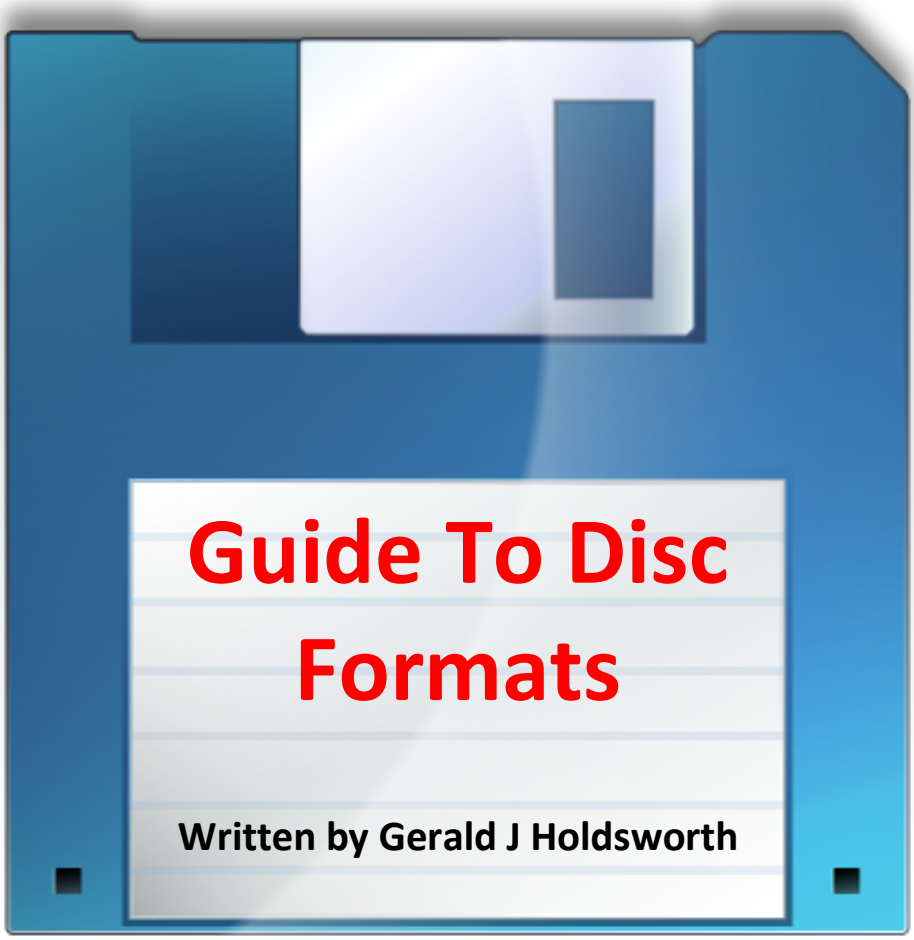

The technical background, in an easy to understand guide, for the Acorn and Watford Disc Filing Systems, Acorn Advanced Disc Filing System, Commodore 1541/1571/1581 and Commodore Amiga.

Introduction 5
Acknowledgements 6
Note on Sinclair/Amstrad Format 7

Acorn Disc Filing System 8
Introduction 8
Specifications 8
Double Sided Interleaving 8
Maps 8
Catalogue information 8
Catalogue Information - Watford DFS 9
File entries (sector 0/2) 9
File details (sector 1/3) 9
File Order 10
Forbidden Characters 10
Identifying a DFS Image 10
Single Sided Images *10*
Double Sided Images *10*
Watford DFS *11*

Acorn Advanced Disc Filing System 12
Introduction 12
Double Sided Interleaving 12
Maps 13
Old Map *13*
Calculating the Checksums 13
New Map *13*
Zone Header 13
Disc Record (zone 0 only) 13
Calculating Zone_Check 14
Allocation Bytes and Indirect Addresses 15
Example, using the same disc 16
Multi-zone discs *17*
Example 18
Start and End of Search *19*
So why these figures? 19
Where to Start 20
Directories 21
Header (Old & New) *21*
Header (Big) *21*
Tail (Old) *21*
Tail (New) *21*
Tail (Big) *22*
What It All Means *22*
StartMasSeq and EndMasSeq 22
StartName and EndName 22
DirLastMark 22
DirCheckByte 22
DirNameLen 23
DirSize, DirEntries and DirNameSize 23
Boot Block 23
Defect List *23*
Hardware-dependent Information *23*
Partial Disc Record *23*
Boot Block Checksum *24*
The Directory Entries 24
Entries (Old & New) *24*

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk 9 January 2021

Entries (Big) 24
Name Heap (Big) 24
Filenames in Big Directories 25
Timestamps and Filetypes 25
Free Space Map 26
Old Map 26
New Map 26
Identifying an ADFS Image 26
Old Map 26
New Map 27
A Note On Hard Drive Images 28
A Word of Warning 28
Commodore 1541, 1571 and 1581 29
Introduction 29
Specifications 29
Fragmentation 29
Converting From Track/Sector to Offset Address 30
Header and Block Availability Map (BAM) – 1541 and 1571 30
Header and Block Availability Map (BAM) – 1581 31
Header 31
Block Availability Map 31
How the Block Availability Maps Work 32
Directory Entries 33
Identifying a 1541/1571/1581 Image 33
1541/1571 33
1581 34
Commodore AmigaDOS 35
Introduction 35
Big Endian vs Little Endian 35
Disc Structures 35
Basic Layout of the Disc 35
Bootblock 35
Disc Identifier 35
Flags 36
Checksums 36
Block Primary and Secondary Types 36
Date and Time stamps 36
Hash Tables 36
Bitmap Block 36
The Blocks 37
Rootblock 37
Directory block 37
File Block 38
File Attributes 39
Hash Chain 39
File Extension Blocks 39
OFS Data Block 39
File Extension Block 40
Identifying an AmigaDOS Image 40
AmigaDOS Block Layout Comparison 41
Inf Files 42

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

*.inf format 42

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

Introduction

This is a, hopefully, simplified explanation of how the various disc image formats are laid out, primarily for use on disc image files. This was largely written to make it easier to work out, as you will need a generous amount of paracetamol to understand the PRMs with regards to the ADFS/FileCore format.

I've tried to avoid using any actual code, and use pseudo code where possible, but hex numbers and operators tend to use the C++ standard (even though I program mostly in Delphi). This is all mainly aimed at floppy disc images, but the theory should also be true for hard disc images.

This guide, and the chapters, is laid out in, what I see, as a logical order. We start with the first filing system, after Tape, on the BBC Micro, the Disc Filing System. This was predated with Acorn's version of DOS on the Atom, which has a very similar format. We then move onto the successor to DFS, the Advanced Disc Filing System. After that, we move from Acorn to Commodore, then onto Sinclair and Amstrad.

The formats I have chosen are actually the formats that Superior Software's excellent game, Repton, appeared on. The reason for this is that I am also responsible for writing the Windows utility application Repton Map Display and wanted to be able to extract the data direct from a disc image. I have included the Amiga Disc Format as the file extension is the same as Acorn ADFS, so I thought it appropriate to be able to distinguish between the two.

I have written this guide alongside writing a class in Delphi, TDiscImage, using various references I have found online, and adjusting appropriately where I have found the information incorrect/inaccurate. I suggest, if you want to write your own utility, that you arm yourself with some example disc images and a good hex dump application. There is one already built into MacOS, available via the Terminal – just type *hexdump -C "<filename>"*, or to dump to a text file, *hexdump -C "<filename>" > "<filename>.txt"*.

Since this guide was initially written, the Delphi class, TDiscImage, and the accompanying Windows application have been further developed allowing me to extend this guide. It is no longer developed in Delphi and has been ported across to Lazarus, which is free to download and use, and, hence, ported to MacOS and Linux, in addition to Windows. The source code is also available for those wishing to port it further, to whatever platform Lazarus will compile on. The project also has a new home at:

https://github.com/geraldholdsworth/DiscImageManager

where you will find a copy of this guide.

How the data is stored on an actual floppy, or hard, disc is beyond the scope of this guide - a disc image is just an electronic form of a physical disc. But, there are details in the RISC OS Programmer's Reference Manual, and also the clever folk at Stardot will also be able to help you, should you wish to know. To give you some idea of the extra data that is stored on an actual floppy (courtesy of Bill Carr) – this is the entire first track of The Big KO floppy disc:

```
THE BIG KO TRACK 0
------------------
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF       gap
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FE                                                    index mark?
00 00 00 00 00 01                                     index?
FE                                                    sector ID mark
00 00 07 01                                           track, head, sector,
                                                      size (sector ID)
68 44                                                 sector ID CRC
FF FF FF FF FF FF FF FF FF FF                         gap
00 00 00 00 00 00                                     sync bytes
FB                                                    data mark
...
[256 bytes]                                           data
...
EB CB                                                 data CRC
```


written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk 9 January 2021

```
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF          gap
00 00 00 00 00 00                                    sync bytes
FE                                                   sector ID mark
00 00 08 01                                          track, head, sector,
                                                     size (sector ID)
78 7A                                                sector ID CRC
FF FF FF FF FF FF FF FF FF FF                        gap
00 00 00 00 00 00                                    sync bytes
FB                                                   data mark
...
[256 bytes]                                          data
...
20 00                                                data CRC
...
sectors 09, 00, 01, 02, 03 ,04, 05
...
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF          gap
00 00 00 00 00 00                                    sync bytes
FE                                                   sector ID mark
00 00 06 01                                          track,   head,   sector,
                                                     size (sector ID)
5B 75                                                sector ID CRC
FF FF FF FF FF FF FF FF FF FF                        gap
00 00 00 00 00 00                                    sync bytes
FB                                                   data mark
...
[256 bytes]                                          data
...
05 85                                                data CRC
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF      gap
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF
DF
```

And, courtesy of Coeus on Stardot:

*“There are two fundamentally different ways to store floppy images for use in emulators. The simple way, which is adequate in many cases, is to store only the data in the image file. All the other information that would be on a real disc is omitted including all the track/sector IDs, the CRC, the gaps etc. When code running in the emulator issues a command to the floppy disc controller to read a sector the emulator performs a simple multiply calculation: track * sector * bytes-per-sector to seek to the appropriate place in the image file and reads the data there. This "data only" approach is what is used in the common SSD and DSD disc images and their ADFS versions (ADF, ADL etc.).*

The other approach is to encode everything that was on the original disc including the IDs, the gaps, even bits with dodgy levels. That can then cope with all the non-standard discs including sectors where the track number in the headers is not the same one as the physical track it is located on, bad CRCs, out-of-order sector numbers etc. I know FDI disc images take this approach but I don't know the internal details - you'd have to find the documentation.”

Each chapter has a section on how to identify the image given just the data (and ignoring the length of data, if possible). The methods I have given are not necessarily how the filing system in question identifies the disc, not is it how other programmers have done so in their code, but I feel is the best method.

Acknowledgements

A big thank you to Jasper Renow-Clarke (github.com/picosonic/bbc-fdc) and Russell King (code.woboq.org/linux/linux/fs/adfs) for their, on-going, input, and also to David Boddie for his Python code (https://www.boddie.org.uk/david/Projects/Python/ADFSlib/) which was also used as a reference along with Jonathan Harston’s mdfs.net page. A big thank-you, also, to the good denizens of the Stardot forum (www.stardot.org.uk). I also used, extensively, the RISC OS Programmer’s Reference Manual (page 2-200 onwards), and the BBC Master Reference Guide Part 1 (section J). For

the Commodore 64/128 formats, I used documents compiled and edited by Peter Schepers (ist.uwaterloo.ca/~schepers/formats.html). I used Laurent Clèvy's comprehensive document as a reference for the Commodore Amiga format (lclevy.free.fr/adflib/adf_info.html), but have written it in a way that I understand.

Note on Sinclair/Amstrad Format

This section has not been written yet, as I am currently still researching it fully.

Acorn Disc Filing System

Introduction

Even thought this section is entitled “Acorn Disc Filing System”, we are also covering a couple of other variants – namely Watford’s version which extended the catalogue so that up to 62 files per side could be saved on a disc, rather than Acorn’s 31; and Angus Duggan’s Hierarchical DFS (or HDFS). There were others, but they all pretty much kept to the same format described here.

The Acorn DFS evolved from the Acorn DOS, as used on the Acorn System 3 and Acorn Atom, which, as far as I can tell, is identical to that described here.

Specifications

The Disc Filing System was based on the 5¼“ floppy discs where the drives could either be 40-track drives or 80-track drives. Some drive manufacturers added a switch to their 80-track drives to double step them, allowing them to read 40-track discs. The number of tracks, therefore, contributed to the maximum storage capacity of the disc.

You could also get dual format discs, which were both 40 and 80 tracks on the same side. This is not covered by this document. There were also some double sided discs, where you took the disc out, turned it over, and re-inserted it, that had 40 track on one side, and 80 track on the other. These are commonly known as ‘flippies’.

The maximum, theoretical, capacity for one side of an Acorn DFS disc is 256KB.

Tracks	Sectors per Track	Sector Size	Storage
40	10	256 bytes	100KB
80	10	256 bytes	200KB

```
no_tracks = disc_size div (sec_size*secpertrack)
```

Although this section mentions tracks, file locations are given in sector numbers. Finding the offset address is a simple case of:

```
offset = sector * sec_size
```

The only time you need to worry about what track it is on, is when it is a double-sided disc image, and even then it can be worked out from the sector number.

Double Sided Interleaving

Double-sided disc images are stored where the tracks are interleaved. That is, you get ten tracks of side 1, followed by ten tracks of side 2, and so on. Converting from actual offsets, as given in the file details, to an offset into the image can then prove problematic…but it is not so:

Given an address into the disc, and the side it appears on:

```
sector = address DIV sec_size
offset = address MOD sec_size
ds_offset = (sector MOD sec_per_track) + (2 * sec_per_track * (sector DIV
sec_per_track)) + (sec_per_track * side)
```

Tracks 0 to 9, side 1 are followed by tracks 0 to 9, side 2. Then tracks 10 to 19 side 1, and so on. Catalogue information appears in sectors 0 and 1. This will be 0x0000 and 0x0100 for side 1 & 0x0A00 and 0x0B00 for side 2. Files are from sector 2 - i.e. 0x0200 for side 1 and 0x0C00 for side 2.

Maps

Catalogue information

The catalogue information, held on track 0, sectors 0 and 1, is as follows. Note that filenames and disc titles do not have their top bit set, as this is used for other purposes.

Offset	Length	Content
0x000	8	First eight characters of disc title (top bit not set), padded with spaces or zeros Top bit of 0x0000 if set, Number of sectors on disc bit 10 (HDFS)
0x008	8*31	31 file entries
0x100	4	Last four characters of disc title (top bit not set), padded with spaces or zeros

0x104	1	Master sequence number (Acorn DFS and Watford DFS); Key number (HDFS)	
0x105	1	Number of catalogue entries multiplied by 8 or, next free catalogue entry offset (from 0x0008/0x0108)	
0x106	1	**bits**	**Meaning**
		0-1	Number of sectors on disc bits 8+9
		2-3	Type of DFS: 0: Acorn DFS or Watford DFS 1: Watford DFS >256KB (actually bit 2 is number of sectors bit 10) 2: HDFS Single Sided (actually bit 2 is number of sides-1) 3: HDFS Double Sided (actually bit 2 is number of sides-1)
		4-5	Boot option
		6-7	unused
0x107	1	Number of sectors on disc bits 0-7	
0x108	8*31	31 file details	

Catalogue Information - Watford DFS

Watford created a variant of DFS that extended the number of files you could have, per side, from 31 to 62. In order to do this, they used the next two sectors to store the additional 31 entries.

Offset	**Length**	**Content**
0x200	8	8 bytes of 0xAA to indicate Watford DFS entries
0x208	8*31	31 file entries
0x300	4	4 bytes of 0x00 to indicate Watford DFS entries
0x304	1	Master Sequence Number for this section (not a copy of 0x104)
0x305	1	Number of catalogue entries*8
0x306	2	Copy of boot option and disc size
0x308	8*31	31 file details

File entries (sector 0/2)

Whether it is Acorn DFS, Watford DFS or HDFS, the file details are the same. Of course, the two variants have extra information hidden in the filename's top bits.

Offset	**Length**	**Content**
0x000	7	File name, padded with spaces (top bit not set)
		Top bit of each character:
		0x000: HDFS start sector bit 10
		0x001: HDFS length bit 18
		0x002: unused – should be clear
		0x003: HDFS file or directory
		0x004: HDFS file not readable
		0x005: Watford DFS length bit 18 HDFS file not writable
		0x006: Watford DFS start sector bit 10 HDFS file not executable
		0x007: File is locked
0x008	1	Directory prefix character

File details (sector 1/3)

These will be the load and execution addresses, along with the length and start sector for where the data occurs. The offset into an image is just simply sector * sec_size. Files on DFS are saved in single chunks, so there is no fragmentation occurring (note that some of this information for Watford and HDFS is held in the top bits of the filename).

Offset	**Length**	**Content**	
0x000	2	Load address bits 0-15	
0x002	2	Execution address bits 0-15	
0x004	2	Length bits 0-15	
0x006	1	**bits**	**Meaning**
		0-1	Start sector bits 8-9

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

	2-3	Load address bits 16-17
	4-5	Length bits 16-17
	6-7	Execution address bits 16-17
0x007	1	Start sector bits 0-7

File Order

The files on a DFS disc are not stored alphabetically, but by reverse sector number order. So the file right at the top of the disc, right next to the header (i.e. sector 2 for Acorn DFS and 4 for Watford DFS) is listed last.
When a file is deleted, only the catalogue information is removed while the data remains intact. This does, of course, leave the data open for being overwritten as DFS then considers this to be free space.

Forbidden Characters

Aside from the top-bit set characters (i.e. those above ASCII 127), and control characters (i.e. those below ASCII 32), there are some other forbidden characters. These are:
'**#**', '*****', '**:**', '**.**', and '**!**' – although these can be forced through using other utilities, but doing so may render the files inaccessible from DFS.

Identifying a DFS Image

There's not much to go on in order to positively identify a set of data is a DFS image. You will have to fall back on what is, and is not, allowed and make a few assumptions, based on the information given above:

Single Sided Images

- Offset 0x001 should have 9 bytes without top bit set and >31 or =0
- Offset 0x100 should have 4 bytes without top bit set and >31 or =0
- Offset 0x105 should have bits 0,1 and 2 clear
- Offset 0x106 should have bits 2,3,6 and 7 clear

The following will fail if the image is double sided:

- Offset 0x107 + bits 0,1 of 0x106, multiplied by sector size of 0x100 should equal the image size

Note that the image size is not always the exact size of the disc, so comparing this to the number of sectors reported in the disc header, multiplied by 0x100, will not always work. However, the number of sectors should be divisible by 10 (10 sectors per track), and dividing the sectors by 10, to reveal the number of tracks should be either 40 or 80, per side.

Therefore:
```
([0x107] + (([0x106] AND 0x3)<<8)) MOD 10 = 0
```
and
```
([0x107] + (([0x106] AND 0x3)<<8)) DIV 10 = 40
```
or
```
([0x107] + (([0x106] AND 0x3)<<8)) DIV 10 = 80
```
However, it should be noted that there are valid DFS discs that have other numbers of tracks, so rejecting an image because it is not 40 or 80, or the number of tracks is not divisible by 10, does not mean it is not valid. It should also be noted that not all double sided discs have the same number of tracks either side (back in the day, many commercial titles were released with 80 track version on one side and 40 on the other, including the BBC Master Welcome Disc).

Another check you could employ is add up all the free space, and the file entries, to ensure the total equals the supplied disc size.

Double Sided Images

- Do the single sided checks above, but add 0xA00 to the offsets and repeat

The following will fail if the image is single sided:

- Offset 0x107 + bits 0,1 of 0x106, multiplied by the sector size of 0x100 plus Offset 0xB07 + bits 0,1 of 0xB06, multiplied by the sector size of 0x100 should equal the total image size, but see above on note about disc image sizes.

Watford DFS

In addition to the checks above:

- Offset 0x200 should contain 8 characters of 0xAA
- Offset 0x300 should contain 4 characters of 0x00
- Offset 0x305 should bits 0,1 and 2 clear
- Offset 0x306/7 should equal offset 0x106/7

Acorn Advanced Disc Filing System

Introduction

The Acorn Advanced Disc Filing System, or ADFS, evolved from the Acorn Disc Filing System, in order to add Winchester Hard Drives to the BBC Micro. Introduced with the BBC B+, Master series and Acorn Electron Plus 3. Since then it has developed and evolved and now (since RISC OS on the Archimedes) is the main part of FileCore, even on the Raspberry Pi running the latest RISC OS 5. The ADFS module in RISC OS, along with all other native filing systems, use the FileCore routines. Here I refer to ADFS but in actual fact is also the RISC OS FileCore format.

As far as image files are concerned, ADFS uses the .adf file extension. It has also been known to use the .adl (for L format), amongst other variations. This is also the format used for hard discs on RISC OS (and BBC before that), but we are only concerning ourselves with the floppy format. However, the theory is the same.

This chapter is laid out thus: the system of fragments is first described, followed by the directory layouts, although the directory system will be mentioned with the fragments (as directories form the basis of the Advanced Disc Filing System. We then end with the directory entries, the files themselves.

Before the explanation, I'll first do the technical bit and provide the specs, from which the explanations will be based on. According to the Programmer's Reference Manual (PRM), these describe the 'perfect' floppy disc. In my opinion, these are just official shapes of floppy discs:

Format	Map	Zones	Directories	Boot Block	Density	Sectors per Track	Bytes per Sector	Heads	Storage	Tracks	Root
S	Old	n/a	Old	No	Double	16	256	1	160KB	40	0x200
M	Old	n/a	Old	No	Double	16	256	1	320KB	80	0x200
L	Old	n/a	Old	No	Double	16	256	2	640KB	80	0x200
D	Old	n/a	New	No	Double	5	1024	2	800KB	80	0x400
E	New	1	New	No	Double	5	1024	2	800KB	80	0x800
E+	New	1	Big	No	Double	5	1024	2	800KB	80	0x800
F	New	4	New	Yes	Quad	10	1024	2	1.6MB	80	See DR
F+	New	4	Big	Yes	Quad	10	1024	2	1.6MB	80	See DR
G	New	7	Big	Yes	Octal	20	1024	2	3.2MB	80	See DR

Storage size can be calculated thus:

```
BytesPerSector*SectorsPerTrack*Tracks*Heads
```

In actuality, there are really only four different types of format:

Disc Shapes	Map Type	Directory Type
ADFS S/M/L	Old	Old
ADFS D	Old	New
ADFS E/F	New	New
ADFS E+/F+	New	Big

And these can describe any type of ADFS disc, or image – this includes floppy discs and hard drives.

Double Sided Interleaving

Like DFS, in the previous chapter, ADFS images can also be interleaved. However, there is no way of determining whether an image is interleaved or not. But most of the images I have seen, only ADFS L shapes are interleaved (with extension adl, but not always so).

To convert from disc address to image offset, and vice versa, we will need to know the track size (incidentally, what I am about to show you will also work with ADFS S and M):

```
TrackSize = SectorsPerTrack * BytesPerSector
```


written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

For an ADFS L, this will be:

```
TrackSize = 16 * 256 = 4096
```

Then, given an offset into an image file, we have:

```
Track = offset DIV (TrackSize * Heads)
Side = (offset MOD (TrackSize * Heads)) DIV TrackSize
DataOffset = offset MOD TrackSize
DiscAddress = (Track * TrackSize) + (Tracks * TrackSize * Side) + DataOffset
```

And, to go the other way, given a disc address, we have:

```
Side = DiscAddress DIV (Tracks * TrackSize)
Track = (DiscAddress DIV TrackSize) MOD Tracks
DataOffset = DiscAddress MOD TrackSize
offset = (TrackSize * Side) + (Track * TrackSize * Heads) + DataOffset
```

Maps

Old Map

Offset	Name	Bytes	Meaning
0x000	FreeStart	82*3	Table of free space start sectors
0x0F6	Reserved	1	Reserved - must be zero
0x0F7	OldName0	5	Half disc name (interleaved with OldName1)
0x0FC	OldSize	3	Disc size in (256 byte) sectors
0x0FF	Check0	1	Checksum on first 256 bytes
0x100	FreeLen	82*3	Table of free space lengths
0x1F6	OldName1	5	Half disc name (interleaved with OldName0
0x1FB	OldId	2	Disc id
0x1FD	OldBoot	1	Boot option (as in *Opt 4,n)
0x1FE	FreeEnd	1	Pointer to end of free space list
0x1FF	Check1	1	Checksum on second 256 bytes

This is then followed by the root directory, which will always follow at the start of the next sector boundary (0x200 for S/M/L, and 0x400 for D, which has a bigger sector size). Note that S, M and L format discs do not use the disc name and OldName0 and OldName1 fields will be ignored for these formats, on a BBC/Electron. If a name is entered, it must be padded with spaces to the ten characters.

Calculating the Checksums

This calculation is a simple case of adding up all of the bytes from 0x0FE down to 0x000, or 0x1FE down to 0x100. Start with a carry set to zero, then add this carry, reset it to zero, followed by adding the byte. If, when adding this byte, it takes the total over 0xFF, set the carry to 1 (and truncate the total to under 0x100, i.e. AND with 0xFF). This means that if the final byte produces a carry, this will not be added to the total.

New Map

Zone Header

Offset	Name	Bytes	Meaning
0x00	Zone_Check	1	Check byte for this zone's map block
0x01	FreeLink	2	Link to first free fragment in this zone
0x03	CrossCheck	1	Cross check byte for complete map

Following the zone header, in zone 0, will be the complete, 60-byte, Disc Record.

Disc Record (zone 0 only)

Offset	Name	Bytes	Meaning
0x00	log2secsize	1	sector size of disc in bytes (1<<log2secsize)
0x01	secspertrack	1	Number of sectors per track
0x02	heads	1	Number of disc heads if sides interleaved Number of disc heads - 1 if sides sequenced (1 for old directories)
0x03	density	1	**Value** **Meaning**

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

			0 hard disc 1 single density (125Kbps FM) 2 double density (250Kbps FM) 3 double+ density (300Kbps FM) 4 quad density (500Kbps FM) 8 octal density (1000Kbps FM)
0x04	idlen	1	Length of id field of a map fragment, in bits
0x05	log2bpmb	1	number of bytes per map bit (1<<log2bpmb)
0x06	skew	1	Track to track sector skew for random access file allocation
0x07	bootoption	1	Boot option (as in *Opt 4,n)
0x08	lowsector	1	**bit** **Meaning** 0 - 5: lowest numbered sector id on a track 6: if set, treat sides as sequenced (rather than interleaved) 7: if set, disc is 40 track
0x09	nzones	1	Number of zones in the map
0x0A	zone_spare	2	Number of non-allocation bits between zones
0x0C	root	4	Disc address of root directory
0x10	disc_size	4	Disc size, in bytes
0x14	disc_id	2	Disc cycle id
0x16	disc_name	10	Disc name
0x20	disctype	4	File type given to disc
ADFS E+ and F+ only:			
0x24	disc_size2	4	High word of disc size
0x28	log2share_size	1	Share size information (1<<log2share_size)
0x29	big_flag	1	**bit** **meaning** 0 RISC OS partition >512MB 1-7 zero
0x2A	nzones_2	1	MSB of number of zones
0x2B	reserved	1	must be zero
0x2C	format_version	4	Version number of disc format 1 is extended, or '+', formats with big directories
0x30	root_size	4	Size of root directory

All unused bytes must be zero. For old map discs, bytes 0x04-0x0B inclusive must be zero.
Partial disc record can be found in the boot block on F and F+ discs. The boot block is always at 0xC00, and the disc record 0x1C0 into this (i.e. 0xDC0). On E and E+ discs, the full disc record will be at offset 0x04 (with the header at 0x00).

Calculating Zone_Check

The code given in the PRM is, in C++:

```
unsigned char map_zone_valid_byte
(
void const * const map, disc_record const * const discrec, unsigned int zone
)
{
	unsigned char const * const map_base = map;
	unsigned int sum_vector0;
	unsigned int sum_vector1;
	unsigned int sum_vector2;
	unsigned int sum_vector3;
	unsigned int zone_start;
	unsigned int rover;
	sum_vector0 = 0;
	sum_vector1 = 0;
	sum_vector2 = 0;
	sum_vector3 = 0;
	zone_start = zone<<discrec->log2_sector_size;
	for ( rover = ((zone+1)<<discrec->log2_sector_size)-4 ;
	rover > zone_start; rover-=4 )
	{
		sum_vector0 += map_base[rover+0] + (sum_vector3>>8);
```


written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

```
                sum_vector3 &= 0xff;
                sum_vector1 += map_base[rover+1] + (sum_vector0>>8);
                sum_vector0 &= 0xff;
                sum_vector2 += map_base[rover+2] + (sum_vector1>>8);
                sum_vector1 &= 0xff;
                sum_vector3 += map_base[rover+3] + (sum_vector2>>8);
                sum_vector2 &= 0xff;
        }
/*Don't add the check byte when calculating its value*/
        sum_vector0 += (sum_vector3>>8);
        sum_vector1 += map_base[rover+1] + (sum_vector0>>8);
        sum_vector2 += map_base[rover+2] + (sum_vector1>>8);
        sum_vector3 += map_base[rover+3] + (sum_vector2>>8);
        return (unsigned char)
        ((sum_vector0^sum_vector1^sum_vector2^sum_vector3) & 0xff);
}
```

What is happening here is a 32bit add-with-carry on all the words from four bytes off the end of the zone, down to the second word at the start of the zone, leaving the first word to be added separately, as it includes the check byte, which is not included. Instead, this byte is substituted with zero and the whole word added as before to the total. Finally, the four separate bytes of the word are then exclusively OR-ed (XOR) together, and ANDed with 0xFF to ensure that it is no bigger than 8 bits.

Allocation Bytes and Indirect Addresses

New Maps are laid out, well, like a map. This map is scaled down so it can be fitted into a sector. The scale is defined by the bpmb entry in the disc record (1<<log2bpmb), where 1 bit in the map represents bpmb number of bytes (bpmb = bytes per map bit). The file objects (i.e. files and directories) are split into 1 or more fragments so that they fit into the disc. These fragments are given IDs, which are labelled on the map. The distance the beginning of these IDs is from the beginning of the map indicates how far into the disc they are. The end of the fragment is marked by a bit being set (after the ID). The distance from the beginning of the ID to, and including, this bit represents the length of the fragment. A file object may be split into more than one fragment and, as such, will have the same fragment ID appearing more than once in the map.

As the map is contained within an entire sector, and the capacity of the discs gets bigger, either the scale will need to be adjusted accordingly, or more sectors will need to be used. As it happens, Acorn chose the latter. Instead of a map spreading across sector boundaries, Acorn chose to use zoning. This is where the map is then split into different zones, which can be likened to being spread across several pages. Each zone is then confined to single sectors, but the scale can then be adjusted to zoom in a little. However, multi-zone maps will be discussed later.

Acorn refer to this layout of the fragments as the Allocation Bytes, and, on E and E+ format discs, appears at adf offset 0x0040 - this is the four-byte zone header followed by the 60-byte disc record.

The file objects' fragment IDs are indicated in the directory entries, and are listed as 'Indirect Address'. This is also the same for the root, which is specified in the disc record. This Indirect Address is a three-byte value (four-bytes with Big Directories), which breaks down to:

```
0fffffff ffffffff ssssssss
```

where f is the fragment ID and s is the sector offset. This is assuming an idlen of 15, which specifies how long the fragment ID can be - the maximum possible, under RISC OS 5.23 currently, is 21 bits (hence the four-byte indirect address with Big Directories). The top three bits of a four-byte indirect address will be the disc number (the three-byte address is made up to four bytes by the addition of the disc number).

So, now we shall go through an actual disc and I will try and make this clearer. Below is a hex dump of the first 0x4F bytes of the EGO: Repton 4 disc, which is an 'E' format disc. I will start by showing how to find the root directory – OK, we already know this is at 0x800 for an 'E' format, but this will also help to illustrate the above.

```
Offset    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Characters
00000000  0C 00 80 FF 0A 05 02 02 0F 07 01 00 00 01 20 05  ..?˘.......... .
00000010  03 02 00 00 00 80 0C 00 4E 01 52 65 70 74 6F 6E  .....?..N.Repton
```


written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk 9 January 2021

```
00000020  34 20 20 20 00 00 00 00 00 00 00 00 00 00 00 00  4   ............
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000040  02 00 00 80 03 80 05 80 06 80 07 80 08 00 00 00  ...?.?.?.?.?....
```

Offsets 0x00 to 0x03 are the zone header, and then offsets 0x04 through to 0x3F are the 60-byte disc record. Both of these are detailed earlier in this chapter. We know that the root address is given in the disc record, and this can be found above at offset 0x10 to 0x12 inclusive, and is given as 0x000203 (and is highlighted). If we use the format given earlier, 0ffffffff ffffffff ssssssss, 0x0002 is the fragment ID, and 0x03 is the sector offset.

We now need to find, in the map, the fragment ID. The map begins at offset 0x40 (I have only shown you the first 16 bytes of the map, for now). Highlighted, right at the start, is fragment 0x0002 – you will also notice the 0x80, but more on that in a bit. As we are starting to look from offset 0x40, the offset will be 0x00. The sector offset is given as 0x03.

We can then use these two pieces of information to work out the root address. We will also need the log2bpmb value, as well as the log2secsize value. These are given as 0x07 (offset 0x09 in the table above) and 0x0A (offset 0x04 in the table above) respectively. As these are 'log2' values, we need to raise 2 to these powers to get the actual value, or shift 1 left this number of times:

```
1<<0x0A = 2^0x0A = 1024
```

and

```
1<<0x07 = 2^0x07 = 128
```

So, to get the block that the fragment starts in, we just multiply the offset (distance from the beginning of the map to the fragment ID) by the bpmb. However, you may have spotted that bpmb stands for "bytes per map bit", the map is in bits, and we are working in bytes (although, with an offset of 0x00, this makes no difference anyway). But, we need to multiply this byte offset by 8, and by the bpmb value (or shift it left log2bpmb times).

We also need to factor in the sector_offset we have been given. This is because a block can contain many fragments but will be no bigger than a sector. As this sector_offset is the start of the data, we need to take 1 away from it, unless it is already 0, then multiply it by the secsize (or shift left by log2secsize times). This would be the

```
(sector_offset-1)*secsize
```

or

```
(sector_offset-1)<<log2secsize
```

So, for the root shown above, offset of the fragment ID is at 0x40, start of the map is also 0x40, sector_offset is 3, bpmb is 128, and secsize is 1024:

```
address = offset * 8 * bpmb + (sector_offset-1)*secsize
address = (0x40-0x40)*8*128+(3-1)*1024 = 2048 or 0x800
```

The '8' is because, as stated, the offsets should be counted in bits, not bytes.

I mentioned about the 0x80 at the end of the fragment. This relates to the length of the fragment – the first bit set indicates how long the fragment is, in bits, from the start of the fragment ID. Remember the start of the fragment ID is 0x00 bits from the start of the map, and the first bit set after this is 0x1F bits after this, making the length 32 bits. Multiply this by the bpmb (or shift it left log2bpmb times) and you will have the length of the root directory:

```
32xbmpb=32x1<<7=32x128=4096=0x1000 bytes
```

Remember that the root will usually follow the allocation map (two copies), so the root_address entry in the disc record can seem largely redundant. As a rule of thumb, E and E+ format discs will have the root at offset 0x800, S/M/L at 0x200 and D at 0x400 - F and F+ require a few more calculations and will be explained later. However, this is not always the case and is best to check the location specified in the disc record.

Example, using the same disc

I will go through some more examples, from the same disc. This time we are looking at the root directory and finding the files and sub-directories that it contains. So, as before:

log2secsize (at adf offset 0x04) is 0x0A, meaning a sector size of 2^10 = 1<<10 = 1024 or 0x400

log2bpmb (at adf offset 0x09) is 0x07, meaning a byte per map bit size of 2^7 = 1<<7 = 128 or 0x80

Root address is 0x000203 (given at offsets 0x10,0x11,0x12) : fragment ID 2, offset 3 - ID 2 is sector 0x40, therefore physical location is (0x40-0x40)*128 + (3-1)*0x400 = adf offset of 0x800, as we have seen already. Below is another 16 bytes of the allocation map, along with the first 16 bytes for reminder, plus the hex dump of the header and entries for the root:

```
Offset    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Characters
00000040  02 00 00 80 03 80 05 80 06 80 07 80 08 00 00 00  ...?.?.?.?.?....
:
00000350  12 80 13 00 00 00 00 00 80 04 00 00 00 00 00 80  .?......?......?
:
00000800  21 4E 69 63 6B 21 52 65 70 74 6F 6E 34 0D 00 44  !Nick!Repton4..D
00000810  FD FF FF BC AB 4C 44 00 08 00 00 00 03 00 08 50  ˝˘˘°´LD........P
00000820  61 73 73 77 6F 72 64 73 0D 44 FF FF FF 17 83 C0  asswords.D˘˘˘.?¿
00000830  74 B9 00 00 00 01 12 00 03 50 61 74 63 68 0D 74  tπ.......Patch.t
00000840  69 63 6B 4C FB FF FF 44 73 7A 8B E1 00 00 00 02  ickL˚˘˘Dsz?·....
00000850  12 00 03 56 54 4A 6F 79 73 74 69 63 6B 43 FA FF  ...VTJoystickC˙˘
00000860  FF 65 A0 E6 EF CC 17 00 00 00 13 00 03 00 00 00  ˘e†ÊÔÃ..........
```

Indirect addresses are highlighted, as they appear in the directory block, from offset 0x800. For the fragment IDs, you will need to look at the allocation map – part of this is in the table above, at offsets 0x040 and 0x350. The fragment IDs are coloured, but not highlighted. The stop bit is included in the colouring above to show the length of the fragment.

Directory "!Repton4"

Details from offset 0x805. Indirect address appears at offset 0x81B.

indirect address is 0x000300: fragment ID 3, sector 0 - ID 3 is offset 0x044.

```
(0x044-0x40)*8*128 + 0 *0x400 = adf offset of 0x01000
```

File "Passwords"

Details from offset 0x81F. Indirect address appears at offset 0x835.

indirect address is 0x001201: fragment ID 12, sector 1 - ID 12 is offset 0x350.

```
(0x350-0x40)*8*128 + (1-1)*0x400 = adf offset of 0xC4000
```

File "Patch"

Details from offset 0x839. Indirect address appears at offset 0x84F.

indirect address is 0x001202: fragment ID 12, sector 2 - ID 12 is also offset 0x350.

```
(0x350-0x40)*8*128 + (2-1)*0x400 = adf offset of 0xC4400
```

File "VTJoystick"

Details from offset 0x853. Indirect address appears at offset 0x869.

indirect address is 0x001300: fragment ID 13, sector 0 - ID 13 is offset 0x352.

```
(0x352-0x40)*8*128 + 0 *0x400 = adf offset of 0xC4800
```

Multi-zone discs

So, now we have the hang of how the indirect addressing works, with the fragments on the allocation map, we can move into multi-zones, as used on the 'F' and 'F+' format discs, and hard discs. If you still have not got the hang of this, I suggest you read no further.

When nzones is greater than 1, things become more complicated. The disc record is always held in zone 0, when there is only the single zone, this will also include the 4 byte zone header, allocation map, and root. When there is more than a single zone, there is a partial disc record held in the boot block at 0xC00. This disc record is 0x1C0 into the boot block, i.e. at offset 0xDC0.

The full disc record, map and root are held in the middle zone of a multi-zone disc, at the beginning of a sector. Finding this can prove difficult, however the following pseudo-code can be followed:

```
dr_size=60 (disc record size)
if nzones>2 then zz=dr_size*8 else zz=0
```

(we're working in bits here, but don't need if there's only the 1 zone)

```
map_address=((nzones div 2)*(8*secsize-zone_spare)-zz)*bpmb
map_start=map_address+4+dr_size
root_address=map_address+nzones*secsize*2
```

or

```
root_address=map_address+(nzones<<(log2secsize+1))
```


written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk 9 January 2021

The rationale behind these calculations is, using an example F format disc:

1. We need the sector size in bits, so we use
   `8*secsize = 8*0x400=0x2000`
   Note that we can also use `8<<log2secsize` instead, or even `1<<(log2secsize+3))`
2. We need to remove the spare bits between zones to find out the zone size. Each zone in the allocation map is a single sector, and zone_spare is the number of bits between the end of the previous zone and the end of the current zone's header.
   `8*secsize-zone_spare = 0x2000-0x640=0x19C0`
3. The map is located in the middle of the disc.
   `nzones div 2 = 4 div 2=2`
4. Find the middle zone
   `(nzones div 2)*(8*secsize-zone_spare) = 2*0x19C0=0x3380`
5. The first zone has the disc record, so has fewer bits than the other zones, so we need it in bits.
   `dr_size*8 = 60*8=480`
6. And remove it from the total
   `(nzones div 2)*(8*secsize-zone_spare)-(dr_size*8) = 0x3380-480=0x31A0`
7. If we then multiply this by the bytes per map bit, we get the address of the map
   `map_address=0x31A0*0x40=0xC6800`
   As before, we can shift the entire result left by log2bpmb instead of multiplying by bpmb to achieve the same result
8. The start of the map follows the 4-byte zone header and the 60 byte disc record.
   `map_start=map_address+0x40 = 0xC6800+0x40=0xC6840`
9. And finally the Root Address. On the map, each zone takes up a sector, and there are two copies of the map. The root will always follow the map.
   `nzones*secsize*2=4*0x400*2=0x2000`
   `root_address=0xC6800+0x2000=0xC8800`

This then complicates finding the appropriate fragment ID, as we now have more than one zone. Each zone, as stated before, takes up a single sector each, and will have a 4-byte zone header. Remember, there are zone_spare bits between end of previous zone and end of header for current zone and zone 0 has the 60-byte disc record, in addition to its 4-byte header.

The offset of the fragment ID still works as before, and we still use map_start to calculate the offset from (rather than the beginning of the zone). Therefore for fragment id at offset ID_offset, where zone is the zone it appears in:

`disc_address=((ID_offset-map_start)-(zone_spare/8)*zone)*8*bpmb`
or we can use:
`disc_address=((ID_offset-map_start)-(zone_spare/8)*zone)<<(log2bpmb+3)`
As before, we then add on the sector offset as `(sector-1)*secsize` or `(sector-1)<<log2secsize`, where sector>0

Example

An ADFS F disc, where zone_spare is 0x640, bpmb is 64, nzones is 4, and map_start is 0xC840. The beginning of each zone is zone 0 at 0xC6840, 1 at 0xC6C00, 2 at 0xC7000 and 3 at 0xC7400. Below is a hex dump of parts of allocation maps, on three different zones, and the root directory header and entries of this 'F' format disc:

```
Offset    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Characters
000C6D10  00 80 A1 01 00 80 A2 01 00 80 A3 01 00 80 A4 01  .?°..?¢..?£..?§.
:
000C7010  00 00 00 00 00 00 00 80 38 03 00 80 39 83 3A 03  .......?8..?9?:.
000C7020  00 80 3B 83 3C 03 00 80 3D 83 3E 03 00 80 3F 03  .?;?<..?=?>..??.
:
000C7710  F8 04 00 80 F9 84 FA 84 01 00 00 00 00 00 00 00  ¯..?ˇ?˙?........
:
000C8800  04 4E 69 63 6B 21 44 65 73 6B 44 75 63 6B 0D 4A  .Nick!DeskDuck.J
000C8810  FD FF FF 04 79 5B A7 00 08 00 00 00 F8 04 88 21  ˝ˇˇ.y[ß.....¯.?!
000C8820  53 79 73 74 65 6D 0D 00 00 4A FD FF FF EE CB 56  System...J˝ˇˇÓÀV
000C8830  A7 00 08 00 00 00 38 03 18 00 00 00 00 00 00 00  ß.....8.........
```

Fragment 01A2 is at 0xC6D16, putting it in zone 1:

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

`0xC6D16-0xC6840=0x4D6-(0x640/8)*1=0x40E*8*64=0x081C00`
Fragment 0338 is at 0xC7018, putting it in zone 2 (***Directory "!System"***):
`0xC7018-0xC6840=0x7D8-(0x640/8)*2=0x648*8*64=0x0C9000`
Fragment 033D is at 0xC7028, putting it in zone 2:
`0xC7028-0xC6840=0x7E8-(0x640/8)*2=0x658*8*64=0x0CB000`
Fragment 04F8 is at 0xC7710, putting it in zone 3 (***Directory "!DeskDuck"***):
`0xC7710-0xC6840=0xED0-(0x640/8)*3=0xC78*8*64=0x18F000`
Note that these examples, and the calculations, have been made using the offsets, etc., as bytes. Remember that the allocation maps are laid out in a bitstream, with idlen and zone_spare being specified in bits. Doing the calculations in bytes works OK when idlen is 15 (as the minimum fragment length will be 15 bits + 1 stop bit = 16 bits = 2 bytes), but when the idlen can be up to 21 bits, you will need to find the offsets in bits, and do the calculations the same.

This makes the above calculation:
`disc_address=((ID_offset-map_start)-zone_spare*zone)*bpmb`
where ID_offset and map_start are in bits, and not bytes.
An alternative to finding fragments is to start your counter at zero, thereby removing the need to use map_start in the calculation. If we then translate this into the example above, but remove the map_start and use bits instead of bytes:
Fragment 01A2 is at offset 0x26B0, putting it in zone 1:
`0x26B0-0x640*1=0x2070*64=0x081C00`
Fragment 0338 is at offset 0x3EC0, putting it in zone 2 (***Directory "!System"***):
`0x3EC0-0x640*2=0x3240*64=0x0C9000`
Fragment 033D is at offset 0x3F40, putting it in zone 2:
`0x3F40-0x640*2=0x32C0*64=0x0CB000`
Fragment 04F8 is at offset 0x7680, putting it in zone 3 (***Directory "!DeskDuck"***):
`0x7680-0x640*3=0x63C0*64=0x18F000`

Start and End of Search

The start and end of your search for the fragments will differ per zone. Using the 'F' format, laid out above, the zones will be:

Zone	Start	Bit offset	End	Bit offset
0	0xC6840	0x0000	0xC6C00	0x1E00
1	0xC6C04	0x1E20	0xC7000	0x3E00
2	0xC7004	0x3E20	0xC7400	0x5E00
3	0xC7404	0x5E20	0xC7800	0x7E00

So why these figures?

The start of the map is at 0xC6800, but each zone has a four-byte header. This would bring zone 0 to 0xC6804, zone 1 to 0xC6C04, zone 2 to 0x7004 and zone 3 to 0xC7404. In addition, zone 0 also has the 60-byte disc record, which then moves the start of the search to 0xC6840.

There is a simple calculation for this also, or rather, some pseudo code:

```
start=zone*secsize
if zone>0 then start=start-dr_size
end=(zone+1)*secsize-(dr_size+header)
```

Where start and end are the bit offsets of the beginning and end of the search, zone is the zone you are searching, dr_size is the disc record size in bits (480), header is the zone header size in bits (32), and secsize is the sector size in bits.

Once you have found the fragment (of which there may be many), the length is then given by the number of bits until the next '1'. That is, after the fragment id, there will be a number of '0's followed by a '1'. The length of the fragment is `idlen+number_of_zeros+stop_bit`. To translate this into number of bytes on the disc, just multiply this by bpmb (or shift it left by log2bpmb).

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

Where to Start

You would think that you start your search from Zone 0, and carry on until you reach the end of, in this case, Zone 3. However, this is not always the case. The first fragment of a file object can appear in higher zones.

To know where to start, you will need to know the maximum number of fragments that can appear in any one zone. Note that Zone 0 will have a lower number due to having the disc record present, but as we are working with averages here, this does not come into it.

```
id_per_zone = ((1 << (log2secsize + 3)) - zone_spare) div (idlen + 1)
```

or

```
id_per_zone = ((secsize*8) - zone_spare) div (idlen+1)
```

idlen+1 is the smallest size a fragment can be. secsize*8 is the size of the sector, in bits. We need to remove the zone_spare from this, as this is not used.
From this, we can work out the initial zone that the fragment appears in:

```
start_zone = fragment_id div id_per_zone
```

An example, given in the source code for Filecore on RISC OS 5.23 may make it clearer:

A fragment ID is 0x2B2. The log2_sector_size is 8 (256 bytes per sector), the idlen is 11 and zone_spare is 4.

```
id_per_zone = (0x100*8 - 4) div 0xC = 0xA8
start_zone = 0x2B2 div 0xA8 = 4
```

Therefore, the starting zone for this object is zone 4.

Zone	Fragments in zone	Position of fragments in object
0	3	3
		4
		5
1	0	
2	1	6
3	1	7
4	2	0
		1
5	0	
6	1	2
7	0	

This object has 8 fragments, identified as belonging to this object by having a fragment id of 0x2B2. They are ordered in the object as follows:

Position of fragment in object	Zone the fragment is in
0	4
1	4
2	6
3	0
4	0
5	0
6	2
7	3

The first fragment in an object is the first fragment on the disc searching from zone (fragment_id div ids_per_zone) upwards, wrapping round from the disc's end to its start. Any subsequent fragments belonging to the same disc object are joined in the order they are found by this search.

This can be made simple by using the MOD operator in most programming languages:

```
FOR zone = 0 TO nzones-1
       search_zone = (zone + start_zone) MOD nzones
       :
       :
NEXT
```


written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

Directories

As before, we will start with the specifications. Each directory has a header, content and a tail. This differs between the various formats that there have been of ADFS. They have been labelled 'Old' (for ADFS S/M/L), 'New' (for ADFS D/E/F) and 'Big' (for ADFS E+/F+). There are also some subtle differences between the three:

Directory	Max Entries	Size (bytes)	Top Bit Set Characters
Old	47	1280	No
New	77	2048	Yes
Big	variable	variable	Yes

Top Bit Set Characters indicates whether the directory allows for characters with an ASCII value >127. As you will see, Old directories used the top bit of the filename for other purposes.

It is also worth pointing out that all objects (either files or directories) on old map images should start on a sector boundary. This should not be much of an issue for S, M or L shapes as the free map works around the 256-byte sector. However, for the D shape, this has 1024 bytes per sector. With that said, I have found that ADFS in RISC OS will happily load in a file not on a sector boundary. But it will not read in a directory which is not sector aligned (you will get a Broken Directory error).

Header (Old & New)

Offset	Name	Bytes	Meaning
0x00	StartMasSeq	1	Update sequence number to check dir start with dir end
0x01	StartName	4	'Hugo' or 'Nick'

Header (Big)

Offset	Name	Bytes	Purpose
0x00	StartMasSeq	1	Sequence number.
0x01	BigDirVersion	3	Version number (reserved, must be 0)
0x04	BigDirStartName	4	4 characters "SBPr"
0x08	BigDirNameLen	4	Length of directory name
0x0C	BigDirSize	4	Length of directory, in bytes
0x10	BigDirEntries	4	Number of entries in directory
0x14	BigDirNamesSize	4	Number of bytes allocated for names
0x18	BigDirParent	4	Indirect disc address of parent dir
0x1C	BigDirName	BigDirNameLen+1	Name of directory, plus CR terminator.

Null padded to word boundary.

Tail (Old)

Offset	Name	Bytes	Meaning
0x00	OldDirLastMark	1	0 to indicate end of entries
0x01	OldDirName	10	Directory name
0x0B	OldDirParent	3	Disc address of parent directory
0x0E	OldDirTitle	19	Directory title
0x21	Reserved	14	Reserved - must be zero
0x2F	EndMasSeq	1	To match with StartMasSeq
0x30	EndName	4	'Hugo' or 'Nick', to match with StartName
0x34	DirCheckByte	1	Check byte on directory

Tail (New)

Offset	Name	Bytes	Meaning
0x00	NewDirLastMark	1	0 to indicate end of entries
0x01	Reserved	2	Reserved - must be zero
0x03	NewDirParent	3	Indirect disc address of parent directory
0x06	NewDirTitle	19	Directory title
0x19	NewDirName	10	Directory name
0x23	EndMasSeq	1	To match with StartMasSeq
0x24	EndName	4	'Hugo' or 'Nick', to match with StartName

0x28	DirCheckByte	1	Check byte on directory

Tail (Big)

Offset	Name	Bytes	Meaning
0x00	BigDirEndName	4	4 characters, 'oven'
0x04	BigDirEndMasSeq	1	To match with StartMasSeq
0x05	Reserved	2	Must be zero
0x07	BigDirCheckByte	1	Check byte on directory

What It All Means

Some of these values are important to the valid structure of the directory. Others, such as the Directory Name, are arbitrary and are specified by the user (except for '$', of course).

StartMasSeq and EndMasSeq

These appear in both the header and the tail, and must match. The value is unimportant, but generally, it starts at zero and is increased every time the directory is written to. StartMasSeq is updated before the write, with EndMasSeq happening last. This means that if the write fails in the middle, the result will be a 'Broken Directory' (i.e. they do not match).

StartName and EndName

These, which are the signature to a directory, also must match. There is nothing in the PRM, or in any documentation that I can find, that states what happens if they do not match, but tests I have performed by changing one resulted in a 'Broken Directory'. These should be 'Hugo' (after Hugo Tyson) for S/M/L and 'Nick' (after Nick Reeves) for 'D/E/F'. The only times that these will differ is with E+ and F+ where they will be 'SBPr' and 'oven' (after Simon Proven) respectively.

DirLastMark

This should be zero. This will mark the end of the entries. If the directory is full, this marker will still be there.

DirCheckByte

This is a checksum to ensure that the directory structure is valid. This a simple calculation operating on most of the entire structure (except for the empty parts), Exclusively ORing words and bytes, which are rotated. The algorithm, given in the PRM, but where I have edited the description after finding the PRM is not terribly accurate, is:

Starting at 0 an accumulation process is performed on a number of values. Whatever the sort of the value (byte or word), it is accumulated in the same way. With Big Directories, remember to include the name heap in your calculations. Assuming dircheck is the accumulation variable and value is the value read this is the accumulation performed:

```
dircheck = value XOR (dircheck ROR 13)
```

or

```
dircheck = value XOR ((dircheck >> 13) OR (dircheck << 19))
```

1. All the whole words at the start of the directory are accumulated. This will leave a number of bytes (0 to 3) in the last directory entry (or at the end of the directory header if it is empty).
2. The last bytes (<4 bytes) at the start of the directory are accumulated individually.
3. Old and New: The first byte at the beginning of the directory tail is skipped. This is done to leave only a whole number of words left in the directory to be accumulated.
   Big: This is included with Big Directories, which does not have the DirLastMark value.
4. Old and New: All the whole words in the tail are accumulated, except the very last word, which is excluded as it contains the check byte.
   Big: The final 3 bytes of the Tail, excluding the final byte, are accumulated.
5. The accumulated word has its four bytes exclusive ORd (XOR) together. This value is the check byte.

Note that this is a reserved field on S, M and L format discs and must be zero. No checking is required.

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

DirNameLen

As you will see with the entries, Big directories can have filenames with more than 10 characters. To achieve this, there is a marker to let the system know how big each name is. This is the same with the directory name. The actual name follows at the end of the header, terminated by a carriage return (DirNameLen does not include the CR). Note that Big directories no longer have a directory title, as this is now superseded by the longer directory name.

DirSize, DirEntries and DirNameSize

Old and New directories were limited to 47 and 77 entries respectively. With Big directories, this is almost unlimited. Coupled with the no-limit filename length, the directory size is now no longer the same size. This is where DirSize comes in – this gives you the size, in bytes, of the directory. DirEntries will tell you how many entries there are (not the maximum, but the exact number negating the need for DirLastMark), and DirNameSize gives you the size of the name heap (see entries below), in bytes.

Boot Block

I have mentioned the boot block in previous sections and here I will go into greater detail. We have already established that the boot block is held at absolute offset address 0xC00 (sector 12), and that a partial disc record exists at 0x1C0 into this block, but only for multi-zone discs (i.e. F/F+ and new map hard discs). The majority of this boot block is largely beyond the scope of this guide, but is covered here for completeness.

The full specification of the boot block is:

Offset	**Contents**
0x000 upwards	Defect list
0x1BF downwards	Hardware-dependent information
0x1C0 - 0x1FB	Partial Disc record
0x1FC - 0x1FE	Non-ADFS partition descriptor
0x1FF	Check sum byte

Defect List

This is a list of all the defects on the disc. Each word is an absolute offset address of the start of the sector which contains the defect. A marker of 0x200000xx terminates the list, where xx is a checksum of the previous words in the defect list. To calculate this, take each word (not including the 0x200000xx end marker) and XOR them together, rotating right 13 bits, starting with a value of 0x00000000 (this is actually the same calculation as the directory check above). The final byte is this final value with each byte of the word XOR-ed with each other (again, as above).

Hardware-dependent Information

This is an area of information that is, well, hardware dependent. There are no guarantees on how much space is used, if any.

Partial Disc Record

This is enough of the disc record to enable the calculation to find the map and the full disc record. It is not the full disc record, and should be discarded once the full record has been located.

Non-ADFS Partition Descriptor

Three bytes are used to describe any partition that is not an ADFS format. Any such partition will occur at the end of the disc and can be found thus:

Offset	**Contents**	
0x1FC	Format identifier and flags:	
	bits	**Contents**
	0 - 3	partition format identifier (1 => RISC iX)
	4 - 7	flags (reserved - must be zero)
0x1FD	Low byte of start_cylinder	
0x1FE	High byte of start_cylinder	

```
disc_address=start_cylinder * heads * secspertrack * secsize
```


written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

Boot Block Checksum

The final byte is a checksum on the entire boot block and is calculated simply by adding each byte of the boot block, except for this byte, together, and adding any carrys – this is the same process for calculating the checksum on the first two sectors of an old map image.

The Directory Entries

Entries (Old & New)

Offset	Name	Bytes	Meaning
0x00	DirObName	10	Name of object + file attributes, old map - top bit of Object Name:

Char	Meaning
0	R: Object has owner read access
1	W: Object has owner write access
2	L: Object is locked
3	D: Object is a directory
4	E: Object is owner execute-only
5	r: Object has public read access
6	w: Object has public write access
7	e: Object has public execute-only
8	P: Object is private
9	not used

0x0A	DirLoad	4	Load address of object
0x0E	DirExec	4	Exec address of object
0x12	DirLen	4	Length of object
0x16	DirIndDiscAdd	3	Start Sector (old) or Indirect disc address of object (new)
0x19	OldDirObSeq or NewDirAtts	1	

Bit	Meaning when set
0	R: Object has owner read access
1	W: Object has owner write access
2	L: Object is locked
3	D: Object is a directory
4	r: Object has public read access
5	w: Object has public write access
6	Reserved (must be zero)
7	Reserved (must be zero)

Entries (Big)

Offset	Name	Bytes	Purpose
0x00	BigDirLoad	4	Load address of object
0x04	BigDirExec	4	Exec address of object
0x08	BigDirLen	4	Length of object
0x0C	BigDirIndDiscAdd	4	Indirect disc address of object
0x10	BigDirAtts	4	Attributes of object (as New, above)
0x14	BigDirObNameLen	4	Length of object name
0x18	BigDirObNamePtr	4	Offset into name heap for name of this object

Name Heap (Big)

Offset	Bytes	Purpose
0x00	BigDirObNameLen+1	Name of file plus CR
BigDirObNameLen+1	0-3	Padding (0 bytes)

The settings that we are interested in here are the disc address (which will be address*sector_size on S/M/L discs, and using the fragment system on D/E/F/E+/F+ discs) so we know where to find the data, the load & execution addresses (for timestamping and filetyping – see next section) and the length (so we know how much data to retrieve). The filename is also of interest so we know what to call it.

Filenames in Big Directories

Traditionally, ADFS has been restricted to 10 characters for a filename (and 7 for DFS filenames). With the introduction of E+ and F+ this restriction was lifted, as the Big directories implemented what is known as a name heap.

Instead of the filename being in the details of the entry, there are two entries: DirObNameLen, which tells you how long the filename is (minus the CR), and DirObNamePtr, which gives you a pointer into the name heap for where to find the filename. The filename is terminated with a carriage return (CR, or 0x0D), and the name heap follows the directory entries. You can work out the offset of the name heap as we already know how many entries there are, and the size of each entry:

```
name_heap = DirEntries * 0x1C
```

which will be the offset from the end of the header. This should already be known, if you are reading in the entries. It can be calculated thus:

```
entries = (( 0x1C + DirNameLen + 1 + 3) div 4) * 4
```

(this will also align it with a word boundary)

The name_heap offset will therefore be:

```
name_heap = DirEntries * 0x1C + entries
```

Timestamps and Filetypes

Files on Arthur and RISC OS file systems are usually filetyped, similar to how Windows, MacOS and Linux has filetyped files. However, Acorn achieved their filetyping differently to adding on extensions to the file's name. This also allowed for the files to be timestamped. On the BBC and Electron systems, the files where stored with a load address (so the system knows where to load the files) and an execution address (so the system knows where to begin execution). In ADFS these have the format of:

Load address	`0xXXXLLLDD`
Execution address	`0xDDDDDDDD`

If XXX is all set (i.e. 0xFFF) in the load address, then LLL will contain the 12-bit filetype (just enter *SHOW FILE$TYPE_* in a RISC OS command line). D will contain the 40-bit timestamp, which is the number of centi-seconds since 00:00:00 1st January 1900.

Note that the RISC OS Filer will determine the Timestamp of an Application from the !RunImage, if one exists, and not from the Timestamp of the directory. This will differ from the RISC OS command line, when you typed *EX on the same directory.

It is that simple – however, as an example, I will show how to convert from RISC OS 40-bit timestamp to Delphi TDateTime.

RISC OS Date and Times are the number of centiseconds from 00:00:00 1st January 1900, while the Delphi TDateTime is 2 days earlier (00:00:00 30th December 1899). The Delphi TDateTime also is split into a whole part (number of days since) and a fraction (time of day). To convert from RISC OS to Delphi:

First, we will need the number of centi-seconds in one day:

```
DayInCSec = 24*3600*100= 8,640,000 csec
```

We can then find out how many days are in the RISC OS Time Stamp:

```
RISCOSTimeStamp div DayInCSec = RISCOSdays
```

Then we need the whole part of the TDateTime:

```
RISCOSdays+2 = TDateTime whole part
```

The '2' comes from the fact that Delphi starts two days earlier. However, as different versions of Delphi may start at different dates, it may prove useful to get this figure using another method:

```
RISCOSdays + EncodeDateTime(1900,01,01,00,00,00,000) = TDateTime whole part
```

And finally, we work out the fraction part of the day. As we have used 'div' to divide, resulting in an integer, multiplying by the same divisor will give us a different number. The difference between the two will be the centi-seconds elapsed on the day. Dividing by DayInCSec will give us the fraction part:

```
(RISCOSTimeStamp-(RISCOSdays*DayInCSec))/DayInCSec = TDateTime fraction part
```


written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

Free Space Map

I have already mentioned, in passing, the free space maps for both Old and New Map discs. If you wanted to write back to an image, then you would need to understand how the free space map works.

Old Map

The free space map on an old map is fairly simple. Residing from offset 0x000 is a list of 3-byte entries pointing to free sectors on the disc (known as 'FreeStart'). Then residing from offset 0x100 is a list of 3-byte entries informing you how long, in sectors, these free spaces are (known as 'FreeLen'). And, finally, at offset 0x1FE is a single byte indicating how many entries there are.

It is as simple as that…nearly. The indicator (known as 'FreeEnd') is a multiple of three, seeing as each entry is 3 bytes long. There is also a maximum of 82 entries possible.

New Map

Things start to get tricky with the new map. But, we have already seen how the allocation map works, so we should know most of the workings.

Each zone has a 4-byte header. Two of these are checks, while the middle two (offsets 0x01 and 0x02) contain a 16-bit value known as 'FreeLink'. This begins a chain of links pointing into the map areas of free space. Each area's length is marked, as the fragments are, but a length of zero bits terminated by a 1 bit. It becomes a chain because the first two bytes of each link point to the next or are zero if this is the last.

The first FreeLink is different in that it has a terminating bit, so you'll need to clear the topmost bit of these 16 bits. This number is then the number of bits to the first area. If it is zero, then there are none.

So, you pick up FreeLink, clear the top bit, divide by 8 (to get the number of bytes) then add this to the address where FreeLink is. The resultant address will be the next one, which will work in a similar fashion (without the top bit being set). The number of bits from the start of this link to, and including, the next set bit (ignoring the actual link itself) is the length of the free space, multiplied by bpmb (as we have seen earlier).

The disc address can be worked out, as we have already seen thus:

```
disc_address=((FreeLink-map_start)-(zone_spare/8)*zone)*8*bpmb
```

To write data back to the disc, you will need to find an big enough area of free space (or spread across enough fragments), and an ID that is unique, before you start writing. Once written, you will then need to update the catalogue (remembering that ADFS stores the entries in alphabetical order), then update the free space and allocation map. Finally, adjust the initial FreeLink for the zone and ensure that the checksum is valid.

Identifying an ADFS Image

Old Map

This is the only check you can do for a valid ADFS disc:

- Run Free Space Map checksums at 0x0FF and 0x1FF

The following checks are just to make sure you have a valid root directory. ADFS will report a broken directory if these checks fail. If the previous checks fail, it will report a disc fault (i.e. not formatted/understood).

- If you find 'Hugo' at 0x201 and 0x6FB then you will likely have an Old Map, Old Directory
- If you find 'Hugo' or 'Nick' at 0x401 and 0xBFB then you will likely have an Old Map, New Directory

The root may not be where it is expected, due to, for example, bad sectors. Never trust the location of the root for the shape of the disc.

Using this information, you will then know where the root is.

- For Old Directory:
  - Offset 0x200 = offset 0x6FA
  - Offset 0x6CA = zero
  - Offset 0x6FF maybe zero. If not, it will be the directory checksum.
- For New Directory:
  - Offset 0x400 = offset 0xBFA
  - Offset 0xBCA = zero
  - Run directory checksum at 0x400, compare with 0xBFF

Both directories also contain the address of the parent which, in the case of the root, will be the address of where we are looking. Worth using that as a check too.

This check is to find out what shape of disc you have. You can have different sizes (e.g. a 'non-standard' disc shape) and still have a valid image, and also, the disc size reported at these locations is not always the total size of the disc, so do not use to determine the format. This is only really valid for S, M, and L discs, which are effectively the same format and S, M and L just refer to the size of the disc.

- Offset 0x0FC-0x0FE * 0x100 and root at 0x200
  163840 = 'S'
  327680 = 'M'
  655360 = 'L'
- Offset 0x0FC-0x0FE * 0x100 and root at 0x400
  819200 = 'D'

Do not fail a disc structure just because it is not a size you expected - it could be an old map hard drive, or have bad sectors which have been mapped out.

New Map

Again, these are the only checks for a valid disc.

- Offset 0x08, idlen, should be <22
- Offset 0x0D, nzones, should be >0
- Run zone checksum at 0x000

If successful, disc_record will be 0x04. If they fail, try:

- Offset 0xDC4, idlen, should be <22
- Offset 0xDC9, nzones, should be >0
- Run boot block checksum from 0xC00 and compare with 0xDFF

Then assume you have disc_record at 0xDC0, so workout bootmap and root_address from disc record and ensure that both are <offset 0xDD0-0xDD3. You can then run the zone checksums.

I actually wrote a loop that on the first pass disc_record=0x004, and all the zone checks are done. If these fail, on the next pass, disc_record=0xDC0 and the same checks made, including calculating the location of the bootmap. This is because the basic structure of an E and an F are essentially the same. So essentially, your checks will be:

- Offset disc_record+0x04 <22 (idlen)
- Offset disc_record+0x09 >0 (nzones)
- On second pass, run boot block checksum from 0xC00 and compare with 0xDFF
- Workout bootmap and check is less than disc_record+(0x10 to 0x13)+nzones*secsize
- Run zone checksums

You can check to see if you have an extended format (E+/F+) by checking offset 0x30-0x33 is a 1, or if you have discovered the boot record is at 0xDC0, then check offset 0xDEC-0xDEF is a 1.

Further checks can be carried out, as the sector sizes that are currently supported are 256, 512, 1024, 2048 and 4096 bytes per sector (relating to 8, 9, 10, 11 and 12 for log2secsize). In addition, idlen must be a minimum of log2secsize+3.

Robert Sprowson has pointed out that 4096 is the upper limit in the current format, because a 4096-byte sector has 32Kbits in it. A zone worth of map must fit in a sector, and their first free link is encoded as a 16-bit offset: 32K + terminating '1'. While possible, there is little point supporting smaller than 256-byte sectors simply because no discs are made that use that.

Again, the following checks only ensure a non-broken directory, depending on where the map is.

- For New Directories:
  - 'Nick' or 'Hugo' at root_address+1 and root_address+0x7FB
  - Offset root_address = offset root_address+0x7FA
  - Run directory checksum on root_address+0x7FF
- And for Big Directories:
  - 'SBPr' at root_address+4 and 'oven' at root_address+[root_address+0xC]-8
  - Offset root_address = offset root_address+[root_address+0xC]-4
  - Run directory checksum on root_address+[root_address+0xC]-1

And finally, what shape do you have?

disc_record+0x10 (disc size)	disc_record (sector size)	disc_record+0x01 (sectors per track)	disc_record+0x03 (density)	disc_record+0x2C (version flag)	Shape
819200	10	5	2	0	'E' floppy
1638400	10	10	4	0	'F' floppy
819200	10	5	2	1	'E+' floppy
1638400	10	10	4	1	'F+' floppy
3276800	10	20	8	0	‘G’ floppy

Again, do not assume it is not a valid disc because it does not match the above – it could be a hard disc image. There is a flag to check if it is a hard disc - this is at disc_record+0x03, and should be zero. There is also an additional word for big discs at disc_record+0x24, and a flag at disc_record+0x29 (which should be 1 for partitions >512MB).

A Note On Hard Drive Images

All of the ‘.hd4’ hard drive images I have come across have the boot block at offset 0xE00, and not 0xC00. All of the documentation and the Filecore source code points towards the boot block always being at 0xC00 for multi-zone discs. A possible reason for this has been suggested that it is because emulators stick a header at the top to store extra information. The simplest way around this is just to add 0x200 onto every address offset when accessing such data.

A Word of Warning

One more thing to note, and I have noticed this with RISC OS 5.28 – when a disc is formatted, the formatter only writes the information it requires for the disc. Therefore, if you have an ADFS D, for example, and re-format it to ADFS F, or F+, all the information from offsets 0x0000 to 0x0BFF will be left intact. This means that you could identify an ADFS F disc as an ADFS D disc, for example, and it will still function as an ADFS D disc.

Commodore 1541, 1571 and 1581

Introduction

The Commodore 1541 drive was the original floppy disc drive, which you could purchase for the Commodore 64. When Commodore redesigned the 64, to the similar shape to the forth-coming 128, they also redesigned the 1541 drive, which became the 1541-II. There also followed a Commodore 1571 drive. This was, essentially, the 1541 but double sided. The Commodore 1581 was the 3½" floppy drive for the Commodore 64/128 but was released too late and most software was already available on 5¼" discs. This made the 1581 a rare beast. However, all three formats are largely similar, hence why they are all covered together, here. The later Commodore Amiga format is very different and is covered in a later chapter.

Specifications

The original 1541 format had only 35 tracks. This was later expanded on to provide 40 tracks. The 1571, the double sided 1541, had twice this – 70 tracks. However, the number of sectors per track varied as the head went across the disc, so early tracks had more sectors than later tracks. Things were a little easier with the 1581, as this had 80 tracks all with the same number of sectors per track.

Fragmentation

The files, and directories, are stored in fragments, with each fragment being no bigger than the size of a sector (254 bytes). The first two bytes of a fragment (making the sector up to 256 bytes) contained the track and sector number of the next fragment. As the tracks are numbered, starting from 1, a track value of 0x00 means that this is the last fragment of the object. The sector value will then contain the amount of that sector that has been used.

For example, a file chain has the following t/s (track/sector) links:

0x11/0x00, 0x11/0x0A, 0x11/0x14, 0x00/0x32

You will see later that the directory entry will specify that there are 3 fragments of this file.

- The first fragment is at track 0x11 (17), sector 0x00 (0) – this will be in the directory entry.
- The second fragment is at track 0x11 (17), sector 0x0A (10) – this will be the first two bytes of the first fragment.
- The third fragment will be at track 0x11 (17), sector 0x14 (20) – this will be the first two bytes of the second fragment. The first two bytes of the third fragment will tell you that the track is 0x00, meaning this is the last, and the sector is 0x32, meaning the amount of data used, in this sector (not including the t/s link), is 50 bytes.

Each fragment will be a maximum of 254 bytes (sector size is 256 bytes, minus 2 for the t/s link), so the maximum file size will be 3*254 bytes = 762 bytes. However, the final t/s link has 0x32 as the sector value (0x32 = 50). This will mean that the file is 2*254 + 50 = 558 bytes long.

There is also a convention that the 1541 system uses to lay the data down. The first fragment of the file will appear on sector 0 of the track, with the second being 10 sectors later, and so on. The starting track will be chosen so as to be central around the BAM and root directory.

The algorithm for selecting the next sector is as follows:

1. Start with sector 0
2. If the sector is not free, move onto the next sector, and so on until you encounter a free sector in that track.
3. For the next sector, add 10 to this.
4. If this sector goes beyond the number of sectors for the track, then subtract the number of sectors for this track plus one (so for a 21 sector track, you subtract 22).
5. If this is not free, move on as before.
6. When you run out of free sectors on the track, move to the next (or previous track, whichever is away from the root) and use the same sector number as you left with.

This means, that for a 21-sector track (say, for example, track 17 on a 1541 disc) the sequence will be:

17/0, 17/10, 17/20, 17/8, 17/18, 17/6, 17/16, 17/4, 17/14, 17/2, 17/12, 17/1, 17/11, 17/3, 17/13, 17/5, 17/15, 17/7, 17/17, 17/9, 17/19 then onto 16/7, 16/17, 16/5, 16/15, etc.

Converting From Track/Sector to Offset Address

As different tracks have different numbers of sectors, a simple calculation is not enough to work out the offset into an image file, given the track and sector numbers. You could have a lookup table, to give the start offset of each sector, or you could use a rather more complicated, but less memory hungry, method.

To start with, we need to know how many sectors per track there are. This is one lookup table you will require:

Tracks	Side	Number of Sectors
1-17	0	21
18-24	0	19
25-30	0	18
31-35	0	17
36-52	1	21
53-59	1	19
60-65	1	18
66-70	1	17

Note that we are ignoring the 40-track variant of the 1541 for now.

So, you will require these lookup tables (note that this is how the 1541 controller does it):

```
hightrack = (71,66,60,53,36,31,25,18,1)
```

which is where the number of sectors per track changes, and:

```
numsects = (17,18,19,21,17,18,19,21)
```

which is the number of sectors per track, in the area we are looking in. Then:

1. Starting at the far end of the hightrack array, and while the track is bigger than the current element, accumulate the number of sectors, less the difference from the previous.
2. Then, add on the final count for this track, and the number of sectors within the track.
3. And finally multiply the final figure by the sector size (256 bytes)

Or, in pseudo code:

```
sector_count = 0
counter = 7
while track > hightrack[counter]
{       sector_count  +=  (hightrack[counter]  -  hightrack[counter  +  1])  *
numsects[counter]
        counter -= 1 }
sector_count += (track - hightrack[counter + 1]) * numsects[counter]
sector_count += sector
offset = sector_count * 0x100
```

For a 1581, just multiply the track number, less one, by 40 (as there are 40 sectors per track, every track, on a 1581), and then multiply this by sector size, which is still 256 bytes.

For a quick reference, track 18, sector 0 is offset 0x16500 and track 53, sector 0 is offset 0x41000 (which you may need for the following section).

Header and Block Availability Map (BAM) – 1541 and 1571

The header, on 1541 and 1571, contains both the header information and the map. It is commonly known as just the BAM, as opposed to the header, and it is always held on track 18, sector 0.

BAM offset	Bytes	Description
0x00	1	Track location of the first directory
0x01	1	Sector location of the first directory
0x02	1	Disc DOS version byte
0x03	1	Sides (top bit set for double sided)
0x04	35*4	BAM entries for each track, in groups of four bytes per track

0x90	16	Disk Name (padded with 0xA0)
0xA0	2	0xA0A0
0xA2	2	Disc ID
0xA4	1	0xA0
0xA5	2	DOS type
0xA7	4	0xA0A0A0A0
0xAB	50	Unused, should be 0x00
0xDD	35	Free sector count for tracks 36-70

- The Track/Sector location of the first directory will always be track 18, sector 1, despite whatever is entered in these locations.
- The Disc DOS version byte should be either 0x41 or 0x00.
- The DOS type should be '2A'

Track 53, sector 0 will contain the BAM for tracks 36-70, but without the initial byte (free sector count), as this is from offset 0xDD on track 18, sector 0 (above). The rest of track 53 is then not used. The directory entries will then commence at track 18, sector 1.

Header and Block Availability Map (BAM) – 1581

The 1581 format is slightly different to that shown above. The big difference is the location of the BAM has been moved out of the header. Also the header is now on track 40, sector 0, with the BAM at track 40, sector 1 for side 0 & track 40, sector 2 for side 1, and the first directory at track 40, sector 3.

Header

The header will always be found at track 40, sector 0.

Header Offset	Bytes	Description
0x00	1	Track location of the first directory
0x01	1	Sector location of the first directory
0x02	1	Disc DOS version byte
0x03	1	0x00
0x04	16	16 character disc name (padded with 0xA0)
0x14	2	0xA0A0
0x16	2	Disc ID
0x18	1	0xA0
0x19	1	DOS Version
0x1A	1	Disc version
0x1B	2	0xA0A0
0x1D	227	Unused (usually 0x00)

- The track and sector location can be safely ignored, as the first directory is always at track 40, sector 3.
- The Disc DOS version byte should be 0x44.
- The DOS and disc version should be '3D' (i.e. DOS version will be 0x34 and disc version will be 0x44)

Block Availability Map

This will be identical format for both sides. Side 0 BAM will be at track 40, sector 1 and side 1 BAM will be at track 40, sector 2.

BAM Offset	Bytes	Description
0x00	1	Track of next bam sector
0x01	1	Sector of next bam sector
0x02	1	Version number
0x03	1	One's complement of version number
0x04	2	Disc ID bytes
0x06	1	I/O byte:

Bit	Description
7	Verify

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

	6	Check header CRC
	0-5	Unused
0x07	1	Auto-boot-loader flag
0x08	8	Reserved for future (should be 0x00)
0x10	40*6	BAM entries for each track, in groups of six bytes per track

- As before, the track and sector entries can be safely ignored as the BAM locations are fixed.
- Version number will be 0x44 – the same as the header offset 0x02.
- One's compliment of version number will therefore be 0xBB.
- Disc ID bytes will be identical to header disc ID bytes at header offset 0x16.

How the Block Availability Maps Work

The 1541 and 1571 both use four bytes per track for the BAM entries, whereas the 1581 use six bytes per track. However, the layout is essentially identical. I will use Peter Schepers explanation and example for this below:

Lets look at the 1581 track 40 entry at bytes 0xFA-0xFF:

```
Offset    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
000000F0                                  24 F0 FF 2D FF FE
```

The first byte, 0x24, is the number of free sectors on that track. The next five bytes represent the bitmap of which sectors are used or free. Since it is five bytes (8 bits per byte) we have 40 bits of storage. Since this format has 40 sectors per track, the whole five bytes are used.

The 1541/1571 format is shorter, so we will look at the first entry at bytes 0x04-0x07:

```
Offset    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000              12 FF F9 17
```

As the 1581, the first byte, 0x12, is the number of free sectors on that track. The next three bytes represent the bitmap of which sectors are used or free. The 1541/1571 tracks have variable number of sectors per track, and is always less than 24 sectors per track, we have some wastage.

These last bytes of any BAM entry must be viewed in binary to make any sense. Using 1581 track 40 as our reference:

0xF0=11110000, 0xFF=11111111, 0x2D=00101101, 0xFF=11111111, 0xFE=11111110

In order to make any sense from the binary notation, flip the bits around.

```
                  111111 11112222 22222233 33333333
Sector  01234567 89012345 67890123 45678901 23456789
        ------------------------ -------- --------
        00001111 11111111 10110100 11111111 01111111
```

Note that if a bit is set, the sector is free. Therefore, track 40 has sectors 0-3, 17, 20, 22, 23 and 32 used, all the rest are free. The 1541/1571 is the same, just half the length:

0xFF=11111111, 0xF9=11111001, 0x17=00010111

Again flip the bits around.

```
                  111111 11112222
Sector  01234567 89012345 67890123
        ------------------------
        11111111 10011111 11101000
```

Since we are on the first track, we have 21 sectors, and only use up to the bit 20 position. Using what is shown above, for 1581, track 1 has sectors 9,10 and 19 used, all the rest are free. Any leftover bits that refer to sectors that don't exist, like bits 21-23 in the above example, are set to allocated.

Remember that for tracks 36-70 on the 1571, the BAM entries are at track 53, sector 0 and do not contain the initial free sector count byte, which is held in the header on track 18, sector 0.

Directory Entries

The first directory (and maybe the only directory) will always be found at track 18, sector 1 on the 1541/1571 and track 40, sector 3 on the 1581.

Offset	Bytes	Description
0x00	32	First directory entry
0x20	32	Second directory entry
etc.		

And each entry:

Entry Offset	Bytes	Description
0x00	1	Track location of next directory sector
0x01	1	Sector location of next directory sector
0x02	1	File type/attributes

Bit	Usage
0-3	The actual filetype
	000 (0x00): DEL - deleted
	001 (0x01): SEQ - sequence
	010 (0x02): PRG - program
	011 (0x03): USR – user file
	100 (0x04): REL - relative
	101 (0x05): CBM – partition/sub-directory (1581 only)
4	Not used
5	Used only during SAVE-@ replacement
6	Locked flag
7	Closed flag (Not set produces "*", or "splat" files)

0x03	1	Track location of first sector of file
0x04	1	Sector location of first sector of file
0x05	16	16 character filename
0x15	1	Track location of first side-sector block (REL file only)
0x16	1	Sector location of first side-sector block (REL file only)
0x17	1	REL file record length (REL file only, max. value 254)
0x18	6	Unused
0x1E	2	Approximate file size in sectors

Generally, directories only take up a single track, with 8 file details per sector. Using the t/s link and fragmentation method (offset 0x00 and 0x01 on the first entry, and hence, the first two bytes of the sector), and filling the track, this makes it possible to have 144 files per disc, on 1541 and 1571, as there are 19 sectors on track 18, leaving 18 sectors for file entries: 18*8=144. For a 1581, there are 40 sectors on track 40, minus the three for the header and BAMs, making it possible to have 37*8=296 files. There are some discs/images where directories span multiple tracks.

The t/s link should contain track 0x00 for the second and subsequent files in the sector, but contain the track and sector link to the next directory sector for the first file in the sector.

The file size is given in sectors making this only an approximation of the actual file size (to within 254 bytes). See earlier in the chapter (under Fragmentation) on how to work out the actual file size. As this file size is given in sectors, this can be used to work out how many fragments the file takes up, as each fragment is a maximum of 254 bytes (i.e. a sector minus the 2-byte t/s link).

Identifying a 1541/1571/1581 Image

These formats do not have a signature that we can positively identify as being one of these images. Also, the cyclic redundancy checks, and checksums, are only on a physical disc and therefore unavailable within an image. Therefore, we have to make some assumptions, given the information above.

1541/1571

BAM is at track 18 sector 0:

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

- BAM offset 0x02 should be 0x41 (“A”) or 0x00
- BAM offset 0xA0, 0xA1, 0xA4, and 0xA7-0xAA should be 0xA0
- BAM offset 0xA5 should be 0x32 and 0xA6 should be 0x41 ("2A")
- BAM offset 0x03 will be 0x00 for 1541 (single sided) and 0x80 for 1571 (double sided)

1581

The header is at track 40 sector 0:

- header offset 0x02 should be 0x44 (“D”)
- header offset 0x03 should be 0x00
- header offset 0x14, 0x15, 0x18, 0x1B, 0x1C should be 0xA0
- header offset 0x19 should 0x33 and 0x1A should be 0x44 ("3D")

BAM, side 0, is at track 40 sector 1:

- BAM offset 0x00, 0x01 should be 0x28 & 0x02
- BAM offset 0x02 should be 0x44 (“D”)
- BAM offset 0x04 & 0x05 should be the same is header offset 0x16 & 0x17

BAM, side 1, is at track 40 sector 2 - as above, except:

- BAM offset 0x00, 0x01 should be 0x00 & 0xFF

Commodore AmigaDOS

Introduction

The Commodore Amiga was the successor computer to the Commodore 128 and, as such, needed a successor filing system. Like the BBC Micro evolving into the Archimedes, gone were the tapes, to be replaced solely by a disc system. Also, like the BBC Micro, which already had a disc filing system, the Commodore 64 and 128 also had their disc filing system. Unlike the BBC, Commodore re-designed the filing system for the Amiga (where Acorn just evolved ADFS onto the newer platform). One big, and very important, change, even though it may seem minor, is that the Amiga was based on the 68000 processor that, as a 16-bit processor, used the big endian numbering system. The 64 and 128 before it, both with the 8-bit 6510 processor, used the little endian numbering system.

Big Endian vs Little Endian

If you did not already know, the endian system is a way of storing bytes. Little endian has the LSB (least significant byte) first, followed by the MSB (most significant byte). Big endian is the other way round – MSB followed by LSB. This means two bytes stored little endian would be 0x12 0xFF, which as a 16-bit number, would be 0xFF12. As big endian, it would be 0x12FF.

Disc Structures

There are two standard types of floppy discs that AmigaDOS recognises:

Type	Sector Size	Sectors/Track	Heads	Tracks	Capacity
Double Density	512	11	2	80	900KB
High Density	512	22	2	80	1.8MB

You can actually derive the capacity from the other information:

```
Total number of sectors = sectors/track * heads * tracks
Total disc size = sectors * secsize
```

Basic Layout of the Disc

Every different block of data stored on an AmigaDOS disc has a header. This header is very largely the same (there is a comparison/summary at the end of this chapter of all the different headers mentioned within the chapter), only differing in a few entries. Not all entries are used in all headers, but similar data is stored at the same offsets into each header (for example, offset 0x14 in the headers is the checksum of the header).

Each block is the same size. On floppies, this is 512 bytes (0x200), which, as luck would have it, the same size as the sectors. The only blocks that are different to all the others are the boot block and the bitmap block. The boot block is found right at the beginning of the disc, and this has a different (but almost similar) header, and the block is 1024 bytes (0x400) in size.

Bootblock

The first thing you'll find in an image is the bootblock. This appears at offset 0x0000 (sectors 0 and 1) in an image floppy, and the header is:

Offset	Size	Description
0x000	4	Disc Identifier + flags
0x004	4	Block checksum
0x008	4	0x370 (is actually known as rootblock, but is the same for both DD and HD)
0x00C	1012	Bootblock code - this is actual 68000 machine code.

However, the bootblock is actually optional, as long as the Disc Identifier is there.

Disc Identifier

The most common identifier used in AmigaDOS is 'DOS'. But you can also have 'PFS' (Professional Filing System). If 'KICK' is the first four characters, then it is a Kickstarter disc and there will be no

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

bootblock, as the next 1020 bytes will be zeros. A super Kickstarter will have KICKSUP at the start, and will not have any bootblock or rootblock.

Flags

bit	Set	Clear	
0	FFS	OFS	Filing system - Fast filing system or original filing system
1	INTL	none	International characters mode
2	DIRC	none	Directory Cache Mode

Checksums

The calculation for the block checksum, like the headers themselves, are very similar. The algorithm is fairly straightforward: Add all the words, with carry, of the entire bootblock, except for the checksum. The difference then comes between the bootblock checksum and the other blocks' checksums. Once you have your number, invert every bit (i.e. checksum = NOT checksum) for the bootblock, or negate the number (i.e. checksum = -checksum) for all other blocks.

Block Primary and Secondary Types

The primary and secondary types can uniquely identify blocks. These are two words found at the beginning (offset 0x000) and end (offset 0x1FC) of the block. They are:

Primary	Secondary	Block
2	1	Root block
2	3	File block
8	n/a	Data block
16	-3	File extension block
2	2	Directory block

Date and Time stamps

Where a date/time stamp is used in a block, it will be found as three words: days, minutes and ticks. The days is the number of complete days elapsed since 1st January 1978; the minutes is the complete minutes elapsed since midnight of the current day; and the ticks are the number of complete ticks since the last minute. A tick is 1/50th of a second.

Hash Tables

AmigaDOS uses a system of hash tables to locate the file's header. To calculate the file block address from, given the filename:

```
l=length(name)
hash=length(name)
for i=0 to l-1
 hash=hash*13
 hash=hash+asc(uppercase(name[i]))
 hash=hash AND 0x07FF
next
hash=hash MOD 72 //i.e. contents of rootblock offset 0x000C : size of hash table
```

This number is then the offset into the hash table for that file. The hash table is, essentially, a table of sectors, so to find the offset multiply this number by secsize. Some files will have the same hash value, so to get around this the file header will contain a link chain to other files with the same hash value.

But if you just want to list the catalogue of a directory, just go through the hash table and follow each link to each successive file/directory header. But more on this later.

Bitmap Block

The bitmap block is pointed to by a value in the rootblock (see below), or will be in the block following the rootblock (i.e. 881 for DD and 1761 for HD), and is a map of the free and allocated space. One bit

represents one sector, so if it is set the sector is free, and if clear it is allocated. The first bit (word 0, bit 0) refers to sector 2 (as the bootblock is not included), and continues to the 1760th sector (for double-density). If there are not enough bits free, the bitmap extension block is then used.

The block header looks like:

Offset	Size	Description
0x000	4	Checksum
0x004	508	Map

25 bitmap blocks is sufficient for up to a 50MB disc. Beyond that, there is the bitmap extension block where the layout of which is the same as the bitmap block pointers, with a 0x00000000 to terminate the list.

The Blocks

Ignoring the bootblock and bitmap block, the other blocks in the file system, as has already been mentioned, are all largely similar and are summarised at the end of the chapter. Following is a more detailed look at each of these blocks. Where entries are marked as unused, they will contain zeros.

Rootblock

The rootblock holds the root directory, and the header is very similar to the directory header (later in the chapter). The rootblock will be at the exact centre of the disc that, for floppies, is at sector 880 (offset 0x6E000) on double-density discs or 1760 (offset 0xDC000) for high-density discs (despite what the entry in the bootblock might tell you). There are no other indicators as to where the rootblock is, so you'll just need to go looking for it. The header is as follows:

Offset	Size	Description
0x000	4	Block primary type
0x004	8	Unused
0x00C	4	Hash table size (= secsize/4 - 56 = 0x48) in words
0x010	4	Unused
0x014	4	Checksum
0x018	288	Hash table (size varies, but will be as dictated by 0x00C)
0x138	4	Bitmap flag : -1 (0xFFFFFFFF) is VALID
0x13C	100	Bitmap block pointers
0x1A0	4	First bitmap extension block (hard discs only)
0x1A4	4	Last root alteration date : days
0x1A8	4	Last root alteration date : minutes
0x1AC	4	Last root alteration date : ticks
0x1B0	1	Length of volume name
0x1B1	30	Volume Name
0x1CF	9	Unused
0x1D8	4	Last disc alteration date : days
0x1DC	4	Last disc alteration date : minutes
0x1E0	4	Last disc alteration date : ticks
0x1E4	4	Filesystem creation date : days
0x1E8	4	Filesystem creation date : minutes
0x1EC	4	Filesystem creation date : ticks
0x1F0	8	Unused
0x1F8	4	FFS first directory cache block, otherwise 0x00000000
0x1FC	4	Block secondary type

The root alteration date is when the last change of the root directory occurred; the disc alteration date is when the last change to the disc occurred; and the filesystem creation date is when the disc was originally initialised (formatted).

Directory block

Very similar to the root block, seeing as the root is a directory itself, and also similar to a file block, as each directory is itself a directory entry, there are only a few minor differences. However, it works exactly the same way as the root block above, and the file block below.

Offset	Size	Description
0x000	4	Primary type
0x004	4	Self pointer
0x008	12	Unused
0x014	4	Checksum
0x018	288	Hash table
0x138	2	Unused
0x13A	2	User ID
0x13C	4	Group ID
0x140	4	Attributes (see File Block, below)
0x144	4	Unused
0x148	1	Comment length
0x149	79	Comment
0x197	12	Unused
0x1A4	4	Last access date: days
0x1A8	4	Last access date: minutes
0x1AC	4	Last access date: ticks
0x1B0	1	Directory name length
0x1B1	30	Directory name
0x1BF	9	Unused
0x1C8	4	FFS : Hardlink chained list
0x1CC	20	Unused
0x1F0	4	Hash chain (next entry with same hash value – see File Block, below)
0x1F4	4	Parent directory
0x1F8	4	FFS : first directory cache block
0x1FC	4	Secondary type

File Block

Each file will have a header with a pointer to the data block chained list, and to other files/directories with the same hash value, in the same directory, as this one.

Offset	Size	Description
0x000	4	Block primary type
0x004	4	Self pointer (to this sector)
0x008	4	Number of data blocks stored here
0x00C	4	Unused
0x010	4	Pointer to first data block
0x014	4	Checksum
0x018	288	Table of data block pointers
0x138	4	Unused
0x13C	2	User ID
0x13E	2	Group ID
0x140	4	Attributes
0x144	4	File size in bytes
0x148	1	Comment length
0x149	79	Comment
0x197	12	Unused
0x1A4	4	Last change date : days
0x1A8	4	Last change date : minutes
0x1AC	4	Last change date : ticks
0x1B0	1	Filename length
0x1B1	30	Filename
0x1BF	9	Unused
0x1C4	4	FFS : Unused
0x1C8	4	FFS : Hardlinks chained list - first is the newest
0x1CC	20	Unused
0x1F0	4	Hash chain (next entry with same hash value)

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk 9 January 2021

0x1F4	4	Parent directory
0x1F8	4	Pointer to first file extension block
0x1FC	4	Secondary type

You have two choices here on how to find the data – the first is similar to the Commodore 64 system, which has a chained list from one data block to the next. This first data block can be found at offset 0x0010. You can then either count the number of blocks as you go (as given by offset 0x0008), or just continue until you find the terminator word in the data block. This will only work on the Original File System, as the Fast File System's data blocks do not have headers.

Your second method to find each data block is to follow the table of data block pointers, starting at the end and working backwards. Again, the counter at offset 0x0008 will indicate how many you need to find, or you just continue until you hit a zero, or the beginning of the table.

File Attributes

If the bit is set, the file will have the following attributes:

Bit	Description
0	Cannot Delete (D)
1	Cannot execute (E)
2	Cannot Write (W)
3	Cannot Read (R)
4	Archived (A)
5	Pure (P)
6	Script (S)
7	Hold
8	Group delete protected (d)
9	Group execute protected (e)
10	Group write protected (w)
11	Group read protected (r)
12	Other delete protected
13	Other execute protected
14	Other write protected
15	Other read protected
16 - 30	reserved
31	SUID, multiuserFS only

Hash Chain

The value at offset 0x1F0 is the chain of links to the other filenames with the same hash value in the parent directory (which there is a pointer to at the following word, 0x1F4).

File Extension Blocks

The file extension blocks are used where the file size exceeds what can be stored in 72 blocks (which is around 72*secsize = 36KB). Each extension block will hold details of where to find each 36K block in the file.

OFS Data Block

This is only valid for the Original File System as the Fast File System does not have headers in the data blocks.

Offset	Size	Description
0x000	4	Primary type
0x004	4	Pointer to file header block
0x008	4	Data block number (starts at 1)
0x00C	4	Size of data in this block
0x010	4	Link to next data block (0 if last)
0x014	4	Checksum
0x018	varies	Data – size is indicated by offset 0x00C

File Extension Block

This is used for each ~36KB of data of a file.

Offset	Size	Description
0x000	4	Primary type
0x004	4	Self pointer
0x008	4	Number of data blocks stored
0x00C	8	Unused
0x014	4	Checksum
0x018	288	Data block pointers
0x138	50	Unused
0x1F4	4	Pointer to file header block
0x1F8	4	Next file header extension block
0x1FC	4	Secondary Type

The data block pointer list works as file data block above. Offset 0x1F8 will contain a link to the next file extension block (or 0 if it is the last).

Identifying an AmigaDOS Image

Identifying an AmigaDOS image is fairly straightforward, given that each block starts at a sector boundary, and each one has a checksum. But you will first need to determine if it has a bootblock. Whether or not this is the case, the first three characters should be 'DOS', or 'PFS'; or the first four should be 'KICK'. If the image does contain a boot block, you should find 0x370 at offset 0x0008, and then check the checksum at 0x0004 for the block.

You can then set about finding the root block. This block will have 2 as the first word of a sector offset (multiple of secsize), and 1 as the final word of the sector. You can then do the check on the checksum at offset 0x014 into the sector offset. You can also use this method to determine whether you have a double-density or a high-density disc image (by the location of the root block). It will also indicate not only if it is a hard disc image, but also the capacity of the disc. It has been noted that the root block will occur at the exact centre of the disc, so the capacity will be twice what the location of the root block is (i.e. double-density disc has a root block at 0x6E000, so the capacity is 2*0x6E000=0xDC000=880KB).

AmigaDOS Block Layout Comparison

	root		file		data*		file extension		directory	
Offset	**Size**	**Description**	**Size**	**Description**	**Size**	**Description**	**Size**	**Description**	**Size**	**Description**
0x000	4	Block primary type (=0x00000002)	4	Block primary type (=0x00000002)	4	Primary type (0x00000008)	4	Primary type (0x00000010)	4	Primary type (0x00000002)
0x004	*8*	*Not Used (all 0x00)*	4	Self pointer (to this sector)	4	Pointer to file header block	4	Self pointer (to this sector)	4	Self pointer (to this sector)
0x008			4	Number of data blocks stored here	4	Data block number	4	Number of data blocks stored	*12*	*Not Used (all 0x00)*
0x00C	4	Hash table size (0x00000048)	*4*	*Not Used (all 0x00)*	4	Size of data in this block	*8*	*Not Used (all 0x00)*		
0x010	*4*	*Not Used (all 0x00)*	4	First data block pointer	4	Next data block (0 if last)				
0x014	4	Checksum								
0x018	288	Hash table	288	Data block pointers			288	Data block pointers	288	Hash table
0x138	4	Bitmap flag : -1 is VALID	*4*	*Not Used (all 0x00)*					*4*	*Not Used (all 0x00)*
0x13C			2	User ID					2	User ID
0x13E			2	Group ID					2	Group ID
0x140			4	Attributes					4	Attributes
0x144	100	Bitmap block pointers	4	File size in bytes					*4*	*Not Used (all 0x00)*
0x148			1	Comment length					1	Comment length
0x149			79	Comment					79	Comment
0x198			1	0x00					1	0x00
0x199			*11*	*Not Used (all 0x00)*					*11*	*Not Used (all 0x00)*
0x1A0	4	First bitmap extension block (hard discs only)								
0x1A4	4	Last root alteration date (days)	4	Last change date (days)					4	Last access date (days)
0x1A8	4	Last root alteration date (minutes)	4	Last change date (minutes)					4	Last access date (minutes)
0x1AC	4	Last root alteration date (1/50sec)	4	Last change date (1/50sec)					4	Last access date (1/50sec)
0x1B0	1	Length of volume name	1	Filename length			*188*	*Not Used (all 0x00)*	1	Directory name length
0x1B1	30	Volume Name	30	Filename	488	Data			30	Directory name
0x1CF	1	0x00	1	0x00					1	0x00
0x1D0	*8*	*Not Used (all 0x00)*	*4*	*Not Used (all 0x00)*					*8*	*Not Used (all 0x00)*
0x1D4			4	FFS : Unused (0x00000000)						
0x1D8	4	Last disc alteration date (days)	4	FFS : Hardlinks chained list					4	FFS : Hardlink chained list
0x1DC	4	Last disc alteration date (minutes)	*8*	*Not Used (all 0x00)*					*20*	*Not Used (all 0x00)*
0x1E0	4	Last disc alteration date (1/50sec)								
0x1E4	4	Filesystem creation date (days)								
0x1E8	4	Filesystem creation date (minutes)								
0x1EC	4	Filesystem creation date (1/50sec)								
0x1F0	*8*	*Not Used (all 0x00)*	4	Next entry with same hash value					4	Next entry with same hash value
0x1F4			4	Parent directory			4	Pointer to file header block	4	Parent directory
0x1F8	4	FFS first directory cache block	4	Pointer to first file extension block			4	Next file header extension block	4	FFS first directory cache block
0x1FC	4	Secondary type (0x00000001)	4	Secondary type (0xFFFFFFFD, or -3)			4	Secondary Type (0xFFFFFFFD or -3)	4	Secondary type (0x00000002)

** Data blocks are only valid for the Original File System. Fast File System data blocks do not have a header.*

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk

Inf Files

Most applications dealing with disc images, including emulators, will utilise a file known as an 'inf' file. These, unsurprisingly, have an extension of '.inf'. They are named the same as the file to which they belong, and which is to be imported into (or has been exported from) one of these applications. The idea is that files residing on a FAT32/NTFS/etc. file system will lose the information that the BBC MOS, and RISC OS, requires.

There has not been any hard and fast format regarding these, so a discussion was held on the Stardot forums to bash out an agreed format, which now follows. Please also note that this also applies to directories, as well as files.

This currently does not extend to non-Acorn formats, as yet.

*.inf format

So, the format agreed is this - a single line, in a text file, containing:

`<filename> <load> <exec> <length> <access> <extra info>`

Where:

Each field separated by at least one space, but could be more.

`<filename>` is the original BBC filename. Quotes are optional, but mandatory if the filename contains spaces. This could be different to the way the file is named on the host system (and hence the inf file).

`<load>` is the file's load address in hex.

`<exec>` is the file's execution address in hex.

`<length>` is the file's length in hex.

`<access>` can be either the access letters (L for DFS, LWRElwre for ADFS), or hex number according to the OSFILE API:

Bit	Meaning
0	'R': Readable by you
1	'W': Writable by you
2	'E': Executable by you
3	'L': Not deletable by you (locked on DFS)
4	'r': Readable by others (NFS, not 8-bit ADFS)
5	'w': Writable by others (NFS, not 8-bit ADFS)
6	'e': Executable by others (NFS, not 8-bit ADFS)
7	'l': Not deletable by others (NFS, not 8-bit ADFS)

For DFS, this will be 0x08 for locked, or 0x00 for not locked.
`<extra info>` is tag value pairs, using quotes where applicable (i.e., contains spaces) for any extra information.

The filename for the host filing system (e.g., Windows) should be valid for that system, with the .inf file matching.

BBC	<->	DOS/Windows/macOS
#	<->	?
.	<->	/
$	<->	<
^	<->	>
&	<->	+
@	<->	=
%	<->	;

Applicable to both files and directories.

written by Gerald Holdsworth gerald@hollypops.co.uk www.geraldholdsworth.co.uk