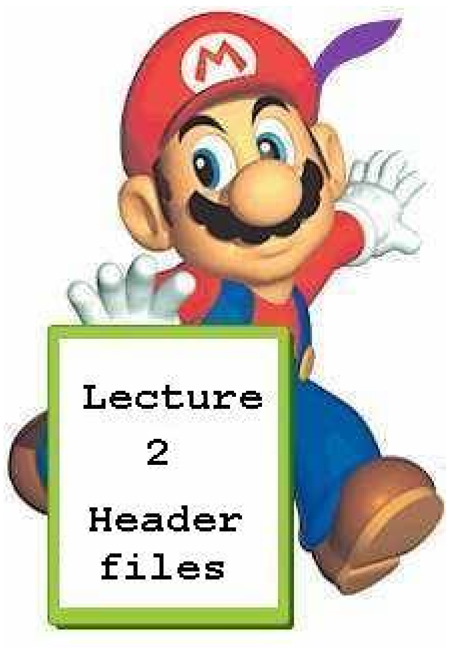

Programming the GBA

When you program the gba you usually write values directly to memory addresses. If you are going to use the addresses directly in your code, you have to remember what the addresses mean. Most people that code gba make header files that contain definitions of macros for these addresses. To make a macro to an address means that you make a descriptive name for an address, so that every time you want to write to that address you use this name instead of remembering the address. This will also make your code easier to read and understand. There are a lot of macros in gba programming and most programmers use the same macro names. But, if you really want to, it is no problem to change the macros to whatever you would want.

Most used macros

Here comes a list of the macros I use in these lessons. If you don’t understand these macros right away I can promise you will understand more and more as you start using them in the programs in these lessons.

gba_types.h

The header gba_types.h contains short forms of the most used variables. For example, instead of defining a variable by writing *unsigned short variablename*, you can just write *u16 variablename*. RGB(r,g,b) is a macro that makes it possible to specify colors in an rgb palette, this is something that is very common, it returns a value between 0 and 32767. On the gba colors are stored in 16 bit values as RGB555, meaning each of the colors (red, green and blue) are represented by 5 bits (the last 16th bit is ignored.) The number for red is stored in the 5 lowest bits from 0-4, green is the 5 bits from 5 to 10 and blue is the 5 highest bits. We will accomplish this by setting the 5 least significant bits to the red value, shift 5 bits to the left to set the green value and shift 10 bits to the left to set the blue value.

```
//
// Variable types
//

#ifndef GBA_TYPES_H
#define GBA_TYPES_H
```

```
typedef unsigned char    u8;
typedef unsigned short   u16;
typedef unsigned long    u32;

typedef signed char      s8;
typedef signed short     s16;
typedef signed long      s32;

#define RGB(r,g,b)   ((u16)(r | (g<<5) | (b<<10)))

#endif
```

gba_regs.h

This header defines the most used registers needed to program graphics on the gba. The 'new' thing in this header is that we are making pointers to addresses in memory. FrontBuffer is such a pointer, when we define it we cast a hexadecimal address to an unsigned short (or a u16 as we called it in gba_types.h).

This is done like this: #define FrontBuffer   ((u16*)0x6000000)

We know that FrontBuffer is a pointer to the address area 0x6000000. Because of that we can index FrontBuffer as a table (array) and write directly to the addresses in memory. One thing that is important to remember is that Frontbuffer[0] equals address 0x6000000, Frontbuffer[1] equals address 0x6000002 and Frontbuffer[2] equals address 0x6000004. The reason for this is that Frontbuffer is defined as a 16bits pointer/table, and that each index in the table equals 16 bits. By the way: it is not possible to write only 8 bits to an address (even though the address 0x6000000 is 8 bit), this is very important to keep in mind when we start programming.

The sentence #define REG_DISPCNT   *(u16*)0x4000000, makes a pointer to an address, but this pointer is de-referenced so we have direct access to the value stored at this address. This address is not indexable like Frontbuffer, but actually a fixed address that you can write to.

The reason we use volatile when defining REG_VCOUNT   *(volatile u16*)0x4000006 is that we wish to continuously update the info at this address. Volatile forces the program to continuously read the value from this address each time we use this macro.

The line #define vsync() while(REG_VCOUNT != 160); is a macro that waits until the screen of the gba is finished drawing before it proceeds with the execution of the program. This macro is often used when you want to make sure something is finished drawing before you start drawing it once more. If you do not use this macro your graphics will flicker, its usually used when drawing sprites to the screen.

```
//
// Registers
//

#ifndef GBA_REGS_H
#define GBA_REGS_H

#include "gba_types.h"

#define OAM_Mem              ((u16*)0x7000000) // Sprites(128), coordinates, size..(total 1Kb)
#define OBJ_PaletteMem       ((u16*)0x5000200) // Sprite Palette(256/16 colors)
#define OAM_Data             ((u16*)0x6010000) // Sprite data (bitmapped)
```

```
#define FrontBuffer     ((u16*)0x6000000) // Front Display Memory (the screen in mode 3-5)
#define BG_PaletteMem          ((u16*)0x5000000) // Background Palette(256/16 colors)

#define REG_DISPCNT         *(u16*)0x4000000 // Display control mode
#define REG_VCOUNT         *(volatile u16*)0x4000006  // Vertical control sync
#define vsync() while(REG_VCOUNT != 160);

#endif
```

gba_keys.h

This header defines a macro for testing which button you have pressed on the gba.

```
#define KEYS         *(volatile u16*)0x04000130
#define keyDown(k)   (~KEYS & k)
```

If we have pressed the start button (only) we will have this pattern of bits in the KEYS macro.

```
KEYS   =  1111 1111 1111 0111
```

To check if start really is pressed, we invert this bit pattern by writing the following in the macro.

```
~KEYS =  0000 0000 0000 1000  ( ~ is used to invert the value)
```

By using the logical AND operator and the value for the button we wish to check (defined in the header with the specific name of the button), we are able to see if that button is pressed.

```
~KEYS           0000 0000 0000 1000
& (AND)         0000 0000 0000 1000  (this value is stored in the macro KEY_START)
Result          0000 0000 0000 1000
```

Not like 0, means we have pressed the START button.

```
//
// Test input from buttons
//

#ifndef GBA_KEYS_H
#define GBA_KEYS_H

#include "gba_types.h"

#define KEY_A                   0x001
#define KEY_B                   0x002
#define KEY_SELECT              0x004
#define KEY_START               0x008
#define KEY_RIGHT               0x010
#define KEY_LEFT                0x020
#define KEY_UP                  0x040
#define KEY_DOWN 0x080
#define KEY_R                   0x100
#define KEY_L                   0x200

#define KEYS         *(volatile u16*)0x04000130

//Use like this: if(keyDown(KEY_A)) {};
```

```
#define keyDown(k)   (~KEYS & k)

#endif
```

gba_video.h

This header contains a macro that is used to set which video mode we wish to use, as well as some other important settings. The 16 bit register REG_DISPCNT is used to do this.
Different macros are used to translate these settings into a more understandable language.
The REG_DISPCNT register must be set before we try to show anything on the screen because this register decides how the different parts of the gba memory will work.

```
//
// Video memory setup
//

#ifndef GBA_VIDEO_H
#define GBA_VIDEO_H

#include "gba_types.h"
#include "gba_regs.h"

//define the screen width and height values to be used
#define SCREEN_WIDTH     240
#define SCREEN_HEIGHT  160

#define MODE_0       0x0     //screen mode 0
#define MODE_1       0x1     //screen mode 1
#define MODE_2       0x2     //screen mode 2
#define MODE_3       0x3     //screen mode 3
#define MODE_4       0x4     //screen mode 4
#define MODE_5       0x5     //screen mode 5

#define backbuffer       0x10    //define the buffer which is used to set the
                                //active buffer(using double buffering)

#define H_BLANK_OAM      0x20    //This bit, when set allows OAM
                                  //(Object Attribute memory) to be updated during a //horizontal blank

#define OBJ_MAP_2D          0x00    //Sprite data is stored in a 2D array(dont use this!!)
#define OBJ_MAP_1D          0x40    //Sprite data is stored in a 1D array

#define BG0_ENABLE          0x100  //Enables background 0
#define BG1_ENABLE          0x200  //Enables background 1
#define BG2_ENABLE          0x400  //Enables background 2
#define BG3_ENABLE          0x800  //Enables background 3
#define OBJ_ENABLE          0x1000           //Enables sprites

#define SetMode(mode) REG_DISPCNT = (mode)

//Set the mode that you want to use, logical OR them together as below:
//e.g. SetMode(MODE_2 | OBJ_ENABLE | OBJ_MAP_1D);

#endif
```

gba_sprites.h

This is the most advanced header in these lessons and it is made to simplify the use of sprites. This header defines a structure with 4 different 16-bit attributes that are used for storing information about each of the 128 sprites. We have made definitions for the different bits so they can be used to set the attribute info in an easy way.

An explanation of the rotation macro is missing.

```
//
// Sprites
//

#ifndef GBA_SPRITES_H
#define GBA_SPRITES_H

#include "gba_types.h"

// Attribute 0
#define ROTATION_FLAG          0x0100
#define SIZE_DOUBLE            0x0200
#define MODE_NORMAL            0x0000
#define MODE_TRANSPARENT       0x0400
#define MODE_WINDOWED          0x0800
#define MOSAIC                 0x1000
#define COLOR_16               0x0000
#define COLOR_256              0x2000
#define SQUARE                 0x0000
#define WIDE                   0x4000
#define TALL                   0x8000

// Attribute 1
#define ROTDATA(n)          (n << 9)
#define HORIZONTAL_FLIP        0x1000
#define VERTICAL_FLIP          0x2000
#define SIZE_8                 0x0000
#define SIZE_16                0x4000
#define SIZE_32                0x8000
#define SIZE_64                0xC000

// Attribute 2
#define PRIORITY(n)         ((n)<<10)
#define PALETTE(n)                 ((n)<<12)

//sprite structure definitions
typedef struct tagOAMEntry
{
        u16 attribute0;
        u16 attribute1;
        u16 attribute2;
        u16 attribute3;
}OAMEntry, *pOAMEntry;

//sprite rotation information (don't worry about this for now)
typedef struct tagRotData
{
        u16 filler1[3];
        u16 pa;
        u16 filler2[3];
        u16 pb;
        u16 filler3[3];
        u16 pc;
        u16 filler4[3];
        u16 pd;
}RotData, *pRotData;

#endif
```

gba_bg.h

This header is made to simplify the use of the background control registers and for setting the Char Base Blocks and Screen Base Blocks.

```
#ifndef GBA_BG_H
#define GBA_BG_H


///BGCNT defines ///
#define BG_MOSAIC_ENABLE              0x40
#define BG_COLOR_256                  0x80
#define BG_COLOR_16                   0x0

#define CharBaseBlock(n)        (((n)*0x4000)+0x6000000)
#define ScreenBaseBlock(n)      (((n)*0x800)+0x6000000)

#define CHAR_SHIFT                    2
#define SCREEN_SHIFT                  8
#define TEXTBG_SIZE_256x256           0x0
#define TEXTBG_SIZE_256x512           0x8000
#define TEXTBG_SIZE_512x256           0x4000
#define TEXTBG_SIZE_512x512           0xC000

#define ROTBG_SIZE_128x128            0x0
#define ROTBG_SIZE_256x256            0x4000
#define ROTBG_SIZE_512x512            0x8000
#define ROTBG_SIZE_1024x1024          0xC000

#define WRAPAROUND              0x1

#endif
```

gba.h

This is just a header file that includes all the other header files we have made. If you include this file in the top of your program it means that you have automatically included all the other header files. To do this just write #include ”gba.h”. We could of course have written all our header files in one big header file, but that would have been more messy and harder to work with.

```
//
// Including all the macros into gba.h
//
```

```
#ifndef GBA_H
#define GBA_H

#include "gba_types.h"
#include "gba_regs.h"
#include "gba_keys.h"
#include "gba_video.h"
#include "gba_sprites.h"
#include "gba_bg.h"

// e.g to include all header files, just write #include "gba.h" in the top of your program

#endif
```

The structure of your programs

When you are going to make a new gba program it is always smart to make your own folder that you put all the project files into. In that way you will get the best overview of your code, and it is easier to find errors in your code. When you input the programs from these lessons you need to have the following files in your project folder.

- gba_types.h
- gba_regs.h
- gba_keys.h
- gba_video.h
- gba_sprites.h
- gba_bg.h
- gba.h
- Make.bat
- Test.cpp

It could be a good idea to copy this folder and just change the folder name. If not you will have to change all the names in your Make.bat and the name of your .cpp file each time you make a new program. This saves you some work, and leaves less room for mistakes, but its really up to you.