Setting the VGA Palette

Intro

This tutorial will help you gain a higher understanding of the DAC color table, and how that is used by the VGA card to get the proper color onto the screen. Lets not rush into anything too quickly. First off, lets get an understanding of how colors are handled and used on the VGA card.

Color

Every visible color can be split up into 3 intensities of Red, Green, and Blue. This is pretty much common knowledge. The big mystery is how this is represented by the computer. Our eyes do all the interpretation and communication to our brains to accomplish our vision, but what about computers? The answer is quite simple.

For the remainder of this section, I refer to color as one of the 3 primary colors, Red, Green or Blue.

Each color is represented by a number. That number's maximum value represents the brightest that color could ever get. An intensity of 0 means that it is black or not visible at all. The interesting thing about this method is that the precision of the color is completely dependent upon how high the number can get. This tells us how many intensities of that color are possible, while in real life we have an unlimited number of intensities. If we use a 2 bit number to represent our intensities, we would have to split up all possible intensities of that color into 4 steps. If we wanted to represent a real color, we would only come up with a gross approximation. What if we used a 4 bit number. Then we would have to split it up into 16 steps. This still isn't acceptable! This is where 8 bit color comes into play. Here we get 256 intensities for each color. Ironically, this is also the standard number of colors available in a VGA mode. Unfortunately, we only end up with 64 shades of each color, instead of 256. Why are we limited to only 256 colors and 64 shades of each intensity? That's because of the DAC Color Table.

DAC Color Table

The DAC color table provides a very quick way to access a color by its index, instead of always providing the RGB intensities whenever the color needs to be put on the screen. The table has 256 indexes that correspond to our 256 colors available at one time. Each index stores the 8 bit value corresponding to each RGB intensity. A crappy thing about each index, is that only the lower 6 bits are used, forcing only 64 shades of each intensity. To display our color after we fill in the table with the 256 colors we want, we simply write the index number to video ram. Video ram uses our index to go into the table and use the 3 8-bit intensities to display our color correctly.

Palette Registers

Palette Mask	0x3c6
Register Read	0x3c7
Register Write	0x3c8
Data Port	0x3c9

Setting the DAC Color Table

The code to actually set the color table is REALLY Simple. Here's my function ripped from my Video class, it will work with DJGPP(32bit) and C++(16bit).

C++ Functions for Writing

```
#define uchar unsigned char

void Video::Write_DAC_C_Palette(uchar StartColor,uchar NumOfColors,uchar *Palette)
{
  outp(0x03C6,0xff);                    //Mask all registers so we can update any of them
  outp(0x03C8,StartColor);              //1st color to input!
  for(short i=0; i<NumOfColors*3; i++ )
     {outp(0x03C9,Palette[i]<<2);
     }
}

void Video::Set_DAC_C(uchar Color,uchar Red, uchar Green, uchar Blue)
{
  outp(0x03C6,0xff);            //Mask all registers even though we only need 1 of them
  outp(0x03C8,Color);           //Color to set
  outp(0x03C9,Red);
  outp(0x03C9,Green);
  outp(0x03C9,Blue);
}
```

That's right. These functions give us absolute power! The fist function sets the whole palette to the colors we want. The only stipulation is that we have our buffer (Palette) filled in with our colors. We send a value of 0xff to the mask port so that we can update all registers. We then tell it that we are going to start giving it colors starting with index 0. From then on, we simply communicate with the Data Port, giving it our intensities. One point worth mentioning is that these functions only use the lower 6 bits of the data that is sent to it. The DAC only uses 6 of the 8. If we are reading from and writing to the DAC using the functions listed in this tutorial, this won't be a problem since we will actually be reading 6 bit values. While this function is running, any colors that are on the screen that have different RGB values will immediately turn into the new colors. This may or may not be what you want, so be careful! The second functions does close to the same thing except it just modifies a single color! I hope this clears up some of the mystery.

BIOS Functions for Writing

```
#define uchar unsigned char

void Video::Set_DAC(uchar Register,uchar Red,uchar Green,uchar Blue)
{ union REGS regs;
  regs.h.ah=0x10;
  regs.h.al=0x10;
  regs.x.bx=Register;
  regs.h.ch=Green;
  regs.h.cl=Blue;
  regs.h.dh=Red;
  int86(0x10,&regs,&regs);
}

void Video::Write_DAC_Palette(uchar StartColor,uchar NumberOfColors,uchar* Palette)
{ REGPACK regs;
  regs.r_ax=0x1012;
  regs.r_bx=StartColor;
  regs.r_cx=NumberOfColors;
  regs.r_es=FP_SEG(Palette);
 regs.r_dx=FP_OFF(Palette);
 intr(0x10,&regs);
}
```

The first function is to be used to set a single color. If you want to be cool and set all of the color palette at once, use the second. The buffer that you pass it must be 768 unsigned characters. It must represent Red,Green,Blue,Red,Green,Blue etc.

NOTE: The Second function will not work under DJGPP because it is a 32 bit compiler, meaning it doesn't address memory in a SEG:OFF scheme. Modify the SetPalette function from above instead!

Reading the DAC Color Table

You'll find out that reading the colors from the DAC Color Palette is very very close to writing to it! Since we're at it, i thought you might enjoy a little function that loads the palette from the DAC color table, and it's in inline assembly. What else is there ?!

Inline Assembly Functions for Reading

```
void Video::Load_DAC_Palette()
{ address=(long)&Palette;
  asm (“
     movl $768,%%ecx          // cx is our counter:  Section #1
     movw $0x03c6,%%dx        // dx is our port
     movw $0xff,%%eax         // ax is the input number to go to port
```

```
    outw %%ax,%%dx
    movw $0,%%eax             //Section #2
    movw $0x03c7,%%dx      //3c7 to read the contents
    outw %%ax,%%dx
    movw $0x03c9,%%dx      // Section #3
    movl _address,%%ebx
     LoopStart:
    inb %%dx,%%eax
    movl %%eax,(%%ebx)
    addb $1,%%bx
    Loop LoopStart"
    :
    :
  :"%ecx","%eax","%dx","%ebx");
}
```

Now all this is really doing is telling the read port in the DAC table that we want to read from it. And we start to read away! In the first section we set cx to the number of repetitions we are going to have our loop iterate. Its 768 since there are 256 colors, and a Red, Green, and Blue component for each. We also set the mask to 0xff so that all colors can be accessed. In section #2, we tell it that we want it to start giving us the colors starting with color 0. In section #3, we set the location of where we are going to be writing the info, and set the port to 0x03c9, because that's where the data is going to show up! In the loop we read from the data port once, and increment our pointer. The only crappy thing about this approach is that the palette must be global since inline asm doesn't use the stack. I'll get over it :) One other item to mention is that the data read from the DAC is only 6 bits, so if you intend to write it after reading it, don't do any bit shifting because it is already aligned!

NOTE: The above inline assembly code will only work under DJGPP. With a little modification it can be ported to normal inline assembly under normal C++(16).

C++ Functions for Reading

```
#define uchar unsigned char

void Video::Read_DAC_C_Palette(uchar StartColor,uchar NumberOfColors,uchar* Palette);
{ outp(0x03C6,0xff);          //Mask all registers so we can update any of them
  outp(0x03C7,StartColor);    //1st color to read!
  for(short i=0; i<NumOfColors*3; i++ )
     {Palette[i]=inp(0x03C9);
     }
}
void Video::Get_DAC_C(uchar Color,uchar& Red,uchar& Green,uchar& Blue)
{ outp(0x03c6,0xff);
  outp(0x03c7,Color);
  Red=  inp(0x03c9);
  Green=inp(0x03c9);
  Blue= inp(0x03c9);
}
```

Notice that there are very few changes from the set of writing functions to convert them into reading functions. Instead of sending our start color to 0x03c8 (DAC Write Port), we're sending it to 0x03c7 (DAC Read Port). From there we can do inports from 0x03c9 (DAC Data Port) again!

BIOS Functions for Reading

```
#define uchar unsigned char

void Video::Get_DAC(uchar Register,uchar& Red,uchar& Green,uchar& Blue)
{ union REGS regs;
  regs.h.ah=0x10;
  regs.h.al=0x15;
  regs.x.bx=Register;
  int86(0x10,&regs,&regs);
  Red=   regs.h.dh;
  Green= regs.h.ch;
  Blue=  regs.h.cl;
}
void Video::Load_DAC_Palette(uchar StartColor,uchar NumberOfColors, uchar* Palette)
{ REGPACK regs;
  regs.r_ax=0x1017;
  regs.r_bx=StartColor;
  regs.r_cx=NumberOfColors;
  regs.r_es=FP_SEG(Palette);
  regs.r_dx=FP_OFF(Palette);
  intr(0x10,&regs);
}
```

The first can be used to load the R,G,B values of a particular color, while the second is for loading the whole palette into a buffer of unsigned characters.

*NOTE: As in the first section, the second BIOS function will not work under DJGPP because it doesn't use the SEG:OFF scheme for addressing memory. If you have any questions, comments or requests, please give me some Feedbac*k!

Many many thanks to Nutty for helping me learn the ways of DJGPP inline assembly, and the great suggestions! His site is listed under my Links section.

Contact Information

I just wanted to mention that everything here is copyrighted, feel free to distribute this document to anyone you want, just don't modify it! You can get a hold of me through my website or direct email. Please feel free to email me about anything. I can't guarantee that I'll be of ANY help, but I'll sure give it a try :-)

Email : deltener@mindtremors.com
Webpage : http://www.invsereality.org