G64 (raw GCR binary representation of a 1541 diskette)

- **Document revision:** *1.9*
- **Last updated:** *Feb 19, 2008*
- **Compiler/Editor:** *Peter Schepers*
- **Contributors/sources:** *Markus Brenner, Immers/Neufeld: “Inside Commodore DOS”, Wolfgang Moser*
- **Wiki rendition:** *Eek/Retrograde*

Introduction

This format was defined in 1998 as a cooperative effort between several emulator people, mainly Per Hakan Sundell (author of the CCS64 C64 emulator), Andreas Boose (of the VICE CBM emulator team) and Joe Forster/STA (the author of Star Commander). It was the first real cooperative attempt to create a format for the emulator community which removed almost all of the drawbacks of the other existing image formats, primarily D64. The G64 format is not specifically designed to hold only 1541 images, but they are presently the only G64 images in existance and why this document only refers to the 1541 and D64's.

In this wiki rendition, formatting - especially around tables - has been reworked to make the information more easy to consume.

File Format

The intention behind G64 is not to replace the widely used D64 format, as D64 works fine with the vast majority of disks in existence. It is intended for those small percentage of programs which demand to work with the 1541 drive in a non-standard way, such as reading or writing data in a custom format. The best example is with speeder software such as Action Cartridge in “warp save” mode or Vorpal and V-MAX which write track/sector data in another format other than standard GCR. The other obvious example is copy-protected software which looks for some specific data on a track, like the disk ID, which is not stored in a standard D64 image.

One protection method that G64 has trouble emulating is data alignment between tracks. Some protection methods rely on data being in exact positions when the head is stepped from one track to another. Imagine two concentric circles representing the data tracks, with a drive head reading data from one track, stepping over to the other track and expecting to find some specific data where it is now. Unless you can read track data from a 1541 so it is aligned with the previous track, write it into the G64 appropriately, and also read the resulting G64 data with this alignment in mind, the protection check will likely fail. Other methods like weak bits are also hard to emulate.

G64 has a deceptively simple layout for what it is capable of doing. We have a signature, version byte, some predefined size values, and a series of offsets to the track data and speed zones. It is what's contained in the track data areas and speed zones which is really at the heart of this format.

Each track data area is simply the raw stream of GCR data, just what the read head would see when a diskette is rotating past it. How the data gets interpreted is up to the program trying to access the disk. Because the data is stored in such a low-level manner, just about anything can be done. Most tracks will be in the standard format with with SYNC markers, GAP, header, data blocks and checksums. The arrangement of the data when it is in a standard GCR sector layout is covered at the end of this document. It is the tracks that don't follow the standard which are the reason for G64's existance and the hardest to decode.

Below is a dump of the header, broken down into its various parts. Following that is a breakdown of the track offset and speed zone offset areas, as they demand much more explanation.

Now, why are there 84 tracks defined when a normal D64 disk only has 35 tracks? By definition, an image of a 1541 must include all the tracks that a real 1541 can access, which is at most 42 tracks and 42 half tracks. Even though using more than 35 tracks is not typical, it was important to define this format from the start with what the 1541 is capable of doing, and not just what it typically does. Some 1541 drives may have problems reading past track 40, and pushing the head past track 42 might be somewhat hazardous to the health of the drive as the head could get stuck.

The typical value seen for the maximum track size is 7928. This is the value used for 1541 images which use standard GCR encoding. This value is determined by the fastest write speed possible (speed zone 0), coupled with the average rotation speed of the disk (300 rpm), and assuming normal Commodore GCR data formatting. After some math, the answer that actually comes up is 7692 bytes. Allowing for a slower disk rotation of -3%, which would allow more data to be written, and some rounding, 7928 bytes per track was arrived at.

Even though it might appear so, it is very important to know that this maximum track size value is not a fixed or hard-coded value. This value depends on the what the original disk was and the GCR encoding used. Non-1541 images such as SFD1001 or 8050 will result in different, likely larger, track sizes. Also, disks with non-standard GCR encoding like those using V-MAX can result in tracks exceeding 8000 bytes.

Since it is a flexible format in both track count and track byte size, file sizes can vary greatly. However, given a few constants like 42 tracks with no halftracks, a consistent track size of 7928 bytes and no speed offset entries, the typical file size will be 333744 bytes.

In my investigation using MNIB (a utility by Markus Brenner that allows you to nibble a 1541 diskette to the PC in G64 format) on a cleanly formatted 1541 disk (using the built-in 1541 format command), I saw the following numbers, compared with the defaults that MNIB uses:

Note that the first size number (7720) is larger than previously mentioned track size of 7692. Why? Likely the drive that I used to create and nibble the clean disk was rotating a little bit slower than 300 RPM (~299 RPM), so more data than normal was stored on each track. I calculated the percentage difference between my numbers and the established benchmark of 7692, multiplied all my numbers by this factor, and arrived at the following chart:

See how close the real numbers come to what MNIB uses? I can attribute the differences of a few bytes to my own rounding errors. Therefore I conclude that the numbers MNIB uses can be taken as the standard that all 1541-compatible G64 tracks should be created with.

All of the above calculations are shown here to establish a safe benchmark to create G64 images in the event that someday we can copy them back to a real 1541 disk. If the G64 track size was too large, it might happen that the track cannot be written back out. By using the above MNIB track size numbers, this problem should be alleviated.

Below is a dump of the first section of a G64 file, showing the offsets to the data portion for each track and half-track entry.

The track offsets require some explanation. When one is set to all 0's, no track data exists for this entry. If there is a value, it is an absolute reference into the file (starting from the beginning of the file).

If an image stored here only contains 35 tracks (e.g. a standard 1541 disk), then all the offset values for track 35.5 and higher will be set to 0. This can be used to detect the maximum track count when converting to a D64 image. Since D64's cannot hold over 40 tracks, and typically only have 35, some information will be lost when converting a G64.

From the track 1.0 entry we see it is set for $000002AC. Going to that file offset, here is what we see...

Following the track data is filler bytes. In this case, there are 368 bytes of unused space. This space can contain anything, but for the sake of those wishing to compress these images for storage, they should all be set to the same value. In the sample I used, these are all set to $FF.

Below is a dump of the end of the track 1.0 data area. Note the actual track data ends at address $20B9, with the rest of the block being unused, and set to $FF.

Now we can look at the speed zone area. Below is a dump of the speed zone offsets.

Starting at $02AC is the first track entry (from above, it is the first entry for track 1.0)

The speed offset entries can be a little more complex. The 1541 has four speed zones defined, which means the drive can write data at four distinct speeds. On a normal 1541 disk, these zones are as follows:

Note that you can, through custom programming of the 1541, change the speed zone of any track to something different (change the 3 to a 0) and write data differently.

From the above speed zone sample, all the zones use 4-byte entries in lo-hi format. If the value of the entry is less than 4, then there is no speed offset block for the track and the value is applied to the whole track. If the value is greater than 4 then we have an actual file offset referencing a speed zone block for the track.

In the above example shown, there were no offsets defined, so no speed zone block dump can be shown. However, I can define what should be there. You will have a block of data, 1982 bytes long. Each byte is encoded to represent the speed of 4 bytes in the track offset area, and is broken down as follows:

It was very smart of the designers of the G64 format to allow for two speed zone settings, one in the offset block and another defining the speed on a per-byte basis. If you are working with a normal disk, where each track is one constant speed, then you don't need the extra blocks of information hanging around the image, wasting space.

What may not be obvious is the flexibility of this format to add tracks and speed offset zones at will. If a program decides to write a track out with varying speeds, and no speed offset exist, a new block will be created by appending it to the end of the image, and the offset pointer for that track set to point to the new block. If a track has no offset yet, meaning it doesn't exist (like a half-track), and one needs to be added, the same procedure applies. The location of the actual track or speed zone data is not important, meaning they do not have to be in linear order since they are all referenced by the offsets at the beginning of the image.

Analysing the GCR data stream

Since the information stored in the track data area is in GCR format, it is not as simple to analyse as a normal 256-byte sector would be. Here is a dump of a portion of the GCR data, and what to look for...

We need to establish a marker by which one can start to interpret the data. Always look for a group of at least 10 1-bits (two 'F's in a row and a bit more), as they establish the SYNC mark. The 1541 actually writes out a SYNC mark of 40 'on' bits (10 'F's in a row). Note that there are 2 groups of SYNC marks quite close together, one for the sector header and one for the sector data. In the above example, there is 2 groups of “FF FF FF FF FF”. The first one is the header SYNC and the second one is the data SYNC.

An important point here: some documentation refers to the minimum SYNC mark as being at least 12 bits wide, and claims that one of that size is still not entirely reliable. Thus Commodore chose to use 40 bits for the SYNC mark, making it impossible for the drive read electronics to miss.

If the GCR data is not in the standard sector layout, then anything goes for interpreting the data. If no standard SYNC mark can be found, then there is no simple way to extract any useful data.

Here's the layout of a standard low-level pattern on a 1541 disk. Use the above example to follow along.

The 10 header info bytes (#2) are GCR encoded and must be decoded down to it's normal 8 bytes to be understood. Once decoded, its breakdown is as follows:

The header gap (#3) is 8 bytes on an early model 1540/1541, but 9 bytes on a later model 1541 and 4040. The 1541 doesn't read the header gap, but simply waits it out to write out the sector data. When sector data is written, the SYNC mark is re-written as well.

There is some controversy over the header gap (#3). Most people assume it to be 9 bytes of `0x55` characters, but the early 1540/1541 drives used only 8. This caused an write incompatability with the existing 4040 disks of the day. In 1541 ROM revision 901225-3 this error was fixed, and now all drives write out 9 of the `0x55` characters for the gap. The book “Inside Commodore DOS” by Immers/Neufeld documents the write incompatibilty and what corruption happens at a low level when writing to a disk with a header gap of 8 bytes on a disk that normally expects a gap of 9 bytes.

The tail gap (#6) is the unused space between the end of one data block and the start of the next. It will vary in size depending on what track you are on, how fast the drive that created the disk was rotating at, and what program was used to format the disk. The stock 1541 format code is supposed to determine how big a track is and divide up the extra unused space into each tail gap. However, many disks will show a much larger tail gap between the last sector and sector 0. In tests that the author conducted on a real 1541 disk, gap sizes of 8 to 19 bytes were seen.

The 325 byte data block (#5) is GCR encoded and must be decoded to its normal 260 bytes to be understood. For comparison, ZipCode Sixpack uses a 326 byte GCR sector (why?), but the last byte (when properly rearranged) is not used. The data block is made up of the following:

The most reliable way to read G64 track data is to read it as bits, not bytes as there is no way to be sure that all the data is byte-aligned. This simulates the way a 1541 drive reads data as well as the head only reads bits as well. The starting location of the track data is know, as well as the track size so the boundaries of the track limits (start and end) are obtainable.

What follows is a very simply point-form list of how to read data, finding sync marks, header blocks and sector blocks.

1. Search for SYNC (at least 10 or more 1 bits)
2. Check for header id after SYNC (GCR `0x52`)
3. If header, read the remaining 9 header bytes
4. Decode header and get sector value
5. Search for SYNC again

6. Check for data id after SYNC (GCR 0x55).
7. If data, read and store with previous header.
8. Have we finished reading the track... stop
9. Start over