VESA Video Modes

Intro

I really wish setting VESA resolutions was as easy as setting a standard video mode. Unfortunately, reality is against us! Back in the old days of Video Cards there was a lot of technological advancements happening. Each big company had its own idea of what abilities should be in the card, and what should be the correct process in making it easy to utilize and program. What turned out was a programmers nightmare! Each company ended up selling a card that was completely incompatible with its competition. FINALLY in 1989, VESA was born. They are the Video Electronic Standard Association. We will be dealing specifically with VESA VBE or Video Bios Extensions. This allows each card to have its own unique set of capabilities, but the VBE creates another level of abstraction between the programmer and the hardware. This creates a nice interface for the programmer which can use the VBE to get information about whatever card is available.

Detecting VESA Availability

First off, lets figure out which (if any) VESA version is available. To do this, we will create a structure to hold some card information.

```
typedef struct VBE_VgaInfo
{ char VESASignature[4];
  short VESAVersion;
  char *OemStringPtr;
  ulong Capabilities;
  ulong VideoModePtr;
  short TotalMemory;
  //VBE 2.0
  short OemSoftwareRev;
  char *OemVendorNamePtr;
  char *OemProductNamePtr;
  char *OemProductRevPtr;
  char reserved[222];
  char OemData[256];
};
```

This is the structure used to hold information about the VESA Graphics information. To start off we will only be using the VESASignature,Version and TotalMemory variables. Later on as we make our code more detailed we can decide wether to support the other structure members. Lets create a function that determines if VESA is available and if so, what version!

```
VBE_VgaInfo vesainfo;
long VESAVerMaj,VESAVerMin;

int Video::Detect_VESA()
{__dpmi_regs r;
 strncpy(vesainfo.VESASignature,"VESA",4); //or VBE2
 r.x.ax=0x4F00;
 r.x.di=__tb&0x0f;
 r.x.es=(__tb >>4)&0xFFFF;
 dosmemput(&vesainfo,sizeof(vesainfo),__tb);
 __dpmi_int(0x10,&r);
 dosmemget(__tb,sizeof(vesainfo),&vesainfo);
 if(r.h.ah||strncmp("VESA",vesainfo.VESASignature,4)||vesainfo.VESAVersion <0x100)
  { return 0; //VESA NOT present
  }
 VESAVerMaj=vesainfo.VESAVersion >>8;
 VESAVerMin=vesainfo.VESAVersion & 0x0f;
 if(VESAVerMaj > 2.0)
  {GetPMInterface();
   Bankptr = &Video::SetBankAlt;
  }
 else
  {Bankptr = &Video::SetBankBIOS;
  }
 return 1;
}
```

This code is to be used in DJGPP, but the same principle applies to normal C++. We use the bios function 0x4F00. We will set ax=4F00 and set ES:DI to a far pointer to our little structure we have created. When we call interrupt 0x10 (video interrupt), the result will be located in ax. If ax=4F00 after the call, then VESA functions are supported, any other number will be interpreted as no. Our structure will be filled in after the call. Notice that we are filling in vesainfo.VESASignature with "VESA" before calling the interrupt. This tells the BIOS that we are requesting VESA information, VESA 2.0 items WILL NOT BE FILLED IN! The new standard is to fill in the signature with "VBE2" so that all VESA 2+ information will be filled in! Also notice that we are setting a function pointer Bankptr. This is used in the MoveMemVESABanked function. If we have access to the protected mode information (VESA 2.+) then we can use a different function for bank switching that doesn't need to go through the BIOS. That function is SetBankAlt, and the much slower one is SetBankBIOS.

```
void Video::PrintVESAInfo()
{ char buffer[255];
  short modes[255];
  long temp;
  short t;

  cout<<"VESA "<<VESAVerMaj<<"."<<VESAVerMin<<endl;
  cout<<"Video Memory: "<<dec<<(long)(vesainfo.TotalMemory*64)<<"k"<<endl;
```

```
  temp=vesainfo.OemStringPtr;
  t = __dpmi_segment_to_descriptor((temp>>16));
 movedata(t,(temp&0xFFFF),_my_ds(),(int)&buffer,255);
  cout<<buffer<<endl;

  temp=vesainfo.OemVendorNamePtr;
  t = __dpmi_segment_to_descriptor(temp>>16);
 movedata(t,(temp&0xFFFF),_my_ds(),(int)&buffer,255);
  cout<<buffer<<endl;

  temp=vesainfo.OemProductNamePtr;
  t = __dpmi_segment_to_descriptor(temp>>16);
 movedata(t,(temp&0xFFFF),_my_ds(),(int)&buffer,255);
  cout<<buffer<<endl;

  temp=vesainfo.OemProductRevPtr;
  t = __dpmi_segment_to_descriptor(temp>>16);
 movedata(t,(temp&0xFFFF),_my_ds(),(int)&buffer,255);
  cout<<buffer<<endl;
}
```

This function prints out all the extended OEM VESA information that is available when you fill in the signature with VBE2.

Getting Supported Modes

Now that we know a little about the card, we will create a function that fetches VESA video mode information. This function in order to be truly useful, must request information from the system about the video mode we have switched to. Remember that VESA modes are larger than 65536 bytes and will require bank switching to update the screen or use the linear frame buffer. To do this properly we must know the some screen information, foremost being the granularity of memory. This structure is as follows:

```
 typedef struct VBE_ModeInfo
 {short ModeAttributes;
  char WinAAttributes;
  char WinBAttributes;
  short WinGranularity;
  short WinSize;
  ushort WinASegment;
  ushort WinBSegment;
  void *WinFuncPtr;
  short BytesPerScanLine;
  short XRes;
  short YRes;
```

```
 char XCharSize;
 char YCharSize;
 char NumberOfPlanes;
 char BitsPerPixel;
 char NumberOfBanks;
 char MemoryModel;
 char BankSize;
 char NumberOfImagePages;
 char res1;
 char RedMaskSize;
 char RedFieldPosition;
 char GreenMaskSize;
 char GreenFieldPosition;
 char BlueMaskSize;
 char BlueFieldPosition;
 char RsvedMaskSize;
 char RsvedFieldPosition;

 //VBE 2.0
 ulong PhysBasePtr;
 ulong OffScreenMemOffset;
 short OffScreenMemSize;

 //VBE 2.1
 short LinbytesPerScanLine;
 char BankNumberOfImagePages;
 char LinNumberOfImagePages;
 char LinRedMaskSize;
 char LinRedFieldPosition;
 char LingreenMaskSize;
 char LinGreenFieldPosition;
 char LinBlueMaskSize;
 char LinBlueFieldPosition;
 char LinRsvdMaskSize;
 char LinRsvdFieldPosition;
 char res2[194];
};

VBE_ModeInfo modeinfo;
```

Pretty darn huge isn't it! Now that we know this structure, here is a function that utilizes it. We are going to be using this function in conjunction with another one to fetch all available video modes supplied by the VESA, then save mode information to a dynamically allocated structure so we only have to bother the VESA one time per mode when we start up.

```
int Video::GetVESAModeInfo(short mode)
{__dpmi_regs r;
 r.x.ax=0x4F01;
 r.x.cx=mode;
 r.x.di=__tb & 0x0F;
 r.x.es=(__tb >>4)&0xFFFF;
  __dpmi_int(0x10,&r);
  dosmemget(__tb,sizeof(modeinfo),&modeinfo);
  if(r.h.ah)
   {return 0; //failure!
   }
  return 1;
}
```

This little section is programmed for DJGPP. It uses a temporary buffer that is already created called tb that is located in lower memory. All we need to know is that our huge mode structure will be filled in correctly after calling this routine. What we really need this function for is to know the granularity of the video mode we specified. Think of granularity as how large the memory chunks are. We need to fetch this vital mode information in order for our memory moving routines to work correctly. Here's the function that utilizes the above function and structure to get all available VESA modes according to the VESA supplier.

```
void Video::GetAvailableVESAModes()
{ long temp;
  short t,*m,count=0;
  temp=vesainfo.VideoModePtr;
  m=(short *)MK_FP(temp>>16,temp&0xFFFF);
  while(m[count] != -1)
   { count++;
   }
  NumberOfModes = count;
  cout<<NumberOfModes<<" Video Modes Available\n";
  Modes = new short[NumberOfModes];
  if(Modes ==NULL)
   { cout<<"Error allocating Mode Number Listing!\n";
   }
  for(temp=0;temp<NumberOfModes;temp++)
   { Modes[temp]=m[temp];
   }
  ModeList = new VBE_ModeInfo[NumberOfModes];
  if(ModeList == NULL)
  { cout<<"Error allocating ModeList!\n";  }
  for(count=0;count<NumberOfModes;count++)
  {GetVESAModeInfo(Modes[count]);
   memmove(&ModeList[count],&modeinfo,sizeof(VBE_ModeInfo));
  }
}
```

```
void * Video::MK_FP(unsigned short seg, unsigned short ofs)
 {if(!(_crt0_startup_flags & _CRT0_FLAG_NEARPTR))
   if(!__djgpp_nearptr_enable())
         return (void*)0;
   return (void *)(seg*16+ofs+__djgpp_conventional_base);
 }
```

In order for this function to work, you must have Modes declared as a short pointer. This function is a must for anyone who wants to support as many modes as the video card can handle. VESA no longer requires manufacturers to follow any guidelines for video mode numbers, so VESA came up with a way to give the programmer a list of available video modes. We fetch the video modes by setting our pointer to the one given to us by the VESA, save them in a list (Modes) with the list terminating with -1, and then get video mode information for each one. This is only done one time during the program, so it is really nice when you need mode information quick, it'll be there waiting for you! Following is the protected mode version of the Make File Pointer function used with standard Real-Mode C.

Setting VESA Modes

Without any further delay, lets go into the function that actually changes the Video Mode. As you can see, we have a lot more options to keep track of compared to normal VGA resolutions!

```
unsigned short Video::GetModeIndex(short Xres,short Yres,short Depth)
{ unsigned short temp=666;
  for(short count=0;count<NumberOfModes;count++)
  { if((ModeList[count].XRes == Xres) &&
       (ModeList[count].YRes == Yres) &&
       (ModeList[count].BitsPerPixel == Depth))
      { temp=count;
      }
  }
return temp;
}

void Video::VESAMode(short index)
{ __dpmi_regs r;
  memset(&r,0,sizeof(__dpmi_regs));
  r.x.ax=0x4F02;
  r.x.bx=Modes[index];
  UsingLinearFrameBuffer(index);

 if(LinearFrameBuffer)
  {r.x.bx+=0x4000; //bit 14 0100 0000 0000 0000
  }
 if(!ClearMem)
   {r.x.bx+=0x8000; //bit 15 1000 0000 0000 0000
   }
```

```
 __dpmi_int(0x10,&r);

 if(r.x.ax !=0x004f)
   {VideoMode(80,25,8);
    cout<<"Failure to set VESA video mode "<<hex<<Modes[index]<<endl;
    sleep(4);
    RevertToSavedModeData();
    VideoMode(Screen_Width,Screen_Height,Screen_BitsPerPixel);
   }
  VideoModeIndex=index;
  CurrentVideoMode=Modes[VideoModeIndex];
  Screen_Width  = ModeList[VideoModeIndex].XRes;
  Screen_Height = ModeList[VideoModeIndex].YRes;
  Screen_Size   = Screen_Width*Screen_Height;
  XCenter = (Screen_Width>>1);
  YCenter = (Screen_Height>>1);
  //Only used when no LFB is present
  NumberOfBanks=Screen_Size>>16;
  MemAfterBanks=Screen_Size&65535;
  if(LinearFrameBuffer)
   {MapLFB(index);
    Move2Vid=&Video::MoveMemStandard;
   }
 else
  {Move2Vid = &Video::MoveMemVESABanked;
  }
  SaveModeData(); //success so save!
  cout<<"Current mode = "<<CurrentVideoMode<<endl;
}
```

Our VESAMode function changed quite a bit since our last version of this tutorial for those that read it. Notice that this function is also getting passed an index number. Again, this index is the index into our supported video mode listing (Modes). All of our saved video mode information is saved under the ModeList list. Ok, from the top we start setting up our registers for our interrupt, setting ax to 0x4f02, bx to our mode number, we then test to see if that mode supports a Linear Frame Buffer by calling UsingLinearFrameBuffer. That function sets the class variable LinearFrameBuffer. It is supported if it is set to one, and no if it is set to 0. We then turn on bit 14 accordingly. We can also choose if we want VGA ram erased or saved after the modeswitch by setting bit 15 and using the class variable ClearMem. We then generate our interrupt using __dpmi_int. We then check to see if we were successful in switching modes. If we weren't we switch to text mode and spit out an error message, sleep 4 seconds, then switch to our previously saved video mode. Do whatever you like in the instance a switch mode fails. We then set a bunch of screen class variables. If we are in a mode that supports a LFB, we then call MapLFB which makes the Linear Frame Buffer accessible to us. We also set our Move2Vid function pointer to one of two functions. MoveMemStandard is used when we have access to a LFB, and MoveMemVESABanked is used when we must do bank switching. When it is all said and done, we call SaveModeData so if the next switch fails, we can revert to this previous mode! Here's a quick listing of what some of those other functions are:

```
void Video::UsingLinearFrameBuffer(short index)
{ if(ModeList[index].ModeAttributes & 0x80)
    {LinearFrameBuffer = 1;
     cout<<hex<<Modes[index]<<"h uses a LFB"<<dec<<endl;
    }
  else
   {LinearFrameBuffer = 0;
   }
}

void Video::SaveModeData()
{savedmode = CurrentVideoMode;
 savedxres = Screen_Width;
 savedyres = Screen_Height;
 saveddepth= Screen_BitsPerPixel;
 savedindex= VideoModeIndex;
}

void Video::RevertToSavedModeData()
{CurrentVideoMode   = savedmode;
 Screen_Width       = savedxres;
 Screen_Height      = savedyres;
 Screen_BitsPerPixel= saveddepth;
 VideoModeIndex     = savedindex;
}
```

These really aren't that hard to decypher. UsingLinearFrameBuffer tests the ModeAttributes attribute in the modeinfo structure for the video mode, testing to see if the correct bit is set, setting LinearFrameBuffer accordingly. SaveModeData and RevertToSavedModeData I HOPE are self explaining :) Now onto using that darn Linear Frame Buffer!!!

Using the Linear Frame Buffer

The Linear Frame Buffer was made available in VESA 2.0. Prior, the VESA Video RAM was larger than 65536 bytes, so in order to access Video RAM that was larger, special proceedures were used. Bank switching let the programmer have access to all of Video RAM. The big problem was that bank switching was really slow! A Linear Frame Buffer allows the programmer to treat Video RAM as one continuous array, just like in good old 320x200!! It is not only easier, but faster!

In Class constructor

```
 my_video_ds=__dpmi_allocate_ldt_descriptors(1);
 __dpmi_set_segment_base_address(my_video_ds,0xA000+__djgpp_conventional_base);
 __dpmi_set_segment_limit(my_video_ds,2000000);
```

```
void Video::MapLFB(short index)
{ __dpmi_meminfo info;
  info.size=vesainfo.TotalMemory<<16;
  cout<<“Mapping “<<info.size<<“ bytes for Linear Frame Buffer”<<endl;

  info.address=(unsigned long)ModeList[index].PhysBasePtr;
  cout<<“Pointer to LFB = “<<hex<<ModeList[index].PhysBasePtr<<“h”<<dec<<endl;
  if(__dpmi_physical_address_mapping(&info)!=0)
    { cout<<“Can’t map Linear Frame Buffer!\n”;
    }

 __dpmi_free_ldt_descriptor(my_video_ds); //allocated in constructor so erase
 my_video_ds=__dpmi_allocate_ldt_descriptors(1);
 __dpmi_set_segment_base_address(my_video_ds,info.address);
 __dpmi_set_segment_limit(my_video_ds,info.size-1);
 cout<<“Base address = “<<hex<<info.address<<dec<<endl;

 video_screen=(unsigned char *)((unsigned long)info.address+__djgpp_conventional_base);
}
```

Before we even think about using the Linear Frame Buffer, we have to map it into our addressable space. We accomplish this by getting the pointer to it from the modeinfo structure for the video mode we want to use. We don’t actually have to be in the mode for this function, just have our modeinfo list filled in correctly. We use index as an index (how genius) into our list. Remember that when we do switch to the video mode, before or after this function, we have to set the LFB bit in the mode attribute. This should be taken care of automatically if we used our cool VESAMode function! After this function exits, it reassigns our video_screen pointer to point to the Linear Frame Buffer in our addressable region! Now we can treat it as normal VGA ram with NO limitations!! The last portion of the code that uses my_video_ds is for those that don’t like to directly access video ram as if it was a linear array and use farpoke or farpeek. my_video_ds provides access to video ram for those functions.

Using the Protected Mode Information

```
typedef struct PM_Int
{ unsigned short SetWindow Packed;
  unsigned short SetDisplayStart Packed;
  unsigned short SetPalette Packed;
  unsigned short IOPrivInfo Packed;
};

 void (*pm_bank)(char);
 void (*pm_setstart)(char);
 void (*pm_setpalette)(char);
 PM_Int* pm_info;
```

```
void Video::GetPMInterface()
{__dpmi_regs regs;
 regs.x.ax=0x4F0A;
 regs.x.bx=0;
 __dpmi_int(0x10,&regs);
 if(regs.h.ah)
  { cout<<"\nError gaining access to VESA Protected Mode Interface!\n";
  }
 if(pm_info != NULL)
   { free(pm_info);
   }
 pm_info = (PM_Int *)malloc(regs.x.cx);
 _go32_dpmi_lock_data(pm_info,regs.x.cx);
 dosmemget((regs.x.es*16)+regs.x.di,regs.x.cx,pm_info);
 pm_bank     = (void*)((char*)pm_info+pm_info->SetWindow);
 pm_setstart = (void*)((char*)pm_info+pm_info->SetDisplayStart);
 pm_setpalette=(void*)((char*)pm_info+pm_info->SetPalette);

 return;
}
```

Using BIOS functions are notoriously slow, so the VESA BIOS provides us with some faster means of getting stuff done. Probably the most important is the pm_bank switch function that can switch VESA video banks without having to go through the BIOS! The structure that is filled in by the BIOS or actually pointed to by the BIOS actually tells us how far from that structure in memory each function resides, hence the wierdo routine for setting the function pointers!

There we have it. For more information on accessing a specific bank or moving memory to Video Ram using an Offscreen Buffer please take a look here. If you have any questions, comments, rude remarks please give me some feedback!

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 : Justin Deltener <deltener@mindtremors.com>
Webpage : http://www.inversereality.org