THE PRINT SHOP COMPANION

THE Print Shop COMPANION

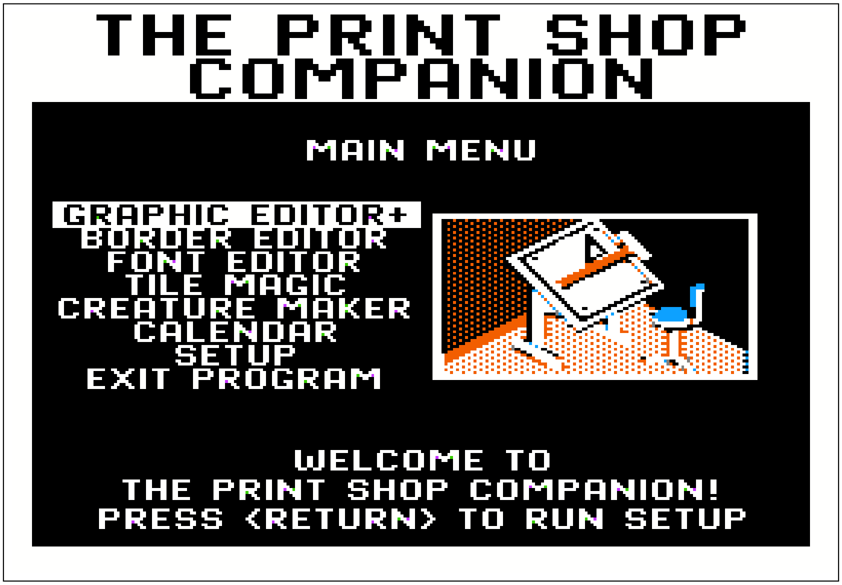
CREATES CALENDARS, FONTS, AND BORDERS FOR THE PRINT SHOP

BY ROLAND GUSTAFSSON

Brøderbund Software®

2017-06-17

Contents


THE PRINT SHOP COMPANION

MAIN MENU

GRAPHIC EDITOR+
BORDER EDITOR
FONT EDITOR
TILE MAGIC
CREATURE MAKER
CALENDAR
SETUP
EXIT PROGRAM

WELCOME TO
THE PRINT SHOP COMPANION!
PRESS <RETURN> TO RUN SETUP

```
--------The Print Shop Companion-------
A 4am crack                  2017-06-17
---------------------------------------
```

```
Name: The Print Shop Companion
Version: 1.2
Genre: graphics
Year: 1985
Credits: Roland Gustafsson
Publisher: Broderbund Software
Platform: Apple ][+ or later (64K)
Media: single-sided 5.25-inch floppy
OS: custom
Previous cracks: none (of this version)
```

Broderbund Software PRESENTS...

THE PRINT SHOP™ COMPANION

COPYRIGHT 1985
BY ROLAND GUSTAFSSON

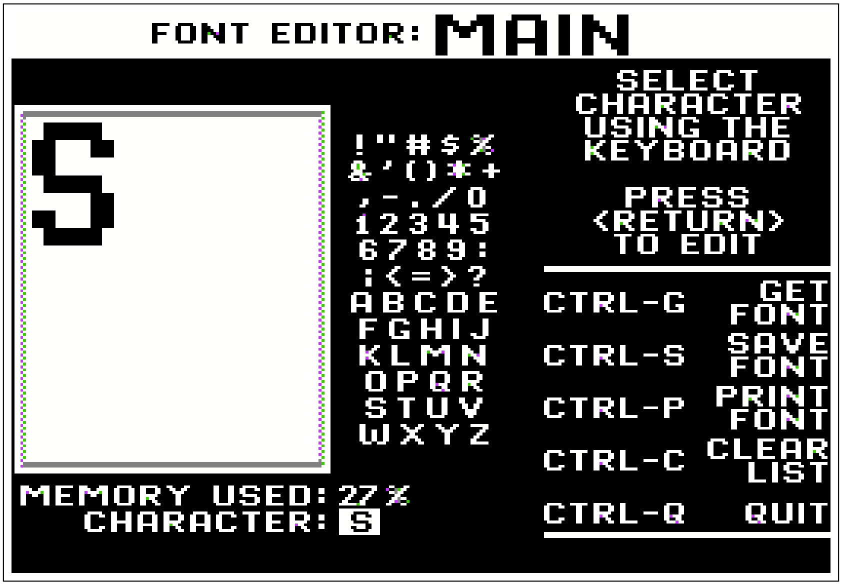
ALL RIGHTS RESERVED

Chapter 0
In Which Various Automated Tools Fail In Interesting Ways

```
COPYA
  read error on last pass

Locksmith Fast Disk Backup
  unable to read track $22
  copy displays title screen then hangs
    on main menu

EDD 4 bit copy (no sync, no count)
  read error on track $22
  copy displays title screen then hangs
    on main menu
```



FONT EDITOR: MAIN

SELECT CHARACTER USING THE KEYBOARD

PRESS <RETURN> TO EDIT

MEMORY USED: 27%
CHARACTER: S

```
Copy ][+ nibble editor
  track $22 is quite unusual, with
  repeated sequences of $D4 $D5 $DE $D4
  and other nibbles, with and without
  timing bits

                 --v--

   COPY ][ PLUS BIT COPY PROGRAM 8.4
(C) 1982-9 CENTRAL POINT SOFTWARE, INC.
---------------------------------------

TRACK: 22  START: 1800  LENGTH: 3DFF

1828: 9A 9A BB DA 97 BE BD D5+  VIEW
1830: FF+CB+A6+AE F6 A5 D7 DF
1838: AD+FF+A5+D4+D5 DE D4 BB
1840: FA F5 AA+FF+BD A9 AB BD
1848: A9 EF EF D6 FF+FF+FF+D4
1850: D5 DE D4 BA FE F5 AA AA+
1858: FF+D4 D5 DE D4 BA FF F5
1860: AA AA AA AA+D4+D5 DE D4
1868: BB FE F5 AA AA AA FF+A9

---------------------------------------

  A  TO ANALYZE DATA  ESC TO QUIT

  ?  FOR HELP SCREEN  /  CHANGE PARMS

  Q  FOR NEXT TRACK   SPACE TO RE-READ

                 --^--
```

Disk Fixer
  track 0 has a custom bootloader
  track $11 has a DOS 3.3-style disk
    catalog that lists "FONT." and
    "BORD." files (presumably fonts and
    borders loadable from the program)
  no way to read track $22 (no sectors)

Why didn't COPYA work?
  non-standard structure on track $22

Why didn't Locksmith FDB / EDD work?
  presumably there is some runtime
  protection check that triggers after
  displaying the main menu, which
  checks the structure of track $22

Next steps:

1. find runtime protection check
2. disable it
3. declare victory (*)

(*) go to the gym

Chapter 1
In Which We Will Not Be Going To The Gym Any Time Soon

Since my copy goes down a different code path than the original, I'm guessing there is a runtime protection check somewhere. One thing that all protection checks have in common is they need to turn on the drive motor by accessing a specific address in the $C0xx range. For slot 6, it's $C0E9, but to allow disks to boot from any slot, developers usually use code like this:

```
   LDX <slot number x 16>
   LDA $C089,X
```

There's nothing that says where the slot number has to be, although the disk controller ROM routine uses zero page $2B and lots of disks just reuse that. There's also nothing that says you have to use the X-register as the index, or that you must use the accumulator as the load register. But most RWTS code does, out of convention I suppose (or possibly fear of messing up such low-level code in subtle ways).

Also, since developers don't actually want people finding their protection-related code, they may try to encrypt it or obfuscate it to prevent people from finding it. But eventually, the code must exist and the code must run, and it must run on my machine, and I have the final say on what my machine does or does not do.

But sometimes you get lucky.

Turning to my trusty Disk Fixer sector
editor, I search the non-working copy
for "BD 89 C0", which is the opcode
sequence for "LDA $C089,X".

```
[Disk Fixer]
  ["F"ind]
    ["H"ex]
      ["BD 89 C0"]

                 --v--

------------ DISK SEARCH ---------------

$03/$06-$26   $03/$0D-$4F   $0A/$0D-$9B
$0D/$0B-$D8

             PRESS [RETURN]

                 --^--
```

The first match on track 3 might be my
jackpot. The protection routine appears
to start at offset $19:

```
                 --v--

T03,S06
----------- DISASSEMBLY MODE ----------
; clear page of memory at $BB00
0019:A0 00          LDY   #$00
001B:A9 FF          LDA   #$FF
001D:99 00 BB       STA   $BB00,Y
0020:C8             INY
0021:D0 FA          BNE   $001D
```

```
; turn on drive motor manually (always
; suspicious)
0023:AE F8 05          LDX   $05F8
0026:BD 89 C0          LDA   $C089,X

; look for that nibble sequence I saw
; in the nibble editor: $D4 $D5 $DE $D4
0029:BD 8C C0          LDA   $C08C,X
002C:10 FB             BPL   $0029
002E:C9 D4             CMP   #$D4
0030:D0 F1             BNE   $0023

; This subroutine just does the LDA/BPL
; loop to get the next nibble. But it
; tells me that this code is probably
; executing from $B6xx in memory.
0032:20 E5 B6          JSR   $B6E5
0035:C9 D5             CMP   #$D5
0037:D0 F5             BNE   $002E
0039:20 E5 B6          JSR   $B6E5
003C:C9 DE             CMP   #$DE
003E:D0 F5             BNE   $0035
0040:20 E5 B6          JSR   $B6E5
0043:C9 D4             CMP   #$D4
0045:D0 F5             BNE   $003C
0047:EA                NOP

; get a 4-and-4 encoded value from the
; next two nibbles
0048:BD 8C C0          LDA   $C08C,X
004B:10 FB             BPL   $0048
004D:2A                ROL
004E:85 26             STA   $26
0050:BD 8C C0          LDA   $C08C,X
0053:10 FB             BPL   $0050
0055:25 26             AND   $26
```

```
; transfer value into Y register
0057:A8             TAY

; continue below
0058:4C B5 B6       JMP   $B6B5
.
.
.
; now check for an epilogue of sorts,
; the two-nibble sequence $F5 $AA
00B5:20 E5 B6       JSR   $B6E5
00B8:C9 F5          CMP   #$F5
00BA:D0 20          BNE   $00DC
00BC:20 E5 B6       JSR   $B6E5
00BF:C9 AA          CMP   #$AA
00C1:D0 19          BNE   $00DC

; Using the 4-and-4 encoded value as
; a lookup into the page we initialized
; earlier, see if we've seen this value
; already. (We initialized all 256
; addresses with #$FF.)
00C3:B9 00 BB       LDA   $BB00,Y

; if we've found this value already,
; skip ahead
00C6:10 14          BPL   $00DC

; otherwise mark this value as "found"
00C8:A9 00          LDA   #$00
00CA:99 00 BB       STA   $BB00,Y
```

```
; loop through the array of "found"
; values and count how many unique
; values we've seen
00CD:AA                 TAX
00CE:A8                 TAY
00CF:B9 00 BB           LDA   $BB00,Y
00D2:30 01              BMI   $00D5
00D4:E8                 INX
00D5:C8                 INY
00D6:D0 F7              BNE   $00CF

; have we seen #$A0 unique values yet?
00D8:E0 A0              CPX   #$A0

; yes, we're done
00DA:B0 03              BCS   $00DF

; no, loop back and read some more
00DC:4C 23 B6           JMP   $B623

; turn off drive motor and exit
00DF:AE F8 05           LDX   $05F8
00E2:BD 88 C0           LDA   $C088,X
00E5:BD 8C C0           LDA   $C08C,X
00E8:10 FB              BPL   $00E5
00EA:60                 RTS
```

--^--

Well that certainly explains why my Locksmith Fast Disk Backup copy hung -- it's looking for a nibble sequence ($D4 $D5 $DE $D4) that doesn't exist because it didn't read anything from track $22. But why can't EDD copy this? What's so special about track $22?

T

Chapter 2
In Which We See Differences But No Clarity

On my non-working bit copy (created by EDD 4), I can edit T03,S06 to insert some useful code before we start the protection check. That sector is loaded at $B600, and the code I want looks like this:

```
[S6,D1=non-working copy]

B619-   20 EB B6    JSR   $B6EB     <--
B61C-   EA          NOP
B61D-   99 00 BB    STA   $BB00,Y
B620-   C8          INY
B621-   D0 FA       BNE   $B61D
...

; all new code here --
; set reset vector to jump to monitor
B6EB-   A9 69       LDA   #$69
B6ED-   8D F2 03    STA   $03F2
B6F0-   A9 FF       LDA   #$FF
B6F2-   8D F3 03    STA   $03F3
B6F5-   49 A5       EOR   #$A5
B6F7-   8D F4 03    STA   $03F4

; reproduce original code from $B619
B6FA-   A0 00       LDY   #$00
B6FC-   A9 FF       LDA   #$FF
B6FE-   60          RTS
```

Now I can press <Ctrl-Reset> and jump to the monitor while the protection check is running.

```
*C600G
...protection check runs and hangs...
<Ctrl-Reset>
```

Now I'm in the monitor and have unfettered access to all of memory. Let's look at the buffer at $BB00 and see which encoded values are being found and which aren't.

```
*BB00.BBFF

BB00- 00 00 00 00 FF FF 00 FF
BB08- 00 FF FF FF 00 00 00 00
BB10- 00 00 00 FF FF 00 00 00
BB18- FF 00 FF 00 FF FF 00 FF
BB20- 00 00 00 FF 00 00 FF 00
BB28- FF FF 00 00 FF FF 00 00
BB30- FF 00 00 00 00 FF 00 FF
BB38- 00 FF 00 FF 00 00 00 00
BB40- FF 00 00 00 FF 00 00 00
BB48- FF 00 00 FF FF FF 00 00
BB50- FF 00 FF 00 FF FF 00 FF
BB58- 00 00 00 FF 00 FF 00 00
BB60- FF FF 00 00 FF FF 00 FF
BB68- FF FF FF FF FF FF 00 00
BB70- FF 00 00 FF FF 00 00 00
BB78- FF FF 00 FF FF FF 00 FF
BB80- FF FF FF FF FF 00 00 FF
BB88- FF FF 00 00 FF FF 00 FF
BB90- FF 00 FF 00 FF FF FF FF
BB98- 00 FF 00 FF FF 00 00 FF
BBA0- FF 00 00 00 FF FF 00 FF
BBA8- FF FF 00 FF FF 00 00 00
BBB0- FF 00 00 FF FF 00 FF FF
BBB8- FF 00 00 FF FF FF 00 00
BBC0- FF 00 00 00 FF FF 00 FF
BBC8- FF FF FF FF 00 00 00 00
BBD0- FF FF 00 00 FF 00 00 FF
BBD8- FF 00 00 FF FF FF FF FF
BBE0- FF FF 00 FF FF 00 00 FF
BBE8- FF FF 00 00 00 FF 00 FF
BBF0- 00 FF 00 FF FF FF 00 FF
BBF8- 00 FF 00 FF FF 00 00 00
```

I can also reuse the counting routine (at $B6CA) to find out how many values we're finding.

```
*B6CAL

B6CA-   A9 00         LDA   #$00
B6CC-   EA            NOP
B6CD-   AA            TAX
B6CE-   A8            TAY
B6CF-   B9 00 BB      LDA   $BB00,Y
B6D2-   30 01         BMI   $B6D5
B6D4-   E8            INX
B6D5-   C8            INY
B6D6-   D0 F7         BNE   $B6CF

; count of zero values is in X register
B6D8-   8A            TXA

; print it (as a hex value) and exit
B6D9-   4C DA FD      JMP   $FDDA

*B6CAG
7E
```

The minimum number of unique 4-and-4 encoded values required to pass the protection is #$A0 (160), but we're only seeing #$7E (126). So we're way short.

Why are we missing so many values?

To find out why, I want to know exactly which values I'm missing. I would really like the original disk to break into the monitor after it successfully completes this protection check, so I can see what the array at $BB00 looks like. Of course I'm not going to modify my original disk (NEVER EVER DO THAT). But I can add some code on my non-working copy to wait for a keypress before starting the protection check, then swap in the original disk before continuing.

That code, still on T03,S06, looks like this:

```
; wait for keystroke
B6EB-   2C 10 C0    BIT   $C010
B6EE-   AD 00 C0    LDA   $C000
B6F1-   10 FB       BPL   $B6EE
B6F3-   2C 10 C0    BIT   $C010

; original code (from $B619)
B6F6-   A0 00       LDY   #$00
B6F8-   A9 FF       LDA   #$FF
B6FA-   60          RTS
```

And after succeeding, I want to jump to
the monitor:

```
B6D8-   E0 A0          CPX   #$A0
B6DA-   B0 03          BCS   $B6DF
B6DC-   4C 23 B6       JMP   $B623
B6DF-   4C 59 FF       JMP   $FF59     <--
```

Rebooting my non-working copy, it gets
as far as the main menu and pauses to
wait for a key (at $B6EB). Ejecting my
copy and inserting the original, then
pressing a key, it runs the protection
check on the original disk and breaks
to the monitor with a very satisfying
beep.

<beep>

Now I can see what the $BB00 buffer
looks like after a successful run on an
original disk:

```
*BB00.BBFF

BB00- 00 00 00 00 FF 00 00 FF
BB08- 00 FF 00 00 00 00 00 00
BB10- 00 00 00 FF 00 00 00 00
BB18- FF 00 00 00 FF FF 00 FF
BB20- 00 00 00 FF 00 00 00 00
BB28- FF 00 00 00 FF FF 00 00
BB30- FF 00 00 00 FF 00 00 FF
BB38- 00 FF 00 FF 00 00 00 00
BB40- 00 FF FF 00 FF 00 00 00
BB48- FF 00 00 00 FF 00 00 00
BB50- FF 00 FF 00 00 FF 00 FF
BB58- 00 00 00 FF 00 FF 00 00
BB60- FF FF FF 00 FF 00 00 FF
BB68- 00 00 FF FF 00 00 00 00
BB70- 00 00 00 FF 00 00 00 00
BB78- 00 FF 00 FF 00 00 00 00
BB80- FF FF 00 FF FF 00 00 00
BB88- FF 00 00 00 FF FF 00 FF
BB90- 00 00 FF 00 00 FF 00 FF
BB98- 00 FF 00 FF 00 00 00 FF
BBA0- FF 00 00 00 FF FF 00 00
BBA8- FF FF 00 FF FF 00 00 00
BBB0- FF 00 00 FF FF 00 FF FF
BBB8- FF 00 00 00 FF 00 00 00
BBC0- FF 00 FF FF 00 00 00 FF
BBC8- FF FF 00 FF FF 00 00 00
BBD0- FF FF 00 00 FF FF 00 00
BBD8- 00 00 00 00 FF FF 00 FF
BBE0- 00 FF 00 FF FF 00 00 FF
BBE8- FF FF 00 00 00 FF 00 FF
BBF0- FF FF 00 00 00 FF 00 FF
BBF8- 00 00 00 FF FF FF 00 00
```

Superimposing the two arrays, I can see
which 4-and-4 encoded values are being

found on the original but never found on my non-working bit copy.

```
BB00- .. .. .. .. .. 00 .. ..
BB08- .. .. 00 00 .. .. .. ..
BB10- .. .. .. .. 00 .. .. ..
BB18- .. .. 00 .. .. .. .. ..
BB20- .. .. .. .. .. .. 00 ..
BB28- .. 00 .. .. .. .. .. ..
BB30- .. .. .. .. .. 00 .. ..
BB38- .. .. .. .. .. .. .. ..
```

And so on. No obvious pattern emerges.

Hmm.

I'm going to switch tactics and look at the first value that appears on the original disk but not on my bit copy.

4-and-4 encoded values are not big mysteries. Even unprotected disks use them in the address field of every sector. There were published lookup tables, even in the 80s. Here's an excerpt of one from an old Copy II Plus manual:

Dec	Hex	Binary	4-and-4
0	$00	00000000	AA AA
1	$01	00000001	AA AB
2	$02	00000010	AB AA
3	$03	00000011	AB AB
4	$04	00000100	AA AE
5	$05	00000101	AA AF
6	$06	00000110	AB AE
7	$07	00000111	AB AF
8	$08	00001000	AE AA
9	$09	00001001	AE AB

The first value missing from my bit
copy was 5 (in location $BB05), so
let's search the original disk for the
nibble sequence "AA AF F5". That's the
4-and-4 encoded value ("AA AF"), plus
the first nibble of the expected
epilogue ("F5 AA").

```
                 --v--

TRACK: 22  START: 1800  LENGTH: 3DFF

2AC8: D4 AB AA F5 AA AA AA AA   VIEW
2AD0: D4 D5 DE D4 AB AB F5 AA
2AD8: FF FF 9A 9A BB DA 95 AB
2AE0: BD D5 FF 9B D4 D5 DE D4
                  ^^^^^^^^^^^
2AE8: AA AF F5 AA FF FF A6 AE  <-2AE8
      ^^^^^ ^^^^^
2AF0: F6 A5 BA EF B5 FF FF B5
2AF8: D5 DE D4 AB AF F5 AA FF
2B00: FF 9A BB DA 95 D5 BD D5  FIND:
2B08: FF EC D4 D5 DE D4 AE AB  AA AF F5

                 --^--
```

I highlighted the three relevant
sequences, starting at offset $2AE4:

```
  - D4 D5 DE D4 (prologue)
  - AA AF       (4-and-4 encoded value)
  - F5 AA       (epilogue)
```

In that same screenshot, we can see
evidence of several other values:

  - "AB AA" (2) at offset $2AC9
  - "AB AB" (3) at offset $2AD4
  - "AE AB" (9) at offset $2B0E

So this confirms what running the
protection code already told me: the
original disk contains the group of
(prologue)(4-4 encoded value)(epilogue)
where the encoded value is 5.

Now let's see what my non-working bit
copy looks like.

[S6,D1=non-working copy]

```
                 --v--

TRACK: 22  START: 1800  LENGTH: 3DFF

2430: AA FF D4 D5 DE D4 AA AB   VIEW
            ^^^^^^^^^^^
2438: F5 AA AA AA D4 D5 DE D4
                  ^^^^^^^^^^^
2440: AB AA F5 AA AA AA D4 D5
                        ^^^^^
2448: DE D4 AB AB F5 AA FF B3
      ^^^^^
2450: A6 AE F6 A5 AA EF B5 FF  <-2450
2458: FF A9 AB BD A9 AA BF D6
2460: FF FF D4 D5 DE D4 AB AE
            ^^^^^^^^^^^
2468: F5 AA FF FF A6 AE F6 A5  FIND:
2470: BA FF B5 96 FF D4 D5 DE  D4 D5 DE
                     ^^^^^^^^

                 --^--
```

Highlighting prologues ("D4 D5 DE D4"), we can see the track contains lots of groups with 4-and-4 encoded values:

- "AA AB" (1) at offset $2436
- "AB AA" (2) at offset $2440
- "AB AB" (3) at offset $244A
- "AB AE" (6) at offset $2466

But no group contains "AA AF" (5), anywhere in the track.

I am no closer to understanding why.

Chapter 3
Now You See It, Now You Don't

Let's return to the original disk and
examine the protection track again.

```
[S6,D1=original disk]

                 --v--

TRACK: 22  START: 1800  LENGTH: 3DFF

2F58: DE D4 AB AA F5 AA AA AA    VIEW
2F60: AA D4 D5 DE D4 AB AB F5
2F68: AA FF 9A 9A BB DA 95 AB
2F70: BD D5 FF FF B5 D5 DE D4
                  ??^^^^^^^^^^^^
2F78: AA AF F5 AA AA AA FF A6  <-2F78
      ^^^^^
2F80: AE F6 A5 BA EF B5 FF FF
2F88: 9A 9A BB DA 95 EB FD D5
2F90: FF FF 9A 9A BB DA 95 D5  FIND:
2F98: BD D5 FF FF 9A 9A BB DA  AA AF F5

                 --^--
```

Waaaaait a minute. That can't be right.
There's the 4-and-4 encoded value for 5
("AA AF", at offset $2F78), and the
epilogue after it ("F5 AA", at offset
$2F7A). But look at the prologue before
it. It should be "D4 D5 DE D4" starting
at offset $2F74, but the first nibble
is missing.

Maybe there are two copies of this
group and I just found the other one
this time? Nope, this is only instance
of "AA AF F5" on the track.

Did I insert the wrong disk? (I do have an overabundance of floppy disks, so this is always a reasonable question.) Nope, this is the same original disk I read last time.

Side note: I realize the offsets are different, but they're not relevant to this mystery. They depend on where the disk was spinning when Copy II Plus started reading the track this time. That could be anywhere; I'd expect the offsets to change every time I re-read a track.

But the nibbles themselves should be the same. That's the data on the disk. Data on the disk shouldn't change every time you read it. That's kind of the point of a storage device.

Unless...

Oh no.

Oh God.

Oh God no.

There's only one thing you can put on a disk that will change every time you read it: nothing. And by "nothing," I mean "a long sequence of zero bits." And that's what is on the original disk between each of these groups: nothing.

A bit of background. When we say a "zero bit," we really mean "the lack of a magnetic state change." The Disk II drive isn't digital; it's analog. If it doesn't see a state change in a certain period of time, it calls that a "0". If it does see a change, it calls that a "1". But the drive can only tolerate a lack of state changes for so long -- about as long as it takes for two bits to go by.

Fun fact(*): this is why you need to use nibbles as an intermediate on-disk format in the first place. No valid nibble contains more than two zero bits consecutively, when written from most-significant to least-significant bit.

So what happens when a drive doesn't see a state change after the equivalent of two consecutive zero bits? The drive thinks the disk is weak, and it starts increasing the amplification to try to compensate, looking for a valid signal. But there is no signal. There is no data. There is just a yawning abyss of nothingness. Eventually, the drive gets desperate and amplifies so much that it starts returning random bits based on ambient noise from the disk motor and the magnetism of the Earth.

Seriously.

(*) not guaranteed, actual fun may vary

It's trivial to write zero bits to a disk; just write a #$00 nibble to write 8 zero bits -- like any other 8-bit nibble. You can write whatever you want to a disk; it doesn't need to be what DOS would consider a "valid" nibble. But when you read that nibble back, the drive can't handle 8 zero bits in a row, so it will actually return some random bits. Which is why no one does that.

Returning random bits doesn't sound very useful for a storage device, but it's exactly what the developer wanted, and that's exactly what this copy protection scheme depends on. Here's why:

Bit copiers can't duplicate a long sequence of zero bits.

Why? Because that's not what they see. What they see is some random bits -- the real zero bits interspersed with phantom "1" bits. So that's what they write to the target disk. Whatever randomness they get when they read the original disk will essentially get "frozen" onto the copy.

Now, why does this matter? Let's look at the protection code again. It looks for the custom prologue "D4 D5 DE D4", then checks the 4-and-4 encoded value that follows and marks that value as "found" in the buffer at $BB00. But each prologue is preceded by a bitstream that changes every time it's read. Sometimes those random bits will align in such a way that they form two full, valid nibbles, and the next prologue will be read correctly. Other times, they will only form a partial nibble that is completed by the first few bits of the prologue, so the prologue will be missed.

As far as I can tell, the sequence of zero bits is 18 bits long. I'm not sure the exact length matters. The important part is the randomness.

Here's what's on the disk (the $D4 and
$D5 are the start of the prologue):

```
  /--00--\/--00--\/--D4--\/--D5--\
0000000000000000110101001101010 1
```

But remember, more than two consecutive
zero bits will form an "abyss" that the
floppy drive will fill with randomness.
Some of those zero bits before the
prologue will randomly transform into
"1" bits, and it'll be a different set
every time you read the disk. How does
that affect the protection check?
Here's one of many possible bitstreams
that might come out of the abyss:

```
  /--FF--\/--9B--\/--D4--\/--D5--\
00111111110011011110101001101010 1
```

That's what I saw the first time I read
the original disk with the Copy II Plus
nibble editor: the abyss coalesces into
two full nibbles ($FF $9B), then I read
the prologue ($D4 $D5 and so on).

Here's another possible bitstream that might come out of the abyss:

```
/--FF--\/--FF--\/--B5--\  /--D5--\
1111111111111111101101010011010101
```

That's what I saw (on the same disk!) the second time I read it, as shown at the beginning of this chapter. This time, the abyss coalesces into two and a half nibbles -- $FF, $FF, and a few extra bits. Those extra bits combine with the bits that are supposed to be part of the $D4 nibble, but because we're in the middle of a nibble when the $D4 bits start, we end up finishing that nibble ($B5) instead, and the $D4 nibble disappears.

The first nibble of the prologue, $D4, gets consumed by the abyss.

The second nibble of the prologue, $D5, survives unscathed, but by then it's too late. Without the full prologue, we'll skip this entire group and the 4-and-4 encoded value within it.

The protection code requires finding 160 unique 4-and-4 encoded values after a full prologue ($D4 $D5 $DE $D4). The loop at $B6CD counts the number of values it's found by incrementing the X register for every value marked "found" in the buffer at $BB00. At $B6D8, it compares X to #$A0 and branches to the success path when it's found enough unique values. It doesn't care which values it finds, or in which order; it only cares about the total count.

If you re-read the original disk enough times, each stream of random bits will (eventually) align so the next prologue is (eventually) read correctly and enough different encoded values are (eventually) marked as "found." You'll miss some of the prologues each time you read the track, but you'll (eventually) find them all as the random bits fluctuate. So the check only passes after some number of reads of a nondeterministic bitstream that sometimes (but not always) corrupts the data after it.

But bit copiers don't preserve long streams of zero bits. Instead, they write out whatever phantom "1" bits they find. On a copy, you'll also miss some percentage of prologues -- and thus skip over some percentage of the 4-and-4 encoded values -- each time you read the track. But that percentage will never increase no matter how many times you read it. No more randomness. No more "eventually."

God, I hate physical objects.

Chapter 4
In Which The Solution Is Trivial Except When It Isn't

None of this affects how I can make my copy work. I just need to patch the protection routine at $B619 so it returns gracefully and unconditionally.

Making a fresh copy (no more hacks), I can turn to my trusty Disk Fixer sector editor and put an "RTS" at the start of the protection code:

```
T03,S06,$19: A0 -> 60
```

```
]PR#6
...boots to main menu, at last...
```

Except...

In the course of testing, I discovered one final small slight really hardly noticeable little problem...

It doesn't print.

I mean, it prints, but it doesn't print what I want it to print. It's all just garbage. At first, I assumed this was a problem with my emulator. (I no longer have a real printer to test with.) But no. Through some emulator trickery, I made an unpatched copy "pass" the protection check by altering the CPU flow without altering the code on disk or in memory, and that copy printed just fine.

Something is detecting that I modified the protection code.

Now, there are any number of ways to
checksum a region of memory, but the
Apple II is not a fast machine, and
realistically it's probably just XORing
the bytes together. Another possibility
is ADDing the bytes together. But throw
away your modern conceptions of some
sort of cryptographic hash function.
In the 80s, any "encryption" was
probably just XOR, and any "checksum"
was probably one byte long and was the
cumulative XOR of all the other bytes.

(Story time! Sierra was notorious for
their anti-tamper checks. Games would
have a self-decrypting protection check
and a separate anti-tamper check that
XOR'd the encrypted protection code.
But the first two bytes of the code
were always the same, and word got
around of how you could defeat the
protection AND the anti-tamper check in
all their games by searching for the
first two bytes and changing them to
known values. In response, Sierra added
a second anti-tamper check that
checksummed only the odd bytes.)

What I'm saying is that I can probably
fool this tamper check without even
finding it, by applying the principle
of transitivity. I changed an $A0 byte
to $60 to defeat the protection. Now I
need to find a $60 byte and change it
to $A0 to defeat the anti-tamper check.

And there is one, in the same sector, at offset $EA: the "RTS" instruction at the end of the subroutine that starts at $B6E5 (but is no longer called since I put an "RTS" at the very beginning of the protection code to bypass the entire thing).

```
T03,S06,$EA: 60 -> A0
```

```
]PR#6
...works, and prints correctly...
```

Sometimes it's better to be lucky than good.

After extensive testing, there doesn't appear to be any further protection.

Quod erat liberandum.

```
---------------------------------------
A 4am crack                    No. 1257
------------------EOF------------------
```