COPY ME - I WANT TO TRAVEL
(HOW TO PROTECT YOUR SOFTWARE)
by Claus Brod

There’s nothing more senseless than copy protection (surprising intro for an article about self-made protection methods, isn’t it?) - For there will never be the perfect one. Nevertheless, it is a fascinating subject, anyway, and one of the most challenging, too. This article shows you how to install a particularly nasty copy protection on your disks...

To understand how to install a copy protection method you should be able to tell a disk from fish & chips (frivolous as I am, I suppose that you can manage this). Furthermore, you'll need some background information. Et voila:

A standard disk has got 80 concentrically circles on it, called tracks. Every track carries its data in specific blocks, called sectors. Between two sectors (and even inside sectors!) is a gap that the controller needs as a little spare time to relax from the hard work of reading data.

Controller: What's that? Well, simplified version first: A device that gets data from the processor, codes it, and writes it onto disk. It also reads bit streams from the disk and decodes it. To transfer data to and from the controller, the built-in DMA chip in the ST carries data from main memory to the FDC (floppy disk controller) and also from the controller to memory. The processor itself doesn't have to do this dirty work; while DMA (direct memory access) is working, the processor can do anything else (have tea or dinner with Gershwin or go shopping...). The controller sends electrical impulses to the disk drive's read/write head (very similar to the one in the ghetto blaster cassette recorder that the son of your neighbour uses to test your resistance against noise pressure). These impulses are coded using the so-called MFM method (modified frequency modulation) which packs clock bits into the data stream. This ensures that reading and writing is safe even with drives that have problems with holding the correct speed.

MORE THAN PURE DATA

On a disk you'll find more than data that you have transferred thereto by a XBIOS or BIOS call. Additionally, some control information is hidden between the data. A typical track looks like this (well, OK, it's concentric in reality; please use your imagination):

- Intro (track header or "Post Index Mark")
- (*) gap bytes (Pre Address Mark)
- 3 synchronization bytes
- address mark & sector info
- checksum
- gap bytes (Post Address Mark)
- 3 synchronization bytes
- data mark & data
- checksum
- gap bytes (Post Data Mark)
- back to (*), until all sectors are written, then:
- Gap bytes to track end

What the heck are synchronization bytes? In short: Bit sequences that tell the controller as it reads them "Go on, old fella, a new byte's starting right now". For the controller just reads a string of bits from disk without knowing exactly where a byte begins. The controller has a

dedicated department inside that only fiddles with sorting those bit combinations out of the bit stream that are meant as sync bytes (as you call them shortly).

A BUG IN THE CONTROLLER

In normal cases, this sync unit (officially called Address Mark Detector) only reacts upon the bytes $A1 and $C2 which have to be stored in a special format (MFM with missing clock bits; just to mention it for the professionals who read my articles in "ST-Computer"). Unfortunately, there is an eddy in the controller (if you want to know exactly about eddies, I recommend you "Life, The Universe And Everything" by Douglas Adams). A bit string of %000101001 can fool the sync byte detection department completely!

Whenever this bit string occurs, the controller thinks there is a sync byte and starts to synchronize (what else should it do). All the following bytes are shifted in a certain way and their bits shattered all over the place. This error only occurs when reading a complete track with all information on it (with the controller's read-track command). When reading sectors (with read-sector), the controller switches off the sync unit, so that it can read without the sync bug confusing it.

To make it clear (for the ones who only believe what they see): Suppose you want to write the following onto disk:

- FE 29 00 01 (all bytes in hex)

Reading these bytes with a read-track command you get:

- FE 14 7F FE

Strange, eh?

The error occurs with the following bytes:

- $29 and previous even byte
- $52/$53 and previous byte dividable by 4
- $A4 to $A7 and previous byte dividable by 8
- $14 and following byte's first bit (MSB) set to 1

Some other byte combinations can cause the error, too (shifts of the byte sequences above, for example).

As long as these data are packed into a sector (which you can easily read correctly using the read-sector command), everything's fine. As soon as you put $FE29 (for example) in an area unreachable by read-sector (or read-address, which reads a sector info sequence), however, you're in a mess. Naturally, copy programmers know about this quirk. Consequently, they use read-sector commands. They also know that gap bytes (which are normally $4E or $00) aren't read correctly in most cases but somehow shifted (from one to seven bits, even by "half bits"). So you cannot rely on what you're reading with read-track. Most programmers therefore suppose that gaps consist of $4E's and $00's and nothing else, basta.

DEVELOPING THE PROTECTION

Now you could insert a string like "(C) 1987 by Claus Brod" into those gap bytes. As normal copy programs don't check the contents of the gap areas, they would fail in copying this string - so you can detect a copy from within your program. Stop press! How can you check your copyright string if the controller reads something different every time you use the read-track command? In one in ten cases, approximately, you will read the correct string (perhaps never!); in all other cases you will get something that your copy protection doesn't recognize - crash down & bomb attack (very user friendly, isn't it?)...

Now those mysterious sync bytes come into action. These bytes make the controller read the following stuff correctly (in most cases). Normally, a track starts with approx 60 gap bytes of $4E's. Instead, we write:

- 4E 4E 4E A1 A1 A1 copyright string 4E 4E ...

This synchronizes the controller in a read-track command so that it can at least read the copyright string correctly. Of course, the copyright string must not contain any bytes or byte sequences that may fool the sync department.

In your copy protection routine you read a complete track and look for your copyright text. If it's there, it's the original disk (or someone's got a very very good copy program). If not, something went into wrong channels. I tried to copy this protection method with several copy programs - all of them were a complete loss except one (I won't tell you which one, no, no, no). So I had to refine the protection. Now I use the "forbidden" sync byte $29 with a previous even byte to synchronize the controller. Synchronization is shifted by half a byte after $29, so we have to add an ordinary $A1 sync byte to make the sun shine again. These two bytes are read as $14 $0B which no copy program recognizes as correct sync sequence. The following byte sequence (your copyright string) is now interpreted correctly with every Read-Track command. And there you are: the perfect protection (at least until someone comes up with a hyper-ultra-jolly-good copy program).

MEN AT WORK

Let's work: How do you program such a protection? To give you full information about programming the controller and DMA chip and whatever relates to this problem I would have to write an article that spans three or four issues of ST NEWS, leaving just enough space for the editorial and nothing else. In order to cut this never-ending story short, you're presented a prefab routine from my software lab which you can use to implement the protection method.

In the PROGRAMS folder, you'll find a GfA Basic program called PROTECT.BAS, which creates a machine routine in a string (for the real big chucks, there is an assembler listing on this disk). This machine code reads and writes an ordinary 9-sector track 41 - nearly ordinary: the sync byte combination followed by a copyright message is copied into the gap before the first sector. Why do I write track 41? 41 in hex gives $29; this byte occurs in every sector intro of track 41 and therefore confuses the read-track command. Therefore, this track is never read correctly; all the copy programs I know switch to sector mode on this track - reducing the probability that someone can copy our protection (which is only possible by reading the whole track, analyzing it, and writing it back with some corrections).

HOW TO USE "THE PROTECTOR"

Just format a disk (single or double sided, doesn't matter) with the DESKTOP format utility and then start the routine in PROTECT.BAS. Choose option 2 to write a protected track 41. You are asked a password string that you want to write onto disk. Afterwards, you copy data and programs to this disk; among them the program to be protected. Option 1 of "The Track 41 Protector" reads a complete track 41 from disk and searches for a password that you can freely choose. The password can have up to 50 characters. If you need more, adjust Gap1$ in Procedure Mktrk.

How do you include the routines in your own programs? GfA BASIC programmers are better off this time; it should be no problem, however, to adapt the routines for FASTBASIC or whatever. To call the machine code routines, you have to poke values into the routine's parameter field. If the program starts at location START, you have to store a mode longword into location START+2 and the address of a track buffer into START+6 (see BASIC listing). A mode of '0' activates the Read-Track routine, '1' makes the program write a track from the buffer referred to by the address pointer in START+6. Assembler programmers are supplied

with the source code (for AS68) on this disk. The routine must be called with 'bsr' or 'jsr' as it terminates with a simple 'rts'. It is fully relocatable (for it had to be transferred into a BASIC program) and PC-relative.

You have to set up the track to be written before calling the machine code. In Procedure Mktrk you'll find all the information you're looking for. To satisfy the curious (curiosity kills the cat, fellas!): Here is another version of the table above including the codes that the controller needs to create a track.

- Intro: 60 bytes of $4E
- (*) gap bytes: 12 bytes of $00
- sync bytes: 3 bytes of $F5
- address mark: $FE
- sector info: track number, side number, sector number, size (0=128 bytes, 1=256 bytes, 2=512 bytes, 3=1024 bytes)
- checksum: $F7
- gap bytes: 22 bytes of $4E
- 12 bytes of $00
- sync bytes: 3 bytes of $F5
- Data mark: $FB
- Data: 512 bytes (lower than $F4!!!)
- checksum: $F7
- gap bytes: 40 bytes of $4E
- back to (*), until all sectors are written, then:
- gap bytes to track end (normally 1400 bytes of $4E, which is more than enough to fill the track)

You may wonder why sync bytes are written as $F5 but read as $A1. The reason is that the controller needs an own control "language" when writing a track. Bytes exceeding a value of $F4 are special control bytes that are written in other formats than ordinary bytes. $F5, for example, is a sync byte of $A1 with certain clock bits missing. $F7 writes a checksum onto disk (consisting of two bytes). $FB and $FE announce data and sector info, respectively. By the way, this control language is the reason why you must not format your disks (using the XBIOS call) with certain "virgin" values (the virgin word's bytes must not exceed $F4).

Some additional comments concerning the BASIC program and its usage: '0' in the main menu terminates the program (what else did you expect?). '3' shows the contents of the track buffer in both hex and ASCII; to stop output, press any key; then press 'X' to leave the routine or any other key to continue. The track buffer (in this program buf$) must be at least 7K. If you want to use 10-sector disks you only have to change the "FOR T=1 TO 9" loop in Procedure Mktrk into "FOR T=1 TO 10".

If you want to know more about disk drives you are given three possibilities: read my "Floppyspielereien" articles in the German computer magazine "ST Computer" or buy my book about floppy programming (out in late fall) or write to ST NEWS. If there are some readers out there who are interested to read more about disk drives in ST NEWS: Write to ST NEWS and reveal your innermost wishes... depending on demand I will continue to write about floppy programming in ST NEWS from time to time.

Claus Brod
Am Felsenkeller 2
D-8772 Marktheidenfeld
West Germany