GCR decoding on the fly

linusakesson.net/programming/gcr-decoding/index.php

1541 isn't fast enough to decode in realtime.

— MagerValp

The little routine presented here is, with all due humbleness, hands down the best code I ever wrote. It is esoteric and might not appeal to the masses, and it certainly has no commercial value. But in terms of personal satisfaction at a job well done, this is about as good as it gets. Here's why:

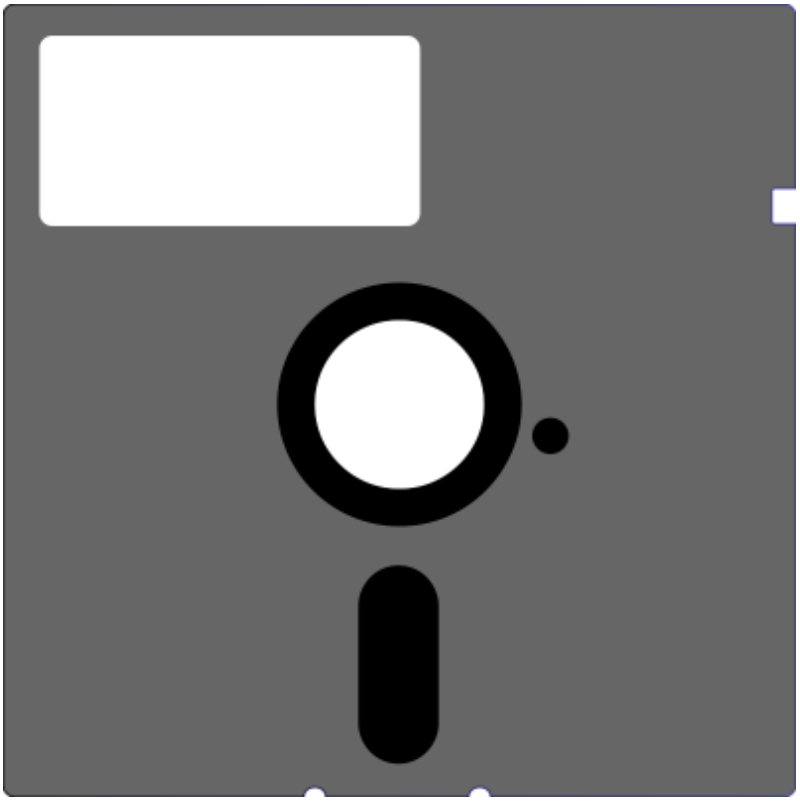
- The code solves a real, practical problem that people have been struggling with for 30 years.
- The key to the solution lies in looking at the problem from a novel angle.
- The implementation would not be possible without plenty of trickery and clever coding techniques. Several instructions do more than one thing.
- The routine is incredibly compact, and leaves very little room for variation; it is THE solution.

Due to the strict timing requirements, there was no way of testing the routine before it was complete. In contrast with the prevalent fashion of trial-and-error software development, writing this code was an exercise of the mind, and my only tools were pen, paper and text editor.

Acknowledgements

I would like to thank Krill of Plush for creating an excellent loader, documenting some of its inner workings and releasing the entire source code, and Ninja of The Dreams for his concise reference material All about your 1541.

The problem



Commodore compatible 5.25" floppy disks store data as magnetic polarity reversals along 35 concentric tracks on the surface. The drive positions the read head (a coil and an amplifier) over a track, spins the disk, and reads off a stream of ones and zeroes. A switch in polarity indicates a one, whereas no switch in polarity for a predetermined amount of time indicates a zero. But because the spindle motors in different drives won't go at exactly the same speed, it is not permitted to store more than two consecutive zeroes on a track. Otherwise there's a risk that the floppy drive counts off too many or too few zeroes, and the data comes out all wrong.

Hence the data is encoded using a scheme called *GCR* (Group Code Recording). For every four bits of data (also known as a *nybble*), five bits are written to the disk surface, chosen from a table. Let's refer to them as a *quintuple*. The encoding is designed to ensure that, no matter how you order the quintuples, there won't ever be more than two zeroes in a row.

Nybble	Quintuple	Nybble	Quintuple
0000	01010	1000	01001
0001	01011	1001	11001
0010	10010	1010	11010
0011	10011	1011	11011
0100	01110	1100	01101
0101	01111	1101	11101
0110	10110	1110	11110
0111	10111	1111	10101

When reading a disk, the floppy drive must convert the incoming bit stream, five bits at a time, into the original nybbles. The bits are, however, lumped together eight at a time, and handed off as bytes to a 6502 CPU. To indicate that a new

byte has arrived, the overflow flag in the processor status register is set by means of the SO pin. It is then up to the CPU to extract the quintuples from the bytes, decode them according to the GCR scheme, and join the resulting nybbles into bytes. This is a realtime job, and the deadlines are very tight: On tracks with the highest storage density, a new byte of raw GCR-encoded data arrives every 26 microseconds.

```
11111222 22333334 44445555 56666677 77788888
```

The quintuples are packed into bytes, so that each byte contains parts of up to three different quintuples. The least common multiple of five and eight is 40, so after 40 bits we're back in phase. Thus, a GCR decoder generally contains a loop that deals with five bytes in each iteration.

Commodore's approach

Some of the floppy drives produced by Commodore sported 2 MHz processors and large ROM chips with lookup-tables to facilitate the GCR decoding. However, Commodore also made the 1541, a low-cost drive, which turned out to be immensely popular, and has earned its place in the standard setup used for C64 demo compos (an unexpanded C64 and a 1541 floppy drive). The 1541 contains a single 6502 CPU clocked at 1 MHz, 2 kB of RAM and 16 kB of ROM.

With such modest specifications, Commodore concluded that the 1541 was not capable of decoding the GCR data in realtime. Instead, the raw data for a complete 256-byte sector is stashed away in a buffer (of 320 bytes, because of the 5:4 expansion). Only afterwards does the CPU revisit each byte in the buffer, use a series of shifting and masking operations to bring each quintuple into the lower part of an index register, and perform a lookup in a ROM table of 32 entries. Actually, there are two such ROM tables, one containing low nybbles and one containing high nybbles, to be joined into a byte using bitwise-or. During this non-realtime pass, the sector checksum is also computed and verified.

```
loop
        bvc     loop
        clv
        lda     $1c01
        sta     ($30),y
        iny
        bne     loop
```

The fetch loop in the 1541 ROM simply reads all the bytes and stores them in a buffer. First it busy-waits until the overflow flag is set. Then it clears the flag in anticipation of the next byte. Then, the current byte is fetched from the hardware register (`$1c01`) and stored into memory. This snippet only reads 256 bytes, but it is followed by a similar loop that reads the remaining bytes.

The loop needs 19 clock cycles to complete one iteration (assuming the `bvc` instruction falls through immediately). This leaves several cycles of slack, to account for differences in motor speed across drives.

A loop with more than 26 cycles would miss bytes, whereas a 25-cycle loop would work well in practice. At exactly 26 cycles you would be pushing your luck.

People were not happy with the performance of the original 1541 ROM routines, not only because of the slow GCR decoding, but also because of the dreadfully slow serial protocol used to transmit the decoded data to the computer. The one thing that Commodore got right in that protocol, though, is that they provided a way to upload code into the 1541, and thus to take over the floppy drive completely. The replacement code and any data buffers must fit in the 2 kB of RAM, but this is enough if all you want to do is load data as quickly as possible from the disk and transmit it as quickly as possible to the computer. Such software is called a *fastloader*.

Krill's approach

As a shining example of a modern fastloader, let's have a look at how GCR decoding is performed in Krill's loader. As

expected, there's a loop that receives five bytes of GCR encoded data, and performs as much partial decoding as possible in between receiving the bytes (i.e. while the sector is passing the read head). The goal is to minimise the execution time of the second, non-realtime decoding pass, without overstepping the maximum execution time of the fetch loop, 130 cycles (26 cycles times five). The partially decoded data is stored in two buffers (512 bytes in total), corresponding to the partially decoded high and low nybbles.

But Krill also realised that there is no need to synchronise with the overflow flag for every byte.

Each track has a statically assigned *bit rate*. As the disk spins at a constant angular velocity (300 rpm), and the bit density of the medium is constant, it is possible to store more bits on the outermost tracks. Commodore defined four bit rates: A byte arrives every 26, 28, 30 or 32 cycles.

After a synchronisation, the first, second and third byte of data are present in the hardware register according to the following schedule. Thus, if we make sure to read the register during the safe areas indicated on the last row, we can read up to three bytes after a `bvc` loop.

```
           cycle
bit rate   0         10        20        30        40        50        60        70
80        90
0
1111111111111111111111111111112222222222222222222222222222223333333333333333333333333333
3333
1
1111111111111111111111111111122222222222222222222222222222333333333333333333333333333333

2
1111111111111111111111111111222222222222222222222222222233333333333333333333333333333
3
11111111111111111111111111222222222222222222222222222333333333333333333333333333
safe areas 1111111111111111111-----      ---222222222222-----            ---333333---
--
```

Because of slight variations in the timing (due to the three-cycle jitter introduced by the `bvc` loop, and the fact that the bit detection timer is reset after every one-bit), it is advisable to read the hardware register a couple of cycles into the safe area, but not much later.

As the loop deals with five incoming bytes per iteration, it is sensible to synchronise twice: Once in anticipation of three bytes, and once in anticipation of two bytes.

After a byte has been read, it is subjected to a number of shifting and masking operations, and the partially decoded quintuples are stored away. During the non-realtime pass, the decoding is completed and the checksum is verified. Interestingly, in Krill's loader, the nybbles are not joined together into bytes, but remain in separate buffers. This is because they are going to be transmitted serially anyway, and thus split up into even smaller parts.

The epiphany

As part of my efforts to learn C64 demo programming, I became interested in GCR decoding. One day, in a flash of inspiration, I started thinking about what would happen if all the shifting operations were removed. Consider the second quintuple. It arrives partly in the first byte, partly in the second, and normally you'd try to right-justify it in an index register. For the second quintuple, it's easiest to shift to the left, like so:

```
        11111222 22333334
        -----222 22------       mask
        ---22222 --------       shift
```

(The digits represent data from the different GCR-encoded quintuples. The dashes are zero-bits.)

But what if you, instead of shifting, combined the masked parts into a single byte using bitwise-or?

```
11111222 22333334
-----222 22------       mask
22---222                bitwise-or
```

That is to say, the three most significant bits remain in the low bits while the two least significant bits remain in the high bits. We could of course use this value as an index into a full-page lookup table, but it would be a sparse table. First, we've masked out three of the bits, so we can only reach 32 out of the 256 entries in the table. But actually, because these are GCR encoded values, we'll only access 16 out of those 32 entries, corresponding to the valid quintuples.

Suppose we do this with all eight quintuples, obtaining eight index values:

```
11111---
22---222
--33333-
4444---4
5---5555
-66666--
777---77
---88888
```

Now, if we were to allocate a full page of RAM for each of these sparse tables, we'd use up the entire 2 kB. But here's the kicker: The whole point of GCR encoding is to ensure that there can never be a sequence of three consecutive zeroes in the bit stream. Since we're masking out three bits, we are introducing a sequence of three zeroes! So every index contains exactly one sequence of three or more zero-bits, and this can be regarded as a key that selects one out of eight sets of table entries. If these sets are non-overlapping, we can combine all eight tables into a single page of memory.

Unfortunately the sets overlap, for the following reason: A GCR-encoded quintuple may begin and/or end with a single zero bit. If the index value contains a sequence of exactly three zeroes, we know that these are the key. If there is a sequence of five zeroes, we know that the middle bits are the key. But if the index value contains a sequence of four zeroes, then we don't know if it's the key followed by a GCR-bit, or a GCR-bit followed by the key. Example: Index `00010110` contains a sequence of four zero-bits (bits 0, 7, 6 and 5, wrapping around from the right edge to the left edge). We don't know if this should be interpreted as `--33333-` (the quintuple would be `01011` which is decoded as `0001`) or `---88888` (the quintuple would be `10110` which is decoded as `0110`).

However, if we use two pages of memory, one for the odd quintuples and one for the even quintuples, then there can be no collisions: If there are four consecutive zeroes in an index value, we know that the three that make up the key must begin at an odd or even bit position. Continuing the example from the previous paragraph, at index `00010110` we'd put the decoded nybble `0001` in the odd table and the decoded nybble `0110` in the even table.

As a bonus, remember that the quintuples are alternating between the high and low nybbles of the decoded bytes. In the odd table, we can shift all the values into the high nybble. This way, to form the final byte out of the first two nybbles, all we have to do is compute the bitwise-or of `odd_table[11111---]` and `even_table[22---222]`.

This is what the tables look like (entries marked "`--`" are not accessed):

Even quintuples

```
-- -- -- -- -- -- -- -- -- 08 00 01 -- 0c 04 05
-- -- 02 03 -- 0f 06 07 -- 09 0a 0b -- 0d 0e --
```

```
-- 02 -- -- 08 -- -- -- 00 -- -- -- 01 -- -- --
-- 03 -- -- 0c -- -- -- 04 -- -- -- 05 -- -- --
-- -- 08 0c -- 0f 09 0d 02 -- -- -- 03 -- -- --
-- 0f -- -- 0f -- -- -- 06 -- -- -- 07 -- -- --
-- 06 -- -- 09 -- -- -- 0a -- -- -- 0b -- -- --
-- 07 -- -- 0d -- -- -- 0e -- -- -- -- -- -- --
-- -- 00 04 02 06 0a 0e -- -- -- -- -- -- -- --
08 09 -- -- -- -- -- -- -- -- -- -- -- -- -- --
00 0a -- -- -- -- -- -- -- -- -- -- -- -- -- --
01 0b -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- -- 01 05 03 07 0b -- -- -- -- -- -- -- -- --
0c 0d -- -- -- -- -- -- -- -- -- -- -- -- -- --
04 0e -- -- -- -- -- -- -- -- -- -- -- -- -- --
05 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

Odd quintuples

```
-- -- -- -- -- 00 -- 40 -- 20 -- 60 -- a0 -- e0
-- -- 80 -- 00 -- 10 -- -- -- c0 -- 40 -- 50 --
-- 80 -- 90 20 -- 30 -- -- -- f0 -- 60 -- 70 --
-- -- 90 -- a0 -- b0 -- -- -- d0 -- e0 -- -- --
-- 00 20 a0 -- -- -- -- 80 -- -- -- -- -- -- --
00 -- -- -- -- -- -- -- 10 -- -- -- -- -- -- --
-- 10 30 b0 -- -- -- -- c0 -- -- -- -- -- -- --
40 -- -- -- -- -- -- -- 50 -- -- -- -- -- -- --
-- -- -- -- 80 10 c0 50 -- 30 f0 70 90 b0 d0 --
20 -- -- -- -- -- -- -- 30 -- -- -- -- -- -- --
-- c0 f0 d0 -- -- -- -- f0 -- -- -- -- -- -- --
60 -- -- -- -- -- -- -- 70 -- -- -- -- -- -- --
-- 40 60 e0 -- -- -- -- 90 -- -- -- -- -- -- --
a0 -- -- -- -- -- -- -- b0 -- -- -- -- -- -- --
-- 50 70 -- -- -- -- -- d0 -- -- -- -- -- -- --
e0 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

Beautiful, aren't they?

Let's recap the algorithm:

Read five bytes.

```
        11111222 22333334 44445555 56666677 77788888
```

Extract twelve fragments using bitwise-and.

```
        11111--- -----222 --33333- -------4 ----5555 -66666-- ------77 ---88888
                 22------          4444---- 5-------          777-----
```

Combine the fragments into eight bytes using bitwise-or.

```
        11111--- 22---222 --33333- 4444---4 5---5555 -66666-- 777---77 ---88888
```

Retrieve decoded nybbles from the tables.

The digits represent decoded bits from now on.

```
1111---- 3333---- 5555---- 7777----
----2222 ----4444 ----6666 ----8888
```

Join them into bytes using bitwise-or.

```
11112222 33334444 55556666 77778888
```

Store the four bytes in a buffer.

You might be thinking: This is a novel and quite interesting approach and all, but there's nothing to suggest that it would be faster than ordinary shifting. And, indeed, much work remains.

The tricks

We'll use plenty of self-modifying code. This is not really a trick, but an established technique on the 6502 processor. In order to shave off a cycle from every instruction that modifies another one, the entire loop will be placed in the zero-page.

Rather than storing the decoded bytes using an indexed addressing mode (five cycles per byte, plus a few more to advance the index register), we'll push them on the stack (three cycles per byte, pointer automatically decremented, and both index registers are free for other purposes). The 6502 stack is restricted to one page, so a complete sector fits perfectly. The 257th byte (checksum) is handled as a special case after the loop. The downside of this method is of course that we cannot use the stack for anything else. In particular, we can't make any subroutine calls.

For masking, we can use the `and` instruction, which puts the result in the accumulator. However, sometimes we'd rather preserve the contents of the accumulator. Using *unintended* (also known as "illegal") opcodes, there are a couple of ways to achieve this: The `sax` instruction is a mix of `sta` and `stx`; it tries to write the contents of both `A` and `X` to memory, and due to the NMOS process used to manufacture the 6502, zeroes are stronger than ones, so bus contentions resolve into bitwise-and operations. Thus, we can use `sax` to store a subset of the bits in `A` (or `X`) to memory without clobbering the rest of the register. Furthermore, we can perform a bitwise-and operation using the `sbx` instruction (a mix of `cpx` and `tax`) with an operand of zero, which will leave the result in `X`.

We can obtain two copies of each incoming byte: The unintended opcode `lax` is a mix of `lda` and `ldx`, and can be used to fetch an encoded byte into both `A` and `X`.

To combine index parts, we can use a bitwise-or operation as described above, but of course addition would work equally well. Addition can be computed implicitly as part of an indexed memory access: Since the odd and even tables are page-aligned, we can store some of the index bits in the least significant byte of the instruction operand, and the remaining bits in an index register.

Finally, consider quintuple number four: It arrives in two parts, one of which is simply the least significant bit of the second incoming byte. Rather than mask this bit and keep it in a register, we can use the `lsr` instruction (the only shifting operation in the entire GCR decoder) to place it in the carry flag. To combine it with the other part of the index, we simply `adc` (add-with-carry) a constant zero. This frees up a register, but that's not all: As a side effect, the addition operation will clear the overflow flag. By executing the `adc` at the right moment, we can get rid of one `clv` instruction, saving two cycles.

Armed with these tools, we are faced with a magnificent puzzle of instructions, registers, bits and cycles. When taking on this challenge, I had no idea if it would be possible to find a solution. By the time I had only a few cycles left to optimise, I could hardly think of anything else for two days. But then everything clicked into place.

The code

The following loop decodes GCR on the fly. The checksum must still be verified in a separate pass.

Start at the label `begin_reading_here`; that is the logical beginning of the loop. In practice, the first iteration is handled partly outside the loop, which is entered at `zpc_entry` ("zpc" is short for zero-page code).

```
zpc_loop
                ; This nop is needed for the slowest bit rate, because it is
                ; not safe to read the third byte already at cycle 65.

                ; However, with the nop, the best case time for the entire loop
                ; is 130 cycles, which leaves absolutely no slack for motor speed
                ; variance at the highest bit rate.

                ; Thus, we modify the bne instruction at the end of the loop to
                ; either include or skip the nop depending on the current
                ; bit rate.

                nop

                lax     $1c01           ; 62 63 64 65   44445555
                and     #$f0            ; 66 67
                adc     #0              ; 68 69         A <- C, also clears V
                tay                     ; 70 71
zpc_mod3        lda     oddtable        ; 72 73 74 75   lsb = --33333-
                ora     eventable,y     ; 76 77 78 79   y = 4444---4, lsb = -------
-

                ; in total, 80 cycles from zpc_b1

zpc_b2          bvc     zpc_b2          ; 0 1

                pha                     ; 2 3 4         second complete byte
(nybbles 3, 4)
zpc_entry
                lda     #$0f            ; 5 6
                sax     zpc_mod5+1      ; 7 8 9

                lax     $1c01           ; 10 11 12 13   56666677
                and     #$80            ; 14 15
                tay                     ; 16 17
                lda     #$03            ; 18 19
                sax     zpc_mod7+1      ; 20 21 22
                lda     #$7c            ; 23 24
                sbx     #0              ; 25 26

zpc_mod5        lda     oddtable,y      ; 27 28 29 30   y = 5-------, lsb = ----
5555
                ora     eventable,x     ; 31 32 33 34   x = -66666--, lsb = -------
-
                pha                     ; 35 36 37      third complete byte
(nybbles 5, 6)

                lax     $1c01           ; 38 39 40 41   77788888
                clv                     ; 42 43
```

```
                and     #$1f                ; 44 45
                tay                         ; 46 47

                ; in total, 48 cycles from b2

zpc_b1          bvc     zpc_b1              ; 0 1

                lda     #$e0                ; 2 3
                sbx     #0                  ; 4 5
zpc_mod7        lda     oddtable,x          ; 6 7 8 9       x = 777-----, lsb = ------
77
                ora     eventable,y         ; 10 11 12 13   y = ---88888, lsb = -------
-
                pha                         ; 14 15 16      fourth complete byte
(nybbles 7, 8)

begin_reading_here
                lda     $1c01               ; 17 18 19 20   11111222
                ldx     #$f8                ; 21 22
                sax     zpc_mod1+1          ; 23 24 25
                and     #$07                ; 26 27
                tay                         ; 28 29
                ldx     #$c0                ; 30 31

                lda     $1c01               ; 32 33 34 35   22333334
                sax     zpc_mod2+1          ; 36 37 38
                ldx     #$3e                ; 39 40
                sax     zpc_mod3+1          ; 41 42 43
                lsr                         ; 44 45         4 -> C

zpc_mod1        lda     oddtable            ; 46 47 48 49   lsb = 11111---
zpc_mod2        ora     eventable,y         ; 50 51 52 53   lsb = 22------, y = -----
222
                pha                         ; 54 55 56      first complete byte
(nybbles 1, 2)

                tsx                         ; 57 58
BNE_WITH_NOP    =       (zpc_loop - (* + 2)) & $ff
BNE_WITHOUT_NOP =       (zpc_loop + 1 - (* + 2)) & $ff
zpc_bne         .byt    $d0,BNE_WITH_NOP    ; 59 60 61      bne zpc_loop
```

Final remarks

Of course, once I had figured out the solution, I needed to write a fastloader around it to see whether it would really work. The loader grew into the *Spindle* trackmo toolchain, which I am saving for another article. To see the GCR decoder in action, have a look at my demo Shards of Fancy.

Posted Sunday 31-Mar-2013 16:40

Discuss this page

Disclaimer: I am not responsible for what people (other than myself) write in the forums. Please report any abuse, such

as insults, slander, spam and illegal material, and I will take appropriate actions. Don't feed the trolls.

Jag tar inget ansvar för det som skrivs i forumet, förutom mina egna inlägg. Vänligen rapportera alla inlägg som bryter mot reglerna, så ska jag se vad jag kan göra. Som regelbrott räknas till exempel förolämpningar, förtal, spam och olagligt material. Mata inte trålarna.

Anonymous
Tue 2-Apr-2013 21:59

Truly awesome work, I am more of a Z80 kinda guy, but I nontheless find this really interesting, particularly the use of "illegal" op-codes and the zero page to speed things up.

Ilbit
Jesper Öqvist
Tue 2-Apr-2013 22:12

Really impressive and interesting article. Good job!

Anonymous
Sun 7-Apr-2013 03:01

http://www.templeos.org/Wb/Kernel/Compress.html

Anonymous
Sun 7-Apr-2013 18:19

This mas is a true genius. Big respect to You

best regards - wegi

Anonymous
Tue 9-Apr-2013 20:49

Interesting read indeed - I wonder though wether some 1541 clone may use a 65c02 and break the code due to the used illegal ops.

Can't help it - don't see a relation to that compress link in the comments there :)

Anonymous
Fri 12-Apr-2013 06:17

Modifying the bne depending on rate? Brilliant. Worthy even of Mel. http://mark.aufflick.com/blog/2003/11/24/the-one-true-mel

Anonymous
Fri 12-Apr-2013 06:21

Since I'm mostly interested in the 6502 for Apple][I didn't quite know where to bookmark this link. So I made a new folder called "Awesome".

Anonymous
Fri 12-Apr-2013 15:03

Impressive. Thank you for this detailed article.

Anonymous
Mon 15-Apr-2013 21:31

Well done.
Since the values of the even table use only the low nybble and those of the odd use only the high nybble, you could overlay these into a single table. I guess there's no reason to do this, though, since it would require more masking and you were able to fit both tables into the 2 kB of RAM.

Anonymous
Tue 16-Apr-2013 23:11

All nice, but I prefer MFM as it stores much more.
But in all honesty that might only have been possible on the Amiga ;-).
Built some efficient MFM loaders for that myself too.

@reumerd

Anonymous
Sat 29-Jun-2013 03:08

Hi Linus,

I've just watched quite a detailed talk about the 6502, that I thought you'd sure like (https://www.youtube.com/watch?v=K5miMbqYB4E). More info is available at visual6502.org. They've analyzed and simulated the chip based on high-res photographs.

-- jn

Anonymous
Tue 2-Jul-2013 22:40

This is a great achievement! So, in your fast loader, did you finally succeed to load a complete track including decoding and checksum test within two revolutions?

lft
Linus Åkesson
Thu 4-Jul-2013 23:33

This is a great achievement! So, in your fast loader, did you finally succeed to load a complete track including decoding and checksum test within two revolutions?

Not including serial transfer, because my goal was to make an IRQ loader. This adds some overhead to the serial protocol, as it must be able to cope with arbitrary delays on the C64 side. I am however down to three revolutions for tracks 18-35 and four for tracks 1-17, as measured on a static display with 25 badlines and no sprites.

Anonymous
Sun 7-Jul-2013 23:13

My hardware-speeder, S-JiffyDOS, can also decode from GCR to hexdec while reading from the disk. It also uses LAX and the the stack. It uses many ROM-tables (so it's 32 instead of 16kB) and it syncronices with BVC every 5th byte only. It's on www.nlq.de
Your short routine that fits into the 1541-RAM is fantastic.

chesterbr
Carlos Duarte do Nascimento
Tue 14-Jan-2014 06:43

Both the decoding table approach and the smart usage of undocumented opcodes are a great job, kudos. I've never

thought of the "zero wins" as a side effect of NMOS, but once you said it, it all made sense to me, pretty much like understanding the Matrix! :-) Great article.

Anonymous
Fri 25-Apr-2014 18:04

Great work. I think if you made receive loops timed for each of the 4 bitrates, you could read 5 bytes before resynching. You could probably even do it with 2 loops (each covering 2 bitrates).
Take a look at the code from Epyx Winter Games if you haven't already. Other than the directory track, the disk doesn't use GCR. And I'm pretty sure (memory isn't perfect after almost 30yrs) it transferred 253 or 254 bytes at a time, pushing them on the stack on the c64 side in order to save a cycle vs sta.

Anonymous
Wed 5-Nov-2014 21:31

Awesome work.

In interest of pure speed, I would love to see this used with a non-IRQ/no-display loader for 2 revolution track transfer. I've looked into it and don't think it's possible on highest density tracks unfortunately.

From the end of this processing until the sector header 2 away you have:
- current sector tail - 2 00s + 4 or more gap = 6+ bytes
- skipped sector header - 5 sync + 10 contents + 9 gap = 24 bytes
- skipped sector data - 5 sync + 325 data + 4 or more gap = 334+ bytes
- for total of 364+ bytes (note these are all in terms of encoded (GCR) not decoded bytes)

So 364 is the minimum number of disk bytes that may fly by in the time you need to transfer the sector just read. After that you need to go back to reading a sector header, etc.

At highest density setting, this is 26us per byte, or 26 * 364 = 9464us.

The fastest serial bit-bang transfer I think has ever been done is Vorpal V1 (like used in Winter Games). It transfers 3 bytes in a 113us loop, for 37.67us / byte.

We need to transfer 257 bytes (256+checksum) in 9464us. This means each byte must be transferred in 9464us / 257 = 36.82us.

So I'd expect Vorpal style serial transfer to allow you to squeeze by with 2 revolutions per track on lower-densities, but just miss the mark and still require 3 revolutions per track on highest density.

I would still love to tweak this code to be more density dependent to eliminate that BVC or two though. That should free enough cycles to replace PHA with STA $1801 and allow a (non-IRQ/no-display) 1 revolution parallel port track transfer :).

Aaron

Anonymous
Fri 27-Feb-2015 09:06

I just had an idea, why bother with decoding GCR at all, how about you use the GCR raw bytes as an index to a 256 byte table to return a 6 bit result. The original bits would have to be scrambled quite a bit resulting in a specially encoded file.

Also, do all the bits occur in order to select some subset of the bits in the GCR? For example, just mask every other bit and call that your decoded data.

Extend this bit mangling even further, make up a new encoding that gives even 2 decoded bits per byte that are stuffed directly to the output (in a 2bit transfer). This is quite wasteful of space however and reduces the data rate quite a bit, but is trivially fast to send and decode.

lft
Linus Åkesson
Fri 27-Feb-2015 13:03

Those are nice ideas, especially the first one. It would slightly reduce the amount of information you can store in a block (6 * 320 / 8 = 240 bytes), so any benefits would have to be weighted against that. You'd also probably want to buffer the data as 320 bytes, either before or after the table lookup, and then transmit each byte as three bit pairs. It's not obvious that a good mapping function actually exists; it should fit into a single page, so the same mapping should work regardless of the alignment between gcr chunks and bytes, even if the encoding would be different for each alignment. I'll think some more about it.