Smarty and the Nasty Gluttons

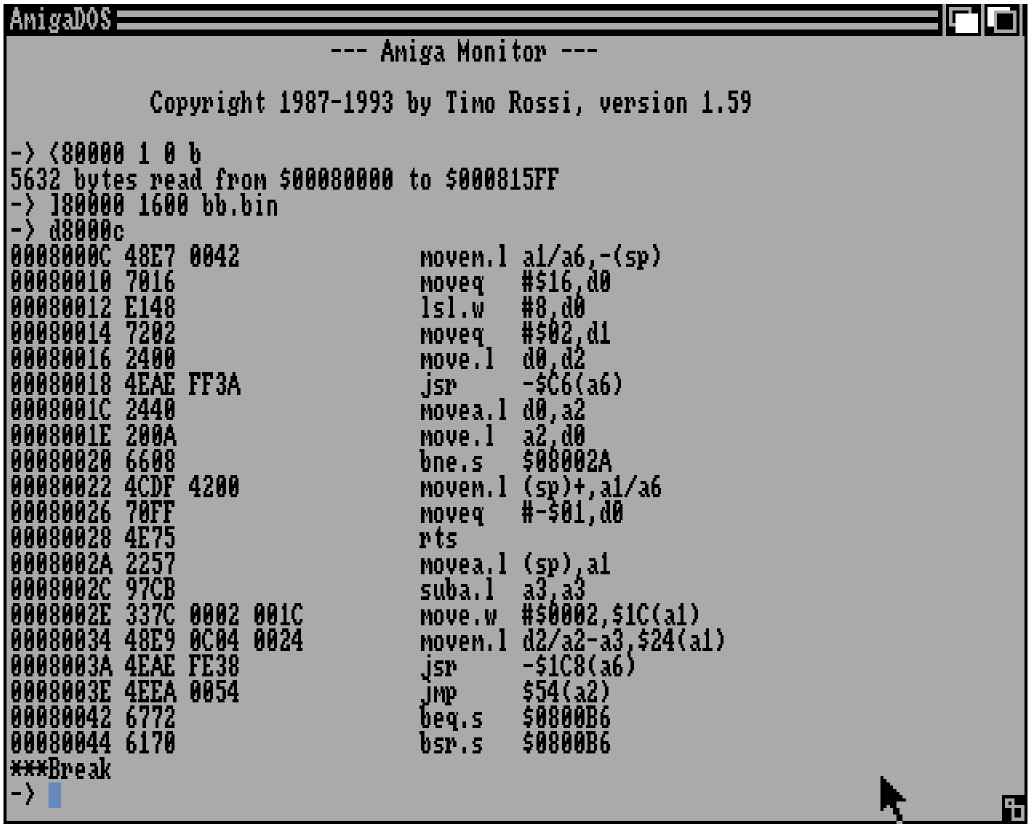
For this tutorial/monolog you would need:

- My original copy of the game preview ;)
- A real Amiga hardware with at least 1MB chip and one or more floppy drives
- RossiMon
- AsmOne1.02+ or 1.02
- A decent cruncher
- An empty OFS formatted disk
- FS-UAE to make nice screen captures, which also explains some inconsistency with the disk drive numbering ;)

Just a quick note to readers. My A600 has Kickstart 3.1 and 2MB CHIP RAM. All the work was done on it using the above mentioned tools. The screen captures were then done using FS-UAE with A500 with 1MB CHIP RAM and 1MB FAST setup just to test the final release under kickstart 1.3 and make nice screen captures.

As a background, a long ago I worked on a game called **Smarty and the Nasty Gluttons** ;) My contribution was the boot up procedures, disk system, compression and a disk duplication system (i.e. use stock Amiga to copy the master disk simultaneously up to four drives). The game got never released due unfortunate events. With the permission of the current owner of the rights (many thanks Eero!) I attempt to crack a playable game preview that I found from a box of old disks that used to belong to one of the game lead developers and make a nice release out of it. Having written the disk system gives me an upper hand but it was over 20 years ago and I actually never had a playable game previews before. I did not have to since the toolset given to developers was made to be fully usable without my intervention.

The cracking method used in this tutorial is going to be very low tech and oldskool – since that's where I am stuck at. My method has no room for ActionReplay or UAE but loads of resets and handwritten notes on a paper. Let's start Timo Rossi's Amiga Monitor (ahh, such a great tool.. and r.i.p Timo) and insert the game disk into DF0:. As expected we get nasty "read-write error" noises and eventually a read error, which indicates us the disk has some kind of a non-DOS format. In order the game to boot at least the track 0 on the lower side, which also contains the boot block, has to be in OFS or FFS format. I'd assume the disk has OFS format but it is not important since the final crack will be in OFS formatted disk in any case. First, I read the entire track 0 into CHIP memory address $80000 and write it back to another disk I have as a DOS file. Next we'll have a look at what there might be on the boot block. The code seems to allocate one track worth of CHIP RAM and load the entire track 0 into RAM and then jump to the allocated memory address + $54. This is rather normal stuff.



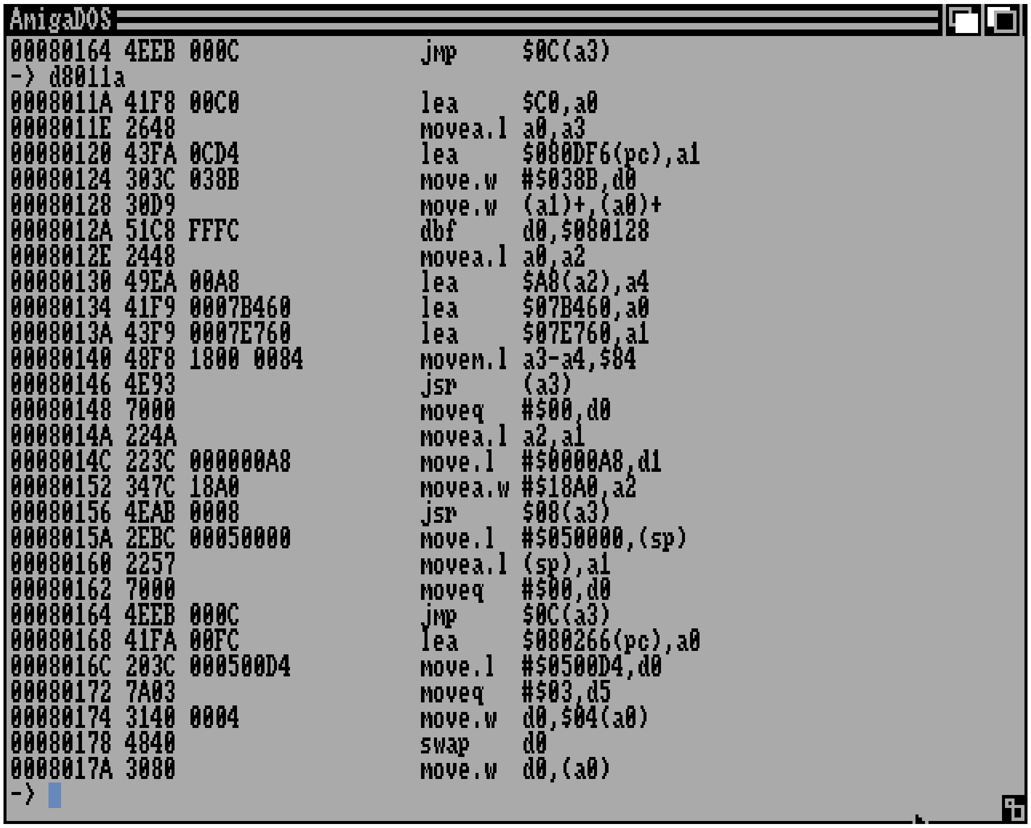
```
                          --- Amiga Monitor ---

           Copyright 1987-1993 by Timo Rossi, version 1.59

-> (80000 1 0 b
5632 bytes read from $00080000 to $000815FF
-> ]80000 1600 bb.bin
-> d8000c
0008000C 48E7 0042                movem.l a1/a6,-(sp)
00080010 7016                     moveq   #$16,d0
00080012 E148                     lsl.w   #8,d0
00080014 7202                     moveq   #$02,d1
00080016 2400                     move.l  d0,d2
00080018 4EAE FF3A                jsr     -$C6(a6)
0008001C 2440                     movea.l d0,a2
0008001E 200A                     move.l  a2,d0
00080020 6608                     bne.s   $08002A
00080022 4CDF 4200                movem.l (sp)+,a1/a6
00080026 70FF                     moveq   #-$01,d0
00080028 4E75                     rts
0008002A 2257                     movea.l (sp),a1
0008002C 97CB                     suba.l  a3,a3
0008002E 337C 0002 001C           move.w  #$0002,$1C(a1)
00080034 48E9 0C04 0024           movem.l d2/a2-a3,$24(a1)
0008003A 4EAE FE38                jsr     -$1C8(a6)
0008003E 4EEA 0054                jmp     $54(a2)
00080042 6772                     beq.s   $0800B6
00080044 6170                     bsr.s   $0800B6
***Break
->
```

There seems to be an interesting code snippet starting at $80062. The code scans through the ExecBase MemList (i.e. ExecBase + $142, type = NT_MEMORY) and searches for the largest memory area that is not in the first 512K of CHIP RAM. So any FAST RAM or CHIP RAM beyond $80000 works as an available memory. The game requires at least 1M of RAM, otherwise it refuses to run (Ed: I took the memory check code and tried to use it as-is. It appears to bug in some cases with some 512K CHIP RAM only systems so I had to modify the code a bit for the crack to work properly – everything will crash'n'burn if there is only 256K of CHIP RAM btw). The code continues to turn off the system, take over the display and poke known AGA registers to make everything OCS compatible. The code starting at $800cc jumps to SuperVisor and then checks for the CPU type. If anything better than 68000 is found the VBR is set to $0. All these steps are probably unnecessary at this point of the system boot but just to be sure I guess. If the start of additional memory is zero the boot block code jumps to address $80168, where a small piece of code will decrunch (vanilla StoneCracker S404) some nice graphics and display a screen saying that at least 1MB is needed. At $80106 the additional memory and its size get stored into memory addresses $8c and $90. We need to remember those since the game itself uses these addresses to find memory areas where, for example, to load data. Both USP and SSP stacks are moved to the top of the found additional RAM. For both stacks $400 bytes are reserved.

The code at address $8011a moves the disk loader to address $c0 (we will study the loader code later in detail). The entire loader is PC-relative, which will be very useful for us later on when we start extracting files from the disk. The code at $80134 stores another two values into memory locations $84 and $88. These are important since later we will find out that address $84 holds the address of the disk loader (here $c0) and address $88 holds the first available "free memory" address after the loader. Before jumping into the loader for the first time (at $80146) registers A0, A1 and A2 get loaded with interesting values. Register A0 contains the address of the MFM buffer, A1 contains an address of a buffer for a single decoded track, and A2 contains the address of the file table. It turns out that the beginning of the loader contains a jump table and the first entry of it has to be called in order to initialize the loader.

After initializing the loader the code between $80148 and $8015a loads the file table from the disk. The size of the file table is $a8 bytes but not all entries contain valid information. I used a typical method of mine to dump the file table: modified the boot loader code to load the file table into some available higher CHIP RAM location, rebooted the machine and the copied the table from memory using RossiMon. No fancy thrills or maneuvers. There are 15 files in the file table instead of possible 21. The file table is used extensively by the loader and the game. After loading the file table, the code between $80154 and $80168 loads the first game code into memory starting at $50000. Note that the register D0 gets set to 0, which for the disk loader means to load the first file from the file table. So far we have learned the following from the loader (assuming when it is located at $c0):

- Calling $c0 initializes the loader:
    - A0=MFM buffer address, A1=buffer address for one track, A2=address of file table
- Calling $c4 loads data from disk:
    - D0=drive number, D1=number of bytes to load
    - A1=destination memory address, A2=byte address on the disk(!)
- Calling $c8 loads data from disk using file table:
    - D0=index to the file table
    - A1=destination memory address



```
00080164 4EEB 000C              jmp     $0C(a3)
-> d8011a
0008011A 41F8 00C0              lea     $C0,a0
0008011E 2648                   movea.l a0,a3
00080120 43FA 0CD4              lea     $080DF6(pc),a1
00080124 303C 038B              move.w  #$038B,d0
00080128 30D9                   move.w  (a1)+,(a0)+
0008012A 51C8 FFFC              dbf     d0,$080128
0008012E 2448                   movea.l a0,a2
00080130 49EA 00A8              lea     $A8(a2),a4
00080134 41F9 0007B460          lea     $07B460,a0
0008013A 43F9 0007E760          lea     $07E760,a1
00080140 48F8 1800 0084         movem.l a3-a4,$84
00080146 4E93                   jsr     (a3)
00080148 7000                   moveq   #$00,d0
0008014A 224A                   movea.l a2,a1
0008014C 223C 000000A8          move.l  #$0000A8,d1
00080152 347C 18A0              movea.w #$18A0,a2
00080156 4EAB 0008              jsr     $08(a3)
0008015A 2EBC 00050000          move.l  #$050000,(sp)
00080160 2257                   movea.l (sp),a1
00080162 7000                   moveq   #$00,d0
00080164 4EEB 000C              jmp     $0C(a3)
00080168 41FA 00FC              lea     $080266(pc),a0
0008016C 203C 000500D4          move.l  #$0500D4,d0
00080172 7A03                   moveq   #$03,d5
00080174 3140 0004              move.w  d0,$04(a0)
00080178 4840                   swap    d0
0008017A 3080                   move.w  d0,(a0)
->
```

Before we dwell into the game code, let’s have a look at the loader. There might be something interesting in it ;) The loader was moved to address $c0 but we can examine it where it is located now i.e. at $80df6.

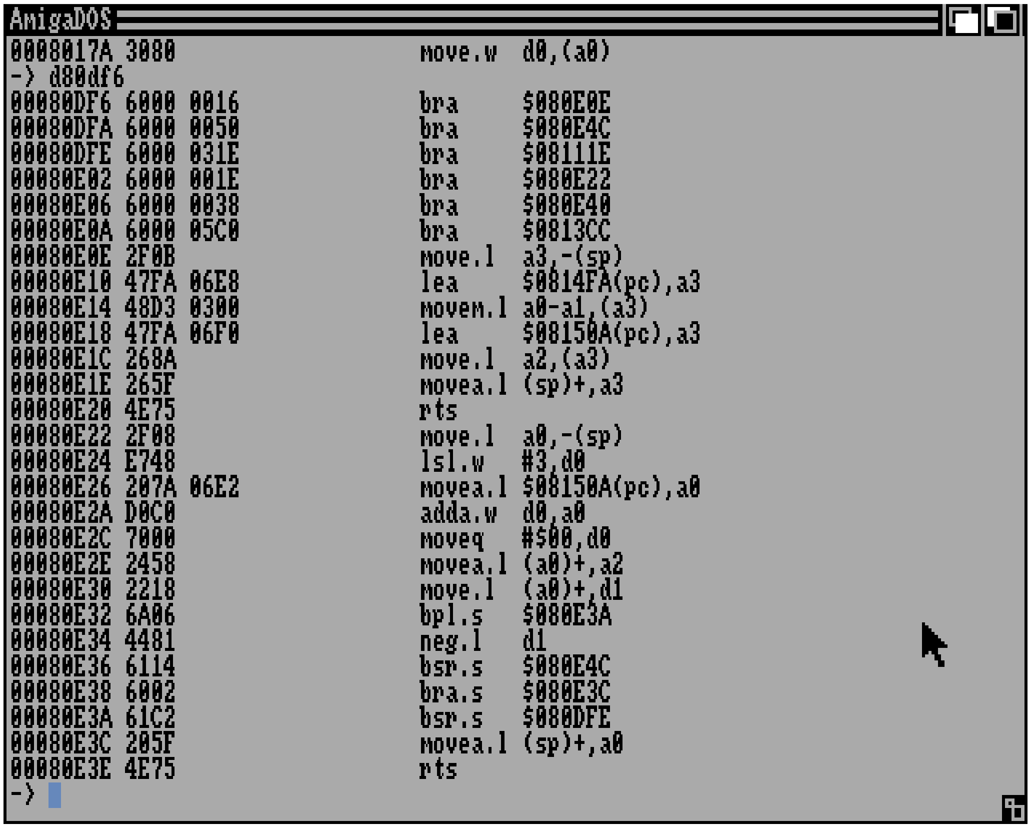
The loader starts with a jump table. The routine at $80df6 i.e. $c0 when the game is running, calls the loader initialization routine at $80e0e. The initialization routine sets up both MFM and track buffer pointers as well as the file table location.

The routine at $80dfa i.e. $c4 when the game is running, jumps to $80e4c. This routine loads a requested amount of data (size has to be modulo of 4) directly from disk (any of four drives). The disk is byte accessed (address has to be modulo of 4 it seems) meaning there is no concept of tracks or sectors from the caller point of view. We will have a closer look at this specific loader routine later on as it seems to be more than just a plain track loader.

The routine at $80dfe i.e. $c8 when the game is running, jumps to $8111e. This loader loads, again, a requested amount of data (size has to be modulo of 4) directly from disk (any of four drives). The disk is byte accessed (address has to be modulo of 4 it seems). Loader routines at $c4 and $c8 are somewhat different, although they share many common subroutines.

The routine at $80e02 i.e. $cc when the game is running, jumps to $80e22, which is the routine to load data/files using a file table. The file table based loader hardcodes the drive to DF0:, which is somewhat odd.

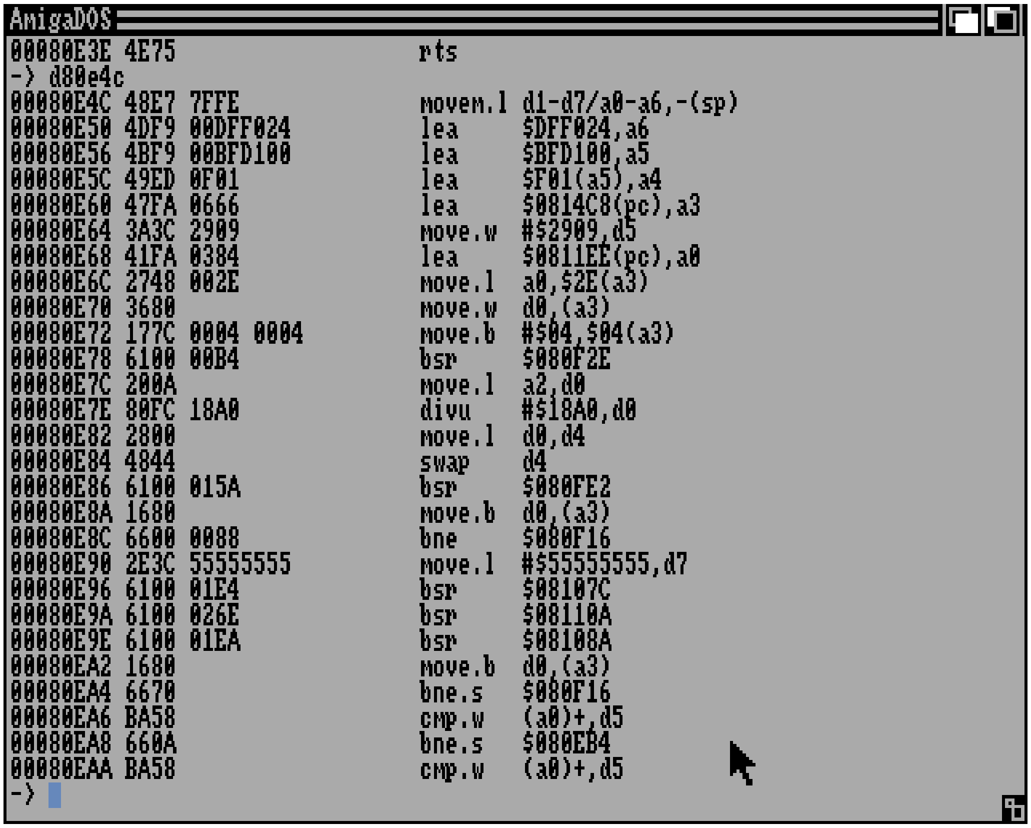
Finally, the routing at $80e0a i.e. $d4 when game is running, jumps to $813cc and this routine is able to save one track at a time to a disk. Obviously it is meant for saving hi-scores or such.



```
0008017A 3080                   move.w  d0,(a0)
-> d80df6
00080DF6 6000 0016              bra     $080E0E
00080DFA 6000 0050              bra     $080E4C
00080DFE 6000 031E              bra     $08111E
00080E02 6000 001E              bra     $080E22
00080E06 6000 0038              bra     $080E40
00080E0A 6000 05C0              bra     $0813CC
00080E0E 2F0B                   move.l  a3,-(sp)
00080E10 47FA 06E8              lea     $0814FA(pc),a3
00080E14 48D3 0300              movem.l a0-a1,(a3)
00080E18 47FA 06F0              lea     $08150A(pc),a3
00080E1C 268A                   move.l  a2,(a3)
00080E1E 265F                   movea.l (sp)+,a3
00080E20 4E75                   rts
00080E22 2F08                   move.l  a0,-(sp)
00080E24 E748                   lsl.w   #3,d0
00080E26 207A 06E2              movea.l $08150A(pc),a0
00080E2A D0C0                   adda.w  d0,a0
00080E2C 7000                   moveq   #$00,d0
00080E2E 2458                   movea.l (a0)+,a2
00080E30 2218                   move.l  (a0)+,d1
00080E32 6A06                   bpl.s   $080E3A
00080E34 4481                   neg.l   d1
00080E36 6114                   bsr.s   $080E4C
00080E38 6002                   bra.s   $080E3C
00080E3A 61C2                   bsr.s   $080DFE
00080E3C 205F                   movea.l (sp)+,a0
00080E3E 4E75                   rts
->
```

Let’s look at the loader routine located at $80e4c. The routine at $80f2e selects the drive, turns on the motor and initializes most of the disk controller registers. There are a couple of immediate observations to make. The disk SYNC word is passed in D5 and we can see that it is set to a custom value $2909. The code between $80e7c and $80e86 looks like calculating the track on a disk, and based on the division value $18a0 we can assume the track data length is $18a0 i.e. 6304 bytes. The call to $80fe2, which seeks to a correct track, and later the call to $8110a to start the disk DMA confirms our assumption. So, we are dealing with a custom disk format with long tracks. Cracking this game just became much more interesting ;)

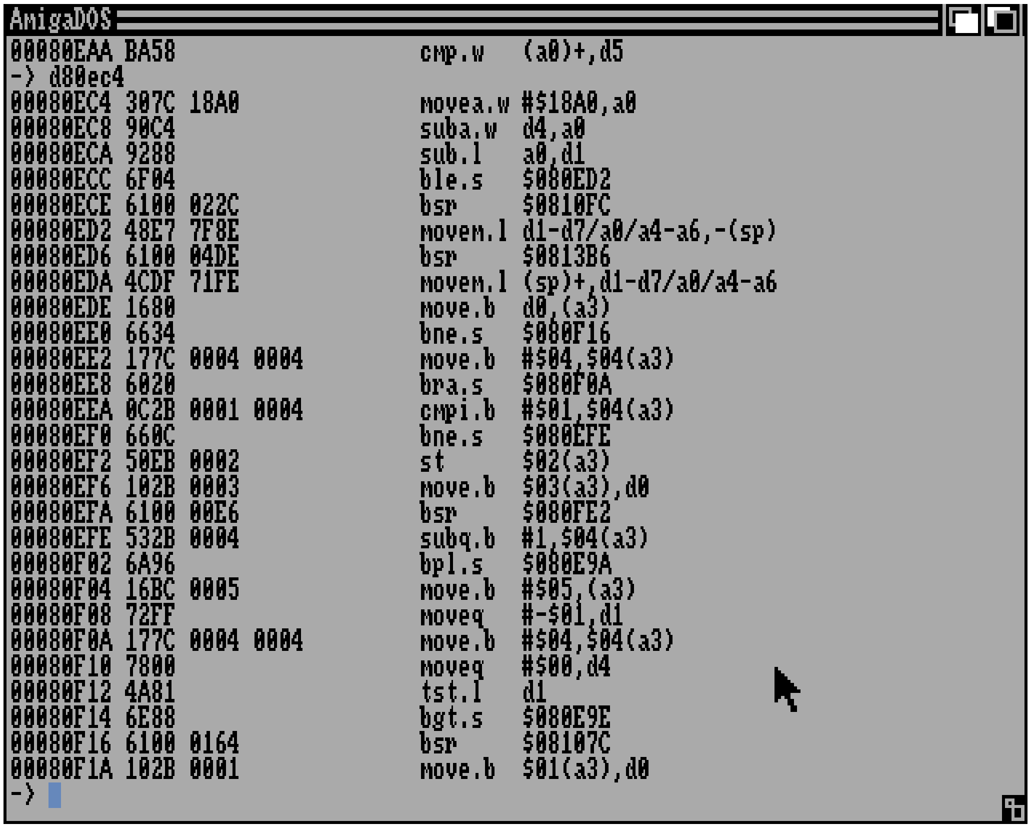
The track decoding code starts around address $80ea6. The actual MFM decoder is located at $810aa. The track format is rather straight forward. A track has no sectors or well, one sector. The track has four custom SYNC words followed by a track data checksum and then the track data itself.



```
00080E3E 4E75                   rts
-> d80e4c
00080E4C 48E7 7FFE              movem.l d1-d7/a0-a6,-(sp)
00080E50 4DF9 00DFF024          lea     $DFF024,a6
00080E56 4BF9 00BFD100          lea     $BFD100,a5
00080E5C 49ED 0F01              lea     $F01(a5),a4
00080E60 47FA 0666              lea     $0814C8(pc),a3
00080E64 3A3C 2909              move.w  #$2909,d5
00080E68 41FA 0384              lea     $0811EE(pc),a0
00080E6C 2748 002E              move.l  a0,$2E(a3)
00080E70 3680                   move.w  d0,(a3)
00080E72 177C 0004 0004         move.b  #$04,$04(a3)
00080E78 6100 00B4              bsr     $080F2E
00080E7C 200A                   move.l  a2,d0
00080E7E 80FC 18A0              divu    #$18A0,d0
00080E82 2800                   move.l  d0,d4
00080E84 4844                   swap    d4
00080E86 6100 015A              bsr     $080FE2
00080E8A 1680                   move.b  d0,(a3)
00080E8C 6600 0088              bne     $080F16
00080E90 2E3C 55555555          move.l  #$55555555,d7
00080E96 6100 01E4              bsr     $08107C
00080E9A 6100 026E              bsr     $08110A
00080E9E 6100 01EA              bsr     $08108A
00080EA2 1680                   move.b  d0,(a3)
00080EA4 6670                   bne.s   $080F16
00080EA6 BA58                   cmp.w   (a0)+,d5
00080EA8 660A                   bne.s   $080EB4
00080EAA BA58                   cmp.w   (a0)+,d5
->
```

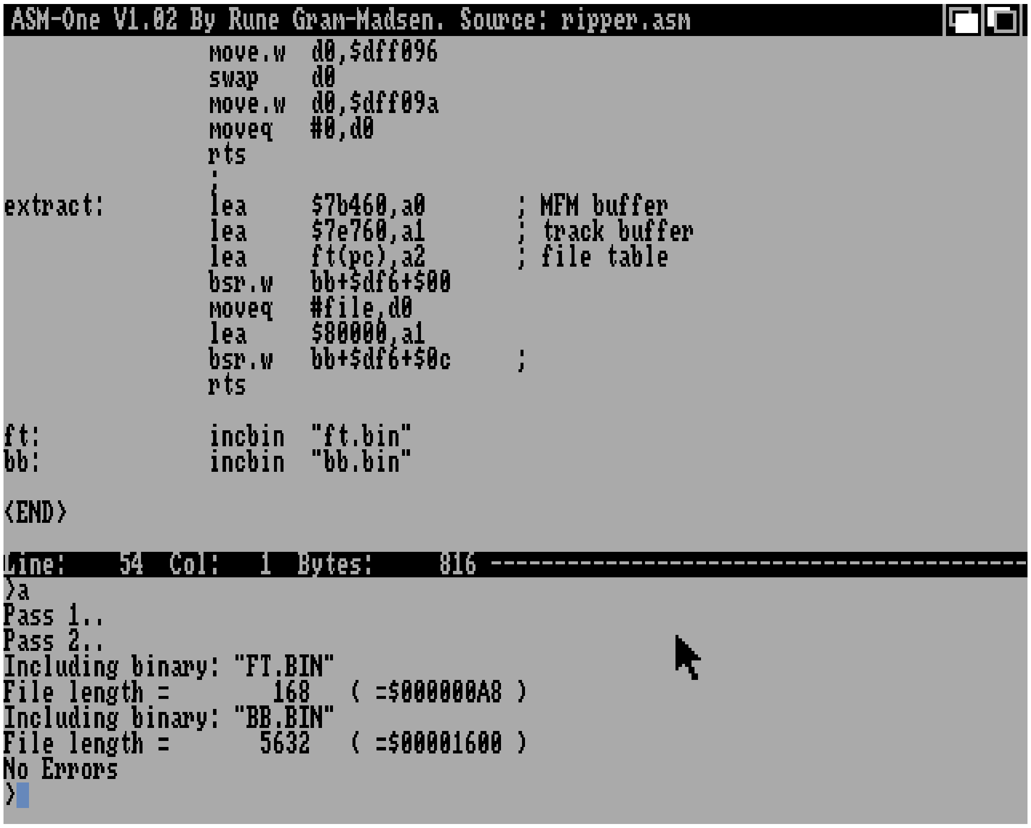
A note about the code between addresses $80e68 and $80e70. The address $811ee happens to be the start address of a special StoneCracker S404 decruncher routine. The loader code seems to "reset" the decruncher to a known state each time a new data/file gets loaded from the disk. Now this is even more interesting. We are not going to look into the decruncher in this tutorial but what it does is to decrunch one track worth of data into memory. The decruncher is able to exit itself anytime when it runs out of data to decrunch saving the exact location where to resume execution next time. Those who are familiar with Amiga's executable decrunchers should now immediately say: "ahahhaaa.. doesn't this look and taste like Titanics cruncher?!" Now having two buffers in the loader starts to make sense. There is one buffer for MFM data and one buffer for track holding data to be decrunched!

The code starting at $80ec4 has to do with loading more data after the first read track. If there is more data (or tracks so to speak) to read from the disk, the call to $810fc advances the drive head to the next track and start the disk DMA. Without waiting for the disk DMA to finish the loader calls a routine at $813b6, which decrunches the *previously* decoded track into destination memory. A neat idea to use the time usually wasted just for polling the disk DMA to finish for something useful like decrunhing. So this loader is able to decrunch S404 crunched files from disk while loading. It also turns out that the file table based loader at $80e02 knows the difference between a normal and a crunched file, and based on that calls either the loader at $80dfa (crunched files) or $80dfe (plain data files). In the file table crunched files have a negative length. Based on what we have now we can estimate that the custom disk format is able to store 1002336 bytes of raw data to the disk and assuming all files were crunched the disk capacity is more than 1500000 bytes (>1.5M). This is of course subject to compression ratio of individual files but not bad at all..



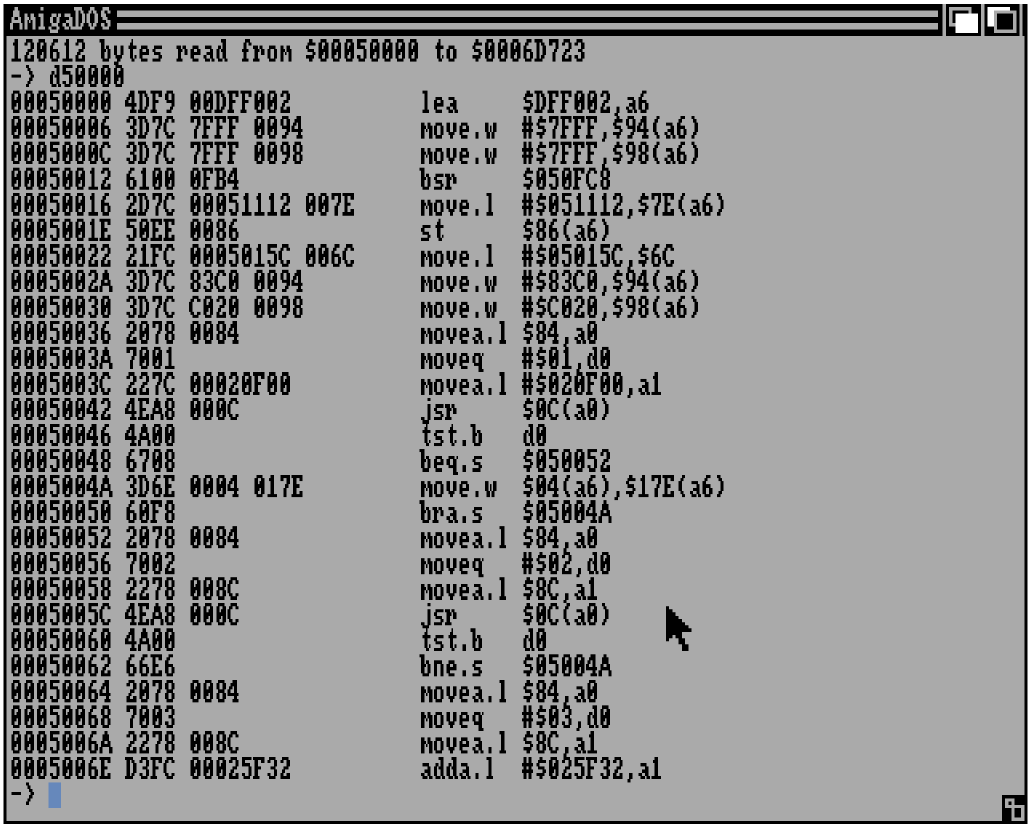
```
00080EAA BA58                   cmp.w   (a0)+,d5
-> d80ec4
00080EC4 307C 18A0              movea.w #$18A0,a0
00080EC8 90C4                   suba.w  d4,a0
00080ECA 9288                   sub.l   a0,d1
00080ECC 6F04                   ble.s   $080ED2
00080ECE 6100 022C              bsr     $0810FC
00080ED2 48E7 7F8E              movem.l d1-d7/a0/a4-a6,-(sp)
00080ED6 6100 04DE              bsr     $0813B6
00080EDA 4CDF 71FE              movem.l (sp)+,d1-d7/a0/a4-a6
00080EDE 1680                   move.b  d0,(a3)
00080EE0 6634                   bne.s   $080F16
00080EE2 177C 0004 0004         move.b  #$04,$04(a3)
00080EE8 6020                   bra.s   $080F0A
00080EEA 0C2B 0001 0004         cmpi.b  #$01,$04(a3)
00080EF0 660C                   bne.s   $080EFE
00080EF2 50EB 0002              st      $02(a3)
00080EF6 102B 0003              move.b  $03(a3),d0
00080EFA 6100 00E6              bsr     $080FE2
00080EFE 532B 0004              subq.b  #1,$04(a3)
00080F02 6A96                   bpl.s   $080E9A
00080F04 16BC 0005              move.b  #$05,(a3)
00080F08 72FF                   moveq   #-$01,d1
00080F0A 177C 0004 0004         move.b  #$04,$04(a3)
00080F10 7800                   moveq   #$00,d4
00080F12 4A81                   tst.l   d1
00080F14 6E88                   bgt.s   $080E9E
00080F16 6100 0164              bsr     $08107C
00080F1A 102B 0001              move.b  $01(a3),d0
->
```

Anyway, let’s leave the loader for a while and extract files from the disk. We have two ways to do that. First, just dump the entire disk using the “normal” loader and then extract individual files from the binary file dump. Second, use the file table based loader to load each file individually and save them. We will use the latter because it turns out that the crunched files on the disk are preprocessed and not decrunchable with a normal StoneCracker decruncher routine. Because I am lazy we’ll let the disk loader to decrunch those files for us. Actually I did both methods because it was easier in that way to find the decrunched lengths of the crunched files from the disk dump. The file table does not have this original file length information – just the size of the files on the disk. We also need to extract those files that are loaded from the boot block and therefore not listed in the file table i.e. the file table itself. The following simple assembly program (ripper.asm – you can find it in the cracked game disk under “stuff” directory) does the thing. We just need to load each file one by one and save them to the disk. I am not showing the parts of the code that save and restore the system. If we take a peek into “normal” files they appear also be crunched with a normal StoneCracker and have a decruncher code attached to them. For now I assume these are program files that are cached in RAM and decrunched & run on need to basis. We’ll see later if the assumption was correct.



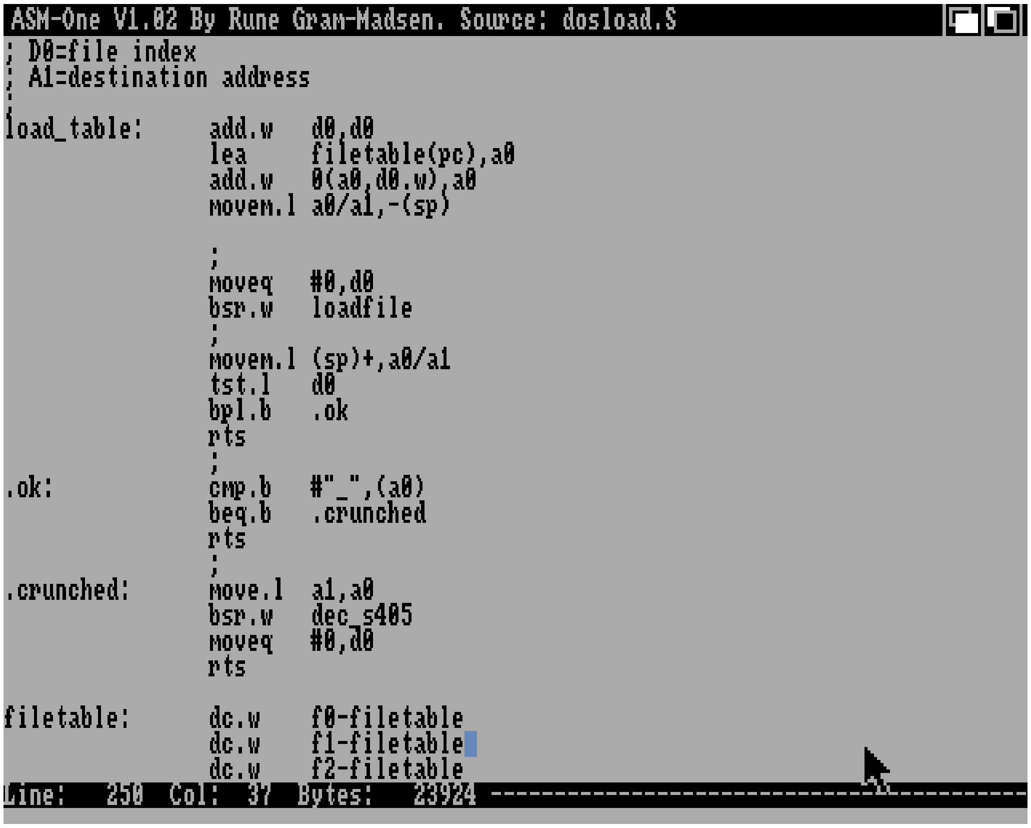
```
                move.w  d0,$dff096
                swap    d0
                move.w  d0,$dff09a
                moveq   #0,d0
                rts
                ;
                ;
extract:        lea     $7b460,a0       ; MFM buffer
                lea     $7e760,a1       ; track buffer
                lea     ft(pc),a2       ; file table
                bsr.w   bb+$df6+$00
                moveq   #file,d0
                lea     $80000,a1
                bsr.w   bb+$df6+$0c     ;
                rts

ft:             incbin  "ft.bin"
bb:             incbin  "bb.bin"

<END>

Line:    54  Col:   1  Bytes:     816 ----------------------------------------
>a
Pass 1..
Pass 2..
Including binary: "FT.BIN"
File length =        168   ( =$000000A8 )
Including binary: "BB.BIN"
File length =       5632   ( =$00001600 )
No Errors
>
```

After extracting all 15 files they sum up to 1104756 bytes, which will definitely *not* fit into our OFS formatted disk. The files have to be recrunched but that is for a later phase. Next we will look into the first loaded game code and what it keeps in. This is the “file 0” in the file table. As we noted earlier the first game code gets loaded into address $50000. The code at $50000 looks (boringly) normal. It does load files 1,2,3,4,5,6,7 and 8 into the memory. The file 1 gets loaded into $20f00, which seems to be next game code to execute. The file 5 gets loaded into $900 and the rest of the files into the additional RAM that we located during the boot. Later in the game more files are loaded when the player has completed the first set of game levels. Anyway, the code looks straight forward in a sense that it always uses the disk loader routines whose entry address is located in address $84 (i.e. where boot block stored the address of the loader). I’ll take the risk and do not even try to modify the game code but just replace the track loader with a DOS file loader and hope the game always uses the same interface to access the disk. There is no sign of saving hi-scores or using any other loader functions than the file table method. Also, there seem to be no protection what so ever (well.. I did not look too hard for them after all).



```
120612 bytes read from $00050000 to $0006D723
-> d50000
00050000 4DF9 00DFF002          lea      $DFF002,a6
00050006 3D7C 7FFF 0094         move.w   #$7FFF,$94(a6)
0005000C 3D7C 7FFF 0098         move.w   #$7FFF,$98(a6)
00050012 6100 0FB4              bsr      $050FC8
00050016 2D7C 00051112 007E     move.l   #$051112,$7E(a6)
0005001E 50EE 0086              st       $86(a6)
00050022 21FC 0005015C 006C     move.l   #$05015C,$6C
0005002A 3D7C 83C0 0094         move.w   #$83C0,$94(a6)
00050030 3D7C C020 0098         move.w   #$C020,$98(a6)
00050036 2078 0084              movea.l  $84,a0
0005003A 7001                   moveq    #$01,d0
0005003C 227C 00020F00          movea.l  #$020F00,a1
00050042 4EA8 000C              jsr      $0C(a0)
00050046 4A00                   tst.b    d0
00050048 6708                   beq.s    $050052
0005004A 3D6E 0004 017E         move.w   $04(a6),$17E(a6)
00050050 60F8                   bra.s    $05004A
00050052 2078 0084              movea.l  $84,a0
00050056 7002                   moveq    #$02,d0
00050058 2278 008C              movea.l  $8C,a1
0005005C 4EA8 000C              jsr      $0C(a0)
00050060 4A00                   tst.b    d0
00050062 66E6                   bne.s    $05004A
00050064 2078 0084              movea.l  $84,a0
00050068 7003                   moveq    #$03,d0
0005006A 2278 008C              movea.l  $8C,a1
0005006E D3FC 00025F32          adda.l   #$025F32,a1
->
```

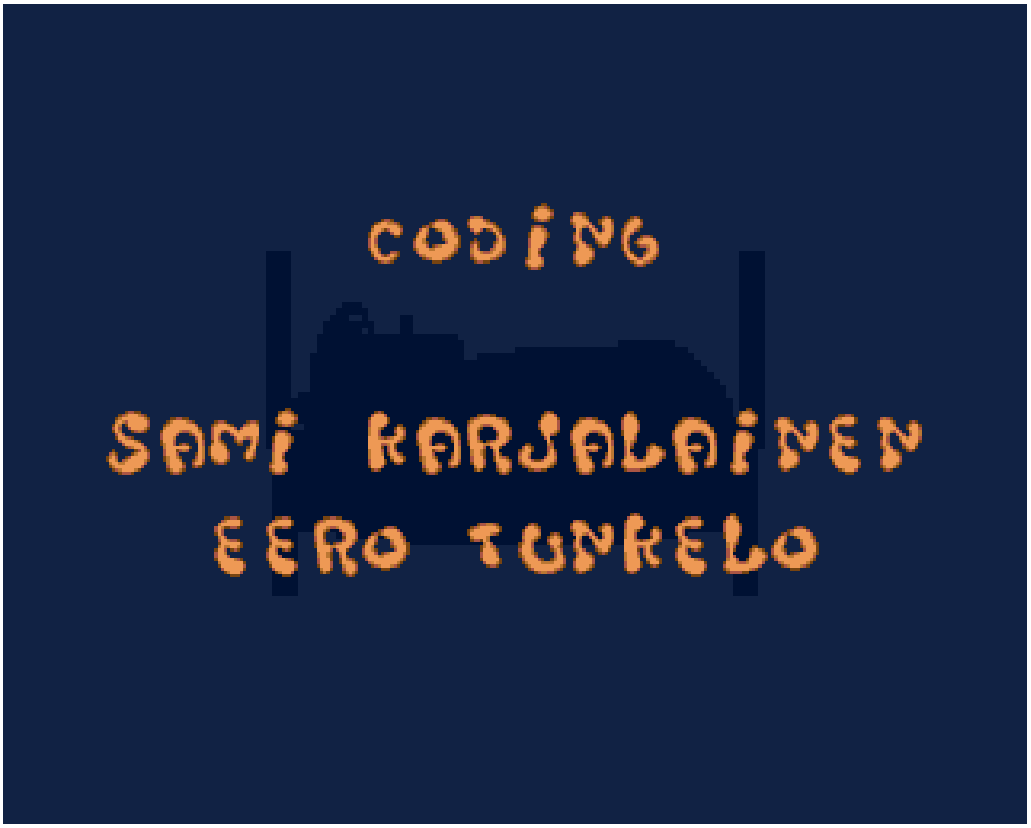
Now that I got all files extracted and a good idea how the game code works I can move to the next part i.e., putting everything together. Those means taking one of my old hardware banging DOS file loaders and make it behave just like the game code expects the track loader to work. I also add a proper startup code that zeroes VBR, takes care of caches and makes the video mode OCS. The “new” loader also includes a fixed memory check so that the game will complain and refuse to run if there is less than 1MB or RAM available. I took the graphics for the “1MB Needed” from the original boot block and decrunched it (the graphics was S404 crunched). Since the original game track loader had plenty of work space (one MFM buffer and one decoded track buffer) at the end of 512K CHIP RAM the new loader can use those as well for its MFM buffer, sector buffer and decruncher work space (yes, StoneCracker v5 needs $A20 bytes of work space). You can find the “new loader” dosload.S in the cracked game disk under the “stuff” directory.



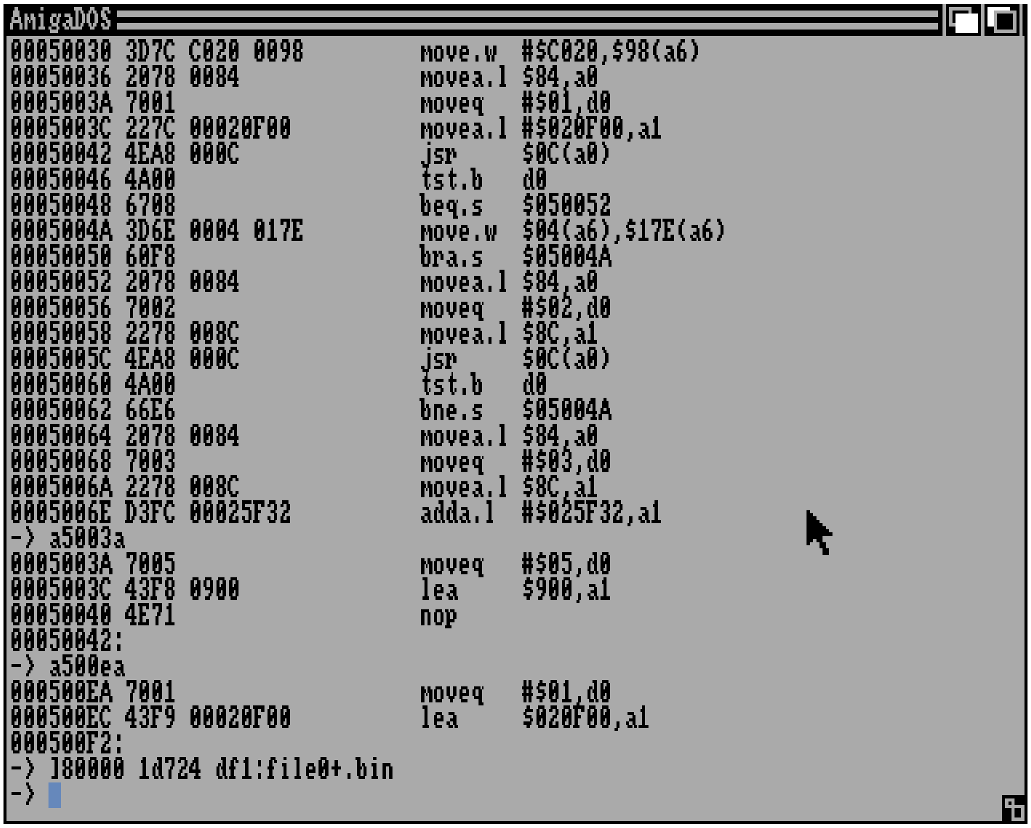
```
ASM-One V1.02 By Rune Gram-Madsen. Source: dosload.S
; D0=file index
; A1=destination address
;
load_table:     add.w   d0,d0
                lea     filetable(pc),a0
                add.w   0(a0,d0.w),a0
                movem.l a0/a1,-(sp)

                ;
                moveq   #0,d0
                bsr.w   loadfile
                ;
                movem.l (sp)+,a0/a1
                tst.l   d0
                bpl.b   .ok
                rts
                ;
.ok:            cmp.b   #"_",(a0)
                beq.b   .crunched
                rts
                ;
.crunched:      move.l  a1,a0
                bsr.w   dec_s405
                moveq   #0,d0
                rts

filetable:      dc.w    f0-filetable
                dc.w    f1-filetable
                dc.w    f2-filetable
Line:   250  Col:  37  Bytes:   23924 ----------------------------------------
```

I’ll use StoneCracker v5 to crunch all files that were crunched “Titanics style” on the original game disk. I do not recrunch normal S404 crunched files – since I am just too lazy for that. Unfortunately I’ll lose the neat decrunch-while-loading feature and crunched files are now loaded entirely first and then decrunched. It would be too much trouble to recreate the “Titanics style” loader for this crack. Once all DOS files have been written to an OFS formatted floppy I still got like 20% free, so plenty of space for intro and the new loader program. The last thing to add is the dumbest possible Startup-Sequence to load the “new loader” program and we are ready to try the cracked game.

The intro starts and music plays. Great! However, when the actual game is about to start we are greeted with the infamous Guru-Meditation. Now, what is the problem?

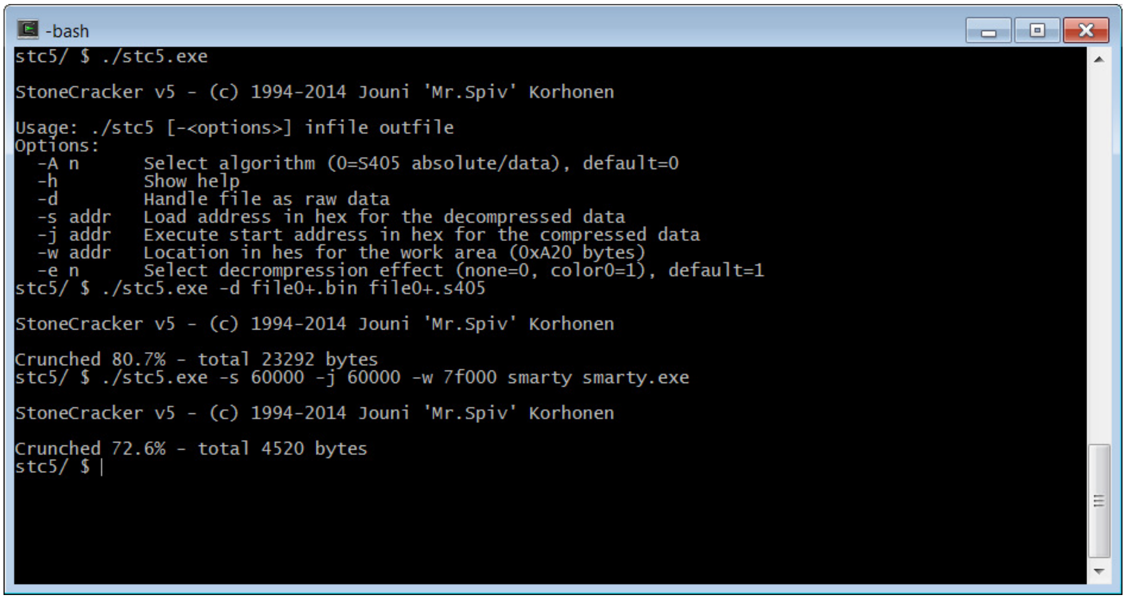


All files worked just fine when tried individually (yes, I tested them all). To cut the furious +30 minutes debug session short with multiple reboots on my A600 I managed to narrow the issue to a rather unpleasant discovery. StoneCracker v5 decruncher overwrites the decrunched memory area by two bytes when the source and the destination memory areas overlap entirely (you cruncher lovers know what situation I am talking about). Sigh! Those overwritten bytes just happen to be the start of code located at $20f00. You remember that the game loads one file to $20f00 and another to $900, and in that order. The latter decrunched file is supposed to end at $20eff but.. I am not in a mood of fixing the decruncher so there has to be another way to go around the issue. I could write the overwritten bytes back to memory after decrunching or load crunched files to some other memory area to avoid overlap situation or reverse the file loading order. I go for the last option and do small editing/assembling using RossiMon. I was really short of RAM to avoid the overlap situation and therefore chose reversing the file loading order.



```
00050030 3D7C C020 0098         move.w  #$C020,$98(a6)
00050036 2078 0084              movea.l $84,a0
0005003A 7001                   moveq   #$01,d0
0005003C 227C 00020F00          movea.l #$020F00,a1
00050042 4EA8 000C              jsr     $0C(a0)
00050046 4A00                   tst.b   d0
00050048 6708                   beq.s   $050052
0005004A 3D6E 0004 017E         move.w  $04(a6),$17E(a6)
00050050 60F8                   bra.s   $05004A
00050052 2078 0084              movea.l $84,a0
00050056 7002                   moveq   #$02,d0
00050058 2278 008C              movea.l $8C,a1
0005005C 4EA8 000C              jsr     $0C(a0)
00050060 4A00                   tst.b   d0
00050062 66E6                   bne.s   $05004A
00050064 2078 0084              movea.l $84,a0
00050068 7003                   moveq   #$03,d0
0005006A 2278 008C              movea.l $8C,a1
0005006E D3FC 00025F32          adda.l  #$025F32,a1
-) a5003a
0005003A 7005                   moveq   #$05,d0
0005003C 43F8 0900              lea     $900,a1
00050040 4E71                   nop
00050042:
-) a500ea
000500EA 7001                   moveq   #$01,d0
000500EC 43F9 00020F00          lea     $020F00,a1
000500F2:
-) ]80000 1d724 df1:file0+.bin
-)
```

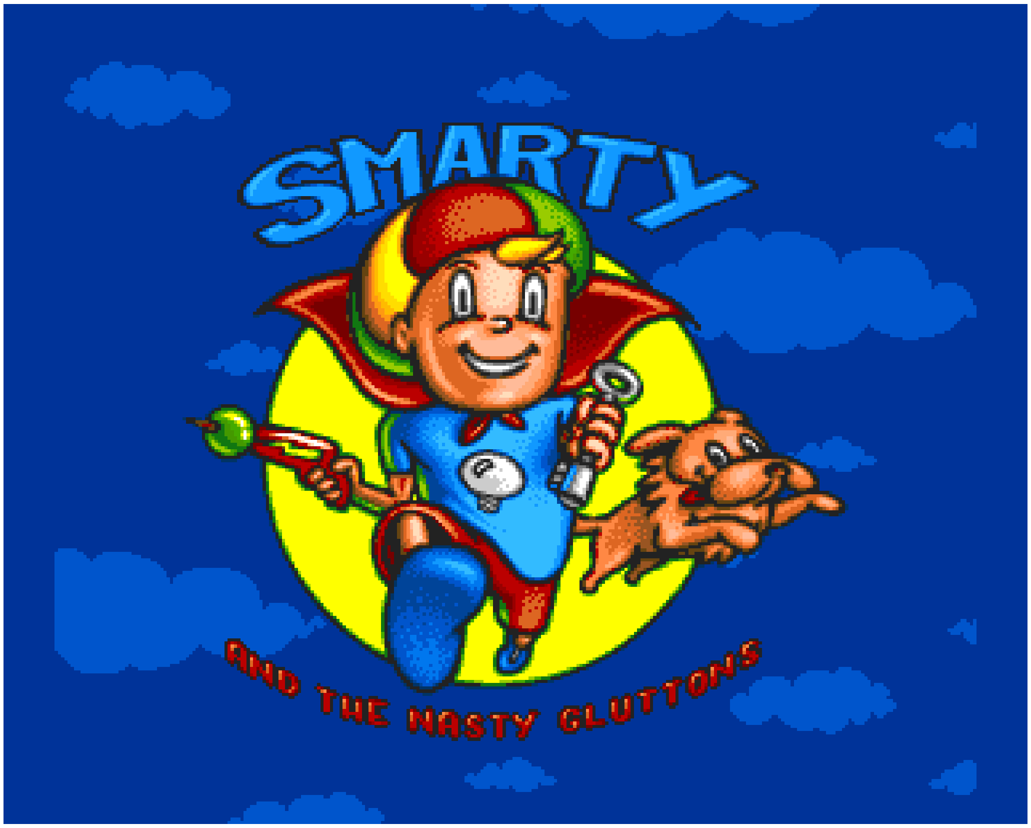
After modifying the game code at $5003a and $500ea I write the file back to a disk. I need to change the new loader as well to match the new file name. After assembling the new loader some crunching takes again place. Unfortunately I need to use Windows for this due my laziness. I still got no Amiga binary for StoneCracker v5. On the other hand crunching these files is blazing fast on my Dell laptop compared to anything on my 68000 A600 ;-)



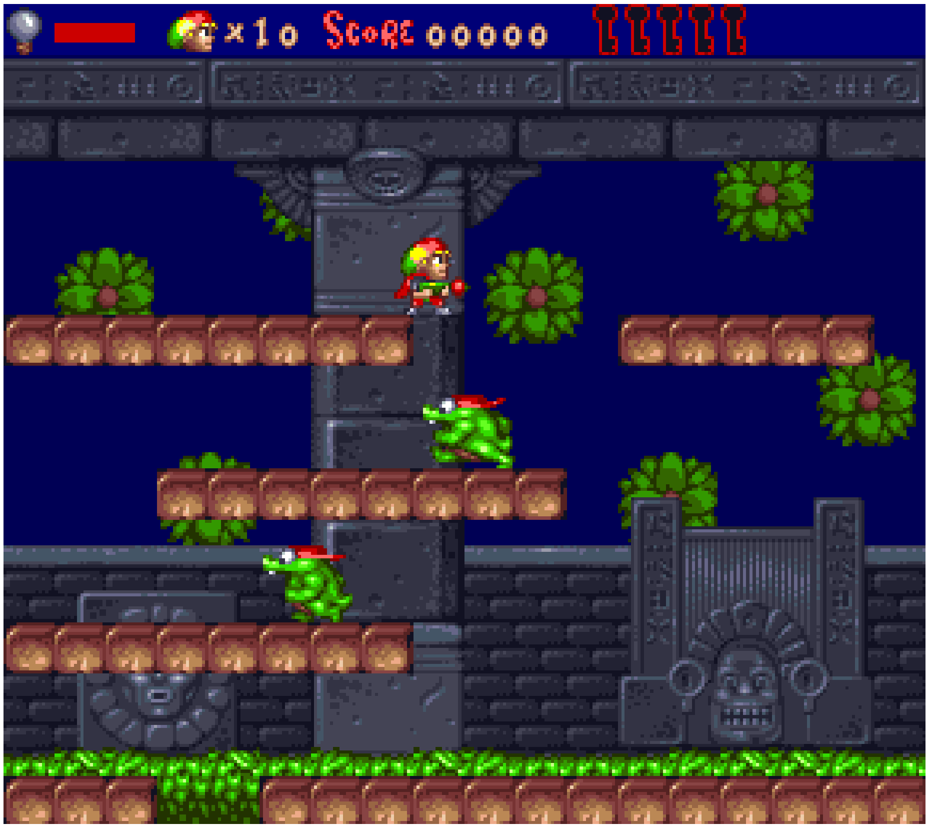
```
stc5/ $ ./stc5.exe

StoneCracker v5 - (c) 1994-2014 Jouni 'Mr.Spiv' Korhonen

Usage: ./stc5 [-<options>] infile outfile
Options:
  -A n      Select algorithm (0=S405 absolute/data), default=0
  -h        Show help
  -d        Handle file as raw data
  -s addr   Load address in hex for the decompressed data
  -j addr   Execute start address in hex for the compressed data
  -w addr   Location in hes for the work area (0xA20 bytes)
  -e n      Select decompression effect (none=0, color0=1), default=1
stc5/ $ ./stc5.exe -d file0+.bin file0+.s405

StoneCracker v5 - (c) 1994-2014 Jouni 'Mr.Spiv' Korhonen

Crunched 80.7% - total 23292 bytes
stc5/ $ ./stc5.exe -s 60000 -j 60000 -w 7f000 smarty smarty.exe

StoneCracker v5 - (c) 1994-2014 Jouni 'Mr.Spiv' Korhonen

Crunched 72.6% - total 4520 bytes
stc5/ $ |
```

Let’s try again.. everything seems to work just fine and the game main screen appears after a while. We nailed it. However, as a side effect changing the order of loaded files the intro sequence during the loading is not a bit out of sync. Sorry.

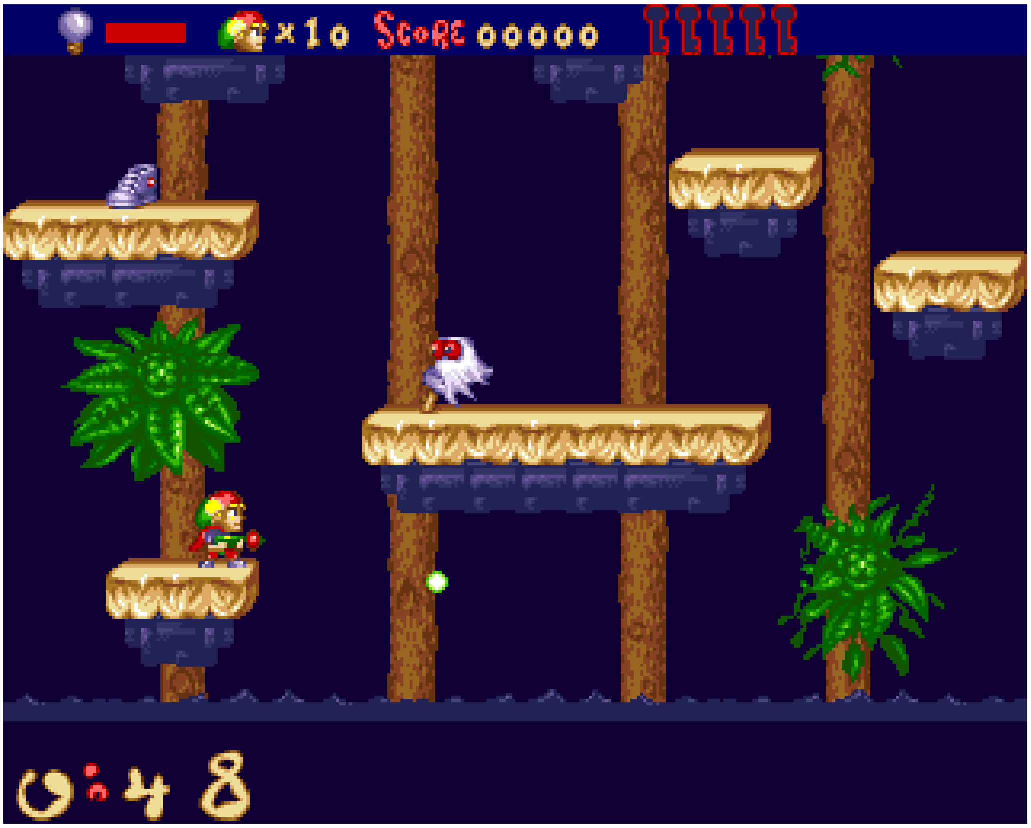
The start screen:



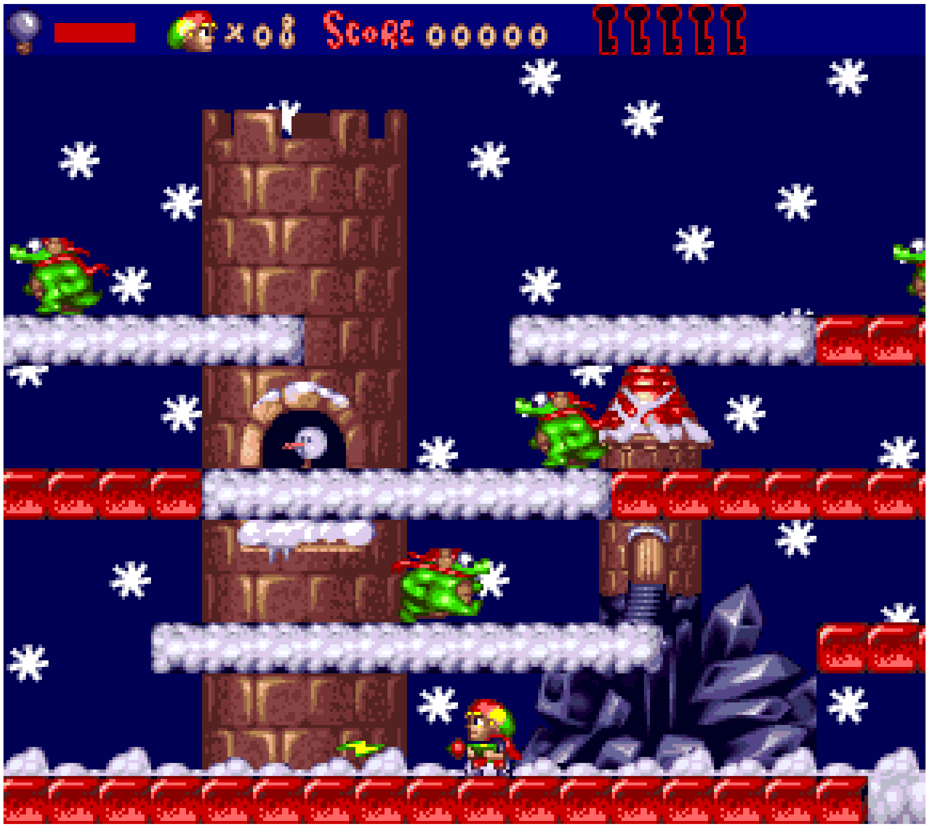
First levels:



First bonus level:



Second levels:



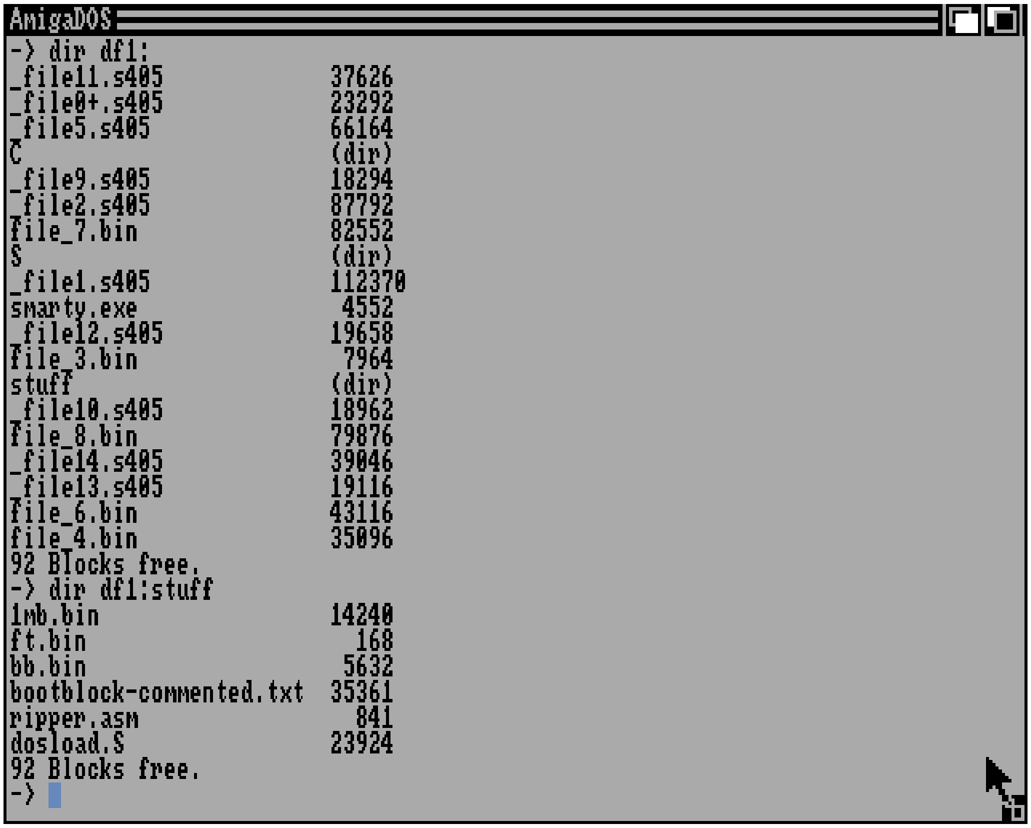
As a summary the only modifications we did to the game were:

- Replaced the boot block based loader with a normal boot block.
- Extracted the game files from the custom formatted disk and saved them as normal DOS files.
- Replaced the custom format track loader with a DOS file loader.
- And the single modification to actual game code swapping the order of loading files 1 and 5.

Obviously the implementation of the "new loader" took a while, especially as it was entirely done on my A600. Swapping files between A600 and Windows UAE for making screen captures is somewhat tedious.

Regarding the game it has two sets of playable levels (and bonus levels). You can fool around and shoot those Nasty Gluttons as much as you want. However, you cannot complete the game or even a single level. To move from level to another, press the left mouse button. Once you have gone through all levels you will be greeted that the dream is over and a reboot will follow.

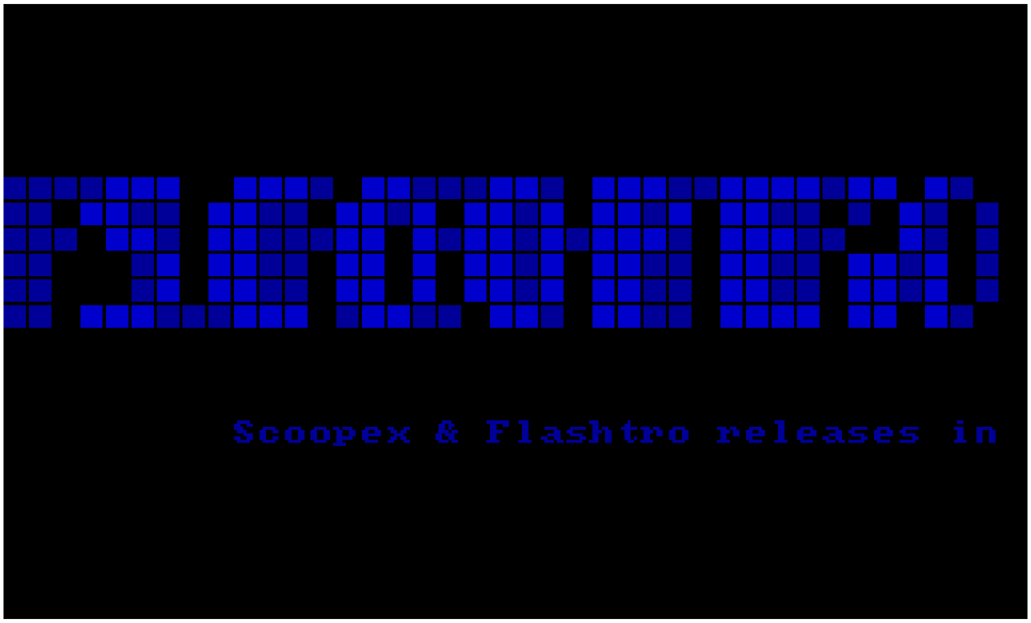
Now that the game works, I can concentrate on adding the crack intro (ahem.. or the Startup-Sequence text) and stuff to the release version of the game preview you have. All additional material helping to check what was done to crack this game preview is in the cracked game disk under the "stuff" directory. The fully commented game disk bootblock disassembly is also there.



```
-> dir df1:
_file11.s405             37626
_file0+.s405             23292
_file5.s405              66164
C                        (dir)
_file9.s405              18294
_file2.s405              87792
file_7.bin               82552
S                        (dir)
_file1.s405              112370
smarty.exe                4552
_file12.s405             19658
file_3.bin                7964
stuff                    (dir)
_file10.s405             18962
file_8.bin               79876
_file14.s405             39046
_file13.s405             19116
file_6.bin               43116
file_4.bin               35096
92 Blocks free.
-> dir df1:stuff
1mb.bin                  14240
ft.bin                     168
bb.bin                    5632
bootblock-commented.txt  35361
ripper.asm                 841
dosload.S                23924
92 Blocks free.
->
```

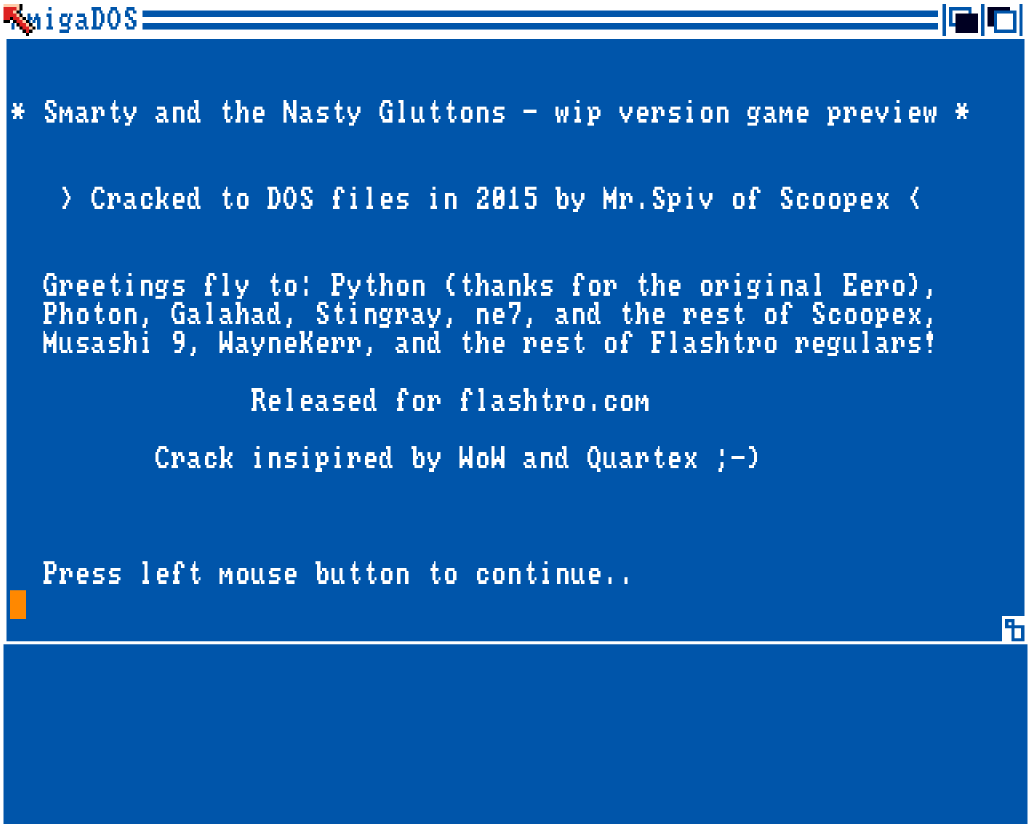
The DOS file loader crack approach was inspired by World of Wonder’s Dugger crack by Eurosoft. Not to mention that crack has the best Amiga crack intro ever! Kudos to Quartex for ultimate Startup-Sequence style cracks that was blatantly copied here. It is all for fun & nostalgia you know ;-)

The last minute bootblock intro had to be added – a true coder “beauty” after few tiring hours of “how did I do that again.. grumbles..”. The sources are also on the crack disk under the “stuff” directory as bootblock.asm.



Scoopex & Flashtro releases in

The infamous Quartex style crack..



```
* Smarty and the Nasty Gluttons - wip version game preview *

   > Cracked to DOS files in 2015 by Mr.Spiv of Scoopex <

  Greetings fly to: Python (thanks for the original Eero),
  Photon, Galahad, Stingray, ne7, and the rest of Scoopex,
  Musashi 9, WayneKerr, and the rest of Flashtro regulars!

               Released for flashtro.com

        Crack insipired by WoW and Quartex ;-)


  Press left mouse button to continue..
```

```
MOVEQ #0,D0
RTS
```