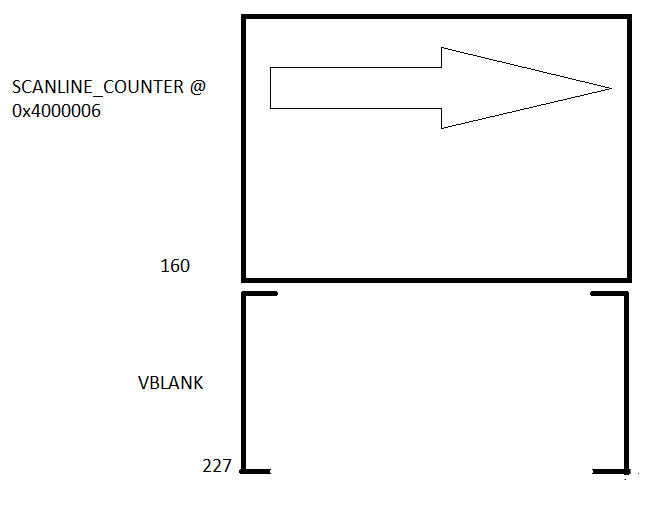
CS2261 Media Device Architecture



* The scanline counter is a register that keeps track of which line the screen is on, and it’s location is at 0x4000006.
  + After finishing, it resets everything to a 0
  + #define SCANLINECOUNTER (\*(volatile unsigned short \*)0x4000006)
    - We use the keyword **volatile** to ensure that the compiler doesn’t try and optimize our code by ignoring the scanline counter (since the scanline counter is changed by the hardware, C doesn’t expect it to change). Using **volatile** keeps the macro up to date and forces C to retrieve the info from memory each time.
* It is important to remember that the CRT visualization we use to track our screen is not really the way the screen works, but is sufficient for us to work with
* A function called waitForVBlank is created that runs until the screen hits the VBlank. After it ends, the user can start setting pixels.
  + This maximizes the time the user has to set pixels and avoid flicker.

void waitForVBlank() {

while(SCANLINECOUNTER > 160);

while(SCANLINECOUNTER < 160);

}

* The first while loop works for when the screen is already in Vblank and runs until the screen leaves Vblank
* The second one runs while the screen is right before Vblank
  + These two put together gives us the full time for setting pixels
* For Mode 3, we can restructure our code to be the most efficient
  + Calculate – do any updating for variables, set rows and columns, color, etc.
  + Delay (optional) – this is only optional because usually the waitForVBlank function is delay enough
  + waitForVBlank() – wait until the screen starts at VBlank for maximum time. Note that this also locks down the animation to 60fps
  + Draw – set pixels at this point
* Only use waitForVBlank once per frame, because using it multiple times will decrease your framerate from 60fps and your animation will look choppy.
* If we can finish setting our pixels in the time of VBlank (1/200th of a second), there will be no flicker. However, if we go over that time, there will be flicker.
  + In Mode 3, there is no fix to this. The code should be short and efficient enough to avoid this.

Buttons

* The only input the GBA has are 10 buttons
  + A (typically primary player action)
  + B (typically secondary player action)
  + Select (typically in-game menu)
  + Start (typically start game/pause menu)
  + Right Bumper (typically next action/right directional action)
  + Left Bumper (typically previous action/left directional action)
  + Up (typically upward movement or jump)
  + Down (typically downward movement or crouch)
  + Right (typically rightward movement)
  + Left (typically leftward movement)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15-10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  | Left | Right | Down | Up | Left Shoulder | Right Shoulder | Start | Select | A | B |
|  | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

* Each button is wired to a location in memory, and these wires are constantly live until you press down on them and cut off the circuit for that button
  + Hence the 1111111111111111 – representing no buttons pressed
  + 0 represents a pressed button
  + #define BUTTONS (\*(volatile unsigned short \*)0x4000130)
    - The circuitry changes which buttons are pressed but C is inclined to ignore these changes by the hardware, which is why we have to use the word volatile
* To tell is a button is on or pressed, make a mask (flag) for each button with its bit on
  + Ex. #define BUTTON\_START (1<<3)
* Check if the bit in question in on (with &) in ~BUTTONS
  + 1111111111110111 BUTTONS
  + 0000000000001000 ~BUTTONS
  + 0000000000001000 BUTTON\_START
  + 0000000000001000 ~BUTTONS & BUTTON\_START

Button Events

* GBA doesn’t have button events so we make our own:
* #define BUTTON\_HELD(key) (~BUTTONS & (key))
  + True for as long as the button is held down
* If we check every frame, the thing we do as a result will happen every frame.
* What if we want something to only happen once?
  + Best solution: we check if the key is down this frame, but not the last one.
* We use the BUTTON\_PRESSED macro to check the keys
* #define BUTTON\_PRESSED(key) (!(~oldButtons & (key)) && (~buttons & (key)))

Review:

* #define BUTTON\_HELD(key) (~BUTTONS & (key))
  + The always-accurate poll of whether the key is down now
  + Check every frame, and it’ll be true until the key is released
* #define BUTTON\_PRESSED(key) (!(~oldButtons & (key)) && (~buttons & (key)))
  + True if the key is down but wasn’t before, otherwise false
  + Check every frame, and it’ll be true only once, until you press the key again