VDC MADE EASY

john.seikdel.net/vdcmadeeasy01.htm

By commodorejohn

Version 0.1

TABLE OF CONTENTS

INTRODUCTION

One of the neatest features of the Commodore 128 (if you ask me) is also one of the most underutilized: the 8563 Video Display Controller chip, which provides the 80-column RGBI video output that, visually, sets the 128 apart from the 64. This is at least partly due to the fact that it's a little slower, practically speaking, than the VIC-II, and also because it requires a special monitor, but probably more thanks to the cryptic and overly-technical nature of most of the scant availabe documentation for the chip. Even in *Mapping The Commodore 128,* otherwise a paragon of clear explanation, the VDC section takes quite a few readings before things begin to make sense. I would like to remedy this, and make things clear to even novice 128 programmers. At least, I'll try. If I've omitted anything, left something unclear, or made an error, kindly drop me a line and I'll correct or improve as necessary. Enjoy!

HOW THE VDC WORKS

The first and most obvious difference between the VDC and the other I/O devices in the Commodore 128 (and, for that matter, just about every other Commodore computer) is that it takes up all of *two* positions in the computer's memory map. Obviously, you can't run a 640x200 video chip off of just 16 bits of data, and in fact you don't. The first VDC register, located at $D600, is a register-select register, and the second is a data register, located at $D601. The VDC is controlled by writing a value to the register-select register to tell the chip which of its internal registers you want to access, waiting for it to be ready (more on that later,) and then writing the value for the selected register into the data register. That's right, *you have to use two registers in the memory map to access all the* actual *registers.* If this seems needlessly complicated to you, you're not alone; 128 designer Bil Herd has this disparity as one of his complaints about the VDC's integration into the computer. The reason this is so out of the normal Commodore operating scheme is because the 8563 originated as part of a later-abandoned computer design based on the Z80 architecture, in which *all* I/O works like this by means of dedicated port-I/O instructions. (Aren't you glad you don't normally have to do this? Pity all those poor Spectrum users with their Z80-based systems ;D) But it's what you're stuck with if you want to use the VDC, and on the bright side, it *does* allow the chip to

have its own dedicated address space outside the memory map, which means a full 16KB of video RAM (64KB if you're the lucky owner of a 128-D or hardcore enough to have upgraded your flat 128) that doesn't take up any more space as far as the CPU is concerned.

ELEMENTARY VDC ACCESS

Anyway, as you may have gathered, reading from and writing to the VDC is a pain in the you-know-what compared to any of the other chips in the system. Which is why Commodore provided a few routines in the computer's ROM to make it easier. The full set will be explained later, but the two routines of immediate interest are WRITEREG, located at $CDCC, and READREG, located at $CDDA. As the names imply, they write to and read from the VDC register set. Being machine-language routines, they are called with the X register set to the register number, and (for WRITEREG) the accumulator/A register set to the value to write. (READREG, obviously, returning the value read in the accumulator.) These are *very* important routines for VDC use, since not only do they make things easier for machine-language programmers, they are the *only* way to access the VDC registers from BASIC. According to official Commodore documentation, the VDC does not look kindly on indirect addressing, which is used by the PEEK and POKE statements normally used for low-level hardware access on Commodore computers. So in order to write to a VDC register from BASIC, you have to use the SYS statement to call these routines, like so:

```
 10 REM WRITE A VALUE TO A VDC REGISTER
20 RE = 31 : VA = 65
30 SYS 52684,VA,RE
```

```
 10 REM READ A VALUE FROM A VDC REGISTER
20 RE = 31
30 SYS 52698,,RE
40 RREG VA : REM PUT THE VALUE READ INTO THE VARIABLE VA
```

Of course, VDC access from *BASIC* will be even slower, but for the sake of completion, there you go. That's the basics of accessing a VDC register.

WRITING TO THE SCREEN

Of course, that doesn't help you much unless you know what the registers *mean.* We'll get to all of them in due time, but the most obvious thing to cover first is accessing the screen memory, in order to write to the screen. This is done by means of three registers, $12, $13, and $1F (or 18, 19, and 31, if you're using BASIC or for some reason like to use decimal values in your assembler code.) Registers $12 and $13 are two halves of a 16-bit address register that specifies where in video RAM to write to, and $1F is the video RAM read/write register. That's right, *another* layer of indirection involved in using the VDC. (Told you it was complicated.) What's even worse is that, unlike any other 16-bit address in the 128, the VDC has its most significant byte (the high eight bits) at the *low* address. Why this is, I couldn't tell

you, since even the Z80 architecture the chip was originally intended to work with uses the same little-endian format as Commodore hardware. But again, best just to shrug and move on. The process for writing to video RAM is therefore to set the address to access via registers $12 and $13 (if you're wondering, $0000 is the start of the character map,) and then write the value you want in that address to register $1F. That's three calls to $CDCC; now do you begin to see why VDC access is considered slow? Fortunately, the address register automatically increments every time a read or write is performed on register $1F, so you don't have to keep setting it if you want to write a block of values in consecutive addresses.

Now, you BASIC users out there may recall the obligatory blurb in Commodore-related texts on POKE-ing 16-bit values like addresses, and it applies to the SYS calls to WRITEREG as well. But here it is again, in case you don't remember it: SYS, like POKE, can only handle eight-bit values (numbers from 0 to 255.) (And you'd have to do this anyway, with the registers-within-registers thing the VDC has going on.) So a little finagling is needed to set the write address from BASIC. The trick is that you're going to have to split the 16-bit value to write into two separate 8-bit values and write them individually. Here's an example of how to do it:

```
 10 REM THE ADDRESS TO SET IS IN THE VARIABLE A
20 LO = A AND 255 : REM GET THE LOW BYTE
30 HI = (A / 256) AND 255 : REM GET THE HIGH BYTE
40 SYS 52684,HI,18 : REM SET THE HIGH BYTE
50 SYS 52684,LO,19 : REM SET THE LOW BYTE
60 REM NOW THE ADDRESS IN SCREEN RAM IS SET AND VALUES CAN BE WRITTEN
```

VDC MEMORY ORGANIZATION

Now, that's *how* to write to video RAM. Now how about the question of *what* to write? As previously mentioned, the default location for the character map is at $0000, and the attribute map (analogous to the color RAM for VIC-II video, but more in a bit) is located at $0800 (2048 for you BASIC junkies.) Both of these are relocatable to any address in VDC RAM; the starting positions are defined by 16-bit register pairs like the memory-address registers discussed earlier. The registers for the character map are $0C and $0D (12 and 13,) and the registers for the attribute map are $14 and $15 (20 and 21.) But what are these "maps" of which I speak? Hang on, we're getting to that.

Character Map

The nice thing about the VDC is that, other than being 80 columns wide instead of 40, it's pretty much arranged like the VIC-II, memory-wise. The character map mentioned a paragraph ago controls which character appears at which position, and like on the VIC-II, it's arranged horizontally, with the position address increasing as you go across the screen, and

each line down the screen starting at the position after the rightmost column of the line above it. This means that the start of each line is located at the character map address plus 80 times the line number (assuming the topmost line is number 0.)

Attribute Map

The attribute map, as mentioned previously, is pretty much the same as the color RAM of the VIC-II, with the low nybble (the low four bits) controlling the color of the character at the corresponding position in the character map. However, the colors of the VDC are different than the colors of the VIC-II:

$0 Black
$1 Dark gray (light black)
$2 Dark blue
$3 Light blue
$4 Dark green
$5 Light green
$6 Dark cyan
$7 Light cyan
$8 Dark red
$9 Light red
$A Dark purple
$B Light purple
$C Dark yellow (brown on most monitors, a sort of tarnished-brass color on others)
$D Yellow
$E Light grey (dark white)
$F White

People who've done text-mode programming on PCs might find this familiar; that's because it's basically the DOS text-mode palette with the high and low bits swapped. The reason for this is that both the old IBM CGA card and the VDC use RGBI video, which is limited to precisely these colors. Why the designers of the VDC didn't arrange the palette to match the PC palette, I don't know, but it does, at least, make a bit more organizational sense.

Anyway, that's the low nybble of the attribute map explained. But unlike the VIC-II color RAM, the high nybble *does* mean something; it controls four special attributes of the character, Flash, Underline, Reverse, and Alt (in order of increasing bit significance:)

```
    Alt   Rev   U/L  Flash       Color
  +-----+-----+-----+-----+-----+-----+-----+-----+
Bit | 7 | 6 | 5 | 4 | 3   2   1   0 |
  +-----+-----+-----+-----+-----+-----+-----+-----+
```

The first three should be self-explanatory; Flash makes the character flash on and off at a predefined rate, Underline fills in the bottom of the character to a selected number of lines, and Reverse inverts the on and off pixels of the character pattern. Alt is not as intuitive, but not complicated at all; it serves as a ninth bit for the corresponding position in the character map, meaning a grand total of 512 character patterns can be displayed (as compared to 256 on the VIC-II.)

Character Set

The 512-character set you're given by default is the same as the two 256-character sets used on the VIC-II (uppercase + graphics and uppercase + lowercase,) but that's not particularily practical for use with the VDC. For one thing, this includes reversed character patterns, which the Reverse attribute makes completely unnecessary; that's a full 256 characters that aren't of any use! And there's also a number of repeated characters between the two sets (the reversed and unreversed digits and punctuation, for example.) Fortunately, as with the VIC-II, you're not stuck with the default character set; in fact, it's not even mapped into ROM like the VIC-II character set. Instead, the 128 ROM copies the character set into VDC RAM on bootup. And since there's nothing else in VDC memory but graphics data, you don't even have to worry about where to put a custom character set like you do with the VIC-II. (Now, lest you think I'm unfairly putting down the VIC-II, let it be said that things like sprites, multicolor mode, and directly-accessible screen memory thrash the VDC six ways from Sunday when it comes to stuff like games.) All you have to do is write your new character set over the old one; it's located at $2000 (8192) in VDC RAM by default.

However, there are two things you should know when writing custom character sets. The first is that character patterns in the VDC take up sixteen bytes by default, as opposed to the VIC-II's eight. This is because the the height of the characters is adjustable. However, a 25-line screen basically requires characters that are eight pixels tall. So your custom characters will be eight bytes of character pattern, followed by eight ignored bytes, and so on for each character. A waste of space, sadly, but there's nothing you can do about it. The second is that VDC pixels are exactly half as wide as VIC-II pixels, but just as tall, so whatever character patterns you use are going to be stretched to twice their height. (Unless you're using interlace mode, but we'll talk about that later.)

Unallocated Space

Now, if you're a math geek like me, you've already figured this out: the 512 16-byte character patterns take up 8KB of space, and the character map and attribute map take up 2000 bytes apiece (80 columns by 25 lines is 2,000 screen positions.) That's another 4KB, given that each map is positioned on a 2KB boundary, for a grand total of 12KB out of our 16KB video RAM. What can you put in that last 4KB? It's all up to you. All the space from $1000 (4096) to $1FFF (8191) in VDC RAM is unused, so you can do what you like with it. It's technically possible to use it for extra character patterns, if you like, but using them will be

complicated, since the maximum 512 patterns are already allocated. It is possible to move the character-set base address down to $0000 (more on this later,) which would make this area the second 256 character patterns, but that means that **A.** you'd be unable to use the other 512 characters at the same time, and, more importantly, **B.** the first 256 patterns would actually be the character and attribute maps. This isn't going to cause your computer to go up in smoke or anything, but it does mean that using the first 256 characters will generate a lot of funky gibberish. So if you do decide to try this, you'll want to have the Alt bit set in every attribute-map position. (Were you attempting to make a game using the VDC, you might use the 512 main patterns for in-game graphics and the 256 alternate patterns as the character set in some kind of submenu that is never displayed at the same time as the game screen.)

Of course, a much simpler use is for alternate character and/or attribute maps. 4KB is enough for another two 80x25 maps; you could have a whole alternate screen by setting up a second character map and attribute map in this space, or if you wanted you could use the same attribute map and have two extra character maps (this would, of course, stick you with the same colors and Alt bit settings on each of the screens, but there are ways to plan around this, depending on what you're doing.) If you wanted, you could even keep the same character map and have extra attribute maps, maybe as a quick way to change colors or something. However, the Kernal text-printing routines will not work on these extra maps.

Another possibility is to use the extra 4KB as extra RAM for your program, as some 128 software already does. Accessing it is going to be much slower than using normal CPU RAM, but if you're tight on space and don't want to require an REU, it could ease the pressure a bit. Before you resort to this, though, remember that using the VDC means that all that RAM that would normally be used by the VIC-II is up for grabs (although this is only 1KB of normal RAM in text mode, plus the color RAM, which is another 1KB but only stores the lower nybble.)

Now, if you wanted to get *really* wild, you could mess around with the entire layout of VDC RAM, period, but this will *completely* break compatibility with the Kernal text-printing routines, so you'd better have your own ready to go. If you're willing to deal with this, one thing you could do is content yourself with 256 characters, set the character-set base address to $0000, and treat the second 256 patterns as free space, giving you an effective 4KB character set and 12KB of space to stick character and attribute maps into as you please; that's six whole 80x25 maps. (If you can make do with 128 characters, you'd get another 2KB to work with, for a total of seven maps, but anything below that won't get you enough space for an extra map.) But, as noted, this means you'll have to do *everything* yourself.