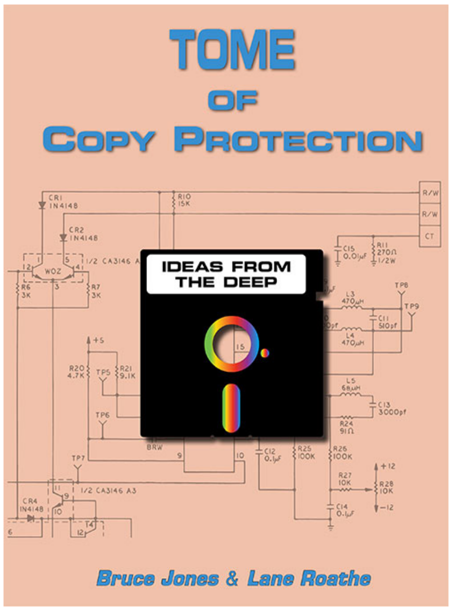
Call–A.P.P.L.E.™

World's Largest Apple User Group *Since 1978* www.callapple.org



TOME OF COPY PROTECTION

Technical Errata for First Printing

Compiled August 2018

Changes are Bold

Page 20 – Half Tracks

```
* BE5A: 0A C9 22 69 00 20 6B BE 4E 78 04 60 N 3D0G <RETURN>
```

Page 22 – Quick Sector Editor

This table upon using "**CALL 36864**" will go to slot 6, drive 1. Use buffer $9000, read-in track 0, sector 1, store 3 bytes starting at $3C of the form "$3C: 09 09 09", and then write the sector back out to disk.

Page 26 – Write Program

```
821B: 48 delay1
```

Page 27 – Write Program Listing alignment & Comment 9

```
8243: 18          wnib9 clc          2    33
8244: 48          wnib7 pha          3    34
8246: 9D 8D C0    wnibl sta $C08D,x  5    36
```

Thus, after the "sta" instruction, the logic state machine will perform a **"lsl"** on the data latch every 4 cycles. The bit that would normally become the carry bit becomes the bit that is written to disk. It can be seen now that if the latch is not loaded with another "sta" instruction the latch will continue to be **"lsl"** with zeroes being written to the disk.

Page 29 – Write Program

29. A delay of **32 cycles** is not afforded the last $FF because its purpose was to provide a smooth transition between the last bit of the $AA and the first bit of $FF.

Page 41 – Diagram 4-3

```
Gap 0 (Track Gap): Sync $FFs ~80
```

2. The program then writes **80** sync $FF's (10-bits).

Page 42 – DOS 3.3 Disk Initialization Steps

5. After these sync bytes a prologue of "**D5 AA AD**", sometimes called the data header, is written before actual data is encoded onto the disk. When a block of data (**342 bytes** + checksum, or 343 total bytes) has been written, it is followed by the epilogue "$DE $AA $EB".

Page 43 – DOS 3.3 Disk Initialization Steps

8. But the **epilogue** of the last data field will tend to overwrite some of these bytes.

Page 46 – Sector Gap Write Routine

```
BC79: A9 D5         lda  #$D5        (2);begin to write $D5 $AA $96
BCE3: 29 0F         and  #$0F        (2);perform modulo 15 on acc.
BCFA: D5 F7 AD AB   hex  D5F7ADAB
```

Page 52 – Forms of Nibble Count Classifications

Type A.1: There is also no incidence of overwrite areas, that is, data areas where the bits of one byte overwrite the bits of another byte. **This is often referred to as a "perfect track" protection scheme.**

Page 54 – Forms of Nibble Count Classifications

Type B.2.a: Moved paragraph to end of "Type B.2" above, and deleted "Type B.2.a"

Type B.1.b: Renamed second instance of "Type B.1.b" to "**Type B.2.a**"

Taking B.2 one step farther, the protection can involve checking the track gap not only for the number of bytes but by type, i.e., if the gap is composed of one-hundred, sync $FD's then any aberration will cause a read error.

Thus with Types B.2, the formatting of the protection is extremely simple and unless the user is aware of what is being done, they are virtually impossible to copy. In order to be copied, the user must instruct the bit-copier to use a different gap as the write buffer.

Page 55 – Functional Code (second to last paragraph)

GEN.CALLER is not used by NEW.INSTALL when formatting tracks for the various versions. It is only used when a protected disk is booted. GEN.CALLER's **entry point is located at $BEB0** which is part of the initialization routine of DOS 3.3.

Page 56 – Fourteen Works of Art

This section describes the program NEW.INSTALL which allows you to protect disks using various nibble count methods. Because the disks protected actually have small programs attached to do the writing and checking of the protection, each version herein is referred to as a program. **The source code for the write and read programs is on pages 153 to 250.**

Page 57 – VERSION 4

(renamed to "BUFFER WRITER", moved before "VERSION 1 - 14" on page 56)

These versions require a program that will write out data contained in a data buffer. The BUFFER WRITER, such a program, has three levels. **The source code is on page 253.**

Page 56 – VERSION 1 - 14

When selecting any program version, the user is presented with a standard pattern of messages. First, the program asks for insertion of master disk. If a mistake is made in selecting that version, **the next message gives user a chance to abort.**

Page 56 – VERSION 1 (end of last paragraph, etc)

Just to make sure the track is read correctly, **five bad reads** are allowed. **Remember, perfect tracks are hard to write, and require a very stable drive speed along with lots of retries.**

Perfect Tracks

(Moved this last paragraph from Version 3 here, and added a new paragraph)

One item that might not be clear is how **perfect tracks (tracks with no overwrite areas)** are created. When the write mode is terminated with the read mode, the drive head will continue to generate unpredictable pulses for at least 4 cycles. Thus, if the user terminates writing at a 36-cycle interval after the last byte is written then the last byte will have the best chance of being written correctly. Besides this, the user must rewrite continually until the last byte written falls on the byte interval of the data already present on the disk. If one type of byte is involved, not that many rewrites must be done. If more bytes are involved, the chances of creating no overwrite are very slim. For example, writing a track of $FF's is easier than writing a track of composed of only "$FF $FD $AD".

The hardware required to create a commercial floppy disk with a perfect track is very expensive, while using normal hardware to write a disk with a perfect track takes a lot more time than with normal disks. Both result in a lot of failures, where the written floppy will not pass its copy protection check. When trying to manufacture thousands of floppy disks for retail distribution, this results in a significant increase in the cost of the final product. For this reason, no known commercial software uses a perfect track protection scheme.

Pages 58-61 – Renamed VERSION 5-12 to VERSION 4-11 (5 became 4...)

Page 58 – VERSION 5 (renamed to "VERSION 4", first paragraph)

This program uses Type A.3, whose structure was covered **earlier in this chapter**. In review, the target track is formatted with the following byte pattern where each byte represents one page (256) of bytes.

Page 58 – VERSION 6 (renamed to "VERSION 5", first paragraph & third paragraph)

This program uses Type B.1.a. Because 4-4 encoded data has not been fully covered this program will use 24 pages of $FD's to simulate data. To be more specific, first **32 pages** of $FF's are written to erase all previous data on the disk.

The read program located at $AE8E looks for the header to make sure **the track** contains valid data.

Page 59 – VERSION 7 (renamed to "VERSION 6", last paragraph)

The read program located at $AE8E will first look for the address header to ensure the **track** contains valid data. The actual count is obtained by looking for sector zero and counting the number of bytes that occur before it is found again.

Page 60 – VERSION 8 (renamed to "VERSION 7", last paragraph)

The read program will check the integrity of the track gap, that is, making sure it has 110 $FF's. It does this by checking all 16 sector gaps. One of these sector gaps is also the track gap. The program checks each gap to see if it is composed of 110 $FF's. **If after 20 gaps (sectors)**, the correct one is not found, the program generates a read error. The read/write program from **VERSION 6** is used to format the target track.

Page 60 – VERSION 9 (renamed to "VERSION 8", first paragraph & last paragraph)

The track formatted in **VERSION 6** can be copied if the user configures a bit-copier to use another gap as the write buffer.

This delay loop is the only difference between the read program of **VERSION 6** and **VERSION 7.**

Page 61 – VERSION 11 (renamed to "VERSION 10")

This program relies on the fact that bit-copiers tend to set the track start at the beginning of the largest sync gap, **and the $96 in step 6 will be included in that gap**. It formats the gap with:

5. 10 sync $FF's **(beginning of gap)**

Page 61 – VERSION 12 (renamed to "VERSION 11")

These are the transient bit-insertion routines from Chapter 6. An error is reported by rebooting instead of sounding the bell. The read program has no check for disk validity.

Page 61 – VERSION 12 (added this new version)

This is a modified version of the invalid bytes code from Chapter 7. An error is reported by rebooting instead of sounding the bell. The read program has no check for disk validity.

Page 62 – Version 13

to:

```
$B93E: 85 FC         sta $FC
```

Page 63 – Examining Nibble Counters in General (fifth paragraph)

The counters in this chapter have some failsafe measures to avoid this by making sure the counts check **numerous times** while writing and then reading the disk more than once when checking for originality.

Page 64 – Examining Nibble Counters in General (listing alignment)

```
test        sei           ;set interrupt
            tax
            cli           ;clear interrupt
```

Page 70 – Diagram 6.2

NOTE: **Numbers represent the number of bits in a byte. Arrows represent where bits are skipped to fit the stream.**

Page 71 – Modifying DOS

The following modifications allow you to test all four cases using DOS 3.3. Slight adjustment of the functional code has been made to be compatible with DOS 3.3, but all **timings remain** the same.

Page 72 – All Cases & Case One

```
B6BD: 4C B9 B8    :1     jmp $B8B9     ;write byte
B8A9: 20 B9 B6           jsr $B6B9     ;add two zeroes to $AB
```

Page 79 – SECTOR NUMBER

```
Example 2.  Take: a b c d e f g h
```

Page 84 – Read Code (added "EXIT" line)

```
EXIT        equ $E000    ;change to your program's exit point from this code

*- ADJUST CODE FOR SUBSEQUENT READ -*
```

Page 85 – Read Code (top of listing alignment & "jmp" change)

```
rd3         and #$7F     ;strip high bit
rd0         sta $4000,x  ;store byte
rdfix       bit $A1      ;time delay for 9/10 sync

*- SHIFT BUFFER -*

            bpl shift1   ;$80
            jmp EXIT     ;exit to caller
```

Page 86 – Read Code (added "EXIT" line & "adj" alignment)

```
EXIT    equ $E000    ;change to your program's exit point from this code
```

```
*- ADJUST CODE FOR SUBSEQUENT READ -*
```

```
adj     sta rd2+1    ;fix code for 23 or 27 cycle read
```

Page 87 – Read Code

```
*- SHIFT BUFFER -*
```

```
        jmp EXIT     ;exit to caller
```

Page 89 – Mechanics (last paragraph)

Referring to Chapter 6 on Bit-Insertion, a delay of 35 cycles can be calculated (**8*4+4*1-1**).

Page 91 – Diagram 9-1

Both "35 Cycle Delay" arrows end immediately **after** the "Read 2" block.

Page 95 – Diagram 9-2

The "23 Cycle Delay" arrow ends immediately **after** the first "8 bits?" block.

Page 107 – Switching Code

```
        beq NORMAL    ;if yes, normalize
        lda #<CHECK   ;else, set up write patch
        sta PATCH+1
        lda #>CHECK
```

```
PATCH   ldy #$E5
:1      sta $B860
```

Page 108 – LOCKIT Addendum

```
INTERNAL POINTS:
$B83E = Sync Byte Used
```

Page 116 – Moving the Drive Head

```
lda  Ph0off,x       ;select appropriate magnet on/off latch
```

Page 122 – Enter Calculator Mode by Pressing "C" – Exit With Null String

```
LIMITS:  $FFFF = 65535
```

Page 124 – ALG.xxx (first paragraph)

This assumes of course that the file loads at $4000, has the instruction **JMP $806** here, and that the rest of the page is 2 byte offsets into the file (one offset for each alg.) with $00's as fillers. Example:

```
$4000: 4C 06 08 01 00 02 00 00 00 00 00 00 00 00 00
```

Page 126 – The Algorithm Toolkit (third to last paragraph)

"In the example, **alg# 50** has passed 5 bytes worth of local parameters (the values 01 through 05). If **alg# 50** is successful the analysis is complete, otherwise control will pass to the 2nd alg# in the program."

Page 152 – *=TEST TRACK

```
CLI         ;CLEAR INTERRUPT FLAG
```

Page 229 – Read Program – Version Four (between the last two]LOOP)

```
CMP #$DD    ;FIND FIRST $DD
```

Page 245 – Invalid Nibbles Read – Version Twelve (renamed title)

Invalid **Bytes** Read – Version Twelve