus software produced by a software developer in Bulgaria [22]. This may be only the beginning : viruses could be designed to disable, by-pass or destroy such software. A war has been waged between virus writers and the anti-virus software writers and there is no reason to suppose the virus writers won't play dirty!

[9] Multi-Partite Viruses

Viruses are appearing which are able to spread by infecting both the boot sector and program files [10]. This type of virus, if loaded into memory from an infected boot sector, will infect programs as they are executed. If, however, the virus is activated by the execution of an infected program then it checks if the virus is already active in memory. If it is not then the virus firstly installs itself into memory and then attempts infection of the boot sector. The FLIP MS-DOS virus [30] is an example of this type.

[10] Random Insertion Virus

It is expected that viruses will either append themselves to the beginning or to the end of a file it is infecting. An IBM virus exists which chooses a pseudo-random point, by using the machine's timer, as an offset within the file to place its code. This makes scanning particularly, difficult as the offset for the signature is different each time. Anti-Virus scanning software is explained in section 1.7.0.

[11] No increase in Filesize after Infection

One problem the virus writer had to solve in order to avoid his virus from being detected was that a virus requires disk space to store its code. This space must be safe from overwriting by the operating system and yet not appear obvious by examination of the directory entries [22]. In general virus writers attempted to keep their viruses as small as possible to reduce the likelihood of detection but this was not sufficient against checking by anti-virus programs.

The '666 - Number of the Beast' MS-DOS virus [20] manages to attach itself to a .COM file with no increase in length whatsoever. It does this by finding files which are between 512 bytes and 64K bytes long with at least one free sector between the end of the file and the end of the last allocated sector. The reason this is possible is that, under MS-DOS, all disk space is allocated in multiples of fixed quanta known as clusters. These clusters are a whole number of sectors, the number depending on the size of disk or disk partition. Sectors themselves are 512 bytes in length, although this value is theoretically variable. For example if a file is 300 bytes long and the cluster size for the disk is 2 sectors then the file occupies 2 sectors on disk even though the latter part of the first sector and the whole of the second sector are empty. Thus any user or anti-virus program will fail to detect the virus unless the actual content of files is inspected. The cluster size is dependent on the size of disk partition : the larger the partition the greater the cluster size. Therefore the larger the partition, the greater the number of files there will be which do not occupy their full allocation of space and can be infected. Also, with increased cluster size, small files will leave larger areas of unused space allowing larger viruses to make use of this method.

[12] 'Stealth' Techniques

As anti-virus programs became more sophisticated and more abundant, some virus writers retaliated with viruses which went to considerable length to evade detection and the result is the 'stealth' virus [10,26].

A virus can often be detected by viewing the contents of the infected file. Memory resident viruses have found a way around this problem. If a virus is resident in memory then it can intercept a request to view an infected sector and replace it with a request to elsewhere where the virus has carefully saved the original uninfected sector.

An example of this is the 4K Virus [19], another MS-DOS 'new generation' virus, which intercepts the INT 21 function 14H (sequential read), function 21H (random record read) and function 27H (random block read). If the request corresponds to a read request from the start of an infected file where the virus is stored, then the virus returns the original contents as they were before infection. The Read function (3FH) operates in much the same way : the virus disinfects the file as it is read thus it appears unmodified. The Write function (40H) is also altered so that when writing to an infected file it is disinfected, only to be re-infected on Closing. This virus also uses an alternative method to avoid the sizes of infected files changing and alerting the user. This method works only when the virus is memory resident : all functions which return the file size, such as function 23H (File size), function 11H (find first matching file) and function 4EH (find first), have their file size decremented by 4K, the size of the virus. Also function 40H (Lseek) is intercepted so that when a program seeks to the end of an infected file, the reported value of the file pointer is decremented by 4k.

The MS-DOS boot sector virus JOSHI [16] also adopts these techniques. It intercepts any attempts to read, write or verify the Master Boot Sector and redirects them to the original uninfected version.

To summarise, if you are infected by a stealth virus then it may circumvent many of the Anti-Virus products available today. As the data within a file, once read into memory, will be identical whether the disk version is infected or not, many programs will fail to identify the viruses presence. If the system is booted from a virus free system then, as the virus will not be resident in memory many of the stealth features will not be active and detection is more likely.

[13] Disabling User Alerts

Another way to prevent Anti-virus programs from informing the user of a virus is implemented in the 666 virus [20] : it re-vectors the INT 24H which is the Fatal Error Handler. The result of this is that any error reporting is disabled. The virus may then attempt to infect a floppy disk, for example, and should no disk be present, the user would not be informed of the attempt, unless the hardware disk activity light was noticed.

[14] Inter-Virus Co-operation

Viruses exist which contain knowledge of other viruses and their mutual existence may trigger certain co-operation between the two [12]. For example the ANTHRAX virus (MS-DOS), a boot sector virus, leaves an extra copy of itself on the last track of the infected disk. If the disk is subsequently disinfected and later infected with the 2100 Bulgarian virus (MS-DOS), then this virus will detect the extra copy of ANTHRAX and reactivate the dormant code causing the re-infection of the boot sector.

[15] Generation Code

The WHALE virus [31] creates new generations of itself. It does this by scrambling the order of the subroutines within itself and changing both the encryption 'lock' and 'key'. It thus creates a new version of itself, entirely dissimilar to the parent, which it places in the file being infected.

1.5 : VIRUS CASE EXAMPLES

Every major computer manufacturer in the U.S.A. admit their machines have sustained attacks. Most of them have publicly stated that these attacks have been successful and that no solution currently exists [23]. Examples include :

1.5.0 THE AIDS TROJAN HORSE

A particularly malicious instance occurred when an "AIDS Introductory Diskette" was distributed using mailing lists of computing magazine publishers by the PC Cyborg Corporation [32]. This diskette contained not a virus, but a Trojan Horse, a program which when executed performs a function which is unexpected by the user. This is further explained in a later section. Although this was not strictly a virus it was a clear example of unauthorised code entering and affecting a system. Its operation was far from benign. It was in fact intended to extort money from users across Europe which is exactly what it achieved.

The disk purported to contain a database of AIDS information and a program which assessed a persons risk of catching AIDS. This disk came with a license agreement which indicated that one should not use these programs unless one was prepared to pay for them. Its install program printed an invoice for the software and created an invisible subdirectory, within which it placed an executable file. It then renamed the AUTOEXEC.BAT file and created a new hidden one which executed the file within the hidden directory before running the old AUTOEXEC.BAT. The executable file set a counter to 90 and each time thereafter when the machine was rebooted this count was decremented. When the count reached zero, the screen displayed the following message:

> *"The software lease for this computer has expired. If you wish to use this computer, you must renew the software lease. For further information turn on the printer and press return"*

It then printed the following :

> *"If you are reading this you software lease from PC Cyborg Corporation has expired. Renew the software lease before using this computer again. Warning do not attempt to use this computer until you have renewed your software lease"*

It then continued to print a demand for payment : $189 for 365 uses or $378 for the lifetime of the hard disk. The payment to be sent to a post office box in Panama.

For the next hour the machine, after a *"do not switch off"* message, performed a task which involved intensive disk activity. After which time the main directory entries had been encrypted and made invisible. Also, all the free disk space had been allocated as a hidden, indexed sequential database filled with blanks.

Two files were also added: a visible CYBORG.DOC - a copy of the registration and a hidden CYBORG.EXE, a file which was executed after every DOS function call to inform the user that he risks losing all his files by continuing to use the machine.

To the user it appeared as if all his files had disappeared to be replaced by one file, CYBORG.DOC, and the disk had shrunk to the size of this file. In actual fact all the data remained unaffected so, if the directory was restored, all returned to normal. As the original entries existed within the CYBORG.EXE file this was possible.

Over 7 000 corporations in the UK, France, West Germany, Italy and Sweden were hit including large institutions such as the London Stock Exchange and the British Ministry of Defence. It was clever and required significant expertise. It was costly : over $100 000 were spent before its launch. It was malicious and its sole purpose was to extort money from naive institutions.

1.5.1 GERMAN 'STRIKE VIRUS'

A German company was the victim of a virus attack which was perpetrated by disgruntled employees [23]. Before going on strike they introduced a virus which gradually slowed down the system until it ground to a complete halt. A previous backup was re-installed but it ground to a halt again. In the end several weeks of work were abandoned in order to restore the system to an uninfected state.

It should be noted that if the delay before triggering of a virus of this type is many months then it is likely that no backup will be uninfected, possibly leading to catastrophic results.

1.5.2 DATACRIME II - MS-DOS VIRUS

Datacrime II [33] is a non-resident, parasitic virus of type GCAM-GTAR. Its code is encrypted and the decryption section contains self-modifying code. It is therefore difficult both to analyse and to detect. This virus differs from many new generation viruses in that it is far from benign in nature. It triggers if the date is later than October the 12th in any year. A positive result will cause a low level format of track zero. This destroys the Disk Boot Sector, the first copy of the FAT, and most of the second copy of the FAT. Thus the contents of the disk will be largely irretrievable, although in theory the information still exists on the drive.

1.5.3 WHALE OR MOTHERFISH : MS-DOS VIRUS

The WHALE virus [31] is the first truly armoured virus to be found in the wild. It is also by far the largest assembled virus in existence, weighing in at 10K. The virus falls short by the fact that its detection is almost immediate, the machine slowing down by over 50% when the virus is active. The vast majority of the code ensures that the code itself is difficult to disassemble and attempts to disinform and confuse the analyst. WHALE uses multiple levels of encryption plus a series of anti-tamper measures. For example the virus implements checksumming on the BIOS data area and its own code to detect use of a debugger or breakpoints. If interference is detected then the virus will attempt to remove itself to prevent any analysis. In memory the virus decrypts an area it is about to process and recrypts the areas with which it is finished. Thus only a small window of the virus's code is visible in memory at any given point. This prevents detection by pattern matching. The virus also mutates itself into 30 possible generations of itself when infecting a file, as mentioned previously. The virus is a memory resident, parasitic virus of the GCAM-GTAR type, as it infects .COM and .EXE files indiscriminately and, upon triggering, merely prints messages to the screen.

1.6 : POSSIBLE VIRUSES OF THE FUTURE

The viruses appearing now are becoming increasingly difficult to detect. It is reasonable to assume that most future viruses will be like this, as primitive viruses will be detected and removed from systems early on in their infection phase.

What may change is the reason for a virus's existence. Current viruses seem to be written because they are a challenge to the writer and often the trigger is benign or non-existent. This is likely to change : viruses will appear which have a specific and malevolent purpose.

1.6.0 BLACKMAIL

With the advent of the AIDS Trojan Horse [32] mentioned in the previous section we may have been offered a frightening but realistic insight into what is to come. Viruses may appear which disable or corrupt systems with the purpose of extortion, offering to undo damage only if some payment is made to the perpetrator. The cost of re-installing a system after such an attack may be enormous and therefore victims of this form of attack may be forced to make the payment.

1.6.1 COVERT PROCESSING

In his paper entitled " Covert Distributed Processing with Computer Viruses" [34], White outlined a mechanism where viruses could be used to distribute elements of a problem to other systems thus covertly harnessing vast quantities of computing power, far in excess of that available to any single user or institution.

Vast quantities of computing power is available collectively in the world. The problem is, how do we access it ? Seeking permission to use it is impossible, therefore it would only be available if it could be accessed covertly. The computer virus provides just such a mechanism : it enters a system unknown to the user and with no permission from the system, performing any task it has been programmed to do.

Therefore a virus could be used to hide within it an "innocuous program to spread a distributed computing task among many users and many computers". Thus "processes and information can be distributed unwittingly by users in the normal process of sharing other information". As with any distributed problem, the inter-communication of the processes generated must be considered. For a virus probably the only reasonable communication mechanism would be a unidirectional one from parent to sibling : a virus when spawning a child would pass the original problem plus any progress it has made on the problem, to it. As White points out this lends itself to Tree Structured algorithms where offsprings are generated by parents and from that point remain independent of each other.

If a final result is calculated the virus could mutate itself into an "Information Carrying" virus and propagate back, eventually reaching the author of the task. Direct mailing of the result is another possibility but this risks identification of the author.

Clearly there are many possible tasks this form of virus could initiate. The task suggested by White was a Brute-Force attack on a Cryptosystem, for example the Data Encryption Standard (DES) encoding scheme. A virus was suggested which would chose random keys and test for success and also spawn other viruses which did the same. For DES which has a key space of size 2^56 he calculated the time required to crack it to be 0.26 years given that there were approximately 10^7 possible machines to infect. Although this was a vast underestimate, as it assumed all machines would become infected and would provide exclusive use to the virus all the time, it does

suggest the possibility of solutions to problems which are considered largely unsolvable. As White points out, as the aggregate computing power in the world increases so might the threat of such covert operations considering that such estimates, as mentioned above, will become more and more realistic.

1.6.2 DATA THEFT

In much the same way as for covert processing, a virus could be made to detect and remove sensitive data within a remote system [35]. The virus could be set the task of searching the files within the area it is executing for information. If it finds the information then it could mutate, as before, into an information carrying virus and attempt to propagate back to the author, or if it failed it could attempt further propagation in its original form. Thus a virus could propagate through high security systems in search of sensitive data.

As an example, suppose a single user machine, which is kept under lock and key within an office in a high security building, contains a bank code stored within a data file. This machine may also be running the highest of security programs to prevent entry. A virus is introduced into the system in the low security area of the building (or potentially another country if you can wait that long). This virus searches for data files which contain the string 'bank' or 'codes' or 'secret' in the title or even more specific search criteria. If it finds one, it places the contents of the file within itself and propagates itself back into the system. If not, it continues to infect files in an attempt to propagate further. If the virus succeeds then eventually the information will appear on a low security machine which can be detected by the author and retrieved.

Obviously there are no guarantees that this would succeed : it presupposes that the machine accepts outside executable files. It is however possible and the implications are very serious.

1.6.3 DATA CORRUPTION

Viruses exist today which corrupt data, deleting files or whole disks. What we may find is that viruses become more particular about what and how they corrupt. For example a virus may search for a specific file type such as accounting data files and change every '2' to a '3' within the file. The consequences of this may be far more damaging than complete data destruction. The virus may go unnoticed for longer and, in its undetected state, the damage it causes may be severe.

1.6.4 VIRUS JUNK MAIL

A virus was planted onto the widely used Compuserve network by a company, which placed an advertisement on the screen on a particular date and then removed itself [23]. It spread by infecting the initialisation files of the Apple Macintosh. Will this happen again? Are we about to enter an era of virus-generated junk mail?

1.6.5 SELECTIVE VENDOR ATTACKS

Selective predators of the SCAM-STAR category may be introduced as a marketing tool. Software developers may write viruses which target software sold by a specific rival vendor. The opportunity exists for malicious developers to attack their competitors without leaving a trace.

1.6.6 THE POSITIVE VIRUS

It may be possible that a virus which is benevolent in nature may appear. Cohen [36] suggested the possibility before viruses actually existed. His example was a 'Compression Virus' which, once attached to a file, compressed the original program code using a lossless compression algorithm. When the program was next executed the virus would decompress the program before running it. Thus, without the knowledge or expertise of the user, filespace would be saved. With this particular example problems exist :

[1] If the amount of space saved by compression is less than the size of the virus then no space is saved. In fact extra space is consumed.

[2] Compression algorithms work best with images and text, not with binary code which is what this virus would be dealing with.

[3] Users may not regard the space saved worthwhile compared with the overheads of decompression with each execution : this would then cease to be a 'positive' virus.

These problems exist only for this particular example. Although it is highly unlikely, maybe some application of viruses could be found which does have a positive effect.

1.7 : CURRENT ANTI-VIRUS TECHNIQUES

As the number of incidences of Computer Virus 'Infection' increases, so does the appearance of software to detect, prevent and remove viruses.

Like the first viruses, the first 'Anti-Virus' software was primitive. The first programs did not prevent the virus gaining entry to the system, but removed the virus and/or detected its presence. These programs were largely virus specific in that they were written for a single virus.

Now a large amount of software exists which is both professional and extensive [37]. Some very professional software of this type exists in the public domain but much of it exists as reasonably expensive software with yearly subscriptions for updates as new viruses are found.

Anti-Virus protection can be split into five distinct classes [38]. Anti-virus software may include one, some or all of these classes of protection. The five classes are :

1.7.0 SCANNING

Scanning software is designed to examine files and/or boot sectors and/or memory for virus signatures. A virus signature is a section of code which exists in the virus (and is non-modifying) but is unlikely to appear in other programs. An example of a virus signature for the MS-DOS DEVIL'S DANCE virus [39] would be :

AD03F3A426C706000003015E1E068CC048

This is a section of the viruses code which is long enough, and when disassembled, unique or at least uncommon enough for it not to appear in any uninfected program.

There are two types of such programs :

[i] Terminate and Stay Resident or 'TSR's

TSR programs are executed on boot and remain in memory scanning each file as they are run. This has the advantage that all programs are checked. On the IBM under MS-DOS this may be done by monitoring the INT 21H Function 4BH and 3DH, which are the execute and open functions, and scanning the file at that point.

Alternatively such programs can check each floppy disk as it is inserted, thus preventing infection entering the system. On the Macintosh this is easy as the system is event driven and disk insertion causes a 'disk_insert' event. On an IBM PC there is no equivalent indication of a disk insert but one way is for the TSR to monitor INT 13H Function 02H, which is the read sector function. If a request to read the boot sector is made then the IBM is making what it reckons to be the first access to the floppy. Checking either the boot sector or the whole disk for viruses at this point is possible. The reason why this is an indication that a disk may have been inserted, is that the IBM reads this sector on the first access to the floppy, to find out what format the disk is in. Checking the boot sector is quite reasonable but checking the whole disk is probably not. The reason for this is that the IBM assumes that the floppy disk may have been swapped in the drive if more than two seconds has passed and therefore the TSR program may scan the disk many times over with significant overhead each time.

In the case of the IBM, using the INT 13H Function 02H to check for boot sector accesses only and scanning for boot viruses, and using INT 21H Functions 4BH and 3DH as mentioned above to check for opening and execution of files and scanning accordingly, together provide the most reasonable solution for resident scanning [40]. However, in the case of the Macintosh, scanning the floppy when inserted provides a virus check without overheads on execution of programs on the hard disk. The hard disk only has to be scanned once and, if all floppies are scanned on entry, no virus (which is scanned for) can be introduced into the system. This has clear advantages over the IBM but is only available to the Macintosh because, unlike the IBM PC, all disk inserts are recognised by the operating system.

[ii] Manual Execution

Manual execution is the most common method, requiring the user to execute the scanning program. When invoked, the scanner checks memory and/or boot sectors and/or all/some of the files stored on all/some of the disks within the system. This has the disadvantage that for a system with a large volume of files, either one must wait for a significant period of time for all the files to be checked or, what is more likely, one will check only a small subset of files and therefore risk missing a possible infection. It has the advantage however of making the user aware of viruses and the system in general by forcing his/her interaction.

Under DOS this type may be executed from within the AUTOEXEC.BAT file to check a specific set of files on boot : if this includes the file COMMAND.COM then this will usually provide a reasonable indication as to whether there is an infection. Similarly for MACOS this type could be executed as a startup application to check the SYSTEM and FINDER files.

EXAMPLES

[1] SAM Virus Clinic™ : Symantec Anti-Virus for Macintosh [41]

> *'SAM Virus Clinic scans files on your volumes for the presence of known viruses and reports its findings ... it also reports "irregularities" in your files if you so desire.*
> *If your files contains a known virus SAM Virus Clinic can often repair the file right then and there.'*

This is a standard Macintosh application which requires the user to invoke it and choose a file, folder or volume to be scanned. It is not a shareware product and regular upgrades are only available to users who pay a yearly subscription. Upgrades are required as new viruses appear each day and Symantec add these to the list scanned for.

Another piece of software in the SAM package, SAM™ Intercept, is a memory resident application which can be configured to scan either the system folder or whole system volume on startup or shutdown. It can also be configured to scan floppy disks either automatically on disk insert or when a key sequence is pressed on insert.

Shareware Equivalent : Disinfectant (c) John Norstad, Northwestern University [42]

[2] VIRSCAN : Virus Scanning Software (c) IBM, for the IBM PC, MS-DOS [43]

This software package comes in three parts. The first is an executable file called VIRSCAN.EXE which is invoked by the user or by the AUTOEXEC.BAT file on startup. The second and third are data files containing virus signatures for Boot Sector Viruses and File Infecting Viruses (parasitic) respectively. Upon execution Virscan checks both itself and the signature files for modification in case a virus has corrupted any of these files. If there has been no modification the program scans firstly memory, then the boot sector, and finally a user defined set of files.

Shareware Equivalent : F-PROT Virus detection/protection/disinfection and utilities (c) F. Skulason

[3] HyperACCESS / HyperCOPY : Combined Communications/Copy & Virus Scanning (c) Firefox [44]

HyperACCESS is a combined communications and virus screening software package. As the serial transfer of files is the slowest part of a file transfer, virus scanning can be achieved with little to no overhead. HyperCOPY is a combined file copy and virus screening package. It behaves much like DOS COPY but incorporates the same virus scanning mechanism as HyperACCESS. Used together, all software entering the system will be scanned.

LIMITATIONS

[1] This is a classic case of 'closing the door once the horse has bolted'. Only after a system is infected will a scanning program detect the problem. Also by the time you have received the signature to add to the scanning software you may have the virus, have spread it, and have had your system corrupted.

[2] As the number of viruses increase so will scanning 'run-times'. There is a limit to how many patterns a scanning program can search for without the overheads becoming too great.

[3] As the rate of increase of the number of viruses increases, so the chance of a virus scanning package being up to date will decrease. The sheer number and increasing complexity of new viruses will swamp the scanning software authors. The work involved in disassembling, decrypting and understanding new viruses is already very time consuming and is also increasing as the viruses increase in complexity [45].

[4] Viruses which enter the system in compressed form will not be detected by standard signature checking until they are decompressed [46]. Therefore scanning floppy disks which contain compressed files which are decompressed by an install utility onto the hard disk, is futile.

[5] Unless booted from a clean floppy, a scanning program may fail to detect a virus if it incorporates 'stealth' functions within its code. As mentioned previously, a stealth virus will intercept attempts to check infected areas of a file and replace them with the original uninfected code.

[6] Scanning packages are often configured to check the boot sector on boot by inserting a command into the AUTOEXEC.BAT file. If a 'stealth' boot sector virus exists in the boot sector the scanner will not detect the virus as it has already been loaded into memory and its stealth abilities activated.

[7] Viruses exist which use the timer to choose a random insertion point during infection. Thus scanners which look for signatures at a fixed offset within the file are not capable of detecting such viruses.

[8] Effort and possibly a considerable amount of money is required for the constant updating of this type of software. For example, SWEEP, a virus scanning program for the IBM PC from Sophos Limited costs £295 + VAT per annum for monthly updates [47]. Public domain on the other hand although free or shareware, requires someone to download the updates from a bulletin board each month incurring telephone costs and considerable effort.

[9] Viruses such as the '666 Number of the Beast' MS-DOS virus [4], which are memory resident, intercept the file CLOSE function. Thus a scanning program can be the best possible agent for the spread : a scanning program may, whilst scanning, OPEN and CLOSE every uninfected file on a disk and for each CLOSE the file will be infected, resulting in complete infection of the disk.

[10] As mentioned previously, a new generation virus such as the 1260 [28] cannot be detected by a signature scan. All but thirty nine bytes are encrypted and the thirty nine bytes have self modifying bytes dispersed through them. Thus the longest section of virus code is three bytes, not enough for a signature as it would be far from unique.

[11] Viruses such as the WHALE [26] virus remain virtually totally encrypted even in memory and are only decrypted before processing. They also recrypt with a new key just after processing. This makes detection unlikely even when scanning memory.

1.7.1 VACCINES

This type of Anti-Virus software is always memory resident. Its purpose is to watch for 'suspicious' activity within the system. Often this type of program offer different levels of protection. A low level of checking offers the advantage that it will require the user to have little knowledge of the system : it will inform the user of activities which are clearly suspicious and leave the user to check for viruses as a result of the warning. Higher levels whilst affording greater security, may require the user to have a fair understanding of how the operating system works.

EXAMPLES

[1] SAM™ Intercept : Symantec Anti-Virus for Macintosh [41]

This part of the SAM package consists of two programs. The first is a Startup Program which automatically launches at startup and remains in memory. Its purpose is to intercept suspicious activities and report them to the user. The second is a Control Panel Program (CDEV) which allows the user to configure the startup program for the machine it is running on. One of the options provided is a choice of protection level : Basic ; Standard ; Advanced ; or Custom. As mentioned above, as the level of protection increases so does the requirement that the user knows how his machine and operating system works. Custom actually allows the user to choose exactly which operations should be checked for and reported, but this requires the user to have a very good knowledge of MACOS.

Two other useful features are present in this software : an ability to 'Learn' operations that the user considers valid so that when the user repeats this action SAM does not report it ; and a lock function using a password so that when the CDEV is used to configure the program this configuration is not alterable by another user.

[2] GateKeeper : virus monitor for the Macintosh [48]

GateKeeper restricts two basic classes of operations:

[1] "File" operations : Operations on information about files that contain programs.

[2] "Res" operations : Operations on the components of programs stored within files.

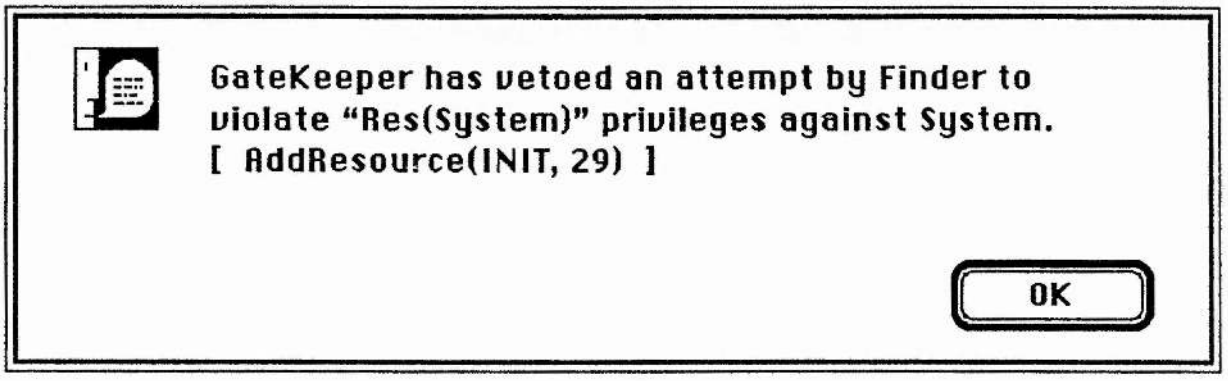
Within each class of operation there are three variants:

[1] "Self." : The file being operated on is the file containing the currently running program, *i.e. the program is operating on itself.

[2] "System" : The file being operated on is the System file.

[3] "Other." : The file being operated on is some other file, i.e. the program isn't operating on itself and it isn't operating on the System file, either.

With these two basic classes of virus operations, each of which has three variants, we see that there are a total of six separate operations for which GateKeeper has to watch. If any operation occurs which would cause modification of a file or resource albeit to itself, or the system or any other executable file, then Gatekeeper will veto the operation and inform the user. An example message is show below :



The reason that GateKeeper defines the three variants to its restricted operations is that some applications require such operations during the normal course of its execution. Such applications can be given privileges, much like file privileges on multi-user systems. For example Unstuffit, a file decompression utility requires "other" access to file operations in order to be able to open and manipulate the contents of files to be decompressed. Gatekeeper may be configured to log all suspicious occurrences and therefore be checked over by a system manager. A choice is also given as to whether the user wishes a suspicious operation to be vetoed or just notified, giving the user a chance to allow the operation. The former is most useful for users with little knowledge of the system, the latter for users with more knowledge.

[3] FluShot Plus : Memory Resident Virus Monitor for MS-DOS [49]

This memory resident (TSR) program monitors both the expected and unorthodox entry points to data held on physical storage. The most obvious, and correct, method of accessing disk data is via the file system. FluShot monitors all File Management Function Calls within INT 21H for suspicious activity. Alternatively one may access the disk in question directly. This would not be used by a normal application as the file operations deal with all necessary direct disk access. FluShot monitors all direct disk functions within INT 26H, INT 13H and INT 40H for any action which would violate normal MS-DOS procedures.

LIMITATIONS

[1] Loopholes in the system

MS-DOS, in particular, has many undocumented features and loopholes and shortcuts. Therefore Vaccine products which monitor 'standard' software interrupts such as INT 13H and INT 21H which relate to disk access and file access can be by-passed with relative ease.

The first viruses which attempt this are appearing. An example of this is the '666 Number of the Beast' virus which, by use of an undocumented DOS feature, obtains the original value of the interrupt vector for INT 13H. It can then access data on disk directly without a software interrupt : therefore no monitor program can detect it.

Another example are files which are opened as read only by a program. Vaccine programs will often allow this as no harm can be done to the file. The DOS System File Table (SFT) can be manipulated directly once the file is opened and the access privilege field altered to allow write access. This permission is then returned to read only after infection and the Vaccine program is none the wiser.

[2] Often perfectly acceptable operations are trapped. Programs during their normal course of operation may attempt operations which are similar if not identical to operations which a virus would attempt. However their reasons for attempting such operations are legitimate. As mentioned in the review of GateKeeper, there are somewhat cumbersome solutions to this problem but it involves keeping vast lists of 'exceptions to the rule' and careful checking that a virus doesn't attempt to enter using one such exception.

1.7.2 ENCRYPTION

This type of Anti-Virus Software writes programs and/or data onto the hard disk in a non-standard way and reverses the process when the files are read back. Thus if a virus does attempt to infect a system the effect is usually a scrambling of data rendering the program totally inoperable. The result of this is that detection is certain [50].

There are many different approaches which may be implemented on a system. One could encrypt every file but for viruses this is overkill. One could encrypt just the 'executables' within the system and therefore reduce the overheads. One could go one step further and only encrypt files which are critical or frequently used within the system.

A method of reducing the expense of encrypting a whole file is to generate a signature block for the file to be protected including, say, a cryptographic checksum of the whole executable, a password from the authoriser and a time stamp [1,51]. This signature block is then encrypted. Upon execution the signature block would be decrypted and the password and time stamp checked, and the checksum recomputed. This is far less computationally expensive than encryption of the whole file.

Under such a system, if a virus is unable to attack a file before it is encrypted and cannot break the encryption algorithm then the virus cannot infect the file. If the virus does infect the file before encryption, and all other files are encrypted, then the virus, although executed when the file is executed, will not be able to propagate.

LIMITATIONS

[1] It is possible to get into a situation where the system is inoperable and cannot be repaired, either by incorrect configuring or by a small change in the encrypted data causing the decrypted version to be totally corrupted.

[2] Part of the System must remain non-encrypted in order to be able to decrypt files and part will still be vulnerable.

[3] If the file being encrypted is already infected, the program will encrypt it regardless.

[4] The cost in maintenance of such a system can be large. This type of system is therefore usually reserved for highly critical systems.

[5] There are penalties on the system due to such techniques : encryption at run time may cause significant delay.

1.7.3 OPERATION RESTRICTION

This type of Anti-Virus Software operates in a similar manner to the Vaccine type of software. Unlike its Vaccine counterpart however, when it detects an illegal activity it will automatically prevent it. The level of restriction depends on the implementation and also in the generality of the restriction. Restrictions may be made which apply to all programs or to a single file. Often these restrictions refer largely to operating system files and areas of the disk which store parameters required by that operating system.

Many of these features are already implemented in the security shells of mainframe computer systems: write protection of files, access restriction and inability to modify system parameters.

LIMITATIONS

[1] The more versatile this type of software becomes the more time required to set it up.

[2] Low-level and non-standard operations may circumvent these checks much in the same way as for vaccines.

[3] A way must exist for an operator to allow certain of the prevented operations for upgrades and legitimate alterations to the system. This provides another gateway for a virus.

1.7.4 CHANGE DETECTION

This type of Anti-Virus Software monitors certain unchanging information within the system. This may include all or some of : system files ; certain program files ; and system configuration. System configuration would include information such as partition location and bounds, volume directories and all information used to recover files from the volume, CMOS RAM data, Interrupt vector locations and any other data used by the system which is likely to come under attack [53,54].

The most common way to detect change in key parameters within a computer system is to keep a copy of the values and make a comparison. This is much the same method as MS-DOS uses, keeping multiple copies of the FATs which detects inconsistencies and

corruption. These saved parameters may be stored in a hopefully secure area of the hard disk or stored on write protected floppy disk for periodic checking. The advantage of this is that it will firstly detect change, and secondly you have a backup of what these values should be.

Simply monitoring the physical size of applications will often alert the user to probable infection either manually using DIR or by using a special utility [52]. More advanced methods use some form of checksum or cyclic redundancy check (CRC) [55]. This will detect changes in files, sectors or areas of memory even if the file length has not been altered. Cryptographic checksums are not easily reproducible and should be used if possible [52].

EXAMPLES

[1] VACCINE : Anti-Virus system from Sophos Limited [47]

This software package calculates cryptographic checksums for specified data and uses these to detect changes. VACCINE generates, under user control, these cryptographic 'fingerprints' and stores them. DIAGNOSE, another program in the package, checks these fingerprints for change and this can be configured to be automatic. The items it can protect are specific files, sets of files, disk sectors (both logical and physical), and memory ranges. It will detect : changes to files, appearances of new files, disappearance of protected files, and changes to disk sectors and/or memory ranges. The approximate cost of this package is £200.

[2] T-CELL : Protection System from Secure Transmission AB [3]

This set of programs have been developed to be incorporated into mainframe systems. Firstly, they protect themselves using a software seal. They then maintain a set of standard 'safe' copies of certain operating system software and utilities, which they compare with operational versions for change.

[3] CA-Examine : MVS Operating System Virus Protection Tool [3]

This operating system specific set of programs use an extensive knowledge base of the internal structure of MVS to protect the system. They check system libraries, memory-resident modules, and system modifications. They use expert system techniques to analyse operating system parameters in order to identify abnormalities and unauthorised changes.

LIMITATIONS

[1] This type of AntiVirus software is reactive rather than proactive. Again it is 'Closing the door once the horse has bolted'. A virus attack will only be detected once it has happened.

[2] Checksums can be cracked [56]. The most common method is a Brute force attack where one generates many different sets of data and corresponding checksums until a match with the original data is found. In the case of a virus, the virus appends itself to the code and then alters a redundant area within the virus until the checksum is that of the original. For a checksum of n bits it takes $\ln(2)*2^n$ mutations before one has a 50%

probability of success. For example for a 16 bit checksum approximately 40 000 mutations would be necessary. Clearly the implementation would choose a number of bits such that a brute force attack would not be practical. Thirty two bits would be satisfactory. However another method exists to break checksums, the Trap Door. If a checksum algorithm is invertable [56] then a Trap Door exists. A Trap Door is a short cut to determine a checksum, and requires far less computational effort than a brute force attack. Unfortunately there is no test for invertability and therefore no test to show that a trap door does not exist for a checksum algorithm. Therefore such algorithms are not as secure as one might be lead to believe.

[3] If such methods include keeping multiple copies of information, there is no reason to suppose that a virus won't attack both copies in a similar manner and may therefore destroy the effective backup that existed and also cause both sets to be identical and to be passed by the change detection software.

[4] Checksumming an application pre-supposes that the original was virus free.

[5] The overheads, both in computing time and operator time, are high in this type of system. New software must be added to the change detection system and regular checks must be made for system integrity, or this system is worthless.

1.7.5 OTHER METHODS OF REDUCING THE VIRUS THREAT

There are many ways apart from pure software means, by which we may reduce the threat of virus infection. The most obvious is the education of the user. If a user is well informed of the risk of viruses, how they are carried and spread, and the possible result of infection, then they may be more responsible in their actions. Training may be sought from many places. Information often accompanies anti-virus software packages, and books exist on the subject [57].

Companies such as SOPHOS offer complete training packages including a video and handbook set [58]. This particular course is split into two parts : one a basic explanation of viruses and their effects and what users should not do ; and the other which deals with prevention, detection and containment which is designed for managers and computer specialists.

A few conferences dedicated to the treatment of viruses have occurred [37,59], and will occur in the future, which inform the more involved computer operators and system managers of the state of the art techniques to prevent virus entry. One such conference entitled 'The First International Conference on Combating Computer Viruses' hopes to bring the key researchers into computer viruses to explain, from practical experience, the current threat, possible future threat, and methods of reducing these threats.

It has been found that employees will ignore warnings about running externally acquired software, even if they know their job is at risk if they are caught. It has therefore been suggested that large organisations which can afford it should provide a 'Dirty PC' [7] which is isolated from all others and is used solely for employees to run demonstration software, games and shareware. This solution therefore allows the

employees the opportunity at no risk and they are less likely to run them on forbidden machines. Also shareware or demonstration disks may be run which may be work related but would otherwise have been forbidden to the user.

Many consider that legislation should prevent the publication and distribution of material which is of use to a potential virus writer. Examples include the book 'Computer Viruses : A High-Tech Disease' by Ralf Burger [60] which contains the source code of several viruses, and a German software package called 'Virus Constructor Set' which includes a menu driven piece of software which will generate custom viruses with no expertise required by the user.

1.8 : VIRUSES AND MAINFRAMES

Although the vast majority of computer viruses reported to date exist on personal computers, the threat to mainframes still exists. Cohen [36] illustrated the threat of computer viruses on mainframes back in 1984 where he introduced one into a VAX system running UNIX. The less consideration we give to the possibility of mainframe infection, the more the likelihood that such viruses will manifest themselves.

Cohen [23] also demonstrated a virus written in UNIX - C which consisted of 200 lines of code and made no use of any implementation lapses within the system. It took him 8 hours to write it and it took only half a second for each infection and an average of 30 minutes to take over the whole system.

VIRUS PREVENTION

The Anti-Viral techniques covered in the previous section all apply here. One extra technique which significantly reduces the chance of virus spread throughout a mainframe system is Access Control Software.

Access Control Software often exists in such systems for other reasons. Since this type of software restricts access to critical programs, and thus prevents modifications, infection is not possible unless the virus exists at a high level of access. Thus users who have only Low Level access to a system may introduce a virus but the propagation will be severely limited.

Clearly users such as System Administrators who enter the system at a High Level of access could introduce a virus which could spread unbounded within the system. Also they could execute software which was introduced at a low level and allow it access to the higher level. These users must therefore be chosen carefully.

New software which enters such a system should be given only restricted access to High Level programs and should be monitored carefully for a 'trial' period before permanent inclusion to the system. However, even if an administrator is careful, protection exists only at a software level, and viruses can and will find routes of infection.

1.9 : ROGUE PROGRAMS

There are other types of 'rogue' program which inflict themselves on the computing community [21,61] and these will also be considered within the scope of this work. These include:

1.9.0 TROJAN HORSES

A Trojan Horse is a program which purports to do one thing but does something entirely different [6]. Data destruction is the usual 'purpose' of these programs. Examples of these include :

DANCERS.BAS An MS-DOS Trojan
Displays dancers animated on the display whilst destroying the FAT tables , thus rendering the computer inoperable with total loss of data (recovery programs possible).

DISCSCAN.EXE An MS-DOS Trojan
Claims to scan the disk for bad sectors - Actually it writes them!

Part of the trouble with these is that it takes little expertise to write one : it takes little effort to write a program to format the hard disk, then all that is required is to think of an interesting name for it. Its ability to spread is more limited however as the source of the trojan is usually more obvious than a virus, so the author must take elaborate steps to inject it anonymously. Also a new piece of software is usually tried out, and therefore discovered, before it is further distributed. This is especially true of public domain libraries which test submitted software, albeit not all that thoroughly in some cases, before making it available to the public. More complex trojans may have a trigger mechanism much the same as a virus and this may make the application seem to function correctly for a period of time during which further distribution may take place. See section 1.5.0 for a case study of a very clever trojan.

1.9.1 WORMS

Worms are self replicating, self-propagating programs which are complete in themselves. Unlike viruses they don't require a program to attach themselves to in order to reproduce. These usually replicate over a network.

The most infamous example of a worm is the 'Internet WORM' [62,63,64]. This worm was written by a graduate student in computer science from Cornell and on November the 2nd 1988 he injected it into the Internet network. In a matter of hours it had invaded over 6,000 machines the length and breadth of the United States. Neither data corruption nor destruction was its aim but its effect was to cause many machines to have to be disconnected from the network or shut down, and thus many hours of computing time were lost.

The worm exploited a set of fairly well known loopholes in the systems it attacked. Firstly it used a debug option on the mail system which allowed it to execute commands on the remote machine. The commands it sent were a bootstrap to copy the worms main binary files onto the remote system and then link them with the local libraries and load them. Once loaded the worm gathered system information as to which hosts were connected. Further propagation was then attempted. Three possible routes were used :

[1] Remote Shell : certain hosts can have trusted user status where connection to a host requires no password. If the worm found this it executed a remote shell.

[2] Remote Finger : it is possible to overflow the buffer of this utility and overwrite the stack, thus modifying the flow of control back into the stack where the worm had placed another bootstrap.

[3] Mail : as described earlier.

Then the worm attempted to break into local accounts on the machine in one of three ways :

[1] check password file for usernames with no password and break in

[2] crack the passwords with a series of possibilities such as the user's first name, last name, account name typed twice, account name typed backwards, etc ...

[3] encrypt each word in the dictionary and attempt a match with the encrypted passwords in the password file.

If a break was successful then again the worm searched for remote connections possible from the account and attempted to propagate.

This worm was benign in many respects but the potential for harm was enormous. It served to point out the vulnerability of computer systems, especially on a network.

1.9.2 MAGIC COOKIES

Magic Cookies are sections of code imbedded into applications by the programmer which are not self replicating. Their purpose is not usually harmful : often these are just messages which appear if a certain key sequence is entered. The potential however for these to do harm is there : It was reported by the German Press Agency (dpa) on January the 11th 1991 [65] that French computer scientists were reputed to have implanted code within war associated electronic systems to : "provoke, after some time, faults and 'profitable repair work' ". Code also existed within these systems

which could be triggered remotely so that, for example, French built EXOCET missiles could be switched off from French ships. Given that this release was during the much publicised 'Gulf Crisis' this information may be considered as fairly suspect and the claims exaggerated and unlikely. They are however plausible and should be considered as a distinct threat as there is self confessed research into such electronic warfare (EW) currently under way [66].

1.9.3 TEXT FILE BOMBS

It is often quoted that viruses cannot be transferred via data files [67]. This is true but it should be mentioned that, although this is the case, there is scope for mischief in this area. Often a word-processor data file may contain certain escape sequences which, when interpreted by the word-processor, will cause bytes from the file to be read as commands and executed or may redefine the keyboard. The result may be alteration of the data within the file, corruption of the system, or redefinition of a key to, for example, reformat the disk. The ability to replicate is highly unlikely, although this would in theory depend on the application in question : most applications will 'interpret' these sequences as a distinct subset of possible commands. It would be unlikely that any application would be written such that it would load 'real' executable code from a data file.

1.9.4 LOGIC BOMB

A Logic Bomb is a section of code, embedded within a program, often inserted by individuals who are attempting an attack on specific individuals or institutions. A program containing a logic bomb will operate perfectly normally until some specific pre-specified condition is met. At that point instead of normal operation, the program will execute the embedded 'bomb' code.

A classic example of this occurred in 1985 when a programmer working for an insurance company placed code within his company's pay-roll processing program. This logic bomb checked for the existence of his pay-roll number and if this number was absent for two consecutive pay-roll calculations then the bomb triggered. On the 21st September, two months after his dismissal, approximately 150 000 client records were destroyed.

Time bombs are a specific case of logic bomb which trigger on a time based event. It may activate on Christmas day with a Yuletide message or format the hard disk on April the first. Alternatively it may simply wait a month from its activation before performing a task, making it more difficult to trace the culprit.

1.10 : SUMMARY

A Virus, unlike many forms of attack has a tremendous range. A typical IBM PC based virus spreads to hundreds of unconnected machines in a matter of weeks. In a well-connected network, spread can be more of the order of thousands of machines in a matter of hours. In a timeshare environment experiments have shown that a typical time to attain all privileges of all users would be a mere thirty minutes.

Viruses will persist indefinitely : a virus may appear to disappear from the computing community only to reappear as it is downloaded from a backup, maybe years later.

As the route a virus takes between the initial infection and any given instant may be complex and lengthy, tracing its source is not normally possible.

The potential threat of a wide-spread computer virus has been much theorised. Most concur that the potential damage to government, financial, business, and academic institutions is extreme [68,69,70].

As Anti-Virus software appears in the marketplace, so do more and more complex and sophisticated viruses. As a loophole is plugged, so another one appears. It is open war between the two and as each new virus which bypasses security appears, we are vulnerable to attack, until such times as the Anti-Virus software is updated.

The 'sting in the tail' of a virus, the code the trigger mechanism executes, could, and will be, wide and varied. Current virus writers have concentrated on the propagation and stealth capabilities of their viruses but as these reach a relative peak of sophistication, we may find their attention moving to the code which is called upon

triggering. This could vary from total destruction to the more alarming subtle alteration which could be equally destructive but much harder to detect. As mentioned above, the threat of theft, blackmail, and covert processing also exist.

The effect of computer viruses is already becoming apparent. As fear grows we are seeing a rapid movement towards isolationist systems. System managers and individuals alike are afraid of free exchange of computer data which will inevitably slow progress in the computing field. As 'fortress mentality' sets in we may also see the decline and eventual removal of Bulletin Boards as demand for such software plummets.

1.11 REFERENCES

[1] **Highland H.** (1988) Computer Viruses Can Be Deadly. *EDPACS,* **15(12),** 1-6

[2] **Adleman L. M.** (1988) An Abstract Theory of Computer Viruses. *CRYPTO '88 : Conference - Advances in Cryptology 1988,* 354-374

[3] **Al-Dossary G. M.** (1990) Computer Virus Prevention and Containment on Mainframes. *Computers and Security,* **9,** 131-137

[4] **Skulason F.** (1990) The Bulgarian Computer Viruses - 'The Virus Factory'. *Virus Bulletin,* June Issue, 6-9

[5] **Anon.** (1990) How does an IBM PC Virus Infect a Computer. *Virus Bulletin,* April Issue, 11-13

[6] **Solomon A.** (1988) A Trojan War. *Personal Computer World,* August Issue, 166-170

[7] **Hruska J.** (1990) Computer Viruses. *Information Age,* **12(2),** 100-108

[8] **Slade R.** (1991) (c) Brain id and Disinfection (PC). *Virus-L Digest* , **4(7),** No 7

[9] **Anon.** (1989) Short Descriptions of Known IBM PC Viruses. *Virus Bulletin,* October Issue, 9-12

[10] **Anon.** (1990) Technical Notes. *Virus Bulletin,* September Issue, 3-4

[11] **Skulason F.** (1990) The Structure of Virus Infection Part I : .COM Files. *Virus Bulletin,* July Issue, 10-11

[12] **Anon.** (1990) Technical Notes. *Virus Bulletin,* November Issue, 3-4

[13] **Skulason F.** (1990) The Structure of Virus Infection Part II : .EXE Files. *Virus Bulletin,* August Issue, 16-17

[14] **Ferbrache D.** (1989) nVIR and its Clones. *Virus Bulletin,* October Issue, 13

[15] **Skulason F.** (1990) The Structure of Virus Infection Part III : Boot Sector Viruses. *Virus Bulletin,* September Issue, 15-16

[16] **Jacobs R.** (1990) Joshi - Spreading Like a Forest Fire. *Virus Bulletin,* December Issue 17-18

[17] **Anon.** (1990) Known IBM PC Viruses. *Virus Bulletin,* August Issue, 7-15

[18] **Bradshaw D.** (1991) Regular Dose of Prevention. *Financial Times,* 19 February

[19] **Anon.** (1990) 4K A New Level of Sophistication. *Virus Bulletin,* May Issue, 10-12

[20] **Bates J.** (1990) 666 - The Number of The Beast. *Virus Bulletin,* May Issue, 13-15

[21] **Simons G. L. et al.** (1990) Protection From Viruses. *Information Age,* **12(2),** 109-115

[22] **Skulason F.** (1990) IBM PC Viruses : The New Generation. *Virus Bulletin,* March Issue, 10-12

[23] **Cohen F.** (1988) On the Implications of Computer Viruses and Methods of Defense. *Computers and Security,* **7,** 167-184

[24] **Radai Y.** (1989) The Israeli PC Virus. *Computers and Security,* **8,** 111-113

[25] **Skulason F.** (1991) New Viruses (PC). *Virus-L Digest* , **4(18),** No 11

[26] **Bates J.** (1990) From Brain to Whale - The Story So Far. *Virus Bulletin,* October Issue, 12-14

[27] **Skulason F.** (1990) Virus Encryption Techniques. *Virus Bulletin,* November Issue, 13-16

[28] **Lammer P.** (1990) 1260 Revisited. *Virus Bulletin,* April Issue, 10

[29] **Chess D. M.** (1991) Obscure Procedure In Yankee Doodle (PC). *Virus-L Digest* , **4(9),** No 14

[30] **Bates J.** (1990) Virus Dissection II - Flip - A Professional's Handiwork? *Virus Bulletin,* September Issue, 18-20

[31] **Bates J.** (1990) Whale ... A Dinosaur Heading For Extinction. *Virus Bulletin,* November Issue, 17-19

[32] **Marmion D.** (1990) Computer Viruses : An Overview. *Library Software Review,* **9 (3),** 139-144

[33] **Bates J.** (1990) Virus Dissection - Datacrime II - Refined Hatred. *Virus Bulletin,* August Issue, 18-20

[34] **White S. R.** (1989) Covert Distributed Processing with Computer Viruses. *CRYPTO '89 : Conference - Advances in Cryptology 1989,* 616-619

[35] **Nathanson L.** (1991) Interesting Use of Viruses. *Virus-L Digest* , **4(38),** No 3

[36] **Cohen F.** (1987) Computer Viruses - Theory and Experiments. *Computers and Security,* **6,** 22-35

[37] **Anon.** (1990) Virus Protection Guidelines. *PC Business Software,* **15(1),** 9-11

[38] **Slade R.** (1990) Antiviral Evaluation Guidelines. *Virus-L Digest* , **4(2),** No 4

[39] **IBM** Virus Scanner Signature List. *VIRUSCAN Scanning Package for the IBM PC.*

[40] **Padgett P.** (1991) Hard Disk Protection (PC). *Virus-L Digest* , **4(9),** No 5

[41] **SYMANTEC Corporation** Documentation and On-Line Help. *SAM Virus Clinic and Intercept v1.1 for the Macintosh.*

[42] **Norstad J.** User Manual. *DISINFECTANT 2.1 for the Macintosh.*

[43] **Jackson K.** (1990) Viruscan. *Virus Bulletin,* September Issue, 22-23

[44] **Jackson K.** (1990) HyperAccess/5 - A Virus Filtering Communications Program. *Virus Bulletin,* Novemeber Issue, 21-23

[45] **Anon.** (1990) Virus Monitoring Software- An Endless Battle. *Virus Bulletin,* July Issue, 15-16

[46] **Jackson K.** (1990) Dynamic Decompression, LZEXE and the Virus Problem. *Virus Bulletin,* June Issue, 12

[47] **SOPHOS** SWEEP and VACCINE Anti-Virus Products. *Distributed Sales Brochure.*

[48] **Johnson C.** Release Notes and User Manual. *GATEKEEPER 1.0.1 for the Macintosh.*

[49] **Highland H. J.** (1988) An Overview of 18 Virus Protection Products. *Computers and Security,* **7 (2),** 157-161

[50] **Pozzo M. et al.** (1987) An Approach to Containing Computer Viruses. *Computers and Security,* **6,** 321-331

[51] **Anon.** (1988) Virus Defense Alert. *Computers and Security,* **7 (2),** 156, 158-161

[52] **Fåk V.** (1988) Are We Valnerable To a Virus Attack? *Computers and Security,* **7 (2),** 151-155

[53] **Jackson K.** (1990) Certus - Product Review. *Virus Bulletin,* June Issue, 18-19

[54] **Highland H. J.** (1988) How To Combat a Computer Virus. *Computers and Security,* **7 (2),** 157, 161-163

[55] **Radai Y.** (1990) V-Analyst - Product Review. *Virus Bulletin,* October Issue, 15

[56] **Varney D.** (1990) Adequacy of Checksum Algorithms for Computer Virus Detection. *Proceedings 1990 ACM SIGSMALL/PC Symposium on Small Systems,* 280-282

[57] **Highland H. J.** (1989) What if a Computer Virus Strikes? *EDPACS* July, 11-17

[58] SOPHOS Video Instruction Package. *Distributed Sales Brochure.*

[59] **Virus Bulletin Ltd.** First International Conference. *Distributed Brochure.*

[60] **Burger R.** (1988) Computer Viruses - A High-Tech Disease. *Abacus.*

[61] **Vail J.** (1991) Eaters of Language. *Virus-L Digest* , **4(12),** No 8

[62] **Spafford E. H.** (1989) Crisis and Aftermath (The Internet Worm). *Communications of the ACM,* **32(6),** 678-687

[63] **Rochlis J. A. & Eichin M. W.** (1989) With Microscope and Tweezers : The Worm from MIT's Perspective. *Communications of the ACM,* **32 (6),** 689-698

[64] **Ferbrache D.** (1990) The Internet Worm - Action and Reaction. *Virus Bulletin,* June Issue, 13-17

[65] **Brunnstein K.** (1991) (no) Viruses in Irak's EXOCET? *Virus-L Digest* , **4(10),** No 13

[66] **Cramer M. L. & Pratt S. R.** (1989) Computer Virus Countermeasures - A New Type of EW. *Defense Electronics,* **21 (10),** 75-86

[67] **Appleyard A.** (1991) Viruses in Text Files. *Virus-L Digest* , **4(27),** No 13

[68] Anon. (1990) The Bug Watch. *EDGE,* July / August Issue, 26

[69] Ross S. (1988) Viruses, Worms and Other (Computer) Plagues. *The EDP Auditor Journal,* **3,** 21-23

[70] Lin J. (1989) The Impact of Computer Viruses on Society. *Computers and Society,* **19(4),** 9-12

CHAPTER 2

A METHOD FOR GUARANTEEING CONTAINMENT AND PROBABLE DETECTION OF COMPUTER VIRUSES

2.0 INTRODUCTION

Presented within this work is a unique hardware solution to the problem of computer virus infection. The design philosophy behind this solution is largely encompassed within the following statements : Firstly, a series of partitions is set up together with a series of well defined rules for transfer between these partitions. Thereafter, any attempts to violate these rules are detected and prohibited. Moreover the schema is wholly independent of the contents of the partitioned areas.

System Integrity has always been of primary importance in a computing system. Many systems incorporate measures to ensure users may only have access to data and the ability to invoke operations if certain well defined criteria are met, thus ensuring that the system's integrity remains secure. However, with the advent of computer viruses and other hostile programs, such systems have failed to provide sufficient system protection.

An example of this would be the file access privileges in the UNIX operating system. This system prevents a user from corrupting system files. However if a virus that has been introduced into a system at a minimum security level is executed by an unsuspecting user with high security access, the virus may corrupt, using all the privileges of the high security level access.

Due to the inadequacy of existing security mechanisms, many security systems dedicated to virus detection appeared. Such systems also prove to be inadequate as they are not only expensive to maintain, but fail to provide a lock-tight solution. Examples of such systems were discussed in Chapter 1.

These short-comings arise because it is not possible to isolate the anti-virus program from the virus. As long as both programs execute on the same microprocessor, it is possible for the virus to circumvent, corrupt or destroy its counterpart. It is a war with neither side having a real advantage.

Presented in this chapter is both method and apparatus for prevention against corruption, or destruction of data within a computer system, by computer viruses or other hostile programs. The method explained below will guarantee system integrity and virus containment for a given system given that certain basic criteria are met. This method will require little user intervention or discipline : it is intended that this method be largely transparent to the user. This method will require no hardware modification of the computer or the hard disk, nor software modification of the computer's operating system. No apriori knowledge of a given virus will be required : this method will be sufficient not only for existing viruses but also all future viruses.

The method used will inhibit the propagation of viruses by controlling the operations which can be performed on particular data or classes of data. This will be implemented by monitoring all read, write and format operations which are to be performed on data on the storage medium of the computer system. Any operation which is considered suspect will be prevented, and if necessary the machine rendered inoperable until a system manager has inspected it. The method is implemented in such a way that both the testing and control of operations are made outwith the virus's reach. In the preferred implementation of this method, testing and control takes place on a dedicated processor placed between the computer system and its storage medium. As the physical access the computer system has on the extra processor is limited, corruption of the method cannot take place from the computer system and, as the virus executes on the computer system, it therefore cannot corrupt the method. Since the method relies on the supervision of operations between system and storage, it will hereafter be referred to as the 'Supervisor'.

The Supervisor will render certain data 'Read Only' so that no corruption is possible. System files, System parameters, and boot code are protected in this manner. Programs introduced into the system which are not self-modifying may also be protected in this manner. The remaining data within the system will be divided such that if a virus is introduced, it is contained and the damage it can cause is severely limited. Detection of the virus is also extremely likely. Containment is made possible as most modern operating systems facilitate the partitioning of storage devices. The Supervisor will guarantee to keep partitions distinct so that viruses may not spread across partition boundaries. Therefore a partition which is virus free and has no infected programs introduced into it from an external source, cannot become infected however rife infection is within another partition on the same medium.

2.1 THE METHOD

2.1.0 GENERAL OVERVIEW

Firstly the storage medium is divided into a plurality of non-overlapping partitions. Most operating systems, including MS-DOS, provide such a facility as standard. The first partition is the Boot partition, and the remaining General partitions. The sectors within each partition are then divided into two segments : those which contain operating system information and those which will eventually contain user-generated information.

The machine may be powered up in one of two possible modes : one requiring a password before entry, known as Unsupervised Mode ; and the default, Supervised Mode. During normal operation of the machine, a user would enter Supervised Mode, Unsupervised Mode being used only when the system requires modification.

In Supervised Mode the Boot partition is designated as 'Read Only' and should encompass all operating system files and boot code. The information stored within this partition will always be accessible, for reading only, when the machine is booted. Any attempt to write to this partition, including alteration to the partition's size, will be denied.

In Supervised Mode Low level formatting of the storage device will be denied.

In Supervised Mode the user shall decide on one partition which he requires after the machine has booted. This can be implemented either by the user choosing a partition from a menu when the BIOS for the storage device is executed, or by automatically designating the first partition the user accesses after booting.

The partition which is made available to the user is referred to as the Active partition. The user is granted Read and Write privileges to the user sectors of this partition and Restricted Read and Write privileges to the Operating System sectors. Certain Operating System sectors may never be altered, such as the sectors which contain partition bound information.

A user, for a single session with a computer system protected by such a Supervisor, will therefore have access to two partitions : the Boot partition from which the user may read, and a single General partition of his choice, called the Active partition, to which the user may read or write. A session with a computer system is defined here as the time from boot to shutdown / power down of the machine. It is necessary that the Active partition not be changed during a session as a virus may be active in memory and then inter-partition propagation could occur. Only if memory is cleared and the operating system re-installed from the virus-free software in the Boot partition, is the system safe for the allocation of an Active partition.

Any attempt to read from the user sectors of a non-active General partition will be denied. The reason for this is that a 'Read' implies a possible 'Execute' and this would allow a virus from a non-active partition to be activated. Any attempt to write to sectors within a non-active General partition is denied as this would allow the copying of infected programs across partition boundaries.

This schema provides a virus independent solution. It does not operate at a file level, nor does it require knowledge as to the content of the files stored within the system. Instead the locations from, and to which, data may be transferred and accessed are monitored and controlled. A virus placed within a system is guaranteed to be fully contained within the General partition into which it was initially introduced. It can be

further guaranteed that no boot sector viruses may infect a device secured by this method. The operating system files and all other files stored within the Boot partition are also guaranteed virus free, provided they were virus free when introduced into the system. Virus detection is also likely, as a virus would normally attempt to infect files within the boot partition, and other partitions in general. When such an attempt is made it is denied, the machine rebooted, and a message displayed. The machine may also be rendered inoperable until a system manager reactivates it.

2.1.1 THE UNSUPERVISED USER

Supervised Mode relies on having a fully configured system which will require no future upgrading or alteration. A mechanism must exist within the schema to allow the system to be configured, including defining partition boundaries and formatting partitions. Software, including the operating system, must be installed within the Boot partition and periodically updated. To facilitate this a mechanism is implemented by which a single user may enter what is to be called Unsupervised Mode.

In Unsupervised Mode the Supervisor permits all operations, rendering it transparent to the system. In this mode no virus protection exists. The user must be trusted and relatively experienced in the dangers of viruses. Software introduced into the Boot partition must be copied from shrink wrapped source disks. No software should be executed unless absolutely necessary (eg FDISK). If program execution is required, only executable files within the Boot partition or clean master disk should be used. Executable files within General partitions are not guaranteed virus free and should never be executed in Unsupervised Mode.

As such a mode provides a potential route of infection, it is password protected. This password should only be divested to system managers whose job it is to maintain the system in question. The password is passed before the operating system is loaded and therefore no virus may be active to intercept it. The password is stored on an area of the storage medium where it is inaccessible even in Unsupervised Mode.

2.1.2 CHOOSING THE ACTIVE PARTITION

Two different possibilities exist for possible mechanisms in determining which General partition to make active. Both methods are valid, so the choice as to which method is used is a matter of preference only.

[1] Explicit Choice

This method relies on the user making a determined choice before the system boots. This would preferably be implemented as a menu to facilitate a quick choice from all possibilities. Configuration of this menu in the form of a text field for each partition may be desired, giving the user an indication of the partition's contents.

The advantage of this method is that the user is conscious of the choice. However, only the text field indicates the contents of the partition and if the choice is incorrect a reboot is required.

[2] Implicit Choice

This method is simple to implement : the first Read or Write operation to a General partition which would cause a violation if the partition was not active, and yet would be permitted if the partition was active, causes activation if no active partition already exists. Thus if a user selects a partition and executes an application then the partition is made active implicitly before the application is launched. This may be announced by a message to the screen.

The advantages of this are that firstly, it is simple to implement and secondly, as reading from system sectors of a non-active partition is allowed, the user may browse the root directory of all partitions before making his choice and committing himself to one specific partition. However, this method may not be as intuitive to users as the explicit method, and may lead to a certain amount of confusion. It would be a matter of market research as to which is the better choice.

2.1.3 REACTING TO OPERATION VIOLATIONS

The reaction the Supervisor may have to any given violation may vary. The possible actions include : warning ; reboot and warning ; and reboot , warning and disablement. The obvious way to decide on an action is to consider the operation type: whether it is a Read, Write or (low level) Format operation. It is possible, and may be desirable, to break these operations down further into groups, such as considering a Write violation to the Boot partition as being separate from that of a Write violation to a non active General partition. Again, within the existing hardware, the choice could be market directed.

CHAPTER 3

IMPLEMENTING THE VIRUS SCHEMA

The chosen implementation was designed to operate on an IBM - AT, or true compatible, running the MS-DOS operating system. The reason for the choice is that this type of machine is by far the most prolific personal computer in the business world. This implementation will therefore provide security to the singularly largest group of potential users. This type of machine is also the one most under attack. To date there are over four hundred MS-DOS viruses recorded, with the number increasing at the rate of one per day.

The Apple Macintosh running MACOS would have been another possibility. These machines are however less numerous, as are corresponding MAC viruses, in the business world.

Mainframe machines running multi-user operating systems such as UNIX or VMS would have been a further possibility. Viruses are far less prolific on these machines to date, however they should not be ignored. It is also important to choose an implementation which will clearly exemplify the method without excess complication. A mainframe solution would be possible but the solution would be more complex to implement and the end product no better at proving the validity of the method. Such operating systems are also implemented on many different platforms and therefore a different implementation must be found for each, although the philosophy would be the same as described in this thesis.

The method outlined is general to any operating system which supports hard disk partitioning.

3.0 HARDWARE

The prototype was implemented as a hard disk adapter card for an IBM AT running MS-DOS version 3.3 or later. The card contains both the Supervisor and a Small Computer Systems Interface for a hard disk (SCSI). The hardware was designed so that access to the SCSI interface is only possible with the Supervisor's prior permission. Figure 3.0 contains a block diagram which illustrates the function of this card.

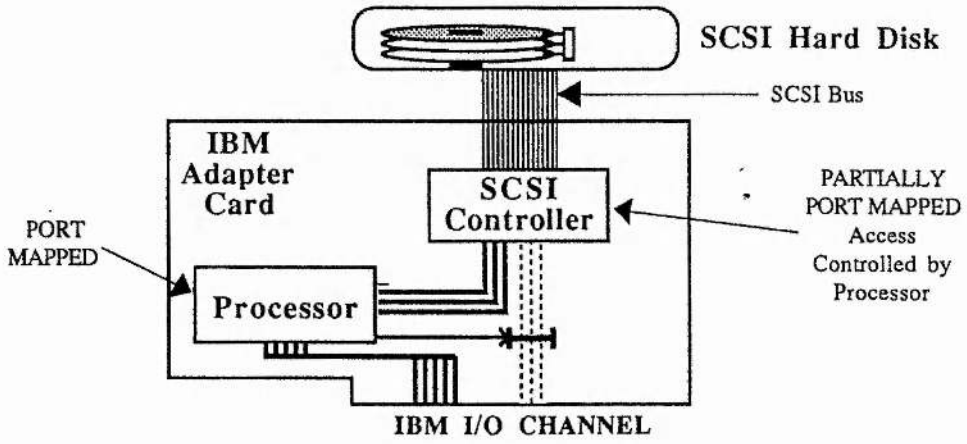


Figure 3.0 : Block Diagram of the Prototype Implementation

3.0.0 IMPLEMENTATION

VERSION 1

This version was designed using purely discrete logic and 'standard' components. It was further designed such that the level of access restriction between the computer and the disk was fully configurable by a series of jumper settings. For full circuit diagrams see the appendices.

Elements of the Design include :

[1] The Z80 Subsystem

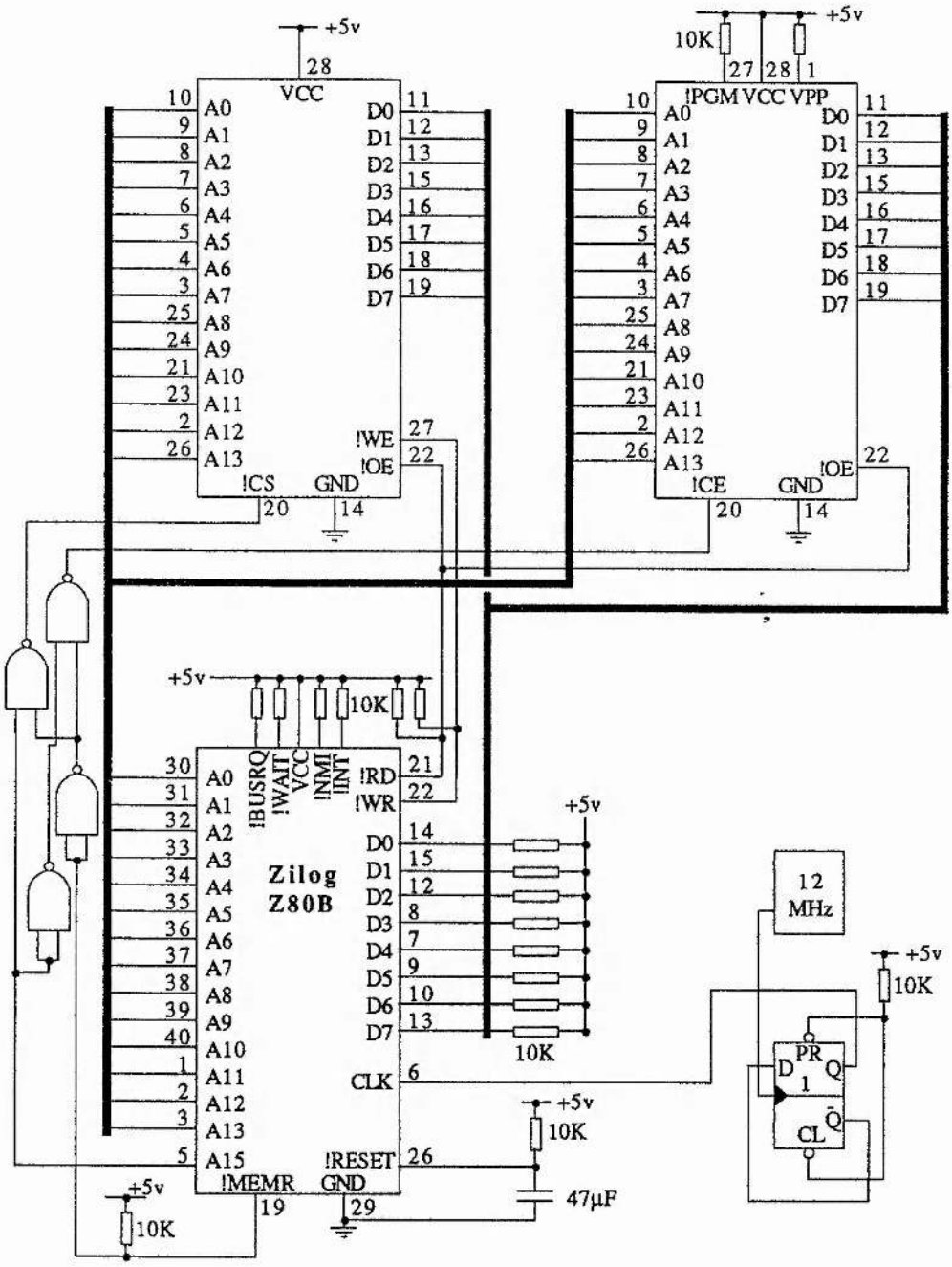


Figure 3.1 : The Z80 Subsystem

A standard Zilog Z80B was used as the essential dedicated microprocessor which is remote from, and cannot be controlled by, any virus. Clearly the choice of processor for any implementation depends on the required performance. For a prototype, the Z80 is ideal : it is simple both to design with, and to program. Since performance at this research level was not of great importance, the Z80 was ideal. It should be noted that 20MHz Z80 processors now exist, with 16 bit data buses and enhanced instruction sets, so an upgrade path does exist.

An 8K Byte Read Only Memory (ROM) was used, since the BIOS could potentially require considerable code space. As the requirement for Random Access Memory (RAM) was not well known at the early development stage an 8K Byte Static RAM (SRAM) was made available.

The memory decoding is simple because the memory map consists of the RAM and ROM only. The ROM is placed in the lowest 8K Bytes in the memory map as required by the Z80, since it begins execution at location 0000H on power up. The RAM is placed in the upper half of the address space, in the 8K Byte block beginning at 8000H.

The sockets are wired so that either or both the RAM or ROM may be replaced by 16K Byte versions with no hardware modification. The ROM would exist with its base at 0000H, and the RAM with its base at 8000H, as before.

[2] The SCSI Interface

The NCR 5380 SCSI Controller provides the hardware required in order to access and control a SCSI hard disk drive. Although an early generation controller, it is both tried

and tested. It provides the possibility of 1.5 megabytes per second data transfer rate, although this is heavily dependent on the computer system to which it is attached. When measuring system performance however, operating system overheads, reduce this rate considerably.

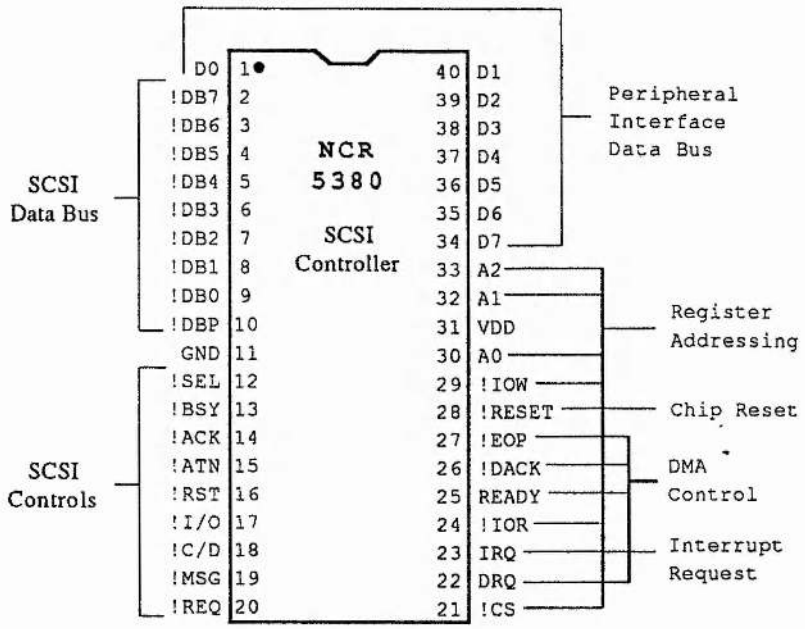


Figure 3.2 : The NCR 5380 SCSI Interface Chip

The chip itself provides an eight bit, bi-directional data bus used during data transfer, and in configuring the chip's internal registers. It uses three lines to address the internal registers and control lines to facilitate both Programmed I/O and DMA transfers. The chip, when operating, produces all necessary signals via dedicated control and data pins, to communicate with a device using the SCSI protocol, as defined by the ANSI X3T9.2 Committee.

The chip is controlled by reading and writing several internal registers. Depending on how this chip is configured it may take on either an initiator or a target role. It is also capable of generating an interrupt to indicate either a task completion or an abnormal bus occurrence. Full arbitration is also supported, facilitating the implementation of a multiple initiator system, if required.

This chip provides the possibility of data transfer between SCSI bus devices on one of four modes :

[1] Programmed I/O

This is the most primitive form of transfer provided. The SCSI handshake signals (REQ and ACK) are monitored and asserted by accessing the appropriate bits within the internal registers of the chip.

Basic handshaking is implemented via these signals. For example during a Write sequence, the initiator loads a byte into the Output Data Register, and waits for the REQ signal to become active by monitoring the REQ bit of the internal STATUS register. When this occurs, the initiator activates the ACK by setting the ASSERT/ACK bit in the INITIATOR COMMAND internal register. The initiator then waits for REQ to become false, at which time it resets ASSERT /ACK. This sequence is repeated for each byte transferred.

This type of transfer is usually reserved for the transferring of small volumes of data such as command blocks, message bytes, or status bytes as it is very slow.

[2] Normal DMA

In this mode the chip outputs a DMA request via the external pin DRQ whenever it is ready for a byte transfer. Hardware is required both to identify such a request, and to reply via the external pin DACK. DRQ becomes inactive some time after DACK is asserted and the resetting of DACK must follow after that.

This mode is normally used for transferring large volumes of data.

[3] Block Mode DMA

This mode is similar to Normal DMA mode except that it allows an external DMA device to perform DMA transfer without relinquishing the data bus to the CPU. This provides no better a handshake mechanism, but, as the DMA controller has full control of the bus, the DMA data transfer rate does increase.

[4] Pseudo DMA

This mode is not technically a separate mode implemented within the chip, rather it is a technique that may be used to interface the chip to a microprocessor. The chip is programmed to transfer using the Normal DMA mode, but instead of using a DMA controller the microprocessor emulates the DMA handshake. Hardware is required to allow the microprocessor to monitor DRQ and generate DACK as required.

Implemented correctly, this mode will provide significant increase in performance over Programmed I/O, without the requirement of a DMA controller. In some computer systems this method may in fact prove to be faster than true DMA.

To summarise, this chip provides all that is required to communicate with any asynchronous SCSI bus device, provided a transfer rate above 1.5 Megabytes per second is not required.

[3] IBM ROM BIOS

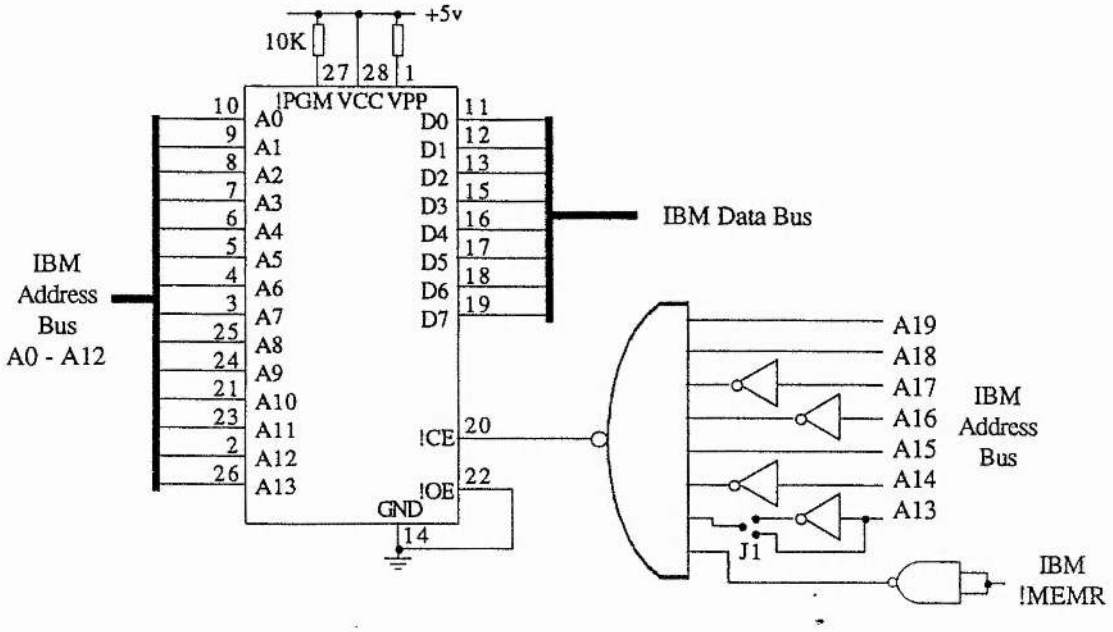


Figure 3.3 : Memory Mapped IBM BIOS ROM

A replacement section of the IBM's BIOS is required to override the existing INT 13H software Interrupt which provides all low level hard disk services. This requires the inclusion of an 8K byte ROM in the design, memory mapped into the IBM beginning at either address C8000 or CA000. In the present design the choice of address is effected by a jumper on the board.

[4] Passing Control of SCSI Between the Supervisor and the IBM

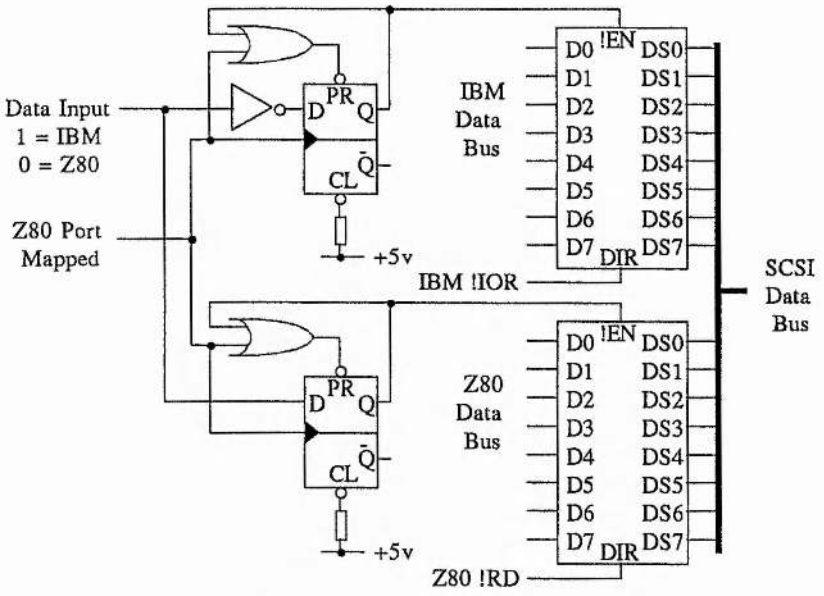


Figure 3.4 : Control Switchover Logic

With the use of two D-Type flip-flops, one must guarantee that during the changeover of control, the Supervisor and the IBM are never both in control at once, even for an instant, as this would cause a system crash. The logic required for this is illustrated in figure 3.4, and the associated timing diagram is given in figure 3.5.

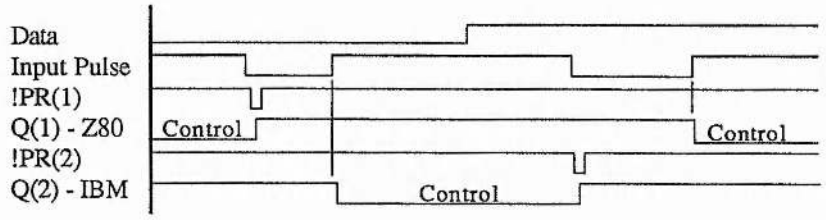


Figure 3.5 : Timing Diagram for Control Switchover Logic

This ensures that there will be no data bus or address bus clashes, nor any unexpected values generated on any of the SCSI control lines.

[5] Control of the Data Bus

The use of bus transceivers 74F245 allows for either the Supervisor or the IBM to access the SCSI controller. The control switchover logic decides which transceiver is enabled, and decoding of the read signal of the active processor decides the direction of data flow.

[6] Control of the Address Bus

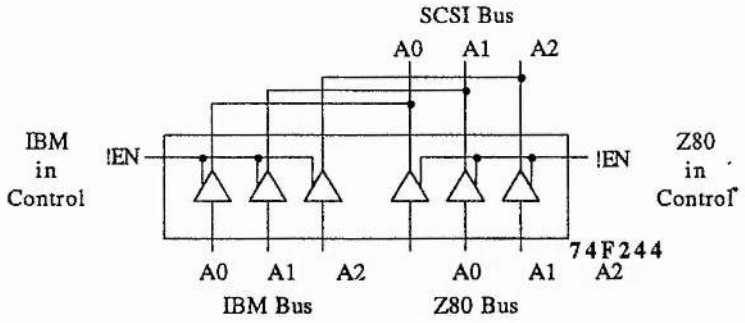


Figure 3.6 : Address Bus Multiplexing

A single 74LS244 octal buffer chip provides all the necessary logic to allow either the Supervisor or the IBM access to the address lines of the SCSI controller, depending on which is in control, see figure 3.6.

[7] IBM to SCSI Internal Register Access Logic

As mentioned previously, this card was designed such that any combination of internal registers could be made accessible to the IBM. This logic provides the required hardware mechanism. 74LS138, 3 to 8 decoder, chips generate a separate signal for each of the eight possible addresses from the values present in the lowest three bits of the IBM address bus. Read and Write attempts to the internal registers are differentiated by using two separate decoder chips. The remaining IBM address bits are used to map the internal registers into the IBM port space in either the range 300H - 307H or 320H - 327H depending on the setting of another hardware jumper.

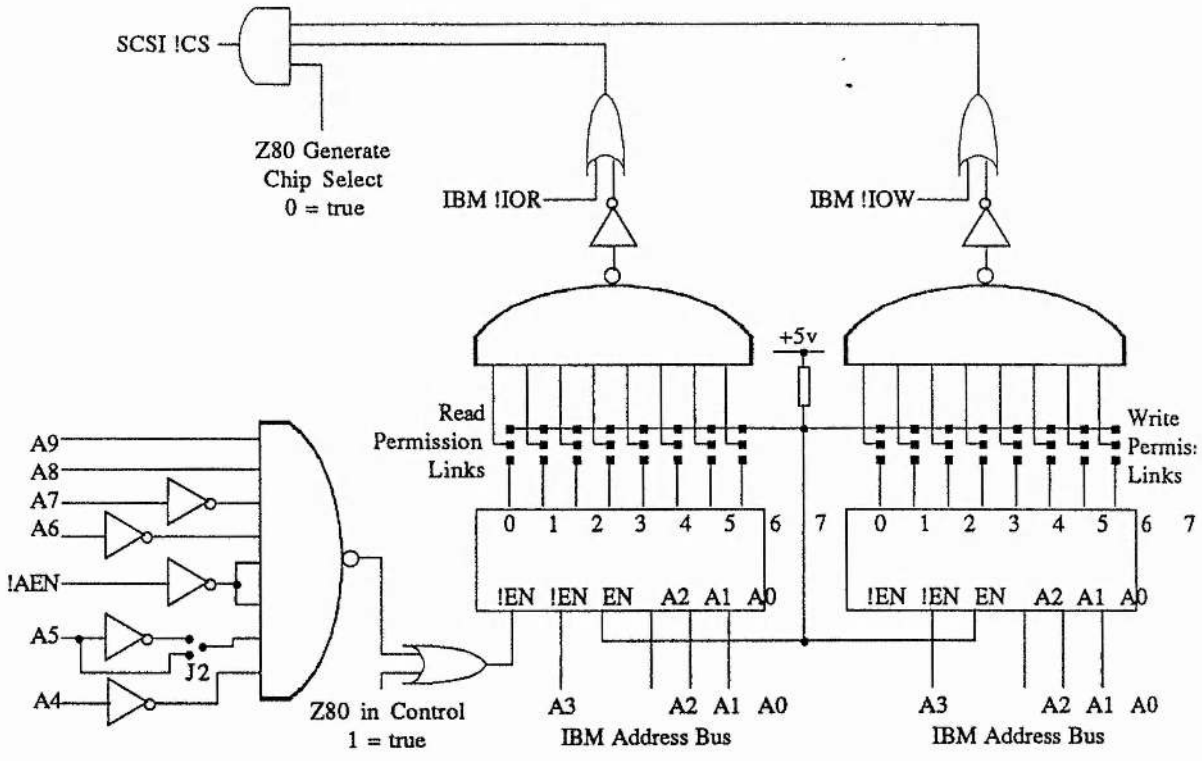


Figure 3.7 : IBM to SCSI Internal Access Logic

If the IBM has control (see part [4], 'Passing Control of SCSI Between the Supervisor and the IBM'), the upper address bits indicate the correct port range and a port Read or Write operation is attempted, then one of the outputs of one of the decoder chips becomes active. If this output has a connecting jumper then the signal propagates through the remaining logic and activates the required control lines on the SCSI controller. If, however, the jumper is connected to five volts, and not the output of the decoder, then the signal proceeds no further and the desired operation is not performed on the SCSI controller. Thus any internal register may be made accessible or denied to the IBM by configurable hardware means.

[8] IBM to SCSI Pseudo DMA Implementation

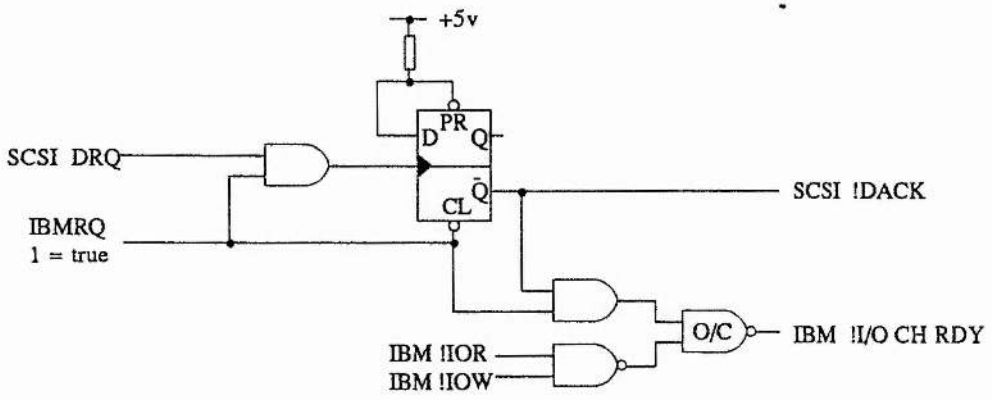


Figure 3.8 : IBM to SCSI Pseudo DMA Logic

This circuitry provides all the necessary hardware for data transfer between the IBM and the SCSI controller using Pseudo DMA.

It operates as follows :

[i] The SCSI controller is configured for DMA and transfer started ;

[ii] The controller either requests a byte or indicates a byte is available activating the DRQ line (active high) ;

[iii] The IBM either requests a byte or indicates a byte is available by activating the IBMRQ line (active high) ;

[iv] A byte is transferred.

If [ii] comes before [iii] then when IBMRQ becomes active it causes a rising edge on the CLK input of the flip-flop and the 'Q bar' output goes low. This in turn activates the DACK pin of the controller. When the controller registers the activation of DACK it causes DRQ to become inactive, then, when IBMRQ becomes inactive at the end of the read or write cycle of the IBM, it clears the flip-flop and DACK becomes inactive, thus completing the handshake.

If [iii] comes before [ii] then when IBMRQ becomes active, no rising edge is generated on the CLK input of the flip-flop and therefore DACK remains inactive. Instead I/O CH RDY is activated, suspending the IBM's operation. When DRQ finally becomes active, the rising edge is generated on the CLK input of the flip-flop, DACK is made active, and I/O CH RDY is inactivated. Operation then proceeds as normal.

[9] IBM to Supervisor Communication

A single eight bit latch (74LS373) is used as a communications mechanism from the IBM to the Supervisor. The latch is placed in the IBM's port space at either 309H or 329H. When the IBM writes to it the value is stored and a flip-flop is set. The latch output is placed in the Supervisor's port space, and when the Supervisor reads it the flip-flop is reset. This flip-flop acts as a flag to indicate the state of the latch to both systems. The Supervisor may read this flag to see if a byte has been sent by the IBM, and the IBM may read this flag to see if the Supervisor has read the last byte it sent. The flag output is also, therefore, mapped into both the IBM and the Supervisor port space.

[10] Supervisor to IBM Communication

Status information may be passed to the IBM from the Supervisor using a latch similar to that mentioned in the previous paragraph ([9]). Only six of the bits of the latch are used so that two are spare for other status generated on the board. One of these spare bits is used to represent the value of the flag mentioned in [9] and the other remains unconnected but was used in development to monitor various signals within the system. A flag is not required for this communication because the IBM could always, if necessary, write to the latch in [9] to inform the Supervisor that it had read this latch.

BUILDING AND TESTING

The card was built in distinct sections, and at each stage the operation of the hardware was tested. This modular approach simplifies error detection as faults will lie either in the newest, untested section, or in the interface between it and other tested sections. During the construction of each section, each wire was checked with a continuity meter after wrapping in order to reduce the chance of errors due to loose or incorrect connections.

STAGE 1 : Standard SCSI Interface Card

The circuitry involving the interfacing between the IBM and the SCSI controller was constructed first. The Changeover control logic was hardwired so that the IBM was in permanent control. Firstly this section was built and the connections verified. Next a ROM was placed in the IBM BIOS socket which was programmed to write purely a message to the screen on boot. When the message appeared, it indicated that the ROM was correctly mapped into the memory map of the IBM and that all address, data and control lines to and from it were correct.

Next, all chips in that section were inserted, together with an INT 13H BIOS ROM configured to the hardware implementation. When the machine booted successfully, with the card inserted and attached to a disk, and allowed the disk to be accessed, it meant that this section was working correctly.

Lastly, a .BAT batch file was written which repeatedly made copies of a set of files and deleted the originals. This was then executed and allowed to run for many hours. After thousands of copies, no error was detected by the operating system and the files were entirely uncorrupted. Hence not only was the section working, but also no intermittent errors existed which could be caused by a timing problem or loose connection.

STAGE 2 : The Z80 Subsystem

This section was built in quite small distinct sections, so each could be tested separately.

Firstly all address, control and data lines between the Z80, RAM and ROM, including memory decoding and system clock generation were wired.

The simple code shown in Figure 3.9 was used to test whether this section was working correctly. The code repeatedly writes the value A5H into an area of the RAM, and reads these values back. If this works, then the Z80 is executing the program from ROM correctly and by default is itself functioning correctly. If A5H is read back then the RAM is also being written to and read from correctly.

```
;Z80 test program [1] RAM/ROM

rambase     equ   0E000h

            org   0000h       ; ensures ROM starts at 0000h
start       ld    a,0A5h      ; load A5h into accumulator
            ld    hl,rambase  ; load base of RAM into reg pair HL
loop        ld    (hl),a      ; write out to address in HL
            ld    a,(hl)      ; read back value
            inc   hl          ; increment address
            jr    nz,loop     ; repeat until top of RAM reached
            jr    start       ; repeat forever

            end   start
```

Figure 3.9 : Z80 RAM/ROM Test Program

A Logic Analyser is used to monitor firstly the control lines to check for enabling of both RAM and ROM, and then the data bus during both Read and Write cyles of the Z80, checking for A5H. As a last check, the value A5H is replaced first with 00H, then FFH to check all bits on the data bus can be driven both high and low. This checks that all bits are 'mobile', ie no stray connections exist to fix a data line's value.

Then the two latches used for IBM-Z80 communication, together with all the associated decoding logic and the status flag were wired up. It was important to check that both latches were being written to and read from correctly and that the status flag worked correctly for the IBM to Z80 latch. To facilitate this, the code in Figure 3.10 was generated for the Z80 ROM.

```
;Z80 test program [2] Z80-IBM

IBMDataR    equ   08h
IBMStatR    equ   09h
PathChW     equ   08h
StatusW     equ   09h

Z80Path     equ   00000000b
IBMPath     equ   00000001b
StatMask    equ   00000001b

            org   0000h


start       in    a,(IBMDataR)      ; Clear status flag
            ld    a,0000h           ; set status reg to 00h
            out   (StatusW),a
again       in    a,(IBMStatR)      ; Get Status
            in    a,(IBMDataR)      ; Read Data
            out   (StatusW),a       ; Send Back to IBM
            jp    again

            end   start
```

Figure 3.10 : Z80-IBM Port Communication Test Program

This code reads a value written by the IBM to the 'input' latch and writes it back to the 'output' latch where the IBM can then read it. If the value is first written by the IBM, read and then written by the Z80, and finally read by the IBM and remains intact, then both latches function correctly. Unfortunately only six bits of the Z80 output latch are mapped to the Z80 bus so only the top six bits can be tested in this manner. Also, we have no absolute indication that the status flag is functioning correctly. A logic analyser was again used to check the lowest two bits of the data bus and to test the logic of the

flip flop used to implement the status flag. Figure 3.11 shows the logic analyser output for the status flag for a single IBM write, Z80 read cycle. One should notice two important features : the IBM does not receive a 'ready' indication from the flag until the end of the Z80 read cycle ; and data is not marked 'available' to the Z80 until the end of the IBM cycle. The logic used to implement this dual flag thus ensures that, with correct monitoring, no corruption is possible due to mutual access of a latch by both the IBM and Z80.

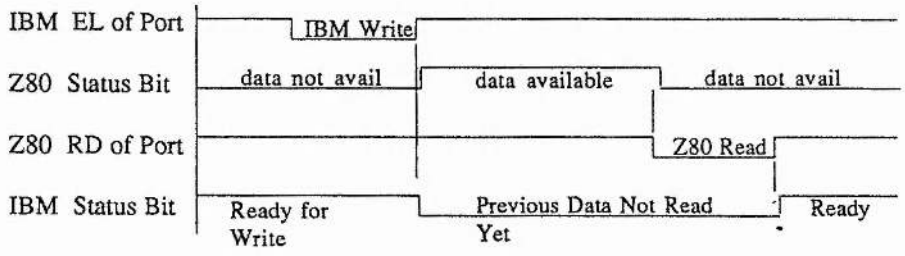


Figure 3.11 : Status Flag Logic Analyser Output

Lastly, the Z80 was wired up to the SCSI controller and the Control Switchover Logic hardwired so that the Z80 was in control. Code was then executed to cause a low level format of the hard disk. When this worked it indicated that a communications link between the Z80 and SCSI, using both control and data lines, had been established.

STAGE 3 : Connecting the Two Sections Together

Once stage one and stage two are complete the next logical step was to connect the two together. In hardware terms this required the connections between the Z80 and the Control Switchover Logic to be made. A simple test was then made : the code in figure 3.12 was used for the Z80 ROM causing control to be passed to the IBM. If the IBM could then boot and recognise and allow access to the disk then this would indicate at least partial success.

```
;Z80 test program [3] Pass Control to IBM

PathChW  equ    08h
SCIResW  equ    0ah

Z80Path  equ    00000000b
IBMPath  equ    00000001b

         org    0000h

start    out    (SCIResW),a          ;Reset SCSI Chip
         ld     a,IBMPath            ;Choose path to IBM
         out    (PathChW),a          ;Set Path

         end    start
```

Figure 3.12 : Firmware Control Switchover by the Z80

Once the last test was concluded, a more advanced version was used. The code in figure 3.13 was used causing the Z80 to execute certain code sections if a command byte was sent by the IBM. The code provided full control switchover ability, allowing control to be granted or denied to the IBM simply, through the DEBUG utility provided with MS-DOS.

```
;Z80 test program [4] Command Byte Driven Control Switchover

IBMDataR    equ   08h
IBMStatR    equ   09h
PathChW     equ   08h
StatusW     equ   09h
Z80Path     equ   00000000b
IBMPath     equ   00000001b

            org   0000h
start       in    a,(IBMDataR)       ; Clear status flag
            ld    a,0000h            ; set status to 00h
            out   (StatusW),a
retry       in    a,(IBMStatR)       ; Get Status
            bit   0,a                ; Test for byte available
            jr    z,retry            ; Repeat if no byte yet

            in    a,(IBMDatR)        ; Read the byte

            cp    01h                ; Set Control for Z80
            jp    z,z80pth

            cp    02h                ; Set Control for IBM
            jp    z,ibmpth

            ; Space for other Commands to be inserted!

            jp     retry             ; should never get here

ibmpth      ld    a,IBMPath          ; Set pathways for IBM
            jr    setpath

z80pth      ld    a,Z80Path          ; Set pathways for Z80

setpath     out   (PathChW),a        ; Set the control
            jp    retry              ; Await next instruction

            end   start
```

Figure 3.13 : Control Switchover Under IBM Control

If control was set to IBM then access should be possible and if control is set to Z80 then access should not be possible. When this proved to be the case hardware testing had been successfully completed.

Such a comprehensive building and testing sequence while time-consuming was worthwhile as errors were easy to detect at this stage, and it provided a certain feeling of confidence in the hardware. Such confidence reduces the time required to detect the cause of future problems encountered, since the possibility of hardware error can be ignored until all other possibilities have been exhausted.

VERSION 2

This version was designed to enhance the performance of the implementation and to remove much of the excess configuration hardware provided in version 1. The hardware requirements in this aspect were well defined by this point. This version was also designed to fit on a half size card and to have a significantly reduced component count to help in PCB prototype development at a later date.

This version of the hardware is functionally identical to its predecessor. An IBM BIOS ROM, Z80 subsystem, SCSI controller, and all required inter-connections are still present but much of it appears in a different form, is more economical and results from the research previously described.

A Z84C50 Superintegration Z80, from Zilog, was used as a replacement for the standard Z80B used in version 1. The reason for this was twofold. Firstly, the Z84C50 provides a 2K byte internal, zero wait state RAM and, when consulting a reasonably complete prototype implementation, using version 1, 2K Bytes was found to be quite adequate. Having this RAM on chip meant significant speed increase coupled with reduced chip count. Secondly, the Z84C50 operates at up to 10MHz providing an overall increase in performance.

All 'glue' logic used in version 1 was reduced to five 16V8 GAL devices and two 74ALS74 chips. This allowed the board to be implemented either as a half card or, a full length card, where the excess space could be used, for example, to provide full floppy control. The resulting reduction in chip count also meant that PCB fabrication costs would be less as well as the cost of board population, should the board ever be produced commercially.

The five GAL devices provide the required logic for the following :

GAL 1 provides decoding logic to map the ROM BIOS into the IBM memory map. It also provides tri-state buffering of the output of the status latch, to the IBM data bus.

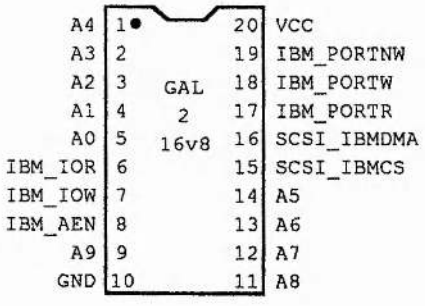
```
                DEVICE 16V8;
        TITLE  SUPERVISOR  PROJECT  GAL1
             NAME  D.  S.  S.  Robb;
               SIGNATURE C2G1;

                          ┌──────┐
      STATUS_BIT  1●      │      │ 20  VCC
  Z80_READ_DATA   2       │      │ 19  unused
       IBM_PORTR  3       │ GAL  │ 18  unused
             A13  4       │  1   │ 17  unused
             A14  5       │16v8  │ 16  IBM_MEMR
             A15  6       │      │ 15  IBM_D0
             A16  7       │      │ 14  unused
             A18  8       │      │ 13  A17
             A19  9       │      │ 12  unused
             GND  10      │      │ 11  MEMR
                          └──────┘

!IBM_MEMR  = A19 & A18 & !A17 & !A16 & A15 & !A14 & !A13 & !MEMR;

IBM_D0.OE  = !IBM_PORTR;
IBM_D0     = !STATUS_BIT;
```

Figure 3.14 : GAL 1 Description for Supervisor Circuit Version 2

The output IBM_MEMR is brought low if a memory Read request is made, by the IBM, to any location from C8000H to C9FFFH in the IBM's memory map. IBM_D0 output is only enabled if IBM_PORTR is active, which occurs when the Z80 attempts to read the status flag. When enabled, it takes the value of the status flag which is driven onto the Z80 data bus on D0.

GAL 2 provides access by the IBM to a predefined subset of the SCSI controller's internal registers. The choice of registers was taken from the working prototype using version one. It further provides the pseudo-DMA decoding logic for data transfer to and from the SCSI controller. Finally, it provides decoding to map the status flag and the two latches used to communicate with the Z80, into the IBM port space.

```
DEVICE 16v8;
TITLE  SUPERVISOR  PROJECT  GAL2
     NAME  D.  S.  S.  ROBB;
        SIGNATURE C2G2;
```




```
SCSI_IBMCS  = ( (!IBM_IOR & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & !A3
                  & !A2 & !A1 &  A0) /* READ P1  */
              |(!IBM_IOR & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & !A3
                  &  A2 & !A1 &  A0) /* READ P5  */
              |(!IBM_IOW & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & !A3
                  & !A2 & !A1 &  A0) /* WRITE P1 */
              |(!IBM_IOW & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & !A3
                  &  A2 & !A1 &  A0) /* WRITE P5 */
              |(!IBM_IOW & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & !A3
                  &  A2 &  A1 &  A0) /* WRITE P7 */
            );


SCSI_IBMDMA = ( (!IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & A3 & !A2 &
                 !A1 & !A0) & (!IBM_IOR | !IBM_IOW));

!IBM_PORTR  = (!IBM_IOR & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & A3 &
               !A2 & !A1 & A0);

!IBM_PORTW  = (!IBM_IOW & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & A3 &
               !A2 & !A1 & A0);

IBM_PORTNW  = (!IBM_IOW & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & A3 &
               !A2 & !A1 & A0);
```

Figure 3.15 : GAL 2 Description for Supervisor Circuit Version 2

The output SCSI_IBMCS is activated if the IBM either writes to ports 321H, 325H or 327H, or reads from ports 321H or 325H. These are the minimum ports required for SCSI data transfer and do not include any which could be used to invoke a SCSI command. The output SCSI_IBMDMA is activated if the IBM either reads from, or writes to, port 328H. This port is used to transfer actual data bytes and SCSI_IBMDMA is used to generate a DACK in the pseudo-DMA cycle. IBM_PORTR and IBM_PORTW are activated when a request is made to port 329H to read or write, respectively. These lines are used to activate the latches used in communication with the Z80. IBM_PORTNW is the same as IBM_PORTW but of the opposite polarity, as this is required elsewhere and no discrete inverters are available.

GAL 3 multiplexes between the Z80 and IBM address buses, to the SCSI controller address bus.

```
          DEVICE  16v8;
TITLE  SUPERVISOR  PROJECT  GAL3
     NAME  D.  S.  S.  ROBB;
       SIGNATURE C2G3;

                 ┌──────────┐
    Z80_A0  1 ●          20  VCC
    Z80_A1  2            19  unused
    Z80_A2  3    GAL     18  unused
    IBM_A0  4     3      17  SCSI_A2
    IBM_A1  5   16v8      16  SCSI_A1
    IBM_A2  6            15  SCSI_A0
  IBM_PATH  7            14  unused
    unused  8            13  unused
    unused  9            12  unused
       GND 10            11  unused
                 └──────────┘

SCSI_A0    =  ( (!IBM_PATH & Z80_A0) |( IBM_PATH & IBM_A0) );

SCSI_A1    =  ( (!IBM_PATH & Z80_A1) |( IBM_PATH & IBM_A1) );

SCSI_A2    =  ( (!IBM_PATH & Z80_A2) |( IBM_PATH & IBM_A2) );
```

Figure 3.16 : GAL 3 Description for Supervisor Circuit Version 2

The logic used, and the operation thereof, of this GAL is self-explanatory. If the IBM is in control then the IBM address lines drive the SCSI controller address bus, otherwise the Z80 address lines do.

GAL 4 multiplexes between the Z80 and and IBM control lines, to the SCSI controller. Furthermore it provides logic which enables either, but never both, transceivers, which connect either the Z80 or IBM data bus, to the SCSI controller data bus. Finally, it provides logic which, together with a single D-Type flip-flop, provides the required electronics to implement the wait state generator used in version one.

```
DEVICE 16v8;
TITLE  SUPERVISOR  PROJECT  GAL4
     NAME  D.  S.  S.  ROBB;
       SIGNATURE C2G4;

                    GAL
                     4
                   16v8

SCSI_IBMDMA  1                20  VCC
 SCSI_IBMCS  2                19  Z80_TEN
 SCSI_Z80CS  3                18  IBM_TEN
   SCSI_DRQ  4                17  SCSI_IOW
   IBM_PATH  5                16  SCSI_IOR
    IBM_IOR  6                15  SCSI_CS
    IBM_IOW  7                14  DMA_CLK
     Z80_RD  8                13  IO_CH_RDY
     Z80_WR  9                12  DMA_NQ
         GND 10               11  unused

!IO_CH_RDY  =  (DMA_NQ & SCSI_IBMDMA);

DMA_CLK     =  (SCSI_DRQ & SCSI_IBMDMA);

!SCSI_CS    =  ((SCSI_Z80CS & !IBM_PATH)|(SCSI_IBMCS & IBM_PATH));

!SCSI_IOR   =  ((!Z80_RD  & !IBM_PATH)|(!IBM_IOR & IBM_PATH));

!SCSI_IOW   =  ((!Z80_WR  & !IBM_PATH)|(!IBM_IOW & IBM_PATH));

!IBM_TEN    =  ((SCSI_IBMCS  & IBM_PATH)|(!DMA_NQ & IBM_PATH));

!Z80_TEN    =  (SCSI_Z80CS & !IBM_PATH);
```

Figure 3.17 : GAL 4 Description for Supervisor Circuit Version 2

The output IO_CH_RDY is used to generate IBM wait states by connecting it via an open collector buffer to the I/O_CH_RDY line of the IBM. This line becomes active if the controller DRQ is not active, indicated by DMA_NQ, and the IBM is attempting to generate a DACK, indicated by SCSI_IBMDMA. Once the controller is ready it will activate DRQ and the wait state generation will cease. The controller's DACK line is activated if DMA_NQ goes low which occurs when a rising edge is generated on DMA_CLK. Such a rising edge will occur only when the controller's DRQ is active and the IBM is attempting to transfer, indicated by SCSI_IBMDMA.

The outputs SCSI_CS, SCSI_IOR, and SCSI_IOW all operate in a similar manner. If the IBM has control then the values of the IBM inputs, which are intended to drive these control lines, are driven out, otherwise the Z80 input values are driven out.

The remaining two outputs are used to activate the data bus transceivers which allow either the IBM or Z80 data bus to connect with the SCSI controller data bus. The IBM's transceiver is activated if it is both in control and is attempting either a Read/Write of a valid SCSI controller internal register, or a DMA data transfer. The Z80's transceiver is activated only if the IBM is not in control and the Z80 is attempting a Read/Write of a SCSI controller internal register. Clearly, from this logic, both transceivers cannot be active at the same time.

GAL 5 provides all the decoding logic required by the Z80 : it maps the Z80 ROM into the Z80 memory map ; it maps the two latches, the status flag, the SCSI reset line, and the control switchover flip-flop, into the Z80 port space.

```
                DEVICE 16v8;
          TITLE  SUPERVISOR  PROJECT  GAL5
               NAME  D.  S.  S.  ROBB;
                 SIGNATURE C2G5;

                        GAL
                         5
                       16v8

      Z80_A0  1                20  VCC
      Z80_A1  2                19  Z80_ROM_EN
      Z80_A2  3                18  SCSI_RESET
      Z80_A3  4                17  Z80_D0
    Z80_IORQ  5                16  Z80_READ_DATA
      Z80_RD  6                15  PATHCH
      Z80_WR  7                14  Z80_WRITE_DATA
    Z80_MREQ  8                13  Z80_CS
     Z80_A15  9                12  unused
         GND  10               11  STATUS_BIT

Z80_WRITE_DATA  =  !Z80_WR & !Z80_IORQ & Z80_A3 & !Z80_A2 & !Z80_A1 &  Z80_A0;

!PATHCH         =  !Z80_WR & !Z80_IORQ & Z80_A3 & !Z80_A2 & !Z80_A1 & !Z80_A0;

!SCSI_RESET     =  !Z80_WR & !Z80_IORQ & Z80_A3 & !Z80_A2 &  Z80_A1 & !Z80_A0;

!Z80_READ_DATA  =  !Z80_RD & !Z80_IORQ & Z80_A3 & !Z80_A2 & !Z80_A1 & !Z80_A0;

Z80_D0.OE       =  !Z80_RD & !Z80_IORQ & Z80_A3 & !Z80_A2 & !Z80_A1 &  Z80_A0;
Z80_D0          =  STATUS_BIT;

!Z80_ROM_EN     =  !Z80_MREQ & !Z80_A15;

Z80_CS          =  !Z80_A3 & !Z80_IORQ;
```

Figure 3.18 : GAL 5 Description for Supervisor Circuit Version 2

The following ports are allocated using the logic within this GAL : port 08H (Write) which causes data to be written to the latch which is read by the IBM ; port 09H (Write)

which causes the IBM to be in control if the lowest bit of the Z80 data bus is high, or the Z80 to be in control if it is low, by writing to the control switchover flip-flop ; port 0AH (Write) which causes a reset pulse on the SCSI controller ; port 08H (Read) which causes the contents of the latch the IBM writes to to be placed on the Z80 data bus ; and port 09H (Read) which causes the value of the status flag to appear on the lowest bit of the Z80 data bus. Also, ports 00H to 07H are mapped to the internal registers of the SCSI controller.

The output Z80_ROM_EN is activated on any memory Read request to the lowest 32K bytes of memory.

BUILDING AND TESTING

As in version 1, this circuit was built in distinct stages. At the end of each stage both, physical connection and functionality were tested.

Firstly, the IBM ROM BIOS was wired up, together with GAL 1 and the IBM address bus buffers. Following the same method as used in version 1, the ROM was programmed to write a message on the screen on boot, and when this occurred it indicated correct operation of this section.

Next, the SCSI controller was wired up and connected into the IBM. This required the wiring of the IBM data bus transceiver, the pseudo DMA flip-flop, GAL2, GAL3 and GAL4. It also required, what would become the output of the control changeover flip-flop, to be tied high through a resistor so that the IBM would be in permanent control. An INT 13H BIOS ROM configured to the hardware implementation was then inserted. When the machine booted successfully, with the card inserted and attached to a disk, and allowed the disk to be accessed, it meant that this section was working correctly.

Lastly the Z84C50, its ROM, GAL5, the status flag, the control changeover flip-flop, and both IBM communication latches were wired. As in version 1, the IBM / Z80 inter-communication was tested to ensure both latches and the status flag operated correctly. Once this was done, slightly modified versions of both fully operational prototype ROMs from version 1 were inserted, and a full test executed.

As fully operational firmware was available by completion of the hardware, testing was a simple matter. A ROM containing the working firmware developed for version 1 was used and the board put through exhaustive software testing. When it functioned identically to version 1, allbeit with increased performance, it proved itself to be an upgraded replacement.

3.1 FIRMWARE

3.1.0 BACKGROUND INFORMATION

[1] MS-DOS BIOS Implementation

ROM Format

A ROM BIOS loader routine is executed on power up. This routine searches the ROM space above the 640K byte limit for any additional ROMs. The location of such ROMs is illustrated in figure 3.19

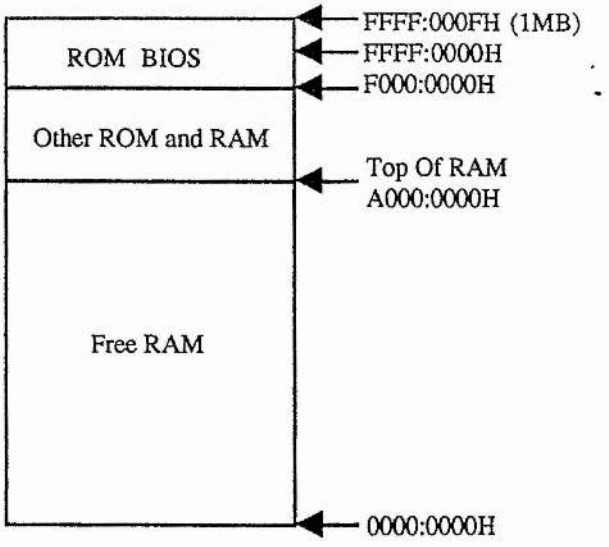


Figure 3.19 : Memory Layout at Startup

The BIOS loader routine scans memory from A000:0000H to F000:0000H for the occurrence of a particular sequence of signature bytes. If an additional ROM BIOS is found then its initialisation routine is called. Scanning involves checking the first two

bytes of each 512 byte block for the signature AA55H. If this signature is found then the third byte indicates the ROM size, in number of 512 byte blocks. The ROM is then checksummed and if the least significant byte is found to be zero then the BIOS is initialised.To initialise the BIOS, the IBM then executes the fourth byte.

BIOS Execution

As mentioned above, a ROM found somewhere between A000:0000H to F000:0000H in memory, with the signature AA55H at the start, has its fourth byte executed on power up. A ROM configured in this manner will therefore usually place a jump to an initialisation routine at this location. The initialisation routine is often present to reconfigure the machine to execute other routines within the ROM, when certain software interrupts are encountered.

Software Interrupts

A series of so called 'software' interrupts exist as part of the MS-DOS operating system. To invoke a software interrupt some required information is placed by an active program within the general purpose registers of the processor and then the interrupt executed by the active program with the INT command. This, in effect, is merely a far call to a routine whose address is found within an interrupt table. For example, INT 13H provides all disk services, hence in order to read from a disk the Read command, which is function 02H, would be placed within the high byte of register AX. One would then place values representing the start address, the number of sectors to be read, and the drive to read from within other defined registers and execute an INT 13H instruction.

The initialisation routine of a ROM BIOS will usually alter certain pointers within the software interrupt table so that they point to replacement routines within that BIOS.

[2] Small Computer System Interface (SCSI)

The Concept

The SCSI interface is a parallel, multimaster I/O bus that provides a standard interface between computers and peripheral devices. Its goal was to provide a non-proprietary interface. Manufacturers build disk drives with varying numbers of cylinders, heads and sectors,with the result that separate drivers were required for each drive type. SCSI overcomes this limitation because the host system can perceive any disk drive, regardless of manufacturer, as a string of consecutive logical blocks. Therefore one variable, the maximum addressable block, fully defines a SCSI drive and this is available by issuing a Read Capacity command. By placing a certain amount of intelligence, required by SCSI, on a drive, and using it to convert between logical addresses and physical ones, a single driver can be used to communicate with any drive.

Features of SCSI

All commands issued to a SCSI drive contain logical addresses. As mentioned above, the drive appears to the SCSI host as a series of contiguous sectors. SCSI therefore deals with all facets of the logical to physical conversion. This includes the mapping out of all defects on the drive which exist when the drive leaves the factory and are generated after the event. A SCSI drive always appears defect free to the host.

SCSI supports not only multiple hosts but also several commands to different peripherals which may be in operation at the same time : an initiator may issue a command and disconnect from the bus until the peripheral is ready to respond, thus allowing another initiator to use the bus or the same initiator to issue another request to another device. SCSI can therefore perform complex concurrent I/O operations.

During normal operation the SCSI bus makes orderly transitions between bus states known as phases. In the following implementation the SCSI bus will be required to support only one host and therefore no arbitration for control of the bus is required.

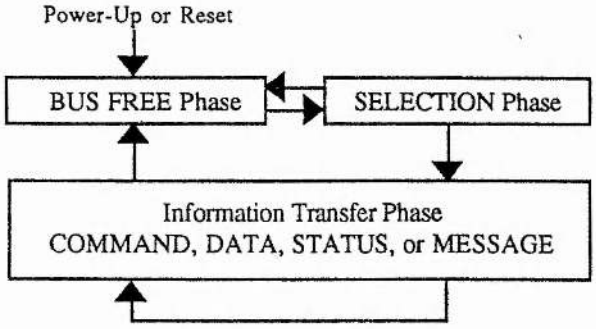


Figure 3.20 : SCSI Phases in a Single Host System

Eight possible SCSI phases exist : BUS FREE, ARBITRATION, SELECTION, RESELECTION, COMMAND, DATA, STATUS and MESSAGE. The latter four are known as the Information Transfer Phases. Arbitration and Reselection will not be discussed as they will not be required in any implementation mentioned within this work. The schematic in figure 3.20 indicates the relationships between phases and show the possible phase transitions. A series of phases which start and end in the BUS FREE phase is defined as a transaction. The actual phases involved in a transaction will depend on the command being executed and the success of that execution, and is controlled by the target device.

On power-up, or following a reset, the system comes up in the BUS FREE phase. An initiator may cause the bus to enter the SELECTION phase by placing the target ID of the required peripheral on the data bus and asserting the SEL line. Success is indicated by the target asserting the BSY line, and to complete the sequence the initiator then de-asserts SEL. If BSY is not asserted within a defined time then the bus will return to the BUS FREE phase, else it will enter one of the Information Transfer Phases.

After selection the target takes control of all bus timing and phase transitions and retains control until the transaction is complete. The three SCSI lines, MSG, C/D, and I/O combined, denote the phase. This is further illustrated in Figure 3.21.

Signal			Phase Name	Description
MSG	C/D	I/O		
0	0	0	Data Out	Initiator sends data to Target
0	0	1	Data In	Target Sends Data to Initiator
0	1	0	Command	Initiator sends command to Target
0	1	1	Status	Target sends status to Initiator
1	0	X		(Reserved)
1	1	0	Message Out	Initiator sends message to Target
1	1	1	Message In	Target sends message to Initiator

Figure 3.21 : SCSI Control Signals Indicating Information Transfer Phase

In the COMMAND phase, the target requests a command byte from the initiator. In the DATA phase, data is transferred. In the DATA OUT phase the initiator sends data to the target and in the DATA IN phase the target sends data to the initiator. In the STATUS phase the target transfers a byte to the initiator to indicate the success or failure of a command and finally, the MESSAGE phase is available to transfer a message from target to initiator, or vice versa.

Bytes are transferred between initiator and target using the REQ and ACK control signals to implement a full handshake protocol. For example, to transfer a byte from an initiator to a target the following occurs : the target indicates it is ready for a byte by asserting REQ ; the initiator indicates a byte is available on the bus by asserting ACK ;

the target reads the data and then de-asserts REQ ; and the initiator then de-asserts ACK when it realises the change on REQ. For a transfer from target to initiator the actions of these lines is similar : target asserts REQ to indicate byte ready ; Initiator reads and asserts ACK ; target notices and de-asserts REQ ; and initiator completes cycle by de-asserting ACK. Alternatively data may be transferred synchronously in the data phase but as the SCSI controller used in the hardware design does not support this, this option is not possible for this implementation.

Issuing SCSI Commands

SCSI specifies only classes and formats for commands. Each SCSI command is sent to a device as a Command Descriptor Block (CDB). The first byte of a CDB defines the type of command, the number of bytes required to define the command, and the command code itself. The following bytes contain extra information required in the execution of the command. For example in the Read command the logical unit number, the source address, and the number of sectors required are all present in bytes 1 to 5 of the CDB. Figure 3.22 contains the layout of such a read CDB.

Byte		
0	Operation Code	
1	Logical Unit Number	Logical Block Address (MSB)
2	Logical Block Address	
3	Logical Block Address (LSB)	
4	Transfer Length	
5	Control	

Figure 3.22 : Layout of a Six Byte Command Descriptor Block - Read Command

In a late revision of the SCSI standard a series of commands for each class of device likely to be using the SCSI bus is defined. This was a result of many of the vendors meeting to develop a Common Access Method for SCSI devices. The table in figure 3.23 indicates the Group 0 standard command set for Direct-Access Devices.

OP Code 650	Type	Command Name	Rodime
00H	O	Test Unit Ready	yes
01H	O	Rezero Unit	yes
02H	V		
03H	M	Request Sense	yes
04H	M	Format Unit	yes
05H	V		
06H	V		
07H	O	Reassign Blocks	yes
08H	M	Read	yes
09H	V		
0AH	M	Write	yes
0BH	O	Seek	yes
0CH	V		
0DH	V		
0EH	V		
0FH	V		
10H	V		
11H	V		
12H	E	Enquiry	yes
13H	V		
14H	V		
15H	O	Mode Select	yes
16H	O	Reserve	yes
17H	O	Release	yes
18H	O	Copy	NO
19H	V		
1AH	O	Mode Sense	yes
1BH	O	Start/Stop Unit	yes
1CH	O	Receive Diagnostic Results	NO
1DH	O	Send Diagnostic	yes
1EH	O	Prevent/Allow Medium Removal	NO
1FH	R		

Types
M = Mandatory Command
E = Required for Self Configuring Drivers
O = Optional Command
R = Reserved Command for future standardisation
V = Available for Vendor Specific Commands

Rodime 650 20MByte SCSI Disk drive used in prototype

Figure 3.23 : Table of Standard Commands for Direct-Access Devices

Each command listed in figure 3.23 has an associated, defined CDB which is to be adhered to during transfer. Clearly vendors may use some of the spare commands of type V for their own defined commands, but the commands defined above are sufficient for normal operation of a direct-access drive. Commands added by the vendor are often to be used only at factory level to test and configure the device before sale.

3.1.1 IMPLEMENTATION

[1] Direct Control IBM BIOS

The first requirement, after hardware testing, was to provide a working INT 13H BIOS replacement which would allow access to a SCSI hard disk drive through the hardware with no Supervisor intervention. This, together with the code in figure 3.12 blown into the Z80 ROM, would provide a firmware implementation of a 'standard' hard disk interface card.

The BIOS consisted of an initialisation routine, executed at boot time, and replacement code for the INT 13H and INT 19H software interrupts.

The 80x86 assembler code in figure 3.24 contains a simplified initialisation routine. Pointers and labels are assumed to be defined previously. The routine first checks that no other hard disks have been installed in the system. This not only simplifies the implementation for the prototype, but is also the preferred setup, since having an unprotected disk on-line introduces another possible source of infection for the active partition. If no other disk is present then the software vectors for INT 13H and INT19H are altered to point to routines within the new BIOS ROM. These vectors correspond to the Disk Services vector and Bootstrap vector, respectively.

```
initialise:
      cmp   hf_num,0          ; check no other hard drives
      je    install_drive     ; continue if so
      sti                     ; else fail
      ret

install_drive:
      mov   word ptr org_vector,offset disk_io  ; new INT13H
      mov   word ptr org_vector+2,cs
      mov   word ptr boot_vec,offset boot_strap ; new INT19H
      mov   word ptr boot_vec+2,cs

      mov   hf_num,1          ; increment number of disks
                              ; in battery RAM

      mov   word ptr hf_tbl_vec, offset RO650A  ; set drive

      mov ax,0                ; reset drive
      mov dx,80h
      mov cx,10
      int 13h

      mov ax,1000h            ; test drive ready
      int   13H

      jnc   drive_up          ; jump if drive ready

      dec   hf_num            ; else remove disk

drive_up:                     ; finished
      ret
```

Figure 3.24 : Simplified Hard Disk BIOS Interrupt Routine

The variable held in the battery RAM of the machine, which represents the number of drives in the system, is then incremented so that the operating system will become aware of its existence. The hard disk is then fixed as a Rodime RO650A drive as this was the drive used during development. The drive is then reset, and a 'test unit ready' command invoked. If this command returns a positive response then initialisation is complete, otherwise the number of drives is set back to zero.

The INT 19H replacement code features, in simplified form, in figure 3.25. The sequence it follows is fairly simple. Firstly it attempts to bootstrap from the floppy : if the reset fails then no disk is present, otherwise it reads the boot sector from the disk into RAM and executes it. Even if a non-system disk has been inserted, executable

code will exist in this sector which prints a message to the screen informing the user that it is not a system disk, if the disk is formatted.

```
BOOT_STRAP:
      sub   dx,dx                    ;reset floppy
      sub   ax,ax
      int   13H

      jc    SKIP_FRD                 ;if fail try hard disk

      mov   ax,201H                  ;read boot sector into RAM
      sub   dx,dx
      mov   es,dx
      mov   bx,offset BOOT_LOCN
      mov   cx,1
      int   13H

      jnc   BOOT_LOCN                ;if success try booting

TRY_HD_BOOT:                         ;else try hard disk

      cmp   hf_num,0                 ;if no hard disk then fail
      je    ENTER_BASIC

      sub   ax,ax                    ;reset hard disk
      mov   dx,80H
      int   13H

      jc    ENTER_BASIC              ;if error then fail

      mov   ax,201H                  ;read boot sector into RAM
      sub   dx,dx
      mov   es,dx
      mov   bx,offset BOOT_LOCN
      mov   cx,1
      int   13H

      jc    ENTER_BASIC              ;if error then fail

      mov   ax,word ptr BOOT_LOCN+510      ;check boot sector
      cmp   ax,0AA55H                      ;signature

      jnz   ENTER_BASIC              ;if error then fail

      jmp   BOOT_LOCN                ;else try boot

ENTER_BASIC:
      int   18H
```

Figure 3.25 : Simplified Hard Disk BIOS Bootstrap Loader

If no floppy is present then bootstrapping of the hard disk is attempted. First there is a check to see if a hard disk exists, then a reset followed by a Read of the boot sector. Finally the boot sector signature is checked to see if it is valid. If any of these tests prove negative then the bootstrap fails and the code attempts to call the ROM based BASIC interpreter if it exists. If all tests pass then execution of the downloaded boot sector occurs.

Both the initialisation and the bootstrap loader routines use the replacement INT 13H code. This software interrupt must be capable of providing all low level disk services required by the system. In total there are eighteen services it must provide, although for a prototype system a small subset of these suffices.

Five of the INT 13H functions must be present for a system to function correctly. These are : RESET DISK SYSTEM ; TEST DRIVE READY ; and READ, WRITE, and VERIFY SECTORS. Each of these are explained separately below :

RESET DISK SYSTEM

This function is used by the system to reset the entire disk system. For a SCSI implementation this requires the SCSI bus to be reset and each drive in the system, in this case only one, to be tested for readiness.

```
disk_reset  proc  near

      ; reset SCSI bus
      mov   dx,icmdp          ; dx=internal command register SCSI
      mov   al,rst            ; al=reset command
      out   dx,al             ; send command to SCSI

      ;DELAY loop here in full version
      mov   al,null           ; finish reset
      out   dx,al

      ;DELAY loop here in full version
      call  tst_rdy           ; test drive ready
disk_reset  endp
```

Figure 3.26 : Reset Disk System Function for Replacement INT 13H BIOS

The code in figure 3.26 shows a slightly simplified implementation of this function. Delay loops have been removed for clarity. The tst_rdy call will return the status of the function which in turn will be propagated out of the RESET function.

TEST DRIVE READY

This function is used by the system to check that a drive is available for use. This is implemented using the TEST UNIT READY command in SCSI.

```
tst_rdy proc near
      mov   al,null      ; SCSI byte-5 - no parameters required
      push  ax
      mov   al,null      ; SCSI byte-4
      push  ax
      mov   al,null      ; SCSI byte-3
      push  ax
      mov   al,null      ; SCSI byte-2
      push  ax
      mov   al,null      ; SCSI byte-1
      push  ax
      mov   al,00H       ; SCSI byte-0 - Test Unit Ready Command
      push  ax
      mov   cmd,al
      jmp   command
tst_rdy           endp
```

Figure 3.27 : Test Drive Ready Function for Replacement INT 13H BIOS

The code in figure 3.27 loads a SCSI format command block onto the system stack, in reverse order, representing the function required. In this case bytes one to five are filled with nulls since no parameters are required for this command. Control is then passed to code which executes the SCSI command. This will be explained later.

READ, WRITE, and VERIFY SECTORS

These commands are grouped in this section because, although their functions are entirely dissimilar, the information required to issue the command is the same. Thus the SCSI command block for each is very similar.

```
disk_write  proc  near
      mov   al,null      ; SCSI byte-5
      push  ax
      mov   al,cmd+4     ; SCSI byte-4 - Number of Sectors
      push  ax
      mov   al,cmd+3     ; SCSI byte-3 - Low order address
      push  ax
      mov   al,cmd+2     ; SCSI byte-2 - Middle order address
      push  ax
      mov   al,cmd+1     ; SCSI byte-1 - High order address
      push  ax
      mov   al,0AH       ; SCSI byte-0 - Write Sectors command
      push  ax
      mov   cmd,al
      jmp   command
disk_write  endp
```

Figure 3.28 : Write Sectors Function for Replacement INT 13H BIOS

Figure 3.28 contains the code required to load a 'Write Sectors' SCSI command block onto the system stack and to execute that SCSI command. Aside from the command byte, the command block includes the three byte logical address to write to and the number of bytes to be written. The Read command is almost identical, with the command byte being the only difference. The verify command requires a ten byte command block but the information it contains is identical to that of Read and Write : the address and number of sectors.

A SCSI command is executed via the assembler routine shown in figure 3.29. No attempt is made to arbitrate for use of the SCSI bus as only one host will exist in this implementation. SCSI can however be configured for multiple hosts sharing the bus.

This code firstly attempts to select the target. If this is successful then the target takes control of the SCSI bus and causes it to enter various phases, dependent on the type of operation, until the command is completed, with or without error.

```
;Disk Command Interface
command proc near
      call  select            ; select target on SCSI bus
      jc    command_1         ; jump on error
      jmp   phaser            ; perform requested phases
command_1:
      jmp   end_seq           ; return to caller with carry set
command     endp
```

Figure 3.29 : Main Routine for Execution of SCSI Commands

The code in figure 3.30 indicates the required sequence to select a drive using the SCSI controller mentioned in the hardware section. Firstly the SCSI controller is configured as an initiator, as opposed to a target, and the interrupts disabled since they are not required. Then Initiator Command Register is reset, as are all the error flags within the controller by reading the Parity/Interrupt register.

```
select:
      mov   dx,modep
      mov   al,00000000B      ; set as initiator / no interrupts
      out   dx,al

      mov   dx,icmdp          ; reset initiator command register
      out   dx,al

      mov   dx,prstp          ; reset parity/interrupts
      in    al,dx

      mov   al,host_targ      ; send host & target data byte
      mov   dx,datap
      out   dx,al

      mov   al,datoutt        ; set to data out target
      mov   dx,tcmdp
      out   dx,al

      mov   dx,icmdp          ; assert SEL
      mov   al,sel
      out   dx,al

      mov   cx,0FFFFH         ; set a big count in case of trouble
```

```
        mov     dx,statlp
select_1:
        in      al,dx
        and     al,01000000B
        jnz     select_good         ; wait for BSY true
        loop    select_1

        mov     dx,icmdp            ; ERROR so reset select false
        xor     ax,ax
        out     dx,al
        stc                         ; indicate error - set carry flag
        ret

select_good:
        mov     dx,icmdp            ; reset select false
        xor     ax,ax               ; note - carry flag reset by this out
        dx,al
        ret
```

Figure 3.30 : SCSI Drive Selection Sequence

Once this has been done, the host and target addresses are placed within a byte and driven onto the data bus. Both addresses correspond to a single bit set within a byte, as only eight SCSI devices may share a bus. The two addresses are combined therefore by a simple logical OR. The next step is to assert the SCSI select line (SEL) to begin the handshake procedure. If the SCSI BSY line becomes active after some time then the chosen target has responded and selection has occurred, or else an error has occurred. To complete the handshake SEL is then reset.

Once the target drive has been successfully selected, its SCSI controller enters a series of distinct phases which, when completed, have executed the required SCSI command. The BIOS firmware must follow the phases that the target controller enters and supply any requested data or read any supplied data. The code within figure 3.31 is used to call one of six subroutines, one for each possible SCSI controller information transfer phase. The choice of subroutine to be called depends on the current phase of the SCSI controller, and this is identified by reading one of the internal status registers.

```
phaser      proc  near

phaser_0:
      mov   dx,stat1p          ; read status
      in    al,dx
      test  al,01000000B       ; test BSY
      jz    exit               ; exit, if lost BSY

      test  al,00100000B       ; test for REQ - pending data trans
      jz    phaser_0           ; try again if not got one

      and   al,phasem          ; mask for phase bits only
      cmp   al,datoutp         ; data out phase
      jz    datout
      cmp   al,datinp          ; data in phase
      jz    dataind
      cmp   al,cmdp            ; command phase
      jz    cmdbyte
      cmp   al,statusp         ; status phase
      jz    status
      cmp   al,msgoutp         ; message out phase
      jz    mesgout
      cmp   al,msginp          ; message in phase
      jz    mesgin
      jmp   phaser_0           ; should never get here!
```

Figure 3.31 : Main Routine Run During SCSI Controller Phase Execution

This routine first checks to see if the phase sequence is complete. If it is not, and data transfer is requested, then this routine identifies the current phase and jumps to the corresponding subroutine which deals with the requested transfer, and returns upon completion. Different SCSI commands step through different sequences of SCSI phases and this routine allows any combination of phases to occur, dependent only on the target SCSI controller's requests. Information that may be required by a subroutine in order for it to complete its task is made available to it before the subroutine is executed. For example, the command bytes required to be sent during the SCSI command phase have been pushed onto the top of the stack.

```
cmdbyte:
        mov     dx,icmdp            ; command phase
        in      al,dx               ; enable output drivers
        or      al,00000001B
        out     dx,al

        mov     dx,tcmdp            ; set up command phase
        mov     al,cmdt
        out     dx,al

        pop     ax                  ; get command byte from stack
        mov     dx,datap
        out     dx,al               ; send command byte
        call    ackdrv              ; acknowledge byte ready

        mov     dx,icmdp            ; disable output drivers
        mov     al,null
        out     dx,al
        jmp     phaser_0            ; get next phase
```

Figure 3.32 : SCSI Command Phase Subroutine

The command phase usually follows directly after the selection phase and the code for this is represented in figure 3.32. This phase allows the target drive to identify the command required by the initiator. To start with, this routine prepares for transfer by enabling the output drivers of the SCSI controller and setting the target command register to match the current phase. The requested command byte is then popped from the stack and placed on the data bus. The subroutine 'ackdrv' then implements full handshake using REQ and ACK, to transfer the byte. The command phase routine will be called repeatedly until all command bytes have been transferred.

The two data exchange phases, the DATA IN phase and DATA OUT phase, provide the required mechanism for mass data transfer. Data is sequentially transferred, byte by byte until the desired block has been relocated. The implementations shown in figure 3.33 and figure 3.34 use the repeat instruction abilities of the Intel processor to fast-transfer one sector at a time since all data moves in multiples of sectors within this phase. After each sector has been moved the tests are repeated to check for transfer continuation.

```
datain:
      mov   ax,x_seg          ; set input buffer
      mov   es,ax

      mov   al,00000000b      ; data in phase using pseudo DMA
      mov   dx,icmdp          ; de-assert data bus
      out   dx,al

      mov   al,datint         ; set up data in phase
      mov   dx,tcmdp
      out   dx,al

      mov   al,00000010b      ; DMA mode
      mov   dx,modep
      out   dx,al

      mov   dx,dmairp         ; start DMA receive
      out   dx,al

datain_0:
      mov   dx,stat2p         ; check status bits
      in    al,dx

      and   al,01001000b      ; test phase match and DMA REQ
      cmp   al,01001000b
      jne   datain_1          ; if not ready jump to further tests

      mov   dx,dmap           ; else read data bytes
      mov   cx,512
      rep   insb
      jmp     datain_0

datain_1:

      test    al,00001000B    ;test phase match
      jz      datain_0        ;if match retry else exit

      mov     dx,modep        ;cancel DMA mode
      mov     al,null
      out     dx,al
      jmp     phaser          ;get next phase
```

Figure 3.33 : DATA IN Phase for SCSI Transfer

The ability to use the repeat instruction comes from the hardware wait state generator, implemented in the previous section. If the SCSI controller has not requested a byte but the repeat instruction has attempted to send one, or the SCSI controller does not have a byte available but the repeat instruction attempts to read one, then wait states are generated until the SCSI controller can oblige.

```
datout:
        mov     ax,x_seg            ; select buffer to output
        mov     ds,ax

        mov     al,datoutt          ; set to data out phase + pseudo DMA
        mov     dx,tcmdp
        out     dx,al

        mov     al,00000001b        ; assert data bus
        mov     dx,icmdp
        out     dx,al

        mov     al,00000010b        ; DMA mode
        mov     dx,modep
        out     dx,al

        mov     dx,dmaisp           ; start DMA send
        out     dx,al

        mov     dx,icmdp            ; enable output drivers
        in      al,dx
        or      al,00000001b
        out     dx,al

datout_1:
        mov     dx,stat2p           ; check status bits
        in      al,dx
        and     al,01001000b        ; test phase and DMA REQ
        cmp     al,01001000b
        jne     datout_2            ; if not ready - further tests

        mov     dx,dmap             ; else output data bytes
        mov     cx,512
        rep     outsb
        jmp     datout_1            ; look for next block


datout_2:
        test    al,00001000b        ; test phase match
        jnz     datout_1            ; if a match - try again else exit

        mov     al,null             ; cancel DMA mode
        mov     dx,modep
        out     dx,al

        mov     dx,icmdp            ; disable output drivers
        out     dx,al

        jmp     phaser              ; get new phase
```

Figure 3.34 : DATA OUT Phase for SCSI Transfer

The code is similar for both transfer phases : the SCSI controller is configured for the transfer, DMA is invoked, and data is transferred. The data transfer repeats until all sectors have been relocated.

```
status:
      mov   dx,datap          ; status phase
      in    al,dx             ; read status byte
      mov   disk_status,al    ; store status
      call  ackdrv            ; acknowledge target
      jmp   phaser_0          ; get next phase
```

Figure 3.35 : STATUS Phase for SCSI Transfer

The operations required for implementation of the STATUS phase are simple, see figure 3.35. A byte is read from the internal data register of the SCSI controller, using the standard handshake procedure, and stored in a variable.

```
mesgin:
      mov   dx,datap          ; message in phase
      in    al,dx             ; read message in byte
      call  ackdrv            ; acknowledge target
      jmp   phaser_0          ; get next phase
```

Figure 3.36 : MESSAGE IN for SCSI Transfer

The implementation of the MESSAGE IN phase is functionally identical to the STATUS phase except that for the purposes of this implementation, the message byte is of no use and is discarded. The only message which provides any useful information for this implementation is the COMMAND COMPLETE message, but alternatively this may be identified by the release of the BSY line by the target.

Once the required phases were implemented, a fully operational SCSI disk interface had been realised. The BIOS was recognised and executed on power-up, and re-routed the bootstrap and disk services interrupts making the drive visible to the system. After power-up the drive could then be high-level formatted and used.

[2] Supervisor Control IBM BIOS & Z80 Control Program

The next stage in development was to route all required disk requests through the Z80. All operations involving the SCSI controller, except the physical transfer of data blocks, were to be performed by the Z80 and indeed made impossible for the IBM to perform.

A command that requires no DATA phase can be executed solely by the Supervisor. This requires the IBM BIOS to transfer the command information to the Z80, and for the Z80 to return status information to the IBM on completion. The IBM BIOS code for a TEST UNIT READY command is illustrated in figure 3.37

```
tst_rdy proc near

      waitforit                ;*Wait for Supervisor ready
      MOV      DX,Sdatout      ;*Output data byte
      MOV      AL,Ctstrdy      ;*Test Unit Ready Command
      OUT      DX,AL
      waitforit                ;*Once read Sstatin cleared

Tretry:
      MOV      DX,Sstatin      ;*Read status of supervisor
      IN       AL,DX

      AND      AL,11111100B    ;*Mask off lower 2 bits
      CMP      AL,00000000B    ;*Test if command finished
      JE       Tretry

      CMP      AL,00000100B    ;*Test if finished and okay
      JE       Tonward

      MOV      AH,error        ;*Error returned
      STC                      ;*set carry flag to indicate error
      ret

Tonward:
      XOR      AX,AX           ;*Clear carry flag to indicate
      ret                      ;*no error

tst_rdy endp
```

Figure 3.37 : Test Unit Ready Function for Supervisor Controlled IBM BIOS

Note that the whole SCSI Command Descriptor Block is not transferred, only the bytes with useful information. For the Test Unit Ready command, the command specifier is enough. The routine then monitors the status byte from the Z80 for either an 'error' or 'status good' indication. If an error is detected then the IBM carry flag is set.

The Z80 ROM continuously polls the register, to which the IBM writes bytes, awaiting instructions. When a byte is written, it is read as a command specifier and used to call a subroutine which will execute the requested SCSI command. This may involve reading more bytes from the IBM. When execution is complete the program returns to polling for another command specifier. Therefore when the IBM sends its test unit ready command specifier to the Z80 the subroutine in figure 3.38 is executed.

```
ltstrdy ld      ix,ramit + tstrdy       ;*Load Command-Set
        call    phaser                  ;*Execute
        ld      a,c                     ;*load in status
        or      noerr                   ;*set finished bit
        out     (StatusW),a             ;*output status returned
        jp      askIBM                  ;*go get another
```

Figure 3.38 : Test Unit Ready Command Execution by the Z80

This routine uses a predefined command descriptor block for the test unit ready command. Elements of this CDB could be altered at this point if required, but the test unit ready command requires no extra instruction bytes. The subroutine phaser is then called. This routine behaves identically to the subroutine phaser for the direct access IBM BIOS in the last section, only this time it is written in Z80 assembler. Thus the command is executed and status returned in Z80 register 'C'. This status is then output to the Status port where the IBM will read it and know the command is finished and whether it was a success.

Commands that require a DATA phase are more complex. Control of the SCSI controller must be passed to the IBM for the actual data transfer, in order not to slow the system down. Figure 3.39 passes a READ SECTORS command to the Supervisor with all associated address and sector number information.

```
disk_read proc near
      waitforit                ;*Wait for Supervisor ready
      mov   dx,Sdatout         ;*Output data byte
      mov   al,Cread           ;*Read From Disk Command
      out   dx,al

      waitforit                ;*Wait for Supervisor ready
      mov   dx,Sdatout         ;*Output data byte
      mov   al,cmd+1
      out   dx,al

      waitforit                ;*Wait for Supervisor ready
      mov   dx,Sdatout         ;*Output data byte
      mov   al,cmd+2
      out   dx,al

      waitforit                ;*Wait for Supervisor ready
      mov   dx,Sdatout         ;*Output data byte
      mov   al,cmd+3
      out   dx,al

      waitforit                ;*Wait for Supervisor ready
      mov   dx,Sdatout         ;*Output data byte
      mov   al,cmd+4
      out   dx,al
```

Figure 3.39 : READ SECTORS Command for Supervised IBM BIOS : PART 1

The supervisor reads the first byte sent and as a result executes the procedure in figure 3.40. This procedure reads the required bytes sent by the IBM and calls the phaser routine in the same way as for the TEST UNIT READY command previously explained.

```
lread
      ld    ix,ramit + dread          ;*load in CDB
      call  await                     ;*wait for IBM byte
      ld    (ix+1),a                  ;*add to CDB
      call  await                     ;*wait for IBM byte
      ld    (ix+2),a                  ;*add to CDB
      call  await                     ;*wait for IBM byte
      ld    (ix+3),a                  ;*add to CDB
      call  await                     ;*wait for IBM byte
      ld    (ix+4),a                  ;*add to CDB
      call  phaser                    ;*execute
```

Figure 3.40 : READ SECTORS Command Z80 Code : PART 1

The main difference occurs during the DATA IN phase, see figure 3.41. The target requests this phase and the Supervisor must allow the IBM access to the bus to transfer the data. The Supervisor configures the SCSI controller for the operation, but it does not initiate the transfer itself. Instead it gives the IBM control of the SCSI controller and sends a status byte to the IBM to indicate that the controller is ready for the transfer.

```
datain
      ld    a,00000000b             ;*de-assert data bus
      out   (icmdp),a

      ld    a,datint                ;*set to data in phase
      out   (tcmdp),a

      ld    a,00000010b             ;*set to dma mode
      out   (modep),a

      ld    a,IBMPath               ;*give IBM control
      out   (PathChW),a

      ld    a,read_data             ;*inform ibm ready
      out   (StatusW),a
```

Figure 3.41 : READ SECTORS Command Z80 Code : PART 2

The IBM has been waiting since sending the command bytes in figure 3.39 for this status to be sent by the supervisor. The code in figure 3.42 contains the loop used to wait for the Z80 to indicate transfer readiness. When ready the IBM initiates the transfer and reads the sector, or sectors, in exactly the same manner as explained for the direct access BIOS.

```
Ztest:
      mov   dx,Sstatin          ;*Read Status of Supervisor
      in    al,dx

      and   al,11111100B        ;*Test status info
      cmp   al,read_ready       ;*Ready for info to be read ?
      jne   Ztest               ;*if not then try again

      mov   ax,x_seg            ;*Set up buffer segment
      mov   es,ax

      mov   dx,dmairp           ;*Start DMA Receive
      out   dx,al

Zatain_0:
      mov   dx,stat2p           ; check status bits
      in    al,dx
      and   al,01001000b        ; test phase match and DMA REQ
      cmp   al,01001000b
      jne   Zatain_1            ; if no then jump

      mov   cx,512              ; else read sector
      cli                       ; clear interrupt flag
      rep   insb                ; string input from port
      sti                       ; set interrupt flag
      jmp   Zatain_0

Zatain_1:
      test  al,00001000B        ; test phase match
      jnz   Zatain_0

      mov   dx,Sdatout          ; inform supervisor finished
      out   dx,al               ; data value not important
```

Figure 3.42 : READ SECTORS Command for Supervised IBM BIOS : PART 2

Once transfer is complete the IBM indicates completion by writing a dummy byte to the write out port to the Z80. The Z80 has been waiting for this indication, see figure 3.43. When it detects that the byte has been written, by reading the status flag, it takes control of SCSI back and completes any further phases requested by the target. It should be noted that at no point, when the IBM has control of the data bus, is it possible to reconfigure the SCSI controller to read or write to an alternative location. Having control of the data bus purely means that the IBM has the ability to transfer data to the SCSI controller, the destination of these data bytes is kept restricted, in hardware, to a minimum set which does not include the configuration registers of the controller.

```
Rwait
      in    a,(IBMStaR)              ;*See if IBM finished
      bit   0,a                      ;*test status bit
      jp    z,Rwait                  ;*retry if no data yet

      ld    a,Z80Path                ;*take control of pathway
      out   (PathChW),a

      ld    a,null                   ;*exit DMA mode
      out   (modep),a

      jp    phaser
```

Figure 3.43 : READ SECTORS Command Z80 Code : PART 3

Upon command completion a status byte is sent to the IBM as normal and the dummy byte, the IBM sent to indicate data transfer completion, is read : this indicates to the IBM that the supervisor has finished. The code for this is shown in figure 3.44.

```
      ld    a,c                      ;*load in status
      or    noerr                    ;*set finished bit
      out   (StatusW),a              ;*output status
      in    a,(IBMDatR)              ;*Dummy read-done!
      jp    askIBM                   ;*go get another
```

Figure 3.44 : READ SECTORS Command Z80 Code : PART 4

Finally, the IBM gets an indication that the Z80 has completed the command, so it reads the status and exits according to the indicated result, see figure 3.45.

```
      waitforit                ;*Wait for Supervisor finish

      in    al,dx              ;*Read status
      and   al,11111100B       ;*Mask of lower 2 bits
      cmp   al,00000100B       ;*Test if finished and okay
      je    Zonward

      mov   ah,time_out        ;*Error returned
      stc
      ret

Zonward:
      xor   ax,ax              ;*Clear carry flag
      ret

disk_read   endp
```

Figure 3.45 : READ SECTORS Command for Supervised IBM BIOS : PART 3

This form of command requires careful passing of control between the two processors. Careful use of status interchange is used to ensure that they remain synchronised and that both are aware of the condition of the transaction at each stage.

So, with the exception of the DATA phase, all execution of SCSI commands was transferred to Z80 control. This involved the routine 'phaser' and all associated subroutines. The data phases, both DATA IN and DATA OUT, required the careful mutual execution outlined above. Once this was implemented the Z80 could begin to take on its full role as Supervisor, as the ability to allow or deny any given SCSI command became possible. The IBM could no longer execute a SCSI command without the aid of the Z80 due to the hardware access restrictions to the SCSI controller's internal configuration registers.

The implementation, at this stage, was functionally identical to that of the previous stage. Although commands were no longer directly issued to the SCSI controller, they were however executed with the result that no change would be apparent to the user.

[3] Restricted Access : A full implementation of the Supervisor

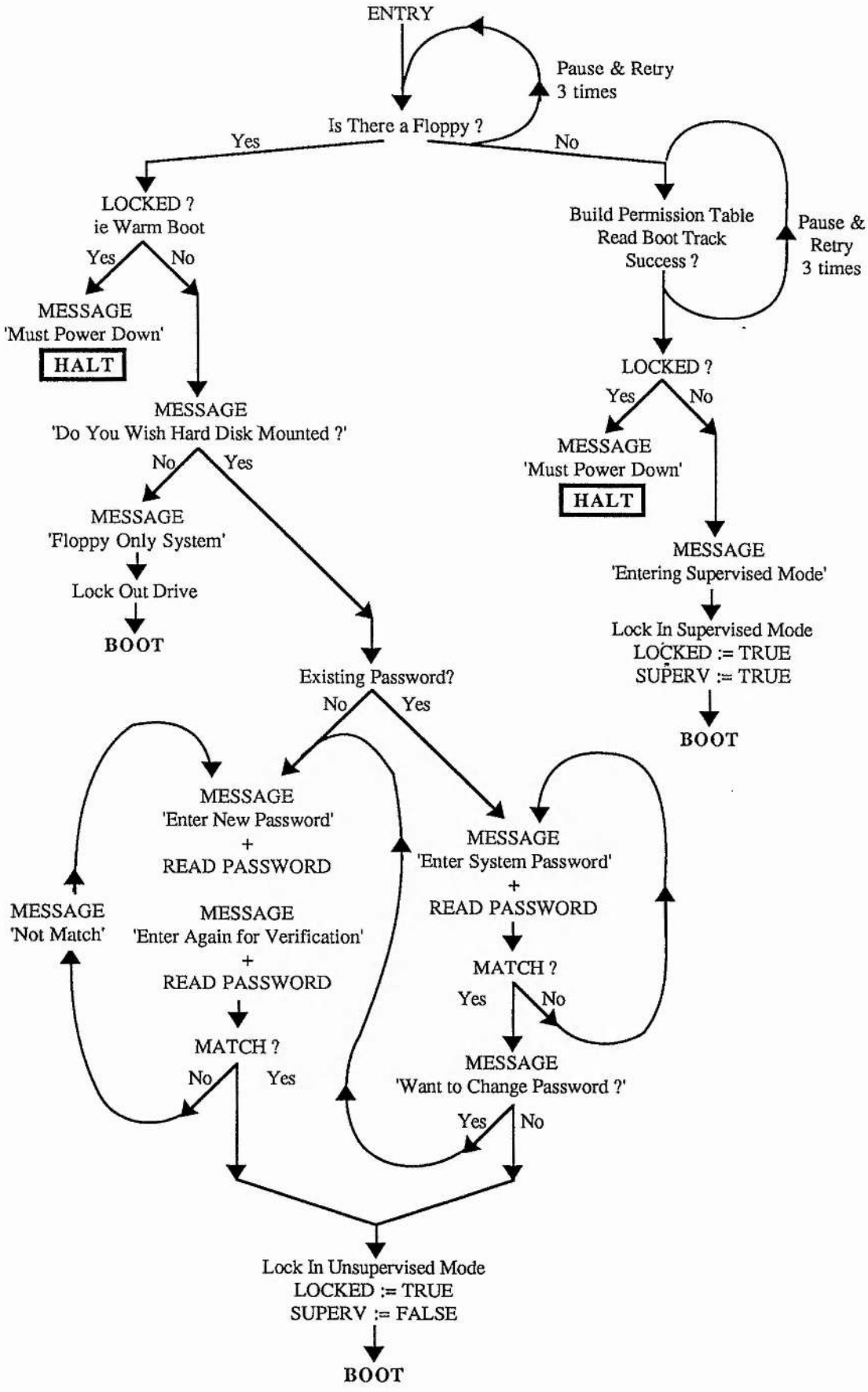
[1] Implementing the Unsupervised Mode

Firstly provision must be made for entry into a mode where Read, Write and format requests are executed without restriction. This is further referred to as unsupervised mode. This facility must be provided so that partitions may be created or modified, and so that the operating system may be updated and new virus-free software placed within the boot partition. Access to this mode must be limited to experienced system managers and in order to facilitate this a password schema was implemented. Without the required password a user is restricted to supervised mode only.

Firstly, the supervisor reserves the last track on the hard disk for storage of this password, and any information it may further wish to store. The size of the disk is reduced accordingly, so the space cannot be allocated to the user. Access restrictions on the area above the last partition ensure that the password cannot be read directly from the disk in supervised mode. The password is passed byte by byte to the supervisor before DOS is even loaded, ensuring no software can be running in the background monitoring the password transfer. Thus the password is entirely secure from both user and virus. No encryption is required either, due to the hardware access restriction.

The flow diagram presented in figure 3.46 illustrates the supervisor's boot sequence. Firstly, the existence of a floppy is checked for. If no floppy is present, then supervised mode is the only possible option, so the permission list is built ready for interrogation, as explained in the next section. A supervisor maintained variable, 'locked', is then checked. The variable is initialised from a cold start as false, which occurs when the machine first powers up. During a later point in the boot sequence, 'locked' is made true, thus, if a warm start is subsequently initiated, 'locked' is found to be true. A warm start is where the processor is reset, usually from a CTRL-ALT-DEL key sequence, when the machine is already powered up, and memory contents are usually unaffected by this type of boot sequence. If 'locked' is found true during the warm start, the machine hangs with the message, 'You must power down'. This ensures the contents of memory are cleared, destroying any memory resident viruses. If locked is false then supervised mode is entered, and the user is informed of this. 'Locked' is then made true and the operating system is loaded from the hard disk.

Over The Page : Figure 3.46 : Supervisor Boot Sequence Flow Diagram


ENTRY
Pause & Retry
3 times
Is There a Floppy ?
Yes
No
LOCKED ?
ie Warm Boot
Yes
No
MESSAGE
'Must Power Down'
HALT
MESSAGE
'Do You Wish Hard Disk Mounted ?'
No
Yes
MESSAGE
'Floppy Only System'
Lock Out Drive
BOOT
Build Permission Table
Read Boot Track
Success ?
Pause &
Retry
3 times
LOCKED ?
Yes
No
MESSAGE
'Must Power Down'
HALT
MESSAGE
'Entering Supervised Mode'
Lock In Supervised Mode
LOCKED := TRUE
SUPERV := TRUE
BOOT
Existing Password?
No
Yes
MESSAGE
'Enter New Password'
+
READ PASSWORD
MESSAGE
'Not Match'
MESSAGE
'Enter Again for Verification'
+
READ PASSWORD
MATCH ?
No
Yes
MESSAGE
'Enter System Password'
+
READ PASSWORD
MATCH ?
Yes
No
MESSAGE
'Want to Change Password ?'
Yes
No
Lock In Unsupervised Mode
LOCKED := TRUE
SUPERV := FALSE
BOOT

If a floppy disk is present then two possible modes can be entered. Firstly, however, 'locked' is checked as for supervised mode entry. This ensures no memory resident virus can be active as before. If 'locked' is false, one can boot from the floppy with the hard disk forced off-line by the supervisor. This is in effect an unsupervised mode but with the hard disk safe. This mode can be accessed by any user for the testing of floppy based software with the knowledge that no harm can come to sensitive data on the hard disk.

The second possibility is to have the hard disk made on-line. This is true unsupervised mode and therefore requires the password mentioned earlier. A check is made to see if a password exists. When first installing this system clearly this is not the case. A signature is placed within the password sector once a valid password exists, and so this signature need only be checked : this is how DOS identifies if a boot sector exists. If no password exists then the user is immediately prompted for one, which is entered twice and compared in case of typing error. The password is then saved and unsupervised mode entered for the first time.

If a password already exists then the user is prompted to enter it, and it is then compared against the stored original. A password correctly entered is followed by the offer of password alteration, a facility which must be provided for any password schema. Unsupervised mode is then entered and the system 'locked' and the variable 'superv' set to false to indicate the mode.

As already outlined, the machine remains in the mode it is in after boot, until a cold start is initiated. For a supervised user to alter active partition, the machine must be cold booted into supervised mode once more, and the new partition selected.

[2] Building a Partition Map of the Hard Disk

MS-DOS version 3.3 and above supports two types of MS-DOS partition : primary and extended. A primary partition can be made bootable by marking it active and formatting it using the FORMAT command with the /S switch set. An extended partition cannot be made bootable but may contain within it a multiple of physically contiguous logical drives. This is illustrated in figure 3.47. The extended partition allows large hard disk drives to be split into a series of smaller, more manageable logical drives.

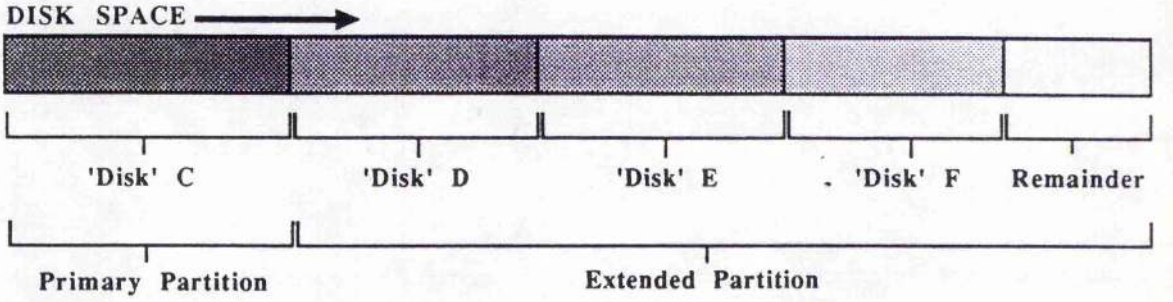


Figure 3.47 : Example Layout of a Hard Disk Under MS-DOS 3.3+

For the prototype implementation it was assumed that the disk would contain only MS-DOS partitions. This would mean that there would be at most two partitions existing in a system, one primary MS-DOS and one extended MS-DOS partition. Should partitions be introduced by another operating system, these could be mapped out, but this would have introduced unnecessary complication.

PARTITION	OFFSET	SIZE	DESCRIPTION
ONE	01BEH	1 byte	boot indicator (00H-non-boot, 80H-boot)
01BFH		1	beginning head (0->)
01C0H		1	beginning sector (1->)
01C1H		1	beginning cylinder (0->)
01C2H		1	system indicator (00H=unknown,01H/04H=DOS)
01C3H		1	ending head (0->)
01C4H		1	ending sector (1->)
01C5H		1	ending cylinder (0->)
01C6H		**4**	**Starting Sector rel to beginning of disk**
01CAH		4	number of sectors in partition
TWO	01CEH	1	boot indicator (00H-non-boot, 80H-boot)
01CFH		1	beginning head (0->)
01D0H		1	beginning sector (1->)
01D1H		1	beginning cylinder (0->)
01D2H		1	system indicator (00H=unknown,01H/04H=DOS)
01D3H		1	ending head (0->)
01D4H		1	ending sector (1->)
01D5H		1	ending cylinder (0->)
01D6H		**4**	**Starting Sector rel to beginning of disk**
01DAH		4	number of sectors in partition

Figure 3.48 : Partition Table for a Physical Disk With Two MS-DOS Partitions

Under the MS-DOS partitioning standard, the first physical sector on the fixed disk is the Disk Boot Sector (DBS) which contains not only a bootstrap program, but also the partition table for the drive. Figure 3.48 contains the partition table for a two partition drive, as mentioned above. The information located within the bytes marked in bold in figure 3.48 are sufficient to locate the Partition Boot Sector of the partition in question. This boot sector contains a loader routine and a BIOS Parameter Block (BPB) with information about the device. The contents of the BPB are illustrated in figure 3.49.

OFFSET	SIZE	DESCRIPTION	SYMBOL
0BH	2 bytes	bytes per sector	
0DH	1	sectors per allocation unit	
0EH	2	reserved sectors (0->)	R
10H	1	number of FATs	n
11H	2	number of root directory entries	N
13H	2	total sectors in logical volume	
15H	1	media descriptor byte (F8=fixed disk)	
16H	2	number of sectors per FAT	S

Figure 3.49 A BIOS Parameter Block

The layout of an MS-DOS file system within a partition is shown in figure 3.50. Using the information contained within the BPB it is possible to identify any of the boundaries that exist within this structure. For example, we can identify the address of the start of the user files area of the partition. This is important as we need to be able to identify system sectors from user sectors within a partition in order to implement the schema outlined in Chapter 2.

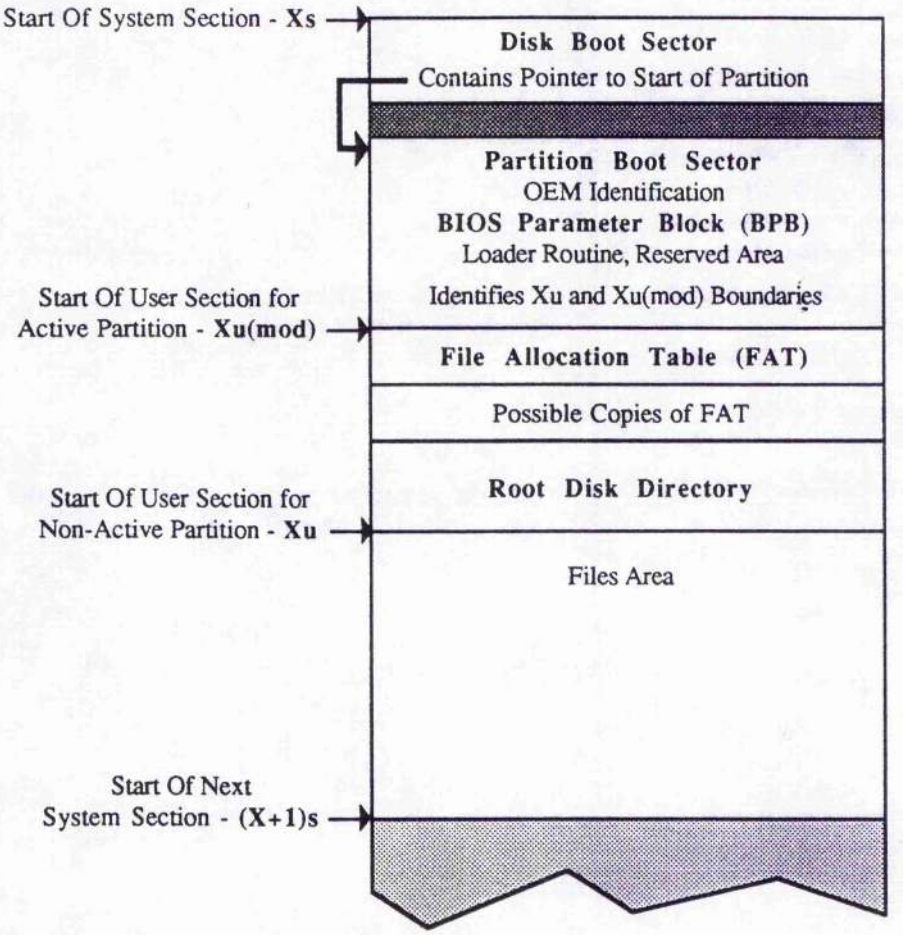


Figure 3.50 : Layout of an MS-DOS File System

The two pointers, SUx and SUx(mod), in figure 3.50 are the ones of which we require knowledge. SUx identifies where the system sectors finish and the user sectors begin, for the partition. A non active partition will use this boundary, as reading from system sectors is allowed and indeed required by the system, whereas reading from user sectors is forbidden. The reason for this is that a Read request implies a possible Execute, and executing an application from a non-active partition cannot be allowed. Writing will not be allowed for either type of sector. When a partition is activated, Read and Write privileges are given to the user sectors, the directory sectors, and the FAT tables within the system sectors. As the permissions are identical, this is implemented most easily by shifting the system/user boundary back to SUx(mod). The system sectors which remain before SUx(mod) retain their 'Read only' status since writing would allow the partition bounds to be altered and therefore would allow the possibility of writing outwith the bounds of the partition.

Figure 3.51 contains a flow diagram indicating how the required address boundaries are obtained from the disk. The algorithm starts at the first physical sector of the fixed disk where, as mentioned above, the partition table exists. Address 00000000H is also, understandably, the start address for the system section of the first partition, SS_1.

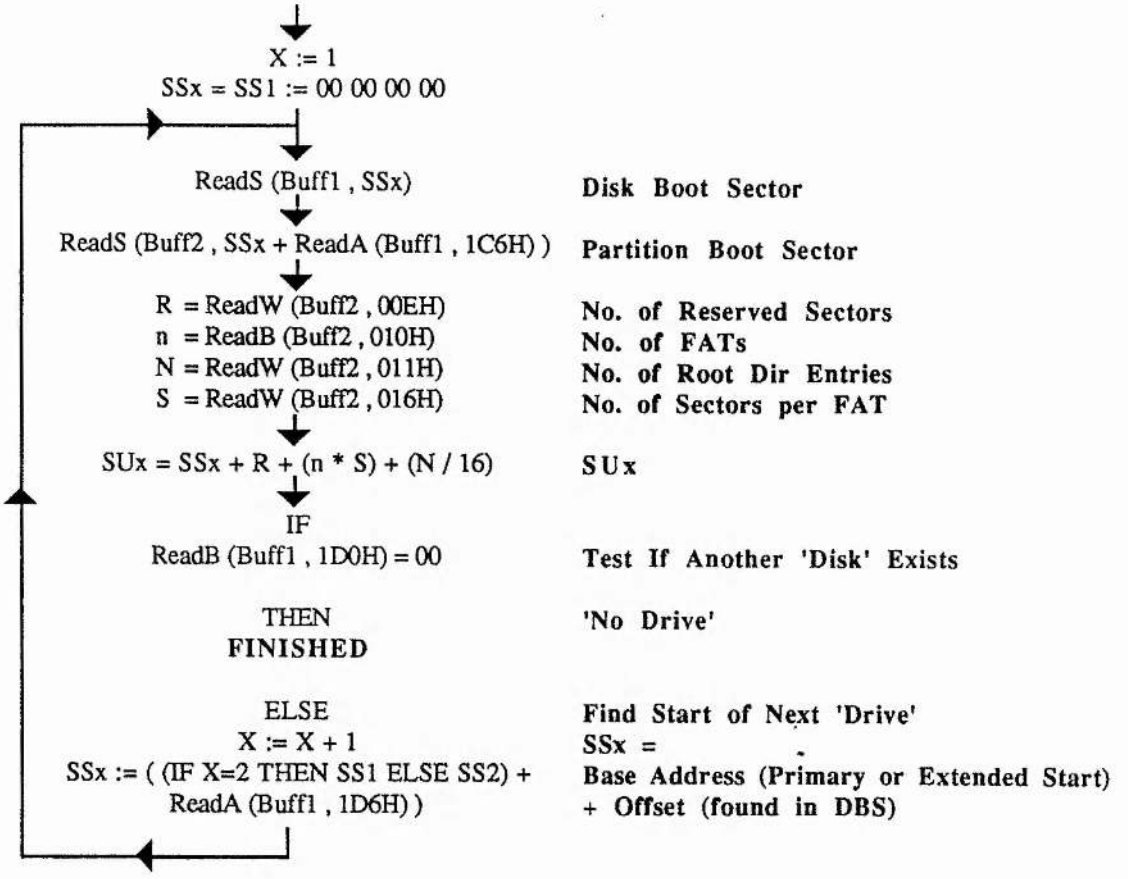


Figure 3.51 : Flow Diagram for Partition Boundary Address Acquisition

Firstly the disk boot sector is referenced to obtain the location of the partition boot sector, and the start address, SU_1, of the user segment is calculated from the information thus obtained, together with the modified address, $SU_{1(mod)}$.The disk boot sector is then re-examined to see if a further partition exists. If it does the address, SS_2, is obtained and the whole operation re-executed. All logical addresses obtained are relative to either the primary partition base address, SS_1, or the extended partition address, SS_2. Each partition created within the extended partition is treated as a logical drive. Each logical drive has its own disk boot sector, located within the first sector .

This in turn contains pointers to the start of the partition itself, and therefore the partition boot sector, and a pointer to any subsequent logical drives. This is illustrated in figure 3.52. The whole operation is repeated until all logical drives have been analysed.

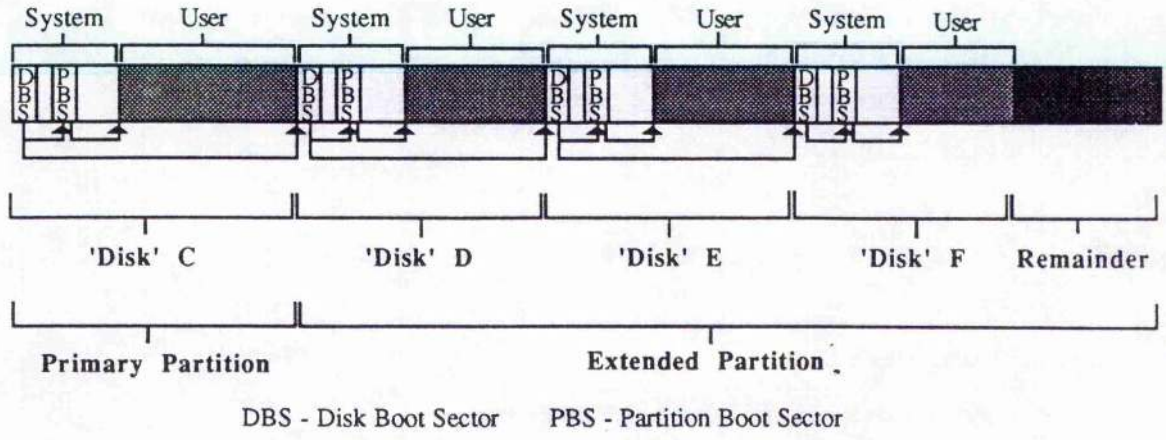


Figure 3.52 : Partition Layout With Regard to Boot Sectors

The Z80 assembler listing which implemented the flow diagram of figure 3.51 is shown in figures 3.53 through to 3.55. When this routine has run to completion, all the required address boundaries have been identified and pushed onto the stack in order.

```
genlist     ld    a,Targ              ;*Read Partition Table
            call  xrbitra             ;*(Disk Boot Sector)
            ld    ix,ramit+address
            ld    iy,buff1
            call  phasez

            ;* Find address of Start of partition
            ;* (where the BOIS Param Block is !!)

            ld    ix,ramit+address    ;*set index register

            ld    b,(ix+4)            ;*load in lower 16 of address
            ld    c,(ix+5)
            push  bc                  ;*Save lower 16 of addr - SSx
            ld    hl,(buff1+A1_L)     ;*load in lower 16 of offset
            add   hl,bc               ;*Add them
            ld    (ix+4),h            ;*And store them
            ld    (ix+5),l

            ld    b,(ix+2)            ;*load in upper 16 of address
            ld    c,(ix+3)
            push  bc                  ;*Save upper 16 of addr - SSx
            ld    hl,(buff1+A1_H)     ;*load in upper 16 of offset
            adc   hl,bc               ;*Add them + possible carry
            ld    (ix+2),h            ;*And return them
            ld    (ix+3),l


            ld    a,Targ              ;* Read BIOS Param Block

            call  xrbitra             ;* (Partition Boot Sector)
            ld    ix,ramit+address
            ld    iy,buff2
            call  phasez
```

Figure 3.53 : Assembler Listing for Partition Boundary Address Acquisition - Part 1

In the first section, contained within figure 3.53, the Disk Boot Sector is read from the first physical sector of the hard disk into buffer, 'Buff1'. The address SS_x is also pushed onto the stack. An offset is obtained for the location of the Partition Boot Sector (PBS) and used to read the PBS into a second buffer, 'Buff2'.

```
                ;* Generate address - SUx  and add to list

                ld      iy,buff2
                ld      b,(iy+n)            ;*B   = n
                ld      d,(iy+S_1)          ;*DE  = S
                ld      e,(iy+S_2)

                ld      hl,00h
                ld      a,00h
loop1           add     hl,de               ;*AHL = S*n
                adc     a,00h
                djnz    loop1

                ld      d,(iy+N_1)          ;*DE  = N
                ld      e,(iy+N_2)

                srl     d                   ;*DE = N/16
                rr      e
                srl     d
                rr      e
                srl     d
                rr      e
                srl     d
                rr      e

                add     hl,de               ;*AHL = N/16 + S*n
                adc     a,00h

                ld      d,(iy+R_1)          ;*DE  = R
                ld      e,(iy+R_2)
                push    de                  ;*Store R

                add     hl,de               ;*AHL = N/16 + S*n + R
                adc     a,00h

                ld      ix,ramit+address    ;*set address pointer

                ld      b,(ix+4)            ;*load in lower 16 of address
                ld      c,(ix+5)
                push    bc                  ;*Store
                add     hl,bc               ;*Add them
                push    hl                  ;*Save lower 16 of addr - SUx

                ld      l,a
                ld      h,00h
                ld      b,(ix+2)            ;*load in upper 16 of address
                ld      c,(ix+3)
                push    bc                  ;*Store
                adc     hl,bc               ;*Add them + possible carry
                push    hl                  ;*Save upper 16 of addr - SUx

                ld      hl,ramit+listlen    ;*Adjust size of list
                inc     (hl)
```

Figure 3.54 : Assembler Listing for Partition Boundary Address Acquisition - Part 2

Contained within the PBS is the BIOS Parameter Block. From this all the information required to calculate the position of SU_x and $SU_{x(mod)}$ is obtained. The code for this appears in figure 3.54. Firstly 'n' and 'S' are extracted and multiplied. They represent the number of copies of the FAT present and the number of sectors that each FAT requires, respectively, and their product indicates the total number of sectors used to store FAT information for the given drive. Next, 'N', the total number of root directory entries, is extracted and divided by sixteen, to convert the number to the sector requirement to store the entries. Each entry requires 32 bytes and there are 512 bytes per sector, therefore each entry requires 32/512 of a sector (ie a sixteenth). Lastly, 'R' is obtained, the number of reserved sectors. Such sectors are used for the partition boot sector itself to reside in. The the sum of the two values calculated above, together with 'R' provide a value for SU_x when added to the start address of the partition. 'R' itself, when added to the start address of the partition, provides the value for $SU_{x(mod)}$.

```
;* Go back and read pointer to next partition

ld    a,(buff1+1d0h)      ;*read sector number
or    a                   ;*test if 00h
                     ;*if 00 return as no more P's
jp    z,endbit

;* Get address of next partition

ld    ix,ramit+addres2    ;*load in copy of address

ld    b,(ix+4)            ;*load in lower 16 of address
ld    c,(ix+5)
ld    hl,(buff1+A2_L)     ;*load in lower 16 of offset
add   hl,bc               ;*Add them

ld    ix,ramit+address    ;*Store copy 1
ld    (ix+4),h
ld    (ix+5),l

ld    a,(ramit+listlen)   ;*load length of list

ld    ix,ramit+addres2    ;*Store copy 2 iff listlen=1
cp    01h                 ;*addres2=start of curr drive
jp    nz,loopx
```

```
                ld      (ix+4),h
                ld      (ix+5),l

loopx           ld      b,(ix+2)            ;*load in upper 16 of address
                ld      c,(ix+3)
                ld      hl,(buff1+A2_H)     ;*load in upper 16 of offset
                adc     hl,bc               ;*Add them + possible carry

                cp      01h                 ;*Store copy 2 iff listlen=1
                jp      nz,loopy

                ld      (ix+2),h
                ld      (ix+3),l

loopy           ld      ix,ramit+address    ;*Store copy 1
                ld      (ix+2),h
                ld      (ix+3),l

                ;* Repeat with next partition block

                jp      genlist
```

Figure 3.55 : Assembler Listing for Partition Boundary Address Acquisition - Part 3

Finally, the code illustrated in figure 3.55, checks for the presence of another logical drive and repeats the whole sequence accordingly. The buffer 'Buff1' is re-examined as it contains the Disk Boot Sector. It is tested for the presence of a second entry in the Partition Table. The easiest way to test for this is to check the value of the 'Beginning Sector' field : valid sector values range from one upwards, so if the value is zero then no entry exists. If this is the case then we have reached the end of the chain and all drives have been identified. A non-zero value indicates the presence of another drive and the pointer to it is obtained. As mentioned before, all addresses obtained are relative to one of two base values : the start of the primary partition or the start of the extended partition. The current base address is stored within the locations pointed to by 'Addres2', and is updated only once when moving from one partition into the other. The code sequence starting at the top of figure 3.53 and moving through Figure 3.54 and figure 3.55 is repeated until no second entry in a DBS Partition Table is detected.

Once this sequence is completed all required values are available on the stack. These values are popped off and placed in a table of the format illustrated in figure 3.56.

Segment	Address Field	Second Address Field	Flags				Access Count
			Read	Write	Alter	S/U	
Cs	Start Address	Unused	R	NW	U	S	0
Cu	Start Address	Modified Start Address	R	NW	U	U	0
Ds	Start Address	Unused	R	NW	A	S	0
Du	Start Address	Modified Start Address	NR	NW	A	U	0
Es	Start Address	Unused	R	NW	A	S	0
Eu	Start Address	Modified Start Address	NR	NW	A	U	0
Fs	Start Address	Unused	R	NW	A	S	0
Fu	Start Address	Modified Start Address	NR	NW	A	U	0
X	Start Address	Unused	NR	NW	U	-	0
	4 Bytes	4 Bytes	1 Byte				1 Byte

10 Bytes Per Entry

FLAGS

Read	R = Read Permission, NR = No Read Permission
Write	W = Write Permission, NW = No Write Permission
Alter	A = Alterable, U = Unalterable
S/U	S = System Segment, U = User Segment

Figure 3.56 : Permission Table

Each entry occupies ten bytes. The first four contain the logical start address for the user or system section of each partition. If the entry is for a user section then the second block of four bytes contains the modified address which will replace the first four bytes if that partition is activated. If the entry is not for a user section then these bytes remain unused. The last two bytes are used for status information. For the prototype, four bits are used : Read permission, Write permission, system/user indicator, and alter permission. Entries for the boot partition and the final entry which covers all unused space at the top end of the drive, are unalterable in that their Read and Write permissions may not be changed. All other entries may have their permissions altered during activation. The second status byte, purely for prototype testing, maintains a count of the number of accesses of that segment of the disk.

[3] Activating a Partition

When the machine has just booted, the permission list will contain the status information illustrated in figure 3.57.

Drive	Segment	Read	Write	Alterable
C	System	Yes	No	No
C	User	Yes	No	No
All Others	System	Yes	No	Yes
All Others	User	No	No	Yes
Above Partitions	N/A	No	No	No

Figure 3.57 : Permission list Status On Boot

Nowhere on the disk has write permission, and read is restricted to drive C and all system segments of the other drives only. The first violation of these permissions, which occurs within an alterable segment, will cause automatic activation of the partition to which that segment belongs. Violation occurs if the system segment of a partition is written to, or a Read or Write is attempted within a user segment.

The flow diagram used to determine Read permission is presented in figure 3.58. Firstly, a variable called 'Superv', which is maintained within the supervisor's RAM space, and initially set 'TRUE' on boot, is interrogated. If the value indicates 'FALSE', then the system is currently in unsupervised mode and all Read operations are permissible. This variable is set during the boot sequence when the mode is decided. If not, then the permission list entry is obtained for that segment of disk. If the Read permission value for that entry indicates that Read operations are allowed then the operation is permitted. If not, then two further tests are made : is the entry unalterable and has a partition already been activated? If either of these tests prove positive then the operation is not permitted as this violates the protection schema. A previous activation is indicated using a variable within the Supervisor's RAM which is initially false and made true after activation. If both tests prove false then the operation is allowable but the partition in question must be activated first.

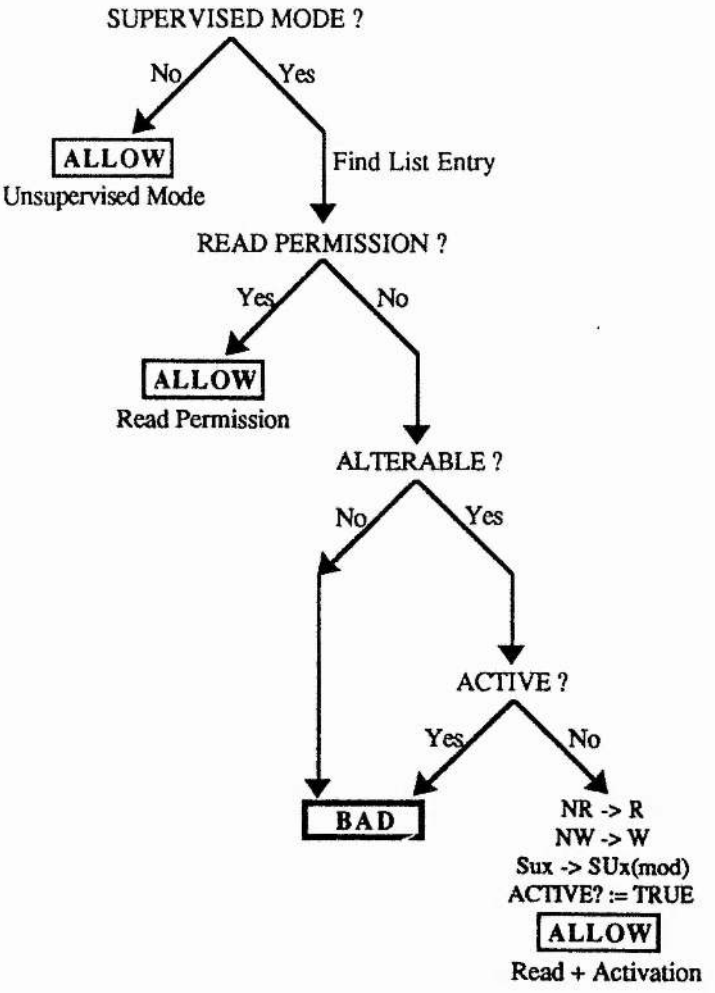


Figure 3.58 : Flow Diagram for Read Permission

As Read permission is only restricted initially within user segments the request therefore can be assumed to be within the user segment of the partition. The permission list entry which has been identified therefore requires only that Read and Write permission be granted and the modified address copied from bytes four through seven into the first four bytes for activation to be implemented. Finally, the variable 'Active?' is updated to indicate that activation has occurred as this may only happen once per session on the machine.

During what may be referred to as 'normal' operation of the machine, where a partition has been activated and all requests are within allowed segments, the number of tests required are kept to a minimum. Tests for unsupervised mode and for Read permission within supervised mode are made first and under normal conditions one of these will grant permission.

Write permission is slightly more complex and the flow diagram which is used to determine such permission is presented in figure 3.59.

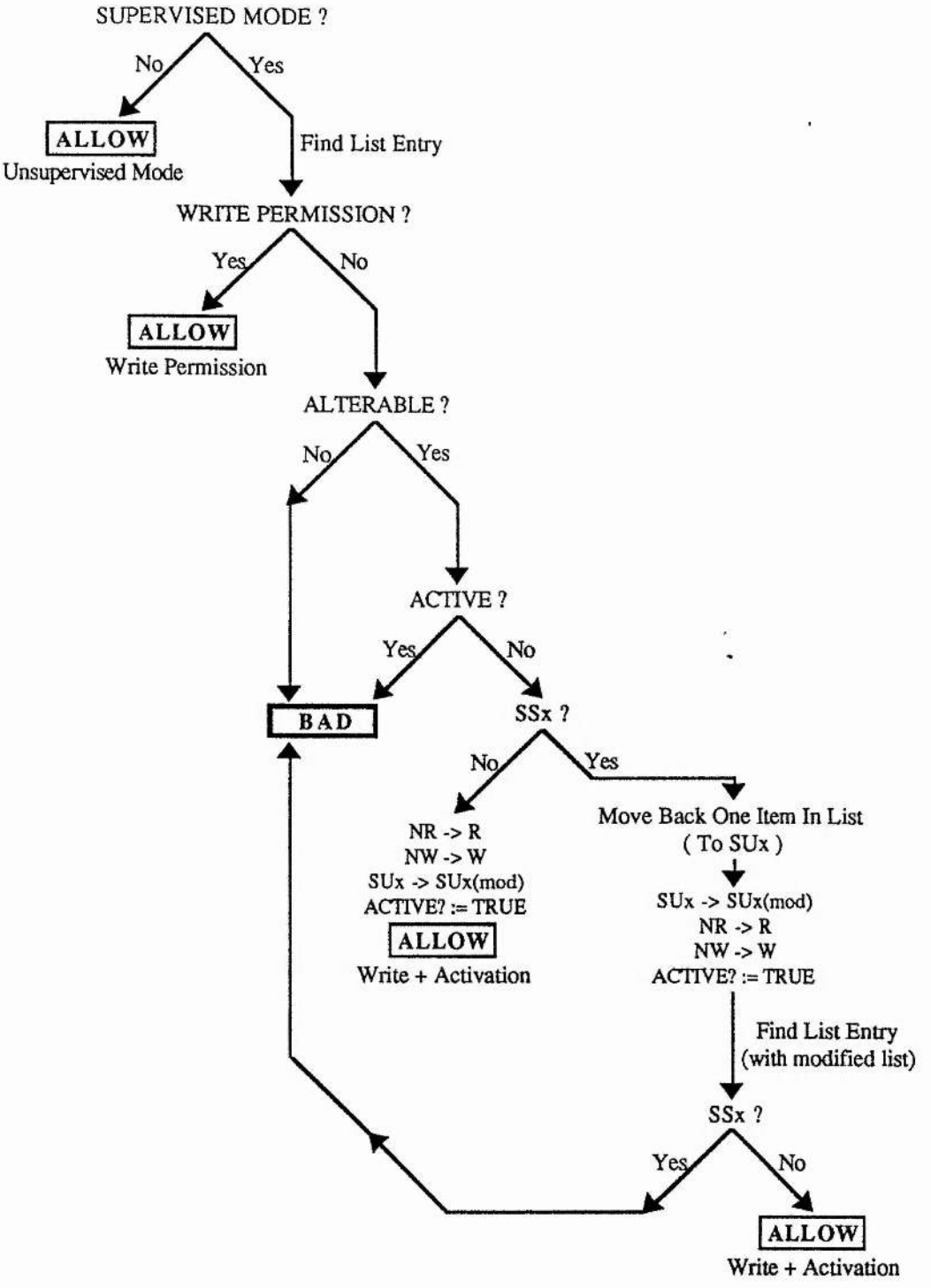


Figure 3.59 : Flow Diagram for Write Permission

The flow diagram closely follows that of Read permission to the point of activation. Once it has been determined that a Write operation is potentially allowable, but activation is required, a complication arises. Firstly, the request may be to write to within either a system or user segment as both have restricted Write and are alterable after boot. If the request is made to within a user segment, determined by testing one of the status bits of the permission list entry, then activation is identical as for Read, and permission is granted. If however the request is made to within a system segment, then in order to activate that partition, the entry corresponding to the user segment must be obtained as it is this entry which is updated, not the system segment entry. Once found, the entry is updated as previously described. The request must then be re-evaluated to find out on which side of the new boundary between system and user segments, it resides. If it still lies within the system segment then it must be a request to modify the boundary of the partition itself which cannot be allowed. If however it now lies within the expanded user segment then it is a request to modify either a directory entry or update a FAT table, either of which is permissible.

Once a partition is activated the permission list is of the format illustrated in figure 3.60.

Drive	**Segment**	**Read**	**Write**
C	System	Yes	No
C	User	Yes	No
Active Partition	System	Yes	No
Active Partition	User (mod)	Yes	Yes
All Others	System	Yes	No
All Others	User	No	No
Above Partitions	N/A	No	No

Figure 3.60 : Permission List Status After Partition Activation.

After activation, no further modifications may be made to the permission list. Attempted violation of access privileges will result in a warning and possible drive disablement.

[4] Finding an Entry Within the Permission List

Identifying the entry in the permission list which describes the segment of disk, which contains a data area to which a data transfer request has been made, is a time critical task. A significant degradation in performance is possible if the identification algorithm is inefficient.

In an attempt to minimise the instructions required to move about the list and test addresses, a series of measures were incorporated. The list is stored in reverse order, thus the first time the requested address is found to be greater than or equal to the entry's address, the entry has been identified. This means that only one bound of each segment need be checked, until the entry is identified. Once identified the upper bound of the request is checked to make sure it is still within that segment. Each entry in the permission list is of fixed length, incurring waste in system segment entries, but this ensures moving between entries can be implemented with a fixed length jump.

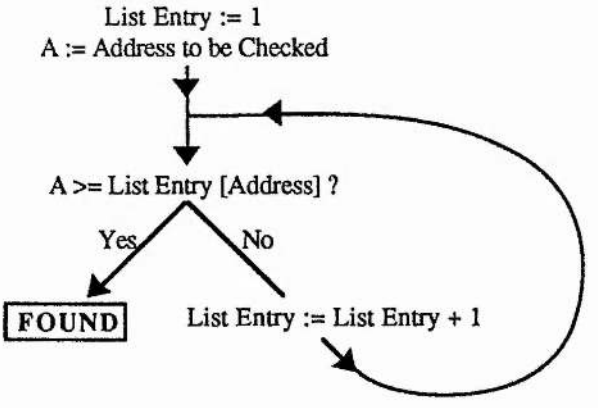


Figure 3.61 : Simple Permission List Search Algorithm

Figure 3.61 illustrates how simple the search algorithm can be made. If the algorithm illustrated in figure 3.62 is incorporated within figure 3.61 then, when testing whether the requested address is greater than, or equal to, the list entry address, minimum tests are carried out to identify whether the relationship holds. In figure 3.62, 'n' represents the byte number within the address starting with the most significant with value zero. If the address highest order byte is found to be greater than the corresponding entry byte then the entry has been found. If the former proves to be less than the latter then the correct entry has not been found so the algorithm is re-executed with the next entry. If neither of these are true then the bytes are equal and the next order byte must be tested. If all bytes in the address have been tested and they all prove equal then the correct entry has been identified.

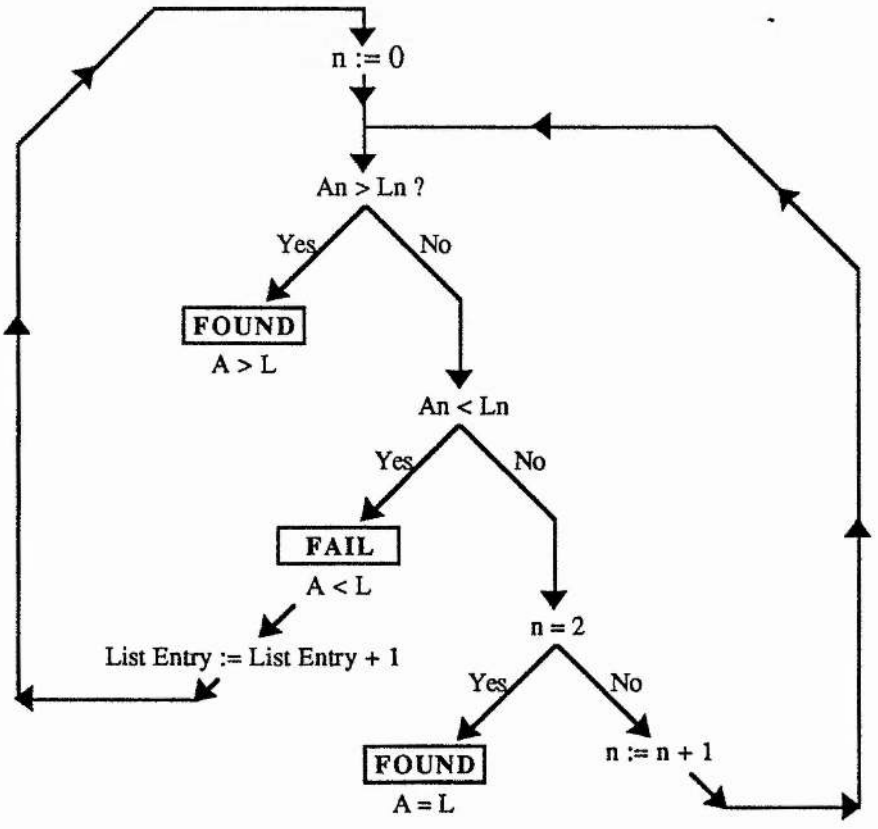


Figure 3.62 : Byte by Byte Testing of Addresses

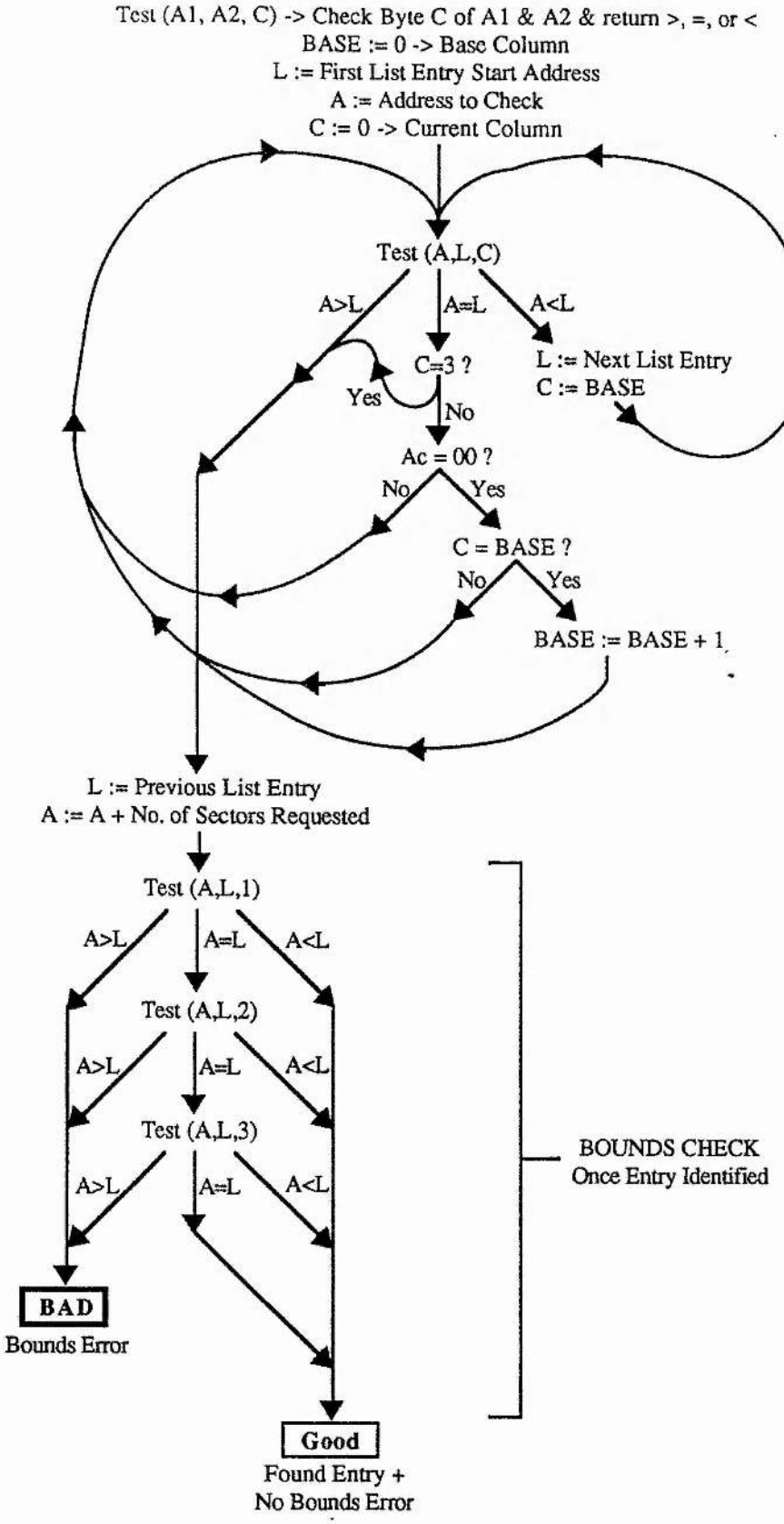
This algorithm ensures that the relationship between the two addresses under scrutiny is identified at the earliest possible stage, for each entry. For the prototype the highest order byte, the fourth, is ignored as the three lower order bytes can address over eight gigabytes of disk and it is not envisaged that drives of greater size than this will appear for the IBM PC family of machines for some considerable time, if ever.

A further improvement can be made, if a moving base is implemented. That is, for all bytes in the requested address that have zero value, once a corresponding zero in the entry table has been identified, that byte order is ignored for further entries. The implementation of this is relatively simple : first, the base is set to zero, the highest order byte. Then, when an event occurs where the requested address and entry address base order bytes are both zero, the base is incremented. When beginning a new test the algorithm always starts with the base order bytes, not the highest order bytes as before. An example of this is illustrated in figure 3.63.

```
A  =      00 00 21        Execution :

L  =      01 1A FF        base := 0
          00 FA 01        A1 < L11       : L1->L2, n:=base
          00 A5 20        A1 = L21 = 00  : base:=1, A1->A2, L21->L22
          00 50 00        A2 < L22       : L2->L3, n:=base
          00 21 1F        A2 < L32  **   : L3->L4, n:=base
          00 11 11        .......
          00 0A 1A        A2 = L82 = 00  : base:=2, A2->A3, L82->L83
          00 00 37        A3 < L83       : L8->L9, n:=base
          00 00 00        A3 > L93  **   : BINGO !!!

Ay  = Requested Address, byte y
LXy = Permission List entry 'X', byte y
**  = base has moved -> result itallic values never checked
For this example 8 tests have been avoided, and associated jumps
```

Figure 3.63 : Example of Moving Base Algorithm Improvement

The final algorithm with all these improvements implemented appears in figure 3.64. This includes a boundary check which is executed only when the correct entry has been found. This check makes sure that the number of sectors requested does not cross over into the next segment.


Test (A1, A2, C) -> Check Byte C of A1 & A2 & return >, =, or <
BASE := 0 -> Base Column
L := First List Entry Start Address
A := Address to Check
C := 0 -> Current Column
Test (A,L,C)
A>L
A=L
A<L
C=3 ?
Yes
No
L := Next List Entry
C := BASE
Ac = 00 ?
No
Yes
C = BASE ?
No
Yes
BASE := BASE + 1
L := Previous List Entry
A := A + No. of Sectors Requested
Test (A,L,1)
A>L
A=L
A<L
Test (A,L,2)
A>L
A=L
A<L
Test (A,L,3)
A>L
A=L
A<L
BAD
Bounds Error
Good
Found Entry +
No Bounds Error
BOUNDS CHECK
Once Entry Identified

Previous Page : Figure 3.64 : Fast Permission List Search Algorithm

The code implementation of this algorithm returns a pointer to the entry in the permission list. This allows for interrogation of the permission status bytes, and possible partition activation as a result.

[5] Parallelism within the system

Clearly, with the addition of a dedicated supervising processor, the possibility of a certain amount of parallelism exists. This occurs in many places within the schema : both processors working after an effective fork where the IBM initiates a supervisor event ; followed by an effective join, where the IBM waits for a status byte to be returned by the supervisor.

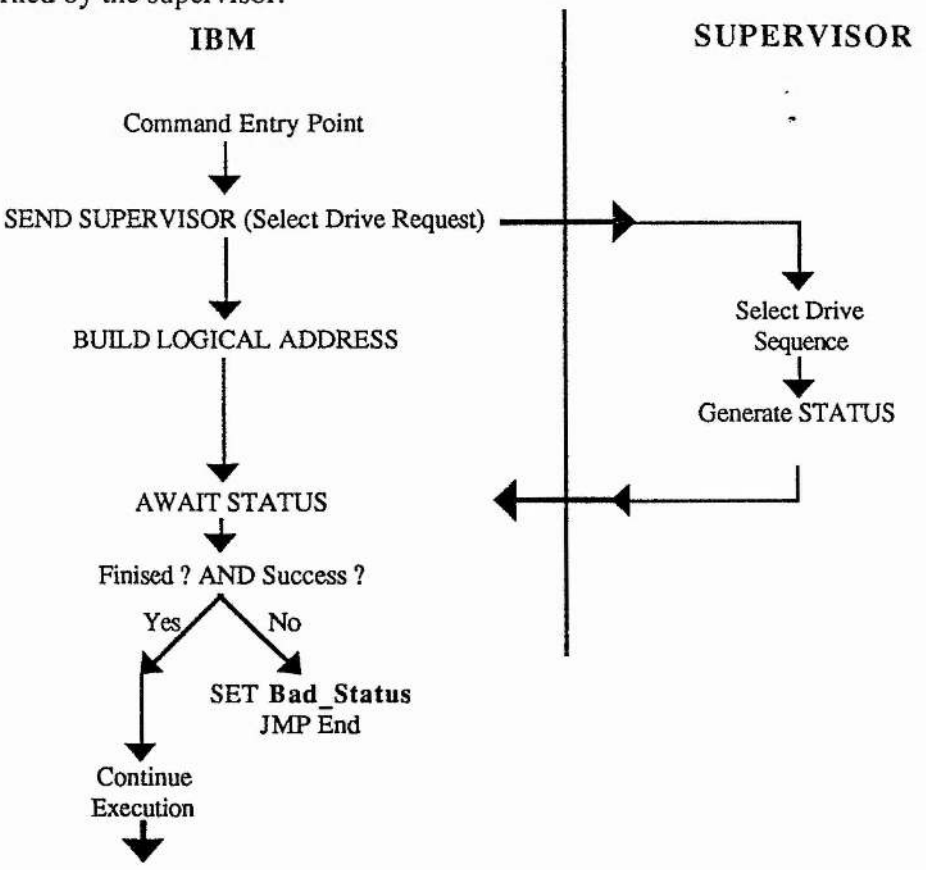


Figure 3.65 : Example of Parallelism

Figure 3.65 illustrates one occurrence of parallelism in the system. As soon as a request for a hard disk function is made, the IBM issues a select drive request to the supervisor. The IBM then computes the logical address required, from the cylinder, head, and sector values passed to the INT 13H call. Whilst this is happening, the supervisor attempts to select the required drive ready for access. When the IBM has completed its task it then waits for a 'success' status from the supervisor before continuing. Thus the drive is selected in effectively zero time.

Savings made in this way offset the overheads incurred during interrogation of the permission list.

3.1.2 TESTING

Manual Testing Using Debug

A routine was provided within the firmware which, when invoked, transfers the contents of the permission list, byte by byte, to the IBM. An executable program was further written, PRLIST.EXE, which invoked the aforementioned routine in firmware and displayed the permission list in readable form. This provided an easy means by which the virus protection schema could be verified.

Firstly, a drive was formatted and partitioned. The permission list was then displayed using PRLIST. Simple calculation could then verify that the system boundaries were correct.

The utility DEBUG.COM, provided by Microsoft, as part of the DOS operating system software was then used. This utility allowed direct calls to the hard disk services interrupt, INT 13H. Using this facility one could read all required sectors in order to manually check the user boundaries present in the permission list.

Once the permission list was checked the primary partition, 'C', could be tested by attempting to write a sector to it using DEBUG again. This was a violation, and a visual warning resulted. This was repeated with both read and write, from and to, the upper disk area, above the top partition. Once this was completed, the 'Unalterable' areas of the disk were shown to be operating correctly.

Partition activation was tested by both writing to the directory of the system segment, and reading and writing, from and to the user segment of a partition. This resulted in the expected alteration of the permission list, which was viewed using PRLIST, once more. Writing to the partition data area of the system segment was also shown to produce a warning, and prevent activation of the partition.

After successful activation, various segments were tested for Read and Write. All types of partition : boot/primary, active and other, were shown to behave as required. Attempts to transfer a block of data which begins within a valid segment, but crosses into another invalid segment, were also made, and failed as expected.

A systematic check of all possible access requests to all possible segments and partitions was made and the system behaved as required, by the definition of the schema. Repetitive batch files were also executed which copied files from one place to another, and then back. After many hours the information being copied was found to be intact, indicating that no intermittent errors were present in the SCSI transfer firmware or hardware.

Introducing a Virus

The ultimate test had to be made. A particularly virulent virus, the DARK AVENGER or 'EDDIE' virus, was introduced into the system. This virus is probably the most widespread and dangerous Bulgarian virus to date. To quote the author of the virus :

" In early March 1989 Version 1.31 of the Dark Avenger virus was called into existence and started to live its own life to all engineers' and suckers' terror. So, never say die. Eddie lives on and on and on.... "

This virus is memory resident and infects during execution or opening of a file. The virus was introduced into the system, by execution of an infected program, which was placed within an active partition. Instantly, the system halted with a warning message. No files were infected as the DARK AVENGER, in common with most viruses, attacks the COMMAND.COM file in drive 'C' first, to ensure that it is executed on boot. This attempt was trapped instantly by the protection system as drive 'C' always has 'read only' status. One should however realise that, although no files were infected in this case, the virus did have the ability to infect, corrupt, or destroy all, or some, of the information stored within the active partition.

CHAPTER 4

POSSIBLE ENHANCEMENTS

4.0 PARTITION ACTIVATION BY EXPLICIT CHOICE

The choice of active partition can be made at boot time, ensuring the user has explicit knowledge of the choice. A menu, similar to the example in figure 4.0, would be presented to the user.

PARTITION REQUIRED	
D	MISCELLANEOUS
E	WORD
F	DBASE
G	ACCOUNTS
H	DAVE - PRIVATE
I	JOHN - PRIVATE
J	FRED - PRIVATE
K	GAMES

Figure 4.0 : Choice of Active Partition from a Menu

A field would be provided, defined by the unsupervised user, which would appear in the menu alongside the drive letter. This field would be used to provide a brief description of the drive's contents. This would provide vital information for the user, as this method does not allow perusal of the root directories of drives before making the decision.

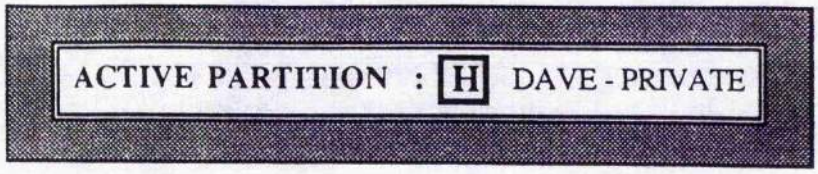


Figure 4.1 : Screen Message After Choice Has Been Made

After the choice has been made the user would be presented with a message on the screen, similar to figure 4.1, reaffirming the decision. This message could be re-invoked by executing a utility program, should the user forget.

This method ensures that the user is fully aware of the partitions which he, or she, may access, and the Read and Write permission associated with them. This method also facilitates certain other improvements introduced below.

4.1 MULTIPLE READ ONLY PARTITIONS

As mentioned previously, any executable file which is virus free, and not self-modifying, may be placed within the boot partition, 'C'. Any executable file thus placed cannot be infected due to the 'Read Only' status of the partition : it is guaranteed virus proof. There are a few situations that may occur where such a guarantee would be desirable, but the boot partition would not be suitable : the 'C' drive may not have the required space, and reformatting the disk and altering the size allocated to 'C' may not be a realistic solution ; or the software may not be self-modifying, but may not be one hundred percent guaranteed virus-free ; or simply, one would not wish to clutter the boot partition further by adding yet another piece of software.

PARTITION REQUIRED	
E	WORD
F	DBASE
G	ACCOUNTS
H	DAVE - PRIVATE
I	JOHN - PRIVATE
J	FRED - PRIVATE
K	GAMES

Figure 4.2 : Menu After Partition 'D' Made Read Only

An improvement that could be made would be to offer the facility of marking a partition 'Read Only' to the unsupervised user. This partition subsequently would not appear as a possible partition for activation. It would, however, be presented as being available, for reading only, in the message which follows after the choice is made, ensuring the user is aware of the possibility. For example, if, in the configuration presented in figure 4.0, the 'Miscellaneous' partition ('D') was made 'Read Only', the menu would appear as in figure 4.2, with the message following shown in figure 4.3.

ACTIVE PARTITION :	H	DAVE - PRIVATE
READ PERMISSION :	C	BOOT
	D	MISCELLANEOUS

Figure 4.3 : Message After Partition 'D' Made Read Only

This facility would provide a dynamic allocation of Write protected disk space, guaranteeing virus protection for programs which are not self-modifying, and protecting data that does not require updating from corruption.

4.2 SECURITY SHELL

Existing within the schema already, is a mechanism for password entry restriction, to unsupervised mode. This concept could be extended into supervised mode, to provide a security shell similar to those found in mainframe, multi-user systems.

The unsupervised user would have the ability of adding a user to the system. Once a username is created, the user, when first attempting to use the machine, would be required to enter a password. These username/password pairs would be stored in the same manner as for the unsupervised mode password where the information is totally secure. Password entry would be by the same method also : byte by byte before DOS is loaded.

If a correct password was entered, then entry would be given to supervised mode. This in itself would insure that no unauthorised users could gain any access to the system. This concept can be extended, however, to provide more versatile and comprehensive security.

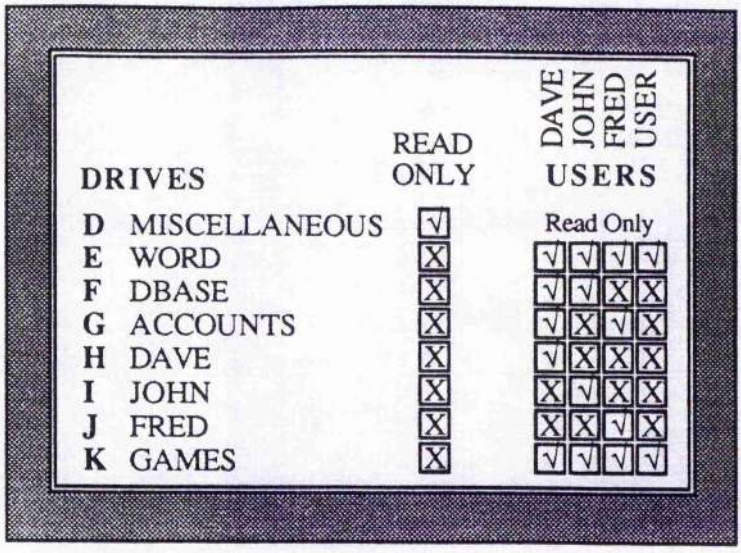


Figure 4.4 : Allocating Partitions to Supervised Users

A mechanism could be further provided, so that the unsupervised user could allocate a subset of the possible partitions to any given user. To continue the example above, figure 4.4 represents a possible security shell implementation, where the unsupervised user defines the partitions available to each user through alteration of a table, generated on demand at boot time. This table would be generated with a column for each defined user, and a row for each defined partition. A column has also been added here to implement the 'Read Only' partition mentioned in the previous section. For each partition which is not rendered 'Read Only', permission of use can be allocated on an individual user basis.

In the example in figure 4.4, users Dave, John, and Fred have a partition allocated to them which is only accessible to them as individuals. Clearly, these partitions can only be infected by the user to which they are allocated, and so one can guarantee that no

infection can be introduced by another user's carelessness. Thus a careful user can be assured that a partition allocated to he, or she, alone cannot have been infected by someone else during a previous session on the machine.

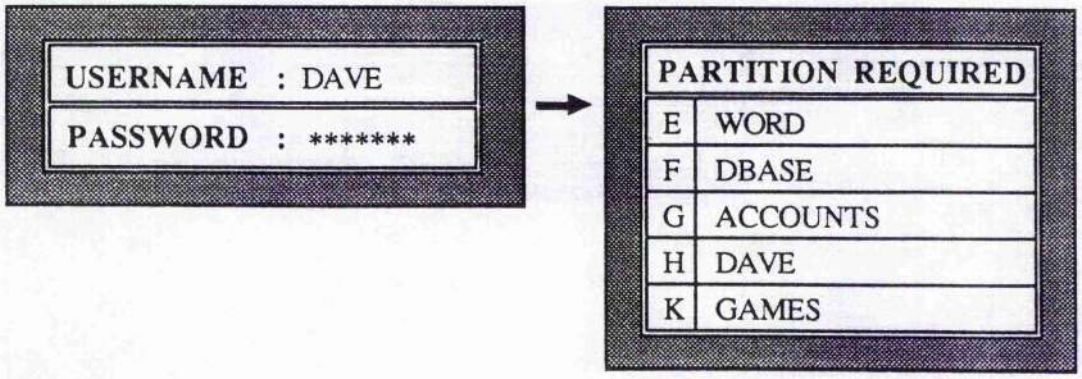


Figure 4.5 : Example Login Sequence

Figure 4.5 illustrates an example login sequence for user 'Dave' for the example configuration in figure 4.4. Considering purely the security aspect of this concept, one can also see that more use can be made of an individual machine if certain individuals cannot access certain information held on that machine. Imagine the scenario where Dave is the manager, Fred the accountant, and John the salesman for the company which own this machine. Previously, it is unlikely that John would be allowed to use the machine as all the salary information is stored within the accounts partition, and he cannot be allowed to have access to this. With this system running, however, access can easily be denied, and John would no longer require a separate machine for his client database, and for writing letters.

User names need not be allocated on an individual basis. In the example above, the username 'User' has been introduced which gives use of the 'Word' and 'Games' partitions. The password of this username could be given to all employees, who have not been allocated an individual username. This allows all employees use of the low security software on the machine, which they may require at some time.

A possible addition to this might be a username, such as 'Guest', which requires no password, but gives access to only the boot partition and 'Read Only' partitions. Such a user would not be able to activate a partition and therefore would not be able to write to the disk. Such a user could not possibly introduce a virus, but could do useful work, saving results onto a floppy disk on completion.

Clearly, such a security shell could provide more versatility, both to the machine in general, and to the virus protection mechanism itself.

4.3 FLOPPY SCHEMES

A major source of infection is the floppy disk. With the virus protection schema we can guarantee that certain areas of the disk remain virus free and that general partitions become contained units, rendering partition to partition infection impossible. Infecting an active partition is, however, still possible via a floppy disk. A few possible methods of reducing, or removing the chance of infection by this method are presented below. They all depend on the addition of the floppy disk controlling hardware onto the supervisor card.

4.3.0 DENIAL OF ALL REQUESTS IN SUPERVISED MODE

This is the simplest solution. All requests to read or write, from or to, a floppy disk, whilst in Supervised mode, are denied. This insures that infection of the active partition cannot take place. It does, however, also prevent the backing up or restoring of software, or the introduction of valid files. This would therefore require the Unsupervised user to introduce all software, and backup all files on a regular basis. This method of restriction would heavily reduce the functionality of the machine, and would impinge on the freedom of the user in the completion of tasks. It is, however, the most rigourous solution, and could be implemented, provided the Unsupervised user was particularly diligent and regularly available.

4.3.1 RESTRICTED ACCESS IN SUPERVISED MODE

With the security shell implemented, as mentioned above, it is possible to provide a restricted access mechanism, in one of two ways.

One way would be to restrict access to certain users. This could be implemented by the addition of a row to the access table where access to the floppy drive can be granted on an individual user basis. The advantage of this would be that a subset of trusted users could have backup abilities, whilst ensuring less trustworthy users, or those who would not require such access, do not. This would reduce the chance of infection and maintain a small, well defined group who are responsible for the state of the machine. No partitions, save 'Read Only' ones, could be guaranteed virus free by this mechanism.

A second way would be to restrict access to sessions on the machine where certain partitions are active. This could be implemented by the addition of a column to the access table. Dependent on the choice of active partition, a user may or may not be granted permission to access the floppy drive. This has the advantage that certain partitions, without floppy permission can be guaranteed virus free as for the floppy scheme presented in section 4.3.0. These partitions would have to be backed up by the unsupervised user. Certain other partitions would have permission and in this case backing up and maintaining a virus free condition would be the responsibility of the users. For example, continuing with the access table in figure 4.4, partitions 'Word', 'Accounts' and 'Dbase' could have floppy access denied and be maintained by the unsupervised user, whilst partitions 'Dave', 'Fred' and 'John' would have floppy access and would be the responsibility of the user assigned to them. The 'Games' partition would probably be given access rights, as this partition would have very low security associated with it, allowing users to introduce games, and possibly viruses, at will. This would prevent employees being tempted to introduce the games elsewhere, risking infection within partitions where important data is kept.

4.3.2 FULL ENCRYPTED ACCESS IN SUPERVISED MODE

Ideally one would wish for complete virus protection, rather than containment, whilst allowing the backing up, and restoring, to and from, floppy disk. This could be implemented using a reasonably complex method involving encryption.

The implementation of this method requires that a portion of the Z80 ROM be filled with a random sequence, different for each ROM. The supervisor would segment this sequence into a machine number, and a unique partition number for each partition.

In supervised mode, all Write requests to a floppy disk would cause the data to be encrypted, before transfer, using the machine number and partition number corresponding to the active partition, as the lock and key to the encryption algorithm. All Read requests would be similarly decrypted. See figure 4.6.

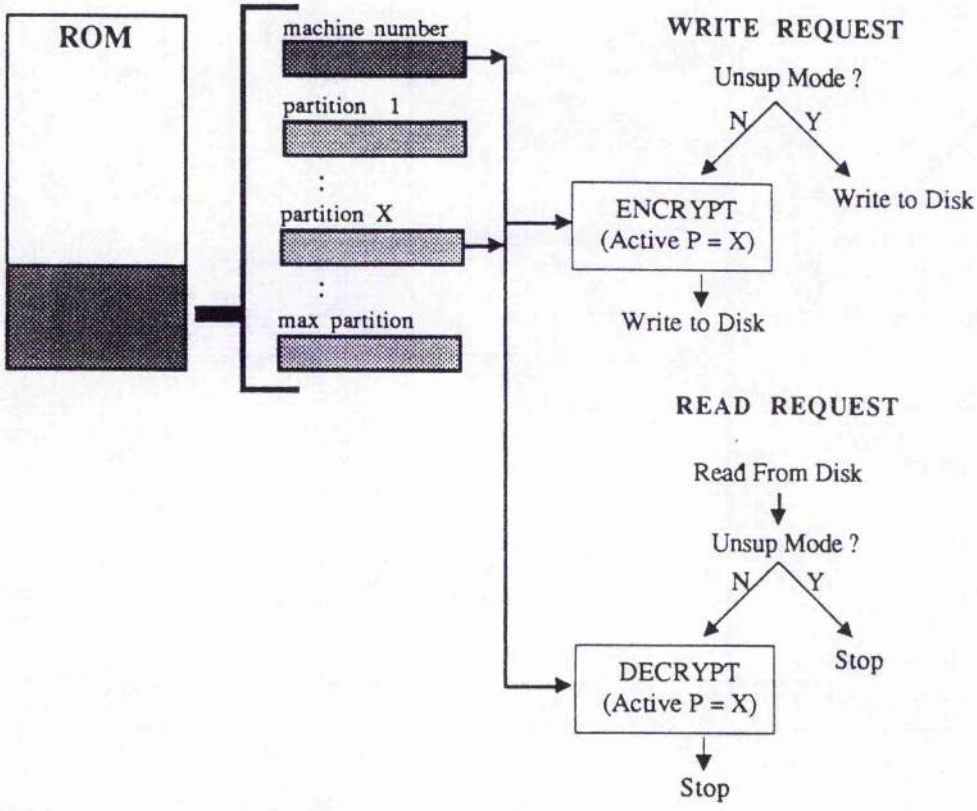


Figure 4.6 : Encryption of Floppy Disk Data in Supervised Mode

Thus, a floppy disk formatted, and written to, by a supervised user, may only be read when the disk is returned to the same machine, with the same partition active. This means that no software may be introduced into a partition, unless it originated from that partition. It also means that this particular floppy disk cannot cause a virus to be spread, within the machine, or to an outside machine, as it is unreadable outwith its defined partition.

A partition with encryption implemented can therefore be guaranteed virus-free, whilst allowing users of the partition floppy access for vital backing up and restoring. It should also be noted that important, sensitive data cannot be removed from the partition and read elsewhere, providing a secure way of preventing theft and careless spread of data.

Using the ROM as storage of the random sequence ensures that no user can gain access to the sequence, whilst guaranteeing that, even after a disk crash, data can be retrieved from backup floppies. Alteration of the partition structures after such a crash from what they had previously been, by the unsupervised user, could mean that the partition which encrypted the floppy disk would no longer exist. The unsupervised user would be provided with a utility which would decrypt any floppy encrypted on the machine by trying all possible unique partition numbers, which could be used either for restoring after a crash, or for introducing information into a different partition. New software could only be introduced by the unsupervised user, which is the ideal case.

It is probably desirable that only certain partitions be protected in this manner, as it would require a reasonable amount of diligence of the user to mark floppies with the machine and partition indicated. Failure to do this would mean that a floppy introduced into an incorrect partition would appear unformatted and it could be mistaken for a blank disk. Partitions requiring only low security, such as a games partition, could be left unprotected. Implementation would require the addition of a row in the access table indicating whether encryption would be required.

This method provides the desired level of protection and is reasonably transparent to the user and operating system. All that is required is that a user is fully aware that a floppy disk, once formatted, can be used solely when using the machine and partition that were in use during the format.

CHAPTER 5

ALTERNATIVE IMPLEMENTATIONS

5.0 PROCESSOR FREE SOLUTION

The virus protection schema, so far described, relies on the use of a second processor, the supervisor, to control access to the hard disk drive. The addition of this processor ensures that decisions can be made without possible intervention or corruption by an advanced virus. An alternative schema was considered which would not require such a processor, but would use only the CPU of the machine, whilst maintaining the incorruptibility of the system.

Most of the firmware would be ported to run directly on the 80x86 processor of the IBM. Using the small circuit in figure 5.0, a small area of RAM would be provided which could be written to after a cold boot, but could subsequently be rendered 'Read Only'. Effectively this RAM could be converted to ROM during a session on the machine, but not converted back. The flip-flop in the circuit powers up with a 'Q' output of logic zero, as the capacitor connected to the asynchronous 'Clear' input ensures that the input is held low long enough to clear the flip-flop's contents. A 'Q' output of zero allows write pulses to propagate from '!WE Input' to '!WE Output' and thus the RAM may be written to. However, a single pulse sent to the 'Clock' input causes the 'Q' output to go to logic one and prevents all write pulses from being propagated. This operation is non reversible and thus the effect is to convert the RAM to a ROM.

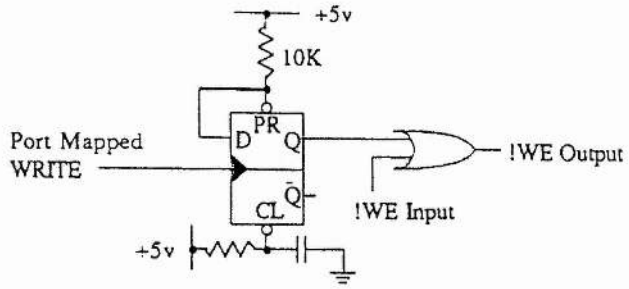


Figure 5.0 : RAM to ROM Conversion in Hardware.

This RAM would be filled with the permission list on boot, and then rendered 'Read Only' before loading the operating system. This would ensure that no user or virus could tamper with the list.

A mechanism, which guarantees the decision making firmware being executed, would also be required. If we can ensure that it is actually the ROM which is being executed and not a program simulating the ROM which is being executed from elsewhere, then this would be the case. The simple circuit in figure 5.1 would ensure this.

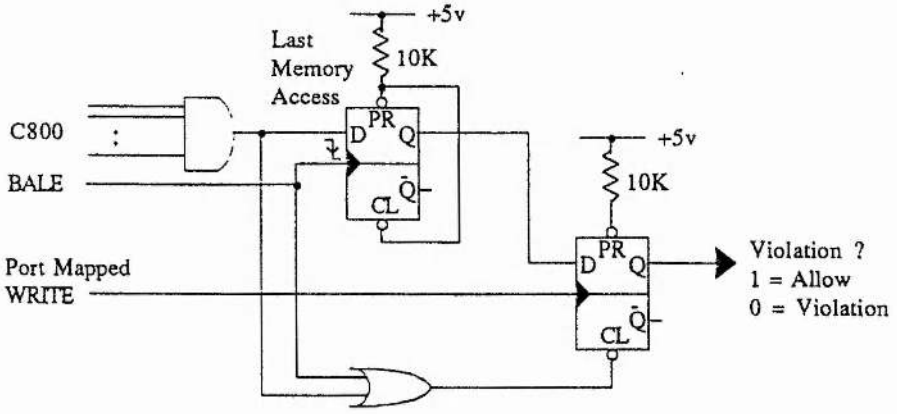


Figure 5.1 : Hardware to Ensure Execution Within ROM

At the start of a disk request, the firmware would write to the latch in figure 5.1, setting the output to one, if and only if, the instruction came from within the supervisor ROM. This would enable the SCSI controller for use. If, at any time in the future, an access is made to an address which resides outwith the controller's ROM and RAM, then this latch is reset, and therefore the SCSI controller disabled. Thus, if this latch is set at the start of a disk request, and has not been reset by the time data transfer is to begin, then the request has been cleared by the supervising firmware. The latch is automatically reset on completion of the disk request, by the far return instruction.

A virus cannot set the latch, as it cannot be executing from the area where the ROM resides. Nor can it interrupt a disk request and divert control as this would cause an immediate reset of the latch. Thus we have reached our objective : no virus can interfere with the supervisor's operation.

The advantage of this implementation would be that the 'Supervisor' would operate at a speed dependent on the system clock. Thus, faster machines would not incur a greater relative overhead from the virus protection system, as potentially they might from a distinct processor operating at a fixed clock rate.

5.1 DRIVE FIRMWARE SOLUTION

SCSI bus devices, by definition, are intelligent. Therefore, in general, SCSI disk drives include a fairly general-purpose processor. The 'Supervisor' firmware could be ported to run on this processor, as an outer shell, executed before the existent firmware. A replacement BIOS ROM would still be required, but many manufacturers provide empty sockets on the motherboard. Communication between the 'Supervisor' and the replacement BIOS ROM,would take place over the SCSI bus using vendor unique commands, created specifically for the task.

This implementation removes the requirement for a second processor, but it requires the co-operation and aid of a SCSI disk drive vendor. The final product would then have to be packaged as a drive/ROM pair, rather than purely a card for replacement in a system with an existent drive.

5.2 IN-LINE SOLUTION

Certain computers cannot be modified by the addition of a card. The 'classic' series of Apple Macintosh come into this category. Such machines are supplied with all requirements installed, and are relatively sealed as far as the owner is concerned. This series of computers has an external SCSI port supplied, and all required hardware and firmware installed within. Thus, all implementations discussed so far would be entirely unsuitable. The possibility of implementing the virus system as a module to be inserted between the machine and drive does however exist.

An in-line implementation would have to maintain two distinct SCSI buses, one connecting itself with the drive, and the other with the computer system. The computer system would make a request to the 'supervisor module' for consideration. If the request was to be permitted, the supervisor module would indicate this to the computer system, and data would be transferred, the supervisor module transferring the data from one bus to the other.

This would be a moderately inefficient implementation, but would be easily installed in an existing system. It is also the only implementation possible for such a closed system.

5.3 USE OF AN ALTERNATIVE BUS

The SCSI bus was used for the prototype due to its speed and versatility. SCSI is an intelligent interface, and far superior to the ST506 standard IBM disk interface. However in order to use SCSI, both hardware and firmware, in the form of a disk BIOS, must be added to the system. As a modified BIOS and an adapter card were both required for this project, this presented no problem.

Other alternatives are now possible : 'Synchronous SCSI' and SCSI II are making their debuts, as is the new 'AT' interface. Implementing synchronous SCSI would require only upgrading the SCSI controller chip, and associated setup and control firmware. The result would be a possible increase in transfer speed to 10 Mbytes per second. SCSI II could also be implemented which would require much the same changes as for Synchronous SCSI, and would also yield the performance benefits due to the advanced capabilities of the bus.

Adapting to the 'AT' interface would require similar changes : in both the controller and the firmware. Whilst eliminating the requirement of mapping the IBM task file address to a SCSI logical address, the problem of maintaining protection would, however, be complicated by the nature of the addressing system.

The virus protection system is not in any way interface dependent. Any modern disk interface could be utilised in the design.

CHAPTER 6

INTRODUCING COMPRESSION

6.0 INTRODUCTION

As hard disk technology approaches the finite limit defined by the physical limitations of the component parts of the drive, new ways must be found to provide an increase in disk space. Disk heads can only be made so small, disk platters spun so fast, and only so much information stored on a finite space.

Only two solutions exist : increase the area and number of disk platters ; or reduce the amount of information required to be stored. The physical size of the device is becoming increasingly important with the advent of the laptop, and now notebook, computer. Thus the former solution can only be considered viable when regarding large systems, not personal machines. The latter solution can be implemented using compression techniques, but this introduces certain major problems.

Two major techniques have been developed which make lossless compression possible. Lossless compression requires that the information compressed must be able to be decompressed to an identical copy of the original. This is clearly a requirement for computer data storage in general. However, when an image is to be compressed, one may accept the very high compression ratios of lossy algorithms, at the expense of exact data reproduction.

The first lossless compression technique was presented in 1952, by David Huffman[1]. Now known as Huffman Coding, it involved reducing the number of bits used to represent frequently appearing characters in a given input stream, whilst increasing the number of bits required for less frequent characters. The second technique, the LZW method[2,3], compresses data by encoding strings of characters, generating an expanded alphabet based on the strings appearing in the input stream. Both these techniques work by reducing the redundant information which exists in the data. An explanation of these techniques is provided in section 6.2.

Real time, in-line compression/decompression is only now becoming a reality, forty years after Huffman first conceived of the idea, for two reasons : firstly, up to now it has been cheaper to maintain research into other aspects of storage, such as improving the physical aspects of drive technology mentioned previously ; and secondly, electronics has only recently advanced to the point where compression can be implemented as a hardware state machine, and therefore be fast enough.

With the arrival of such compression chips, investigation seemed appropriate. It appeared that, although compression was now possible, an appropriate allocation schema was proving to be somewhat of a stumbling block. With a block of data compressed to an indeterminable size, would allocation prove to be as much of an enormous task as memory allocation in multiprocessing computer systems?

Including both compression and virus protection as a complete and transparent add-on to a machine would be desirable. The supervisor could take control of the compression so that it was transparent, even to the IBM BIOS. Certain other benefits, provided by the strictly maintained regime of the virus protection system, might also be found.

6.1 THE SCHEMA

The largest problem with implementing a compression system for hard disks is the allocation of disk space. Clusters would no longer be of fixed length. Such problems have been researched before with the allocation of memory in multiprocessing systems, and in the file disk space allocation by an operating system. The techniques involve maintaining complex lists of free space available, and allocating sections of the free space to requests using various methods to reduce fragmentation.

Allocation on this basis would have to be dynamic. A file allocated with space on the disk might be read, amended, and no longer fit after recompression in the original space. The file would therefore require to have alternative space reallocated.

A possible solution exists which may circumvent the requirement for such complex allocation mechanisms. It involves a static mapping from the operating system logical address, to the hard disk logical address, coupled with a system of overflow storage. For this solution to work, one must have a reasonable idea of the compression ratio, found by experiment, which can be expected for the system. This is used to find a sector size which would be adequate to store a high proportion, between seventy and ninety per cent, of compressed DOS clusters. The disk is thus segmented, and a mapping found from cluster address to disk address. The clusters which do not compress to the required size will have the overflow stored within a reserved area on the disk, to be recalled as required. This is presented formally in figure 6.0.

Definition DOS sector size $= 2^K$
DOS minimum cluster size $= Mc$ (all cluster sizes a multiple of this)

Action Reduce disk sector size $= 2^k$ **where k<K**
(this gives more grades of possible compression)

From Experiment choose compression ratio (C) so that 80% say compress to at least this extent where :

$$C \in \left\{ Mc * 2^{\left(\frac{(K-k)}{x}\right)} \mid x = 1 .. Mc * 2^{(K-k)} \right\}$$

Choose an area (**R**) large enough to store the overflow data of the clusters which are 'oversize' after compression.

Logical disk size : $L = (P-R)*C$ (where P = Physical size of disk)

C_i = number of sectors to be transferred by DOS
C_o = number of compressed sectors actually transferred to disk

$$C_o = C_i * 2^{\left(\frac{(K-k)}{C}\right)}$$ (where Ci in 2^K size sectors, Co in 2^k size sectors)

A_i = DOS logical address
A_o = Disk drive logical address

$$A_o = \frac{C_o}{C_i} * A_i$$

we now have a **P to L capacity drive** and if physical space fills up before logical does then the surplus logical space can be consumed by the addition of a single hidden file

Figure 6.0 Possible Allocation Schema

This is best explained using an example, and this is presented in figure 6.1.

OS Sector size	512 bytes	-> K = 9
Min Cluster Size	4	-> Mc = 4
Disk Sector size	choose 256 bytes	-> k = 8
C ∈ {8,4,8/3,2,8/5,8/6,8/7,1}	choose 2 by expt	-> C = 2
Overspill area	choose 10 Mbyte	-> R = 10
Physical disk size	choose 60 Mbyte	-> P = 60
Logical disk size	L = (60-10)*2	-> L = 100
Number of clusters relation	$C_o = C_i * 2/2$	-> $C_o = C_i$
Address Mapping	$A_o = 1/1 * A_i$	-> $A_o = A_i$

Thus 60 Mbyte drive becomes a **60 to 100 Mbyte** drive assuming compression of 2:1 for majority of clusters.

Note the address and cluster number mappings are one to one as this is a special case where disk sector size is reduced by a factor of two and compression is by a factor of two, therefore they cancel out.

6.1 Example Using the Allocation Schema

It would be preferred if the overspill area was not contiguous, but evenly distributed around the disk, ensuring less latency delays when access is required. Also it would be desirable for the sector size of these areas to be further reduced, to reduce wastage due to small overspills. A mechanism would have to be found for storage and retrieval of this information, but the task would be less critical regarding both optimum allocation and time. Only 20 per cent of requests approximately would involve access to this space. The sector tag of the last sector of the oversize cluster could have a pointer to the overspill placed within it. Also a protocol would have to be implemented to inform the drive when writing, and the machine when reading, that an overspill must be transferred, together with the compressed cluster.

This schema would require firmware changes to the disk drive together with relatively expensive hardware and firmware on an adapter card. However it does not require a rebuild of the drive, or modification to the operating system in any way, above the level of software interrupt. Methods which require either of these modifications invite trouble. With a large enough compression ratio, the cost of the extra electronics could easily be offset. The system ensures compatibility and fully informs the user of the range of possible capacities which depend on the data stored.

6.2 COMPRESSION TECHNIQUES REVIEWED

For compression of computer data, an algorithm must be lossless, uniquely decodable, and preferably optimum and instantaneous. A compression algorithm is optimum if the average code size is equal to the entropy of the code[1]. A code is instantaneous if, when looking at the incoming stream of bits one at a time, it is possible to identify the value of the input upon reaching the end of the code word. The two most used compression methods which fall into this category are described below.

6.2.0 HUFFMAN METHOD

Huffman coding[1,2,3,4] is a simple and yet elegant method of lossless data compression. It works by assuming that the distribution of symbols used is not uniform. For example, a data file containing a piece of prose written in English, would probably contain more space characters than any other, as every word would be terminated by one. Also statistics would indicate that the letter 'e' would be present far more often than the letter 'z'. Thus, such a data file would contain an uneven distribution of symbols, and this is true for almost all types of data stored in a computer system, from code files to accounting files.

Assuming that an uneven distribution of symbols exists, Huffman goes on to indicate formally how one may assign differing symbol lengths to symbols of differing probabilities of occurrence. The more frequent the symbol, the shorter the symbol should be.

The formal method begins with the acquisition of a symbol probability table. This can be compiled from statistical observations of the type of data being compressed, or generated by prescanning the file. The former will be faster but, being more general, will produce less than optimum results. The latter approach requires more time but ensures optimum compression.

An encoding tree is generated from this table. Such a tree is binary, with a symbol at each leaf. Construction is simple : first, one takes the two symbols of lowest probability and combines them as two leaf branches under a node ; the probabilities are then combined and this new node and probability returned to the list ; this is then repeated until a single tree exists with probability equal to one. An example of this is

included in figure 6.2, for a file containing merely the characters, 'a', 'b', 'c', 'd', and 'e'. The resultant compression ratio is poor due to the extremely small number of symbols, chosen to simplify the example.

Symbol Probability Table

'A' = 0.10
'B' = 0.15
'C' = 0.11
'D' = 0.06
'E' = 0.58

Tree Generation

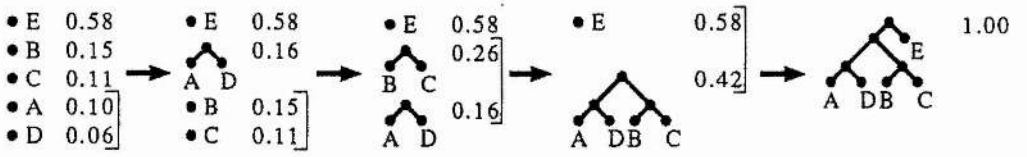


Figure 6.2 : Huffman Encoding Tree Generation

By assigning binary values '1' and '0' to the branches of the encoding tree, one can simply find any Huffman code for a symbol by reading from the leaf containing the symbol up to the root. As mentioned above, the symbols with highest probability have the shortest paths from root to leaf, thus they require the least bits to store. Decompression requires merely that one starts at the root and uses the Huffman code to find the route to the leaf, where the symbol can be found. This is illustrated in figure 6.3. In a serial stream, one does not require code delimiters as once a leaf is reached, a symbol has been found, and one returns to the root to decode the next. Dependent on the ordering of equal probabilities during the encoding tree's generation, and the binary allocation to the branches, the Huffman algorithm may generate multiple solutions which differ in length of code as well as choice of representatives of that code. All solutions are, however, optimum and will give the same compression.

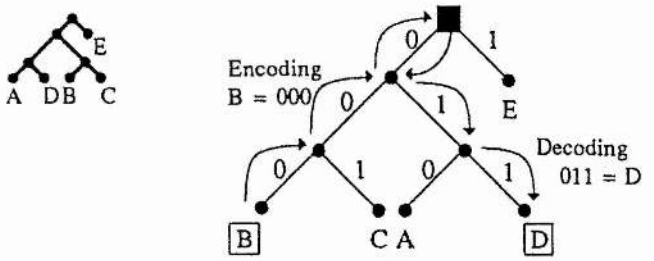


Figure 6.3 : Using the Encoding Tree

With a moderate loss in compression it is possible to generate the Huffman coding tree on the fly while reading and actively compressing. At the start compression is fairly poor, but as the tree 'matures', near optimum compression is reached.

6.2.1 LZW METHOD

The LZW method[2,3] is, like Huffman's method, simple in concept. It works by extending the alphabet, using additional characters to represent strings of characters which appear in the data. For example, if we consider compressing eight bit ASCII codes, one would use a nine bit code, leaving an extra 256 codes spare. These extra codes would then be determined by strings which appear in the input. These codes, together with the string they represent, must be stored in a table for referral during further compression, and of course decompression.

The algorithm works as follows : start with a null string and then read in a character from the input string and append it to the string. Next one checks the character table for the existence of that string and repeats the reading of characters and appending to the string until a match is not found in the table. Once this is the case we have generated a new entry for the table and it is inserted. To the output one sends the code from the table representing the last substring to be found existing in the table. The final character which made up the new string is then used as the basis for the next string.

```
while ~eof(infile) do  !repeat until file compressed
begin

  !keep going until not in dict or end of file
  !literal -stores current character code read from input file
  !code    -stores code from dict of string so far
  !         updated within 'in.dictionary' when found
  repeat
      literal := decode(read(infile))
  while in.dictionary(oldcode,newcode) and ~eof(infile)

  !output 'code' = the last identified string in dict
  output.code

  !add an entry into dictionary - code + literal pair
  if ~eof(infile) do add.entry(code,literal)

  !last character becomes basis for next string
  code := literal

end
output.code  !last character must be output
```

Figure 6.4 : S-Algol Code Routine for LZW Compression

Figure 6.4 illustrates this algorithm, presenting a highly documented section of the main routine. The full source code implementing LZW, and providing visual dictionary output as in figure 6.5, appears in the appendices.

Figure 6.5 shows an example of LZW compression. The input stream 'This is a' is compressed. The output stream matches the input whilst the dictionary builds up, until the second tuple 'is' appears in the input stream. This is found in the dictionary and can therefore be represented by a single nine bit code.

Input	**Dictionary entry-string**		**Output**
T			
h	256	T+h	T
i	257	h+i	h
s	258	i+s	i
<sp>	259	s+<sp>	s
i	260	<sp>+i	<sp>
s			
<sp>	261	258+<sp>	258 = i+s *!Compression here*
a	262	<sp>+a	<sp>
			a

Figure 6.5 : Example of LZW Compression Algorithm

This algorithm requires significant amounts of memory. The larger the dictionary, the greater the compression, but the larger the memory required. Also, however large the dictionary is made, it will fill up eventually. Three methods of dealing with this are possible. One way is to freeze the dictionary and create no new entries which will result in poor compression if the type of data changes from one end of the stream to the other. Alternatively the dictionary could be cleared all together, and building begun again, which would reduce compression whilst the dictionary is reasonable empty, but reduce the effect of data change which occurs with a frozen dictionary. One other approach is possible : one could keep track of the frequency of use of an entry, and when the dictionary fills, start replacing the least used entries with new ones. Although more complex, if anything there would be an improvement in compression, not a reduction.

6.3 IMPLEMENTATION

6.3.0 SOFTWARE

Both Huffman and LZW methods were simulated in software, the code appearing in the appendices. This was primarily to give an indication of the probable compression ratios under different conditions. Crude experiments indicated that both Huffman encoding trees and LZW dictionaries prove to be highly data specific, and general versions fail to produce significant compression.

Huffman encoding trees were generated for different types of files : ASCII, database, program source and binary code. Each type of file was compressed using all generated trees. The binary file expanded when any tree but its own was used, as the other trees expected seven bit ASCII data, with a few possible irregularities. However, all files compressed when using the binary file's encoding tree to a reduced extent over their

generated optimum tree. It was, albeit crudely, estimated that a computer disk with approximately a fifty-fifty split of ASCII and binary information could expect savings of 25 per cent if the single binary file encoding tree was used. Even assuming that a better general encoding tree could be found, compression ratios could not be increased dramatically, and compression to this degree would not be worthwhile.

Using a reasonable size 8K byte entry dictionary, various dictionaries were generated, and used to compress other files, and file types. The dictionary proved to be even more data specific than the Huffman tree which is to be expected, as only a small, finite number of possible strings can be given entries in the dictionary and these are likely to appear frequently within only the given file, not in general.

Also compression was attempted using nibbles, bytes and words as input code lengths. The shorter the code length the more the compression ratio depended on the distribution of datum, and therefore the more the chance of a general compression being found. The larger the code length the more the compression relies on the non-existence or near non-existence of certain codes, for average to small size files. General solutions would not be appropriate as different codes would be absent for different files. Thus, small code lengths suffer as they cannot be compressed significantly. Large code lengths can be better compressed but require specific encoding trees or dictionaries.

From a combination of crude experimentation, and information gathered from related literature, it appeared that dynamic generation of an encoding tree or dictionary, for a given file, was the only route to reasonable compression. As this could not be done in software without noticeable degradation in a computer systems performance, a hardware implementation was required.

6.3.1 HARDWARE

Incorporating compression hardware, on the adapter card, would have certain advantages. If the compression occurred before transfer, then the time saved on transfer due to the compression would offset the time taken to compress. Also, a certain amount of parallelism would be possible, with the Z80 supervising the compression, whilst the IBM continued with other operations.

The IC105 data compression coprocessor from InfoChip Systems[5,6,7] was chosen as this chip guarantees error free, lossless compression using proven proprietary algorithms developed by information theorist, Claude Shannon.

This chip is capable of continuous compression at 1.5 Mbytes per second, and decompression at three to four Mbytes per second. Importantly, it guarantees that expansion is not possible because, if the algorithm would cause expansion, the data stream is passed straight through without 'compression'.

It was necessary to discover what compression ratios could be obtained, given the short input streams of one MS-DOS cluster which is required for implementation of the schema mentioned above. To facilitate this, a minimal hardware system was built which would allow definable compression and decompression, together with the ability to view and alter the dictionary memory it requires. Infochip claim a possible 2 to 8 times compression.

Figure 6.6 contains a block diagram of the prototype compression circuit. It contains a 32K byte RAM for the dictionary, a 4K byte input buffer, and a 4k byte output buffer. All the RAM had to be accessable by both the IC105 and the IBM and thus electronics effecting dual port access was included. The IC105 has a complex, non-standard interface for access to the internal configuration registers and status registers. This facilitates interleaved read and write requests, but also significantly complicates interfacing.

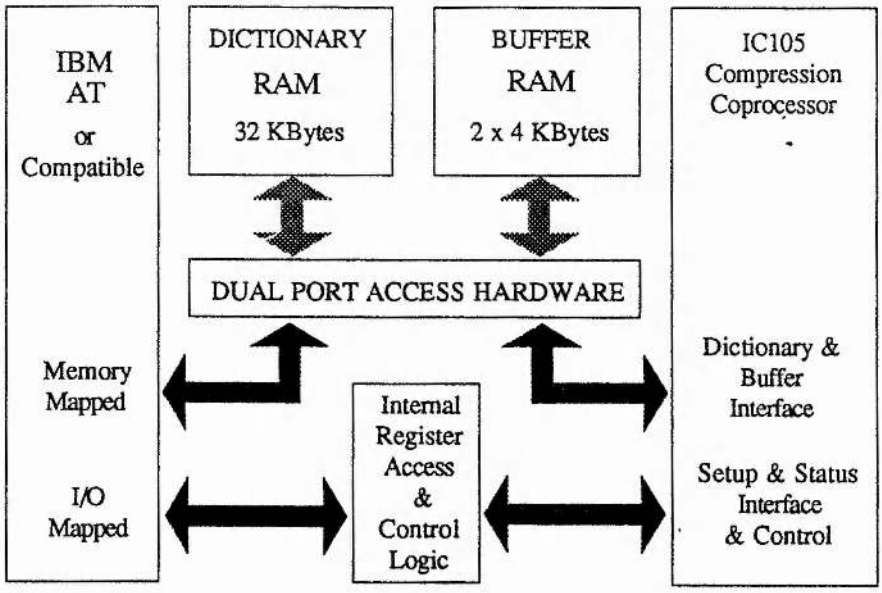


Figure 6.6 : Compression Circuit Block Diagram

Data is only valid from a read request after a fixed number of clock cycles after the request is made, and only for one cycle. A request is made by generating a single pulse, one clock cycle in length on the CMD line, a clock cycle being only 50 nanoseconds for this implementation.

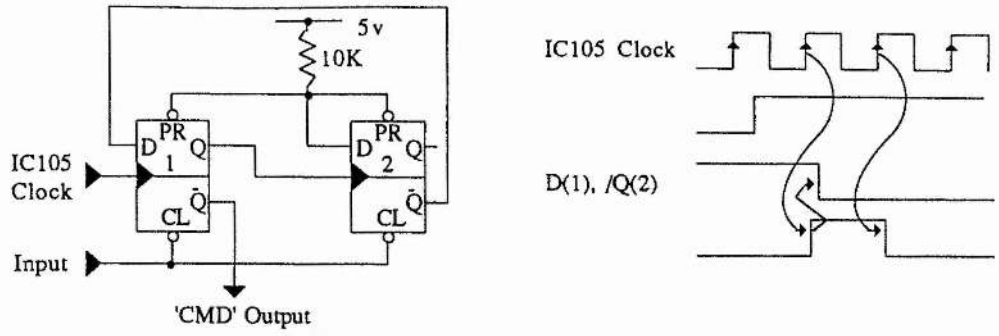


Figure 6.7 : CMD Line Circuit and Timing Diagram

Figure 6.7 indicates how the CMD pulse was achieved. The input is port mapped to the IBM : a write by the IBM to the port causes an active high pulse, many IC105 clock cycles in length. This relinquishes the permanent 'clear' hold on the flip-flops. A logic '1' is propagated through the first flip flop on the first subsequent rising edge of the IC105 clock. This in turn causes a rising edge on the clock input of the second flip flop, which propagates a logic '0' to the '/Q' output and therefore on the 'D' input of the first flip flop. This logic '0' is then passed through on the next rising edge of the IC105 clock. Thus a single pulse, one IC105 clock cycle in length is generated. This is not repeated until the flip flops are cleared, which will only occur after the end of the input pulse.

Writing to the internal registers of the IC105 requires no additional electronics, over and above the port mapping into the IBM port space. Both the address and data are valid when the IC105 samples them during the IBM I/O write cycle.

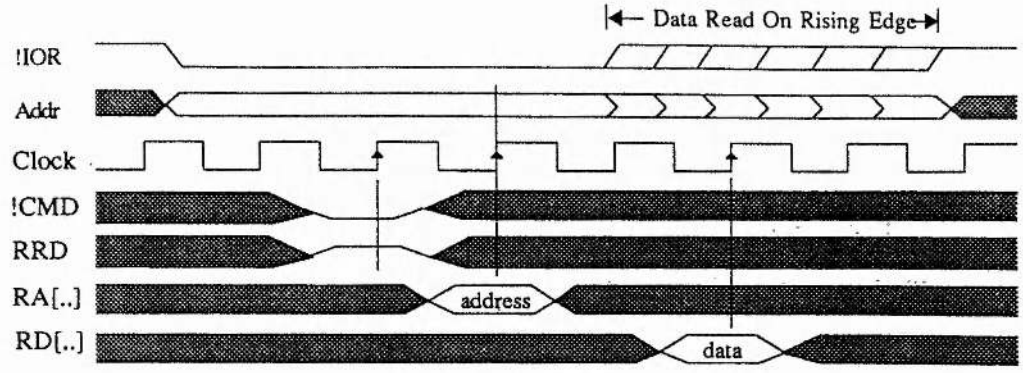


Figure 6.8 : Read Internal Register of IC105 Timing Diagram

Figure 6.8 contains the timing diagram for a status register read, and the timing for a read I/O cycle for an IBM (top two lines). When considering the I/O read cycle of the IBM, one can see that attempting to read bytes from within the IC105 is more complex. Again, the address lines are valid when the IC105 samples them, but the IC105 places the data on the bus for only one clock cycle and so the data must be latched. Figure 6.9 provides both circuit and timing diagrams for this.

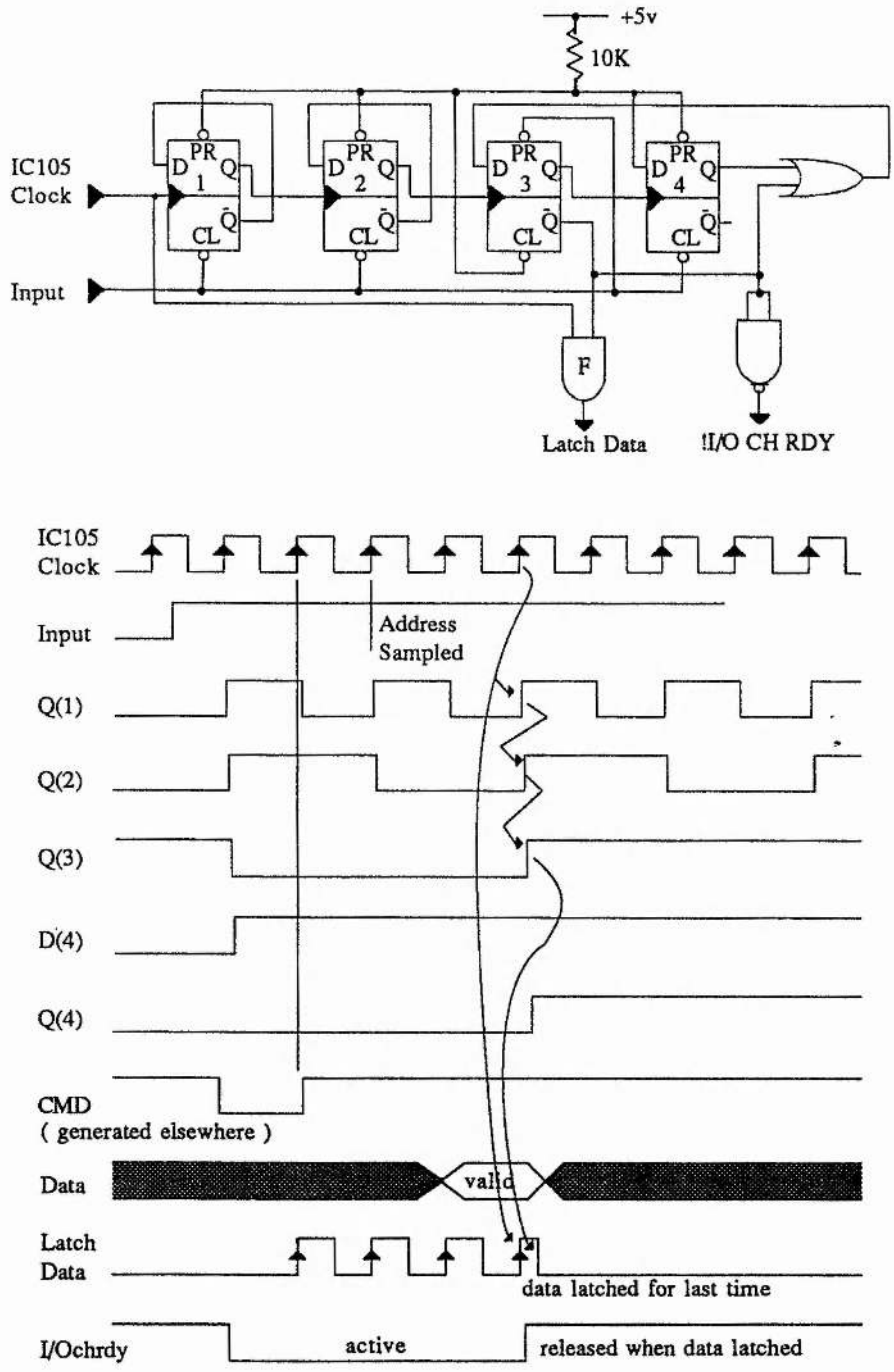


Figure 6.9 : IC105 Register Read Circuit and Timing Diagrams

Flip flops 1,2 and 3 divide down the system clock to provide a rising edge at the required point when the data from the IC105 is valid. Flip flop 4 ensures that once the output, 'Q', of flip flop 3 is high, it remains so until the input pulse is complete, ensuring that there is no possible repeat latching. The line I/O CH RDY is activated at the start of the input pulse until the data is latched, this extends the read cycle of the IBM to ensure the data has been latched by the time the IBM samples it. The AND gate is used to minimise the propagation delay between the rising edge of the IC105 clock and the latching of the data. Data is latched on every rising edge from the start of the input pulse until the output, 'Q', of flip flop 3 returns high. Thus the last data to be latched occurs at the desired timing point with only the one AND gate propagation delay from the rising edge of the clock.

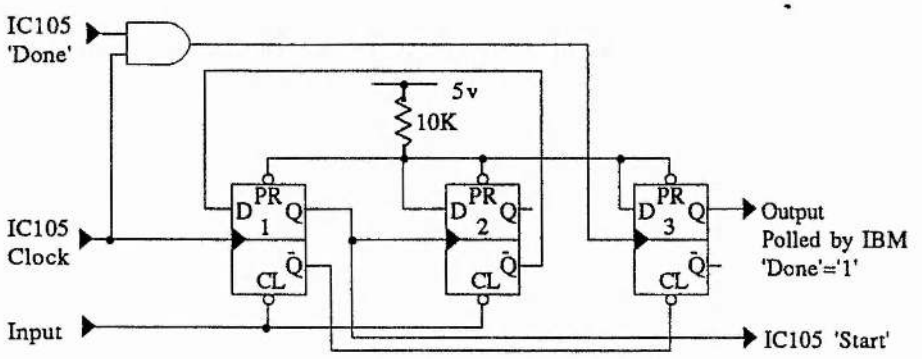


Figure 6.10 : IC105 'START' and 'DONE' Circuit

Two other pins on the IC105 required attention : 'START' and 'DONE'. The circuit required is outlined in figure 6.10. Once a compression or decompression has been configured, using the internal registers, the task is initiated by issuing a pulse, a single IC105 clock pulse in length, to the 'START' input. This is effected using exactly the

same single pulse generator used for the 'CMD' line. A third flip flop is required which is reset on 'START' pulse generation, and set when the 'DONE' line goes active, which is sampled during the rising edge of the IC105 clock. The IBM can then poll this flip flop to determine when an operation is complete.

With the electronics above, one can communicate with the IC105, and configure all compression and decompression sequences. All that is further required is a memory interface which will allow both the IBM and the IC105 access to the buffer and dictionary RAMs. Between a 'DONE' pulse and a subsequent 'START' pulse, the IC105 places all address and data lines tristate and, using this knowledge, one can simulate dual port access to the RAMs by use of tristate buffering of the IBM address and data buses, together with certain logic functions to ensure only single access to any given RAM at any given time.

A single 8K byte RAM is used for the buffer RAM. The upper address bit is fed by either the 'BRD' pin of the IC105 or the 'MEMR' of the IBM, dependent on which requires access, so that a read request accesses the lower 4K bytes of the RAM, and a write request accesses the upper 4K bytes.

```
                    IC105_BRD | 1                20 | VCC
                       A19_18 | 2                19 | A16
                          A17 | 3     GAL        18 | BRAM_WE
                          A12 | 4    16v8        17 | BRAM_OE
                          A15 | 5                16 | BRAM_CS
                     IBM_MEMR | 6                15 | DRAM_WE
                     IBM_MEMW | 7                14 | DRAM_OE
                    IC105_DWT | 8                13 | BRAM_A12
                    IC105_DRD | 9                12 | DRAM_CS
                          GND | 10               11 | IC105_BWT
```

```
!BRAM_CS    = A19_A18 & !A17 & A16 &  A15 & (!IBM_MEMR | !IBM_MEMW)

BRAM_A12    =   (A19_A18 & !A17 & A16 & A15) & A12)
              | (A19_A18 & !A17 & A16 & A15) & IC105_BRD)

BRAM_OE     = !(A19_A18 & !A17 & A16 & A15 & !IBM_MEMR) & IC105_BRD

BRAM_WE     = !(A19_A18 & !A17 & A16 & A15 & !IBM_MEMW) & IC105_BWT

!DRAM_CS    = A19_A18 & !A17 & A16 & !A15 & (!IBM_MEMR | !IBM_MEMW)

DRAM_OE     = !(A19_A18 & !A17 & A16 & !A15 & !IBM_MEMR) & IC105_DRD

DRAM_WE     = !(A19_A18 & !A17 & A16 & !A15 & !IBM_MEMW) & IC105_DWT
```

Figure 6.11 : Boolean Logic Required to Effect Dual Port Access to RAMs

The IBM may access either RAM when the IC105 is not active, as both the IC105 address bus and data bus are tristate. Any attempt by the IBM to access a RAM causes the required address buffers to be enabled, together with a transceiver on the data bus, with its direction set corresponding to the nature of the access. Also required is decoding logic to activate the output enable, 'OE', and write enable, 'WE', of the RAM chip. If either the IC105 or the IBM require activation of any of these lines, then activation takes place. As the IBM only attempts access when the IC105 is inactive, both will never make a simultaneous request. The logic required appears in figure 6.11.

The full circuit diagrams for the prototype compression board appear in the appendices. The result is a hardware compressor mapped into the IBM's port and address space, as defined in figure 6.12.

```
Address Space

Dictionary RAM          D000:0000 -> 7FFF
Buffer RAM        I/P   D800:0000 -> 0FFF
Buffer RAM        O/P   D800:1000 -> 1FFF


Port Space

IC105 'RESET'     W     Port 308H
IC105 'START'     W     Port 30CH
IC105 'DONE'      R     Port 308H (Bit0=1 if Done)
IC105 internal    R/W   Port 300H -> 307H
```

Figure 6.12 : Compression Hardware IBM Mapping

Compression can be achieved simply, using the MS-DOS utility DEBUG.COM to issue the required port and memory instructions, illustrated in figure 6.13.

```
[1] Configure the IC105 : Send the following information :
    Port 300H          C8H          'compress'
    Port 301H          77H          RAM wait state info
    Port 302-304H      nnnnnnH      Number of bytes to compress
    Port 305-306H      aaaaH        Start Address

[2] Fill Buffer with Data to be compressed : D800:0000 - 0FFF

[3] Clear Dictionary : D000:0000 - 7FFF = 00H

[4] Issue 'START' command : write to port 30CH

[5] Poll 'DONE' flag for finish condition : read port 308H bit0

[6] Once 'DONE' true :
    Port 300H          Status -> bit 7 indicates error
    Port 301-303H      Input Count
    Port 304-306H      Output Count
```

Figure 6.13 : Required Sequence for Compression

By following the simple steps in figure 6.13 one can compress any number of bytes and investigate the compression ratios possible. Once compression is complete, one can also view the contents of the output buffer, and the dictionary.

Preliminary investigation, using this hardware indicated 3:1 compression of ASCII data and 2:1 compression of binary code could regularly be achieved when compressing whole files. However, compression of a file cluster by cluster reduced the compression and was more erratic in nature. ASCII files could typically be compressed by 2:1, whereas binary code was reduced by only 7:5. Not only was this compression poor, but compression was very uneven, as some individual clusters compress extremely well, others very little.

6.4 CONCLUSION

With large cluster sizes, used by MS-DOS for large partitions, a general compression ratio of 2:1 could be possible. This would then fit into the schema mentioned in this work, providing a static allocation method for indeterminate compression.

A better solution, and one that would work for all partitions, might be to attempt to generate a fixed dictionary or partial dictionary, which would be preloaded before compression. The reason that the compression ratio is poor for a cluster is that, for LZW, compression improves as the dictionary matures. A mature dictionary might match a very large string to a single character, whereas a relatively new dictionary contains only strings of two to three characters. If a dictionary could be generated which was fairly general, then there would be an immediate improvement in compression. If however, the dictionary proved to be less than general, compression might even be replaced by expansion.

A partial dictionary would leave entries available for allocation when non-general strings are encountered, giving the improvement desired, whilst accepting the differences between datum in the system.

To include compression within the virus schema, one would require that only user areas were compressed, and that overspill areas were kept distinct for each partition. If dictionary preloading was implemented, then one could have one dictionary for each partition, which could be specific for that partition. As only one partition can be active at any one time, dictionary switching would not be required. Housekeeping could be handled by the supervisor processor, keeping the implementation transparent to the operating system.

Compression could prove to be the way forward with regards to large volume disk requirements, and this schema would offer a simple and non-intrusive way of implementing such a system.

6.5 REFERENCES

[1] **Huffman D.** (1952) A Method for the Construction of Minimum-Redundancy Codes. *Proceedings IRE,* **40**, 1098-1101

[2] **Apiki S.** (1991) Lossless Data Compression. *BYTE,* **March Issue**, 309-314,386-387

[3] **Bacon F.** (1988) How to Quadruple Dial-up Communications Efficiency. *Mini-Micro Systems,* **February Issue,** 77-81

[4] **Hiss S. et al.** (1985) Text Compression Using Huffman Codes. *Proceedings 17th South Eastern Symposium on System Theory,* 273-277

[5] **Vaughan-Nichols S.** (1990) Getting Your Byte's Worth. *BYTE,* **November Issue,** 331-336

[6] **InfoChip Systems Inc.** (1990) IC-105 Compression / Decompression Coprocessor. *Data Sheet.*

[7] **InfoChip Systems Inc.** (1990) IC-105 Compression Coprocessor. *Hardware Design Guide.*

CHAPTER 7

CONCLUSION

The use of the word 'virus' to depict this highly virulent form of hostile program is very apt. Like their biological counterparts, complete eradication is virtually impossible. Every computer virus that has come into existence, and has been injected into the public domain, still exists. It may appear extinct, only to appear months or even years later, re-introduced from an old backup floppy, or rarely used machine or disk. Only when machines become resistant to them will they cease to be a threat to the user.

They enter the system and systematically take control, corrupt and destroy. The more time that passes, the greater the threat they pose : the more the infection spreads ; the greater the number of new viruses ; and the more complex the new viruses become. Viruses cannot be considered individually as they have been in the past. Anti-virus methods which rely on knowledge of the virus will serve little purpose, as the rate of increase of 'new virus' introduction increases and they become more and more armoured. Anti-virus programs will fail to keep up with virus growth and will be subject to attack from the viruses themselves. The odds are in favour of the virus writer: he has only to find one loophole to destroy a system. An anti-virus program must be general and provide complete protection, which is a goal which may prove impossible to attain.

A system must be found which guarantees protection, reduces the virus spread, and detects, not just the viruses of today, but all future viruses. Only then will the user be safe from virus infection, and the incentive be lost to virus writers. The schema presented in this work goes a long way to providing just that level of protection.

The boot track, the partition information, the operating system and its associated utilities, all partition boot sectors, directories, and FAT tables are guaranteed protected against all virus attack. This ensures that infection by 'Boot Sector' viruses is impossible, as is infection of the 'COMMAND.COM' which is a common attack point as infection of this file ensures that the virus will be loaded immediately on boot.

Applications that are not self-modifying may also be protected by placing them in a read only partition. Virus spread across a partition boundary also becomes impossible. Thus, a virus introduced into what we define as an active, user partition, cannot spread into any other partition in the system. The result is guaranteed virus containment. The only source of infection possible is through removable media, direct into the active partition.

Various floppy schemes have been presented which reduce the threat of infection via floppy disks. They vary in the protection they provide from merely reducing the likelihood to complete protection via auto-encryption methods.

With a user shell in operation, one can prevent access to certain sensitive partitions and prevent use of the floppy drive to less trusted users, either for removal of classified data or for introduction of viruses.

With all this protection, a business user can be certain that system integrity is kept, and that viruses carelessly introduced into a 'games' partition, for example, cannot infect across into more sensitive areas. Viruses are contained, and any attempt to violate the protection schema will raise an alarm, giving early warnings of a virus' existence.

A fully operational system including the virus schema, a security shell, a floppy protection method, and even possibly a compression facility for individual partitions, would provide levels of protection and versatility to a business system, which cannot be replicated by any combination of currently available software. Using this system will limit possible viral damage to the point where little to no time or money is lost, as viruses will be detected early and will be severely contained. No other system can make such guarantees if the computer virus is to be regarded, as it rightly should, as a general and not a specific phenomenon.

APPENDICES

APPENDIX A

VIRUS SUPERVISOR VERSION 1 -

CIRCUIT DIAGRAM

APPENDIX B

VIRUS SUPERVISOR VERSION 2 -

CIRCUIT DIAGRAM & GAL DESCRIPTIONS

APPENDIX C

VIRUS SUPERVISOR SOFTWARE -

IBM <-> SCSI LOGICAL ADDRESS CONVERSION

PERMISSION LIST ACQUISITION AND DISPLAY

APPENDIX D

IC105 COMPRESSION BOARD -

CIRCUIT DIAGRAM & GAL DESCRIPTIONS

APPENDIX E

COMPRESSION SOFTWARE -

HUFFMAN COMPRESSION

LZW COMPRESSION

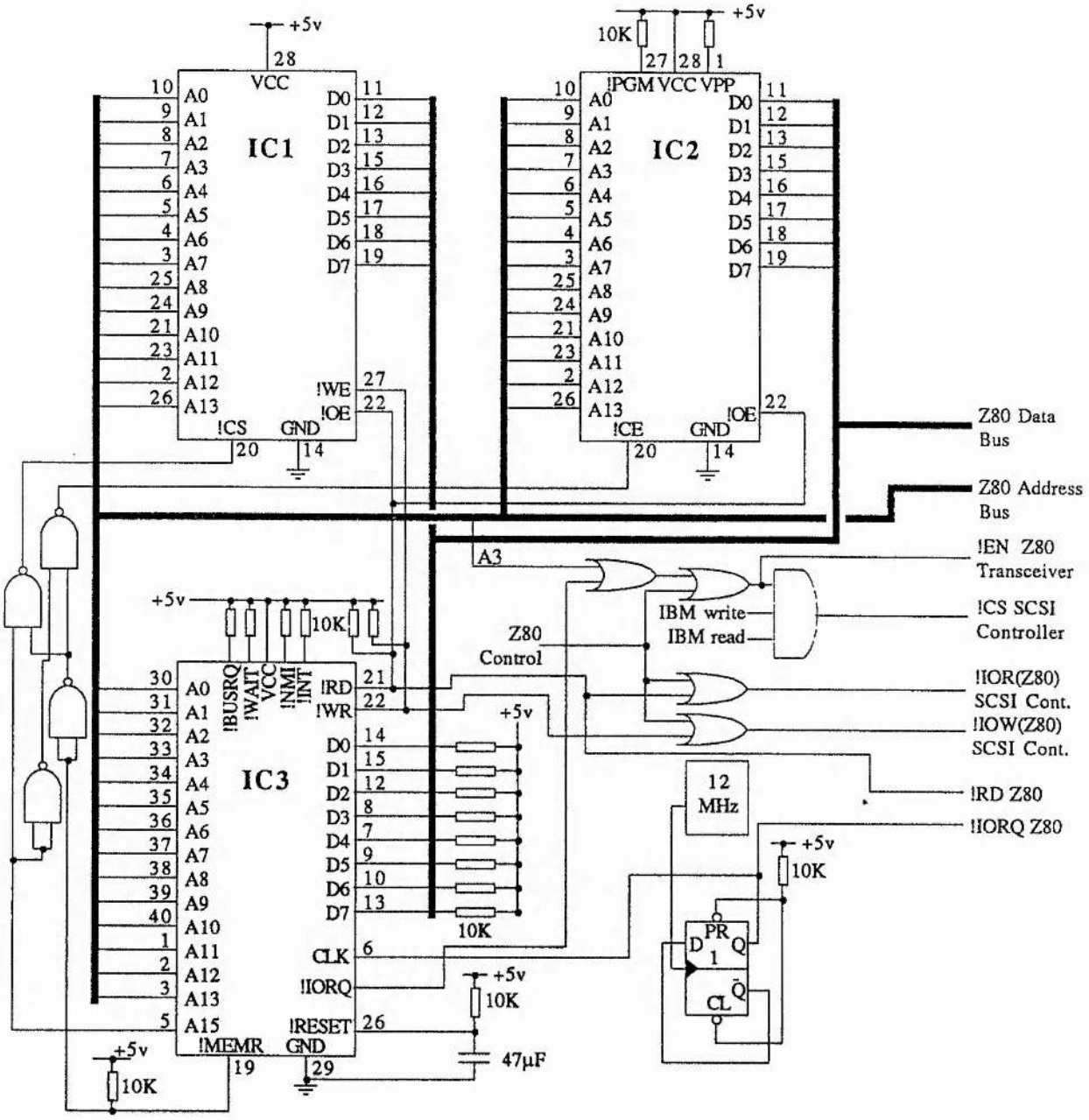
IC105 COMPRESSION

APPENDIX A

VIRUS SUPERVISOR VERSION 1

CIRCUIT DIAGRAM

Virus Supervisor Circuit
Version 1 - Page 1

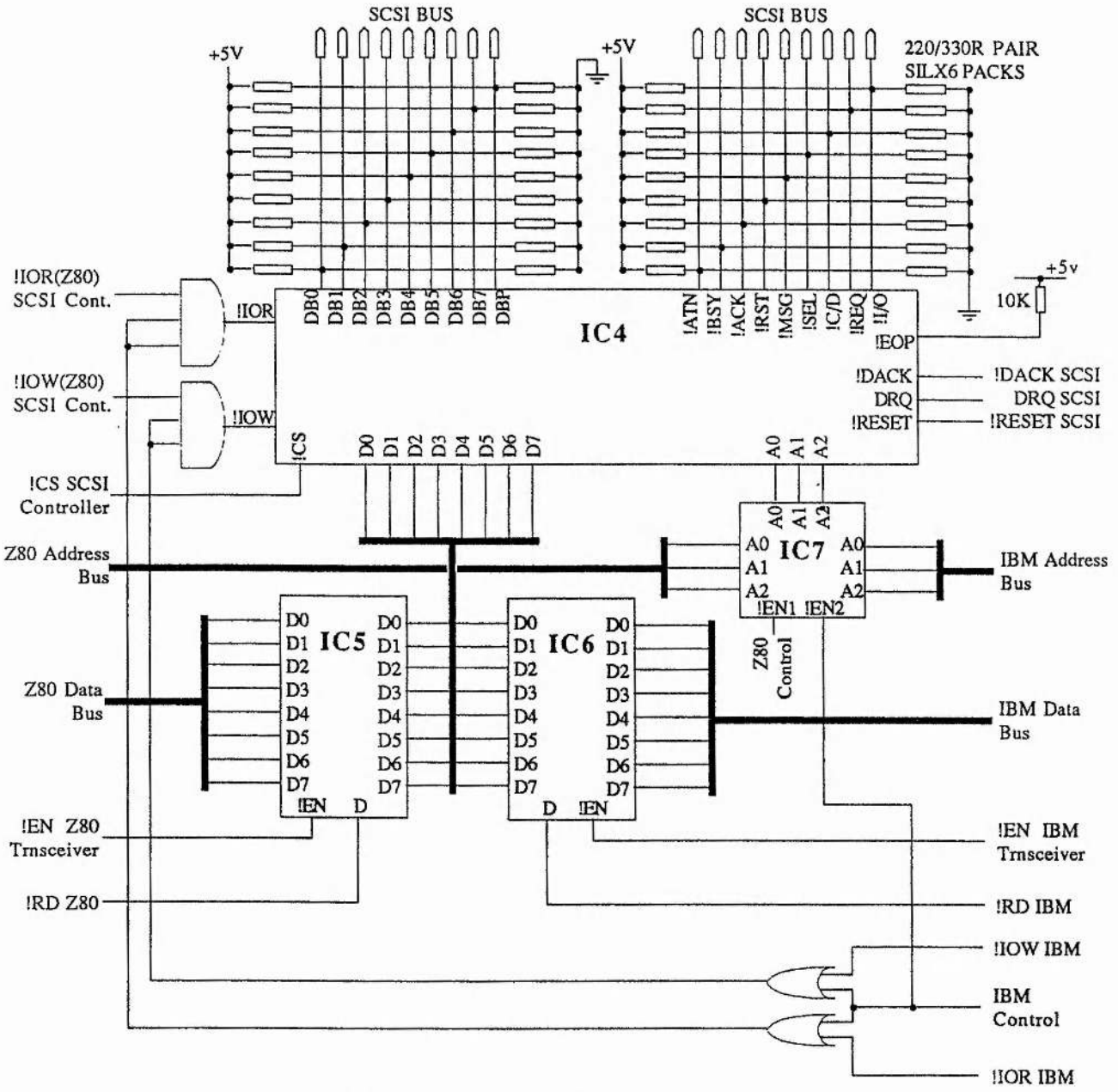


IC1 6264 8Kx8 RAM

IC2 2764 8Kx8 EPROM

IC3 Zilog Z80B @ 6Mhz

Virus Supervisor Circuit
Version 1 - Page 2



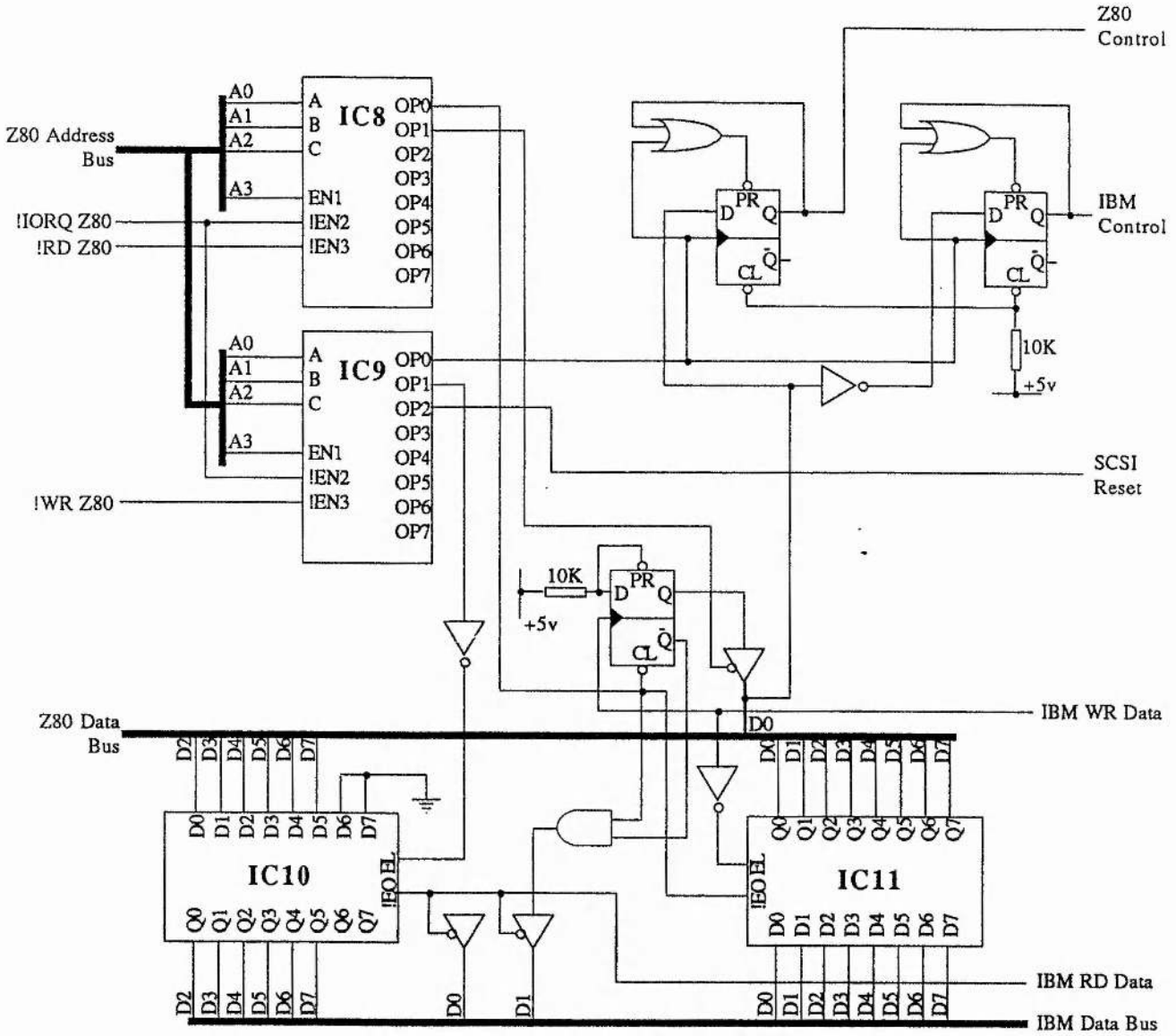
IC4 NCR 5380 SCSI Controller

IC5 74LS245 Tranceiver

IC6 74LS245 Tranceiver

IC7 74LS244 Octal Buffer

Virus Supervisor Circuit
Version 1 - Page 3



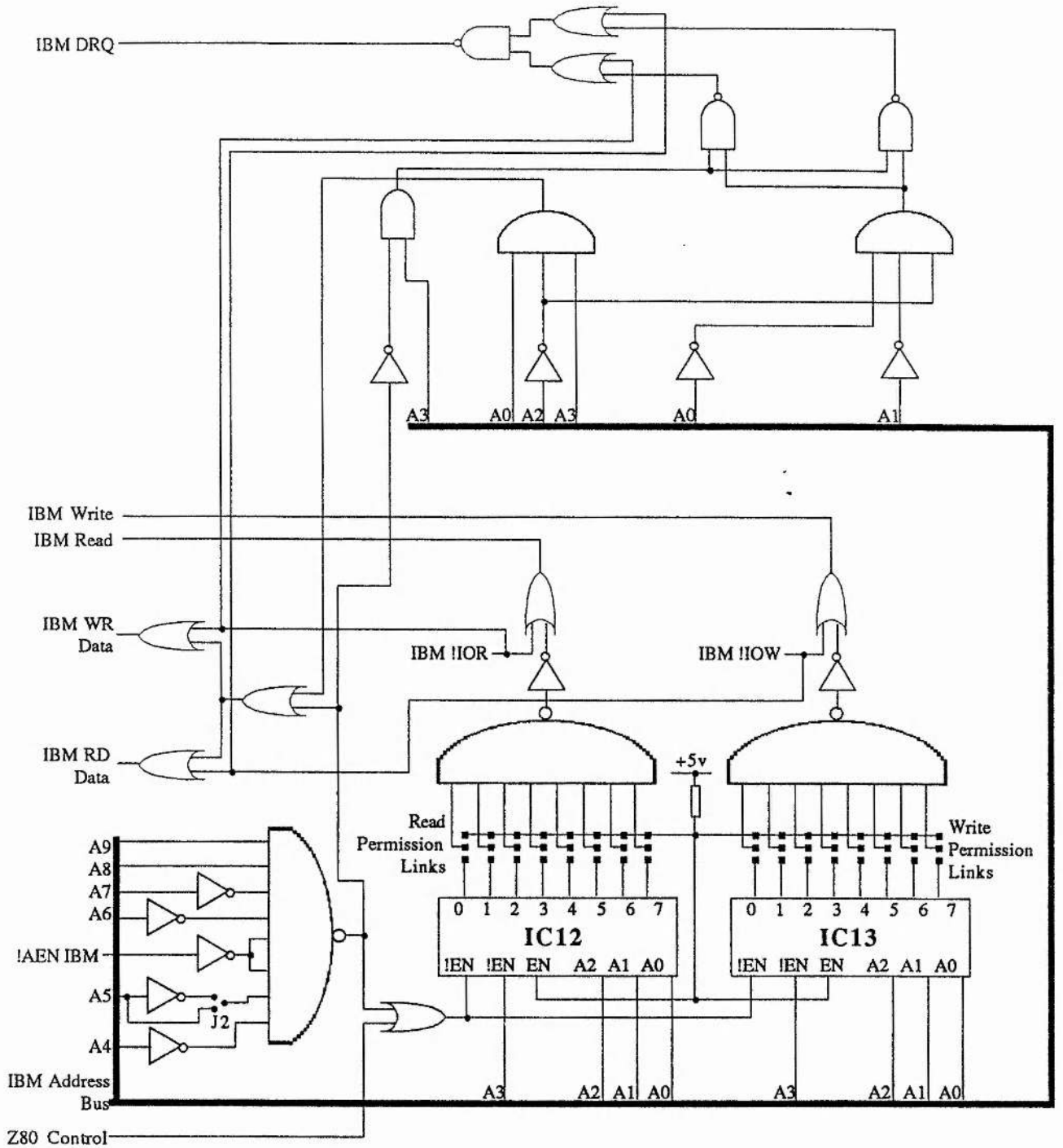
IC8 74LS138 3 to 8 Decoder

IC9 74LS138 3 to 8 Decoder

IC10 74LS373 Octal Latch

IC11 74LS373 Octal Latch

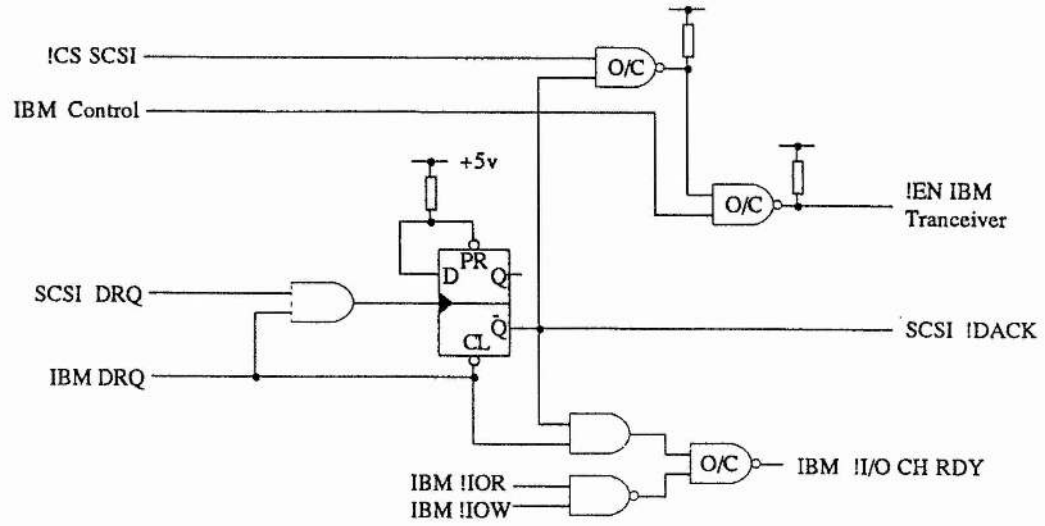
Virus Supervisor Circuit
Version 1 - Page 4

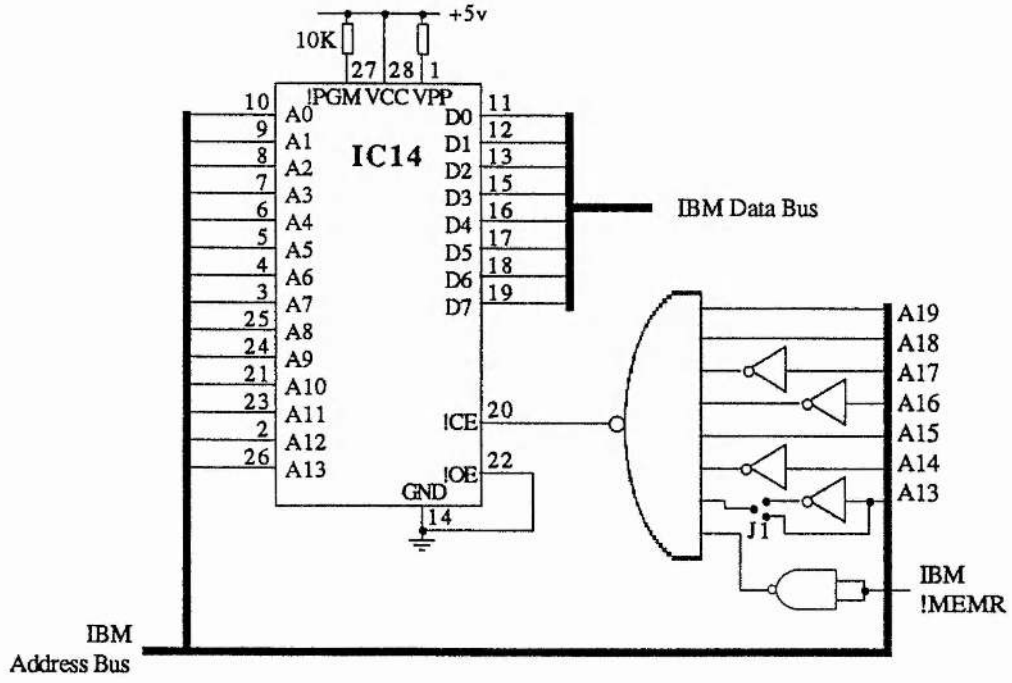


IC12 74LS138 3 TO 8 Decoder

IC13 74LS138 3 TO 8 Decoder

Virus Supervisor Circuit
Version 1 - Page 5





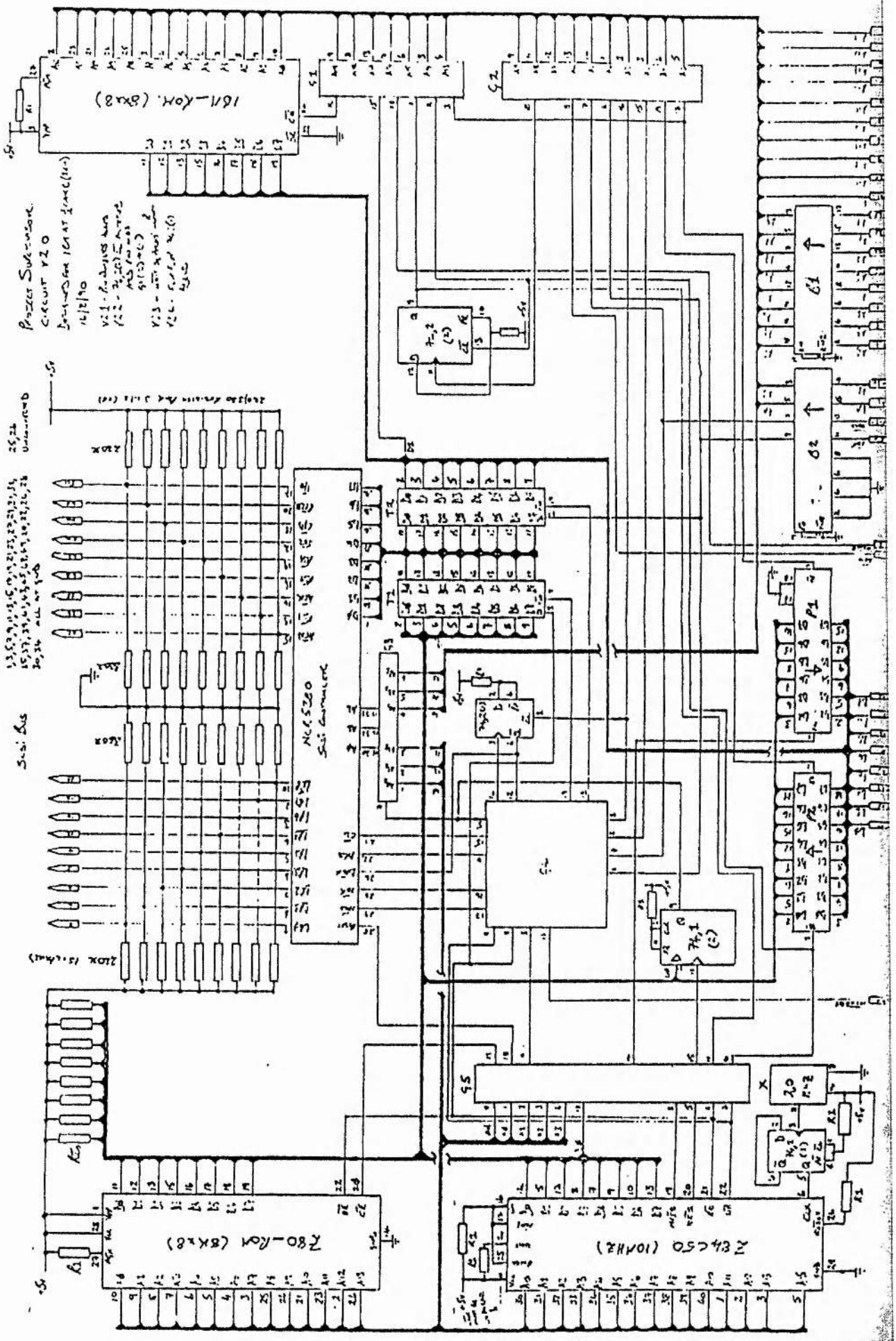
IC14 2764 8Kx8 EPROM

APPENDIX B

VIRUS SUPERVISOR VERSION 2

CIRCUIT DIAGRAM & GAL DESCRIPTIONS

SUPERVISOR PROJECT CIRCUIT
D. S. S. Robb;

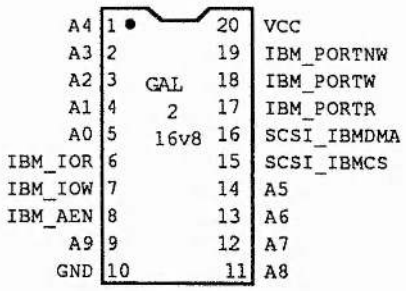


```
DEVICE 16V8;
TITLE  SUPERVISOR  PROJECT  GAL1
NAME  D.  S.  S.  Robb;
SIGNATURE C2G1;
```

STATUS_BIT	1		20	VCC
Z80_READ_DATA	2		19	unused
IBM_PORTR	3	GAL	18	unused
A13	4	1	17	unused
A14	5	16v8	16	IBM_MEMR
A15	6		15	IBM_D0
A16	7		14	unused
A18	8		13	A17
A19	9		12	unused
GND	10		11	MEMR

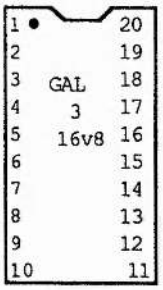
```
!IBM_MEMR     = A19 & A18 & !A17 & !A16 & A15 & !A14 & !A13 & !MEMR;

IBM_D0.OE     = !IBM_PORTR;
IBM_D0        = !STATUS_BIT;
```

```
DEVICE 16v8;
TITLE  SUPERVISOR  PROJECT  GAL2
NAME  D.  S.  S.  ROBB;
SIGNATURE C2G2;
```



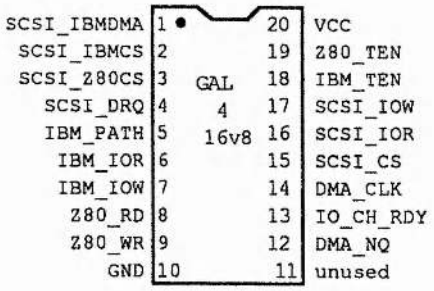

```
SCSI_IBMCS   = ( (!IBM_IOR & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & !A3
                   & !A2 & !A1 &  A0) /* READ P1  */
                 |(!IBM_IOR & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & !A3
                   &  A2 & !A1 &  A0) /* READ P5  */
                 |(!IBM_IOW & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & !A3
                   & !A2 & !A1 &  A0) /* WRITE P1 */
                 |(!IBM_IOW & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & !A3
                   &  A2 & !A1 &  A0) /* WRITE P5 */
                 |(!IBM_IOW & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & !A3
                   &  A2 &  A1 &  A0) /* WRITE P7 */
               );

SCSI_IBMDMA  = (  (!IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & A3 & !A2 & !A1 & !A0)
                & (!IBM_IOR | !IBM_IOW));

!IBM_PORTR   = (!IBM_IOR & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & A3 & !A2 & !A1 & A0);

!IBM_PORTW   = (!IBM_IOW & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & A3 & !A2 & !A1 & A0);
IBM_PORTNW   = (!IBM_IOW & !IBM_AEN & A9 & A8 & !A7 & !A6 & A5 & !A4 & A3 & !A2 & !A1 & A0);
```

```
DEVICE  16v8;
TITLE  SUPERVISOR  PROJECT  GAL3
NAME  D.  S.  S.  ROBB;
SIGNATURE C2G3;
```



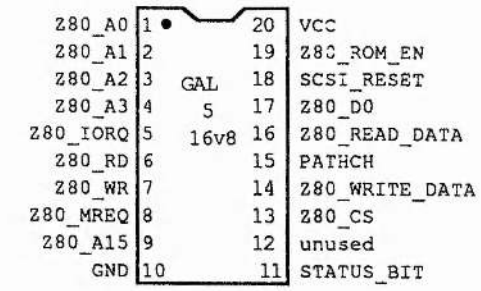

```
SCSI_A0     =   ( (!IBM_PATH & Z80_A0) |( IBM_PATH & IBM_A0) );

SCSI_A1     =   ( (!IBM_PATH & Z80_A1) |( IBM_PATH & IBM_A1) );

SCSI_A2     =   ( (!IBM_PATH & Z80_A2) |( IBM_PATH & IBM_A2) );
```

```
DEVICE 16v8;
TITLE  SUPERVISOR  PROJECT  GAL4
     NAME  D.  S.  S.  ROBB;
       SIGNATURE C2G4;
```




```
!IO_CH_RDY   =  (DMA_NQ & SCSI_IBMDMA);

DMA_CLK      =  (SCSI_DRQ & SCSI_IBMDMA);

!SCSI_CS     =  ((SCSI_Z80CS & !IBM_PATH)|(SCSI_IBMCS &  IBM_PATH));

!SCSI_IOR    =  ((!Z80_RD  & !IBM_PATH)|(!IBM_IOR &  IBM_PATH));

!SCSI_IOW    =  ((!Z80_WR  & !IBM_PATH)|(!IBM_IOW &  IBM_PATH));

!IBM_TEN     =  ((SCSI_IBMCS  & IBM_PATH)|(!DMA_NQ & IBM_PATH));

!Z80_TEN     =  (SCSI_Z80CS & !IBM_PATH);
```

```
DEVICE 16v8;
TITLE  SUPERVISOR  PROJECT  GAL5
NAME  D.  S.  S.  ROBB;
SIGNATURE C2G5;
```




```
Z80_WRITE_DATA  =  !Z80_WR & !Z80_IORQ & Z80_A3 & !Z80_A2 & !Z80_A1    &  Z80_A0;

!PATHCH         =  !Z80_WR & !Z80_IORQ & Z80_A3 & !Z80_A2 & !Z80_A1    & !Z80_A0;

!SCSI_RESET     =  !Z80_WR & !Z80_IORQ & Z80_A3 & !Z80_A2 &  Z80_A1    & !Z80_A0;

!Z80_READ_DATA  =  !Z80_RD & !Z80_IORQ & Z80_A3 & !Z80_A2 & !Z80_A1    & !Z80_A0;

Z80_D0.OE       =  !Z80_RD & !Z80_IORQ & Z80_A3 & !Z80_A2 & !Z80_A1    &  Z80_A0;
Z80_D0          =  STATUS_BIT;

!Z80_ROM_EN     =  !Z80_MREQ & !Z80_A15;

Z80_CS          =  !Z80_A3 & !Z80_IORQ;
```

APPENDIX C

VIRUS SUPERVISOR SOFTWARE

IBM <-> SCSI LOGICAL ADDRESS CONVERSION

PERMISSION LIST ACQUISITION AND DISPLAY

```
/*****************************************************/
/*  PROGRAM TO CONVERT IBM CYL, HEAD AND SECTOR TO   */
/*  SCSI LOGICAL ADDRESS AND REPORT IN DEC AND HEX   */
/*                                                   */
/*  Copyright David S. S. Robb 1990                  */
/*****************************************************/


program ibmtolog;

var         i,N2,N3,number,Cyl,Temp,RCX,RDX : longint;
            Head,Sec,Cyl_upper,Cyl_lower : longint;
            h1,h2,h3,h4,h5,h6,t : longint;
            number1,number2,number3,number4 : longint;
            N,hh1,hh2,hh3,hh4 : char;

const       Mhd=4;
            Ms =17;

function c(h : longint):char;              /*dec 0-16 to char*/
begin
  if h>9 then c := chr((h-10)+97)          /*number*/
         else c := chr(h+48);              /*letter*/
end;

procedure c2(h : longint);                 /*takes 0-255*/
begin                                      /*turns ascii*/
  write(c((h div 16)));                    /*and prints*/
  write(c((h mod 16)));
end;

begin
    writeln;
    write('Cylinder > ');                  /*cyl - char to dec*/
    number := 0;
    for i := 1 to 3 do
    begin
      read(N);                             /*read a character */
      N2 := orD(N);                        /*convert to number*/
      if N2>96 then N3:=N2-97+10           /*if letter-> 10-15*/
               else N3:=N2-48;             /*else number-> 0-9*/

      number := (number*16) + N3;          /*update number*/
    end;
    read(N);read(N);                       /*discard CR & LF*/


    write('Head     > ');                  /*Head- char to dec*/
    number2:= 0;
    for i := 1 to 2 do
    begin
      read(N);
      N2 := ord(N);
      if N2>96 then N3:=N2-97+10
               else N3:=N2-48;

      number2:= (number2*16) + N3;
    end;
    read(N);read(N);
```

```
/*******************************************************/
/*  PROGRAM TO CONVERT IBM CYL, HEAD AND SECTOR TO     */
/*  SCSI LOGICAL ADDRESS AND REPORT IN DEC AND HEX     */
/*                                                     */
/*                        Page 2                       */
/*******************************************************/


    write('Sector   > ');                  /*Sec - char to dec*/
    number3:= 0;
    for i := 1 to 2 do
    begin
      read(N);
      N2 := ord(N);
      if N2>96 then N3:=N2-97+10
               else N3:=N2-48;

      number3:= (number3*16) + N3;
    end;


    /*generate logical address*/

    writeln;
    number4 := (number*Mhd*Ms) + (number2*Ms) + number3 -1;


    /*and inform user - decimal*/

    writeln('Logical address = ',number4,' (d)');


    /*and inform user - hex (ascii)*/

    h1 := number4 mod 256;
    t  := number4 div 256;
    h2 := t   mod 256;
    t  := t   div 256;
    h3 := t   mod 256;
    h4 := t   mod 256;

    write('Logical address = ');
    c2(h1); c2(h2); c2(h3); c2(h4);

end.
```

```
/************************************************/
/*  PROGRAM TO CONVERT SCSI LOGICAL ADDRESS TO  */
/*  IBM CYL, HEAD AND SECTOR VALUES & CALCULATE */
/*  REGISTER VALUES FOR USE IN INT 13H CALL     */
/*                                              */
/*  Copyright David S. S. Robb 1990             */
/************************************************/


program logtoibm;

var     i,N2,N3,number,Cyl,Temp,RCX,RDX  : longint;
        Head,Sec,Cyl_upper,Cyl_lower     : longint;
        h1,h2,h3,h4,t                    : longint;
        N,hh1,hh2,hh3,hh4                : char;

const   Mhd = 4;
        Ms  = 17;

function c(h : longint):char;           /*dec 0-16 to char*/
begin
  if h>9 then c := chr((h-10)+97)       /*number*/
         else c := chr(h+48);           /*letter*/
end;

procedure c2(h : longint);              /*takes 0-255*/
begin                                   /*turns ascii*/
write(c((h div 16)));                   /*and prints*/
  write(c((h mod 16)));
end;

procedure c3(h : longint);              /*takes 0-4095*/
var t1 : longint;                       /*turns ascii*/
begin                                   /*and prints*/
  t1 := h div 16;
  if t1>15 then c2(t1) else write(c(t1));
  write(c((h mod 16)));
end;

begin
    writeln;
    write('3 Byte Logical Addr > ');
    number := 0;                        /*number holder*/

    for i := 1 to 6 do                  /*dec from hex(asc)*/
    begin
      read(N);                          /*read a character */
      N2 := ord(N);                     /*convert to number*/
      if N2>96 then N3:=N2-97+10        /*if letter-> 10-15*/
               else N3:=N2-48;          /*else number-> 0-9*/

      number := (number*16) + N3;       /*update number*/
    end;
    writeln('Decimal conversion  : ',number);

    /*compute cyl,head,sec breakdown*/

    Cyl     := number div (Mhd*Ms);
    Temp    := number mod (Mhd*Ms);
    Head    := Temp   div Ms;
    Sec     := Temp   mod Ms +1;
```

```
/************************************************/
/*  PROGRAM TO CONVERT SCSI LOGICAL ADDRESS TO  */
/*  IBM CYL, HEAD AND SECTOR VALUES & CALCULATE */
/*  REGISTER VALUES FOR USE IN INT 13H CALL     */
/*                                              */
/*                    Page 2                    */
/************************************************/


 if Cyl>1023 then writeln('Max Cyl Exceeded')
             else
               begin

                 /*inform user of cyl,head,sec breakdown*/

                 write('IBM Address (C,H,S) : ');
                 c3(cyl)   ; write(' ');
                 c2(head)  ; write(' ');
                 c2(Sec)   ; writeln(' (h)');

                 /*find values to place in IBM registers*/

                 Cyl_upper := (cyl div 256)*64;
                 Cyl_lower := (cyl mod 256)*256;

                 RCX := Cyl_lower + Cyl_upper + Sec;
                 RDX := head*256 + $80;

                 /*convert back to ascii and inform user*/

                 writeln;
                  h1 := RCX mod 256;
                  h2 := RCX div 256;
                 write('RCX = ');c2(h2);c2(h1);writeln;
                  h1 := RDX mod 256;
                  t  := RDX div 256;
                 write('RDX = ');c2(h2);c2(h1);writeln;

                end;
end.
```

```
/*****************************************/
/*  PROGRAM TO INTERROGATE THE SUPERVISOR */
/*  AND DISPLAY PERMISSION LIST OBTAINED  */
/*                                        */
/*  Copyright David S. S. Robb 1990       */
/*****************************************/


program prlist;

var   number,val : byte;
      number2,length,val2,i,j,k,l,t : integer;

procedure waitforit;                        /*wait for Superv*/
var temp : byte;                            /*to read byte*/
begin
  repeat                                    /*repeatedly*/
    temp := port[$329]                      /*read status*/
  until ((temp and $02) = $02);             /*until 'byte read'*/
end;

function c(h : longint):char;               /*dec 0-16 to char*/
begin
  if h>9 then c := chr((h-10)+97)           /*number*/
         else c := chr(h+48);               /*letter*/
end;

begin
  port[$329] := $87;                        /*send Sup 'prlist'*/
  waitforit;                                /*pause*/

  number := port[$329];                     /*RD no. of entries*/
  number2 := (number and $f0) div 16;       /*ans- upper nibble*/
  length := number2*20+10;                  /*no.of bytes to RD*/

  /*banner*/

  writeln;
  writeln('There are ',number2,' partitions');
  writeln('The permission table is as follows :');
  writeln;
  writeln('Boundary Addrs   Modified BAs   X    SWR  No.');

  /*ask Supervisor for required number of bytes*/

  port[$329] := length;
  waitforit;                                /*wait - Sup to RD*/
  port[$329] := 0;                          /*inform Sup ready*/

  /*read permission list and inform user*/

  for i:= 1 to (length div 10) do /*for each entry*/
  begin
    for k := 0 to 9 do              /*for each item in entry*/
    begin
    for j := 1 to 2 do              /*for each nibble of item*/
      begin
        waitforit;                  /*when byte read S ready*/
        val := port[$329];          /*read nibble*/
        port[$329] := 0;            /*tell S to get next nib*/
```

```
/*****************************************/
/*  PROGRAM TO INTERROGATE THE SUPERVISOR */
/*  AND DISPLAY PERMISSION LIST OBTAINED  */
/*                                        */
/*                Page 2                  */
/*****************************************/


  val2 := (val and $f0) div 16;    /*ans- upper nibble*/

       /*if modified address for user segment - don't care*/

       if ((k>3)and(k<8)and((i mod 2) = 1))
       then
         write('X')

       else

       /*else if attribute byte - binary representation*/

       if k=8
       then
         begin
           t := val2;
           if t div 8 =1 then write('1') else write('0');
           t := t mod 8;
           if t div 4 =1 then write('1') else write('0');
           t := t mod 4;
           if t div 2 =1 then write('1') else write('0');
           t := t mod 2;
           write(t);
         end

       else

       /*else - the remainder - hexadecimal representation*/

       write(c(val2));

       /*space between bytes*/

       if (j mod 2 = 0) then write(' ');

     end; /*item*/

     /*spacing between columns*/

     if (k = 3) then write ('     ');
     if (k = 7) then write ('   ');
     if (k = 8) then write (' ');

   end; /*entry*/
   writeln;

 end; /*list*/

 writeln;writeln(' "XX" = Dont Care');
end.
```

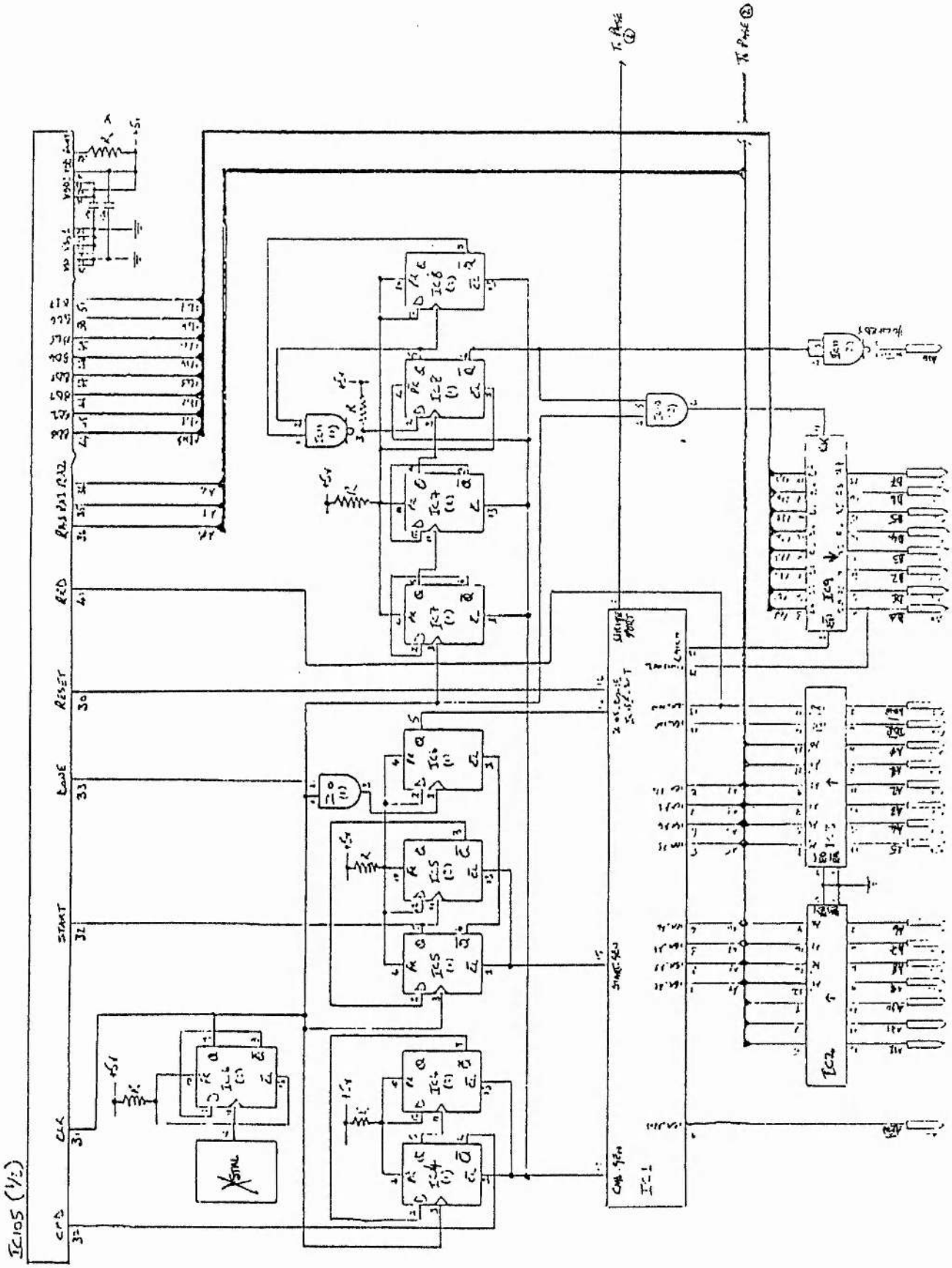
```
;*********************************************/
;*  SUPERVISOR ASSEMBLER ROUTINE TO SEND IBM */
;*  THE PERMISSION LIST WHEN REQUESTED       */
;*                                           */
;*  Copyright David S. S. Robb 1990          */
;*********************************************/


;* command 87H start point

prlist  ld    a,(ramit+listlen)  ;*load in list size
        sla   a                  ;*shift to upper nibble
        sla   a
        sla   a
        sla   a
        or    00001111b
        out   (StatusW),a        ;*and o/p listlen

        call  AWAIT              ;*wait for IBM ready
                                 ;*and read bytes required
        ld    b,a                ;*and store - length of repeat

        ld    iy,lists           ;*get pointer to list


loop    ld    a,(iy)             ;*send upper nibble
        inc   iy                 ;*update pointer
        ld    c,a                ;*temp store for low nibble
        and   11110000b
        out   (StatusW),a

        call  AWAIT              ;*inform IBM nibble ready

        ld    a,c                ;*send lower nibble
        sla   a
        sla   a
        sla   a
        sla   a
        out   (StatusW),a

        call  AWAIT              ;*inform IBM nibble ready

        djnz  loop               ;*repeat for all bytes in list


        jp    askIBM             ;*return to await new command
```

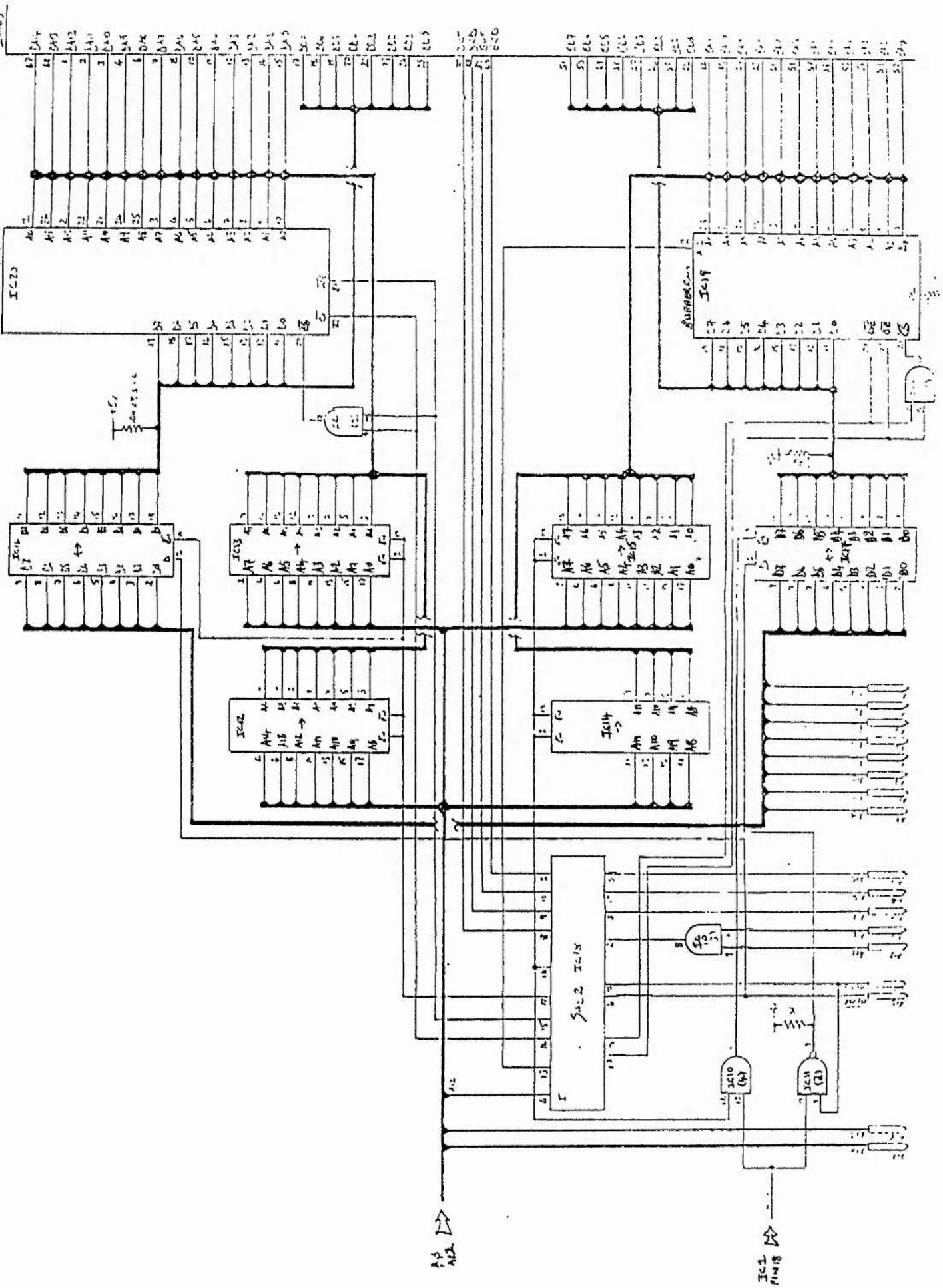
APPENDIX D

IC105 COMPRESSION BOARD

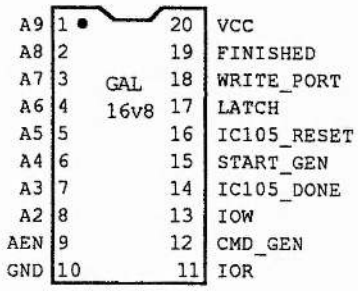
CIRCUIT DIAGRAM & GAL DESCRIPTIONS

Compression Project Circuit
Page 1
D. S. S. Robb;



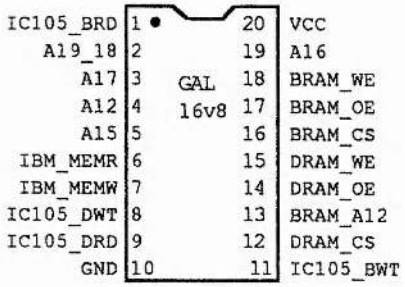


```
DEVICE  16v8;
TITLE  Compression  Project  GAL1.5
       NAME  D.  S.  S.  Robb;
        SIGNATURE COMP1.5;
```




```
CMD_GEN       =  A9 &  A8 & !A7 & !A6 & !A5 & !A4 & !A3 & !AEN & (!IOW | !IOR);

START_GEN     =  A9 &  A8 & !A7 & !A6 & !A5 & !A4 &  A3 &  A2 & !AEN & !IOW;

IC105_RESET   =  A9 &  A8 & !A7 & !A6 & !A5 & !A4 &  A3 & !A2 & !AEN & !IOW;

!LATCH        =  A9 &  A8 & !A7 & !A6 & !A5 & !A4 & !A3 & !AEN & !IOR;

FINISHED.OE   =  A9 &  A8 & !A7 & !A6 & !A5 & !A4 &  A3 & !A2 & !AEN & !IOR;
FINISHED      =  IC105_DONE;

!WRITE_PORT   =  A9 &  A8 & !A7 & !A6 & !A5 & !A4 & !A3 & !AEN & !IOW;
```

```
DEVICE  16v8;
TITLE  Compression  Project  GAL2.3
NAME  D.  S.  S.  Robb;
SIGNATURE COMP2.3;
```




```
!BRAM_CS     = A19_A18 & !A17 & A16 &  A15 & (!IBM_MEMR | !IBM_MEMW)

BRAM_A12     =    (A19_A18 & !A17 & A16 & A15) & A12)
                ! (A19_A18 & !A17 & A16 & A15) & IC105_BRD)

BRAM_OE      = !(A19_A18 & !A17 & A16 & A15 & !IBM_MEMR) & IC105_BRD

BRAM_WE      = !(A19_A18 & !A17 & A16 & A15 & !IBM_MEMW) & IC105_BWT

!DRAM_CS     = A19_A18 & !A17 & A16 & !A15 & (!IBM_MEMR | !IBM_MEMW)

DRAM_OE      = !(A19_A18 & !A17 & A16 & !A15 & !IBM_MEMR) & IC105_DRD

DRAM_WE      = !(A19_A18 & !A17 & A16 & !A15 & !IBM_MEMW) & IC105_DWT
```

APPENDIX E

COMPRESSION SOFTWARE

HUFFMAN COMPRESSION

LZW COMPRESSION

IC105 COMPRESSION

```
!************************************************************
!*  S-Algol program to calculate the Optimum Huffman Coding*
!* For a given distribution                                 *
!************************************************************

!tree structure used to generate huffman coding tree
structure tree(int order ; real value ; pntr right , left)

!number of symbols - max depth of tree = dep-1
let dep := 16

!artificial distribution (sorted) of 16 symbols
let vals=@0 of *real[@0 of real[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
                     @0 of real[22,19,14,9,5,5,5,4,4,3,3,2,2,1,1,1]]

!results store
let huff = vector 0 :: dep-1 of 0

!recursively traverse tree measuring root to leaf lengths
!and storing results
procedure travtree(pntr node ; int depth)
begin
  if node(order)~=-1                    !if leaf
   then                                 !then
    huff(node(order)) := depth          !store current depth with
   else                                 !symbol in results
    begin                               !else
      travtree(node(right),depth+1)     !recursive search right
      travtree(node(left),depth+1)      !and left branches
    end
end


procedure optihuff(int depth ; **real values)
begin

  let vstore = vector 0 :: depth-1 of nil     !create pointer list
  for i = 0 to depth-1 do                     !and for each symbol
    vstore(i) := tree(i,values(1,i),nil,nil)  !create structure

  for i = depth-1 to 1 by -1 do    !repeat until single tree
  begin

    !join two lowest entries in list
    let tempval = values(1,i-1) + values(1,i)   !add frequencies
    values(1,i-1) := tempval                    !update frequencies

    let temppnt = vstore(i-1)                   !store old pointer
    vstore(i-1) := tree(-1,0,temppnt,vstore(i)) !create new pointer

    !resort list by frequency
    let swapped := false
    repeat                    !repeat until no swaps in single pass
      begin
        swapped := false

        for j = 0 to i-1 do                     !bubble sort
          if values(1,j) < values(1,j+1) do     !if lower freq lower
           begin                                !in list then rises
```

```
!*************************************************************
!*  S-Algol  program  to  calculate  the  Optimum  Huffman  Coding*
!* For a given distribution                                     *
!*                              Page 2                          *
!*************************************************************

            let t0 := values(0,j+1)              !swap
            let t1 := values(1,j+1)
            values(0,j+1) := values(0,j)
            values(1,j+1) := values(1,j)
            values(0,j) := t0
            values(1,j) := t1
            let pnt = vstore(j+1)
            vstore(j+1) := vstore(j)
            vstore(j) := pnt
             swapped := true                     !swap in that pass
          end
      end
    while swapped
  end
  travtree(vstore(0),0)    !trav tree for results- depth=no. of bits
end

!find optimum huffman for distribution
optihuff(dep,vals)

!and print results
let huffy := 0.0
let tot := 0.0
for i = 0 to dep-1 do
  begin
    write i," ",huff(i)," 'n"          !Bit lengths for each symbol
    huffy := huffy + huff(i)*vals(1,i) !Calculate Compression
    tot := tot + vals(1,i)
  end

write "'nThe uncompressed file required ",tot, " Bytes"
write "'nThe compressed file would require ",huffy/8:2," Bytes"

!RESULTS  -  For  distribution  given
!                       symbol     no.  of  bits
!                                  required

                        0          2
                        1          2
                        2          3
                        3          4
                        4          4
                        5          5
                        6          5
                        7          5
                        8          5
                        9          5
                        10         5
                        11         6
                        12         6
                        13         6
                        14         7
                        15         7

The uncompressed file required      251   Bytes

The compressed file would require   95    Bytes
```

```
{*********************************************************}
{* Works out the distribution of data in a given file *}
{*                  version 1.0   (26/10/90)              *}
{*                     David S.S. Robb                    *}
{*********************************************************}

{takes files and generates a distribution file}
{takes a distribution file and generates a huffman coding file}
{calculates the compression for a given distribution and huffman}

{Generated using Borland Turbo Pascal for the IBM PC}

{Compiler Directives}

{$R-}    {Range checking off}
{$B+}    {Boolean complete evaluation on}
{$S+}    {Stack checking on}
{$I+}    {I/O checking on}
{$N-}    {No numeric coprocessor}
{64k stack , 1 meg heap !}

{The Program}

program dist;
uses crt,graph,dos;

type     treeptr          = ^tree;          {tree structure}
         tree             = record
                              order : integer;
                              value : real;
                              right : treeptr;
                              left  : treeptr;
                            end;
         holder           = array[0..100] of integer;

var      progfile         : file of byte;
         valfile          : file of real;
         huffile          : file of byte;
         name,name2       : string;
         ch,ans           : char;
         i,j,nib,bynib    : integer;
         answer           : integer;
         nibble_sum       : array[1..3] of real;
         byte_sum         : real;
         nibble_huff_sum  : array[1..3] of real;
         nibble_dist_sum  : array[1..3] of real;
         byte_huff_sum    : real;
         glob_byte        : byte;
         glob_hi_nib      : byte;
         glob_lo_nib      : byte;
         byte_count       : array [0..2,0..255] of real;
         nibble_count     : array [0..9,0..15] of real;
         huff             : array [1..3,0..15] of byte;
         huff2            : array [0..255] of byte;
         shuff2           : array [0..255] of holder;
         holdit           : holder;
         counter          : integer;

procedure convert;  {split byte into two nibbles}
begin
  glob_hi_nib := glob_byte div 16;
  glob_lo_nib := glob_byte mod 16;
end;
```

```
{*********************************************************}
{* Works out the distribution of data in a given file  *}
{*                  version 1.0  (26/10/90)              *}
{*                        Page 2                         *}
{*********************************************************}

{create new distribution file}
procedure newfile;
var       len,typ  : byte;
          finished : boolean;
begin
  write('Please enter source filename  - ');
  readln(name);
  assign(progfile,name);
  reset(progfile);
  write('Please enter destination filename - ');
  readln(name2);
  assign(valfile,name2);
  rewrite(valfile);
  finished := false;
  while not(eof(progfile)) do
  begin
    read(progfile,glob_byte);
    byte_count[1,glob_byte] := byte_count[1,glob_byte] + 1;
    convert;
    nibble_count[1,glob_hi_nib]:= nibble_count[1,glob_hi_nib] + 1;
    nibble_count[1,glob_lo_nib]:= nibble_count[1,glob_lo_nib] + 1;
    nibble_count[2,glob_hi_nib]:= nibble_count[2,glob_hi_nib] + 1;
    nibble_count[3,glob_lo_nib]:= nibble_count[3,glob_lo_nib] + 1;
  end;
  for i := 0 to 255 do
    write(valfile,byte_count[1,i]);
  for i := 0 to 15 do
  begin
    write(valfile,nibble_count[1,i]);
    write(valfile,nibble_count[2,i]);
    write(valfile,nibble_count[3,i]);
  end;
  close(valfile);
  close(progfile);
end;


{load in distribution file}
procedure oldfile;
var  len,typ  : byte;
     finished : boolean;
begin
  write('Please enter filename - ');
  readln(name2);
  assign(valfile,name2);
  reset(valfile);
  for i := 0 to 255 do
    read(valfile,byte_count[1,i]);
  for i := 0 to 15 do
  begin
    read(valfile,nibble_count[1,i]);
    read(valfile,nibble_count[2,i]);
    read(valfile,nibble_count[3,i]);
  end;
  close(valfile);
end;
```

```
{*********************************************************}
{* Works out the distribution of data in a given file *}
{*                 version 1.0   (26/10/90)               *}
{*                          Page 3                        *}
{*********************************************************}

{load a huffman coding file from disk}
procedure loadhuff;
var  len,typ  : byte;
     finished : boolean;
begin
  write('Please enter filename - ');
  readln(name2);
  assign(huffile,name2);
  reset(huffile);
  for i := 0 to 255 do
    read(huffile,huff2[i]);
  for i := 0 to 15 do
  begin
    read(huffile,huff[1,i]);
    read(huffile,huff[2,i]);
    read(huffile,huff[3,i]);
  end;
  close(huffile);
end;

{save a huffman coding file to disk}
procedure savehuff;
var temp, len,typ,d1,d2  : byte;
     finished : boolean;
begin
  write('Please enter filename - ');
  readln(name2);
  assign(huffile,name2);
  rewrite(huffile);
  for i := 0 to 255 do
    write(huffile,huff2[i]);
  for i := 0 to 15 do
  begin
    write(huffile,huff[1,i]);
    write(huffile,huff[2,i]);
    write(huffile,huff[3,i]);
  end;
  for i := 0 to 255 do
  begin
    temp := lo(round(byte_count[0,i]));
    writeln(temp);
    write(huffile,temp);
  end;
  for i := 0 to 255 do
  begin
    for j := 0 to huff2[i]-1 do
    begin
      d1 := hi(shuff2[i,j]);
      d2 := lo(shuff2[i,j]);
      write(huffile,d1);
      write(huffile,d2);
    end;
  end;
  close(huffile);
readln(nib);
end;
```

```
{*********************************************************}
{*  Works out the distribution of data in a given file   *}
{*                  version 1.0  (26/10/90)              *}
{*                        Page 4                         *}
{*********************************************************}

{sort the nibble distribution counts}
procedure simple_sort_byte;
var  t0,t1 : real;
     swapped : boolean;
begin
  repeat
    begin
      swapped := false;
      for i := 0 to 254 do
      begin
        if byte_count[1,i] < byte_count[1,i+1] then
        begin
          t0 := byte_count[0,i+1];
          t1 := byte_count[1,i+1];
          byte_count[0,i+1] := byte_count[0,i];
          byte_count[1,i+1] := byte_count[1,i];
          byte_count[0,i] := t0;
          byte_count[1,i] := t1;
           swapped := true;
        end;
      end;
    end
  until not(swapped)
end;


{sort the nibble distribution counts}
procedure simple_sort_nibble(a,b : integer);
var  t0,t1,t3,t5 : real;
     swapped : boolean;
begin
  repeat
    begin
      swapped := false;
      for i := 0 to 14 do
      begin
        if nibble_count[a,i] < nibble_count[a,i+1] then
        begin
          t0 := nibble_count[b,i+1];
          t1 := nibble_count[a,i+1];
          nibble_count[b,i+1] := nibble_count[b,i];
          nibble_count[a,i+1] := nibble_count[a,i];
          nibble_count[b,i] := t0;
          nibble_count[a,i] := t1;
           swapped := true;
        end;
      end;
    end
  until not(swapped)
end;
```

```
{*********************************************************}
{* Works out the distribution of data in a given file  *}
{*                 version 1.0   (26/10/90)              *}
{*                        Page 5                         *}
{*********************************************************}

{sort the byte huffman codes}
procedure simple_sort_huff(a : integer);
var  t0,i : integer;
     swapped : boolean;
begin
  repeat
    begin
      swapped := false;
      for i := 0 to 14 do
      begin
        if huff[a,i] > huff[a,i+1] then
        begin
          t0 := huff[a,i+1];
          huff[a,i+1] := huff[a,i];
          huff[a,i] := t0;
          swapped := true;
        end;
      end;
    end
  until not(swapped)
end;

procedure results;
var typ : integer;
begin                                  {total symbols in file-nibble}
  for i := 0 to 15 do
    for j := 1 to 3 do
    begin
      nibble_sum[j] := nibble_sum[j] + nibble_count[j,i];
    end;

  for i := 0 to 255 do                 {total symbols in file-byte}
  begin
    byte_sum := byte_sum + byte_count[1,i];
  end;

  {sort results}
  simple_sort_byte;
  simple_sort_nibble(1,4);
  simple_sort_nibble(2,5);
  simple_sort_nibble(3,6);

  {proportions of each symbol in the file}
  for i := 0 to 15 do
    for j := 1 to 3 do
    begin
      nibble_count[j+6,i]    :=    nibble_count[j,i]/nibble_sum[j];
      nibble_dist_sum[j]     :=    nibble_dist_sum[j] +
                                   nibble_count[j,i]/nibble_sum[j];
    end;
  for i := 0 to 255 do
  begin
    byte_count[2,i] := byte_count[1,i]/byte_sum;
  end;
end;
```

```
{*********************************************************}
{* Works out the distribution of data in a given file *}
{*                  version 1.0  (26/10/90)              *}
{*                        Page 6                         *}
{*********************************************************}

{find compression - sum of frequency*huffman code length
procedure huffsum;
begin
  for i := 0 to 15 do
    for j := 1 to 3 do
    begin
      nibble_huff_sum[j]:=nibble_huff_sum[j] +
                nibble_count[j,i]/nibble_sum[j] * huff[j,i];
    end;
  for i := 0 to 255 do
  begin
    byte_huff_sum:=byte_huff_sum +
                byte_count[1,i]/byte_sum * huff2[i];
  end;
end;

{recursive procedure to suck info from tree - all nibble (place)}
procedure ntravtree(node:treeptr ; depth,place:integer);
begin
  if node^.order<>-1
    then
      huff[place,node^.order] := depth  {huffman code lengths}
    else
      begin
        ntravtree(node^.right,depth+1,place);
        ntravtree(node^.left ,depth+1,place);
      end;
end;

{recursive procedure to suck info from tree - byte}
procedure btravtree(node:treeptr; depth:integer; list:holder);
begin
  if node^.order<>-1
    then
      begin
        huff2[node^.order] := depth;    {huffman code lengths}
        shuff2[node^.order] := list;
      end
    else
      begin
        list[depth] := counter;
        counter := counter+1;
        btravtree(node^.right,depth+1,list);
        list[depth] := counter;
        counter := counter+1;
        btravtree(node^.left ,depth+1,list);
      end;
end;
```

```
{*********************************************************}
{*  Works out the distribution of data in a given file   *}
{*                  version 1.0  (26/10/90)              *}
{*                        Page 7                         *}
{*********************************************************}

procedure optihuff(A,b,c,depth : integer); {general to byte&nib}
var       arr     : array[0..1,0..255] of real;
          vstore  : array[0..255] of treeptr;
          one     : treeptr;
          b1,b2,b3: integer;
          t,t0,t1 : real;
          swapped : boolean;
begin
  for i := 0 to depth-1 do
  begin
    case A of
    1 : begin                              {fill local array}
          arr[0,i] := byte_count[0,i];     {with ordered dist}
          arr[1,i] := byte_count[1,i];
        end;
    2 : begin
          arr[0,i] := nibble_count[b,i];
          arr[1,i] := nibble_count[c,i];
        end;
    end;
    new(one);                              {tree root}
    one^.order := i;
    one^.value := arr[1,i];
    one^.right := nil;
    one^.left  := nil;
    vstore[i] := one;
  end;
  for i := depth-1 downto 1 do
  begin                                    {take bottom two and join}
    t := arr[1,i-1]+arr[1,i];
    new(one);
    one^.order := -1;
    one^.value := t;
    one^.right := vstore[i-1];
    one^.left  := vstore[i];
    vstore[i-1] := one;
    arr[1,i-1]  := t;                      {put result back in array}
    repeat                                 {and resort list}
    begin
      swapped := false;
      for j := 0 to i-1 do
      begin
        if arr[1,j] < arr[1,j+1] then
        begin
          t0 := arr[0,j+1]       ; t1 := arr[1,j+1];
          arr[0,j+1] := arr[0,j] ; arr[1,j+1] := arr[1,j];
          arr[0,j] := t0         ; arr[1,j] := t1;
          one := vstore[j+1];
          vstore[j+1] := vstore[j];
          vstore[j] := one;
          swapped := true;
        end;
      end;
    end
    until not(swapped);

  end;                                 {and repeat until one entry only}
```

```
{*********************************************************}
{* Works out the distribution of data in a given file *}
{*                  version 1.0   (26/10/90)             *}
{*                        Page 8                         *}
{*********************************************************}

  {now we have a Huffman tree}

  {get huffman code lengths from trees}
  counter:=0;
  if A=1 then btravtree(vstore[0],0,holdit)
         else ntravtree(vstore[0],0,b-3);


  {sort resultant huffman code lengths}
  case A of
  1 : repeat                              {byte}
      begin
        swapped := false;
        for j := 0 to 254 do
        begin
          if huff2[j] > huff2[j+1] then
          begin
            b1 := huff2[j+1];
            huff2[j+1] := huff2[j];
            huff2[j] := b1;
             swapped := true;
          end;
        end;
      end
      until not(swapped);
  2 : for i := 1 to 3 do                  {nibble}
        simple_sort_huff(i);
  end;
end;

procedure genhuff;
var i : integer;
begin
  optihuff(1,0,1,256);              {Byte Huffman}
  for i := 1 to 3 do                {and three nibbles!}
    optihuff(2,i+3,i+6,16);
end;


procedure banner;
begin
   writeln;
  writeln('Optimum Huffman Code Generator');
   writeln;
end;
```

```
{*********************************************************}
{* Works out the distribution of data in a given file *}
{*                 version 1.0  (26/10/90)                *}
{*                        Page 9                          *}
{*********************************************************}

procedure print_to_screen;      {prints the Nibble distribution}
var       size : real;
          typ : integer;
begin
  writeln('Nibble distribution (sorted)');
  for i := 0 to 15 do
  begin
    write(nibble_count[5,i]:2:0,' = ',nibble_count[8,i]:3:2);
    writeln(' - ',huff[2,i]:2,' -
                  ',nibble_count[8,i]*huff[2,i]:4:2);
  end;
  writeln;
  writeln('      ',nibble_dist_sum[1]:3:2,'
                  ',nibble_huff_sum[1]:4:2);
end;

procedure print_res;
begin
  writeln;
  writeln('    NIBBLES                                    . BYTES');
  writeln('  All   : Upper : Lower  |');                    .
  for i := 0 to 15 do
  begin
    write(nibble_count[7,i]:2:2,' ',huff[1,i]:2,':');
    write(nibble_count[8,i]:2:2,' ',huff[2,i]:2,':');
    write(nibble_count[9,i]:2:2,' ',huff[3,i]:2,' | ');
    for j := 0 to 15 do
      write(huff2[(j*16+i)]:1,' ');
    writeln;
  end;
  writeln('-------:-------:--------|------------------------');
  write(' ',nibble_huff_sum[1]:3:2);
  write('  : ',nibble_huff_sum[2]:3:2);
  write('  : ',nibble_huff_sum[3]:3:2,'   |');
  write('                    **(',byte_huff_sum:3:2,')**');
  writeln;
end;

procedure initialise_variables;
begin
  for i := 1 to 3 do        {clear values in arrays}
  begin
    nibble_sum[i] := 0;     {upper,lower & both sums}
    nibble_huff_sum[i] :=0;
    nibble_dist_sum[i] :=0;
  end;

  byte_sum := 0;
  byte_huff_sum   := 0;

  for i := 0 to 255 do      {for each symbol represented by a byte}
  begin
    byte_count[0,i]:=i ; byte_count[1,i]:=0;
    holdit[i] := 0;
    for j := 0 to 100 do
      shuff2[i,j]:=0;
  end;
```

```
{*********************************************************}
{* Works out the distribution of data in a given file *}
{*                  version 1.0  (26/10/90)              *}
{*                        Page 10                        *}
{*********************************************************}

  for i := 0 to 15 do      {for each symbol repres. by a nibble}
  begin
    nibble_count[0,i]:=i;
    for j := 1 to 3 do nibble_count[j,i]:=0;
    for j := 4 to 6 do nibble_count[j,i]:=0;
    for j := 7 to 9 do nibble_count[j,i]:=0;
  end;
end;

{Main Routine}
begin
  banner;
  initialise_variables;

  {generate or load a distribution for a given file}
  write('Use an existing DST file or create new one? (1/2) - ');
  readln(answer);
  if answer=1 then oldfile else newfile;

  results;

  {generate or load a huffman coding}
  write('Use an existing HUF file or create & use optimum one?
         (1/2) - ');
  readln(answer);
  if answer=1 then loadhuff else begin genhuff ; savehuff end;


  huffsum;       {calculate the resulting compression}
  print_res;     {print results}
  readln(i);     {wait for keypress}

  for i := 0 to 18 do
  begin
    for j := 0 to huff2[i]-1 do
      write(shuff2[i,j],' ');
    writeln;
  end;
  readln(i);     {wait for keypress}
end.
```

```
{****************************************}
{*  generates huffman codes from a file  *}
{*    of optimum huffman code lengths    *}
{*         version 1.0   (26/10/90)      *}
{*              David S.S. Robb          *}
{****************************************}

{Compiler Directives}

{$R-}     {Range checking off}
{$B+}     {Boolean complete evaluation on}
{$S+}     {Stack checking on}
{$I+}     {I/O checking on}
{$N-}     {No numeric coprocessor}
{64k stack , 1 meg heap !}

{The Program}

program hufgen;
uses crt,graph,dos;

var       outfile,huffile     : file of byte;
          name                : string;
          i,j,n               : integer;
          huff                : array [1..3,0..15] of byte;
          huff2               : array [0..1,0..255] of byte;
          shuff2              : array [0..255,0..100] of integer;
          lookup              : array [0..509] of integer;

{Load file of optimum huffman code lengths}
procedure loadhuff;
var d1,d2 : byte;
begin
  write('Please enter source filename - ');
  readln(name);
  assign(huffile,name);
  reset(huffile);
  for i := 0 to 255 do
    read(huffile,huff2[1,i]);
  for i := 0 to 15 do
  begin
    read(huffile,huff[1,i]);
    read(huffile,huff[2,i]);
    read(huffile,huff[3,i]);
  end;
  for i := 0 to 255 do
    read(huffile,huff2[0,i]);
  for i := 0 to 255 do
    for j := 0 to huff2[1,i]-1 do
      begin
        read(huffile,d1);
        read(huffile,d2);
        shuff2[i,j]:=(d1*256)+d2;
      end;
  close(huffile);
end;
```

```
{****************************************}
{*  generates huffman codes from a file  *}
{*    of optimum huffman code lengths    *}
{*         version 1.0   (26/10/90)      *}
{*                  Page 2               *}
{****************************************}

{procedure to save resultant huffman codes}
procedure saveres;
var d : byte;
begin
  write('Please enter destination filename - ');
  readln(name); assign(outfile,name);
  rewrite(outfile);
  for i := 0 to 255 do write(outfile,huff2[1,i]);
  for i := 0 to 255 do write(outfile,huff2[0,i]);
  for i := 0 to 255 do
    for j := 0 to huff2[1,i]-1 do
      begin
         d := lo(shuff2[i,j]);
         write(outfile,d);
      end;
  close(outfile);
end;

procedure gencode;
var temp : integer;
begin
  for i := 0 to 255 do
    for j := 0 to huff2[1,i]-1 do
      shuff2[i,j] := lookup[shuff2[i,j]];
end;

procedure printres;
var len : byte;
begin
  for i := 0 to 255 do
  begin
    write(huff2[1,i],' - ');
    for j := 0 to huff2[1,i]-1 do
      write(shuff2[i,j]:2,' ');
    writeln;
  end;
  readln(i);
end;


procedure initialise_variables;
begin
  for i := 0 to 509 do
    lookup[i] := i mod 2;
end

begin
  initialise_variables;
  loadhuff;
  gencode;
  printres;
  saveres;
end.
```

```
{*************************************}
{*            Hufman codes a file!   *}
{*          version 1.0 (05/11/90)   *}
{*                David S.S. Robb    *}
{*************************************}

{Compiler Directives}

{$R-}     {Range checking off}
{$B+}     {Boolean complete evaluation on}
{$S+}     {Stack checking on}
{$I+}     {I/O checking on}
{$N-}     {No numeric coprocessor}
{64k stack , 1 meg heap !}

{The Program}

program dohuf;
uses crt,graph,dos;
type list = array [0..100] of byte;
var   outfile,infile,huffile  : file of byte;
            name,name2              : string;
            i,j,n                   : integer;
            huff2                   : array [0..1,0..255] of byte;
            shuff2                  : array [0..255] of list;
            buffer                  : array [0..2047] of byte;
            top                     : integer;


{load in the Huffman coding}
procedure loadhuf;
var d : byte;
begin
  write('Please enter Huffman filename - ');
  readln(name);
  assign(huffile,name);
  reset(huffile);
  for i := 0 to 255 do
    read(huffile,huff2[1,i]);
  for i := 0 to 255 do
    read(huffile,huff2[0,i]);
  for i := 0 to 255 do
    for j := 0 to huff2[1,i]-1 do
      read(huffile,shuff2[i,j]);
  close(huffile);
end;


{dump 256 byte buffer to destination file}
procedure dumpbuffer;
var   temp  : byte;
      i,j   : integer;
begin
  writeln('dumped buffer');
  for i := 0 to 255 do                {put bits together}
  begin                               {and write out}
    temp := 0;
    for j := 0 to 7 do
      temp := temp*2 + buffer[i*8+j];
    write(outfile,temp);
  end;
  top := 0;                           {buffer empty}
end;
```

```
{**************************************}
{*            Hufman codes a file!    *}
{*          version 1.0 (05/11/90)    *}
{*                  Page 2            *}
{**************************************}

{compress the file}
procedure dothehuff;
var   address     : byte;
      count             : integer;
begin
 count := 0;
  while not(eof(infile)) do                {repeat until source read}
    begin
    count := count +1;                     {keep a byte count}
      read(infile,address);
      for i := 0 to huff2[1,address]-1 do {rept for bit len. of code}
      begin
        if top=2048 then dumpbuffer;       {if buffer full then dump}
                                           {into dest file}
        buffer[top]:=shuff2[address,i];    {dump bits of code to buff}
        top := top+1;                      {keep buffer fill level}
      end;
    end;
    writeln(count);
end;

procedure banner;
begin
  writeln;
  writeln('Hufman Compression Utility');
  writeln;
end;

{select source filename and destination for compressed version}
procedure initialise_variables;
begin
  write('Please enter source filename - ');
  readln(name);
  assign(infile,name);
  reset(infile);
  write('Please enter destination filename - ');
  readln(name);
  assign(outfile,name);
  rewrite(outfile);
end;

{Main Routine}
begin
  banner;
  initialise_variables;
  loadhuf;
  dothehuff;
end.
```

```
{**************************************}
{*          Hufman decodes a file!      *}
{*          version 1.0 (07/11/90)      *}
{*             David S.S. Robb          *}
{**************************************}

{Compiler Directives}
{$R+}   {Range checking off}
{$B+}   {Boolean complete evaluation on}
{$S+}   {Stack checking on}
{$I+}   {I/O checking on}
{$N-}   {No numeric coprocessor}
{64k stack , 1 meg heap !}

{The Program}
program dohuf;
uses crt,graph,dos;
type   list = array[0..100] of byte;

var    outfile,infile,huffile   : file of byte;
       name,name2               : string;
       i,j,n                    : integer;
       huff2                    : array [0..1,0..255] of byte;
       shuff2                   : array [0..255] of list;
       buffer                   : array [0..2047] of byte;
       top,bottom               : integer;
       readc                    : longint;
       fillbuff                 : integer;
       finished                 : boolean;
       funk                     : byte;

{load in the Huffman code}
procedure loadhuf;
var d : byte;
begin
 write('Please enter Huffman filename - ');
 readln(name);
 assign(huffile,name);
 reset(huffile);
 for i := 0 to 255 do read(huffile,huff2[1,i]);
 for i := 0 to 255 do read(huffile,huff2[0,i]);
 for i := 0 to 255 do
  for j := 0 to huff2[1,i]-1 do
   read(huffile,shuff2[i,j]);
 close(huffile);
end;

{test for a match between the top of the buffer and a selected huffman code}
function compare( v : integer ):boolean;
var    i       : integer;
       same    : boolean;
begin
 same := true;
 for i := 0 to huff2[1,v]-1 do
  if same then
   if shuff2[v,i]<>buffer[(bottom+i)mod 2048] then  same := false;
compare := same;
end;
```

```
{*************************************}
{*            Hufman codes a file!          *}
{*          version 1.0 (05/11/90)          *}
{*                  Page 2                   *}
{*************************************}

{fill up the cyclic input buffer}
procedure fillbuffer;
var   temp,temp2  : byte;
      i,j         : integer;
      test,full   : boolean;
begin
  if eof(infile)
  then finished:=true
  else
  begin
    full := false;
    while not(full) do
    begin
     {test for full}
      if (bottom<=top)or(bottom>top+8)    then test:=true
                                          else test:=false;
      if (not(eof(infile)))and(test) then
      {not full and still bytes in source}
      begin
        readc := readc +1;
        read(infile,temp);    {read a byte from source}
        temp2 := 128;
        for j := 0 to 7 do
        begin                 {split into bits and place in buffer}
          buffer[top+j]:= temp div temp2;
          temp  := temp mod temp2;
          temp2 := temp2 div 2;
        end;
        top := (top+8) mod 2048;
      end;
      if (bottom>=top)and(bottom<top+8)   then full:=true
                                          else full:=false;
      if eof(infile) then full:=true;
    end;
  end;
end;

{see if input buffer requires filling - if whole of symbol not in
buffer}
function nearlyempty(v : integer):boolean;
var test : boolean;
begin
  if bottom<top then if top-bottom<=huff2[1,v]
                                then test:=true
                                else test:=false
                else if top+2048-bottom<=huff2[1,v]
                                then test:=true
                                else test:=false;
nearlyempty:=test;
end;
```

```
{*************************************}
{*           Hufman codes a file!      *}
{*         version 1.0 (05/11/90)      *}
{*                 Page 3              *}
{*************************************}

{decompress the file}
procedure dotheunhuf;
var   address     : byte;
      count       : integer;
      found       : boolean;
      val,topy    : integer;
begin
  count  := 0; bottom := 0; top := 0;
  finished := false;
  fillbuffer;
  while not((finished)and(top=bottom)) do
    begin
      count := count +1;                    {byte count}
      found := false;
      val   := -1;
      while not(found) do                   {keep trying}
      begin
        val := val +1;
        if nearlyempty(val) then fillbuffer;
        found := compare(val);              {until huff code match}
      end;

      write(outfile,huff2[0,val]);          {found -write byte to dest}
      bottom := (bottom + huff2[1,val]) mod 2048;
                                            {and update buffer & rept}
    end;
 end;

procedure banner;
begin
  writeln;
  writeln('Hufman Decompression Utility');
  writeln;
end;

{select compressed source filename and dest for decompressed version}
procedure initialise_variables;
begin
  write('Please enter source filename - ');
  readln(name);
  assign(infile,name);
  reset(infile);
  write('Please enter destination filename - ');
  readln(name);
  assign(outfile,name);
  rewrite(outfile);
  fillbuff := 0;
  readc :=0;
end;

{Main routine}
begin
  banner;
  initialise_variables;
  loadhuf;
  dotheunhuf;
end.
```

```
!****************************
!*  LZW  Compression  program  *
!*       version 1.0           *
!*       D.Robb 1991           *
!****************************

!each entry - code/pointer to another entry
let dictionary = vector 0::8191,1::2 of 0

!number of literals - top entries of dict
let last.lit = 300

!number of bits in code word
let number.of.bits := 9

let infile := nullfile
let entry.no := last.lit +1                !current entry number
let max.entries = 8192
let compressed.size := 0

!open source file to be compressed
procedure openfile
begin
  write "Enter filename to compress - "
  let ans = read.a.line
  infile := open(ans,0)
  if infile=nullfile do {write "ERROR" ; openfile}
end

!pretty print for results output
!prints <space> and <ret> chars as strings
!recursively travels pointers to gain full dictionary entry
procedure get.codes(int code1 -> string)
  (if code1<=last.lit
      then (if code1=13 then "<ret>"
                        else  if code1=32 then "<space>"
                                          else code(code1))
      else get.codes(dictionary(code1,1))) ++
              (if dictionary(code1,2)=13
                  then "<ret>"
                  else  if dictionary(code1,2)=32
                           then "<space>"
                           else code(dictionary(code1,2)))

!keeps a track of compressed file length
!would output to destination file here
procedure output.code( int code )
begin
  let temp :=0
  case true of
      codes >= 4096      : temp :=13
      codes >= 2048      : temp :=12
      codes >= 1024      : temp :=11
      codes >= 512             : temp :=10
      default            : temp :=9
 compressed.size := compressed.size + temp
end
```

```
!****************************
!*  LZW  Compression  program  *
!*       version 1.0           *
!*          Page 2             *
!****************************

!add entry to the dictionary
procedure add.entry(int code,lit)
begin
  dictionary(entry.no)(1) := code    !code for last character
  dictionary(entry.no)(2) := lit     !pointer to rest of entry
  entry.no := entry.no + 1
end

clear.output
openfile
let oldcode := decode(read(infile))      !read first byte
let newcode := 0
let input.count := 1                     !have read one byte

!check if already in dictionary
procedure in.dictionary(int code,lit -> bool)
begin
  let found := false
  for i = last.lit +1 to entry.no do
    if dictionary(i,1)=code and dictionary(i,2)=lit do
      { found := true ; oldcode := i }
found
end

!compress file and build dictionary
while ~eof(infile) do
begin
   repeat
    {newcode := decode(read(infile)) ;
     input.count := input.count + 1}      !read byte
  while in.dictionary(oldcode,newcode) and ~eof(infile)
      !repeat while in dictionary
  output.code(oldcode)                   !output code
  add.entry(oldcode,newcode)             !add entry to disk

  !dictionary entries grow in length on boundaries
  case entry.no of
    512,1024,2048,4096  : number.of.bits := number.of.bits + 1
    max.entries         : begin write "EMPTY DICT"
                                entry.no := last.lit + 1
                                number.of.bits := 9
                          end
    default             : {}

  oldcode := newcode
end
output.code(oldcode)  !last byte
```

```
!****************************
!*  LZW  Compression  program  *
!*       version 1.0           *
!*          Page 3             *
!****************************


!print out results

write "'norigional number of bits - ",input.count*8
write "'n'ncompressed to - ",compressed.size
let num := last.lit +1
write "'nDictionary Created :- 'n"
while dictionary(num,1)~=0  do
begin
  write "'n",dictionary(num,1)," : ",dictionary(num,2)
  write "  =  ",get.codes(dictionary(num,1))
  write " + ",
      if dictionary(num,2)=13
            then "<ret>"
            else  if dictionary(num,2)=32
                        then "<space>"
                        else   code(dictionary(num,2))
  num := num + 1
end
```

```
!************************************************
!*  LZW  Compression  program  -  Example  Execution  *
!*                    version 1.0                     *
!*                    D.Robb 1991                     *
!************************************************

!test  source  file

This is a

!program  output

Enter filename to compress - testing

origional number of bits -    72          !9 x 8bits

compressed to -               72          !8 x 9bits

Dictionary Created :-

          84      :          104    =  T  + h
         104      :          105    =  h  + i
         105      :          115    =  i  + s
         115      :           32    =  s  + <space>
          32      :          105    =  <space>  + i
         303      :           32    =  i s + <space>
          32      :           97    =  <space>  + a
```

```
{*****************************}
{*  IC105  Controller  Program  *}
{*          Version 1.0          *}
{*       David S. S. Robb        *}
{*****************************}


program comp3;
uses crt;

var infile                              : file of byte;
    filename                            : string;
    ch                                  : char;
    i,j,clusters                        : integer;
    value                               : longint;
    temp,buffers,leftover,size          : integer;

function read_results : longint;
var v1,v2,v3 : integer;
begin
  v1 := port[$304];
  v2 := port[$305];
  v3 := port[$306];
  read_results := v1 + v2*$100 + v3*$10000;
end;

procedure pause(length : integer);
var i : integer;
begin
  for i := 1 to length do begin end;
end;

procedure compress_buff(bytes : integer);
const  busy       = 254;
var    t          : byte;
       status     : byte;
begin
  for i := 0 to bytes-1 do
    read(infile,mem[$d800:i]);
  port[$30c] := 00; {compress}
  status := port[$308];
  while status=busy do
     begin status:=port[$308] end; {wait until finished}
end;

procedure setup(bytes : integer);
begin
  port[$308] := 0; {reset chip}
  for i := 1 to 10000 do begin end;
  port[$304] := bytes div $10000;
  bytes := bytes mod $10000;
  port[$303] := bytes div $100;
  bytes := bytes mod $100;
  port[$302] := bytes;
end;
```

```
{*****************************}
{*  IC105  Controller  Program  *}
{*         Version 1.0          *}
{*           Page 2             *}
{*****************************}

begin
     writeln;
     writeln; writeln('File Compression'); writeln;
     write('Enter input filename - ');
     readln(filename);
     assign(infile,filename);
     reset(infile);

     value  := 0;
     size := filesize(infile);
     buffers := size div 4096;
     clusters := buffers*2;
     leftover := size mod 4096;
     if leftover>2048 then clusters := clusters+1;
     if leftover>0 then clusters := clusters+1;
     writeln('File is ',size,' bytes');
     writeln('buffers = ',buffers);
     writeln('leftover = ',leftover);

     setup(size);
     writeln;writeln('Compressing ...');
     for j := 1 to buffers do
     begin
       writeln('Compressing buffer ',j);
       compress_buff(4096);
     end;
     compress_buff(leftover);
     pause(10000);              {wait for registers to be valid}
     value := read_results;

     writeln; writeln('Compressed ',clusters,' clusters');
     writeln('It compressed to ',value div 2048,' clusters');
     writeln('with ',value mod 2048,' over');
     readln(ch);

     close(infile);
end.
```

ACKNOWLEDGEMENTS

During the course of this project I have had help and guidance from many sources and for all this help I am thankful. To my supervisor Dr. Reg Killean, I am particularly indebted, for his help, encouragement, and undying sense of humour.

My thanks to Alasdair and Sheena for proof-reading this work and for feeding me drinks in times of great need. My thanks also to Wendy, who not only corrected my appalling grammar, but also put up with me whilst I wrote it.

Finally I would like to thank my parents for giving me support in all of its many forms.