Real Mode Hardware: i386

Overview

The x86 line of processors, prior to implementation of UEFI, booted in ***real mode*** – a 16-bit backwards-compatible environment. (Modern x64 processors continue to support BIOS and real mode at boot for older operating systems.) In real mode, while an operating system / resident monitor is present, programs often interact directly with hardware, or communicate with the BIOS and/or "terminate-and-stay-resident" (***TSR***) programs to communicate with hardware. A TSR resides in memory but is dormant unless invoked by a hardware interrupt or a request from a currently running program. These requests are often referred to as "software interrupts".

Programs can directly interact with hardware by communicating with them through the memory bus, when mapped into memory, or through the data bus. Alternatively, programs can make use of TSR drivers already loaded into memory. This document will cover a few relevant examples of memory and data bus communication directly with hardware devices as well as TSR-driver-based device polling. It includes specific examples for output devices (VGA controller) and input devices (mouse-style pointers).

"Software Interrupt" Handlers

On x86 systems, software can make requests of other resident programs via the INT instruction. These request handlers typically fall into three main categories. Some handlers are provided by standards-compliant systems via BIOS routines. An additional set are traditionally offered through the operating system or resident monitor. For example, INT 0x80 is the Unix System Call routine. Finally, a third group of handlers are loaded into memory at or after boot to provide additional services (TSRs). This is typical of some driver routines, such as mouse driver TSRs.

These requests include at a minimum a handler number. They may also include a function number, parameters, and return registers. For example, this request would fetch the keyboard country code:

INT Example	**Value**	**Meaning**
Handler	0x16	Keyboard
Function	AH = 0x50	Country Code
Parameter(s)	AL = 0x01	Get Current Code
Return Register(s)	AL = Status (00 = success, 02 = error) BX = Country code	

You will not need to use this handler; it is just an example. 😊

To execute this handler, we would execute the following instructions:

```
mov ah, 0x50        // Set function
mov al, 0x01        // Set Parameter(s)
int 0x16            // Invoke handler
```

Intel asm syntax is different because <u>OF COURSE IT IS</u>...

In this example, when the routine returns, we can fetch the success/failure status from register AL and the country code from register BX.

Video Graphics Adapter (VGA)

A standards-compliant video graphics adapter (VGA) device makes use of both the memory bus – for pixel data – as well as the data bus for setting up colors in indexed modes.

Video Mode

Changing the video mode, though standardized, is complex. While direct hardware access is possible the x86 system BIOS provides routines for several tasks, including setting video mode.

Set Video Mode	**Value**	**Meaning**
Handler	0x10	Video
Function	AH = 0x00	Set Mode
Parameter(s)	AL = *Mode*	New Mode
Return(s)	AL = Status (BIOS dependent)	

Mode	**Description**
0x00	Text: 40x25 (Low-Res)
0x03	Text: 80x25 (High-Res)
0x12	Graphics: 640x480 (16 Colors)
0x13	Graphics: 320x200 (256 Colors)

Setting the Palette

The VGA standard – along with many before it and many embedded systems – used ***indexed*** formats for pixel representation. Rather than directly storing the RGB values for each pixel, the pixel data was an index into a ***color palette***. This allows each pixel to be represented by fewer bits (thus requiring less video memory). The VGA/MCGA standards uniquely allowed the palette to be set (unlike earlier standards using fixed palettes). The number of colors in the palette was limited by the number of bits per pixel. Consider a typical 320x200 screen with varied bit widths:

Pixel Bits	**Colors**	**Memory**	**Hardware**
1	2	7.8125 KiB	Monochrome
2	4	15.625 KiB	CGA
4	16	31.25 KiB	EGA
8	256	64 KiB (62.5 KiB Raster + 1.5KiB Palette)	VGA/MCGA
16	65536	125 KiB	SuperVGA (High-color)
24	~16M	250 KiB	SuperVGA (True-color)

By using a 24-bit palette (3 bytes per color) and 8-bit indexing (256 colors), VGA can display rich and colorful scenes with just 64 KiB of video memory. Thus, the VGA standard has had continued use in various applications long after its heyday in the PC era of the 1990s.

Program can set up palette colors via the data bus as follows:

1) Set the palette mask to allow all colors by writing to the ***Palette Mask*** register.
2) Alert video adapter to color number we wish to set via the ***Color Index*** register.
3) Successively write red, then green, then blue values to the ***RGB Data*** register.
4) Repeat (**2**) & (**3**) for each color index.

Example: Set Color 0 to Bright-Yellow (ASM)

```
mov dx, 0x03C6  // Load mask port
mov al, 0xFF    // Don't mask anything
out dx, al      // Write mask to port

mov dx, 0x03C8  // Load color index port
mov al, 0x00    // Set color # to 0
out dx, al      // Write color # to port

mov dx, 0x03C9  // Load RGB data port
mov al, 0x3F    // Set value to 63 (max)
out dx, al      // Write value as RED
out dx, al      // Write value as GREEN
mov al, 0x00    // Set value to 0 (min)
out dx, al      // Write value as BLUE
```

Data Port	**Write Register**	**Range**
`0x03C6`	Palette Mask	`0x00-0xFF`
`0x03C7`	Read Color Index	`0x00-0xFF`
`0x03C8`	Write Color Index	`0x00-0xFF`
`0x03C9`	RGB Data	`0x00-0x3F`

The palette can also be read from the adapter using port `0x3C7`.

Vertical Syncing

When writing to video memory, it is usually preferable to begin writing just after a new vertical sync has started. This is especially important on older CRT displays but can still impact modern LCD / LED displays. To do this, the origran will need to read VGA's **Input Status Register 1**.

VGA Input Status Register 1								
Data Bus Input Port	**Register Bits (Flags)**							
	`0x00`	`0x01`	`0x04`	`0x08`	`0x10`	`0x20`	`0x40`	`0x80`
`0x03DA`	Display Disabled	Reserved		V. Retrace Active	Reserved			

To find the beginning of the retrace, the program must wait for the current retrace to end, then for the next one to begin – i.e., wait for the retrace bit to clear, and then for it to be set:

```
  mov dx, 0x3DA // Store port in DX
wait_end:       // Loop: wait for retrace end
  in al, dx     // Read status register into AL
  and al, 0x08  // Check the retrace bit
  jnz wait_end  // If active, loop

wait_start:     // Loop: wait for retrace start
  in al, dx     // Read status register into AL
  and al, 0x08  // Check the retrace bit
  jz wait_start // If inactive, loop
```

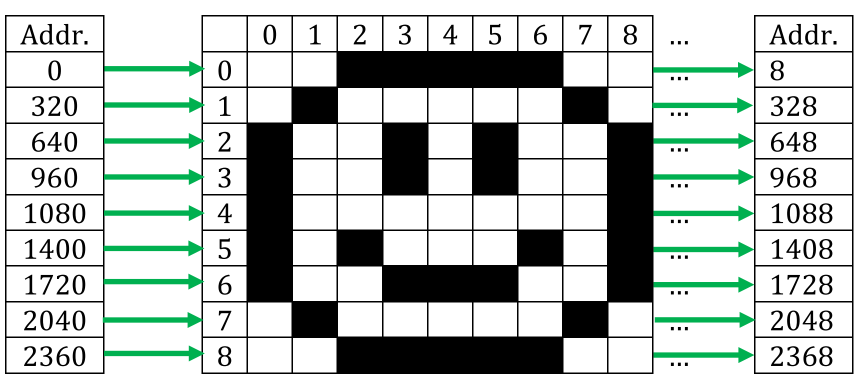
```
// C/C++ version

#include <conio.h>

void waitForRetrace()
{
    while (inp(0x3DA) & 0x08);
    while (!inp(0x3DA) & 0x08);
    writePixelsToVRAM();
}
```

Writing Pixels

Once we are ready to write the pixels, we will need to write to the real mode video memory address range - **`0x0A0000:0x0BFFFF`**. This does not require calling the BIOS or working on the data bus – ***video memory is just memory***! (To make the pointers, see the Real Mode Tools & Techniques document.) Unlike many video modes, pixels in indexed palette mode (0x13) are not interleaved. Additionally, video is stored as one byte per pixel – the color number, which is an index into the palette. These pixels are in order, row-by-row – starting at the first row (y = 0) and going across the columns (from x=0 to x=319). This means a program can easily calculate the location of a pixel's data and fetch or write to it quickly and with fewer CPU cycles. It also makes storing backups of the data – and quickly writing it – feasible.



Video Memory																														
Row 0										*Row 1*										*Row 2*										…
1	1	0	0	0	0	0	1	1	…	1	0	1	1	1	1	1	0	1	…	0	1	1	0	1	0	1	1	0	…	…

Mouse-Style Pointers

Mouse-type devices return two primary pieces of data – the ***button states*** (pressed and released) and ***movement***. Mouse movement is measured in **mickeys** (yes, really), which represent the smallest unit of movement for the mouse (typically around 0.1mm). A common horizontal range for a VGA display or similarly sized screen might be around 1600; in other words, for a screen / resolution 320 pixels wide, the mickey-to-pixel ratio will often be higher than 1-to-1.

Mice and similar pointer devices proliferated on x86 platforms after BIOS standards had become standardized; as a result, mouse-style hardware typically uses a driver TSR via handler 0x33. The mouse buttons numbers are **LEFT = 0**, **RIGHT = 1**, and **MIDDLE = 2**.

Real Mode Mouse Handler Requests (0x33): Direct Device Calls			
Description	**Function**	**Param(s)**	**Return(s)**
Check for Mouse	AX = 0x00	-	AX = 0xFFFF: Present AX = 0x0000: Absent
Get Button Press Changes	AX = 0x05	BX = Button #	AX = Button States (Bit # = Button #) BX = New Press Count
Get Button Release Changes	AX = 0x06	BX = Button #	AX = Button States (Bit # = Button #) BX = New Release Count
Get Movement Changes	AX = 0x0B	-	CX = New Horizontal Mickey Count DX = New Vertical Mickey Count

Real Mode Mouse Handler Requests (0x33): Interpreted Calls			
Description	**Function**	**Param(s)**	**Return(s)**
Show Cursor	AX = 0x01	-	-
Hide Cursor	AX = 0x02	-	-
Get Button/Cursor State	AX = 0x03	-	BX = Buttons (Bit # = Btn #) CX = Horizontal Position DX = Vertical Position
Set Cursor Position	AX = 0x04	CX = Horizontal DX = Vertical	-
Set Cursor Horizontal Limits	AX = 0x07	CX = Minimum DX = Maximum	-
Set Cursor Vertical Limits	AX = 0x08	CX = Minimum DX = Maximum	-
Set Cursor Mickey Rate	AX = 0x0F	CX = H. Mickeys / 8 Coords DX = V. Mickeys / 8 Coords	-
Set Cursor Sensitivity	AX = 0x1A	BX = Horizontal [1:100] CX = Vertical [1:100] DX = Double Threshold*	-
Get Cursor Sensitivity	AX = 0x1B	-	BX = Horizontal [1:100] CX = Vertical [1:100] DX = Double Threshold*

This is an abbreviated table. For more details, see Ralph's Interrupts for 0x33.

Note: Mouse drivers use vertical and horizontal BIOS cursor coordinate ranges of [0:199] and [0:639] respectively, even in MCGA mode; this means horizontal coordinates are 2x the pixel location!

***The "double threshold" is number of mickeys/sec after which the speed of the cursor accelerates (speed doubles).**

For example, to read the number of left button presses since the last call, we might use this routine:

```
// Inline assembly example (C/C++)
int getLeftPresses()
{
  int pressCount;
  _asm
  {
    mov ax, 0x05          // Get presses
    mov bx, 0x00          // Button 0 (left)
    int 0x33              // Invoke handler
    mov [pressCount], bx  // Copy count to variable
  }
  return pressCount;
}
```

Bringing it Together

Here's an example of how a program might access the screen and mouse.

1) Set the video to graphical mode (BIOS call `0x10`)
2) Check for mouse driver/hardware (BIOS call `0x33`)
3) Set the palette mask and palette colors (Data Bus, various ports)
4) While program is running…
    a. After V-Sync (Data Bus `0x03DA`), write pixels to screen (Memory Bus – `0xA0000`)
    b. Get and process mouse state (BIOS call `0x33`)
5) Set video to text mode (BIOS call `0x10`)