Drawing In Protected Mode

From OSDev Wiki

Now that you know how you can easily write text to the screen using Hardware VGA support, you might be wondering how you'll be able to display nice images, windows, menus, icons, fancy cursors and buttons, etc.

Well, to quote Curufir, "Switch to a graphical mode and write directly in video memory".

Contents

Graphics Modes

Main article: Getting VBE Mode Info

Well the VGA (And VESA) modes can be selected using the standard BIOS interrupt 0x10. Int 0x10 (http://www.ctyme.com/intr/int-10.htm) seems like a decent enough reference for int 0x10 (No VESA extension) while VESA (http://www.vesa.org/) contains the various VESA standards.

Vga is limited to a 640x480x16, VESA (Depending on your card) can present much higher resolutions.

Switching

The cleanest way to set up your video mode is to go through the video BIOS. It can be performed through the regular `Int 0x10` interface, or through the (optional) Protected mode interface offered by VBE3. As you can guess, `Int 0x10` requires a 16-bit environment, so you can only use it in Real Mode or Virtual 8086 Mode

Practically, the options are (in order of difficulty):

- You set up the mode you want at early stage (in the bootloader) before entering protected mode.
  - You let GRUB do the switch for you. (Currently only works with patched GRUB (http://www.smksoftware.co.za/software/vbe-grub/) , or GRUB 2

(http://www.gnu.org/software/grub/grub-2.en.html) . You may download the patches directly from here (http://www.smksoftware.co.za/cgi-bin/cgit/vbe_grub.git/tree) .)

- You switch back to Real Mode or Unreal Mode for setting the proper video mode (Napalm at rohitab.com (http://www.rohitab.com/discuss/topic/35103-switch-between-real-mode-and-protected-mode/) has a neat little function for reference.)
- You write a VGA driver that can do low-resolution modes on practically all hardware
- You use the PMID from VBE3, if present
- You set up a V8086 monitor that will execute the mode-switching code
- You run some software code translation tool to produce pmode code out of bios rmode code. (SANiK is on the catch (http://www.osdev.org/phpBB2/viewtopic.php?t=10321))
- You write a driver for your specific graphics card

Locating Video Memory

For standard VGA video modes the video memory will either be at address `0xA0000` or `0xB8000`. To find out which one look at the following table (quoting http://www.uv.tietgen.dk/staff/mlha/PC/Prog/ASM/INT/INT10.htm): "text" means `0xB8000`, CGA graphics modes are also at `0xB8000`, and VGA/EGA is at `0xA0000`. Note that most EGA modes (and high res VGA modes) use several bit planes so you won't be able to use all the colors by simply writing to video memory.

00	text 40*25 16 color (mono)
01	text 40*25 16 color
02	text 80*25 16 color (mono)
03	text 80*25 16 color
04	CGA 320*200 4 color
05	CGA 320*200 4 color (m)
06	CGA 640*200 2 color
07	MDA monochrome text 80*25
08	PCjr
09	PCjr
0A	PCjr
0B	*reserved*
0C	*reserved*
0D	EGA 320*200 16 color
0E	EGA 640*200 16 color
0F	EGA 640*350 mono
10	EGA 640*350 16 color
11	VGA 640*480 mono
12	VGA 640*480 16 color
13	VGA 320*200 256 color

For VESA modes, the framebuffer address is stored in the mode info block. This is the **physical** address of the linear framebuffer (it's not a 16-bit far pointer but a 32-bit linear pointer) : if you use paging, you have to map it somewhere to use it.

Plotting Pixels

Location

Let's say you want to plot a pixel in red in the middle of your screen. The first thing you have to know is **where** the middle of the screen is. In 320x200x8 (mode 13), this will be at 100x320+160 = 32160. In general, your screen can be described by:

width	how many pixels you have on a horizontal line
height	how many horizontal lines of pixels are present
pitch	how many *bytes* of VRAM you should skip to go one pixel down
depth	how many *bits* of color you have
"pixelwidth"	how many bytes of VRAM you should skip to go one pixel right.

"pitch" and "width" may seem redundant at first sight but they aren't. It's not rare once you go to higher (and exotic) resolutions to have e.g. 8K bytes per line while your screen is actually 1500 pixels wide (32-bits per pixel). The good news is that it allows smooth horizontal scrolling (which is mainly useful for 2D games :P)

Pitch and pixel width are usually announced by VESA mode info. Once you know them, you can calculate the place where you plot your pixel as:

```
unsigned char *pixel = vram + y*pitch + x*pixelwidth;
```

Color

The second thing to know is what value you should write for "red". This depends on your screen setup, again. In EGA mode, you have a fixed palette featuring dark-red (color 4) and light-red (color 12). Yet, EGA requires you to plot each bit of that on different pixel plane, so refer to EGA programming tutorials if you **really** want such modes supported. In conventional 320x200x8 VGA mode, you have the same colours 4 and 12 as in EGA so you would plot your red pixel with

```
*pixel = 4;
```

Yet, in VGA, the palette is reprogrammable (as you can learn in FreeVGA documents), so virtually any value between 0..255 could be 'red' if you program the palette so :P

Finally, in VESA modes, you usually have truecolor or hicolor, and in both of them, you have to give independent red, green and blue values for each pixel. modeinfo will (again) instruct you of how the RGB components are organized in the pixel bits. E.g. you will have `xRRRRRGGGGGBBBBB` for 15-bits mode, meaning that #ff0000 red is there `0x7800`, and #808080 grey is `0x4210` (pickup pencil, draw the bits and see by yourself)

```
/* only valid for 800x600x16M */
static void putpixel(unsigned char* screen, int x,int y, int color) {
    unsigned where = x*3 + y*2400;
    screen[where] = color & 255;              // BLUE
    screen[where + 1] = (color >> 8) & 255;   // GREEN
    screen[where + 2] = (color >> 16) & 255;  // RED
}

/* only valid for 800x600x32bpp */
static void putpixel(unsigned char* screen, int x,int y, int color) {
    unsigned where = x*4 + y*3200;
    screen[where] = color & 255;              // BLUE
    screen[where + 1] = (color >> 8) & 255;   // GREEN
    screen[where + 2] = (color >> 16) & 255;  // RED
}
```

Optimizations

It can be tempting from here to write fill_rect, draw_hline, draw_vline, etc. from calls to putpixel ... don't. Drawing a filled rectangle means you access successive pixels and then advance by "pitch - rect_width" to fill the next line. If you do a "for(y=100;y<200;y++) for(x=100;x<200;x++) putpixel (screen,x,y,RED);" loop, you'll recompute 'where' about 10,000 times. Even if the compiler has done good job to translate y*3200 into adds and shifts instead of multiplication, it's silly to run that so much time while you could do

```
static void fillrect(unsigned char *vram, unsigned char r, unsigned char
    unsigned char *where = vram;
    int i, j;

    for (i = 0; i < w; i++) {
        for (j = 0; j < h; j++) {
            //putpixel(vram, 64 + j, 64 + i, (r << 16) + (g << 8) + b);
            where[j*4] = r;
            where[j*4 + 1] = g;
            where[j*4 + 2] = b;
        }
        where+=3200;
    }
}
```

That should be enough to get you started coding (or googling for) a decent video library.

Drawing Text

Once in graphic mode, you no longer have the BIOS or the hardware to draw fonts for you. The basic idea is to have font data for each character and use it to plot (or not to plot) pixels. There are plenty of ways to store those fonts depending on whether they have multiple colors or not, alpha channel or not etc. What you will basically have, however is:

```
// holding what you need for every character of the set
font_char* font_data[CHARS];

// rendering one of the character, given its font_data
draw_char(screen, where, font_char*);

draw_string(screen, where, char* input) {
    while(*input) {
        draw_char(screen,where,font_data[input]);
        where += char_width;
        input++;
    }
}

draw_char(screen, where, font_char*) {
    for (l = 0; l < 8; l++) {
        for (i = 8; i > 0; i--) {
            j++;
            if ((font_char[l] & (1 << i))) {
                c = c1;
                put_pixel(j, h, c);
            }
        }
        h++;
        j = x;
    }
}
```

Font Encoding

The most common encoding that allows you not to overwrite the background over which you draw your text is the *font bitmap*, that is, an "A" character will e.g. be encoded as

```
...XX... = 0*128+0*64+0*32+1*16+1*8+0*4+0*2+0*1 = 0x18
..XXXX.. = 0x3C
.XX..XX. = 0x66
.XXXXXX. = 0x7E
.XX..XX. = 0x66
.XX..XX. = 0x66
........ = 0x00
........ = 0x00
```

In which case you test each bit of the font data to tell whether it's 1 or 0 and only put the pixel if it's 1. For larger fonts you might want to use RLE encoding instead, for instance. Finally, state-of-the-art true-type fonts will require you to support the "freetype" library.

Optimizations

Use of a "put_pixel()" function is almost always a performance problem. For an 8 x 8 character you can find out the address of the first (top left) pixel and increment it to get the address of the next pixel in the row, and add the number of bytes per line to get the address of the next row. This is much faster than calculating "address = video_address + y * horizontal_resolution + x" 64 times (per character).

However, it's even faster to work on more than one pixel at a time. If you look at the font encoding above you'll notice that there's an 8-bit number for each row of the character. This 8-bit number can be used as the index in a lookup table containing masks. For example, for 8 bits per pixel you could do 8 pixels at a time, like this:

```
uint32_t font_data_lookup_table[16] = {
    0x00000000,
    0x000000FF,
    0x0000FF00,
    0x0000FFFF,
    0x00FF0000,
    0x00FF00FF,
    0x00FFFF00,
    0x00FFFFFF,
    0xFF000000,
    0xFF0000FF,
    0xFF00FF00,
    0xFF00FFFF,
    0xFFFF0000,
    0xFFFF00FF,
    0xFFFFFF00,
    0xFFFFFFFF
}

draw_char(uint8_t *where, uint32_t character, uint8_t foreground_colour,
    int row;
    uint8_t row_data;
    uint32_t mask1, mask2;
    uint8_t *font_data_for_char = &system_font_data_address[character * 8
    uint32_t packed_foreground = (foreground << 24) | (foreground << 16)
    uint32_t packed_background = (background << 24) | (background << 16)

    for (row = 0; row < 8; row++) {
        row_data = font_data_for_char[row];
        mask1 = font_data_lookup_table[row_data >> 16];
        mask2 = font_data_lookup_table[row_data & 0x0F];
        *(uint32_t *)where = (packed_foreground & mask1) | (packed_backgr
        *(uint32_t *)(&where[4]) = (packed_foreground & mask2) | (packed_
        where += bytes_per_line;
    }
}
```

See Also

- Extra Notes
- Covers basically the same, and is ASM-oriented. (http://bos.asmhackers.net/forum/viewtopic.php?id=65)

Retrieved from "http://wiki.osdev.org/index.php?title=Drawing_In_Protected_Mode&oldid=17193"
Categories: Video | Graphical UI

- This page was last modified on 1 December 2014, at 15:03.
- This page has been accessed 80,351 times.