The Nintendo Reverse Engineering Project

Day 1

Things covered on day 1:

- Getting to know the GBA
- Figuring out these wonderful tools
- My first GBA demo

Getting to know the GBA

As the title of this chapter suggests, this will mainly be an introduction to the capabilities of the GBA. Below is a description of the system and a bit about its capabilities.

Processor: 32bit ARM7TDMI processor that runs on the RISC architecture. It is clocked at 16.78mhz. You can get all the documentation for the processor at http://www.arm.com/. As far as we have been able to determine the processor is identical to the standard ARM7TDMI. For the most part, every instruction on the arm takes one machine cycle to execute and the only waits imposed are due to memory reading and writing. For those of you who will be delving a bit deeper and playing a bit with the arm assembler here is a link to the Otaku's instruction list. This is a very nice reference of all the arm phonetics as well as instruction timing.

http://www.otakunozoku.com/gbacribsheet/index.html

One thing of note about the arm processors is that most, including the arm7, have two separate instruction sets. Both, 16-bit and 32-bit instructions are used on the arm which is an important concept to keep in mind. The processor must be made aware of switches between the two instruction sets. Fortunately the compiler and linker handle most of the details for us. We just specify, for each c file, what mode we want it compiled in. If our project mixes the two modes then the "interworking" flag must be set when compiling allowing the linker to link the two differing code blocks together. To simplify things all the code in this tutorial will just enable interworking.

The reason for the two sets is fairly straight forward. 32-bit code (called arm code) can fit a lot more functionality into each instruction making for faster executing programs. 16-bit (or thumb) is the code we will normally use since it has the advantage of being the same size as the data bus to ROM. This means that each instruction can be read in a single cycle (plus the wait state due to the slower memory). The basic rule is that if your code is running from ROM it is best to use thumb. If you need a bit of added oompf (say for a 3D triangle rasterizer or sound mixing code) compile the necessary code in ARM and place it into internal ram which has a 32-bit data bus.

Memory: The GBA has a lot of ram when compared to other 2D consoles. It has 96KB of video memory, 32KB of fast internal ram and 256KB of external ram. Reading and writing to ram is sometimes a bit confusing as certain areas have certain restrictions. Below is a list of the areas and how you can write/read to them:

Internal Working Ram (iwram): Begins at 0x3000000 and runs for 32KB. This is where your variables and stack are normally kept. 8, 16, and 32 bit reads/writes are allowed and all access is single cycle making iwram work at a speed comparable to a built in cache.

External Working Ram (ewram): Begins at 0x2000000 and runs for 256KB. 8, 16, and 32 bit read/write allowed. Unfortunately there is a two cycle penalty for reading and writing to this ram making it much slower then iwram.

ROM: Read only memory begins at 0x8000000 and runs for a maximum of 32MB. This is where your game cartridge data resides. If you are using a flash cartridge your code will begin its execution starting at this point. When reading from the cartridge there is a 3 cycle penalty unless you are reading sequential addresses, in which case there is only a one cycle penalty. This means that ROM is relatively slow for random access but it is actually quicker then ewram when copying chunks of data.

Register Memory: Beginning at 0x4000000 and continuing for some many bytes is the memory mapped register area. It is through the locations in this address range that all the internal communication between the arm processor (your code) and the various support chips (sound and video processor, DMA controller, etc…) take place. Much of this book is focused on understanding these registers so if this is all unclear now don't worry.

Video Memory: Video memory has several separate portions some of which have different capabilities and limitations. The video bus is 16 bits wide for all areas except Object Attribute Memory which is 32. You can read and write 16 or 32 bits at a time to all areas. DO NOT WRITE TO ANY VIDEO MEMORY WITH 8 BIT READ/WRITE. 8 bit operations (char) result in doubling of memory access. What this means is that if you try to write to an address not on a 16 bit boundary it will be adjusted to 16 bits and the value you write will be written (or read) twice. This effect will prove painful when we discuss bitmap graphics later on.

Background and Sprite palette memory: 1024 bytes of ram beginning at 0x5000000 holds the available colors for the sprites and backgrounds. The first 256 16-bit entries are for the background and the second 256 are for sprites. The background and sprites operate off of two separate 256 color palettes allowing a total of 512 colors on screen in paletted modes.

Object Attribute Memory: This memory begins at 0x7000000 and stores the sprite descriptions (such as height, width, location of sprite graphics data, etc…) for 128 sprites. This area of memory will be discussed in much more detail in the sprite chapters of this tutorial.

Main Video Memory: This memory begins at 0x6000000 and runs for 64KB. This is where we place our data for our bitmap backgrounds as well as our map data for tile

modes. This area also will be discussed at great length when the appropriate time has come.

Sprite character memory: This begins at 0x6010000 and runs for 32KB. We use this area to store the actual graphics for our sprites. The layout of this memory is quite flexible and even a little tricky so stand by. Only half of this memory is available when operating in bitmap modes (mode 3,4,5).

The GBA can operate in 6 different screen modes at a maximum resolution of 240x160 pixels. A brief description of each mode follows but a much more detailed description will come when we talk about actually programming these modes.

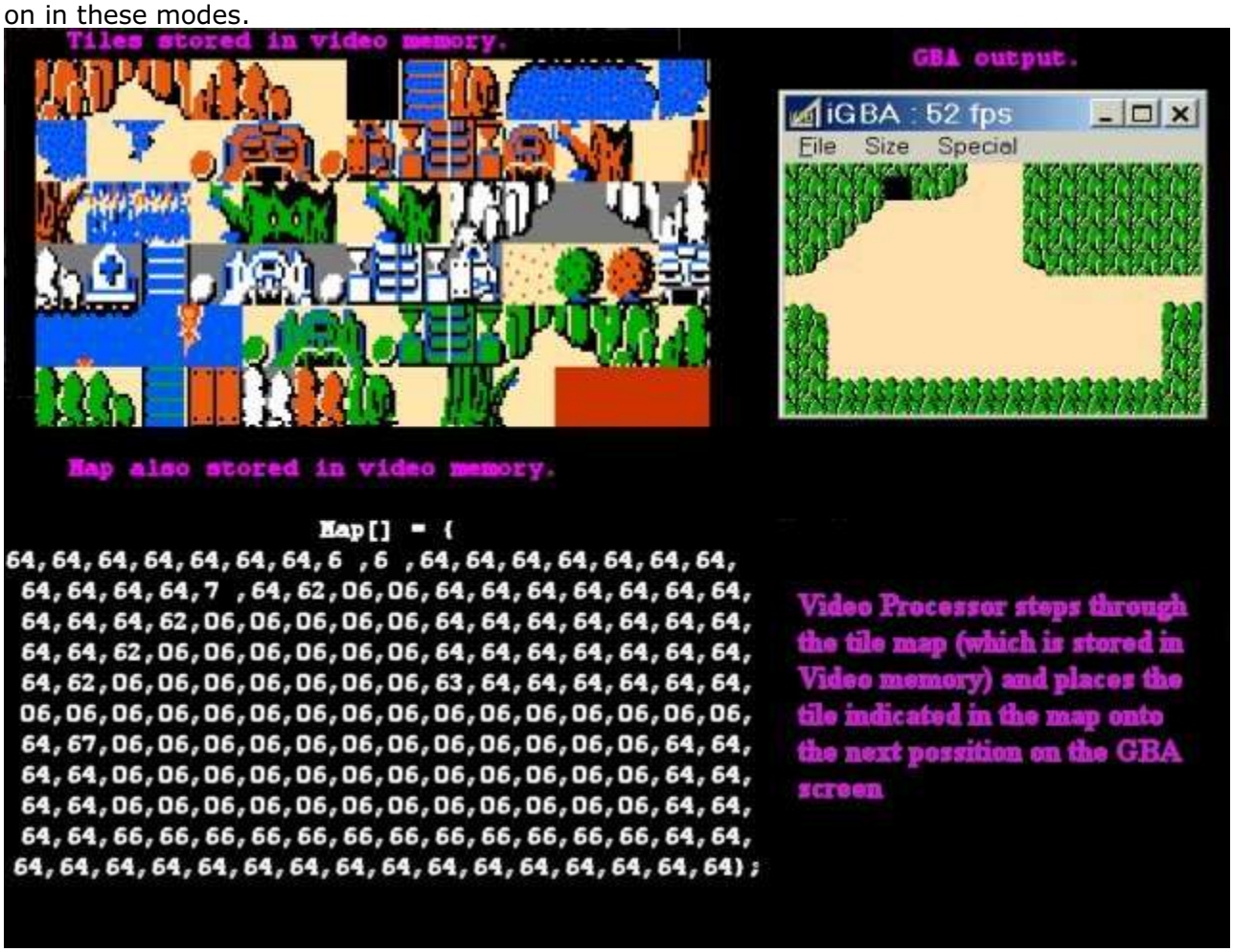
Mode 3
I like to start with this mode because it is the easiest to understand and use. Mode 3 is just a linear buffer of 16-bit pixels. What this means is that you can access the display as a simple array of colors. The first demo written in this series will use mode 3.

Mode 4
This is probably the next easiest mode to use and the best for accessing the screen as a big bitmap. If you have ever programmed for DOS in mode 13h then you will find this mode very familiar. Mode 4 is like mode 3 except the pixels are 8 bit indices into a 256-color 16-bit palette. This means that you store the colors that your background is going to use in palette memory and then access the screen as an array. Instead of just putting the color into the array you put an index in the array. The value can be from 0 to 255 and each number refers to a color in the palette. There are two major advantages to this:
1: It only takes half the time to draw a pixel to the screen (sort of).
2: It leaves enough memory left to have two screens in memory at the same time.
This allows you to draw to one screen while displaying the other and when you are done you just tell the video processor to switch to that screen, a process known as double buffering. This allows for incredibly smooth animation**.** Unfortunately there is also a major disadvantage to this mode and that has to do with the limitations of the video hardware. Since the video memory can not be accessed 8 bits at a time writing a single pixel becomes very difficult. We will talk about this and ways around it in day 2.

Mode 5
This has aspects of both Mode 4 and mode 3. Like mode 3 it is also a 16 bit linear buffer, but it unlike 3 it can be double buffered. The way this is done is that it only allows access to part of the screen; otherwise there would not be enough memory for two buffers. The screen is only 160x128 pixels in this mode. This mode is mainly reserved for video play back and some 3D demos.

Mode 0,1,2
Modes 0-2 operated as tile modes; the screen is broken down into 8x8 pixel squares that are put together to form the screen. What this means is that you will place several tiles (up to 1024) into video memory, then you will place a map in video memory that is just an array with each element a tile number (and a few attributes). The video processor looks at your map array and builds the screen out of the tiles you specify. The screen can be anywhere from 256x256 pixels to 1024x1024 (32x32

to 128x128 tiles). Look at the picture to get a better understanding of what is going on in these modes.



The GBA can display as many as four different backgrounds when operating in tile mode. Each background can be controlled independently using different maps and different tiles. Their scrolling attributes can also be controlled independently.

Color 0 is always transparent allowing higher priority backgrounds to be viewed through the lower level ones. By default the rendering of the backgrounds is in the order of 3,2,1,0 meaning background 0 will appear above background 3, but this can be changed. Also some backgrounds have the ability to be rotated and scaled. This is a very cool effect for any map.

Tiles can either be stored as 256-color bitmaps or 16-color bitmaps. They both use the same palette and different backgrounds can use different color modes. 16-color palettes work by splitting the 256-color palette into 16 separate palettes. So even in 16-color mode you can still have all 256 colors on the screen as long as each tile only uses one of the 16-color palettes.

The three tile modes differ in the features they have available to them. Not all modes can use all the background layers and not all modes allow you to rotate and scale backgrounds. The following table outlines the basic abilities of each of the tile screen modes.

Screen Modes

	MODE 0	MODE 1	MODE 2
Backgrounds Available	**All 4**	**0,1,2**	**2,3**
Rotation/Scaling?	**NO**	**Background 2**	**Background 2,3**

Each background layer can be a rotation or a non-rotation background depending on the screen mode. We refer to non-rotation backgrounds as text backgrounds. The following describes the capabilities of the two types of backgrounds:

Background Features

	Max Number Of Tiles	Size of Screen in Pixels	Number of Colors per Tile	Special Features
Rotation/Scaling	**256**	**128x128, 256x256 512x512, 1024x1024**	**256**	**Rotation and scaling**
Plain text backgrounds	**1024**	**256x256, 512x256 256x512, 512x512**	**256 or 16**	**Horizontal and vertical flipping of tiles**

All Backgrounds can be scrolled, Mosaic'ed, Faded in and out, and be semitransparent (alpha blended)

Sound:
The GBA has 6 sound channels. 4 that are comparable with the original GB/GBC and two additional sound channels that allow direct sound sampling (like playing a wave file). That is nearly the extent of my sound knowledge but we will cover simple digital sound play back and then find ourselves a nice prewritten library to do our music and sound effects. The reason for this is laziness on my part, but in my defense it would take a good 40 or 50 pages to do the topic any justice so I will leave writing your own MOD or MIDI playback library as an exercise for the reader ☺

Sprites:
The GBA supports hardware rendered sprites in all modes. They have there own palette that is separate from the Background palette and can be either 16-color or 256-color. This means that even in 8-bit color mode there can be 512 colors on the screen at one time. They can be a wide variety of sizes and can be scaled and rotated in hardware. There is room for defining 128 separate sprites and there is 32KB of memory for sprite graphics (only 16KB in bitmap modes 3, 4, and 5). It is even possible to trick the GBA into displaying more then 128 sprites per frame but there is rarely a time when that is necessary…then again when is programming a GBA ever necessary?

This was just an overview of the GBA specs; the details will come as you read further. But first, we must set our selves up to compile and run some code for the GBA.

Figuring out these wonderful tools
Fortunately things are much simpler for you then when we first started coding on the GBA. It used to be that you had to download the source code for GCC and compile it yourself which was a painful and somewhat buggy process on a good day. But thanks to people like Jeff Frohwein and Wntrmute we now have a one click install for your compiler setup (okay maybe two). One of the added benefits to this is that instead of a 50MB download and several hours of headache, it is now down to about 8MB counting an emulator or two.

Without further ado I present to you Wntrmute's wonderful pre-built GCC. As usual with the files on my site if I did not write them I try to direct you to their source as I am notorious for having old versions laying about.

Get the Compiler here. (It is under downloads and titled devkitARM).

If for some reason that link is dead here is a local copy that is likely to be outdated so please let me know if my links are failing and I will fix them. Local copy.

You will also need the source code for this tutorial. You can download it as you go or all at once…since it is not that big I recommend grabbing it now.

The last software tool you will need is xboo communicator. This handy little program is how we are going to get our code onto the GBA. We will also put it to more interesting and powerful uses later on.

In summary you should have downloaded:

devkitARM
Tutorial source code
XBoo Communicator

My directory structure simply has devkitARM and examples in c:\devkitARM. Where you place your tools is important because you will need to be able to find them at a later time.

Emulator:
Next we need an emulator. The reason for this is quite simple; testing code on an emulator is a much quicker process then trying to do it on the GBA.

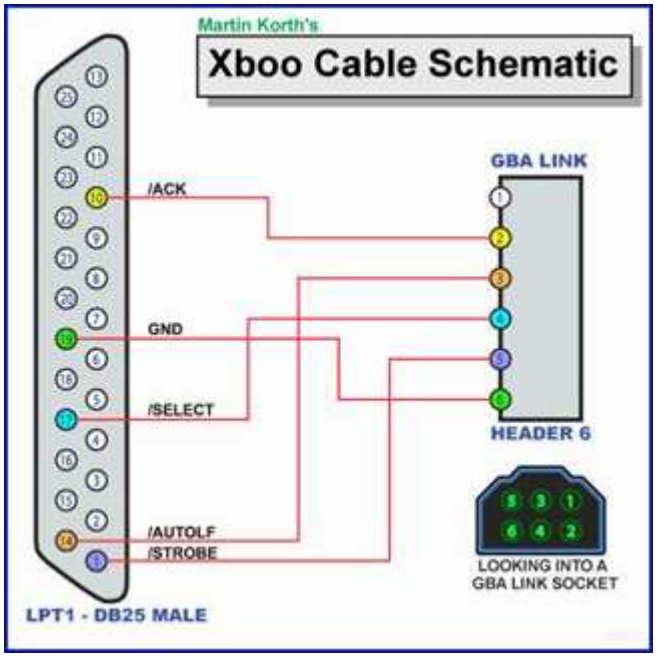
I recommend you get this emulator:

Visual boy advanced with sdl

Using it is very simple and even if you have never used an emulator before you should have no problem finding the file->open menu option.

Hardware:
Now for the final piece of the puzzle: Hardware! Let's face it, if you are not going to run your code on an actual GBA then you may as well be devving on windows. There

are three choices for hardware each having advantages and disadvantages. First I will tell you about the important one, and by important I mean practically free.



This is called the xboo cable and the reason it is free is because we are going to build it ourselves. Now if you are not quite as geeky as me and don't have spare electronic parts floating about then you may need to make a trip to radio shack but I promise it wont be too bad. First, let's look at a picture of the device we are going to build. This picture was provided by r6502.

What is the Xboo and how does it work? The nice people at Nintendo finally acknowledged one of the biggest complaints gamers had when playing multiplayer games. It seems people really did not like the fact that they had to have two copies of the game to go head-to-head. Nintendo fixed this by allowing one GBA to boot another over the link port. Now with a bit of work some people smarter then me and with better toys figured out how to make their PC act like that second GBA and boot the real GBA with code that you have written. Pretty slick huh?

The real draw back to this method is that because there is no cartridge we only have the available ram to play with…this means that our games are limited to 256KB in size. But, you will find this is more then enough for just about anything you need to test and only when your games start getting bigger will you need a flash cart.

Building an xboo cable

You are welcome to skip this section if you decide to go with the other options but, even if you do end up buying a flash cart or an xport, I do recommend building this cable; there is not really any faster way to test your code on hardware and having a nice debug console is pretty slick to.

To create this lovely and complex device you cut a GBA link cable in half (be careful because some link cables do not have all the required cables and/or pins). Then use a multi-meter to determine which cable connects to which pin. Find your self a DB25 Male connector and solder it all together. The connector should only be a few dollars at radio shack. I will try to put together a list of cables that are verified to work as I come across them. The only one I know for certain is the mad-catz gamecube->GBA adapter.

I am actually building one of these for the first time as I write this so let's see how it goes. It just so happens that I have a spare link cable and a db25 connector in my drawer. The link cable is MadCatz GBA->GC cable.

7:55pm
First thing I do is snip the link cable near the base of the little junction box so I have a lot of cable to work with. If you are using a standard GBA->GBA link I would snip it in the middle so you can use both ends if you need to. I then strip the cable and find to my bitter disappointment that there are only 3 wires…

7:58pm
Well fortunately this cable has three link cables so on further inspection I notice that I had actually snipped the cable that went to the cube. I move on to the next connector and repeat and am happy to see this one has five wires and no pin one so things are looking good. Next I use the multi-meter on resistance setting to determine which wire is connected to which pin. Easy to do, but remember you are looking at the male end and the drawing above is the GBA side which is the female so things are a bit backwards.

I find that on mine:
White – pin 2
Blue – Pin 3
Green – Pin 4
Brown – Pin 5
Black – Pin 6

8:25pm (wife had me do a few chores)
All that is left is the exciting process of soldering the right colors onto the right pins. Unfortunately I have run into another snag…seems my solder is a bit too thin to be in any way useful. I should go to the store and get a decent thickness but I am both too lazy and too cheep so I will make do…Wish me luck.

8:46pm (in-laws stopped by but managed to ignore them other than a quick hello)
All put together…now time to test. First I download Xboo communicator from wntrmute's site and then I give it a go…lets see what happens….didn't work…

9:06pm
So after a bit of looking and being generally angry I find that I connected pin 1 to pin 12 instead of pin 10…after heating up my soldering iron once more I find that XBOO is a success.

Seems it takes a bit over an hour with a couple interruptions and a few mistakes but all in all not too bad. If you have a soldering iron or a friend with one I recommend this option highly. It is cheep and easy even if you don't have much experience with that sort of thing.

Flash Cart
The next option is purchasing a Flash cartridge. These are generally only available outside of the US and Nintendo has a habit of shutting down the distributors of these fine products…it seems that 99% of the people who buy them are doing it just to play pirated games they downloaded off the net. Below is a link to a site that still sells them and if you buy one it does support this site. When the site makes money my wife generally leaves me alone long enough to update it so it's up to you!

Using the flash cart is self explanatory as they generally come with a windows GUI and are built for ROM pirate types who don't know a damn thing about computers.

Xport
The final and by far the most interesting option is something called the XPORT. Here is a link to their wonderful site.
www.charmedlabs.com
That is all I will say about the xport as I don't have the energy to go into the millions of things you can do with this thing. Okay, I lied…I have one thing more to say about the xport. Here is the only interesting thing I have done with it so far:
www.thepernproject.com/rpi/NEEP_FINAL.ppt

My first GBA demo
The best way to get into this adventure is to jump right in with some source code. We will step through compiling it and executing it on the emulator as well as the actual hardware. We are not going to go into the details of the workings of the actual code at this time but I think you will be able to follow it as it is quite simple.

Tutor 1 day 1

```
#include <gba.h> //everything you need for gba deving

////////////////// C code entry (main())//////////////////////
int main()
{
    unsigned char x,y;

    SetMode(MODE_3 | BG2_ENABLE);

    for(x = 0; x < SCREEN_WIDTH; x++)
        for(y = 0; y < SCREEN_HEIGHT; y++)
            VideoBuffer [x+ y * SCREEN_WIDTH] = RGB16(31,0,0);

    while(1){}
 }//end main
```

Extract the tutorial files into a directory of your choice. Click on "build.bat". If your compiler is installed correctly then you should get no errors. If you installed devkitARM to a directory other then c:\devkitarm you will need to edit the batch file to reflect this. Just change this line:
set DEVDIR=C:\Devkitarm

to the directory in which you installed the devkit. Also ensure you have all the header files that are being used for this tutorial. If you missed them get them here: headers.

If that worked you should have a tutor1_1.gba in your source directory. Open it up in an emulator by first launching the emulator and selecting file->open. You should see a red screen. Congratulations you just compiled your first GBA demo! Now I suppose you want to know how it works. The batch file calls the compiler and linker from the command line and links the files together with the proper command line arguments. If you do not really understand batch files, compilers and linkers, I recommend you at least look inside the batch file and ponder its simplicity before thinking that it is beyond you. It is just as simple as if you typed in the commands from a DOS or UNIX prompt as for the command line options themselves they are simply telling the compiler the type of CPU, what level of optimization to use, warning level and that sort of thing. There are two that defiantly stick out though and deserve some explanation.

-mthumb-interwork tells the compiler to produce code that can be called from thumb or arm mode. Without this option your code will likely fail to function unless you use strictly arm or thumb code.

-specs=gba_mb.specs tells the compiler to use the startup code and linker script that produce a GBA binary and to make it multiboot compatible (meaning its address space begins at 0x2000000 –EWRAM- instead of 0x8000000 –ROM-) to ensure it will run from EWRAM without issue. The other option is –specs=gba.specs which produces a normal GBA binary for use on a flash cart. The multiboot version checks when it is loaded to figure out were it is and will function from a flash cart or multiboot, but it still takes up valuable EWRAM space. If you are devving from a flash cart it is recommended you use gba.specs.

Now this would not be much fun if we did not fire up our new XBOO cable and test it out on the GBA. First connect your GBA to the xboo cable and to your parallel port and turn on the GBA with no cartridge. Next, simply open xboo communicator and click on the GBA icon. Finally, it is just a matter of navigating to your tutor1_1.gba and selecting it. You should see the xboo sending the image to the GBA and you should also see the GBA run your first GBA demo. If this failed to work, try adjusting the Xboo communicator settings by increasing the delay and ensuring that your parallel port is the one selected in the drop down box.

That is it for day one. Before moving on to day two I recommend trying to add a c file with a simple function, even one that does nothing, and getting it to compile. If anything in this day is unclear feel free to email me and I will try to elaborate on anything covered. Just keep in mind that this is just day one and no real GBA programming knowledge was discussed. Today was just an introduction and if the code in the tutorial makes no sense that is okay. When we get into Day 2 you will learn all about that code. If you could not get the tutorial to compile or you were not able to add another C file to the build.bat file then that will be a problem. If you want to see it on hardware then you need the XBoo cable or the Flash advance linker. So go and build an Xboo! Or you can just click the link below and get a flash cart :)