

Devices_Manual

COLLABORATORS

	<i>TITLE :</i> Devices_Manual		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 18, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Devices_Manual	1
1.1	Amiga® RKM Devices: Appendix C : Floppy Boot Process and Physical Layout	1
1.2	Appendix C / Commodore-Amiga Disk Format	1
1.3	Appendix C / MFM Track Encoding	3

Chapter 1

Devices_Manual

1.1 Amiga® RKM Devices: Appendix C : Floppy Boot Process and Physical Layout

The first two sectors on each floppy disk contain special boot information. These sectors are read into the system at an arbitrary position; therefore, the code must be position independent. The first three longwords come from the include file `devices/bootblock.h`. The type must be `BBID_DOS`; the checksum must be correct (an additive carry wraparound sum of `0xffffffff`). Execution starts at location 12 of the first sector read in.

The code is called with an open `trackdisk.device` I/O request pointer in `A1` (see the `''Trackdisk''` chapter for more information). The boot code is free to use the IO request as it wishes (the code may trash `A1`, but must not trash the I/O request itself).

The boot code must return values in two registers: `D0` and `A0`. `D0` is a failure code - if it is non-zero then a system alert will be called, and the system will reboot.

If `D0` is zero then `A0` must contain the start address to jump to. The strap module will free the boot sector memory, free the boot picture memory, close the `trackdisk.device` I/O request, do any other cleanup that is required, then jump to the location pointed to by `A0`.

Boot code may allocate memory, use `trackdisk.device` to load relocatable information into the memory, then return with `D0=0` and `A0` pointing to code. The system will clean up, then call the code.

Commodore-Amiga Disk Format
MFM Track Encoding

1.2 Appendix C / Commodore-Amiga Disk Format

The following are details about how the bits on the Commodore-Amiga disk are actually written.

Gross Data Organization:

3 1/2 inch (90mm) disk
double-sided
80 cylinders/160 tracks

Per-track Organization:

Nulls written as a gap, then 11 or 22 sectors of data.
No gaps written between sectors.

Per-sector Organization:

All data is MFM encoded. This is the pre-encoded contents
of each sector:

two bytes of 00 data (MFM = \$AAAA each)
two bytes of A1* ("standard sync byte" -- MFM
encoded A1 without a clock pulse)
(MFM = \$4489 each)
one byte of format byte (Amiga 1.0 format = \$FF)
one byte of track number
one byte of sector number
one byte of sectors until end of write (NOTE 1)
[above 4 bytes treated as one longword
for purposes of MFM encoding]
16 bytes of OS recovery info (NOTE 2)
[treated as a block of 16 bytes for encoding]
four bytes of header checksum
[treated as a longword for encoding]
four bytes of data-area checksum
[treated as a longword for encoding]
512 bytes of data
[treated as a block of 512 bytes for encoding]

NOTE:

The track number and sector number are constant for each particular sector. However, the sector offset byte changes each time we rewrite the track.

The Amiga does a full track read starting at a random position on the track and going for slightly more than a full track read to assure that all data gets into the buffer. The data buffer is examined to determine where the first sector of data begins as compared to the start of the buffer. The track data is block moved to the beginning of the buffer so as to align some sector with the first location in the buffer.

Because we start reading at a random spot, the read data may be divided into three chunks: a series of sectors, the track gap, and another series of sectors. The sector offset value tells the disk software how many more sectors remain before the gap. From this the software can figure out the buffer memory location of the last byte of legal data in the buffer. It can then search past the gap for the next sync byte and, having found it, can block move the rest of the disk data so that all 11 sectors of data are contiguous.

Example:

The first-ever write of the track from a buffer looks like this:

```
<GAP> |sector0|sector1|sector2|.....|sector10|
```

sector offset values:

```
11      10      9      .....      1
```

(If I find this one at the start of my read buffer, then I know there are this many more sectors with no intervening gaps before I hit a gap). Here is a sample read of this track:

```
<junk>|sector9|sector10|<gap>|sector0|...|sector8|<junk>
```

value of 'sectors till end of write':

```
2      1      ....      11      ...      3
```

result of track re-aligning:

```
<GAP>|sector9|sector10|sector0|...|sector8|
```

new sectors till end of write:

```
11      10      9      ...      1
```

so that when the track is rewritten, the sector offsets are adjusted to match the way the data was written.

Sector Label Area

This is operating system dependent data and relates to how AmigaDOS assigns sectors to files. Reserved for future use.

1.3 Appendix C / MFM Track Encoding

When data is MFM encoded, the encoding is performed on the basis of a data block-size. In the sector encoding described above, there are bytes individually encoded; three segments of 4 bytes of data each, treated as longwords; one segment of 16 bytes treated as a block; two segments of longwords for the header and data checksums; and the data area of 512 bytes treated as a block.

When the data is encoded, the odd bits are encoded first, then the even bits of the block.

The procedure is: Make a block of bytes formed from all odd bits of the block, encode as MFM. Make a block of bytes formed from all even bits of the block, encode as MFM. Even bits are shifted left one bit position before being encoded.

The raw MFM data that must be presented to the disk controller will be twice as large as the unencoded data. The relationship is:

```
1 -> 01
0 -> 10    ;if following a 0
0 -> 00    ;if following a 1
```

With clever manipulation, the blitter can be used to encode and decode the MFM.