Amiga Disk Encoding Schemes

MFM? GCR? Please explain!

by Betty Clay

Betty Clay, a former mathematics teacher, chose writing for her second career. She is the Amiga columnist on STARTEXT (a videotext service of the Ft. Worth Star-Telegram). She has a regular column with ICPUG, and her work has also appeared in The Amigan. Betty can be reached via CompuServe (PPN:76702,337), QuantumLink (bjc) or by mail at 1605 Glasgow Drive, Arlington, TX. 76015.

Disk drives are fascinating things. When they work properly, they manage data and software so beautifully that there is little need to think about the way in which they do it. When something goes wrong though, the loss is (or can be) so great that the inner workings of the drive become of utmost importance.

When I first began to ask about how disk drives work, I was usually told that drives were intended to be black boxes and that I was not supposed to understand them - just use them with gratitude. (Was someone as ignorant as I, or was he being a bit sexist?)

Eventually, a disk editor entered my life, and I learned to read and interpret the hexadecimal representations of what was on my disks. Poring over the disk manuals provided some information, and I learned to read directory tracks, find where things were actually located on my disk, and sometimes rescue things that would otherwise have been lost. It was through the writings and teaching of Mike Todd in England, and of Dick Immers and Gerry Neufeld in North America, that I learned about disk encoding schemes, headers, header gaps and all the other niceties that make up the *real* layout of the disk. It's not that I ever actually saw a disk block encoded in CCR, but I did know what was written there, how it was written, and how it was read back.

One of my first projects was to work out the disk layout for the then new Amiga. It was a worthy task, because corrupted disks were very common in those days, and we did not have DiskSalv to rescue them.

Among the very few utilities available in those earliest days was an editor called DiskEd. Perhaps I was the only person who really liked DiskEd, but it enabled me to learn much of the disk structure early on. There are ways in which I find it superior to any of the later, more popular, editors. With DiskEd, and the sparse information in the ***ROM Kernal Manual***, I was able to learn about the header information within each block and the hash tables used in directory blocks, and to follow a file from the root to its actual location on the disk.

But for me, this was still not enough. I was reading a representation of what was on the disk, but I still did not know what was really there.

The Rom Kernal Manual told us from the first that the Amiga drives were capable of writing in either GCR or MFM format. It did not give a further description of either. GCR was familiar because it is used in the other non-Amiga Commodore drives,

MFM, while an old method of writing on disks, was new to me. I went about seeking information on this method, but found little. Everywhere I turn, people say that "this is an MFM drive" or "this is a GCR drive" or "this is a(n) xxxx drive", but no one could ever explain exactly what these designations meant. Days of searching through libraries have been fruitless. There seem to be people who *do* know it, and they use the terms freely, but I could find very few references in print. The information that I have discovered has been in tiny bits and pieces. Put together, those tidbits seem to be sensible and correct. Here is what I have learned.

Why encode?

Within the computer, information is stored as on-bits and off-bits (represented most often by zeros and ones), but it decodes them so that we see alphanumeric characters. Similarly, the data on the disk can be written as a string of zeros and ones -or more accurately, as a string of magnetized bits of oxide.

The read/write head on a disk drive is very much like a coil. As current passes through this coil, it creates a magnetic field, which is recorded onto the magnetic disk. If the current goes through in one direction, the bit is recorded with a 'north pole'; if another way, with a 'south pole'. If nothing at all is recorded, the disk will contain a null at that spot. Nulls occur only in the gaps between data; the Amiga places a group of nulls at the end of the series of logical sectors on a track. Some kinds of drives place nulls between individual sectors, which allows the reading of a single sector from a track, rather than in whole tracks as done on the Amiga.

When the read-head is passing over a track, it ignores the nulls in the gap. If it passes over a magnetic field, a pulse is generated, and is interpreted as data. Since there are no nulls within sectors, the presence of a pulse can be interpreted as an on- or one-bit; the absence of a pulse as an off- or zero-bit.

The disk spins in the drive at a constant and carefully controlled speed of 300 rpm. Amiga drives are constructed to always be within 1.5% of this speed. This allows about .2 seconds for the reading of a whole track of data, including eleven sectors and their headers. The bits must be written to the disk such that the drive can accurately determine whether each bit is a zero or a one. Physical restraints of the drive require that there be at least two microseconds between polarity changes, or 1-bits.

In order to make reading more accurate, encoding schemes have been developed that make it impossible for two 1-bits to occur too close together for accurate reading. They also establish rules for writing data, making it possible to write some bits that could not possibly *be* data. These schemes serve two purposes. The drive must be able to determine where the data starts if it is to interpret the binary digits accurately, and it also must be able to accurately determine whether any given bit is zero or one.

The start of data is marked on the disk with a synchronization mark - a pattern of bits that could not possibly occur in normal data. On the 8-bit Commodore drives, the sync mark consists of ten or more 1-bits in a row, since normal data is not allowed to have more than eight consecutive ones. On the Amiga, a data bit can never have more than four zeros, and a sync mark. Bits of data are encoded in ways that prevent too many on-bits or too many off-bits in a row, so that the drive can accurately read them. While there are many methods of encoding the data, the two mentioned here are the two found in the Amiga, GCR and MFM.

GCR encoding

There are several forms of GCR encoding, but the one described here is the one used in the 8-bit Commodore drives. The Amiga is capable of using GCR, but no one seems ever to use it. There is one apparent advantage to GCR - more data can be stored on a disk encoded with GCR than with the MFM we normally use.

GCR stands for **Group Code Recording**, perhaps because the bits are grouped together before encoding. In order to use this system, each byte is separated into two four-bit nybbles. Each nybble is then encoded according to a GCR table, with each four-bit nybble being encoded as five GCR bits. Using GCR, it can be determined that there will never be more than eight consecutive one-bits or two consecutive zero-bits. The laws of probability tell us that for *n* distinct members of a set, there can be two to the *n* power possible arrangements of them.
Thus, for groups of four bits, there can be exactly sixteen ways in which they might be encoded. The GCR table, therefore, consists of sixteen translations:

Decimal	Hexadecimal	Binary	OCR equivalent
0	$0	0000	01010
1	$1	0001	01011
2	$2	0010	10010
3	$3	0011	10011
4	$4	0100	01110
5	$5	0101	01111
6	$6	0110	10110
7	$7	0111	10111
8	$8	1000	01001
9	$9	1001	11001
10	$A	1010	11010
11	$B	1011	11011
12	$C	1100	01101
13	$D	1101	11101
14	$E	1110	11110
15	$F	1111	10101

When a byte of data is to be encoded, the process goes some-thing like this:

Take a byte of data, say the number 18, which is $12.

Binary notation:	0001 0010
Convert the high nybble to OCR:	01011
Convert the low nybble to OCR:	10010
And the concatenated byte is now:	0101110010.

Note that there are never more than two zeros in a row. No item in the lookup table consists of all ones, and an examination of the table shows that there can never be more than eight consecutive ones.

Since GCR bytes contain 10 bits, an 8-bit computer byte always leaves some bits left over after filling the allotted space. For this reason, it is customary for a OCR drive to handle four 10-bit bytes at a time. This gives a total of 40 OCR bits, which can be handled in the same manner as five regular bytes. The number is chosen because 40 is evenly divisible for translation of 8-bit bytes, being the least common multiple of eight and five.

Since four binary bits are encoded as five OCR bits, there is 80% efficiency. That is, 80% of the available bits can be used either for your own data or for overhead information needed by the drive.

This would appear to let a disk hold far more information than could be stored under most other methods. However, since GCR permits the use of as many as eight on-bits in a row, the drive cannot interpret them at full speed. It is necessary to write or read at only half the normal speed, in order to insure accuracy. When the writing speed is slowed to four rnicroseconds per bit instead of the normal two, the density of the data is only half as much, cutting drastically into the storage advantage. On older drives for both the Apple and Commodore machines, increased capacity was more important than speed, so GCR was used by both. The GCR in the Amiga is actually compatible with the Apple version. The lookup tables are the same for both, but the actual implementation is a bit different.

I once asked Jay Miner why the GCR table and capability were built into the Amiga but never used. He said that they wanted it for compatibility reasons, but preferred the faster and easier MFM encoding for normal use. Should you ever need it, it is sitting there waiting for you, and it does not take up very much room.

MFM encoding

This is the encoding method currently in use on the Amiga floppies - and on most of the hard drives as well. This format puts both data bits and clock bits onto the disk. Since there is one clock bit recorded for each data bit, only 50% of the bits on the disk represent information that is needed. The clock bits are added as the data is written out, and are deleted as it comes back in.

The system does not require a lookup table, because the encoding rules are very simple. If either of the two adjacent data bits is a 1-bit, a zero is inserted between them. If two adjacent data bits are both zeros, a one is inserted between them. With this system, it is certain that two 1-bits will never be adjacent to each other. The system can thus read and write at two microseconds per bit as opposed to four microseconds for OCR.

Let us take that number 18 again:

```
Binary notation:        1     0     0     1     0
Get the clock bits:  0     0     1     0     0

So, encoded, it is:  0  1  0  0  1  0  0  1  0  0
```

Timing also determines the interpretation of these pulse/no-pulse bits. The computer knows it is supposed to be passing a new bit every two microseconds, so it interprets the data as having one zero each two microseconds if it does not determine a change of magnetism in that time.

When data is written to the disk, an entire track is written at one time, but it is managed as though it were done in sectors. Each Sector has a header of sixteen bytes, but this header is not seen by normal users of the drive. According to the *ROM Kernal Manual*, the Amiga sector headers, before encoding, contain:

- two bytes of $00 data, represented in MFM by $AAAA
- two bytes of $Al, a standard MFM sync byte, which is $A1 without a clock bit
- one byte to tell what format (for * 1.0, this format type was $FF)
- one byte to hold the track number
- one byte to hold the sector number
- one byte to hold the number of sectors to the end of the write
- sixteen bytes of operating system recovery information

- four bytes of header checksum
- four bytes of data checksum and this is followed by the 512 bytes in the sector that we are able to read with a disk editor.

The encoding of these bytes is done in a curious way. The first four bytes of the header are encoded one byte at a time. First, the odd bits are separated from the even bits. The odd bits are then encoded, following the rules for MFM, and are written out to the disk. Next the even bits are shifted left one position, and then they are encoded and written out.

Using our number 18, should it be in one of those bytes, it would be handled thus:

```
18  = $12           hexadecimal
    = 1 0 0 1 0     binary
      1   0   0     for the odd bits
    0   0   1       for the odd bits clock-bits
    0 1 0 0 1 0     written out to the disk
        0   l       for the even bits
    0   l           shifted left one position
  1   0             for the clock bits       I
  1 0 0 1           written out to the disk
```

So that the entire representation of the number 18 on the disk would be 0100101001.

This would be done for each of the four bytes of header and sync marks. The next four bytes (format, track, sector, and number of sectors to end) are all encoded in a single block, following the same pattern: separate the odd bits from the even, encode the odd bits and write them out, encode the even bits and write them out. The 16 bytes of operating system recovery information are then encoded as a single block, in the same manner. Then the four bytes of header checksum are handled as a single block, followed by the four bytes of data checksum as another unit. After this, the entire data sector of 512 bytes is encoded as a single unit: first the odd bits, then the even ones.

All encoding is done in the blitter. The data is read from the disk, decoded in the blitter, and placed in a user buffer. What you see in your buffer will always be decoded. It is probably possible to bypass the blitter and read the actual encoded information, but I do not know of anyone who has done so.

Blocks to end of write

Perhaps this is not really a part of the encoding process, but the ‘blocks to end of write' is a curious feature of AmigaDos.

When you write to an Amiga floppy, the drive heads are given a track number to which they can write. They then move immediately to the assigned track and begin writing. They do not wait for a sync mark, nor do they look for a particular sector, and they are not likely to write on the same part of the track that was used before.

Remember that sectors on the Amiga are logical, not physical. There are no null gaps between sectors, though there is a rather large gap of nulls at the end of the last sector on the track. The heads just start writing at the spot on which they have found the track. If this is the very first time that track has been written to, the sectors on the track will be written in consecutive order. The heads will write a group of null bytes, and then write out the sectors. In the sector headers, the sectors to end of write' (the offset) would be like this:

Sector Number:	0	1	2	3	4	5	6	7	8	9	10
Offset:	11	10	9	8	7	6	5	4	3	2	1

This offset number tells the drive how many more sectors there will be before the gap of nulls. When this track is read back, the heads will again begin to read as soon as they find the correct track, not waiting to look for any kind of marking. As these sectors are read, decoded, and put into the user buffer, these offsets go along with them.

Suppose that when the head began to read, it was over sector five. Then the read would be much like this:

Sector Number:	5	6	7	8	9	10	11	nulls	1	2	3	4	5
Offset	7	6	5	4	3	2	1		11	10	9	8	7

Notice that sector five must be read twice, because it is not likely that the first read began at the beginning of the sector. In practice, the first read of this sector would be discarded for that very reason. In the user buffer, the information would then be moved, but not as you might expect. Sector six will be moved to the beginning of the buffer, followed by the others through sector eleven. The nulls are discarded, and sectors one to five will follow sector eleven.

When this information is written back to the disk, the sector offsets will be changed to reflect the new order of the sectors:

Sector Number:	6	7	8	9	10	11	1	2	3	4	5	nulls
Offset	11	10	9	8	7	6	5	4	3	2	1	

Notice that there is never a zero in the offset byte because this offset must be read before the sector is read, and there is always at least one more to read before the nulls at the time the offset is checked. This is quite important because the track could end with any sector, and the drive must know when to stop reading.

The sixteen bytes of operating system information are not normally used. The user of the disk will never see the information here, but it is possible for programmers to store data there, so long as the space is not required for some future revision of the operating system. It is not suitable for copy protection, but it might be used for storing copyright information, serial numbers, and the like. Your imagination is required here!

The placement of the nulls is the reason why AmigDos disks appear to be good upon first format, yet are reported as bad disks upon later writes. The earlier accesses wrote to good parts of the track; the bad parts landed in the nulls that follow the last data sector, so the disk was reported to be good. On a later write to the disk, the information begins at a different place on the track, so the bad part will no longer be in the gap of nulls.

The next frontier

The Amiga drives still hold quite a few mysteries. Perhaps they will all be revealed in time, perhaps not. The entire subject of hard disks is yet to be explored. Those black boxes yield their secrets slowly, but with persistence, they *can* be discovered.

Further reading:

Clay, Betty, 'More on Amiga File Structure", *Transactor,* Volume 7, Issue 06, May, 1987, pp.72-73.

Grote, Gelfland, Abraham, *Amiga Disk Drives Inside and Out,* Abacus, 5370 52nd Street SE, Grand Rapids, Ml 49508, first edition (1988).

Immers, Richard and Neufeld, Gerald, *Inside Commodore DOS,* Datamost, 20660 Nordhoff Street, Chatsworth, CA 91311-6152, (1984).

Peck, Robert A., Sassenrath, Carl, and Deyl, Susan, *Amiga ROM Kernal Manual,* Volumes I and II, Cornmodore~Amiga, Inc., (1985).

Sterling, Thom, ~Fun with the Amiga Disk Controller", *Amazing Computing,* Vol.1, No.6, July, 1986, pp 73-74.
Pim Publications, Inc., P.O. Bos 869, Fall River, MA. 02722.