

Table of Content

NDS/Tutorials - Dev-Scene

NDS/Tutorials	4
NDS/Tutorials	4
Contents	4
[edit] Day 1: Setting up	4
[edit] Day 2: NDS Introduction	4
[edit] Day 3: 2D Raster Graphics and Input	4
[edit] Day 4: 2D Tile Graphics	4
[edit] Day 5: 2D Sprites	4
[edit] Day 6: 2D Hardware Effects	5
[edit] Day 7: Sound and Music	5
[edit] Day 8: 3D Introduction	5
[edit] Day 9: Wifi and Networking	5
[edit] Day 10: A bit more advanced 3D Graphics	5
[edit] Game Programming 101: Collision Detection	5
[edit] Game Programming 101: Animation	5
[edit] Tiling, Sprites and Animation	5

NDS/Tutorials Day 1 - Dev-Scene

NDS/Tutorials Day 1	6
NDS/Tutorials Day 1	6
Contents	6
[edit] Introduction (feel free to skip this)	6
[edit] Me (Dovoto)	6
[edit] Who Should Read This?	6
[edit] Requirements	6
[edit] Why The Nintendo DS?	6
[edit] Bit o' History	6
[edit] What You Need To Know	6
[edit] Installation Of Tools	7
[edit] For windows	7
[edit] For Mac OS X	7
[edit] Setting Up An IDE	8
[edit] Building A Demo And Testing The Installation	8
[edit] Emulators	9
[edit] A Few Good Emulators	9
[edit] Running The Demo On Hardware	9
[edit] First Demo	9

NDS/Tutorials Day 2 - Dev-Scene

NDS/Tutorials Day 2	11
NDS/Tutorials Day 2	11
Contents	11
[edit] Hardware Overview	11
[edit] Memory Layout	11
[edit] Main Memory	13
[edit] ARM 7 Fast Ram (IWRAM)	13
[edit] ARM 9 Caches	13
[edit] Fast Shared Ram	13
[edit] Video Ram	13
[edit] Virtual Video Ram	13
[edit] Sound Hardware	14
[edit] Wifi	14
[edit] Input	14
[edit] Touch Screen	14
[edit] Buttons	14
[edit] Microphone	14
[edit] Real-Time Clock	14
[edit] Upgradeable Firmware	14
[edit] Graphics Overview	14
[edit] 2D	15
[edit] 3D	15
[edit] Toolchain Explained	15
[edit] Compiler	15
[edit] Linker	16
[edit] Build A Demo The Hard Way	16
[edit] Conclusion	17

NDS/Tutorials Day 3 - Dev-Scene

NDS/Tutorials Day 3	18
NDS/Tutorials Day 3	18
Contents	18
[edit] What is a register	18
[edit] Twiddling Bits	19
[edit] Numbering Systems	19
[edit] Bitwise Operations	20
[edit] Talking to the keypad	21
[edit] Frame buffer...finally	23

[edit] Display Control	24
[edit] VRAM Control	24
[edit] DS Color Formats	24
[edit] Frame buffer 101	25
[edit] Pixels and things	25
[edit] Touching things	28
[edit] Bitmap Graphics Modes	29
[edit] Bitmap on the sub display	32
[edit] Background struct	32
[edit] Working With Graphics Files	33
[edit] The Double Buffer	35
[edit] Raster 101	38
[edit] Bresenham Lines	38
[edit] Blitting Things	41
[edit] Drawing Pictures	41
[edit] Polygons	41
[edit] Circles	41

NDS/Tutorials Day 4 - Dev-Scene

NDS/Tutorials Day 4	43
NDS/Tutorials Day 4	43
Contents	43
[edit] Introduction	43
[edit] Tile Modes	43
[edit] Background Memory Layout and VRAM Management	44
[edit] Map Entries	45
[edit] First Map Demo	46
[edit] Text Backgrounds	47
[edit] Rotation Backgrounds	47
[edit] Extended Rotation Backgrounds	47
[edit] Creating Map Data	47
[edit] From an Image	47
[edit] Using a Map Editor	47
[edit] Meta Tiles	47
[edit] Scrolling	47
[edit] Horizontal Scrolling	47
[edit] Vertical Scrolling	47
[edit] Scrolling Both Ways	47
[edit] Dynamic Tile Loading	48

NDS/Tutorials Day 5 - Dev-Scene

NDS/Tutorials Day 5	49
NDS/Tutorials Day 5	49
Contents	49
[edit] intro	49
[edit] Setting your sprite up for GRIT	49
[edit] Make File	49
[edit] C++ Code	50
[edit] oamset functions	51

NDS/Tutorials Day 6 - Dev-Scene

NDS/Tutorials Day 6	52
NDS/Tutorials Day 6	52

NDS/Tutorials Day 7 - Dev-Scene

NDS/Tutorials Day 7	54
NDS/Tutorials Day 7	54

NDS Tutorials Day 8 - Dev-Scene

NDS Tutorials Day 8	56
NDS Tutorials Day 8	56

NDS/Tutorials Day 9 - Dev-Scene

NDS/Tutorials Day 9	58
NDS/Tutorials Day 9	58

NDS/Tutorials Day 10 - Dev-Scene

NDS/Tutorials Day 10	60
NDS/Tutorials Day 10	60

NDS/Tutorials Collision Detection - Dev-Scene

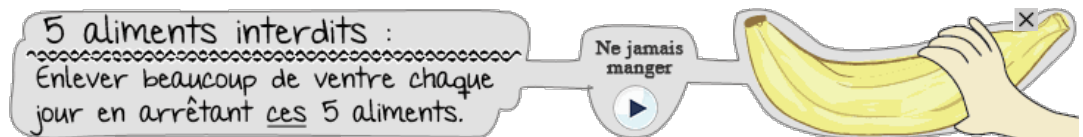
NDS/Tutorials Collision Detection	62
-----------------------------------	----

NDS/Tutorials Collision Detection	62
Contents	62
[edit] 2-D Collision Detection	62
[edit] Intersecting Rectangles	62
[edit] Pixel Collision	63
[edit] Row-Rectangle Intersection	63
[edit] Rectangle Subdivision	64
[edit] R-Squared Method	64
 NDS/Tutorials Animation - Dev-Scene	
NDS/Tutorials Animation	65
NDS/Tutorials Animation	65
Contents	65
[edit] Animated sprites for embedded devices	65
[edit] Getting your feet wet...	65
[edit] The Manly Way	66
[edit] Animating: The Manly Way	67
[edit] The Womanly Way	68
[edit] Animating: The Womanly Way	69
[edit] Conclusions: Which One, Which One?	69
[edit] Notes	69
 NDS/Tutorials/Captain Apathy/Tiling - Dev-Scene	
NDS/Tutorials/Captain Apathy/Tiling	71
NDS/Tutorials/Captain Apathy/Tiling	71
Contents	71
[edit] Intro	71
[edit] Tiling	71
[edit] Getting started in non 32x32 arrangements	71
[edit] So why 64x64?	71
[edit] Hey, you said you would talk about scrolling!	71
[edit] Setting Display order	72
[edit] Sprites	72
[edit] Starting out	72
[edit] Where the sprites information is held	72
[edit] Movement and Rotation	73
[edit] Animation	73
[edit] Sprite Index Changing	74
[edit] Image uploading	74
[edit] Sprite Buffer swapping or vram to vram	74
[edit] So which one should be used?	74
[edit] Dynamic sized 2d tile systems	74

Dev-Scene	
Home	↗
News	↗
Forum	↗
Blogs	↗
Planet	↗
Library	↗



Navigation	
Nintendo DS	↗
- NDS Homebrew Catalog	↗
Nintendo Wii	↗
- Wii Homebrew Catalog	↗
Current events	↗
Community portal	↗
Developers/Donations	↗
Personal tools	
Log in / create account	↗
Toolbox	
Random page	↗
Recent changes	↗
Help	↗
What links here	↗
Upload file	↗
Special pages	↗



NDS/Tutorials

[< NDS](#)

Contents

- [1 Day 1: Setting up](#)
- [2 Day 2: NDS Introduction](#)
- [3 Day 3: 2D Raster Graphics and Input](#)
- [4 Day 4: 2D Tile Graphics](#)
- [5 Day 5: 2D Sprites](#)
- [6 Day 6: 2D Hardware Effects](#)
- [7 Day 7: Sound and Music](#)
- [8 Day 8: 3D Introduction](#)
- [9 Day 9: Wifi and Networking](#)
- [10 Day 10: A bit more advanced 3D Graphics](#)
- [11 Game Programming 101: Collision Detection](#)
- [12 Game Programming 101: Animation](#)
- [13 Tiling, Sprites and Animation](#)

[\[edit\]](#) [Day 1: Setting up](#)

Setting up the system to start coding.

[\[edit\]](#) [Day 2: NDS Introduction](#)

An introduction to the capabilities of the DS hardware. This tutorial covers the DS hardware at a high level as well as many of the key concepts necessary for understanding any of the hardware at a low level (including memory manipulation with pointers, register based IO, and video memory management).

[\[edit\]](#) [Day 3: 2D Raster Graphics and Input](#)

Lines, circles, and bit blits will be covered as we explore the many 2D raster modes the DS has to offer. Also covered will be how to grab input from buttons and the touchpad.

Of course, before we can tend to these more interesting facets of programming we will take a peak at just what a register is, how to write to it, binary and hexadecimal numbering systems, and bitwise operations.

[\[edit\]](#) [Day 4: 2D Tile Graphics](#)

Tile based graphics are the staple of the Nintendo DS and this chapter should prove to be the most informative and useful. We cover the generation of maps and tile graphics and how to put them all together to create your game worlds.

[\[edit\]](#) [Day 5: 2D Sprites](#)

Hardware sprites free us from the fixed tiles we have looked at so far. These are crucial to any 2D game.





[\[edit\]](#) [Day 6: 2D Hardware Effects](#)

Mosaic, alpha blending, windowing, hardware rotation and scaling, and finally display capture will be covered..

[\[edit\]](#) [Day 7: Sound and Music](#)

This will be the first day in which we need to write code for both processors. First will be a simple direct sound play back then some very simple music playback will be looked upon. Finally we will grab and play back some noise from the microphone.

[\[edit\]](#) [Day 8: 3D Introduction](#)

This will explore the 3D capabilities of the Nintendo DS. This will be broken down into several sub chapters and follow to some degree the Nehe tutorials which inspired the example code.

[\[edit\]](#) [Day 9: Wifi and Networking](#)

An exploration of Sgstairs wifi library as we create a simple client server app and a very simple multiplayer game (pong sounds about my speed)

[\[edit\]](#) [Day 10: A bit more advanced 3D Graphics](#)

Although the Nintendo DS is not the most performant of 3D renderers it does have a plethora of interesting and rather advanced features. This chapter will be devoted to Fog, "Toon" Cel-Shading, Texture formats, Hardware Box tests, and 3D picking.

[\[edit\]](#) [Game Programming 101: Collision Detection](#)

Probably the question that comes up most as people begin to explore 2D game programming. This will cover many approaches to collision detection between player and world as well as player and other movable objects.


[\[edit\]](#) [Game Programming 101: Animation](#)

Second on the list of frequently asked questions is animation and there are many ways to go about it... this guide will attempt to do the topic justice.

[\[edit\]](#) [Tiling, Sprites and Animation](#)

Captain Apathy here, and adding to the tutorials. This one is currently on using the NDS tiling engine, scrolling, working with a 64x64 tiling area, using and animating Sprites. Future changes will include extending past a 64x64 tiling area.

Dev-Scene (c)



Search

Dev-Scene

Home
News
Forum
Blogs
Planet
Library

1

Navigation

Nintendo DS
- NDS Homebrew Catalog
Nintendo Wii
- Wii Homebrew Catalog
Current events
Community portal
Developers/Donations

Personal tools

Log in / create account

Toolbox

Random page
Recent changes
Help
What links here
Upload file
Special pages

Curso Linux Avanzado

www.seas.es

Descubre sus secretos y conviértete en experto!

NDS/Tutorials Day 1

< [NDS](#)

Contents

- [1 Introduction \(feel free to skip this\)](#)
 - [1.1 Me:\(Dovoto\)](#)
 - [1.2 Who Should Read This?](#)
 - [1.3 Requirements](#)
 - [1.4 Why The Nintendo DS?](#)
 - [1.5 Bit o' History](#)
- [2 What You Need To Know](#)
- [3 Installation Of Tools](#)
 - [3.1 For windows](#)
 - [3.2 For Mac OS X](#)
- [4 Setting Up An IDE](#)
- [5 Building A Demo And Testing The Installation](#)
- [6 Emulators](#)
 - [6.1 A Few Good Emulators](#)
- [7 Running The Demo On Hardware](#)
- [8 First Demo](#)

[\[edit\]](#) Introduction (feel free to skip this)

[\[edit\]](#) Me:([Dovoto](#))

I feel it is important to introduce myself before we begin so you understand where I am coming from, where my biases might lie, and most importantly what my limitations are. From this understanding I hope you can look at whatever follows with a critical eye and understand the reason I do things the way I do. If you do so, you will be better prepared to adapt these things to your own projects and abandon them completely when necessary.

I am not quite formally trained in programming. I have taken a few classes and will likely even have a degree in the matter by the time you read this (I am newly a senior at [Oregon State University](#) in Electrical Engineering and Computer Science and will be complete in spring of 2007). I am mainly self-taught and as such have many bad habits, most of which I am not even aware of. My approach to most problems tends to be functional and practical more so than procedural, because of this many of my methodologies do not scale well to large problems.

I have done some small work as a game programmer but none of note and I have several years of professional experience doing non game programming (which is why many of my tools are written for the [.NET platform](#)).

I don't much care for PC programming nor PCs in general and use them mostly as tools for developing my console applications. For the most part I just want my PC to work without my having to know the details and thus I prefer Windows to Linux. I will cover setup on Linux systems as I am not completely inept in that regard but I will not be able to answer many Linux specific questions.

Perhaps the only real credentials I have is that in the past my tutorials have been well received, and I wrote a good portion of the library (libnds) that most people are using for homebrew on the DS.

[\[edit\]](#) Who Should Read This?

This tutorial is geared towards people with a moderate understanding of the [C language](#). This is the only assumption made. These tutorials should prove beneficial to anyone interested in starting game programming and even to those just learning C.

My approach is from the hardware out, and although we will develop some useful code along the way, it is not the intention to produce an [API](#) or library for game programming.

[\[edit\]](#) Requirements

- A PC (just about any will do)
- devkitPRO(Use the automated installer here: <http://goo.gl/0KIMC>)
- A Nintendo DS with a flashcart like Supercard DSTWO: <http://supercard.sc>

OR

- You can get away with just an [emulator](#)

[\[edit\]](#) Why The Nintendo DS?

There are many reasons the DS makes a great game development platform for amateur developers. The first is its amazing array of features. Dual screens, dual processors, touch screen, microphone, wifi, powerful 2D graphics engine (two of them actually), wonderful 3D hardware that is limited in performance but rich in abilities. So many things for homebrew developers to explore make programming the DS very satisfying.

Although one could argue that a PC also has these features, the DS has them all standard. You know exactly which hardware you are going to be developing for and a direct channel to it (no silly operating system always trying to get in the way). And let's face it; it is much cooler to show a friend some code you have running on your handheld Nintendo DS than on some computer screen.

Another reason the DS is a viable platform for us is the homebrew [compiler](#) tools and libraries are solid, well-maintained, and very easy to setup. Development on the DS is now nearly as simple as development on a PC thanks to people like [Dave Murphy](#).

[\[edit\]](#) Bit o' History

Darkfader, Joat, Sgstair, WinterMute, Firefly, Passthrough -> passme -> datel, ndslib -> libnds.
Flashme, wifime, WMB

[\[edit\]](#) What You Need To Know



C or C++ experience would be very beneficial as would any PC game programming knowledge. Some might disagree, but the DS console is a great place to learn game programming, even if your only experience is with a few lines of [QBasic](#) code or a [Java](#) class in high school. If you have never written C/C++ code before you may want to look about for a few Internet tutorials or a decent [C book](#) before adventuring into the console programming world.

If you don't know C and don't feel like finding a good book or taking a class then I urge you to at least wander over to [Bee's Guide to C Programming](#) and give yourself a few days with it. Play around a bit and write some code. Don't get discouraged if you don't get it all... it takes a number of years to get really good with C, but if you find yourself hating the process then perhaps you should look for other hobbies.

[\[edit\]](#) Installation Of Tools

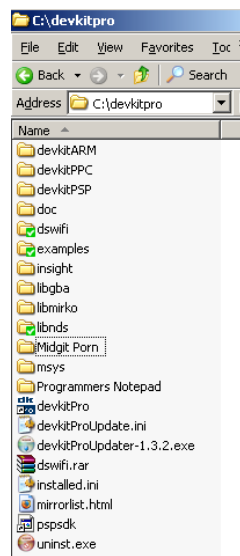
[\[edit\]](#) For windows

You will need a [compiler toolchain](#) to build your DS applications. You will want a C/C++ compiler, a [linker](#), [libraries](#), and [header files](#) as well. The compiler package – a custom build of [GCC](#) – is maintained by [Dave Murphy](#) and can be found at <http://www.devkitpro.org>.

On this site you will find a **devkitPro** package for Windows, Linux, and OSX that contains not only the compilers but also the homebrew header files, libraries, and examples you will need/want. These may come in separate packages for Linux and OSX.

Installation is a snap. If you are on Windows (preferably Windows 2000 or newer) you simply need to run the installer. On Linux or OSX just follow the guide (this involves unzipping the package to a folder of your choice and setting 2 or 3 environment variables). Dave does a much better job explaining the installation than I could so follow [his setup instructions](#).

[DevkitPro.org](#) also has a [FAQ section](#) which addresses any issues you might encounter as well as how to integrate the toolchain with programming environments such as [Visual Studio](#) and [Eclipse](#). For the remainder I will assume you have followed the instructions on that site and have a similar directory structure to the one below:



Be sure to select the examples for download if you are on Windows or grab the examples package directly from the site (it currently says "GBA examples" in the description but actually contains examples for all devkitPro platforms) on [SourceForge.net](#).

My examples are placed in **C:\devkitpro** but you can place them anywhere as long as there are no spaces in the path (so no expanding them on your Desktop and expecting them to work). You'll want the examples if you want to investigate anything beyond the scope of this tutorial, or if you need the whole thing already coded for you if you're running into problems.

[\[edit\]](#) For Mac OS X

I want to give credit to [CoderJoe.net](#) for most of this installation. This setup is for command line users, so I recommend you get comfortable with using Terminal.

Step 1) Get the compiler. You need to be able to compile with the MAKE command. Download and install xcode from [APPLE](#). This should be free once you sign up as a developer.

Step 2) Download the installation files. Get the following files at [sourceforge.net](#) and download them to your home directory. You will need devkitARM, libnds, default_arm7, and the NDS examples (optional)

Step 4) Open Terminal. The Terminal can be found by opening your Finder/Applications/Utilities.

Step 5) Setting up a development path. We will be installing most of your downloaded files to `/usr/local/devkitPRO/`. Type in the following below and your system password:

```
sudo mkdir /usr/local/devkitPRO
```

Step 6) Install the devkitPro binaries. Assuming you've downloaded the devkitARM archive to your home directory, this can be accomplished by changing to the devkitPRO directory you've just created and running. You may have to adjust the file name to match the version you've downloaded.

```
cd /usr/local/devkitPRO/
tar -xjvf ~/devkitARM_r24-osx.tar.bz2
```

the devkitARM tar.bz2 contains a root directory called devkitARM which when compressed will contain the toolchain and a series of helpful binaries located within its "bin" directory.

Step 7) Decompress the libnds source. Next we will decompress libnds. Unlike the devkitARM archive, libnds does not contain its own base directory, so we should create one for it now, and decompress to that directory. Again you may have to adjust the file name to match the version you've downloaded.

```
sudo mkdir /usr/local/devkitPRO/libnds
cd /usr/local/devkitPRO/libnds
tar -xjvf ~/libnds-1.3.1.tar.bz2
```

Now you have the source and tool chain necessary to build libnds into a fully fledged library for your linking pleasure.

Step 8) Set some environment variables. We need to set up some environment variables so the makefiles know where the tool chain and other libraries live.

First we need to set the DEVKITPRO variable to point to our devkitPRO install directory:

```
export DEVKITPRO=/usr/local/devkitPRO
```

Then we need to set DEVKITARM to point to our devkitARM install location

```
export DEVKITARM=${DEVKITPRO}/devkitARM
```

To make things easier, I recommend added the following three lines to the bottom of your ~/.bash_profile file so every interactive shell would have access to these environment variables. Use your favorite text editor and create a file called .bash_profile in your home directory, such as "nano ~/.bash_profile"

```
#Set Variables for DEVKIT_PRO
export DEVKITPRO=/usr/local/devkitPRO
export DEVKITARM=${DEVKITPRO}/devkitARM
```

To exit nano, hit control+x and save. You can logout and log back in to load your bash_profile, or just type the following:

```
source ~/.bash_profile
```

Step 9) Decompress default_arm7. You will need this to compile the NDS examples. Type in the follow and adjust the file to match your version.

```
cd /usr/local/devkitPRO/libnds
tar -xjvf ~/default_arm7-20081210.tar.bz2
```

Congratulations, you now have a successfully built Nintendo DS development tool chain and libraries.

If you'd like to test your toolchain you can extract the example source tarball that I suggested you download in the previous tutorial, and type make in it's root directory. If everything is correctly set up, it should create a Nintendo DS ROM for each example in the examples directory.

The examples may not make much sense now, but make sure to poke around the examples. They're one of the most useful resources a Nintendo DS homebrew developer has at his or her disposal.

[\[edit\]](#) Setting Up An IDE

An IDE is an [integrated development environment](#) and there are many to choose from. They allow you to write, compile and test code as well as manage your project files from a single interface.

Creating an [executable](#) on the NDS is not exactly straight forward. As it has two processors, you actually need one binary for each and these need to be assembled into a standard file format. Fortunately DevkitARM and WinterMute have made this process rather painless with a couple of tools and a very flexible makefile template.

The only IDE I am going to recommend and spend any time on is [Visual Studio Express](#). It is free, awesome, and integrates easily with devkitPro. I recommend you read Dave Murphy's guide [here](#) on how to set it up. These instructions work equally well for the full version of Visual Studio and in the FAQ section of the site there are instructions on setting up other IDEs.

DevkitPro also has [Programmer's Notepad](#) packaged as a small and easy to use IDE if you are afraid of the monster download of Visual Studio Express.

You'll get a better experience from Visual Studio if you add the paths for the libnds header files to your C++ directories. This will make [IntelliSense](#) work (to some extent) with libnds. These options are located under **Tools -> Options -> Projects and Solutions -> VC++ Directories** and the libnds headers are found in path/to/devkitPro/libnds/include - that's C:\devkitPro\libnds\include if you used the installer's default path.

I will assume for the remainder that you were too lazy to actually stop and do any of this setup and are in fact doing your development in [Notepad](#).

[\[edit\]](#) Building A Demo And Testing The Installation

devkitPro supplies about 30 examples with devkitARM that demonstrate how to use the tools and do certain things on the hardware. To test your installation of the toolchain navigate to one of the example folders in a terminal or DOS window and type **make**.

If you are using something other than Windows you may need to grab these examples as a separate download.

If you get no errors you will have a **.nds** file in the root directory (if you do encounter errors check the FAQ section of <http://www.devkitpro.org> and feel free to post your issues on <http://forum.gbadev.org> as well). You can run this .nds file on an emulator or directly on a Nintendo DS.


```

F:\WINDOWS\system32\cmd.exe
C:\>cd devkitpro
C:\devkitpro>cd examples
C:\devkitpro\examples>cd nds
C:\devkitpro\examples\nds>cd Graphics
C:\devkitpro\examples\nds\Graphics>cd 2d
C:\devkitpro\examples\nds\Graphics\2D>cd hello_world
C:\devkitpro\examples\nds\Graphics\2D\hello_world>make
main.cpp
arm-eabi-g++ -MMD -MP -MF /c/devkitpro/examples/nds/Graphics/2D/hello_world/build/main.d -g -Wall -O2 -mcpu=arm9tdmi -mt
-math -mthumb -mthumb-interwork -I/c/devkitpro/examples/nds/Graphics/2D/hello_world/include -I/c/devkitpro/examples/nds/
tPro/libnds/include -I/c/devkitpro/libnds/include -I/c/devkitpro/examples/nds/Graphics/2D/hello_world/build -DARM9 -fno-
mples/nds/Graphics/2D/hello_world/source/main.cpp -o main.o
linking hello_world.elf
built ... hello_world.arm9
Nintendo DS rom tool 1.29 - Jun  8 2006 23:32:11 by Rafael Vuijk (aka DarkFader)
built ... hello_world.nds
C:\devkitpro\examples\nds\Graphics\2D\hello_world>

```

[edit] Emulators

You will want an emulator which can run your DS executable on your PC. This is a requirement if you do not have a DS or have not made the necessary arrangements to run code on it yet.

[edit] A Few Good Emulators

Dualis <http://dualis.1emulation.com/>

NO\$GBA <http://nocash.emubase.de/gba.htm>

Dualis is free and slightly more advanced than NO\$GBA. But the \$15 homebrew version of NO\$GBA has some very impressive and incredibly useful debugging features. I recommend you grab Dualis and if you get serious about homebrew you can grab NO\$GBA another time.

You can find more DS emulators [here](#).

[edit] Running The Demo On Hardware

Most of the fun in developing on a console is in the ability to run your game or demo on the actual system. The DS (like most consoles) has some security features in place to prevent piracy; unfortunately these measures also hinder running your own applications.

There are two pieces of equipment you will need: A storage device which the DS can access and some tool to convince your DS to run your code.

For storage you have many options but I recommend one of the following:

[GBAMP](#) (Gameboy Advanced Movie Player) available from [Lik-Sang.com](#). It is cheap (\$25 plus shipping) and uses [Compact Flash](#) giving you virtually unlimited storage.

[XPORT](#) from <http://www.charmedlabs.com>. Not cheap (\$140 to \$200 depending on the model) and does not have a lot of storage (only 4MB) but if you have any desire to do hardware projects (a DS controlled robot comes to mind) this is probably one of the most interesting devices around. It also has the nice advantage of allowing you to program your DS without taking out and putting back in your flash card.

There are other devices available but most are designed specifically to run pirated GBA or DS games and as such are generally overpriced and often even illegal to import. A quick google of "backup" devices for Nintendo DS should give you an idea as to the range of options.

The other tool you need is one which tricks the DS into running your game even though it is not an official DS game. The [Passme](#) type devices are quite common.

For comparison check [Passme](#)

[edit] First Demo

Our first demo will be a simple one. We will initialize our DS and paint one of the screens a wonderful red color.

First copy the template folder from the examples pack installed with devkitPro. For me the template is installed at **C:\devkitpro\examples\nds\templates\arm9** but your installation directory may vary.

Copy the entire arm9 folder to a new directory (one without any spaces in the path in case your OS allows such things). Name the folder something useful like **demo1** or some such. Once copied, open **main.c** in the source folder and delete everything (you can do this in any text editor). Then type the following:

```

#include <nds.h>
#include <stdio.h>

int main(void)
{
    int i;

    consoleDemoInit();

    videoSetMode(MODE_FB0);

    vramSetBankA(VRAM_A_LCD);

    printf("Hello World!\n");
    printf("www.Drunkencoders.com");

    for(i = 0; i < 256 * 192; i++)
        VRAM_A[i] = RGB15(31,0,0);

    while(1){
        swiWaitForVBlank();
    }

    return 0;
}

```

Alternatively you can grab the tutorial source file which contains the demo1 source code and a pre-built binary from [here](#).

I am not going to go into detail on the workings of this code just yet but a brief description is as follows:

The first real line of code (**consoleDemoInit()**) initializes the console (DOS-like) functionality in the library so **printf** will print text to one screen. Printing is the most complicated thing this demo is doing by far.

Next we set the video mode of the DS. The DS has many video modes with differing uses, all of which will be covered as we go along. For now, just realize that we are putting the DS into **frame buffer** mode. What this is and how it works will be the subject of much of the next chapter.

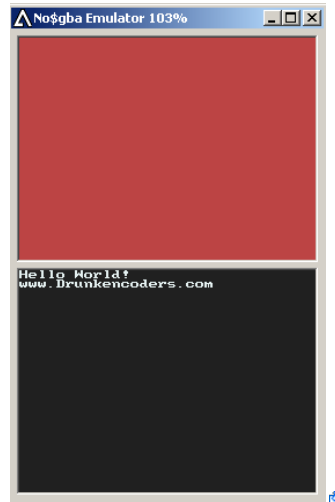
The DS has a very flexible video system that allows memory to be assigned to many locations and for many purposes. The next line of code just assigns the first bank of video memory to the LCD. This both unlocks the memory for easy writing and allows the DS to use it as the direct display memory.

The next lines print the infamous hello world.

Finally we write the color data (red in this case) to video memory. Because the screen is 256*192 we loop through each pixel and set its color to red.

VRAM_A is a pointer to the video memory bank A. RGB15 converts the color value to a 15-bit value the DS can understand (more on how color works in the next chapter).

To build this code simply type **make** from a DOS prompt or terminal window while in the **demo1** directory. Enjoy your achievement by launching it on an emulator or trying it out on hardware.



Okay, once you figure all that out, head over to [NDS Tutorials Day 2](#) and learn about the capabilities of the DS hardware.

Dev-Scene (c)



Dev-Scene
Home
News
Forum
Blogs
Planet
Library



Navigation
Nintendo DS
- NDS Homebrew Catalog
Nintendo Wii
- Wii Homebrew Catalog
Current events
Community portal
Developers/Donations
Personal tools
Log in / create account
Toolbox
Random page
Recent changes
Help
What links here
Upload file
Special pages



NDS/Tutorials Day 2

< [NDS](#)

Contents

- [1 Hardware Overview](#)
 - [1.1 Memory Layout](#)
 - [1.1.1 Main Memory](#)
 - [1.1.2 ARM 7 Fast Ram \(IWRAM\)](#)
 - [1.1.3 ARM 9 Caches](#)
 - [1.1.4 Fast Shared Ram](#)
 - [1.2 Video Ram](#)
 - [1.3 Virtual Video Ram](#)
 - [1.4 Sound Hardware](#)
 - [1.5 Wifi](#)
 - [1.6 Input](#)
 - [1.6.1 Touch Screen](#)
 - [1.6.2 Buttons](#)
 - [1.6.3 Microphone](#)
 - [1.7 Real-Time Clock](#)
- [2 Upgradeable Firmware](#)
- [3 Graphics Overview](#)
 - [3.1 2D](#)
 - [3.2 3D](#)
- [4 Toolchain Explained](#)
 - [4.1 Compiler](#)
 - [4.2 Linker](#)
 - [4.3 Build A Demo The Hard Way](#)
 - [4.4 Conclusion](#)

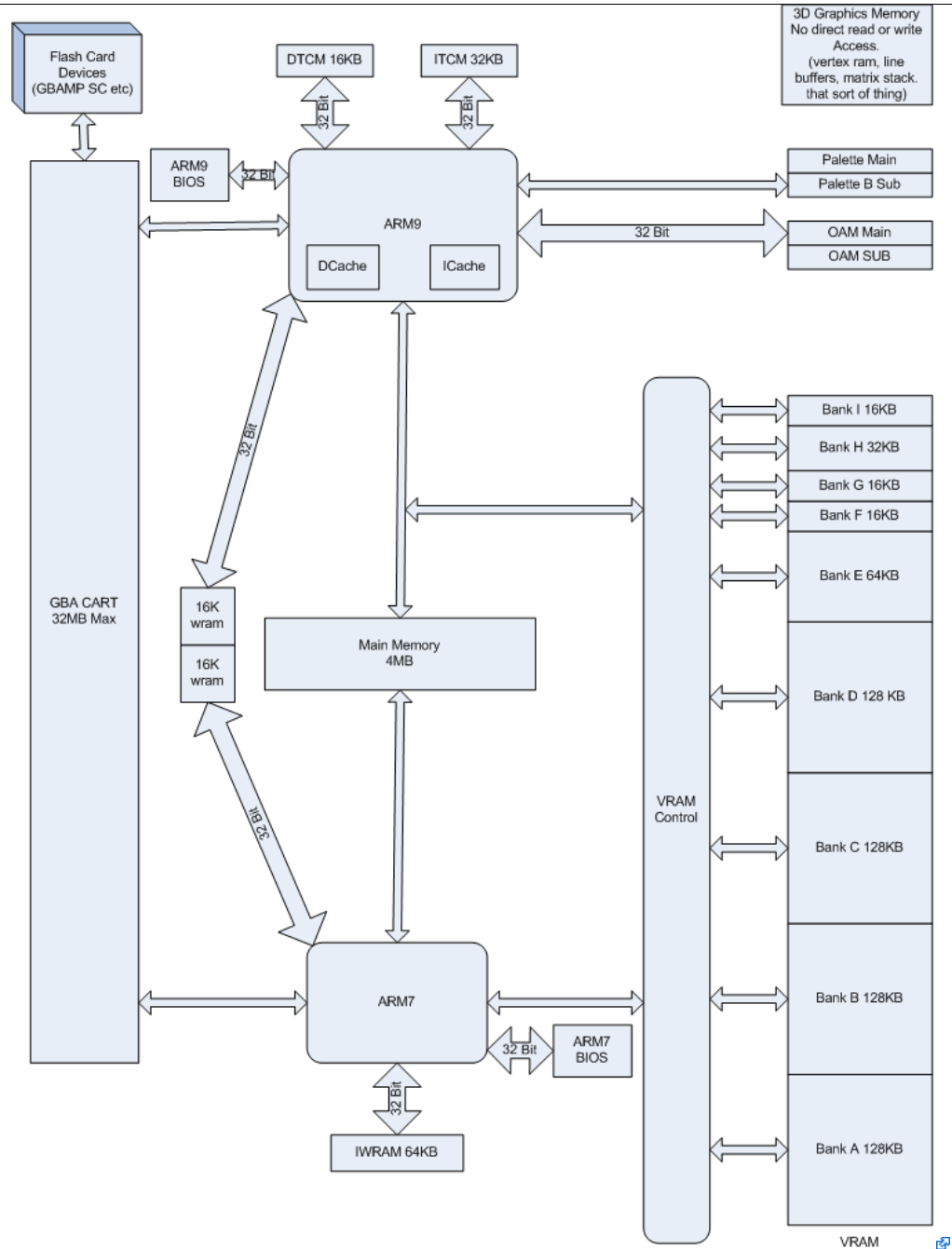
[\[edit\]](#) Hardware Overview

The Nintendo DS is rich in features. It possesses one of the most advanced 2D rendering systems ever seen on a console system, abundant memory resources (many, many times that of the [SNES](#)), dual processors capable of outperforming the [Nintendo 64](#) ([floating point](#) operations aside), integrated wireless networking, a modest 3D system with easy to understand interface, a microphone, and touch pad input.

What follows is a brief description of these features and a foreshadowing of the things you might accomplish with the knowledge gained in this guide.

[\[edit\]](#) Memory Layout

The memory footprint of the Nintendo DS is one of its more intimidating features for newly introduced console programmers. Understanding where memory is and what its uses are is key to getting the most from your applications and in many cases it is key to doing anything at all. Often a picture can be helpful in understanding how memory is arranged.



(Unless otherwise stated the data width for each bus is 16 bits.)

The table below depicts both how these pieces of memory are physically addressed and their actual sizes.

Memory Map

ARM 9				
Name	Start Address	Stop Address	Size	Wait State
Main	0x02000000	0x023FFFFF	4MB	?
BIOS	0xFFFF0000	0xFFFF7FFF	32KB	?
ITCM	0x00000000	0x00007FFF	32KB	?
DTCM	0x0B000000	0x0B003FFF	16KB	?
Shared WRAM Bank 0	0x03000000	0x03003FFF	16KB	?
Shared WRAM Bank 1	0x03004000	0x03007FFF	16KB	?
ARM 7				
Main	0x02000000	0x023FFFFF	4MB	?
BIOS	0x00000000	0x00003FFF	16KB	?
IWRAM	0x03800000	0x0380FFFF	64KB	?
Shared WRAM Bank 0	0x03000000	0x03003FFF	16KB	?
Shared WRAM Bank 1	0x03004000	0x03007FFF	16KB	?
Video RAM				
Main OAM	0x07000000	0x070003FF	1KB	?

Sub OAM	0x07000400	0x070007FF	1KB	?
Main Palette	0x05000000	0x050003FF	1KB	?
Sub Palette	0x05000400	0x050007FF	1KB	?
Bank A	0x06800000	0x0681FFFF	128KB	?
Bank B	0x06820000	0x0683FFFF	128KB	?
Bank C	0x06840000	0x0685FFFF	128KB	?
Bank D	0x06860000	0x0687FFFF	128KB	?
Bank E	0x06880000	0x0688FFFF	64KB	?
Bank F	0x06890000	0x06893FFF	16KB	?
Bank G	0x06894000	0x06897FFF	16KB	?
Bank H	0x06898000	0x0689FFFF	32KB	?
Bank I	0x068A0000	0x068A3FFF	16KB	?
Virtual Video RAM				
Main Background	0x06000000	0x0607FFFF	512KB	?
Sub Background	0x06200000	0x0621FFFF	128KB	
Main Sprite	0x06400000	0x0643FFFF	256KB	?
Sub Sprite	0x06600000	0x0661FFFF	128KB	?

[\[edit\]](#) Main Memory

```
Start Address : 0x0200:0000
End Address   : 0x023F:FFFF
Mirror       : 0x0240:0000
```

Sometimes referred to simply as main memory, it is the 4 megabyte block of RAM which generally holds your ARM9 executable as well as the vast majority of all game data.

Both the ARM7 and the ARM9 can access this memory at any time. Any bus conflicts are delegated to the processor which has priority (the ARM7 by default but changeable via a control register) causing the other processor to wait until the first has finished its operation.

Although it is possible to execute both ARM7 and ARM9 code from main RAM at the same time, devkitPro defaults to placing the ARM7 into the 64K of fast iwram for performance reasons. Official games generally place both ARM7 and ARM9 executables into Main Memory after which the ARM7 copies the majority of its own code to iwram..

[\[edit\]](#) ARM 7 Fast Ram (IWRAM)

```
Start Address : 0x03800000
End Address   : 0x0380FFFF
```

The ARM7 has exclusive access to 64KB of fast 32 bit wide memory. It is this region that contains the ARM7 executable and data. When designing ARM7 code it will be in your interest to keep the binary small.

[\[edit\]](#) ARM 9 Caches

The ARM9 contains both a data cache and an instruction cache. Although the operation of these caches is a bit complex and really out of scope for this document a few things are worth noting.

Main memory is cacheable by default. This means all data and code being accessed from main memory will be stored temporarily in the cache. Because the DMA circuitry and the ARM7 do not have access to the cache often you will get unexpected results if you attempt to DMA from main memory or share data between ARM7 and ARM9 via main memory.

To help utilize the cache effectively the mirror of main memory that begins above 0x02400000 is not cacheable. There are also several functions provided by the library which allow you to flush the data cache and ensure main memory is in sync.

Although the cache adds a certain level of complexity its boost to performance is well worth this small inconvenience.

[\[edit\]](#) Fast Shared Ram

There are two small 16KB banks of fast 32 bit ram that can be assigned to the ARM7 or ARM9. Access to either block by both CPUs at the same time is prohibited. Commonly, both banks will be mapped to the ARM7 as they form a continuous block with ARM7 IWRAM effectively giving the ARM7 96KB of ram.

[\[edit\]](#) Video Ram

The Nintendo DS has nine banks of video memory which may be put to a variety of uses. They can hold the graphics for your sprites, the textures for your 3D space ships, the tiles for your 2D platformer, or a direct map of pixels to render to the screen. Figuring out how to effectively utilize this flexible but limited amount of memory will be one the most challenging endeavors you will face in your first few days of homebrew.

Below is a table of the banks along with a description as to what uses they can be put. You should not worry about understanding this at the moment but it might be handy to bookmark or print out for later use.

[View large intimidating table](#)

[\[edit\]](#) Virtual Video Ram

In order for the 2D systems to function they need RAM. One of the major differences between the 2D graphics engine on the Gameboy Advanced and those on the DS is the DS has almost no memory

dedicated to the 2D system. Instead of setting aside a given amount of video memory for the 2D system it allows you to map the video RAM banks into 2D engine memory space.

This might be a bit difficult to grasp at first. An example might be helpful.

Scenario: You want to render a tile based map to the screen using the main 2D graphics engine.

Because you are an uber Nintendo DS programmer you already know two things:

1. Where the 2D graphics engine expects the map and tile data to be
2. What video RAM banks can be mapped to this "virtual" 2D graphics memory to hold your tiles and map.

Solution: Tell the Nintendo DS to map a [video RAM bank](#) to the right place...in this case we might map video RAM bank A (VRAM_A) to 0x6000000 for use as 2D background memory but we could have chosen another bank (turns out almost all vram banks can be mapped to main background memory).

```
videoSetBankA(VRAM_A_MAIN_BG_0x6000000);
```

We will revisit this topic when we create our first few 2D demos.

This might seem intimidating and difficult at first, but it does offer you a fair bit of flexibility and power over where everything is.

[\[edit\]](#) Sound Hardware

What would a game be without its compliment of blips, bleeps, and chip tunes?

Sound and music are an important piece of any good game and to ensure your next graphical adventure is accompanied by an equally astounding audio experience the DS comes equipped with some impressive hardware.

16 independent audio channels can pump digitized music in 8 bit, 16 bit, or ADPCM format. Each channel has its own frequency, volume, panning, and looping controls allowing for virtually CPU free [MOD](#) quality playback.

[\[edit\]](#) Wifi

What is there to say but that it supports communication with 802.11 standard access points. A full socket library has been implemented which allows porting of PC network code to the DS.

[\[edit\]](#) Input

User input is where the DS excels and is the basis for its much lauded inventive game play. 8 Buttons, 4 direction D-Pad, Touch screen, and microphone make for an interesting mix of possibilities.

[\[edit\]](#) Touch Screen

As I am sure you have already noticed the DS has a touchpad. It is very standard in operation and communicates to the DS via a serial interface to the ARM7. Using the default ARM7 binary from libnds causes the touch screen values to be read once per frame and reported to an area you can reach with the ARM9.

In the next chapter we will cover how to access the touchpad values in code.

[\[edit\]](#) Buttons

Button presses are detected by reading registers on the ARM7 and ARM9. Some buttons are only detectable by the ARM7: the door open-close, the X and Y buttons, and the pen down "button" are all detected on the ARM7 and recorded in shared Main Memory for access by the ARM9.

Our first example demo in the [next chapter](#) will include ways to read the buttons.

[\[edit\]](#) Microphone

Perhaps one of the most interesting additions to the Nintendo DS was the inclusion of a microphone. I have not played with it much to be honest but many interesting ideas come to mind and we will defiantly do a demo or two which captures input from the microphone.

[\[edit\]](#) Real-Time Clock

Being able to know what time it is to the second is pretty handy. Your game can respond differently based on the time of day, you can tell how long it has been since the player last played the game, or it can be used as simple in-game clock. And best of all, reading the date and time is a snap.

[\[edit\]](#) Upgradeable Firmware

The firmware on the Nintendo DS is stored in flash memory. It can be overwritten with custom firmware. For more information on how to achieve this check [here](#)

Upgrading the firmware is useful to developers because it allows you bypass the RSA check when downloading [wifi demos](#). This means we can send our own .nds files to our DS via Wifi instead of just officially signed ones. Also, the hacked firmware will check the GBA slot prior to booting. If it finds an .nds file signature it will execute the code automatically eliminating the need to use a passthrough based device each time you wish to run code from your GBA cart-based flash cart.

If you currently use a [passthrough](#) device to boot your .nds files from the GBA slot, upgrading the firmware is an easy and relatively safe process.

[\[edit\]](#) Graphics Overview

Believe it or not, the Nintendo DS is a very capable very advanced graphics power house. It has an interesting combination of 2D and 3D rendering hardware.

[\[edit\]](#) 2D

The Super Nintendo is considered by many to be the best 2D console ever made (by many I really mean me...nobody else counts). SNES possessed a 16-bit 3.58Mhz processor, 128KB of 8 bit ram and 64KB of video ram. By comparison the Nintendo DS has a 32-bit 66Mhz processor, 4MB of main ram, and 656KB of video ram and that's not counting all its little caches of fast 32 bit ram nor its second 33Mhz processor. This is a very capable machine.

There are two separate graphics cores on the DS. They are referred to as Main and Sub graphics cores. Each core has similar features which vary depending on their mode of operation. The major differences between the cores are as follows:

- The main core has two extra modes which are capable of rendering large bitmaps.
- The main core can give up one of its background layers to the 3D engine.
- The main core can bypass the 2D engine and render from memory to the screen directly in what is often referred to as frame buffer mode.

As alluded to above, each core operates in one of several modes. Below is a table of these modes.

Graphics Modes

Main 2D Engine				
Mode	BG0	BG1	BG2	BG3
Mode 0	Text/3D	Text	Text	Text
Mode 1	Text/3D	Text	Text	Rotation
Mode 2	Text/3D	Text	Rotation	Rotation
Mode 3	Text/3D	Text	Text	Extended
Mode 4	Text/3D	Text	Rotation	Extended
Mode 5	Text/3D	Text	Extended	Extended
Mode 6	3D	-	Large Bitmap	-
Frame Buffer	Direct VRAM display as a bitmap			
Sub 2D Engine				
Mode	BG0	BG1	BG2	BG3
Mode 0	Text	Text	Text	Text
Mode 1	Text	Text	Text	Rotation
Mode 2	Text	Text	Rotation	Rotation
Mode 3	Text	Text	Text	Extended
Mode 4	Text	Text	Rotation	Extended
Mode 5	Text	Text	Extended	Extended

Text backgrounds are general purpose tiled backgrounds; rotation backgrounds are also tiled and can be rotated and scaled. Extended rotation backgrounds support a larger set of tiles (at the expense of a larger map), support more palettes, and can operate in a bitmap mode as well as tiled modes.

These modes and background types will be explored as we go along.

[\[edit\]](#) 3D

It may not possess the poly pushing, texture blending, hardware pixel shading capabilities of the current generation GPUs but where the Nintendo DS lacks in performance and eye candy it excels in features.

Limited to 6144 vertexes per frame (about 2048 triangles or 1536 quads) the 3D system might seem a bit sparse. But given the small screen size a lot can be done with even this small number of available points.

Hardware fog, lighting, and transformation along with non blending texture mapping, toon-shading, and edge anti-aliasing make up a rather impressive set of features for an otherwise lackluster 3D machine.

The 3D core operates as a very OpenGL like state machine allowing much of its functionality to be wrapped in gl compliant code. One major difference between open gl and the DS core is the absence of floating point number support. All operations on the DS are carried out in fixed point precision.

If you want to get a jump on 3D look at the 3D examples included with libnds and the [NeHe](#) tutorials the source code originated from.

[\[edit\]](#) Toolchain Explained

Understanding how your code goes from being a text file to being executed on the Nintendo DS will become very important as your projects progress in complexity. To aid in that understanding we are going to recreate [Demo 1](#) from scratch and build it step by step from the command line.

Before we begin there are a few terms you are likely familiar with but I feel necessary to go on about anyway.

[\[edit\]](#) Compiler

The compiler is the first tool you pass your C source through. It is responsible for interpreting that code and translating it into machine based assembly language. From there the assembly language is further reduced into its binary machine code equivalent by another tool known as the assembler (which the compiler will call for you).

The output of the compiler is generally not executable but is instead in what is known as an object file format. Although the instructions have been translated to machine code binary, the decisions about where that code and associated data is to be physically placed in memory have been left undecided.

[\[edit\]](#) [\[Linker\]](#)

The tool used to combine the object files and determine physical addressing such that functions from a multitude of object files can operate in a coherent fashion is called the linker. By passing your object files to the linker you can produce an executable binary file.

Because the linker is responsible for determining where things should be placed physically within the NDS system it must be told a fair amount of information about the memory layout of the DS. This description is located in a linkscript file which describes both the memory layout of the DS and the way in which we want the different regions of our code to map to it. Here is a small piece of the devkitARM default linkscript for the arm9 (yes there is a separate one for the arm7 since it has a different memory layout).

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)

MEMORY {
    rom : ORIGIN = 0x08000000, LENGTH = 32M
    ewram : ORIGIN = 0x02000000, LENGTH = 4M - 4k
    dtcm : ORIGIN = 0x0b000000, LENGTH = 16K
    itcm : ORIGIN = 0x01000000, LENGTH = 32K
}

__itcm_start = ORIGIN(itcm);
__ewram_end = ORIGIN(ewram) + LENGTH(ewram);
__eheap_end = ORIGIN(ewram) + LENGTH(ewram);
__dtcm_start = ORIGIN(dtcm);
__dtcm_top = ORIGIN(dtcm) + LENGTH(dtcm);
__irq_flags = __dtcm_top - 0x08;
__irq_vector = __dtcm_top - 0x04;

__sp_svc = __dtcm_top - 0x100;
__sp_irq = __sp_svc - 0x100;
__sp_usr = __sp_irq - 0x100;
```

There is much more to this script; most of which is utterly incomprehensible and any of which can have extremely difficult to understand consequences if muddled with. It is good to understand these scripts do exist and their general purpose, but actually editing them is well beyond the scope of this document.

The snippet above was chosen because it is somewhat comprehensible; it describes the 4 primary memory regions of the DS that will be used to contain code and data.

- rom is the GBA cartridge space; it is 32MB in size and begins at an absolute address of 0x8000000
- ewram is external working ram and is the slow 4MB of main memory for the DS.
- dtcm stands for data tightly coupled memory and is a special area of memory on the ARM9 intended for use as fast data memory. The standard link script places the stack in this area. dtcm is a mere 16k so be careful with those local variables.
- itcm stands for instruction tightly coupled memory and is another special area intended for use as fast instruction memory. This area is 32k and may be used for small functions which need to be fast. libnds uses this area for the interrupt dispatcher.

The rest of the script file deals with mapping of your code to these regions (read only data, code, variables, stack, global data, constructors for C++ stuff, etc). It does this using an obscure and rather intimidating expression type language that I will not even pretend to understand completely. Fortunately a few people like [Jason Wilkins](#), [Jeff Frohwein](#), and most recently (and most successfully) [Dave Murphy](#) have done the grunt work; what was once something which caused countless "interesting" issues can now be relied upon confidently to just work.

Generally speaking there is no need to worry about the linkerscript unless you have some pressing need to change where things go in memory. Everything except the stack goes in main memory by default so all you have to worry about is fitting everything into 4meg.

[\[edit\]](#) [Build A Demo The Hard Way](#)

To sum things up you first compile your source files into object files and then link them into a binary executable. Normally this would be the end of the process but, alas, our little DS is a bit more complex as it has not one but two processors and each need their own separate binary executable.

What are we to do? Create both of course. The process is identical with the only difference being we use the linkscript for ARM7 when linking the arm7 object files.

The final step in the process is packaging the binaries into a single file that we can then load onto the DS. Fortunately [Darkfader](#) has created a tool for us (included in the [devkitPro](#) package) which does just this. Official NDS game carts happen to use a file format which suites our needs rather well. This format includes a small header with an embedded logo and a short description of the .nds file in several languages, followed eventually by the executable binaries for the arm7 and arm9. The included logo and description text shows up when you boot a game card from the firmware or start to download one over wireless multiboot. (After a bit of investigation it turns out the logo is not actually embedded in the header but exists separately...it is however pointed to by the header)

Now that we have a feel for the process let us create a full .nds file from the command line (so we can confidently never do it again).

Okay...one quick thing to mention. When we created demo1 you may have noticed we had no arm7 code in the project. The reason we are able to get away with this is there is an arm7 executable binary already present in the devkitPro package that you can include by default. This binary performs some very basic things such as read the touch pad and real time clock as well as some very simple sound playback. For anything more advanced you will be providing your own arm7 code or at least modifying the supplied code.

Now...on to the demo. Follow the instructions from [Day 1](#) on building your first demo with the following exception: Instead of grabbing the arm9 template grab the template labeled **combined**. Within you will find an **arm9** folder and an **arm7** folder. Replace the code in the **template.c** inside the arm9 folder with the demo1 code (or leave it the same as it does not really matter what we

compile for this exercise).

You will notice a makefile inside the **combined** directory. If you were to navigate to that directory in a Dos/terminal window (try this now in fact), you could simply type make and the scripts contained inside the makefile would do all the steps we are about to do by hand.

Here are the commands needed to build the nds file from the command line.

```
F:\WINDOWS\system32\cmd.exe
C:\dev\DS\tutorial\demo1_hard>cd arm9\source
C:\dev\DS\tutorial\demo1_hard\arm9\source>arm-eabi-gcc -c template.c -I C:/devkitpro/libnds/include -DARM9
C:\dev\DS\tutorial\demo1_hard\arm9\source>arm-eabi-gcc -o arm9.elf template.o -LC:/devkitpro/libnds/lib -lnds9 -specs=ds_arm9.specs
C:\dev\DS\tutorial\demo1_hard\arm9\source>arm-eabi-objcopy -O binary arm9.elf arm9.bin
C:\dev\DS\tutorial\demo1_hard\arm9\source>ls
arm9.bin arm9.elf template.c template.o
C:\dev\DS\tutorial\demo1_hard\arm9\source>cd ../../arm7/source
C:\dev\DS\tutorial\demo1_hard\arm7\source>arm-eabi-gcc -c template.c -I C:/devkitpro/libnds/include -DARM7
C:\dev\DS\tutorial\demo1_hard\arm7\source>arm-eabi-gcc -o arm7.elf template.o -LC:/devkitpro/libnds/lib -lnds7 -specs=ds_arm7.specs
C:\dev\DS\tutorial\demo1_hard\arm7\source>arm-eabi-objcopy -O binary arm7.elf arm7.bin
C:\dev\DS\tutorial\demo1_hard\arm7\source>ls
arm7.bin arm7.elf template.c template.o
C:\dev\DS\tutorial\demo1_hard\arm7\source>cd ../../
C:\dev\DS\tutorial\demo1_hard>ndstool -7 arm7/source/arm7.bin -9 arm9/source/arm9.bin -c demo1.nds
Nintendo DS rom tool 1.29 - Jun  8 2006 23:32:11 by Rafael Uuijk (aka DarkFader)
C:\dev\DS\tutorial\demo1_hard>ls
makefile arm7 arm9 demo1.nds
```

Before we explain what is going on it is best to take a moment and absorb just how much compiling from the command line sucks...

Good, now that we know let us figure out what we did.

What is not shown is me setting my path so that the devkitARM tools could be found. On windows this is simply:

PATH=c:/devkitpro/devkitarm/bin

First we invoked the compiler on the arm9 template.c file. This translated it into an object file (template.o). We passed it the file as an argument as well as the include directory for the libnds header files we are using. Because libnds does different things depending on if you are constructing an arm7 or an arm9 binary we must define ARM9 with the -D option.

Next we link the file into an executable (.elf file). We pass the object file as an argument, we tell it we would like to include libnds (-lnds) and then we tell it where to look for the linkscript and what default libraries to use (-specs=ds_arm9.specs).

Because our loader does not handle the .elf format very easily we strip away all that extra info using objcopy. This leaves with a nice flat binary for execution. (the .elf file contains debug information and other things which are useful; you will need the .elf file to use the remote debugger or the source level debugger in no\$gba if you can afford such luxuries).

This entire process is repeated for the arm7 leaving us with an arm7.bin and an arm9.bin. We next combine these binaries into an .nds file using ndstool.

If there is anything you should take from all this it is the convenience of the template makefiles. All you do is drop .c/.cpp/.s files into the source directories for the processors, .h files into the include directories and .bin data files into the data directory (more on this when we talk about getting your data into your program) and type **make**. The script will automate this entire process in an efficient and easy to use fashion which reduces the entire painful process of above into the single command: **make**.

[\[edit\]](#) Conclusion

Today we took a short peek at the capabilities of the DS and learned a bit more detail on the process of creating executable from code. Much of the hardware descriptions were intentionally vague as the real detail will come in the following chapters.

Tomorrow we will begin looking at the hardware in detail when we explore raster graphics.

Dev-Scene (c)

DWG to JPG, TIF, BMP, GIF

www.anydwg.com

Batch convert DWG/DXF to JPG, TIF, BMP, GIF, PNG, PCX, TGA, EMF, WMF.



DWG to JPG, TIF, BMP, GIF

www.anydwg.com

Batch convert DWG/DXF to JPG, TIF, BMP, GIF, PNG, PCX, TGA, EMF, WMF.

NDS/Tutorials Day 3

< [NDS](#)

Contents

- [1 What is a register](#)
 - [1.1 Twiddling Bits](#)
 - [1.1.1 Numbering Systems](#)
 - [1.1.2 Bitwise Operations](#)
- [2 Talking to the keypad](#)
- [3 Frame buffer...finally](#)
 - [3.1 Display Control](#)
 - [3.2 VRAM Control](#)
 - [3.3 DS Color Formats](#)
 - [3.4 Frame buffer 101](#)
 - [3.4.1 Pixels and things](#)
- [4 Touching things](#)
- [5 Bitmap Graphics Modes](#)
 - [5.1 Bitmap on the sub display](#)
 - [5.2 Background struct](#)
 - [5.3 Working With Graphics Files](#)
- [6 The Double Buffer](#)
- [7 Raster 101](#)
 - [7.1 Bresenham Lines](#)
 - [7.2 Blitting Things](#)
 - [7.3 Drawing Pictures](#)
 - [7.4 Polygons](#)
 - [7.5 Circles](#)

[[edit](#)] What is a register

The first concept we must get under our belts (and the only one that really matters at the moment) is the concept of memory mapped registers.

Now, I am sure you are aware that the DS has several different chips inside responsible for creating the images and sounds that accompany most games. There is sound hardware responsible for producing annoying chip tunes, the video hardware which puts all your convoluted data together in a nice and pretty display, the memory chips that hold the data for our programs, and the CPUs which are in overall control of the whole shebang (in all honestly many of these "chips" are actually just parts of one large integrated circuit and not really separate chips).

When you are writing c code to describe the events in your game you are directly controlling the DS processors. But, the CPUs do not work alone and generally we like to have some control over what the rest of the system is doing. The method by which this is accomplished is the use of hardware registers.

Beginning at the memory address of 0x4000000 and running for quite some many bytes is the memory mapped register space. What this means is that if I write some arbitrary value to the address 0x4000000 then I will be writing to a register that will have some effect on how the system renders (or fails to render) my video game. Understanding registers is vital in understanding console development.

Using memory mapped registers requires some knowledge I hope you already possess (if you know any c at all) and that is: how to write data to any specific address.

Hopefully you remember the concept of a pointer but, if not, that's okay because I will cover it briefly. Recall that a pointer is a type of variable that holds not data but, instead, the address of some type of data. In this example, let us say we know (which we do) that the register at 0x4000000 was 32 bits in length and controls the display and we hence call it DISPLAY_CR; we might use code like the following to write to this address:

```
unsigned int *DISPLAY_CR = (unsigned int *) 0x4000000;
*DISPLAY_CR = somevalue;
```

Now, this would work just fine but there are some issues with this code. First, because these are hardware registers it is possible (even likely) that the values stored at these addresses will be changed by the hardware directly. This is something the compiler needs to know else it will try to optimize our





code and we would miss these changes. The way we tell the compiler that variables change outside of the c code is to declare them volatile.

The other issue is we are using a variable (RAM) to store a constant. We would be much better served if we just used a `#define`...this also allows us to dereference the register in its declaration and makes writing to it a bit simpler. Here is the new, more proper code.

```
#define DISPLAY_CR (*(volatile unsigned int *) 0x4000000)
DISPLAY_CR = somevalue;
```

Notice there is no longer any need to dereference the register prior to use as it is implicit in the definition. Also, this code uses no space in memory for the pointer as it is just a constant (of course the compiler being as smart as it is even had you used a variable it likely would have been smart enough to optimize and the result would have been the same).

I hope this concept is clear to you; about 30% of the following pages are nothing but descriptions and examples of how to use the hardware registers to control the many features of the DS

[\[edit\]](#) Twiddling Bits

It will rapidly become apparent that controlling hardware via registers will require an understanding of how to target specific bits inside the register. That is, we must be able to set or clear some bits in a register while leaving the rest untouched. Even though this is rather simple and talked about in many other places, it is important enough that we must waste the time of the 90% of readers who already know it in order to ensure the 10% who have no clue are not left in the dust.

[\[edit\]](#) Numbering Systems

Most of you can begin at 0 and count all the way to 9 (a feat by anyone's reckoning). If so, you probably realize there are in fact 10 unique digits you will come across in this endeavor. Oddly enough, the numbering system we use on a daily basis is called base 10 (in academic elitist societies you may also hear it referred to as Decimal).

First let us review an interesting detail about base 10 that you already know: the significance of the placement of the digits in any number. If the digit is on the right-hand side of a number it is generally weighted less than those digits appearing on the left to such a degree one might call it exponential. For instance consider the following examples of decimal numbers:

```
1) 1    = 1 * 10^0
2) 10   = 1 * 10^1
3) 100  = 1 * 10^2
4) 1000 = 1 * 10^3
```

This ringing any bells? You realize quite readily that the weight of the digit in question is equal to 10 to the power of the place of that digit in the number.

Unfortunately for us, computers do not use the same numbering system we do. The reason for this is a simple one. They only know 2 digits. Why is this unfortunate for us? It turns out that 90 percent of the time we can ignore the fact that computers use Base 2 because the compilers and tools we use automatically convert our base 10 numbers to binary for us. But, sometimes an understanding of the computer numbering system is crucial to coding.

The binary system works much in the same way as does our own decimal system with the exception that it has fewer digits and instead of weighting the value of a digit by 10 to the power of the place we instead weight it by 2 to the power of the place. For instance here are the same numbers as above in base 2 and their decimal equivalents.

```
1) 1    = 1 * 2^0 = 1
2) 10   = 1 * 2^1 = 2
3) 100  = 1 * 2^2 = 4
4) 1000 = 1 * 2^3 = 8
```

You might notice writing the value 8 in binary requires 4 digits, this may not seem an issue off hand but as numbers increase writing binary rapidly becomes cumbersome. A string of ones and zeros is prone to error and very difficult to read. To combat this, a new base was developed making manipulation of numbers on computers easier to handle. That numbering system is base 16; often referred to as Hexadecimal or Hex.

Hex numbering uses the digits 0 – 9 and A – F and you end up with numbers which look like the following:

```
4d6172696f
4b69726279
4c75696769
5a656c6461
```

At first you may be wondering why the hell you would ever go through such seeming pain to write numbers in such a way. Well it turns out the conversion between Hex and Binary is very simple. Because there are 16 digits in hex (a power of 2 mind you) you can represent each hex digit with 4 binary numbers. All you need do is become familiar with counting in binary from 0 to 15 to convert back and forth.

A few examples might be in order:

Take the binary number 110100100101010101010001 To convert this number to decimal would require you to look at each bit and add 2 to the power of the place if there is a one in that position. This number in decimal becomes:

```
1 + 16 + 64 + 256 + 1024 + .... and then you give up and put it in your calculator and get: 220
```

To do the same conversion to Hex you break the number into 4 bit nibbles and convert so it becomes:

```
1101 0010 0101 0101 0101 0101 0001
D    2    5    5    5    5    1    = D255551
```

Now that you believe that hex to binary might be simpler than binary to decimal you may still be a bit unclear on why we don't just write everything in decimal and let the compiler figure it out (because it will). As we said before computers are binary in nature and as such hardware is controlled by specific bits at specific addresses. Because we need to set specific bits in the binary number we must at some point think of the number in binary. This will become more apparent as we actually set those bits.

Another good reason is bit boundaries play a significant role in memory addressing on most systems. For instance DS video memory for one of the graphics units begins on a boundary. The address of this memory can be written in hex as 6000000, that same address in decimal is 100,663,296. Although you could store the value as a pointer and write to video memory using either numbering system the hex value is much easier to remember and much easier on the eyes.

As a final note in C hexadecimal numbers are denoted with an 0x at the beginning of the number.

```
0x3FF
0x3ff
0x60000000
```

Notice case is not an issue.

[\[edit\]](#) Bitwise Operations

Being able to 'and' and 'or' bits together is important when attempting to enable certain features of hardware. Below is a summation of how bitwise operators work in C and how you might use them.

AND operator: '&'

Different than the logical and '&&' the single ampersand denotes two operands should be 'anded' together. Each number will be compared against the other bit by bit and if either number has a 0 in that bit position a 0 will be stored in the result.

AND is useful for checking the status of bits. For instance to see if the first bit of a register is set simply AND the register with the value 1. The result can only be 1 if the first bit of the register is also 1.

```
Register value = 1101 0100 1101 1101
                & 0000 0000 0000 0001
-----
Result = 0000 0000 0000 0001
```

It is also good for clearing bits. If you want the lower 8 bits of a number cleared to 0 simply AND the value with a number that has all the other bits set to 1. This is where hex comes in very handy.

```
Original Number      = 1101 0100 1101 1101
                    & 1111 1111 0000 0000 = 0xFF00
-----
Result with low 8 bits clear 1101 0100 0000 0000
```

OR operator: '|'

A bit by bit comparison which causes the result bits to be set if either of the bits in the arguments are set.

OR is good for setting bits. To set a bit, simply OR the register with a value that has that bit and that bit only set. Here is an example of setting bit 9 of a 16 bit number (bits are numbered right to left beginning at 0)

```
Old Value =          1101 0100 1101 1101
                | 0000 0010 0000 0000
-----
New value with bit 9 set 1101 0110 1101 1101
```

XOR Operator: '^'

XOR is great for flipping between states. In the above example, if an XOR was used in the place of an

OR then the bit would be set if it was clear and it would be cleared if it were set. This is very useful anytime you need certain bits to alternate states every frame.

NOT Operator: '~'

NOT inverts the bits in a number rendering all 1s to 0s and all 0s to 1s. This is very useful for clearing bits. For instance if you know the main graphics engine will render to the top LCD when Bit 15 of the power control register is set to 1 and on the bottom when set to 0. We can 'or' the power control register with bit 15 to set it and we can 'and' the register with bit 15 inverted to clear it. Here is a snippet from libnds.

```

    /// Forces the main core to display on the top.
    static inline void lcdMainOnTop(void) { POWER_CR |= POWER_SWAP_LCDS; }

    /// Forces the main core to display on the bottom.
    static inline void lcdMainOnBottom(void) { POWER_CR &= ~POWER_SWAP_LCDS; }

```

In this case POWER_CR is defined as a pointer to the power control register and POWER_SWAP_LCDS is defined as bit 15.

The final bitwise operations to talk about are the shift operations '>>' and '<<'. When used these operators cause the binary number to be shifted to the left or right the specified number of places:

```

0101011 << 1 = 1010110
0101011 << 2 = 0101100
0101011 << 3 = 1011000

```

```

0101011 >> 1 = 0010101
0101011 >> 2 = 0001010
0101011 >> 3 = 0000101

```

If you will recall the weight of a digit is proportional to the base raised to the power of its position in the number. When we shift numbers to the right '>>' we are reducing the weight of the digits effectively dividing the number by 2^n where n is the amount we shifted. Similarly a left shift '<<' will multiply by a power of 2.

Often it is beneficial to use shift operators when division and multiplication are required as they execute more quickly. Do not get carried away though as they are less readable and the compiler will convert multiplications to shifts when ever possible for you.

The shift operator has other uses and plays a big role in fixed point arithmetic which we will cover shortly.

[\[edit\]](#) Talking to the keypad

It is difficult to do any interesting yet simple demo programs without understanding how to read user input. Fortunately for us, getting the state of the DS keys is exceedingly simple (if you understood the above discussion that is).

The state of each button is stored as a bit in memory mapped register space. To know if a key is pressed or released we just read the state of a specific bit. All we need to process the keys is the knowledge of where these values are stored.

Let us write our first real demo that checks for key presses and prints their state on the screen. Before we get to the code let us look at the main register used for key state on the DS.

(insert key pad register description here).

One thing you might note is the glaring absence of the X and Y keys. A bit further down we will demonstrate a more refined approach to handling input and introduce the functionality built into libnds and see if we can't find those missing buttons. For now let us get our first real demo out of the way.

```

#include <nds.h>
#include <stdio.h>

int main(void)
{
    consoleDemoInit();

    while(1)
    {
        if(REG_KEYINPUT & KEY_A)
            printf("Key A is released");
        else
            printf("Key A is pressed");

        swiWaitForVBlank();

        consoleClear();
    }

    return 0;
}

```

Much of this code is as you have seen before. We initialize the print console so printf prints to the sub

screen using default settings. The next two lines initialize the libnds interrupt handler and enable the vblank interrupt. This is necessary for something we do in the main loop but it is a bit out of scope for this first day. We will talk at much greater length about interrupts (IRQs) on a later day.

The main loop just checks the KEY_A bit of the input register. This happens to be bit 0 and when the key is pressed that bit will be clear. This is all there is to checking for key presses on the DS.

The next two lines of code force the DS to wait until the screen is done drawing and then clears the screen. This prevents some nasty looking text flickering. Again, understanding this bit of code requires some knowledge of interrupts which will have to wait.



Now that simple reading of the key presses has been covered it is time to consider a bit more advanced needs...such as what about those X and Y buttons?

Unfortunately the designers of the DS were a bit lazy and stole the GBA input hardware; it seems our wonderful GBA input was a bit lacking in the number of buttons available to the user. What this resulted in is that we can read the A, B, Up, Down, Left, Right, Start, Select, and the Left and Right shoulder buttons from one place but the X, and Y buttons are in a different register...and can't even be read at all by the main CPU!. It seems in our very first foray into DS programming we must face the complexities of a dual processor system.

The solution is to read the keypad from the ARM 7 (which can read the X and Y buttons plus the hinge "button" on the DS lid and the pen state for the touch pad) and put the results someplace readable by the ARM 9.

If you are like me then this code seems a bit awkward. It would be nice if we had some way of wrapping all these bits into a single location to simplify the reading of key presses. Libnds provides just such a wrapper. Let us see how we would do the same code using the libnds wrapper then take a closer look at what the wrapper is doing.

```
#include <nds.h>
#include <stdio.h>

int main(void)
{
    consoleDemoInit();

    while(1)
    {
        scanKeys();
        int held = keysHeld();

        if( held & KEY_A)
            printf("Key A is pressed\n");
        else
            printf("Key A is released\n");

        if( held & KEY_X)
            printf("Key X is pressed\n");
        else
            printf("Key X is released\n");

        if( held & KEY_TOUCH)
            printf("Touch pad is touched\n");
        else
            printf("Touch pad is not touched\n");

        swiWaitForVBlank();

        consoleClear();
    }

    return 0;
}
```

Two things to note in this new demo is the use of a scanKeys() call every frame and the change to positive logic: Now the bits are set if the key is pressed and clear if they are released. Along with keysHeld() is a keysDown() which will only be true if the key was pressed since the last time you checked (ie it will return true once but unless the player releases the key and presses it again it will return false).

Some other useful functions are keysUp() which returns the released keys and keysDownRepeat which returns true after a certain delay (measured in number of scanKey() calls) even if the keys have been held down. Check the documentation for input.h for more information on how to use these other functions.

Basically the way scanKeys works is to combine the bits from REG_KEYINPUT and IPC->buttons and apply a little bit of state tracking to determine which have been pressed since the last call. Here is an excerpt from the key handling code in libnds:

```

#define KEYS_CUR (( (~REG_KEYINPUT)&0x3ff) | ((~IPC->buttons)&3)<<10) | ((~IPC->buttons)<<6)

void scanKeys(void) {
    keysold = keys;
    keys = KEYS_CUR;

    ///..some code for handling key repeats
}

uint32 keysHeld(void) {
    return keys;
}

uint32 keysDown(void) {
    return (keys ^ keysold) & keys;
}

```

The statement at the beginning does most of the work by negating the register state and masking out / recombining the two sources of key state. If you have not noticed by now you will need to have a decent understanding of bit operations to work with register controlled hardware.

[\[edit\]](#) Frame buffer...finally

It is nice to finally get to graphics programming...I don't know about you but two full days of fluff is about all I can take.

If you have actually followed along with the subjects and code presented so far you will find doing frame buffer graphics on the DS is surprisingly simple. All we need do is put the DS into frame buffer mode and begin writing images to the screen.

If you recall from yesterday's topic the DS supports many graphics modes with the main screen supporting a simple frame buffer. It is this mode we will turn to first as it is very easy to set up and even easier to use.

I figured I would start this chapter with some code and use that to explain the frame buffer mode.

```

#include <nds.h>
#include <stdio.h>

int main(void)
{
    int i;

    //initialize the DS Dos-like functionality
    consoleDemoInit();

    //set frame buffer mode 0
    videoSetMode(MODE_FB0);

    //enable VRAM A for writing by the cpu and use
    //as a framebuffer by video hardware
    vramSetBankA(VRAM_A_LCD);

    while(1)
    {
        u16 color = RGB15(31,0,0); //red

        scanKeys();
        int held = keysHeld();

        if(held & KEY_A)
            color = RGB15(0,31,0); //green

        if (held & KEY_X)
            color = RGB15(0,0,31); //blue

        swiWaitForVBlank();

        //fill video memory with the chosen color
        for(i = 0; i < 256*192; i++)
            VRAM_A[i] = color;
    }

    return 0;
}

```

This code is very similar to the code for the day 1 introduction demo. As before we enable interrupts and turn on the vblank interrupt. Next we set the video mode to a frame buffer mode which uses the first video ram bank. We then set the first VRAM bank to be writable by the CPU and to act as a buffer for the LCD (recall the first VRAM bank is VRAM_A).

The main loop creates a color as a 16 bit unsigned short integer and sets its value to red. If A or X are pressed then the color is altered to be green or blue respectively. Finally we wait for the screen draw to finish and fill VRAM_A with the selected color.

Now that we have a base understanding of the code we need to get to the details, namely:

- How do videoSetMode and vramSetBankx work?
- How does the DS treat color?

- What the hell is a frame buffer and how do I write pixels to it?

These are the questions we will now explore.

[\[edit\]](#) Display Control

(todo: add description of display control register here)

[\[edit\]](#) VRAM Control

(todo: add some guidance on VRAM control...or delete this section and move it to the next chapter)

[\[edit\]](#) DS Color Formats

The DS has several ways in which it represents color. These ways generally fall into two categories: Paletted and Direct.

Direct color means the value directly controls the intensity of red, green, and blue that is fed to the pixel. There are technically two direct color formats used by the DS but you will see the variance between the two is minimal.

Direct color uses 5 bits to represent how bright each color component can be (red, green, and blue). We refer to this format as 555 or sometimes 15 bit color. If you recall from our discussion on binary numbers, 5 bits amounts to 32 levels of intensity for each of the three colors.

To describe a color in this format we need to combine our desired values of red, green, and blue to form one 15-bit number. The color components are stored as follows:

xBBBBBGGGGRRRRR

This can be translated as the least significant 5 bits hold the red component, the next 5 bits hold the green and the remaining 5 hold the blue. We refer to this as BGR format. This would be a good time to flex our bit twiddling muscles and see if we cant define some colors.

To do this we specify an intensity for each of the 3 components and then shift the values into the correct place, finally we OR them all together to get our 15 bit value.

```
int red = 31;
int blue = 0;
int green = 0;

unsigned short int color_red = (blue << 10) | (green << 5) | red;
```

Simple enough? Since we don't normally want to concern ourselves with this detail we use the macro provided by libnds (or write our own) which is depicted below:

```
#define RGB15(r,g,b) ((r)|((g)<<5)|((b)<<10))

//to use:

unsigned short int color = RGB15(red, green, blue);
```

Taking a short break from theory you should now glance up to the demo we just wrote. You will notice I used this macro to paint the screen red. It should be apparent at this point that the frame buffer utilizes 15 bit Direct color format.

The other Direct color format is nearly identical to the one just discussed. The only difference is in the most significant bit (depicted as the 'x' in xBBBBBGGGGRRRRR above). When utilizing this format this bit is known as the 'alpha' bit and when set to 0 will prevent the color from appearing onscreen. Most 16 bit graphics operations on the DS utilize this "alpha" bit to determine transparency of the rendered pixel.

When we move on to Direct color bitmap modes you will quickly discover not setting this alpha bit will result in nothing on screen. Recall to set this bit requires some more bit operations:

```
//set alpha
color = color | (1<<15);

//clear alpha
color = color & ~(1<<15);
```

Although direct color formats provide a wide range of colors they have a serious drawback: They take up 16 bits for each pixel. Although the DS has an abundant amount of memory and CPU power compared to early 2D systems it still pales in comparison to most modern machines. You will be surprised how rapidly you will fill memory with a 16 bit image or how much stress you will place on the hardware if you attempt to blit large 16 bit images.

To alleviate this the DS utilizes many space saving tricks. The most prevalent is the use of paletted colors. Instead of specifying color components directly we instead build a table of colors and specify an index into this table.

Let us say we have a 256 color table (we will refer to this table as the "palette") which contains 256 direct color values. We can then set pixels onscreen to these values by specifying an index. Because the table is small we would only need an 8 bit index to describe the pixel color...a savings of 50%!

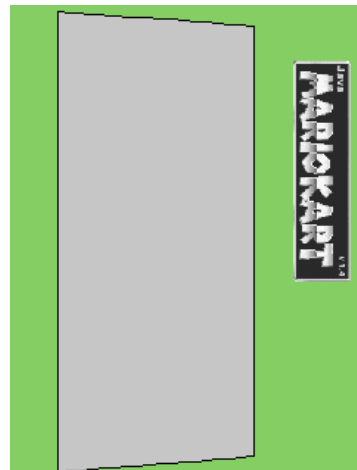
The DS supports 8 bit indexed palettes as well as 4 bit and we will figure out the mechanics of their use as we proceed.

[\[edit\]](#) Frame buffer 101

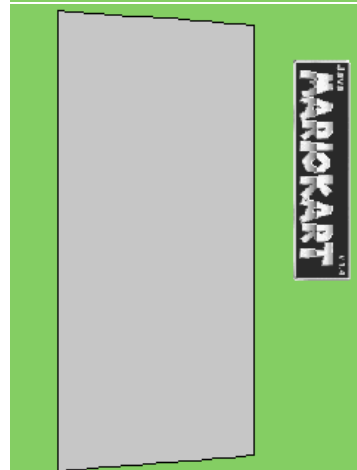
A frame buffer can be described as a direct map of memory values to onscreen colors. By simply writing the correct color value into memory we can set a pixel as we see fit.

Framebuffer memory can be completely described by three things: The address at which it begins, the color format of the pixels, and the number of pixels per horizontal line.

The memory for a frame buffer is a single linear map such that the first W entries correspond to the top row of pixels on the screen (W in this case is the "width" of the buffer). The next row of pixels follows and occupies entries W to $2*W - 1$.



[\[image\]](#) (image of linear memory)



[\[image\]](#) (image of memory as it represents 2D space)

[\[edit\]](#) Pixels and things

To accurately place pixels onscreen we must have some idea how to specify location. This is normally done using a modified Cartesian coordinate system where we specify how many pixels from the left and how many pixels from the top we wish our value to be placed.

[Image:Coordinate sys.png](#)

The distance from the left hand side of the screen is usually referred to as the X coordinate of the pixel and the distance from the top is the Y coordinate. (Those of you who are math whizzes might see the disparity between Cartesian coordinates as Y usually is measured from the bottom up...live with it).

To figure out the offset into framebuffer memory we need to perform a simple calculation based on the X and Y coordinates we wish to affect. Because memory is arranged linearly in the buffer to get to the correct horizontal line we simply multiply the number of pixels on a line by the value of the Y coordinate. We then add the value of X and we have our offset.

[Image:Pixel offsetting](#)

```
unsigned short* frame_buffer = address_of_the_buffer;

frame_buffer[y * width_in_pixels + x] = color;
```

Let us translate this new knowledge of pixel plotting and color formats and see if we can produce an interesting (sort of) demo.

For our first pixel demo we will do a starfield with little floating dots. Each dot will make its way across the screen at a random speed. When it reaches the end we will move it back to the beginning and give it a new random height and new random speed. This should give us a nice star-trekie feeling demo of a moving star field.

Here is the source in its entirety which we will pick apart below; you can cut and paste this code into your main.c (or template.c if it is so named) from our first few demos. You can then build and run the demo.

```
#include <nds.h>
#include <stdlib.h>

#define NUM_STARS 40

typedef struct
{
    int x;
    int y;
    int speed;
    unsigned short color;
} Star;

Star stars[NUM_STARS];

void MoveStar(Star* star)
{
    star->x += star->speed;

    if(star->x >= SCREEN_WIDTH)
    {
        star->color = RGB15(31,31,31);
        star->x = 0;
        star->y = rand() % 192;
        star->speed = rand() % 4 + 1;
    }
}

void ClearScreen(void)
{
    int i;

    for(i = 0; i < 256 * 192; i++)
        VRAM_A[i] = RGB15(0,0,0);
}

void InitStars(void)
{
    int i;

    for(i = 0; i < NUM_STARS; i++)
    {
        stars[i].color = RGB15(31,31,31);
        stars[i].x = rand() % 256;
        stars[i].y = rand() % 192;
        stars[i].speed = rand() % 4 + 1;
    }
}

void DrawStar(Star* star)
{
    VRAM_A[star->x + star->y * SCREEN_WIDTH] = star->color;
}

void EraseStar(Star* star)
{
    VRAM_A[star->x + star->y * SCREEN_WIDTH] = RGB15(0,0,0);
}

int main(void)
{
    int i;

    irqInit();
    irqEnable(IRQ_VBLANK);

    videoSetMode(MODE_FB0);
    vramSetBankA(VRAM_A_LCD);

    ClearScreen();
    InitStars();

    //we like infinite loops in console dev!
    while(1)
    {
        swiWaitForVBlank();

        for(i = 0; i < NUM_STARS; i++)
        {
            EraseStar(&stars[i]);

            MoveStar(&stars[i]);

            DrawStar(&stars[i]);
        }
    }

    return 0;
}
```

We begin with a structure to define our star. It needs a location in the form of an X and Y coordinate, it needs speed, and finally it needs color.

```
typedef struct
{
    int x;
    int y;
    int speed;
    unsigned short color;
}Star;
```

We then need an array of stars we can track across the screen:

```
#define NUM_STARS 40

Star stars[NUM_STARS];
```

Before we start the demo, we need to clear the pixels of any color information they currently have. In other words, we are making sure we start with a black screen.

```
void ClearScreen(void)
{
    int i;

    for(i = 0; i < 256 * 192; i++)
        VRAM_A[i] = RGB15(0,0,0);
}
```

To start the demo off it would be nice if we could arrange our stars randomly about the screen. We do this with an initialize function.

```
void InitStars(void)
{
    int i;

    for(i = 0; i < NUM_STARS; i++)
    {
        stars[i].color = RGB15(31,31,31);
        stars[i].x = rand() % 256;
        stars[i].y = rand() % 192;
        stars[i].speed = rand() % 4 + 1;
    }
}
```

This function loops through all stars and sets the color to white, the speed to a random value between 1 and 4 and the X and Y to some random location on screen. If the '%' is unfamiliar to you I will give a brief explanation.

'%' performs a division and returns the remainder of that division.

Rand() returns a random short integer so modding the value with a number will result in a value which is between zero and that number. To generate a random number in any range between MIN and MAX is simply:

```
Num = rand() % (MAX-MIN) + MIN;
```

Next we need some function to move, draw, and erase the star. Let us begin with erase.

```
void EraseStar(Star* star)
{
    VRAM_A[star->x + star->y * SCREEN_WIDTH] = RGB15(0,0,0);
}
```

To erase just set the location of the star in the framebuffer to the background color (black in our case).

Similarly to draw the star we set the location of the star in the frame buffer to the color of the star:

```
void DrawStar(Star* star)
{
    VRAM_A[star->x + star->y * SCREEN_WIDTH] = star->color;
}
```

The final step is to move the star to its new location:

```
void MoveStar(Star* star)
{
    star->x += star->speed;

    if(star->x >= SCREEN_WIDTH)
    {
        star->color = RGB15(31,31,31);
        star->x = 0;
        star->y = rand() % 192;
        star->speed = rand() % 4 + 1;
    }
}
```

Moving a star is simple, we just add its speed to its current x position. The caviate is we must then check if the star has gone off screen. To do that we compare its x location to the width of the screen. If it is greater we know we are off the screen and we can take appropriate action.

When a sprite goes off screen we move it back to the left by settings its X value to 0. We then give it another random speed and random Y value making it look like a new star has come on screen.

The main loop which controls the demo consists of looping through each star and first erasing it from its old position, then moving it to its new position, and finally redrawing it at its new location:

```
//we like infinite loops in console dev!
while(1)
{
    swiWaitForVBlank();

    for(i = 0; i < NUM_STARS; i++)
    {
        EraseStar(&stars[i]);

        MoveStar(&stars[i]);

        DrawStar(&stars[i]);
    }
}
```

And so ends our pixel plotting demo. Below are a few more demos explained in the same excruciating manner as above.

[Color Bar Demo](#)

[[edit](#)] Touching things

The touch pad is an amazing addition to a handheld video game system that not only makes for an interesting gameplay experience but so too does it add a new level of fun to game programming.

This section will introduce the touch pad, show you how it works, and give a quick demo on its use.

The DS touchpad utilizes a resistive coating which changes conduction depending on the area of the contacting object. This change is measured by some analogue to digital converts on a special chip inside the DS and translated to an X and Y location. These measurements can also be used to determine the area of the contact point which, to some degree, can be translated into pressure.

To get to this raw data we must communicate with this chip via a serial interface which is only accessible via the ARM 7. Currently I do not have the stomach to go into serial comms in this tutorial but if you have a mind to explore such things the source code is in the arm7 code base of libnds.

For now I am just going to do a bit of hand waving and tell you there is code running on the arm7 in the default arm7 template which reads out the state of the touch pad. This data is unformed and does not correlate exactly to the dimensions of the screen meaning some processing must be done to convert this raw location data to useful pixel data.

The libnds arm7 stub does the appropriate conversion and communicates the result to the ARM9. These values are then read using the touchRead function which writes the raw coordinates and the transformed pixel coordinates into the pointer parameter.

You can also get a go - nogo test of the pen by using the scanKeys() macro we discussed as above. This tells you if the pen is up or down so you know when to read the touch pad.

Now for a simple demo. We will make the simplest of art programs possible. It will render random colored dots wherever you touch the screen.

```
#include<nds.h>
#include<stdlib.h>

int main(void)
{
    touchPosition touch;

    videoSetMode(MODE_FB0);
    vramSetBankA(VRAM_A_LCD);

    //notice we make sure the main graphics engine renders
    //to the lower lcd screen as it would be hard to draw if the
    //pixels did not show up directly beneath the pen
```

```

    lcdMainOnBottom();

    while(1)
    {
        scanKeys();

        if(keysHeld() & KEY_TOUCH)
        {
            // write the touchscreen coordinates in the touch variable
            touchRead(&touch);

            VRAM_A[touch.px + touch.py * 256] = rand();
        }
    }

    return 0;
}

```

There is not too much to say about this demo...but of course I am going to say it anyway.

You should notice I skipped the interrupt setup for this demo. This is for two reasons....there is no real animation cycle and we only render one pixel at a time. This means even if we draw while the screen is rendering our pixels there wont be anything to tear.

Notice also the use of scanKeys(); we use the keysHeld() macro instead of the keysDown() because keysDown() would only return true the first time the pen touches while keysHeld() returns true until the pen is lifted up again. This allows us to draw our dots without lifting the pen.

When using the touchRead function, we need to pass in the pointer to the variable touch rather than the variable itself. Putting an ampersand, &, in front of a variable replaces it with the address to that data.

Color is selected at random using rand() –recall rand() returns a random 16 bit value which is convenient as color is also 16 bit.

When drawing rapidly you probably noticed big gaps between your dots as apposed to nice smooth curves. This is because the touch coordinates are only updated once per frame and you can move the pen a lot faster than that. We will make this demo a little prettier when we learn how to draw lines a few pages hence.

[\[edit\]](#) Bitmap Graphics Modes

I talked a bit before about the DS graphics modes and alluded to being able to compose a scene from layers of graphics. Normally all rendering is done to one of these layers and this section will be the first real use of the 2D engine. Before we talk about the specifics let us look again at the possible graphics modes and what each layer can do in these modes.

Graphics Modes

Main 2D Engine				
Mode	BG0	BG1	BG2	BG3
Mode 0	Text/3D	Text	Text	Text
Mode 1	Text/3D	Text	Text	Rotation
Mode 2	Text/3D	Text	Rotation	Rotation
Mode 3	Text/3D	Text	Text	Extended
Mode 4	Text/3D	Text	Rotation	Extended
Mode 5	Text/3D	Text	Extended	Extended
Mode 6	3D	-	Large Bitmap	-
Frame Buffer	Direct VRAM display as a bitmap			
Sub 2D Engine				
Mode	BG0	BG1	BG2	BG3
Mode 0	Text	Text	Text	Text
Mode 1	Text	Text	Text	Rotation
Mode 2	Text	Text	Rotation	Rotation
Mode 3	Text	Text	Text	Extended
Mode 4	Text	Text	Rotation	Extended
Mode 5	Text	Text	Extended	Extended

You will notice each engine has several modes of operation with each mode having different background layer configurations. The background configurations we are interested in this case are the ones marked "extended" graphics layer.

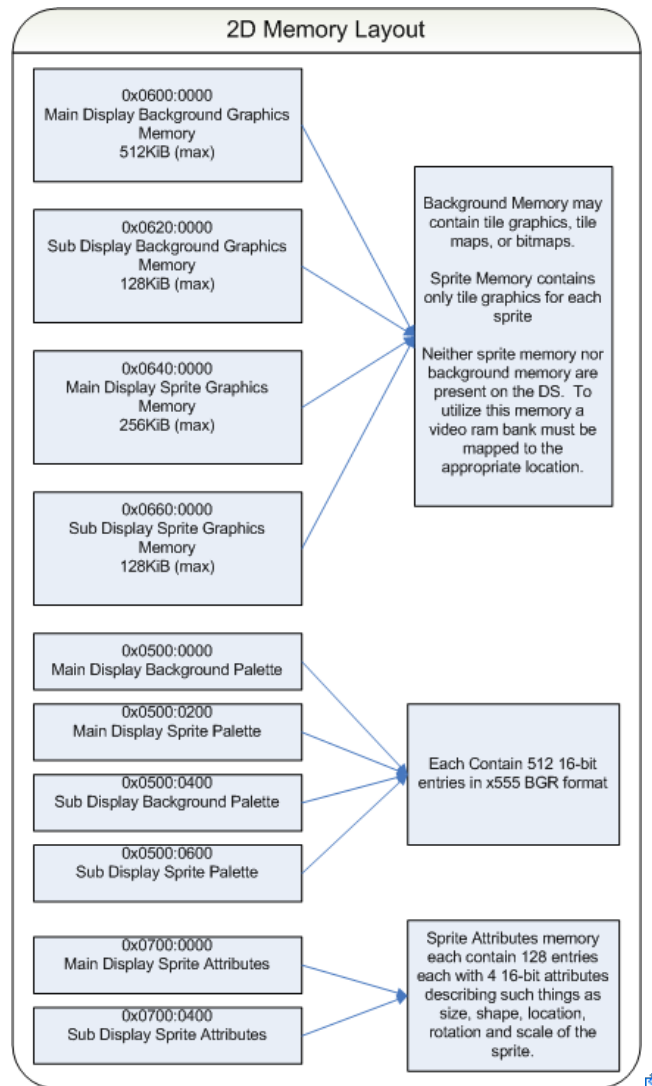
Extended rotation background layers can be configured as linear frame buffers, much like we have been using in the examples above.

When we talk tomorrow about tile based graphics we will cover in detail the capabilities of the "extended" backgrounds as well as the text and rotation backgrounds. For now we need only to understand a few things. First is how to put the display into a mode which supports an extended background, next is how to turn on the correct background, and finally we need to know how to

initialize the background properly.

The first thing to remember when using the 2D engine is the lack of memory available. In fact, the DS has no memory assigned to its 2D units by default other than Sprite attributes and base palettes.

In order for us to do anything we must map video memory somewhere the 2D engine can find it. To do this we must know where the engine is going to expect memory to be and we need to know what video memory can be mapped to these regions. What follows is the layout of 2D graphics memory.



The areas we are concerned with now are the background graphics memories. To use background layers we must map video memory to at least one of these regions.

Let us start with a small example which paints the screen red and see how using a background layer differs from direct frame buffer access.

```
#include <nds.h>

//-----
int main(void) {
//-----

    int i;

    //set video mode to mode 5
    videoSetMode(MODE_5_2D);

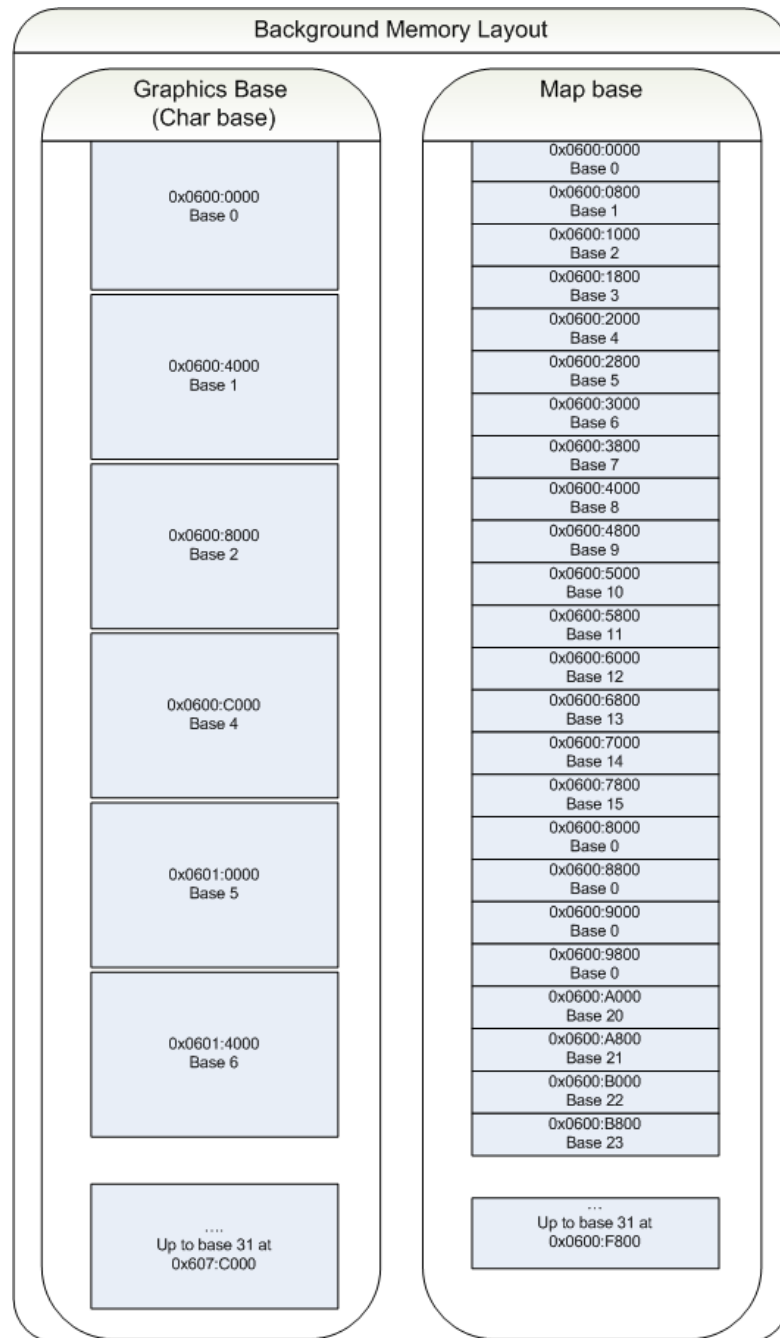
    //map vram a to start of background graphics memory
    vramSetBankA(VRAM_A_MAIN_BG);

    //initialize the background
    bgInit(3, BgType_Bmp16, BgSize_B16_256x256, 0,0);

    //write the color red into the background
    for(i=0; i<256*256; i++){
        BG_GFX[i]=RGB15(0,31,0) | BIT(15);
    }
    while(1) {
        swiWaitForVBlank();
    }
    return 0;
}
```

To better understand the memory layout for 2d backgrounds, we must first look at the image below. It

is logically divided into 32 blocks of memory for graphics (also 32 blocks for map data but that is a topic for tomorrow). We will revisit this organization of graphics memory tomorrow but for today it is enough to know that the background will pull data starting at one of these blocks and which block it pulls from is controlled via the background control register.



First, we set the video mode. If you look back at the video mode table, you will realize mode 5 allows for background layers 2 and 3 to be extended rotation backgrounds. We chose background 3 for this demo although background 2 would have worked just as well.

```
//set video mode to mode 5
videoSetMode(MODE_5_2D);
```

Next we map vram bank A to main background memory. The vram table shows we could have mapped other vram banks to this region as well. Picking which vram bank to map takes a bit of planning, but for our simple demos it will not be too difficult.

```
vramSetBankA(VRAM_A_MAIN_BG);
```

Next, we initialize the background. bgInit follows the header from the nds/arm9/Background.h: bgInit(int layer, BgType type, BgSize size, int mapBase, int tileBase). In this demo, we wanted a bitmap mode and since we have not really discussed palettes yet we are going to stick with 16 bit color. We choose a size of 256x256 to fill the entire screen. Once this code is completed, BG_GFX will be set to the address of the initialized layer; in this case, it is layer 3.

```
//initialize the background
bgInit(3, BgType_Bmp16, BgSize_B16_256x256, 0, 0);
```

The final step is to paint the screen red. Take a look at the following lines and see if you can note the difference between the straight framebuffer code.

```
//paint the screen red
for(i=0; i<256*256; i++){
    BG_GFX[i]=RGB15(0,31,0) | BIT(15);
}
```

Hopefully you noticed the setting of bit 15 of the color value. Remembering back to a previous talk about color formats on the DS you might recall there is a 16-bit color mode with the normal 5 bits of red green and blue along with one bit for alpha. 16-bit bitmap backgrounds use this color format.

The one bit of alpha tells the DS to render that pixel. If you leave this bit clear the pixel will not be drawn and anything behind it will show through. This is a very useful feature when you have two layers of background and want part of the top layer to be transparent.

Next we will look at paletted bitmaps through the somewhat practical exercise of decoding graphics files.

[\[edit\]](#) Bitmap on the sub display

For completeness sake here is an example which puts both the main and sub engines in mode 5 and renders to bitmap backgrounds. Notice the alternative register access using the background struct, instead of the bgInit function.

```
#include <nds.h>

//-----
int main(void) {
//-----

    int i;

    //point our video buffer to the start of bitmap background video
    u16* video_buffer_main = (u16*)BG_BMP_RAM(0);
    u16* video_buffer_sub = (u16*)BG_BMP_RAM_SUB(0);

    //set video mode to mode 5 with background 3 enabled
    videoSetMode(MODE_5_2D | DISPLAY_BG3_ACTIVE);
    videoSetModeSub(MODE_5_2D | DISPLAY_BG3_ACTIVE);

    //map vram a to start of main background graphics memory
    vramSetBankA(VRAM_A_MAIN_BG_0x06000000);
    vramSetBankC(VRAM_C_SUB_BG_0x06200000);

    //initialize the background
    BACKGROUND.control[3] = BG_BMP16_256x256 | BG_BMP_BASE(0);

    BACKGROUND.bg3_rotation.hdy = 0;
    BACKGROUND.bg3_rotation.hdx = 1 << 8;
    BACKGROUND.bg3_rotation.vdx = 0;
    BACKGROUND.bg3_rotation.vdy = 1 << 8;

    //initialize the sub background
    BACKGROUND_SUB.control[3] = BG_BMP16_256x256 | BG_BMP_BASE(0);

    BACKGROUND_SUB.bg3_rotation.hdy = 0;
    BACKGROUND_SUB.bg3_rotation.hdx = 1 << 8;
    BACKGROUND_SUB.bg3_rotation.vdx = 0;
    BACKGROUND_SUB.bg3_rotation.vdy = 1 << 8;

    //paint the main screen red
    for(i = 0; i < 256 * 256; i++)
        video_buffer_main[i] = RGB15(31,0,0) | BIT(15);

    //paint the sub screen blue
    for(i = 0; i < 256 * 256; i++)
        video_buffer_sub[i] = RGB15(0,0,31) | BIT(15);
    while(1) {
        swiWaitForVBlank();
    }
    return 0;
}
```

[\[edit\]](#) Background struct

libnds defines a struct for easy access of the background registers.

```
typedef struct {
    u16 x;
    u16 y;
} bg_scroll;

typedef struct {
    u16 hdx;
    u16 hdy;
    u16 vdx;
    u16 vdy;
```



```

    u32 dx;
    u32 dy;
} bg_transform;

typedef struct {
    u16 control[4];
    bg_scroll scroll[4];
    bg_transform bg2_rotation;
    bg_transform bg3_rotation;
} bg_attribute;

#define BACKGROUND      (*(bg_attribute *)0x04000008)
#define BACKGROUND_SUB  (*(bg_attribute *)0x04001008)

```

[[edit](#)] Working With Graphics Files

Being able to decode graphics files is a useful skill and although this is usually done on the PC we are going to do it directly on the DS just for kicks. There are a lot of different graphics files out there, and each file has advantages and disadvantages. For our purposes we need one that is easy to decode and is supported by many graphics applications. Some good choices would be: GIF, BMP and PCX.

I am going to tackle the BMP for this example. It is simple and supported by just about every graphics application on Earth.

To decode a file format you must first seek out its spec. A quick search on Google gives me the following information for bitmap files.

There is a short bitmap header followed by a variable length image header (this variable length header turns out to be 40 bytes in length almost always). Next comes the palette (if there is one) and finally the pixel data. Here is a bit more detail:

Bitmap File

Bitmap Header		
Offset	Size in bytes	Description
0	2	The characters "BM"
2	4	Filesize in bytes
6	4	Reserved (usually set to 0)
10	4	Offset to data
Image Header		
0	4	Size of image header (normally 40)
4	4	width of the image
8	4	Height of the image
12	2	Number of planes (normally 1)
14	2	Color depth (bits per pixel)
16	24	The rest is not interesting and hardly ever used
Palette Data		
The color palette stored as Red, Green, Blue bytes (with an extra byte of padding)		
Graphics Data		
The pixel data. This will either be indexes into the palette or raw Red, Green, Blue color data.		

BMPs support a number of bitmaps types and although this example will assume 8 bit 256 color bitmaps it could very easily be extended for other color depths (an excellent exercise for the reader if you are of a mind for those sorts of endeavors).

For a first step let us write a short demo which looks at the header, checks if it is bitmap file by reading the signature, and prints out the bits per color, height, and width.

[Image:Nds day3 bmp show.png](#) [Bitmap Header Decode](#)

We are going to use a trick called overlay to read the header. Instead of parsing each byte in and figuring it out we are going to define a structure which is the same size as the header. We can then pretend the start of the bitmap is the start of this structure and access all the attributes like normal structure members. Let's start with the bitmap header struct.

```

typedef struct
{
    char signature[2];
    unsigned int fileSize;
    unsigned int reserved;
    unsigned int offset;
}__attribute__((packed)) BmpHeader;

```

The idea is to look at the layout of the header and design a struct to match it. The first two characters are the signature so we add a character array of length 2 to line up with this. We do this for each element in the header.

Unfortunately for us the C language does not describe how exactly a compiler treats the memory assigned to a structure and often a compiler will pad a structure with empty bytes to make it a multiple of 32 bits in length. It does this because processing structs aligned so is generally more efficient. For us, we need the structures to be packed together so we can lay them on top of our bitmap data with no padding throwing us off. To ensure this is the case, we add the packed attribute to the structure definition...how you do this varies between compilers but for gcc this works like a charm.

Next we need a struct to hold the image header which we will just assume is 40 bytes, even though it could technically be variable.

```
typedef struct
{
    unsigned int headerSize;
    unsigned int width;
    unsigned int height;
    unsigned short planeCount;
    unsigned short bitDepth;
    unsigned int compression;
    unsigned int compressedImageSize;
    unsigned int horizontalResolution;
    unsigned int verticalResolution;
    unsigned int numColors;
    unsigned int importantColors;
}__attribute__((packed)) BmpImageInfo;
```

Notice again the use of the packed attribute.

Finally we define a structure to hold the entire bitmap and image header.

```
typedef struct
{
    unsigned char blue;
    unsigned char green;
    unsigned char red;
    unsigned char reserved;
}__attribute__((packed)) Rgb;

typedef struct
{
    BmpHeader header;
    BmpImageInfo info;
    Rgb colors[256];
    unsigned short image[1];
}__attribute__((packed)) BmpFile;
```

A bitmap file just consists of the two headers back to back followed by palette data and finally the image. This is where forgetting the packed attribute would byte you in the ass as gcc would stick in a few bytes of padding in-between the structs and things would not align (go ahead...try it).

The palette colors are stored as blue, green, red bytes followed by one empty byte. Since we are only going to decode 256 color bitmaps we are going to hardcode this into our bitmap structure. If you want to extend this you will need to do a small amount of parsing and first read in the bits per pixel and the number of colors before you do anything with the palette or image data.

The final entry in the bitmap file might seem a bit odd as it is an array of length one. Since we don't know how big the image array needs to be in advanced we give it a length of one, since it is an overlay we can just keep reading as far as we like.

Finally the code to decode the bitmap header:

```
//-----
int main(void) {
//-----

    BmpFile* bmp = (BmpFile*)beerguy_bin;

    consoleDemoInit();

    printf("%c%c\n", bmp->header.signature[0], bmp->header.signature[1]);
    printf("bit depth: %i\n", bmp->info.bitDepth);
    printf("width:      %i\n", bmp->info.width);
    printf("height:     %i\n", bmp->info.height);

    return 0;
}
```

This demo (if it were complete) would overlay our bitmap structure onto a bitmap file and print out the signature, bit depth and dimensions of the file. Hopefully, after all that discussion, there should only be one question remaining: How did I get a bitmap file into my DS application?

It turns out there are a lot of ways to get data into your application. You can use a file system and read it in from your compact flash or SD card. You can read it in from a wifi source like the internet or

a file share. You can run it through a converter and output the data as a big c array and compile it in or you can use object copy and create an object file you can link in.

The simplest way (thanks to wintermutes wonderful make file) is the object copy method. To include data in our project we just create a folder called "data" in the project folder (right next to source and include) and drop in a file. If we add a ".bin" to the end of the file name the make file will pick it up, run it through object copy, and create a header file with some ease of use variables declared.

For instance in the above demo I dropped in a bmp file called beerguy.bmp into the data folder. I then renamed it to beerguy.bin and typed make. A header file was created called beerguy_bin.h which contained the following:

```
extern const u8 beerguy_bin_end[];
extern const u8 beerguy_bin[];
extern const u32 beerguy_bin_size;
```

To access the data I just include the header file. I recommend you use this method as it works on any media and is simple. If you need file access because the 4MB limit is too much then the built in fatlib included with devkit pro is a great option.

I think we are finally ready to display this bitmap. All we have to do is copy the palette to the backgrounds palette memory and copy the image data to the backgrounds bitmap memory. Each of these steps have a small quirk.

The palette entries from the bitmap file are in 8 bits per color and we need 5. To fix this we need to chop off the lower 3 bits of each element. If you remember our conversation on bit operations this is a simple matter of shifting the number to the right by 3. Copying in the palette is done as follows:

```
//copy the palette
for(i = 0; i < 256; i++)
{
    Background_palette[i] = RGB15(bmp->colors[i].red >> 3, bmp->colors[i].green >> 3, bmp->colors[i].blue >> 3);
}
```

The quirk for the image data is that the image is (for some strange reason) stored upside down. We have to flip the image by reading the bottom of the bitmap into the top of the video buffer.

```
//copy the image
for(iy = 0; iy < bmp->info.height; iy++)
{
    for(ix = 0; ix < bmp->info.width / 2; ix++)
        video_buffer[iy * 128 + ix] = bmp->image[(bmp->info.height - 1 - iy) * (bmp->info.width + 3) + ix];
}
```

We do this by starting at line height minus 1 and subtracting the y index. Because video memory only accepts 16 or 32 bit writes we copy two bytes at a time (this is why you see the width divided by 2 in several locations and why video memory and bitmap image memory are declared as pointers to short instead of char).

[Demo Source](#)

[Image:Nds day3 bmp decode.png](#)  Displaying the Bitmap

[\[edit\]](#) The Double Buffer

Double buffering is a common technique used to address a common problem with raster graphics. Usually you do not want the user to see what you are rendering until you are done rendering it. To ensure this happens you have one of two options.

First, you can only render during the time the display is not rendering—vblank and hblank. This method works well and in fact is an often used mode on the DS as the hardware does much of the rendering work for us. But, often you just need more time to get your scene in order.

This brings us to method number two: The double buffer. For this we simply render to an off screen buffer. When we are done we make the off screen buffer visible and whatever was visible before becomes our new off screen buffer. This gives us as much time as we need to compose the scene with the slight drawback of needing a copy of visible video memory. Fortunately the DS is more than roomy enough to accommodate.

There are actually several ways to go about implementing a double buffer but the easiest is to use a single background's video memory and allocate one half to the visible buffer and one half to the non visible buffer (the "back buffer").

To demonstrate the use of double buffers let us extend our bitmap demo and animate a series of 10 frames. A friend was kind enough to render me up some full screen graphics and for this first go we will do it without a double buffer.

```
#include <nds.h>
#include "beerguy0010_bmp.h"
```

```

#include "beerguy0020_bmp.h"
#include "beerguy0030_bmp.h"
#include "beerguy0040_bmp.h"
#include "beerguy0050_bmp.h"
#include "beerguy0060_bmp.h"
#include "beerguy0070_bmp.h"
#include "beerguy0080_bmp.h"
#include "beerguy0090_bmp.h"
#include "beerguy0001_bmp.h"

extern void DisplayBmp(unsigned short* video_buffer, unsigned short* pal, unsigned char* bmp_data);

u8* bmps[10] =
{
    (u8*)beerguy0010_bmp,
    (u8*)beerguy0020_bmp,
    (u8*)beerguy0030_bmp,
    (u8*)beerguy0040_bmp,
    (u8*)beerguy0050_bmp,
    (u8*)beerguy0060_bmp,
    (u8*)beerguy0070_bmp,
    (u8*)beerguy0080_bmp,
    (u8*)beerguy0090_bmp,
    (u8*)beerguy0001_bmp
};

//-----
int main(void) {
//-----

    int frame = 0;

    BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(0);
    u16* video_buffer = (u16*)BG_BMP_RAM(0);

    irqInit();
    irqSet(IRQ_VBLANK, 0);

    videoSetMode(MODE_5_2D | DISPLAY_BG3_ACTIVE);
    //map vram a to start of background graphics memory
    vramSetBankA(VRAM_A_MAIN_BG_0x06000000);

    //initialize the background

    BG3_XDY = 0;
    BG3_XDX = 1 << 8;
    BG3_YDX = 0;
    BG3_YDY = 1 << 8;

    while(1)
    {
        DisplayBmp(video_buffer, BG_PALETTE, bmps[frame]);

        if(++frame > 9)
            frame = 0;

        swiWaitForVBlank();
    }

    return 0;
}

```

Compile and run this demo and you will see a bit of tearing and a lot of ugliness. The reason for this is my bitmap decode function is very slow and cannot complete in the vblank time frame.

The only thing different between this demo and the one before is the inclusion of 10 bitmap files which we cycle through once per vblank.

Let us use a double buffer and fix the problem.

```

#include <nds.h>

#include "beerguy0010_bmp.h"
#include "beerguy0020_bmp.h"
#include "beerguy0030_bmp.h"
#include "beerguy0040_bmp.h"
#include "beerguy0050_bmp.h"
#include "beerguy0060_bmp.h"
#include "beerguy0070_bmp.h"
#include "beerguy0080_bmp.h"
#include "beerguy0090_bmp.h"
#include "beerguy0001_bmp.h"

extern void DecodeBmp(unsigned short* video_buffer, unsigned short* pal, unsigned char* bmp_data);

u8* bmps[10] =
{
    (u8*)beerguy0010_bmp,
    (u8*)beerguy0020_bmp,
    (u8*)beerguy0030_bmp,
    (u8*)beerguy0040_bmp,
    (u8*)beerguy0050_bmp,
    (u8*)beerguy0060_bmp,
    (u8*)beerguy0070_bmp,
    (u8*)beerguy0080_bmp,
    (u8*)beerguy0090_bmp,
    (u8*)beerguy0001_bmp
};

```

```

(u8*)beerguy0060_bmp,
(u8*)beerguy0070_bmp,
(u8*)beerguy0080_bmp,
(u8*)beerguy0090_bmp,
(u8*)beerguy0001_bmp
};

//-----
int main(void) {
//-----

int frame = 0;
int swaped = 1;
int i = 0;

unsigned short old_palette[256];

BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(3); //---changed to (3)
u16* back_buffer = (u16*)BG_BMP_RAM(0);

irqInit();
irqSet(IRQ_VBLANK, 0);

videoSetMode(MODE_5_2D | DISPLAY_BG3_ACTIVE);

vramSetBankA(VRAM_A_MAIN_BG_0x06000000);

BG3_XDY = 0;
BG3_XDX = 1 << 8;
BG3_YDX = 0;
BG3_YDY = 1 << 8;

while(1)
{

    swiWaitForVBlank();

    if(swaped)
    {
        swaped = 0;
        BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(0);
        back_buffer = (u16*)BG_BMP_RAM(3);
    }
    else
    {
        swaped = 1;
        BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(3);
        back_buffer = (u16*)BG_BMP_RAM(0);
    }

    for(i = 0; i < 256; i++)
        BG_PALETTE[i] = old_palette[i];

    //decodes a bmp to the backbuffer
    DecodeBmp(back_buffer, old_palette, bmps[frame]);

    if(++frame > 9)
        frame = 0;

}

return 0;
}

```

In this demo we set up much as before only this time we set the starting point of background graphics to base 3. If you recall each base represents 16 KB of memory and if you are quick with math you will notice that a 256x192 image takes up exactly 3 blocks of memory. This means we set aside the first three blocks as visible and the next three as our render buffer.

```

BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(3);
u16* back_buffer = (u16*)BG_BMP_RAM(0);

```

We create a Boolean value which tracks which buffer is visible and alternate each frame. To swap the buffers we simply tell background 3 to render from one base and set the back buffer to the other base. This ensures we are always rendering to the off screen buffer and displaying the on screen buffer.

```

if(swaped)
{
    swaped = 0;
    BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(0);
    back_buffer = (u16*)BG_BMP_RAM(3);
}
else
{
    swaped = 1;
    BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(3);
    back_buffer = (u16*)BG_BMP_RAM(0);
}

```

The only other difference to note might seem a bit odd. It turns out that all my bitmaps use a separate palette so I must double buffer the palette as well! For this I use an in memory buffer.

The decode bitmap copies the palette to my local palette array (old_palette), and when I swap the visible buffer I also swap in this palette so the right palette is loaded with the right buffer.

```
for(i = 0; i < 256; i++)
    BG_PALETTE[i] = old_palette[i];

//decodes a bmp to the backbuffer
DecodeBmp(back_buffer, old_palette, bmps[frame]);
```

That is about all I am going to say on double buffers. They are useful and fairly simple to implement and can greatly enhance the look and feel of your program.

[[edit](#)] Raster 101

Raster graphics are the means by which most early games were rendered. It simply means to draw to a display on a per-pixel basis. We are going to let the DS hardware do most of our rendering for us, but there is something to be said for doing things the hard way every once in a while. We will cover line, circle, and polygon raster graphics and throw in a bit of optimization discussion along the way.

[[edit](#)] Bresenham Lines

We will begin our raster discussion with line drawing. The deceptively simple task of connecting two points on a 2D display by a series of pixels has been the subject of much research and countless papers. Perhaps we should start by defining a line.

In the majority of mathematical realms a line is an infinite, straight, one dimensional projection through space. To define a line all you need is the location of one point on that line and some indication of its direction.

Because one dimensionality is tough to achieve on a computer display and an infinite line might take too long to render we will have to restrict this definition a bit. For us all lines will lie on the plane of the screen, the length will definitely be finite, and that one dimensional thing will be very loosely applied. Let us take a close look at what a line on a computer display looks like to get a feel for what we need to accomplish.

<image of line here>

As you can see the line can only move in discrete steps of pixels. To render a line we just iterate through one dimension and plug the other into the equation for a line. Let us remember way back to geometry class and recall that a line can be defined as follows:

$$Y = mX + b;$$

Where Y is the axis we are trying to calculate, X is the axis we are iterating through, b is the value of Y when the line crosses the X axis and m is the slope of the line defined as change in Y divided by change in X (rise over run). I don't know about you but when I want to render a line I plan on just picking the two ends points and the color and having the algorithm do the rest. We can certainly calculate these values from two points but there is a slightly less common equation for a line that will be easier to work with:

$$Y - y = m (X - x)$$

In this case X Y and m are as before but x and y are the coordinates of any point on the line. As you might notice we don't have to worry about the intercept (previously 'b') in this form which simplifies things a bit.

So let us see if we can translate this equation into a line on the screen. First we will need to calculate the slope. Let us define our function as DrawLine(x1, y1, x2, y2, color) where the x's and y's are coordinates of the two points we wish to connect.

```
float m = (y1 - y2) / (x1 - x2)
```

One thing you should note right away is that slope will most likely be fractional in nature requiring us to use floating point math. You may also remember the DS has no floating point hardware and if you are particularly astute you will very shortly realize all this discussion is going to lead to some more interesting way of drawing a line.

Some pseudo code for drawing the line using this point slope equation would be as follows:

```
void DrawLine(float x1, float y1, float x2, float y2, short int color)
{
    float m = (x1 - x2) / (y1 - y2);

    for( ; x1 < x2; x1++)
    {
        float y = m * (x1 - x2) + y2;

        Buffer[x + y * BUFFER_WIDTH] = color;
    }
}
```

I don't recommend trying to compile this as there are several flaws in the algorithm. First we are kind

of assuming the line changes more in x than it does in y, if that is not the case we are going to be jumping more than 1 pixel in y each time we iterate through the loop and leave large gaps in the line. Second we are assuming that x1 is smaller than x2 when really we could have specified any two points for the function. You could of course modify the above and use this method to draw lines on the DS but it is certainly not the best way to go about.

This brings us to a more realistic way of rendering lines. It involves only integer math, no division, and no multiplication.

Bresenham lines:

Let us again look at how a line ends up looking on screen and see if we cant find a way to iterate through X and change Y accordingly without calculating the slope. First look at a line that changes more in the X direction than it does in the Y:

<image of a small slope closup line>

From inspection you can probably note the slope on this line is 1 / 3. If we zoom in a bit we see this 1 / 3 slope holds as the line drops down 1 pixel in the Y direction every 3 pixels in the X.

To calculate the slope we would normally do something like this:

$$M = (y1 - y2) / (x1 - x2)$$

But instead let us treat the difference in y and the difference in x separately and call them ydiff and xdiff respectively.

For this case:

```
Xdiff = x1 - x2;
Ydiff = y1 - y2;
```

It might be helpful at this point to consider how we would draw the above line if we had infinite resolution.

We would first plot a pixel at x1, y1 then move one X to the right. If this were an infinite display we would also move 1/3 of a pixel down and plot the pixel. Because our real display is finite the best I can do is move one pixel in X and 0 in the Y. If I were to continue I would move another pixel in X and another 1/3 of the way down in Y. Again, because we have a finite display and 2/3 of a pixel is pretty meaningless we settle for no change in Y. Finally, as I move for the third time in X I reach a point where I should be a full pixel in Y further down and I can render at the correct location.

This process is the basis of Bresenham's algorithm.

We keep track of this difference between the line we SHOULD draw and the line we CAN draw in an error term; when that error reaches a threshold we know we can correct by adjusting our Y value by one pixel up or down (down in this case). But what is that threshold and how much do we adjust the error term each time? Well, the easy answer would be to use 1.0 as the threshold and add the slope to the error each time. If we did this things would move along smoothly...unfortunately calculating slope requires a floating point division and tracking the error term would require even more floating point operations. Fortunately there is a simpler way.

We store the difference between x1, and x2 (xdiff) as well as the difference between y1 and y2 (ydiff). These two values will be proportional to the numerator and denominator of the slope. For instance, in the line depicted above the coordinates are (0,10) and (30, 0). This amounts to an xdiff of 30 and a ydiff of 10 (and a slope of 1/3 incase you have forgotten). To render the line we iterate in the X direction (because it changes more than the Y direction) and keep track of how far the line we are drawing is from the line we would like to draw.

We can use a threshold of xdiff and increment the error term by the ydiff (or the other way around for a line that changes more in Y than it does in X). In this case we add 10 to our error term each time we move in X and when it reaches 30 we move one step down in Y. We then reset the error term by subtracting 30 and continue on (notice we don't set it to zero as in most cases the ydiff and xdiff will not align so well). If we continue we will draw a line at the correct slope from the two points.

```
threshold = xdiff;

for(x = x1; x < x2; x++)
{
    //increment the error term
    error += ydiff;

    //if the error gets big enough we correct by moving down one
    //y increment
    if(error > threshold)
    {
        y++;
        error = error - threshold;
    }

    buffer[x + y * BUFFER_WIDTH] = color;
}
```

That is the basis of the bresenham algorithm. We just keep going one direction building up an error term until the error term reaches a certain point, then we correct by incrementing the other direction and reset the error term by the threshold.

Unfortunately there is a bit more to it. As before we only handle the case where the second point is

above and to the left of the first point and the line changes more in X than it does in Y. Fortunately handling these other cases turns out to be pretty easy.

Here is the final line algorithm, following will be a short explanation of the changes needed to make the code handle the other cases.

```
void DrawLine(int x1, int y1, int x2, int y2, unsigned short color)
{
    int yStep = SCREEN_WIDTH;
    int xStep = 1;
    int xDiff = x2 - x1;
    int yDiff = y2 - y1;

    int errorTerm = 0;
    int offset = y1 * SCREEN_WIDTH + x1;
    int i;

    //need to adjust if y1 > y2
    if (yDiff < 0)
    {
        yDiff = -yDiff; //absolute value
        yStep = -yStep; //step up instead of down
    }

    //same for x
    if (xDiff < 0)
    {
        xDiff = -xDiff;
        xStep = -xStep;
    }

    //case for changes more in X than in Y
    if (xDiff > yDiff)
    {
        for (i = 0; i < xDiff + 1; i++)
        {
            VRAM_A[offset] = color;

            offset += xStep;

            errorTerm += yDiff;

            if (errorTerm > xDiff)
            {
                errorTerm -= xDiff;
                offset += yStep;
            }
        }
    } //end if xdiff > ydiff
    //case for changes more in Y than in X
    else
    {
        for (i = 0; i < yDiff + 1; i++)
        {
            VRAM_A[offset] = color;

            offset += yStep;

            errorTerm += xDiff;

            if (errorTerm > yDiff)
            {
                errorTerm -= yDiff;
                offset += xStep;
            }
        }
    }
}
```

Notice we broke the cases where change in Y is greater and change in X is greater so we could loop through either X or Y accordingly. Also if the x and y values for the points were opposite from expected we just take the absolute value of the difference and change the x and y step so we step the other way through the line.

Now that we have an okay understanding of line drawing let us modify our drawing demo from before to connect the pixels we were drawing with lines. This should fill the gaps and make a nice smooth line as we trace around.

```
int main(void)
{
    touchPosition touch;

    int oldX = 0;
    int oldY = 0;

    videoSetMode(MODE_FB0);
    vramSetBankA(VRAM_A_LCD);

    lcdMainOnBottom();

    while(1)
    {
        scanKeys();
```



```

touchRead(&touch);

if (!(keysDown() & KEY_TOUCH) && (keysHeld() & KEY_TOUCH))
{
    DrawLine(oldX, oldY, touch.px, touch.py, rand());
}

oldX = touch.px;
oldY = touch.py;

swiWaitForVBlank();
}

return 0;
}

```

We make an old X and Y value to hold the previous position, grab the new position, and draw a line between them. Finally we update the old x and y and repeat. This code only draws a line if the pen is down for more than two frames because we need two points to draw a line. This is done by not drawing the line if the keysDown() touch is set.

When putting the code together, do not forget to #include <nds.h> and either copy and paste the DrawLine function above the main function or #include a header file with it defined.

[\[edit\]](#) Blitting Things

(todo: add a section on software blitting...not that important all things considered)

[\[edit\]](#) Drawing Pictures

[\[edit\]](#) Polygons

(todo: some software 3D)

[\[edit\]](#) Circles

This code is in pascal, but some changes will turn it easily in c++ :

```

procedure DrawPixel(X,Y:Integer;Color:Integer);
var
i : Integer;
begin
    i:=X + Y * SCREEN_WIDTH;
    if Color=0 then
        VRAM_A[i] := RGB15(0,0,0)
    else if Color=1 then
        VRAM_A[i] := RGB15(31,31,31);
end;

```

To draw a Circle

```

procedure DrawCircle (Rayon,X_Centre,Y_Centre : Integer; Color : Integer);
var
    x, y, m : Integer;
begin
    x := 0 ;
    y := rayon ;           // Place on the top of the circle
    m := 5-4*rayon ;       // initialisation
    while x <= y do begin  // while we are in the second half
        DrawPixel( x+x_centre, y+y_centre, Color ) ;
        DrawPixel( y+x_centre, x+y_centre, Color ) ;
        DrawPixel( -x+x_centre, y+y_centre, Color ) ;
        DrawPixel( -y+x_centre, x+y_centre, Color ) ;
        DrawPixel( x+x_centre, -y+y_centre, Color ) ;
        DrawPixel( y+x_centre, -x+y_centre, Color ) ;
        DrawPixel( -x+x_centre, -y+y_centre, Color ) ;
        DrawPixel( -y+x_centre, -x+y_centre, Color ) ;
    end;

```

Fill the circle or remove this 4 drawing lines if empty circle

```

LineBresenham( x+x_centre, y+y_centre, -x+x_centre, -y+y_centre, Color ) ;
LineBresenham( y+x_centre, x+y_centre, -y+x_centre, -x+y_centre, Color ) ;
LineBresenham( -x+x_centre, y+y_centre, x+x_centre, -y+y_centre, Color ) ;
LineBresenham( -y+x_centre, x+y_centre, y+x_centre, -x+y_centre, Color ) ;

```

Don't miss the end of the code

```

if m > 0 then begin        //choix du point F
    y := y - 1 ;
    m := m-8*y ;

```

```
end ;  
x := x+1 ;  
m := m + 8*x+4 ;  
end ;  
end ;
```

Dev-Scene (c)

Curso Linux Avanzado

www.seas.es

Especializate con un Curso Online. 150 hs 6 ECTS. Título Universitario





NDS/Tutorials Day 4

< [NDS](#)

Contents

- [1 Introduction](#)
- [2 Tile Modes](#)
 - [2.1 Background Memory Layout and VRAM Management](#)
 - [2.1.1 Map Entries](#)
 - [2.1.2 First Map Demo](#)
 - [2.2 Text Backgrounds](#)
 - [2.3 Rotation Backgrounds](#)
 - [2.4 Extended Rotation Backgrounds](#)
- [3 Creating Map Data](#)
 - [3.1 From an Image](#)
 - [3.2 Using a Map Editor](#)
- [4 Meta Tiles](#)
- [5 Scrolling](#)
 - [5.1 Horizontal Scrolling](#)
 - [5.2 Vertical Scrolling](#)
 - [5.3 Scrolling Both Ways](#)
 - [5.4 Dynamic Tile Loading](#)

[\[edit\]](#) Introduction

In yesterday's chapter we talked at length about how to compose an image on screen by manipulating each pixel until our scene was formed. While this gave us a great deal of control over the final result we quickly realized the DS is not quite a software rendering powerhouse.

To compensate for a relatively slow processor and an even more limiting amount of VRAM the DS includes several hardware features, the result of which is the most advanced dedicated 2D processing systems ever placed in a video game console.

Tile based rendering is the key component of this 2D technology and understanding it will allow you to squeeze enormous, detailed, and fully interactive worlds from the seemingly limited DS resources.

[\[edit\]](#) Tile Modes

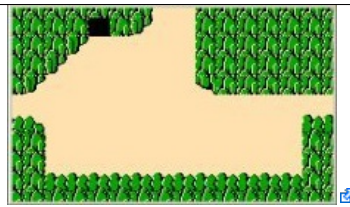
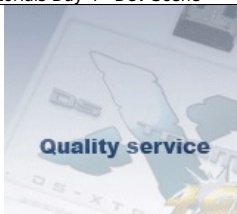
What are tile-based graphics? Put simply, it means to describe your scene using a mapping of tile indexes to tile graphics. Instead of describing the screen as a 2D matrix of pixels we are going to describe it as a matrix of tiles, where each tile represents a small bitmap. Let us look at one of the better-known tile based games and get a feel for how it was put together.

Below is all the graphics used to construct the entire overworld of the original Zelda.



You may recognize these little 16x16 chunks as pieces of the Zelda world and perhaps you could imagine that in order to describe the look of the overworld all one would have to do is store which tile goes where. For instance the following familiar scene could be represented by an array of tile numbers.





Such as this:

```
short map[] = {
  64,64,64,64,64,64,64,06,06,64,64,64,64,64,64,64,
  64,64,64,64,07,64,62,06,06,64,64,64,64,64,64,64,
  64,64,64,62,06,06,06,06,06,64,64,64,64,64,64,64,
  64,64,62,06,06,06,06,06,06,64,64,64,64,64,64,64,
  64,62,06,06,06,06,06,06,63,64,64,64,64,64,64,
  06,06,06,06,06,06,06,06,06,06,06,06,06,06,06,
  64,67,06,06,06,06,06,06,06,06,06,06,06,64,64,
  64,64,06,06,06,06,06,06,06,06,06,06,06,64,64,
  64,64,06,06,06,06,06,06,06,06,06,06,06,64,64,
  64,64,66,66,66,66,66,66,66,66,66,66,64,64,
  64,64,64,64,64,64,64,64,64,64,64,64,64,64
};
```

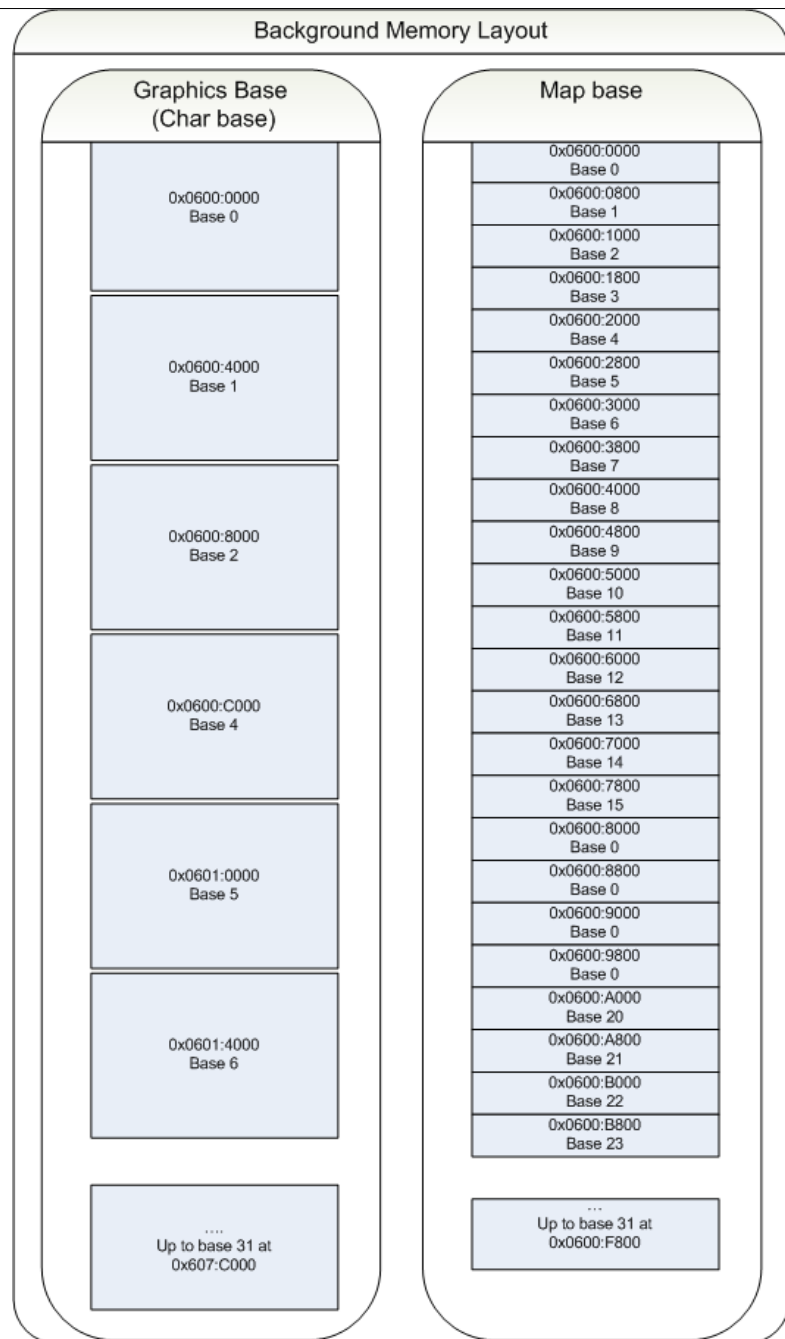
To render the scene all you would have to do is loop through the indexes stored in the map array and use those values to blit each tile to the screen. The entire world map would only end up being a few KBs in size instead of the 10s of Megabytes it would take to store it as a big image. The trade off of course is the increase in the amount of time it takes to render because we have to do a conversion between this map and the final bitmap we want on the screen.

Fortunately (and hopefully obviously at this point) the NDS 2D hardware is built for just this purpose making rendering tile based worlds a snap. All we really need to do is create a map and a tile set, place them into 2D video memory and tell the DS where to find them and it will do the magic for us.

[\[edit\]](#) Background Memory Layout and VRAM Management

In order to meet this first goal of placing tiles and maps into memory we must know where in memory to place them and in what format the NDS expects this data. This brings us back to the seemingly ever-present task of video memory management and memory layout.

The image below will look familiar if you read yesterday's chapter on bitmap graphics modes.



The above depicts the layout of main background memory as seen by the 2D engine. Background memory is divided into character memory (where you stick the actual tile graphics) and map memory (where you stick the map data).

It turns out you can place maps anywhere in the first 64KB of background memory and you can place tiles anywhere in the first 256KB; the blocks depicted above are only logical offsets and nothing prevents data from crossing these boundaries.

To load a map into memory you pick a block offset and write the map there, then tell the DS were to find it. You then do the same for your tile graphics. Finally you load a palette and call it a day (there may be a few more details).

One thing we need to figure out is the format of tile and map data. It turns out this is rather straight forward.

[[edit](#)] Map Entries

There are two forms of maps. Ones with 8 bit entries and ones with 16-bit. The 8-bit flavor are simply an offset into character memory.

For instance if you want the third entry in your map to use tile number 4 you just stick a 4 in that maps entry.

8 bit indexed maps are used only for "Rotation" backgrounds. "Text" and "Extended Rotation" use the more flexible 16 bit indices.

16-bit indexes are broken up into character index and control bits. The low 10 bits represent the index of the character and allow you to address up to 1024 unique characters. The next two bits will cause the character to flip vertically or horizontally. Finally there are 4 bits which let you choose a palette.

Bits: **15 14 13 12** **11** **10** **9 8 7 6 5 4 3 2 1 0**

To create a map you fill an array of short ints with these character indexes and you can control not only which character is rendered to the screen but what palette it uses and if it is flipped. If you ignore the flip bits and the palette bits you can treat it as a simple character index (you are still limited to 1024 tiles though).

I think at this point we know just enough about maps to get into trouble so let us see if we can trick the DS into displaying one.

[\[edit\]](#) First Map Demo

This should serve as an introduction to rendering a map. In the next sections we will go into detail about maps and character graphics.

```
#include <nds.h>

//create a tile called redTile
u8 redTile[64] =
{
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
};

//create a tile called greenTile
u8 greenTile[64] =
{
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
};

//-----
int main(void) {
//-----
    int i;

    //set video mode and map vram to the background
    videoSetMode(MODE_0_2D | DISPLAY_BG0_ACTIVE);
    vramSetBankA(VRAM_A_MAIN_BG_0x06000000);

    //get the address of the tile and map blocks
    u8* tileMemory = (u8*)BG_TILE_RAM(1);
    u16* mapMemory = (u16*)BG_MAP_RAM(0);

    //tell the DS where we are putting everything and set 256 color mode and that we are using a
    REG_BG0CNT = BG_32x32 | BG_COLOR_256 | BG_MAP_BASE(0) | BG_TILE_BASE(1);

    //load our palette
    BG_PALETTE[1] = RGB15(31,0,0);
    BG_PALETTE[2] = RGB15(0,31,0);

    //copy the tiles into tile memory one after the other
    swiCopy(redTile, tileMemory, 32);
    swiCopy(greenTile, tileMemory + 64, 32);

    //create a map in map memory
    for(i = 0; i < 32 * 32; i++)
        mapMemory[i] = i & 1;
    while(1)
        swiWaitForVBlank();

    return 0;
}
```

Like other demos we first pick a video mode which does what we want and turn on the things that need turned on. We then map in vram as appropriate.

```
videoSetMode(MODE_0_2D | DISPLAY_BG0_ACTIVE);
vramSetBankA(VRAM_A_MAIN_BG_0x06000000);
```

As you can see we chose mode 0 and background 0 and if you look on the graphics mode table you will notice this gives us a "Text" background. That means we will use 16 bit indexes.

The next thing we do in this demo is choose a tile block and a map block as the starting points for our data. You can choose any combination of blocks you like as long as your map data and your tile data do not end up in the same place.

A good approach is to stick your maps in the first few map blocks and the tiles starting at tile block 1. This gives you 16K for maps which is normally enough.

```
u8* tileMemory = (u8*)BG_TILE_RAM(1);
u16* mapMemory = (u16*)BG_MAP_RAM(0);
```

Now that we have made this choice we need to let the DS know about it through the background control register.

```
BG_CR = BG_32x32 | BG_COLOR_256 | BG_MAP_BASE(0) | BG_TILE_BASE(1);
```

We also put the background in 256 color mode. We could have picked 16 color mode but that is a touch harder to create data by hand for. We also opted to go with a 32 by 32 tile map.

If you look towards the top of the file you notice we created two tiles filled with 1 and 2 respectively. We arbitrarily called them red and green so I suppose we should make sure the color in palette entry 1 is in fact red and the one in palette entry 2 is green.

```
BG_PALETTE[1] = RGB15(31,0,0); //red
BG_PALETTE[2] = RGB15(0,31,0); //green
```

Next we use a bios call to copy the tiles into tile memory. First one then the other. The call takes the source of the data, the destination, and the size (in halfwords). The red tile is copied into tile offset 0 and the next 64 bytes later which is tile offset 1.

```
swiCopy(redTile, tileMemory, 32);
swiCopy(greenTile, tileMemory + 64, 32);
```

Now all we need to do is put a map in map memory. If we set the tile index to 0 a red tile will be drawn. If we set it to 1 a green tile will be drawn. Because creating a map by hand is a pain we use a short loop which alternately sets the index to 1 or 0 (as always if this is unclear be sure to review the bit manipulation techniques we discussed in day 2).

```
for(i = 0; i < 32 * 32; i++)
    mapMemory[i] = i & 1;
```

If you compile and run this demo you will be greeted by your first tile based background...a simple set of vertical strips which alternate red and green.

The next step in our endeavor will be to explore the Text backgrounds fully.

[\[edit\]](#) Text Backgrounds

Please keep on this splendid tutorial! It has begun to be very excited and now which way will I go further? Plz carry it on!

[\[edit\]](#) Rotation Backgrounds

[\[edit\]](#) Extended Rotation Backgrounds

[\[edit\]](#) Creating Map Data

[\[edit\]](#) From an Image

[\[edit\]](#) Using a Map Editor

[\[edit\]](#) Meta Tiles

[\[edit\]](#) Scrolling

[\[edit\]](#) Horizontal Scrolling

[\[edit\]](#) Vertical Scrolling

[\[edit\]](#) Scrolling Both Ways

[\[edit\]](#) Dynamic Tile Loading

Dev-Scene (c)

DWG to JPG, TIF, BMP, GIF

www.anydwg.com

Batch convert DWG/DXF to JPG, TIF, BMP, GIF, PNG, PCX, TGA, EMF, WMF.





NDS/Tutorials Day 5

< [NDS](#)

Hardware sprites free us from the fixed tiles we have looked at so far. These are crucial to any 2D game.

Contents

- [1 intro](#)
- [2 Setting your sprite up for GRIT](#)
- [3 Make File](#)
- [4 C++ Code](#)
- [5 oamset functions](#)

[\[edit\]](#) intro

This is a tutorial to teach you the basics of sprites,

first of create a new project, in your new folder create the following folders:

```
source
data
gfx
```

[\[edit\]](#) Setting your sprite up for GRIT

place your sprite in the gfx folder for the purpose of this tutorial call it sprite.png (make sure its 8-bit), in the gfx folder you also need to make a file called sprite.grit, open it up with programmers notepad or notepad and fill it with this code:

```
#8 bit bitmap
-gB8

#Tile the image
-gt

#cyan transparent
-gT 00FFFF
```

I have set this up so cyan is transparent, the transparency is the hex color on the last line, it can be changed as you like.

[\[edit\]](#) Make File

in your project folder create a file named "makefile" open it with programmers notepad and paste the following code:

```
.SUFFIXES:

ifeq ($(strip $(DEVKITARM)),)
$(error "Please set DEVKITARM in your environment. export DEVKITARM=<path to>devkitARM")
endif

include $(DEVKITARM)/ds_rules
TARGET := $(shell basename $(CURDIR))
BUILD := build
SOURCES := source
DATA := data
INCLUDES := include
GRAPHICS := gfx

ARCH := -mthumb -mthumb-interwork

CFLAGS := -g -Wall -O2\
-march=armv5te -mtune=arm946e-s -fomit-frame-pointer\
-ffast-math \
$(ARCH)
```



```

CFLAGS += $(INCLUDE) -DARM9
CXXFLAGS := $(CFLAGS) -fno-rtti -fno-exceptions

ASFLAGS := -g $(ARCH)
LDFLAGS = -specs=ds_arm9.specs -g $(ARCH) -mno-fpu -Wl,-Map,$(notdir $*.map)

LIBS := -lnds9

LIBDIRS := $(LIBNDS)

ifneq ($(BUILD),$(notdir $(CURDIR)))

export OUTPUT := $(CURDIR)/$(TARGET)

export VPATH := $(foreach dir,$(SOURCES),$(CURDIR)/$(dir)) \
    $(foreach dir,$(GRAPHICS),$(CURDIR)/$(dir)) \
    $(foreach dir,$(DATA),$(CURDIR)/$(dir))

export DEPSDIR := $(CURDIR)/$(BUILD)

CFILES := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.c)))
CPPFILES := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.cpp)))
SFILES := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.s)))
BINFILES := $(foreach dir,$(DATA),$(notdir $(wildcard $(dir)/*.*) ) )
PNGFILES := $(foreach dir,$(GRAPHICS),$(notdir $(wildcard $(dir)/*.png)))

ifeq ($(strip $(CPPFILES)),)
    export LD := $(CC)
else
    export LD := $(CXX)
endif

export OFILES := $(addsuffix .o,$(BINFILES)) \
    $(PNGFILES:.png=.o) \
    $(CPPFILES:.cpp=.o) $(CFILES:.c=.o) $(SFILES:.s=.o)

export INCLUDE := $(foreach dir,$(INCLUDES),-I$(CURDIR)/$(dir)) \
    $(foreach dir,$(LIBDIRS),-I$(dir)/include) \
    $(foreach dir,$(LIBDIRS),-I$(dir)/include) \
    -I$(CURDIR)/$(BUILD)

export LIBPATHS := $(foreach dir,$(LIBDIRS),-L$(dir)/lib)

.PHONY: $(BUILD) clean

$(BUILD):
    @[-d $@ ] || mkdir -p $@
    @make --no-print-directory -C $(BUILD) -f $(CURDIR)/Makefile

clean:
    @echo clean ...
    @rm -fr $(BUILD) $(TARGET).elf $(TARGET).nds $(TARGET).arm9 $(TARGET).ds.gba

else

DEPENDS := $(OFILES:.o=.d)

$(OUTPUT).nds : $(OUTPUT).arm9
$(OUTPUT).arm9 : $(OUTPUT).elf
$(OUTPUT).elf : $(OFILES)

%.bin.o : %.bin
    @echo $(notdir $<)
    @$ (bin2o)

%.s %.h : %.png %.grit
    grit $< -fts -o$*

-include $(DEPENDS)

endif

```

[\[edit\]](#) C++ Code

now in your source folder create a new cpp file called "main.cpp", thn fill it with the following code:

```

/*****
 * NDS Sprite Tutorial
 * Author: opearn
 *****/

#include<

//include

#include <nds.h>

//include sprites

#include "sprite.h"

int main() {
    // Setup the video modes.
    videoSetMode( MODE_0_2D );
    vramSetPrimaryBanks(VRAM_A_MAIN_SPRITE,VRAM_B_LCD,VRAM_C_LCD,VRAM_D_LCD);

```

```
//setup sprites
oamInit(&oamMain, SpriteMapping_Bmp_1D_128, false); //initialize the oam
u16* gfx = oamAllocateGfx(&oamMain, SpriteSize_64x64, SpriteColorFormat_256Color); //make room for gfx
dmaCopy(spriteTiles, gfx, spriteTilesLen); //copy the sprite
dmaCopy(spritePal, SPRITE_PALETTE, spritePalLen); //copy the sprites palette
oamEnable(&oamMain);

while (1) //infinite loop
{
    oamSet(&oamMain, 0, 64, 32, 0, 0, SpriteSize_64x64, SpriteColorFormat_256Color, gfx, 0, false, false, false);
    swiWaitForVBlank();
}
return 0;
}
```

You should now have a basic app that shows your sprite at 64 32, but now to make it practical, you'll want to create some variables using

```
int x=0;
int y=0;
```

in the oam set line of code can you tell what the sprites coordinates are? if so replace them with x and y


```
oamSet(&oamMain, 0, x, y, 0, 0, SpriteSize_64x64, SpriteColorFormat_256Color, gfx, 0, false, false, false);
```

[\[edit\]](#) oamset functions

<http://libnds.devkitpro.org/a00100.html#605548d2e83c72673de90d505e0f816e>

Dev-Scene (c)





Search

Dev-Scene[Home](#)[News](#)[Forum](#)[Blogs](#)[Planet](#)[Library](#) 1**Navigation**[Nintendo DS](#)[- NDS Homebrew Catalog](#)[Nintendo Wii](#)[- Wii Homebrew Catalog](#)[Current events](#)[Community portal](#)[Developers/Donations](#)**Personal tools**[Log in / create account](#)**Toolbox**[Random page](#)[Recent changes](#)[Help](#)[What links here](#)[Upload file](#)[Special pages](#)

¿QUIERES PROGRAMAR VIDEOJUEGOS PARA iOS Y ANDROID?




Curso DESARROLLADOR JUEGOS PARA MÓVILES


NDS/Tutorials Day 6


[< NDS](#)

There is currently no text in this page, you can [search for this page title](#) in other pages or [edit this page](#).

Dev-Scene (c)



V8880 Android 4.2 Tablet Only **\$59.99**
Dual Core Cortex A9 1.5GHz




[Evolve your Tablet](#)
Everbuying.com






R4
Revolution for DS

\$9.95 In Stock!
Same day free shipping



 **Dev-Scene.com**

Search

Dev-Scene	
Home	↗
News	↗
Forum	↗
Blogs	↗
Planet	↗
Library	↗

¿Qué siente por ti?



Descúbrelo con el

TAROT DEL AMOR


100% GRATIS

NDS/Tutorials Day 7

< [NDS](#) [↗](#)

There is currently no text in this page, you can [search for this page title](#) [↗](#) in other pages or [edit this page](#) [↗](#).

Dev-Scene (c)



Navigation	
Nintendo DS	↗
- NDS Homebrew Catalog	↗
Nintendo Wii	↗
- Wii Homebrew Catalog	↗
Current events	↗
Community portal	↗
Developers/Donations	↗
Personal tools	
Log in / create account	↗
Toolbox	
Random page	↗
Recent changes	↗
Help	↗
What links here	↗
Upload file	↗
Special pages	↗


electrobee



R4
Revolution for DS


\$9⁹⁵ In Stock!
Same day free shipping





Search

Dev-Scene	
Home	
News	
Forum	
Blogs	
Planet	
Library	

**TP Móvil 200**

- 200 min. en llamadas a cualquier operador
- 200MB de navegación a máxima velocidad

Por sólo **5 €/mes**
(6,05€ con IVA)
Cada línea Para siempre

NDS Tutorials Day 8


There is currently no text in this page, you can [search for this page title](#) in other pages or [edit this page](#).

Dev-Scene (c)

Any DWG to DWF Converter

www.anydwg.com

Batch convert DWG to DWF,DXF to DWF High Quality & More Features.



Navigation	
Nintendo DS	
- NDS Homebrew Catalog	
Nintendo Wii	
- Wii Homebrew Catalog	
Current events	
Community portal	
Developers/Donations	
Personal tools	
Log in / create account	
Toolbox	
Random page	
Recent changes	
Help	
What links here	
Upload file	
Special pages	



R4
Revolution for DS

\$9⁹⁵ In Stock!
Same day free shipping

















 **Dev-Scene.com**

Search

Dev-Scene	
Home	
News	
Forum	
Blogs	
Planet	
Library	



Navigation	
Nintendo DS	
- NDS Homebrew Catalog	
Nintendo Wii	
- Wii Homebrew Catalog	
Current events	
Community portal	
Developers/Donations	
Personal tools	
Log in / create account	
Toolbox	
Random page	
Recent changes	
Help	
What links here	
Upload file	
Special pages	

Curso Linux Avanzado

www.seas.es

Descubre sus secretos y conviértete en experto!



NDS/Tutorials Day 9

< [NDS](#) 

Excellent tutorial. Please keep on with the good stuff.

Dev-Scene (c)


DWG to JPG, TIF, BMP, GIF

www.anydwg.com


Batch convert DWG/DXF to JPG, TIF, BMP, GIF, PNG, PCX, TGA, EMF, WMF.



















 **Dev-Scene.com**

Search

Dev-Scene	
Home	
News	
Forum	
Blogs	
Planet	
Library	



Navigation	
Nintendo DS	
- NDS Homebrew Catalog	
Nintendo Wii	
- Wii Homebrew Catalog	
Current events	
Community portal	
Developers/Donations	
Personal tools	
Log in / create account	
Toolbox	
Random page	
Recent changes	
Help	
What links here	
Upload file	
Special pages	



¿Qué siente por ti?



Descúbrelo con el

TAROT DEL AMOR

100% GRATIS

NDS/Tutorials Day 10

< [NDS](#) 

There is currently no text in this page, you can [search for this page title](#)  in other pages or [edit this page](#) .

Dev-Scene (c)

Any DWG to DWF Converter

www.anydwg.com

Batch convert DWG to DWF,DXF to DWF High Quality & More Features.





R4
Revolution for DS

\$9⁹⁵ In Stock!
Same day free shipping



Dev-Scene

[Home](#)[News](#)[Forum](#)[Blogs](#)[Planet](#)[Library](#)

1

Navigation

[Nintendo DS](#)[- NDS Homebrew Catalog](#)[Nintendo Wii](#)[- Wii Homebrew Catalog](#)[Current events](#)[Community portal](#)[Developers/Donations](#)

Personal tools

[Log in / create account](#)

Toolbox

[Random page](#)[Recent changes](#)[Help](#)[What links here](#)[Upload file](#)[Special pages](#)

Any DWG to DWF Converter

www.anydwg.com

Batch convert DWG to DWF, DXF to DWF High Quality & More Features.

NDS/Tutorials Collision Detection

[< NDS](#)

There are most probably other, more advanced ways to detect collisions than the ones listed here. If you know of any, please add them!

Contents

- [1 2-D Collision Detection](#)
 - [1.1 Intersecting Rectangles](#)
 - [1.2 Pixel Collision](#)
 - [1.3 Row-Rectangle Intersection](#)
 - [1.4 Rectangle Subdivision](#)
 - [1.5 R-Squared Method](#)

[\[edit\]](#) 2-D Collision Detection

[\[edit\]](#) Intersecting Rectangles

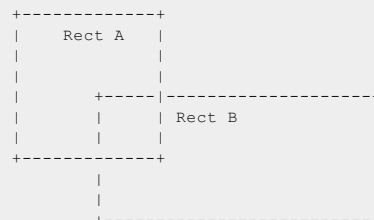
This method encloses two objects in rectangles and tests to see if the rectangles intersect. This works most perfectly for objects that are actually rectangular, but because this is one of the simplest (and fastest) ways to do collision detection, game programmers often use this for non-rectangular objects as well, where perfect accuracy is not required.

There are two cases where two rectangles could collide:

- A point from one rectangle is contained in the other.

Figure 1

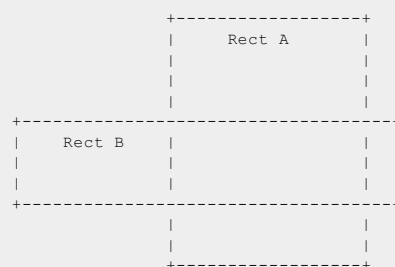
A way that rectangles can intersect. Note that a corner point of Rect A is inside Rect B, and vice versa.



- Adjacent points on one rectangle flank the other rectangle, so that a border from one rectangle crosses completely over the other rectangle.

Figure 2

Another way that rectangles can intersect. Note that although none of the corner points are enveloped by another rectangle, each rectangle has corner points on two opposite sides of the other rectangle. However, if just one border breaks one border of the other rectangle, that is sufficient to show that the rectangles intersect.



The only case where the second case could be triggered and not the first is a situation similar to that of Figure 2, where the rectangles from something like a plus sign (+). If the rectangles are too wide or tall for one rectangle to completely cross the other like Figure 2 in one animation frame, type 2 checking could possibly be ignored because it would never be triggered without already triggering type





1. However, for the sake of portable code, it may be best to include type 2 checking because even with its inclusion, checking two rectangles for intersection is trivial.

Rectangle intersection can also be used to "short-circuit" algorithms for more CPU-intensive collision detection. That is, you could have your program check potential intersections using more advanced algorithms *only if* this simpler rectangle intersection algorithm returns true. For example, you could have a function like this:

```
bool intersect(Sprite a, Sprite b)
{
    if (simpleRectanglesIntersect(a, b))
        return moreComplexIntersect(a, b);
    else
        return false;
}
```

Or, more simply:

```
bool intersect(Sprite a, Sprite b)
{
    // let the && operator do the short-circuiting
    return simpleRectanglesIntersect(a, b) && moreComplexIntersect(a, b);
}
```

Since most game objects only collide for one frame before disappearing, this short-circuiting method can make the game run much faster.

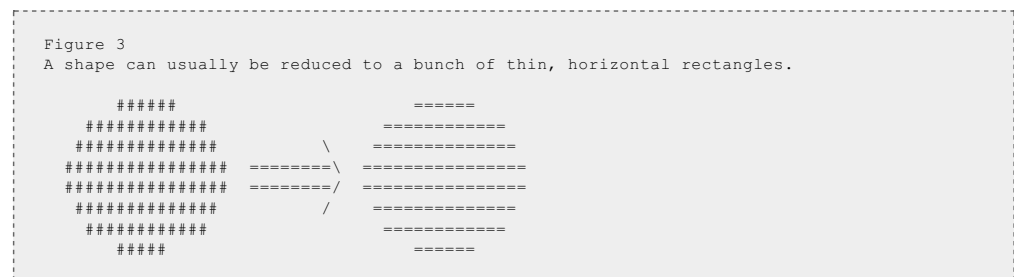
[\[edit\]](#) Pixel Collision

This method individually checks every pixel in both objects to see if they occur at the same point.

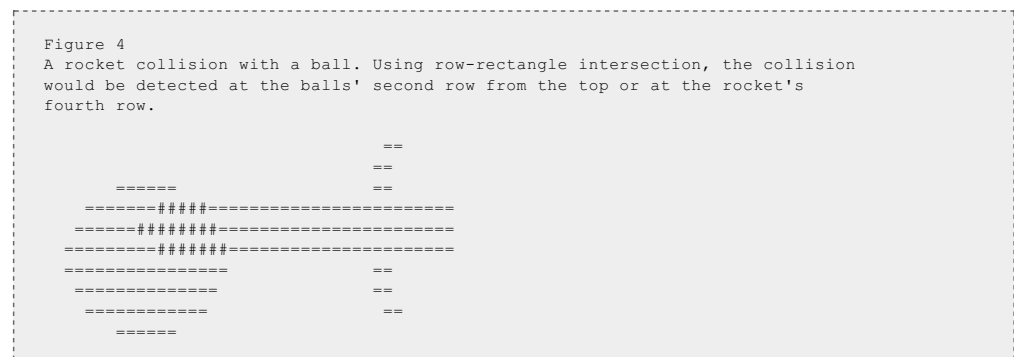
In general, don't ever use this brute-force method if you can help it. While it is perfectly accurate, it is *extremely* *slo-o-ow*. There are usually better, similarly accurate methods like row-rectangle intersection.

[\[edit\]](#) Row-Rectangle Intersection

Usually, an image can be split into one-pixel-high rectangles traversing the object horizontally, as in Figure 3:



Most frequently, there is only one such rectangle per image row. If each of these rectangles are checked instead of individual pixels, the algorithm could be much faster. This is because entire rows are checked at the same time.



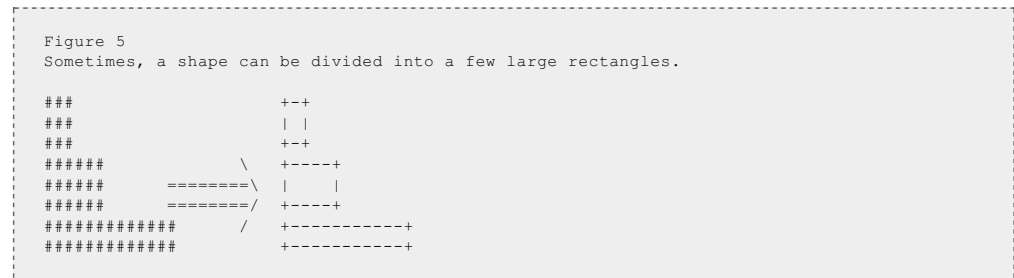
This algorithm can be very efficient, as far as pixel-perfect algorithms go. Don't make it complicated; checking each row in `ballSprite` against each row in `rocketSprite` is horribly inefficient, taking `ballSprite->size.y*rocketSprite->size.y` row checks to complete. Instead, by using `ballSprite->origin.y - rocketSprite->origin.y`, you can do one-to-one comparisons between the rows. This is much more efficient; the number of row checks necessary is `min(ballSprite->size.y, rocketSprite->size.y)` at maximum.

The simple row-rectangle method, however, is not able to check for "armpits" in the image or interior gaps. If such detection is necessary, a more advanced version of this algorithm could check multiple rectangles per row to account for such interior gaps.

In any case, due to the tediousness of hand-coding collision maps for each image (suppose the image is changed?) it is recommended to have your program generate these collision maps automatically from the image files before collision detection is necessary (like at game load, or even at compilation time using a specialized program to generate header files containing this information).

[[edit](#)] Rectangle Subdivision

Here, each image is separated (preferably by an algorithm; I wouldn't want to do this by hand) into major rectangular components:



Each of the rectangles in each shape are then checked against each other using plain vanilla rectangle intersection. If shape A has n subrectangles, and shape B has p subrectangles, this operation requires $n \cdot p$ rectangle intersection checks.

Depending on the number of subrectangles (like if $n \cdot p$ is more than $A \rightarrow \text{size.y}$ or $B \rightarrow \text{size.y}$), it may eventually be more efficient to use row-rectangle intersection instead, because fewer checks would be required, and each check is simpler to execute.

[[edit](#)] R-Squared Method

If primarily circular objects are being tested (as in a billiards game), it is possible to check for collisions using inequalities derived from the Pythagorean Theorem, which is:

$$(1) x^2 + y^2 = r^2$$

Where r is the radius of the circle, and (x, y) is some point exactly on the circle's circumference.

A point $P(x, y)$ is inside the circle if this inequality is true:

$$(2) P_x^2 + P_y^2 \leq r^2$$

Since taking the square-root of a number is very computationally expensive, we can get away with simply comparing the squares of the numbers as shown above in (2). This check can test a circle for collisions with rectangles, points, and other circles.

For rectangles, it is necessary to check the nearest border and corner point for intersection. The inequality for checking the corner point was related in (2). For checking the closest rectangle border (r_x, y) or (x, r_y) if the circle center is (c_x, c_y) the inequality is:

$$(3a) r_x^2 + c_y^2 \leq r^2 \text{ -- if the border is vertical}$$

$$(3b) c_x^2 + r_y^2 \leq r^2 \text{ -- if the border is horizontal}$$

On an ARM9 processor, integer multiplies take between 2-7 times as long as addition, which is not too bad. I wouldn't be afraid to check all the corners and borders for intersection.

For two circles A and B, the inequality is:

$$(4) (A_x - B_x)^2 + (A_y - B_y)^2 \leq (A_r + B_r)^2$$

Dev-Scene (c)




Accesorios Wii

Envío gratis mundial

US\$

16.54

~~\$23.16~~



NDS/Tutorials Animation

< [NDS](#)

Contents

- [1 Animated sprites for embedded devices](#)
- [2 Getting your feet wet...](#)
- [3 The Manly Way](#)
 - [3.1 Animating: The Manly Way](#)
- [4 The Womanly Way](#)
 - [4.1 Animating: The Womanly Way](#)
- [5 Conclusions: Which One, Which One?](#)
- [6 Notes](#)

[\[edit\]](#) Animated sprites for embedded devices

OK. We've seen simple rectangles and circles drifting across the screen, but it's time for something more dynamic. We are going to learn how to program animated sprites. Instead of a grid of static pixels floating around, we can effectively have small movies all over the screen. Consider the possibilities:

- a bouncy ball that deforms when it hits the screen edge;
- spaceship explosions;
- the heroic samurai brandishing his deadly blade; or
- an evil alien overlord, waxing eloquent over his plot to explode the solar system into the 13th dimension... who is cleverly disguised as your pet bunny rabbit, complete with cute ear twitches and nose wiggles.

Unfortunately, libnds doesn't give you any convenience functions for doing animated sprites (unlike SDL, DirectX and Cocoa in desktop programming environments). This means we are going to learn how to code animated sprites using "manual transmission", creating our own animation functions and understanding what is actually happening in memory.

But don't worry, it's really not that hard.

Animated sprites are made simply by swapping out images in sequence and blitting them to the screen.

That is this whole article in a nutshell. Now, let's go look at some code.

[\[edit\]](#) Getting your feet wet...

Let's start off with looking in the examples folder to see some demo code. Navigate to `nds/Graphics/Sprite/animate_simple/`, then run make and open the executable .nds file in an emulator to see what it does.

It's pretty simple. There are two different sprites that can face four directions each, and there is an animation for walking in each direction, which is triggered whenever you press a button.

If you look in `animate_simple/sprites/`, you'll see two sprite sheets and a grit file. Often, programmers put the frames of an animated sprite in a single file as an organizational aid. Trust me, it simplifies things a lot. For more info about grit files, see [NDS/Tutorials Day 5](#).





There is just one source file, `template.c`. I've divided it up into sections for you here.

↓ `template.c`, lines 45-50

```
#include <nds.h>
#include <man.h>
#include <woman.h>

#define FRAMES_PER_ANIMATION 3
```



After including the `libnds` library, we add in the header files generated by `grit`, `man.h` and `woman.h`. These contain pointers to the bitmap data we need for our sprites. The `#define` just holds how many frames there are for each direction of sprite movement.

Since the man and the woman sprites use different animation techniques, I'll document them in separate sections.

[\[edit\]](#) The Manly Way

The man sprite works by copying the needed section of `man.png` over to VRAM. Then, the sprite is blitted (drawn) to the screen. That is, each frame is copied from RAM to VRAM just before it is blitted.

↓ `template.c`, lines 54-64

```
typedef struct
{
    int x;
    int y;

    u16* sprite_gfx_mem;
    u8* frame_gfx;

    int state;
    int anim_frame;
} Man;
```

This struct has some interesting variables in it:

- `sprite_gfx_mem` — points to a space in VRAM that the blitter will look at when drawing your sprite
- `frame_gfx` — a repository in normal RAM for the entire Man bitmap
- `state` — what direction the sprite is facing. The possible values are in this `enum` statement:

↓ `template.c`, line 89

```
enum SpriteState {W_UP = 0, W_RIGHT = 1, W_DOWN = 2, W_LEFT = 3};
```

- `anim_frame` — how many frames we are into the directional animation. This will range from 0 to `FRAMES_PER_ANIMATION - 1`. The value be used later to calculate where inside `frame_gfx` the next animation frame begins.

Moving on...

↓ `template.c`, lines 112-117

```
void initMan(Man *sprite, u8* gfx)
{
    sprite->sprite_gfx_mem = oamAllocateGfx(&oamMain, SpriteSize_32x32, SpriteColorFor
    sprite->frame_gfx = (u8*)gfx;
}
```

This code isn't anything new; it just allocates some space in VRAM for a single frame of a 32x32 sprite, and stores a pointer to where the bitmap is in normal RAM. Since we've allocated graphics using `&oamMain` the man sprite will be on the top screen.

Now, let's take a look at `main()`. (Don't worry, we'll get to `animateMan()` in a bit.)

↓ `template.c`, line 150

```
Man man = {0,0};
```

We start `man` at `x=0, y=0` (the top left corner of the screen).

↓ `template.c`, line 168

```
initMan(&man, (u8*)manTiles);
```

After setting up the graphics subsystem, we initialize our man with the bitmap data pointed to by `manTiles`. This pointer will be created by `grit` and defined in `man.h` just before compile time.

↓ `template.c`, line 171

```
dmaCopy(manPal, SPRITE_PALETTE, 512);
```

Copy the 512-byte color palette over to VRAM. We use `dmaCopy()` since it uses dedicated hardware and is faster than a `for()` loop (see the main Wikipedia article about [DMA](#)). By the way, for this section of code, I would have used the `#define manPalLen 512` inside `man.h` that `grit` defined for me, just in case I changed the color depth of my source images later on.

[\[edit\]](#) Animating: The Manly Way

Inside the `while()` loop in `main()`, there are first a whole bunch of simple `if()` statements that just change the coordinates of our sprites depending on what D-Pad button is pressed. I won't bother documenting that, because that's easy. After the `if()` statements about the buttons is the interesting stuff:

↓ `template.c`, line 214

```
man.anim_frame++;
```

Move on to the next frame, but...

↓ `template.c`, line 217

```
if (man.anim_frame >= FRAMES_PER_ANIMATION) man.anim_frame = 0;
```

...if we've moved past the last frame in that direction, restart from frame 0.

Now, we call `animateMan(&man);` in line 222. Let's take a look at that function definition.

↓ `template.c`, lines 99-106

```
void animateMan(Man *sprite)
{
    int frame = sprite->anim_frame + sprite->state * FRAMES_PER_ANIMATION;
    u8* offset = sprite->frame_gfx + frame * 32*32;
    dmaCopy(offset, sprite->sprite_gfx_mem, 32*32);
}
```

OK. Now, as you know, computers can't really store 2-D data. So what they really do is they store the 2-D grid of pixels as a 1-D list of numbers. So for an 8x8 bit sprite, it will turn

```
0 1 2 3 4 5 6 7
8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63
```

into a 1-D list of pixels in the order indicated.

In a similar way, `grit` took our sprite frames, which were laid out like so:

```
0 1 2
3 4 5
6 7 8
9 10 11
```

and flattened them into a 1-D string of bytes in memory laid out from frame 0 through frame 11.

Because of the strategic way in which the `SpriteState` enum was defined, our sprite frame number is `sprite->anim_frame + sprite->state * FRAMES_PER_ANIMATION`, as set in the `frame` variable at the beginning of `animateMan()` above.

Also, since each pixel (assuming 256-bit color) is 1 byte, and each frame is 32*32 pixels in size, our offset relative to the beginning of the sprite sheet bitmap is:

```
u8* relativeOffset = 32*32*frame;
```

Because `sprite->frame_gfx` is the address of the start of our sprite sheet bitmap, our total offset is:

```
u8* offset = sprite->frame_gfx + relativeOffset;
```

Compare this to the actual source code above, and you'll see that our code is equivalent to theirs.

Now that we have the correct memory address, we just use `dmaCopy()` to move the correct frame into VRAM.

Now we jump back into `main()`:

↓ `template.c`, lines 230-31

```
oamSet(&oamMain, 0, man.x, man.y, 0, 0, SpriteSize_32x32, SpriteColorFormat_256Col,
man.sprite_gfx_mem, -1, false, false, false, false, false);
```

And we're back in easy territory. We just set the sprite position and other attributes...

↓ `template.c`, lines 236-38

```
swiWaitForVBlank();

oamUpdate(&oamMain);
```

...wait for the screen, and then blit. That's the Manly Way.

[\[edit\]](#) The Womanly Way

The Womanly Way differs from the Manly Way mainly in that the Womanly Way loads the entire sprite sheet into VRAM and just changes the graphics pointer to advance to the next frame. As you can imagine, this is much faster than the Manly Way, where you have to actually copy over the next graphic at every single frame change. The drawback is that the Womanly Way (in this example case) takes 12x the VRAM as the Manly Way, and VRAM is relatively scarce. More on this and related topics later.

↓ `template.c`, lines 72-84

```
typedef struct
{
    int x;
    int y;

    u16* sprite_gfx_mem[12];
    int gfx_frame;

    int state;
    int anim_frame;

}Woman;
```

The difference between the man's `struct` and the woman's `struct` is in the middle two lines of the `struct`. Since the woman sprite is going to be storing all of her data in VRAM, we don't need to keep a pointer to a repository of frames in regular RAM like for the Man. The Man, as you recall, needed this to copy the frame to VRAM, and then the graphics subsystem blitted him to the screen. Since the Woman will already be in VRAM, `sprite_gfx_mem[]` and `gfx_frame` will tell the graphics subsystem where to blit from.

↓ `template.c`, lines 133-143

```
void initWoman(Woman *sprite, u8* gfx)
{
    int i;

    for(i = 0; i < 12; i++)
    {
        sprite->sprite_gfx_mem[i] = oamAllocateGfx(&oamSub, SpriteSize_32x32, SpriteColorFormat_256Col);
        dmaCopy(gfx, sprite->sprite_gfx_mem[i], 32*32);
        gfx += 32*32;
    }
}
```

Note that the Woman's initializer is slower than the Man's, because of the series of calls to

`oamAllocateGfx()` and `dmaCopy()`. (On a picky side point: even though DMA doesn't usually take cycles away from the processor, there are only 4 DMA channels available to the arm9. Especially on larger sprite frames, DMA hardware calls after the fourth channel is taken will have to wait on a previous DMA call to complete before it can get to work itself.) However, the extra time taken is less critical here at load-time than in-game.

[\[edit\]](#) Animating: The Womanly Way

Since much of the Womanly code in `main()` is basically the same as for Man, I'll skip the boring stuff.

↓ `template.c`, lines 124-127

```
void animateWoman(Woman *sprite)
{
    sprite->gfx_frame = sprite->anim_frame + sprite->state * FRAMES_PER_ANIMATION;
}
```

The animation code just changes the `gfx_frame` index, which is faster than the Manly just-in-time call to `dmaCopy()`. Eventually, at the end of the loop in `main()`, `oamSet` is called like so:

↓ `template.c`, lines 233-234

```
oamSet(&oamSub, 0, woman.x, woman.y, 0, 0, SpriteSize_32x32, SpriteColorFormat_256C,
      woman.sprite_gfx_mem[woman.gfx_frame], -1, false, false, false, false, false)
```

The index that was recalculated in `animateWoman()` is used here to point to where the graphics subsystem needs to blit from.

That's it for the Womanly Way.

[\[edit\]](#) Conclusions: Which One, Which One?

By no means are these two methods the only ways to animate sprites. Inside the comments in `template.c` we find:

A more advanced approach is to keep track of which frames of which sprites are loaded into memory during the animation stage, load new graphics frames into memory overwriting the currently unused frames only when sprite memory is full. Decide which frame to unload by keeping track of how often they are being used and be sure to mark all a sprites frames as unused when it is "killed"

However, Rogers also noted in the same comment block that CPU cycles, not to mention DMA, can handle the Manly Way quite ably, and said that he would generally choose the Manly Way over the Womanly Way because of scarce VRAM, compared to main RAM.

That is not to say that Womanly Way should never be used at all. If you would like to free up as many DMA channels as possible and reduce the load on the processor while some complicated, cycle-hogging process is completing between animation frames, the Womanly Way could be better than the Manly Way. For example, if you want to load and initialize the next game map as your hero approaches a boundary, you may want to avoid using DMA for the Man at every single frame, and opt for the Womanly Way instead.

[\[edit\]](#) Notes

Sprites larger than 8x8 pixels are not stored linearly in RAM/VRAM, row by row. Instead, they are broken down into 8x8 sections, and these sections are then laid out in memory left-to-right, top-to-bottom. This means that the pixels that belong in the same row are not contiguous! There will be 8 pixels together here, then 64 locations later, the next 8 pixels in the row will appear. I have written a few functions that will return the position of a pixel inside the sprite data array, given the coordinates of the desired pixel (top-left being the origin). I only wrote functions for square sprites, but the rest can be extrapolated:

```
inline unsigned int sprite8XY(unsigned int x, unsigned int y) {
    return x + (y<<3);
}
inline unsigned int sprite16XY(unsigned int x, unsigned int y) {
    return sprite8XY(x & 0x07, y & 0x07) +
        (x & 0x08)<<3 +
        (y & 0x08)<<4;
}
inline unsigned int sprite32XY(unsigned int x, unsigned int y) {
    return sprite8XY(x & 0x07, y & 0x07) +
        (x & 0x18)<<3 +
        (y & 0x18)<<5;
}
inline unsigned int sprite64XY(unsigned int x, unsigned int y) {
    return sprite8XY(x&0x07,y&0x07) +
        (x&0x38)<<3 +
        (y&0x38)<<6;
}
```

For an interesting discussion of various oddities an NDS developer may encounter while taking advantage of DMA, see Vijn's article about cache and RAM synchronization at <http://www.coranac.com/2009/05/dma-vs-arm9-fight/>.

Dev-Scene (c)

Embedded Systems

www.nabladesigns.com

Consulting / Design / Manufacturing FPGA / Embedded / Linux





Search

Go

Search

History

Edit

Discussion

Article

Dev-Scene

[Home](#)[News](#)[Forum](#)[Blogs](#)[Planet](#)[Library](#)

1

Navigation

[Nintendo DS](#)[- NDS Homebrew Catalog](#)[Nintendo Wii](#)[- Wii Homebrew Catalog](#)[Current events](#)[Community portal](#)[Developers/Donations](#)

Personal tools

[Log in / create account](#)

Toolbox

[Random page](#)[Recent changes](#)[Help](#)[What links here](#)[Upload file](#)[Special pages](#)

Finally: POS with Design

Shouldn't a POS System Look Good? You be the judge. Watch Now!

Watch on YouTube

YouTube

NDS/Tutorials/Captain Apathy/Tiling

[< NDS](#) | [Tutorials](#)

Contents

- [1 Intro](#)
- [2 Tiling](#)
 - [2.1 Getting started in non 32x32 arrangements](#)
 - [2.2 So why 64x64?](#)
 - [2.3 Hey, you said you would talk about scrolling!](#)
 - [2.4 Setting Display order](#)
- [3 Sprites](#)
 - [3.1 Starting out](#)
 - [3.2 Where the sprites information is held](#)
 - [3.3 Movement and Rotation](#)
- [4 Animation](#)
 - [4.1 Sprite Index Changing](#)
 - [4.2 Image uploading](#)
 - [4.3 Sprite Buffer swapping or vram to vram](#)
 - [4.4 So which one should be used?](#)
- [5 Dynamic sized 2d tile systems](#)

[\[edit\]](#) Intro

I'm going to assume you've read the current Day 4 tutorial at least through the first example. There's a bunch of good information there, and I don't want to repeat it if I can avoid it.

[\[edit\]](#) Tiling[\[edit\]](#) Getting started in non 32x32 arrangements

By default, the DS will be using a 32 by 32 tile map. This is enough for a single screen in the x direction, and some leeway in the y direction. There are, however 3 other options.

32x64(BG_32x64)

64x32(BG_64x32)

64x64(BG_64x64)

When wanting to use one of these other options, when setting the BGn_CR register, include what mode to use in the list. For example:

```
BG0_CR = BG_64x64 | BG_COLOR_256 | BG_MAP_BASE(0) | BG_TILE_BASE(1);
```

That tells the DS that we want a map that is 64 tiles wide, and 64 tiles tall.

There is a small caveat; The map is arranged in a series of 2 or 4 32x32 squares. So if using a 64x64 arrangement, to find the location of the tile at x location 3, y location 3, it will be at index $99(3(x)+32*3(y))$. To access the tile at location 50,50, the index is $3666(x-32+(y-32)*32+3072)$.

Confused as to where the 3072 came from? That's from the size of a single map block($32*32$ or 1024), and since we are working on the 4th block, we multiple the a map block size by 3.

[\[edit\]](#) So why 64x64?

The reason we have 64x64 at most in hardware is because the screen is 256 pixels wide which means we need at least 32 tiles to fill the entire screen. We double this to have leeway room when scrolling the screen. Once we have reached a certain point on the screen, we can just start uploading data to either a new map area, and just redirect the hardware to render from there, or start loading the data into off screen parts that we've already left. So for hardware, this is basically the minimum size required and be able to scroll.

[\[edit\]](#) Hey, you said you would talk about scrolling!

I will. Right now.





When scrolling, there are two registers to use to move the map. Those registers are BGn_X0 and BGn_Y0. Simple set those values to the new position, and that's where the hardware will render from.

These are not readable registers however. So it is recommended that you use a variable, and change that, and then store the variable into that register.

Example:

```
//Set up the tiles and the map before this..
//Also, we're assuming we're using BG0.
int scrollX = 0;
while (1) {
    scrollX += 1;
    BG0_X0 = scrollX;
    swiWaitForVBlank();
}
```

[\[edit\]](#) Setting Display order

When using multiple BGs, you may want to display them in a certain order. That's done using the BG_PRIORITY() macro when setting the Control Register(BGn_CR). The higher the number, the sooner it gets drawn, so layers that have a higher priority will get overlaid with layers with a lower priority.

[\[edit\]](#) Sprites

Sprites are nice to have, they're very similar to tiles, but you can move them individually rather than in large groups. I have a Sprite demo working, though I'm not 100% clear on all things sprites, but I thought I would go ahead and share what I've learned. If others can fill in this information better, please do.

[\[edit\]](#) Starting out

To enable sprites, when you set the mode, add DISPLAY_SPR_ACTIVE and DISPLAY_SPR_1D (assuming you're using tile mode). Another thing to do, is to set a bank to the sprite display area. I've been using Bank E (vramSetBankE(VRAM_E_MAIN_SPRITE);).

[\[edit\]](#) Where the sprites information is held

The sprites attributes is stored in the Object Attribute Map. This is an area of the graphics system that holds information on the position of sprites, and rotations. There is enough space there to hold 128 entries on the position and various settings of sprites, and 32 entries on their rotations. The data overlaps each other a bit though. This is how I get around it:

```
SpriteEntry *spriteEntry = new SpriteEntry[128];
SpriteRotation *spriteRotation = (SpriteRotation*)spriteEntry;
```

Now you are able to access the entries and the rotation information separately which is a good thing(tm).

First thing to do is to make sure you initialize the information in the structure.

```
int i;
for (i = 0; i < 128; i++) {
    spriteEntry[i].attribute[0] = ATTR0_DISABLED;
    spriteEntry[i].attribute[1] = 0;
    spriteEntry[i].attribute[2] = 0;
}
for (i = 0; i < 32; i++) {
    spriteRotation[i].hdx = 256;
    spriteRotation[i].hdy = 0;
    spriteRotation[i].vdx = 0;
    spriteRotation[i].vdy = 256;
}
```

Whenever you modify either spriteEntry or spriteRotation, and wish for that to be updated in hardware, you need to update the OAM. This is a simple task:

```
void updateOAM() {
    DC_FlushAll();
    dmaCopy(spriteEntry, OAM, 128 * sizeof(SpriteEntry));
}
```

We now have a place to store the data, and update the data. Now to upload the images. First, to set an entry as active, do this:

```
spriteEntry[index].attribute[0] = ATTR0_COLOR_256 | ATTR0_ROTSCALE;
spriteEntry[index].attribute[1] = ATTR1_ROTDATA(rotindex) | ATTR1_SIZE_64;
spriteEntry[index].attribute[2] = id;
```


Now for what all of that means.

ATTR0_COLOR_256 says we are using 256 colors in this sprite. The other option is ATTR0_COLOR_16 which says we are using 16 colors in tile mode, 16 bit in bitmap mode. ATTR0_ROTSCALE says that this sprite can be rotated and scaled.

ATTR1_ROTDATA indicates what rotation data to use(I haven't confirmed this, but I think that this is accurate...). ATTR1_SIZE_64 says we are using a 64 by 64 sprite.

the id we set in attribute 2 indicates where we are storing the data in ram.

Other options:

- ATTR0_ROTSCALE_DOUBLE: Makes it twice the size, but beyond that, no different then ATTR0_ROTSCALE
- ATTR0_NORMAL: No rotation and scaling for this sprite.
- ATTR0_SQUARE: Square sprite. So if size indicates 64, sprite is 64x64. Default
- ATTR0_WIDE: Sprite is wider then it is tall. So if size indicates 64, sprite is 64x32.
- ATTR0_TALL: Sprite is taller then it is wide. So if size indicates 64, sprite is 32x64
- ATTR0_TYPE_NORMAL: Normal rendering. Pays attention to the alpha bit for if it should be drawn or not. Default
- ATTR0_TYPE_BLENDED: Use alpha blending (for a finer tuning of how things get blended). Recommended reading is here: [\[1\]](#)
- ATTR1_FLIP_X: Flip along the X axis. Only works for ATTR0_NORMAL!
- ATTR1_FLIP_Y: Flip along the Y axis. Only works for ATTR0_NORMAL!
- ATTR1_SIZE_8/16/32: Indicates which size we are using.
- ATTR2_PRIORITY(n): What drawing priority to use. Sprites get drawn on top of layers of the same priority.
- ATTR2_PALETTE(n): Indicates which palette to use(may only work with 16 color sprites)

Ones I haven't played with(I've marked defaults with a '*'):

- ATTR0_TYPE_WINDOWED, ATTR0_BMP (Both of these are with ATTR0_TYPE_NORMAL and ATTR0_TYPE_BLENDED)
- ATTR0_MOSAIC (information on the mosaic effect, and how it's done on the gba can be found here: [\[2\]](#). The DS probably has a similar layout.
- ATTR2_ALPHA(n)

Now that we have the sprite added, now we need to copy the palette and sprite image up to the proper locations. These would be SPRITE_PALETTE, and SPRITE_GFX.

```
dmaCopy(palette, SPRITE_PALETTE, palette_size);
dmaCopy(image, &SPRITE_GFX[id * 16], image_size);
```

Once uploaded, just update the OAM.

[\[edit\]](#) Movement and Rotation

Moving a sprite is relatively easy. To move a sprite, do this:

```
spriteEntry[index].attribute[1] = spriteEntry[index].attribute[1] & 0xFE00 | (x & 0x01FF);
spriteEntry[index].attribute[0] = spriteEntry[index].attribute[0] & 0xFF00 | (y & 0x00FF);
```

Rotations and scaling is done using something called the affine transformation matrix. This gives a lot of ability and power to do many cool translations. But to really make full use, you need to understand linear algebra. I will however opt to simply provide a method to do simple rotations.

```
s16 s = -SIN[angle & 0x1FF] >> 4;
s16 c = COS[angle & 0x1FF] >> 4;

spriteRotation[index].hdx = c;
spriteRotation[index].hdy = -s;
spriteRotation[index].vdx = s;
spriteRotation[index].vdy = c;
```

It should be noted that angle does go from 0 to 512 instead of 0 to 360. To convert from 360 to 512 world, use this: $\text{angle} * 512 / 360$.

For more information on Affine Transformations, I suggest you read [\[3\]](#) as it covers the subject. As a warning, it is written the the gba, however it still applies on the DS.

Don't forget to update the OAM after rotations and translations!

[\[edit\]](#) Animation

There are three good ways to do animation:

- Sprite index changing
- image uploading.
- Sprite buffer swapping.

[\[edit\]](#) Sprite Index Changing

This is a good way to rapidly animate at relatively no cost. Simply modify attribute 2 to what index you wish to use. The problem with this is that you have to keep the different frames in your Sprite Buffer, which can be relatively small. With 64x64 sprites, at 256 bit color, assuming you're using the E bank like I suggested, you have enough space to store 16 sprites. That's not a lot. You can use some of the larger banks to hold 64 sprites, but even that's fairly limited. However, assuming you're using 8x8 sprites, you can hold 1024 sprites with Bank E.

[\[edit\]](#) Image uploading

This method has a higher cost, though you don't have the sprite buffer limitation. It's been reported that there isn't a noticeable slow down at 4 64x64 sprites being uploaded. Also with the dmaCopy, there won't be any tearing effects. Using this technique the animations can be much more advanced, however it's an expensive operation, so it should be avoided for large numbers of objects.

[\[edit\]](#) Sprite Buffer swapping or vram to vram

This method takes up two image banks instead of just one. It's like double buffering for Sprite buffers.

You set one bank to the main sprite, and one to one of the memory locations. The idea is that you upload new sprites to the one in memory (probably using dmaCopy or dmaCopyAsync if you can do it all at once), while using the sprites in the current buffer. When you need some of the animations from the second buffer, you change buffers to the second one. The disadvantage with this is that it increases the complexity of the system, as you have to make sure that each sprite has the next frame or frames of animation in the new buffer.

[\[edit\]](#) So which one should be used?

They all have their advantages and disadvantages. Myself, I would see a blend of index changing and image uploading would probably be the best because they're both simple to implement, and they each solve a problem the other one has, and if used for their advantage, they can get along no problem.

[\[edit\]](#) Dynamic sized 2d tile systems

The hardware is limited to at most 64x64 tile maps. There are a couple of ways we can get around this.

The first is to upload the visible screen every time we move the map around. This has the disadvantage that we would not be making use of the hardware scrolling functionality of the DS, and we're doing a lot of uploads.

Another option is to only upload every so often. This way we do a lot less uploads to the hardware, but we can do better.

The nice thing is that hardware will only be showing at most 33x25 tiles at any one time. All of the off screen tiles are available for modifications.

The reason why I took awhile in completing this tutorial is that I wanted to implement my own tile system. I've finally managed to complete it, so I'll mainly be talking about what I did.

The primary thing that I did to implement the tile map is when you cross a boundary, you load an off screen section of data with the needed data, then do the move. I placed those boundaries at every 64 pixels (aka, I divided the screen into 8ths).

Whenever the user shifts the tile map, I check to see if they have crossed one of those boundaries. If they have crossed one of those boundaries, I then take the section that is just off screen in that direction, and I upload the data to that section's near visible regions. So if you're moving to the right, I upload 8 tiles above the top left corner of what is going to become visible, to 8 tiles below the bottom section. This way I upload 6 squares of data, 4 of which will become visible, and 2 extra regions above and below.

This places a restriction of shifting at most 8 tiles, or 64 pixels in one call. If I wanted to support larger shifts per call, I could simply double, triple, or quadruple the data uploaded. I don't know how valuable that would be, as moving 255 pixels is a large jump that would probably be hard for a player to keep up with.

That's about all I have to say on it. It's fairly simple to do theoretically, but it does have a number of pitfalls to watch out for. I do plan on releasing the libraries I developed for this tutorial, but I currently have a bug or two to work out before I release them to the public. If you're interested in seeing what I have done, and maybe fixing the bug, show up in the dsdev chatroom, and I may be on there under the nickname of CaptainApathy.

Dev-Scene (c)

