GameBoy Advance Barebones

From OSDev Wiki

Finally what you were expecting is going to be real. I'm going to show you the GameBoy Advance barebones.

Contents

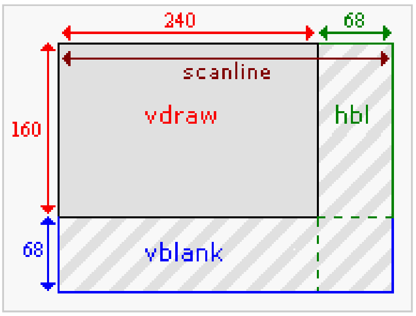
Requirements

- Some C/ARM Assembly knowledge
- A GameBoy Advance Emulator - VisualBoy Advance (http://vba.ngemu.com/) is a good one
- DevkitPro (http://www.devkitpro.org/) Toolchain

The Screen

The Nintendo GameBoy Advance screen, is an LCD with 240x160 resolution, that is entirely refreshed every 60th of a second. After a scanline has been drawn (the HDraw period, 240 pixels), there is a pause (HBlank, 68 pixels) before it starts drawing the next scanline. Likewise, after the 160 scanlines (VDraw) is a 68 scanline blank (VBlank) before it starts over again. To avoid tearing, positional data is usually updated at the VBlank. This is why most games run at 60 or 30 FPS.

Subject	Length	Cycles
Pixel	1	4
HDraw	240px	960
HBlank	68px	272
Scanline	HDraw+HBlank	1232
VDraw	160*Scanline	197120
VBlank	68*Scanline	83776
Refresh	VDraw+VBlank	280896



The diagram of the GameBoy Advance Display

Let's Code

Here I'm going to show you two ways to do it: One using Assembly(that it's better to begin with it, because you understand better how the machine works) and the second one using C. But it will be just a simple pixel-plotting at the time. Later I will add more tutorials about other things.

ARM Assembly

```
.arm
.text
.global main

.macro plot_pixel x, y, color
    mov r0, #0x6000000
    mov r1, \color
    ldr r2, =(\x+\y*240)*2
    str r1, [r0, r2]
.endm

main:
    mov r0, #0x4000000
    mov r1, #0x400
    add r1, r1, #3
    str r1, [r0]

    plot_pixel 115, 80, #0x1F
    plot_pixel 120, 80, #0x03E0
    plot_pixel 125, 80, #0x7C00

infin:
    b infin
```

C

```
int main()
{
    *(unsigned int*)0x04000000 = 0x0403;

    ((unsigned short*)0x06000000)[115+80*240] = 0x001F;
    ((unsigned short*)0x06000000)[120+80*240] = 0x03E0;
    ((unsigned short*)0x06000000)[125+80*240] = 0x7C00;

    while(1);

    return 0;
}
```

Compiling

To compile the Assembly source code, you just need to do this:

```
arm-elf-gcc -mthumb-interwork -specs=gba.specs kernel.S
arm-elf-objcopy -O binary a.out a.gba
```

To compile the C source, just use this Makefile:

```
PATH := $(DEVKITARM)/bin:$(PATH)

# --- Project details -------------------------------------------

PROJ    := first
TARGET  := $(PROJ)

OBJS    := $(PROJ).o

# --- Build defines ---------------------------------------------

PREFIX  := arm-eabi-
CC      := $(PREFIX)gcc
LD      := $(PREFIX)gcc
OBJCOPY := $(PREFIX)objcopy

ARCH    := -mthumb-interwork -mthumb
SPECS   := -specs=gba.specs

CFLAGS  := $(ARCH) -O2 -Wall -fno-strict-aliasing
LDFLAGS := $(ARCH) $(SPECS)


.PHONY : build clean

# --- Build -----------------------------------------------------
# Build process starts here!
build: $(TARGET).gba

# Strip and fix header (step 3,4)
$(TARGET).gba : $(TARGET).elf
        $(OBJCOPY) -v -O binary $< $@
        -@gbafix $@

# Link (step 2)
$(TARGET).elf : $(OBJS)
        $(LD) $^ $(LDFLAGS) -o $@

# Compile (step 1)
$(OBJS) : %.o : %.c
        $(CC) -c $< $(CFLAGS) -o $@
```

```
# --- Clean -----------------------------------------------------

clean :
        @rm -fv *.gba
        @rm -fv *.elf
        @rm -fv *.o

#EOF
```

Explanation

This is a quick and dirty explanation of the earlier code. Those previously mentioned warped minds for whom this section is intended will probably prefer it that way. A more detailed discussion will be given later.

As I said, GBA programming is low-level programming and sometimes goes right down to the bit. The 0x04000000 and 0x06000000 are parts of the accessible memory sections. These numbers themselves don't mean much, by the way; they just refer to different sections. There aren't really 0x02000000 between these two sections. As you can see in the memory map, these two sections are for the IO registers and VRAM, respectively.

To work with these sections in C, we have to make pointers out of them, which is what the unsigned int* and unsigned short* do. The types used here are almost arbitrary; almost, because some of them are more convenient than others. For example, the GBA has a number of different video modes, and in modes 3 and 5 VRAM is used to store 16-bit colors, so in that case casting it to halfword pointers is a good idea. Again, it is not required to do so, and in some cases different people will use different types of pointers. If you're using someone else's code, be sure to note the datatypes of the pointers used, not just the names.

The word at 0400:0000 contains the main bits for the display control. By writing 0x0403 into it, we tell the GBA to use video mode 3 and activate background 2. What this actually means will be explained in the video and bitmap mode chapters.

In mode 3, VRAM is a 16-bit bitmap; when we make a halfword pointer for it, each entry is a pixel. This bitmap itself is the same size as the screen(240x160) and because of the way bitmaps and C matrices work, by using something of the form array[y*width + x] are the contents of coordinates(x, y) on screen. That gives us our 3 pixel locations. We fill these with three 16-bit numbers that happen to be full red, green and blue in 5.5.5 BGR format. Or is that RGB, I always forget. In any case, that's what makes the pixels appear. After that there is one more important line, which is the infinite loop. Normally, infinite loops are things to be avoided, but in this case what happens after main() returns is rather undefined because there's little to return to, so it's best to avoid that possibility.

Links

- GBA LCD Video Controller (http://nocash.emubase.de/gbatek.htm#gbalcdvideocontroller)
- GBA Reference Document (http://gbadev.org/download.php?section=documentation&ID=7)
- GBA Quick Reference Guide Rev. 2 (http://gbadev.org/download.php?section=documentation&ID=4)