INTRODUCTION TO NINTENDO DS PROGRAMMING

A guide for beginning programmers with an interest in programming the Nintendo DS

Jaeden Amero

Winter 2006

v3.0

TABLE OF CONTENTS

PREFACE

The main issue

You love playing games on your Nintendo DS. Every game you've played has been a wonderful experience, each one leaving a lovely aftertaste on your gaming tongue. You may have wondered to yourself what it'd be like to create games, to offer your own software up for licking. You've drawn diagrams of games you'd love to make, worlds you want to share with others. But how to go about it? You think and are lost: you are stuck. Where do I start? What's going on inside that pretty little dual-screen box?

This manual is designed to help you get an idea of what's going on inside the Nintendo DS. With a bit of effort and time, you'll be on your way to creating your own games. Join us, the homebrew community. You'll have a great time giving others a great time, collaborating on projects, and feeling the rush of intense and under pressure coding for numerous coding competitions.

The Solution

This manual is the start of the solution. In it, I will cover the basics of programming the Nintendo DS starting with an explanation of the politics behind the homebrew movement and through the emergence of passthrough devices, how to choose a passthrough device, setting up the programming environment, displaying backgrounds, using sprites, and basic game programming techniques. All these things will be discussed in the context of the creation of a simple game I concocted one weekend entitled "Orange Spaceship."

How to Use this Manual

I assume you know a bit of C or C++ coding. If not, spend at least 20 hours making a variety of programs on your own in addition to completing some tutorials. It doesn't matter what you write, but make sure you have a solid understanding of structs, classes, the heap, loops, bitwise and logical operators. I recommend the following tutorial and reference as a great place to get started learning, http://www.cplusplus.com/doc/tutorial / and http://www.cppreference.com/.

Next, just read through the chapters one by one, making sure you understand the current chapter before moving to the next. Code listings will be on a gray background. Follow along with the code listings, writing your own code based on the listings.

CHAPTER ONE

Politics of the Nintendo DS Homebrew Movement

Background Information

Since the Nintendo DS debut, Nintendo enthusiasts ranging from pre-pubescent kids to 30-year-old college dropouts have been wanting to develop their own games and applications for the Nintendo DS. Nintendo has stated that the DS stands for "Developer's System". For those worthy enough to land a nice developing contract with Nintendo, it truly is. However, most people will never receive this contract, special permission from Nintendo to commercially produce games for the Nintendo DS. In order to obtain a contract with Nintendo, you must prove your worthiness by showcasing an amazing game or other piece of software to them. You must have a stable financial history and expected stable financial future. You must have ample funding to buy all the official Nintendo equipment to develop for the system. Most game development houses don't even get that far. Most games on the market today are put out by what is referred to as a publisher. Game development houses will produce their game partially, show it to a publisher, and the publisher (who already has this development contract with Nintendo) will fund the game development house and market their game. All this bureaucracy makes it very difficult for the common person to produce their own, personal-use games.

This is where the homebrew movement comes in. Dedicated hobbyists spend weeks reverse engineering the Nintendo DS hardware, just to make it possible for common people to produce for the system (by providing a cheap alternative to official Nintendo development). These dedicated hobbyists come from all walks of life and cultures, many from Europe and the U.S., and bring with them great knowledge. These people make up the homebrew movement.

Is Homebrew Legal?

Homebrew is legal for a number of reasons. You own the hardware you reverse engineer, so you are free to do with it as you will so long as you don't break the law. Breaking the law would include breaking proprietary copy protection, pirating video games, publishing trade secrets, or otherwise trying to profit off someone else's hard work. Homebrew poses no threat to the official developer kit, as it is so primitive in comparison. Even if you made something to compete with officially produced hardware, it would be near impossible to publish it. Companies often benefit from homebrew communities. Although software pirates often steal from homebrew discoveries to pirate software, the homebrew community abhors piracy and takes a strong stance against it.

When you buy a piece of hardware, you own it. This means that you are free to break it open, dive into it, reverse engineer it, and so forth. You may void your warranty, but that's the price for learning the intimacies of any system. The only illegal things on this line would be to put into production and sell products made with patented features (without negotiating a production deal with he patent owner), bypassing copy-protection, or stealing software code. Reverse engineering to learn about how the hardware works and to make something fun for the community is totally fine.

The homebrew tools available for game programming are far behind anything the game company who produced the system could provide (the official development kits). Game system developers have an intimate knowledge of the hardware, as they developed it. The homebrew community has only outsider knowledge through experimentation with the hardware.

It would be close to impossible to publish a game made with homebrew tools. Nintendo would not license your game. It would be hard to find another publisher who would try to publish something made with homebrew tools against Nintendo's will. On other systems besides the Nintendo DS, this is also true.

Companies often don't have a problem with homebrew because it increases the demand for their gaming systems and helps them to learn more about their consumer base. One example of this is with the Xbox. The Xbox homebrew community made the Xbox do things that Microsoft never thought consumers wanted, like emulation of classic game systems, running the Linux operating system, and so forth. Microsoft then included a lot of these features (excepting Linux, of course) in their new gaming console, the Xbox 360. If a company wants to squash homebrew developers for whatever reason, they'll be smashing an essential fan base that loves that company's hardware design and has potential to improve it (all at no cost to the company). Homebrew caused such a high demand for the Xbox that it would not have been in Microsoft's best interests to ignore or punish it.

The downside of homebrew is that software pirates often steal from the discoveries of homebrew and use that information to bypass copy-protection and to pirate games. Some companies may take a stance against homebrew for this reason, but doing so is unproductive. Piracy is regrettably inevitable in any industry. It is extremely destructive, annihilating game development houses because publishers will no longer publish their games due to a high piracy rating on the platform the game developers are developing for. Homebrew knows this, and as the amateur brothers of the official game developers, they share the pain. Homebrew will usually keep all information regarding copy-protection in high secrecy; even if they know how to copy games, they will not share the information. The homebrew community does not want to see the death of the system they love come to an early death.

CHAPTER TWO

What is a passthrough device and how do I use one?

Purpose of the Passthrough

The purpose of the pass through is to allow the running of programs on the Nintendo DS through the Game Boy Advance (GBA) cartridge slot. This is done because normal Nintendo DS games, which run from the Nintendo DS (NDS) card slot, are encrypted. Since it might be illegal to break that encryption, as it is a form of proprietary copy protection, we have to get the Nintendo DS to run code from a different place than the NDS card slot. Also, it is much easier to bypass the encryption than to try and break it.

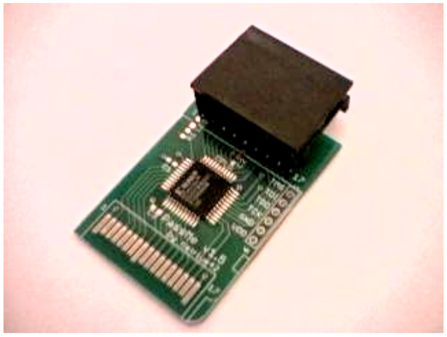
How it works

When the Nintendo DS first boots, it reads a header from the Nintendo DS card inserted into the NDS card slot. The Nintendo DS will read a small block of information from the beginning of the game, known as the header, which contains information about the game (including the title of the game, date it was released and by who, the icon, and some hardware information). This header contains a pointer to a location in memory to begin executing code. The passthrough's job is to read this header and modify it to point to a location someplace on the GBA cartridge inserted into the GBA cart slot. What location would this be? Why, our code of course.

History of the Passthrough

DarkFader (Rafael Vuijk) may have been the first to create a passthrough device. He designed his passthrough with an FPGA (Field Programmable Gate Array) and later packaged it into a single CPLD. Many other people (including Kraln, Firefly, and natrium42) have developed a passthrough independently from DarkFader, but he is self-credited as its initial creator.

Over the next few months following the creation of the passthrough, Lynx and Natrium42, both major players in the early homebrew community, started their own online stores selling pre-made passthrough devices called "PassMe"s. Many budding DS programmers bought their first passthrough from Lynx or Natrium42's stores (myself included). Lynx's store, DSPassme.com, is still up and running today, providing a great option when looking to purchase a passthrough device.



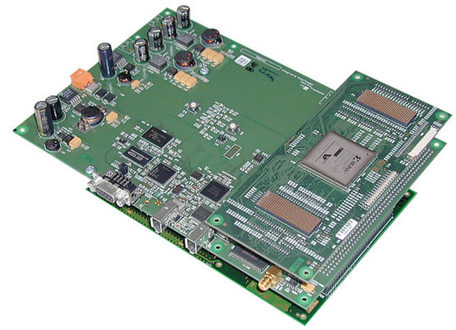


Figure 2.0 Picture of an early PassMe (left) and an FPGA (right)

How do I get a Passthrough?

Today, many passthrough devices are on the market. It is highly recommended by the homebrew community to buy one from a fellow developer and not a company that commercially produces hardware meant for pirating Nintendo DS software. Thus, one should not buy a SuperPass or SuperKey from SuperCard, a MagicKey from NeoFlash, a PassKey from G6Flash, a PassKey or PassCard 3 from M3Adapter, or a Max Media Loader from the horrid Datel (a very unprofessional company run by immature anti-Nintendo kiddies). Each of these companies manufactures hardware that is used to pirate Nintendo DS software and should be avoided at all costs. The best place to buy a passthrough device today is from DSPassme.com (http://www.dspassme.com/oscommerce/catalog/index.php).

Which Passthrough should I buy?

Depending on what kind of Nintendo DS you have, you can buy either of two different types of passthrough devices. The first one, akin to the original made by DarkFader, is the "PassMe". The PassMe will probably work on most Nintendo DS systems.

However, if your Nintendo DS is newer, you will most likely need a "PassMe 2". The need for a PassMe 2 came about because of a change in the firmware of newer Nintendo DS systems. These newer systems do not allow the header to be changed to point to code in the GBA slot. However, it is possible to point to code in the GBA cart's SRAM still. The PassMe 2 points to some certain instructions that have been loaded into memory by the currently inserted DS game. This location is an SWI (software interrupt) call to the SRAM on the GBA port. The GBA cart's SRAM contains the necessary code to jump to code located on GBA cart. The DS then is told to run this code and thus redirects to code in the

GBA slot. Each PassMe 2 has to be programmed to work with one specific game, as the certain code to redirect to the GBA slot is found in different places within different DS games.

There are a few tricks you can pull to determine your firmware version. It involves pictochat and pulling a cartridge out of your Nintendo DS. The screen will change a certain color. Based on this color you can determine if you need a PassMe 2 or not. See Figure 2.1.

Checking the Firmware Version

1. Insert and Nintendo DS game into the DS card slot.
2. Turn on the Nintendo DS. (If you have auto start enabled, hold down Start and Select when you boot the system)
3. Start Pictochat by pressing the Pictochat logo on the menu screen.
4. Join any room you wish.
5. Remove the Nintendo DS game inserted into the back of the system.
6. You will notice either Pictochat freezing or both screens will change color.

Observed Results	Firmware Version
Pictochat Froze	Version 1
Both screens turned periwinkle	Version 2
Both screens turned dark green	Version 3, iQue, or Flashme
Both screens turned marigold	Version 4
Both screens turned magenta	Version 5
Both screens turned dark blue	Version 6

Figure 2.1 When ejecting the game, you'll discover your firmware version as shown.

It is recommended that you purchase a PassMe 2 type passthrough device if you are unsure which to get, as it is guaranteed to work with all Nintendo DS systems. Also, if you have a friend with a newer DS who may need to borrow your passthrough to play your games, it's always helpful to have a passthrough that will work on any system.

When ordering a PassMe 2, be sure to select one pre-programmed to a common game that you own, such as the Metroid Prime Hunters Demo or Super Mario 64 DS. Each PassMe 2 must be programmed to a specific game and it is best to select a game that is in large supply. Also, you'll need a GBA cart with some SRAM on it. Most GBA flash carts have sram, and many older GBA games do as well, if you don't mind losing your save games on that cart.

How do I use my Passthrough?

Using a passthrough device is quite simple. Simply insert a game into the card slot on the passthrough and plug the device into the DS card slot on the top of your DS, like any other game. See Figure 2.1.

Figure 2.2 The PassMe inserted into the DS card slot

What to do with your passthrough

One of the best things to do once you get your passthrough is to install a patch to your Nintendo DS firmware known as “FlashMe.” This firmware patch allows for the running of unsigned wifi binaries as well as redirecting the Nintendo DS to boot from the GBA slot in NDS mode. This means that after installing FlashMe, you no longer need your PassMe. FlashMe is the preferred and recommended method of running your own software. You no longer have to lug around a passthrough or play with a large block sticking out of the back of your DS. Also, you can send your code via wifi if you so desire and avoid having to use a flash cart. In addition to all those things, in case you ever come across some malicious code that zaps your firmware or otherwise messes up your DS, FlashMe keeps a recovery program in the protected firmware space that you can run to save your DS. Because of this feature alone, FlashMe is better than the standard firmware.

CHAPTER THREE

How do I get programs into my Nintendo DS?

The Methods

There are a few ways of getting your code into the Nintendo DS. The first of which is the simple GBA flash cart. These flash carts are generally quite expensive, have a low availability, and don't hold very much memory. They fit into the DS perfectly and do not stick out from the bottom as other things do. The second way of running code is on a removable memory device, such as the M3 Adapter, G6 Flash, NeoFlash, SuperCard, or the GBA Movie Player. The first four of those devices are produced by supporters of piracy and should be avoided. It is recommended to use the GBA Movie Player (GBAMP).

The GBA Movie Player is a wonderful device which can run your software from a Compact Flash card. Compact Flash cards are very cheap and in high supply. If you have more SD cards than CF cards around, unfortunately, the only way to run NDS software from an SD card at this time is with the SD versions of the SuperCard or M3 Adapter. The GBAMP also sticks out from the bottom of the Nintendo DS a little, as shown in Figure 3.0.

Figure 3.0 Comparison of GBAMP (left) and a GBA flash cart (right) inserted into a DS

With the GBA flash cart, the process of loading your programs into memory is a bit slow. Also, each GBA flash cart writer is specific to certain carts and often have closed source drivers. This means that most flash carts will be incompatible with Linux or Macintosh computers. Really, the only good thing about a GBA flash cart is that it does not stick out from the Nintendo DS, as can be seen in Figure 3.0.

Running Multiple Software Titles at Once

If you chose to go with the GBA flash cart, you can use a utility called Darkain's MultiNDS Loader to load multiple programs onto your flash cart. This way, you don't have to re-flash your cart each time you want to run a different application.

If you chose to go with the GBAMP, then I'd highly recommend DragonMinded's DSOrganize. It supports booting multiple programs, text editing, address books, calendars, and more. It's a great application for the DS. You can get it from http://www.youngmx.com/?loc=ndsdev/DSOrganize. However, you cannot use a stock GBAMP to run NDS programs. You have to flash it with some custom firmware. Instructions and firmware are available from http://www.ndshb.com/modules.php?name=Content&pa=showpage&pid=26.

CHAPTER FOUR

How do I create programs?

All About DevkitPro

DevkitPro is a collection of toolchains for homebrew developers. Toolchains are available for Game Boy Advance, GP32, Playstation Portable, GameCube, and the Nintendo DS. The toolchain we are most interested in is known as devkitARM.

DevkitARM is a specific toolchain of devkitPro. It allows the compiling of ARM binaries from most all computers. It is based on gcc, the gnu compiler collection. DevkitARM includes everything you'll need to create software for the Nintendo DS, GBA, and GP32; all of which are run by the ARM processor. However, we will be using something to make our job much easier.

The Wonderful World of Libnds

Libnds, the library for Nintendo DS, started out it's life as NDSLIB. NDSLIB was a simple library created by joat (Michael Noland) and dovoto (Jason Rogers). The name was changed to libnds over the course of a few months and the maintainer has been changed to WntrMute (Dave Murphy).

NDSLIB started out as a collection of defines for common memory locations in the DS. This is useful so you can simply reference BG_BMP_RAM instead of 0x06000000. Eventually, the library began to include structs and other useful constructs that help to simplify the programmers job and abstract certain portions of the hardware from the programmer.

Today, libnds is an incredibly useful library that over 96% of the Nintendo DS homebrew community uses.

Installing DevkitARM

Installing DevkitArm is quite simple. Directions are already documented on their website. Visit http://www.devkitpro.org/setup.shtml for directions. Although more geared towards Windows, the installation is fairly straight forward. Should there be more demand for it, I would be happy to write up more complete instructions for Linux and Macintosh, but do to the brevity of the first edition of this manual, I will not be including directions at this time.

Installing Libnds

Libnds' install is less documented than DevkitPro's install, but is also quite simple.

To install libnds

1. Simply download the latest source from http://sourceforge.net/project/showfiles.php?group_id=114505&package_id=151608

2. Extract it to $DEVKITPRO /libnds.

```
mkdir $DEVKITPRO/libnds
mv libnds-src-*.tar $DEVKITPRO/libnds/
cd $DEVKITPRO/libnds
tar -xvjf libnds-src-*.tar.bz2 $DEVKITPRO/libnds
```

3. Change your current directory to $DEVKITPRO /libnds and type make.

```
cd $DEVKITPRO/libnds
make
```

4. If DevkitARM is installed properly, libnds will compile in a matter of seconds and you'll be on your way to developing software for the Nintendo DS.

The Next Step

Now that you have devkitARM and libnds installed on your computer, you have everything you need to start coding, excepting perhaps a bit of knowledge on how to code. In the next chapter, we'll cover the basics of displaying a bitmap on the screen.

CHAPTER FIVE

How do I display a background?

Initializing the Hardware

In order to get the hardware to do what we want, we have to first initialize it. This means turning on the 2D core and setting up a VBlank IRQ handler. This is where we fall in love with libnds. Libnds makes it incredibly simple to do these two things. Add the following code to your main function in a new main.cpp C++ code file.

```
#include <nds.h>

int main () {
      //turn on the 2D core
      powerON(POWER_ALL_2D);

      //turn on the 2D core
      irqInit();
      irqSet(IRQ_VBLANK, 0);

      return 0;
}
```

Configuring the VRAM Banks

After we get the basic setup done, we now have to set up the graphics hardware to display data how we'd like it to be displayed. Let's make a function called initVideo();

```
void initVideo() {
      /*map vram to display a bg on the main and sub screen and give us lots
        of sprites*/
      vramSetMainBanks( //map A and B to main background memory
                        //this gives us 256KB which is a healthy amount for
                        //  16-bit gfx
                        VRAM_A_MAIN_BG_0x6000000,
                        VRAM_B_MAIN_BG_0x6020000,
                        //map C to sub background memory
                        VRAM_C_SUB_BG_0x6200000,
                        //map D to LCD free space
                        VRAM_D_LCD);


      //map a bank for use with sprites
      vramSetBankE(VRAM_E_MAIN_SPRITE);
      //mapping E to main sprites gives us 64k for sprites
      //(64k is the max space that 1024 tiles take up in 256 color mode)

      //set the video mode on the main screen
      videoSetMode( MODE_5_2D |             //set the graphics mode to Mode 5
                    DISPLAY_SPR_ACTIVE |    //turn on sprites
                    DISPLAY_BG3_ACTIVE |    //turn on background 3
                    DISPLAY_SPR_1D);        //this is used when in tile mode
```

```
	//set the video mode on the sub screen
	videoSetModeSub(MODE_5_2D | DISPLAY_BG3_ACTIVE);

}
```

There are 9 VRAM banks in total on the Nintendo DS. See Figure 5.0 for details about them. Our 16bit background images take up 128KB of memory each. Thus, each background has to have one whole VRAM bank assigned to it. Not all VRAM banks can be used for all purposes, however. Refer to Appendix A, for more detailed information.

VRAM Bank	Control Register Address	Control Register	VRAM Bank Size
VRAM_A	0x04000240	VRAM_A_CR	128KB
VRAM_B	0x04000241	VRAM_B_CR	128KB
VRAM_C	0x04000242	VRAM_C_CR	128KB
VRAM_D	0x04000243	VRAM_D_CR	128KB
VRAM_E	0x04000244	VRAM_E_CR	64KB
VRAM_F	0x04000245	VRAM_F_CR	16KB
VRAM_G	0x04000246	VRAM_G_CR	16KB
VRAM_H	0x04000248	VRAM_H_CR	32KB
VRAM_I	0x04000249	VRAM_I_CR	16KB

Figure 5.0 VRAM Bank Information

Mode 5 Information

Mode 5 is a very common graphics mode on the Nintendo DS. It is very flexible and allows for amazing special effects. Mode 5 consists of four different backgrounds each with their own capabilities. Figure 5.1 shows how flexible Mode 5 can be.

Background	Purpose
0	Tiled Mode, 2D with 3D support
1	Tiled Mode, 2D
2	Extended Rotation Background
3	Extended Rotation Background

Figure 5.1 Mode 5 Information

Setting up the Extended Rotation Backgrounds

Extended rotation backgrounds, exrot bgs for short, extend beyond visible screen space and can be rotated, scaled, sheered and translated. This transformation work is done by what is called an Affine Transformation Matrix. Since linear algebra is beyond the scope of this manual, I will cover briefly how to set up a basic exrot bg, but not get into rotating, scaling, sheering, or translating it. Let's proceed to make a function called initBackgrounds() which will set up our exrot bgs.

```
void initBackgrounds() {
      //setup exrot bg 3 on main as a 16bit color background
      BG3_CR = BG_BMP16_256x256 | BG_BMP_BASE(0) | BG_PRIORITY(3);
      //attributes of the affine translation matrix
      BG3_XDX = 1 << 8; //scale x
      BG3_XDY = 0; //rotation x
      BG3_YDX = 0; //rotation y
      BG3_YDY = 1 << 8; //scale y
      BG3_CX = 0; //translation x
      BG3_CY = 0; //translation y

      //setup exrot bg 3 on sub
      SUB_BG3_CR = BG_BMP16_256x256 | BG_BMP_BASE(0) | BG_PRIORITY(3);
      //attributes of the affine translation matrix
      SUB_BG3_XDX = 1 << 8; //scale x
      SUB_BG3_XDY = 0; //rotation x
      SUB_BG3_YDX = 0; //scale y
      SUB_BG3_YDY = 1 << 8; //scale y
      SUB_BG3_CX = 0; //translation x
      SUB_BG3_CY = 0; //translation y
}
```

The Basics of DMA

DMA stands for Direct Memory Access. DMA allows the reading and writing of memory independently of the CPU. The Nintendo DS has special, dedicated DMA hardware to do quick and efficient moving of memory. Libnds provides us with a few functions to make use of the DMA hardware in the Nintendo DS.

Whenever you have the opportunity to use DMA, you should. It is always better to use DMA than to use a for loop to copy data. When using DMA to copy from main memory, do not forget to flush main memory before using DMA. Another issue would be that in the middle of a DMA, the main CPUs are paused. This can cause awkward bugs with interrupt handling. For this reason, and swifastcopy may be safer, and isn't too much slower. The safest bet is always memcopy and memset, if you are running into some bugs.

The declaration of dmaCopy from libnds is as follows.

```
static inline void dmaCopy(const void * source, void * dest, uint32 size);
```

In our program, we will use dmaCopy to load some graphics into memory. Let's start out by writing some functions to display our backgrounds. As we've already set up the hardware to display the data in the desired manner, right after we copy we will get some nice images displayed on our screens. If we didn't set up our hardware first, we'd most likely get garbage on the screen.

```
void displaySplash() {
      dmaCopy(Splash_bin, BG_BMP_RAM_SUB(0), Splash_bin_size);
      //sub bg is BG_BMP_RAM_SUB(0)
}

void displayStarField() {
      dmaCopy(StarField_bin, BG_BMP_RAM(0), StarField_bin_size);
      //main bg is BG_BMP_RAM(0)
}
```

Working with the Makefile

The default template makefile will turn your graphic files into object files for linking into your program. Never include data as a header file.

The graphics must be in raw binary (.bin) format. Considering the scope of this edition of the manual, I will not be covering how to convert graphics into .bin format. On Linux, image conversion is usually done by a program called gfx2gba. Be warned however that gfx2gba does not properly convert to a 16bit graphics format compatible with the Nintendo DS. Some modification (setting the alpha bit) is still necessary. You can use gfx2gba and a program from PataterSoft called hConvert (http://www.patatersoft.info /hconvert.html) to make the graphics you need. hConvert can convert the output from gfx2gba to a compatible format for use with the Nintendo DS. On the Macintosh, programmers usually use Graphic Converter from Lemke Software (http://www.lemkesoft.de/en/graphcon.htm) saving as a "Byte-Array Header File" and converting that .h file to a .bin with hConvert.

The default libnds template makefile is a good base for most all projects. It will look in a folder called "data" (in the same directory as the makefile) for your graphics. If any are found, it uses a special bin2o rule to convert your images into .o files which can be linked into your program. The bin2o rule will create a header file (.h) for your data. The name format for them works like so: if a file is called "OrangeShip.bin" the header file will be called "OrangeShip_bin.h".

For our project, we'll be putting the .bin graphic files into the data directory and having the makefile run bin2o on them.

Putting in the Star Fields

Let's now put these functions into our main() function to get everything working together.

```
#include <nds.h>

//gfx
#include "StarField_bin.h"
#include "Splash_bin.h"

//other functions we made go here

int main () {
	//turn on the 2D core
	powerON(POWER_ALL_2D);

	//turn on the 2D core
	irqInit();
	irqSet(IRQ_VBLANK, 0);

	initVideo();
	initBackgrounds();

	//display backgrounds
	displayStarField();
	displaySplash();

	return 0;
}
```

Compiling

Check over your code, referring to the included examples if needed. Make sure you have the graphic files in the data directory in your project directory. Bring up the command line and set your current working directory to the directory which contains the makefile for your project. Type "make" and if all goes well, you'll have good success. See figure 5.2. Copy your program to your DS using the method you have chosen.

Figure 5.2 The program should look like this when run.

CHAPTER SIX

What is a sprite? How do I use them?

Magical Fairies?

No, sprites are not magical fairies. It is a term in 2D graphics programming which refers to an image or animation. The Nintendo DS has dedicated hardware for dealing with sprites. This makes the system very useful for 2D. Most gaming systems do not have a 2D core, and all sprites and other 2D graphics have to be handled manually by painting 2D images on the side of a quad within 3D space.

The OAM

The OAM manages all the sprites. This is an immense job, most of it done behind the scenes for us. After using it, you might think it magical. It's not as magical as it seems, however. OAM stands for Object Attribute Memory. It is the place in memory we use to keep track of and control our sprites. The OAM works with a SpriteEntry and a SpriteRotation struct to manage the attributes of our sprites.

Sprite Attributes

Sprites can spin and flip and mosaic and all sorts of fun hardware effects. We will cover how to update, initialize, move, and rotate only. Let's write some functions to do each of those, in that order. Our first step will be to create a new header file. Let's call it "Sprites.h". Put that file into the include folder in your project directory.

Updating the OAM

Updating the OAM is very straightforward. First, we flush local memory (a must whenever performing a DMA operation). Then, we tell the OAM to look into the SpriteEntry struct we will create later for information about each one of our sprites.

```
//Update the OAM
void updateOAM(SpriteEntry * spriteEntry) {
      DC_FlushAll();
      dmaCopy(spriteEntry, OAM, 128 * sizeof(SpriteEntry));
}
```

Initializing the OAM

The first thing we do when initializing the OAM is to clear all the sprite data in the OAM. After that, we'll make a call to our afore written updateOAM function.

```
//Initialize the OAM
```

```
void initOAM(SpriteEntry * spriteEntry, SpriteRotation * spriteRotation) {
	//For all 128 sprites on the DS, disable and clear any attributes they
	//might have. This prevents any garbage from being displayed and gives
	//us a clean slate to work with.
	for(int i = 0; i < 128; i++) {
		spriteEntry[i].attribute[0] = ATTR0_DISABLED;
		spriteEntry[i].attribute[1] = 0;
		spriteEntry[i].attribute[2] = 0;
	}
	//For all 32 possible sprite rotations, set them to default values
	for(int i = 0; i < 32; i++) {
		spriteRotation[i].hdx = 256;
		spriteRotation[i].hdy = 0;
		spriteRotation[i].vdx = 0;
		spriteRotation[i].vdy = 256;
	}
	updateOAM(spriteEntry);
}
```

Moving Sprites

Now for some real fun. Moving sprites in hardware, and not having to worry about clipping, buffers, or anything, is such a wonderful feeling. To move a sprite, we simply change some attributes in that sprite's SpriteEntry. Attribute 1 in a sprite always contains, in bits 0-8, the X position of the sprite. Attribute 0, among other things, contains the Y position of the sprite, in bits 0-7.

```
//Move a Sprite
void moveSprite(SpriteEntry * spriteEntry, u16 x, u16 y) {
	spriteEntry->attribute[1] &= 0xFE00;
	spriteEntry->attribute[1] |= (x & 0x01FF);

	spriteEntry->attribute[0] &= 0xFF00;
	spriteEntry->attribute[0] |= (y & 0x00FF);
}
```

Rotating Sprites

Let's get to spinning. This is a bit more difficult, but still fun. It's always nice to not have to make a separate sprite for each rotation position the sprite will be presented in. We have to use a transformation derived from our time spent playing with an affine transformation matrix. Lucky for you, I did this already. If you have a background in linear algebra, I'd recommend reading up on this portion of the hardware at http://user.chem.tue.nl/jakvijn/tonc/affine.htm.

```
//Rotate a Sprite
void rotateSprite(SpriteRotation * spriteRotation, u16 angle) {
      s16 s = -SIN[angle & 0x1FF] >> 4;
      s16 c =  COS[angle & 0x1FF] >> 4;

      spriteRotation->hdx =  c;
      spriteRotation->hdy = -s;
      spriteRotation->vdx =  s;
      spriteRotation->vdy =  c;
}
```

Using the Sprites

Now that our Sprites.h file is finished, let's get on to how sprites are stored in memory, how to load them, and so forth. So put away your Sprites.h file into the include directory of your home folder and let's get back into our main.cpp file.

How sprites are stored in memory

Sprites are broken into 8x8 pixel pieces. This is called tiling. When drawn to screen, the hardware pieces these tiles together, like a puzzle where the pieces have no distinguishing edges. See figure 6.0 for information about layout. The conversion process will not be covered in the early editions of this manual, as it's a more involved process. On Linux, programmers usually use gfx2gba. On the Macintosh, programmers usually use Graphic Converter from Lemke Software (http://www.lemkesoft.de/en/graphcon.htm) saving as a "Byte-Array Header File" and a program from PataterSoft called hConvert (http://www.patatersoft.info /hconvert.html).

```
const u16 data[] = {
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0F0F, 0x0F0F, 0x0F0F, 0x0F0F, 0x0F0F, 0x0F0F, 0x0F0F, 0x0F0F,
0x2020, 0x2020, 0x2020, 0x2020, 0x2020, 0x2020, 0x2020, 0x2020,
0x2F2F, 0x2F2F, 0x2F2F, 0x2F2F, 0x2F2F, 0x2F2F, 0x2F2F, 0x2F2F,
0x4040, 0x4040, 0x4040, 0x4040, 0x4040, 0x4040, 0x4040, 0x4040,
0x4F4F, 0x4F4F, 0x4F4F, 0x4F4F, 0x4F4F, 0x4F4F, 0x4F4F, 0x4F4F,
0x6060, 0x6060, 0x6060, 0x6060, 0x6060, 0x6060, 0x6060, 0x6060,
0x6F6F, 0x6F6F, 0x6F6F, 0x6F6F, 0x6F6F, 0x6F6F, 0x6F6F, 0x6F6F,
0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080,
0x8F8F, 0x8F8F, 0x8F8F, 0x8F8F, 0x8F8F, 0x8F8F, 0x8F8F, 0x8F8F,
0xA0A0, 0xA0A0, 0xA0A0, 0xA0A0, 0xA0A0, 0xA0A0, 0xA0A0, 0xA0A0,
0xAFAF, 0xAFAF, 0xAFAF, 0xAFAF, 0xAFAF, 0xAFAF, 0xAFAF, 0xAFAF,
0xC0C0, 0xC0C0, 0xC0C0, 0xC0C0, 0xC0C0, 0xC0C0, 0xC0C0, 0xC0C0,
0xCFCF, 0xCFCF, 0xCFCF, 0xCFCF, 0xCFCF, 0xCFCF, 0xCFCF, 0xCFCF,
0xE0E0, 0xE0E0, 0xE0E0, 0xE0E0, 0xE0E0, 0xE0E0, 0xE0E0, 0xE0E0,
0xEFEF, 0xEFEF, 0xEFEF, 0xEFEF, 0xEFEF, 0xEFEF, 0xEFEF, 0xEFEF};

const u16 data[] = {
0x0000, 0x0000, 0x0000, 0x0000, 0x0F0F, 0x0F0F, 0x0F0F, 0x0F0F,
0x2020, 0x2020, 0x2020, 0x2020, 0x2F2F, 0x2F2F, 0x2F2F, 0x2F2F,
0x4040, 0x4040, 0x4040, 0x4040, 0x4F4F, 0x4F4F, 0x4F4F, 0x4F4F,
0x6060, 0x6060, 0x6060, 0x6060, 0x6F6F, 0x6F6F, 0x6F6F, 0x6F6F,
0x0000, 0x0000, 0x0000, 0x0000, 0x0F0F, 0x0F0F, 0x0F0F, 0x0F0F,
0x2020, 0x2020, 0x2020, 0x2020, 0x2F2F, 0x2F2F, 0x2F2F, 0x2F2F,
0x4040, 0x4040, 0x4040, 0x4040, 0x4F4F, 0x4F4F, 0x4F4F, 0x4F4F,
0x6060, 0x6060, 0x6060, 0x6060, 0x6F6F, 0x6F6F, 0x6F6F, 0x6F6F,
0x8080, 0x8080, 0x8080, 0x8080, 0x8F8F, 0x8F8F, 0x8F8F, 0x8F8F,
0xA0A0, 0xA0A0, 0xA0A0, 0xA0A0, 0xAFAF, 0xAFAF, 0xAFAF, 0xAFAF,
0xC0C0, 0xC0C0, 0xC0C0, 0xC0C0, 0xCFCF, 0xCFCF, 0xCFCF, 0xCFCF,
0xE0E0, 0xE0E0, 0xE0E0, 0xE0E0, 0xEFEF, 0xEFEF, 0xEFEF, 0xEFEF,
0x8080, 0x8080, 0x8080, 0x8080, 0x8F8F, 0x8F8F, 0x8F8F, 0x8F8F,
0xA0A0, 0xA0A0, 0xA0A0, 0xA0A0, 0xAFAF, 0xAFAF, 0xAFAF, 0xAFAF,
0xC0C0, 0xC0C0, 0xC0C0, 0xC0C0, 0xCFCF, 0xCFCF, 0xCFCF, 0xCFCF,
0xE0E0, 0xE0E0, 0xE0E0, 0xE0E0, 0xEFEF, 0xEFEF, 0xEFEF, 0xEFEF};
```

Figure 6.0 The upper text shows information as it would be on a non-tiled background. The lower text shows the same data, tiled, for use in tiled graphic modes.

Loading in a Sprite

Now, to see a sprite in action. Let's load in the OrangeShuttle graphic. Make a new function called initSprites. Make sure to include OrangeShuttle_bin.h. The first step is to initialize the OAM. After that, we assign a graphics ID to our ship. After that, we set the initial sprite attributes for our sprite (which we will place in the spriteEntry struct as index 0). Then, the rotation attributes. Next, we simply copy over the palette data and then the graphics data. Lastly,

```
void initSprites(SpriteEntry * spriteEntry, SpriteRotation * spriteRotation) {
      //init OAM
      initOAM(spriteEntry, spriteRotation);

      Coordinate position;
      position.x = SCREEN_WIDTH/2 - 64;
      position.y = SCREEN_HEIGHT/2 - 64;

      //create the ship sprite
      int orangeShipGfxID = 64;

      spriteEntry[0].attribute[0] = ATTR0_COLOR_256 |
                                    ATTR0_ROTSCALE_DOUBLE | //able to
                                                             rotscale
                                    (int)position.y;

      spriteEntry[0].attribute[1] = ATTR1_ROTDATA(0) |
                                    ATTR1_SIZE_64 | //size 64x64
                                    (int)position.x;

      spriteEntry[0].attribute[2] = orangeShipGfxID;

      //set initial rotation attributes
      rotateSprite(&spriteRotation[0], 0);

      //copy in the sprite palettes
      dmaCopy(OrangeShuttlePalette_bin, //from address
              (uint16 *)SPRITE_PALETTE, //to address
              OrangeShuttlePalette_bin_size); //size of data to copy

      //copy the sprite graphics in obj graphics mem
      dmaCopy(OrangeShuttle_bin,          //from address
              &SPRITE_GFX[orangeShipGfxID * 16], //to address
              OrangeShuttle_bin_size);    //size of data to copy

      //update the OAM
      updateOAM(spriteEntry);
}
```

Displaying the Sprite

In our main function, we now need to create the structs which hold our sprite data. Then, we'll make a call to the initSprites function we just created.

```
#include <nds.h>

//gfx
#include "StarField_bin.h"
#include "Splash_bin.h"
#include “OrangeShuttle_bin.h”
#include "OrangeShuttlePalette_bin.h"

//other functions we made go here

int main () {
	//turn on the 2D core
	powerON(POWER_ALL_2D);

	//turn on the 2D core
	irqInit();
	irqSet(IRQ_VBLANK, 0);

	initVideo();
	initBackgrounds();

	//display backgrounds
	displayStarField();
	displaySplash();

	//create the sprite entry table
	SpriteEntry * spritesMain = new SpriteEntry[128];

	//create the sprite rotation table, assigning it to the same location
	//as spritesMain because the attributes overlap in memory
	SpriteRotation * spriteRotationsMain = (SpriteRotation *)spritesMain;

	//load and init the sprites (just one sprite actually)
	initSprites(spritesMain, spriteRotationsMain);


	return 0;
}
```

COMPILING

If all goes well, you’ll compile with no problems and the output will look as in Figure 6.1.

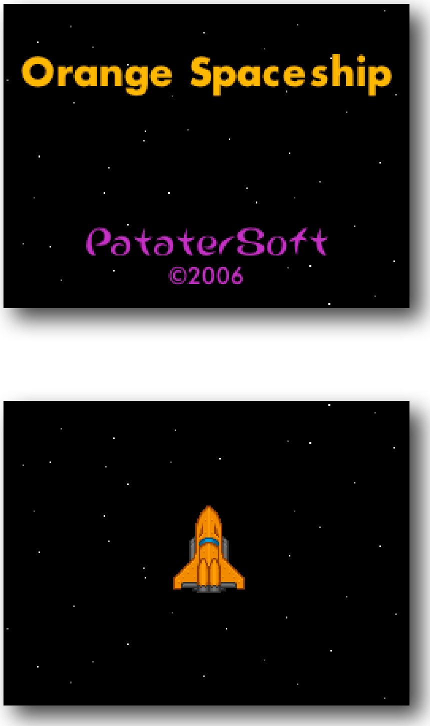


Figure 6.1 Output with both backgrounds and a sprite.

Chapter Seven

Basic Game Mechanics Applied to the Space Shooter Genre

The Importance of Object Oriented Programming

Object oriented programming (OOP) is essential to making good games on a modern system. Although it is very much possible without object oriented programming, OOP is an incredible tool that greatly improves code reusability, readability, modularization, and abstraction. It makes the programmer's job a lot easier. Also, due to modularization, collaborating on projects with your friends or coworkers is easily ten fold easier.

The Ship Class

The first thing we'll make is a Ship class. This class will encapsulate all the properties and functionality of any ship in an easy to use and understand format. Think of things a ship can do, on a high level. What should come to mind is the ability to turn both ways, shoot weapons, accelerate, move at a given velocity (coasting), and maybe some more things if you are creative enough. What properties of a ship can you come up with? Perhaps turning speed, thrust, mass, maximum speed, velocity, position, shields? Well, after you are done brainstorming, the next step is to write out the functionality and properties we need to put into our Ship class. You could make a table, as in Figure 7.0, or draw some diagrams on a piece of paper. Either way, you want to make sure your ideas all get onto some physical paper.

Properties	Functionality
shipHeight	accelerate
shipWidth	moveShip
position	turnClockwise
velocity	turnCounterClockwise
angle	getPosition
turnSpeed	reverseTurn
thrust	getAngle
maxSpeed	
mass	

Figure 7.0 Table of Ship properties and functionality.

Making the Ship Class

I have provided a skeleton framework file for you to write your class in. It is all set and ready for you to implement in the Ship.cpp file. The header file, Ship.h is also included. On your own, with your own classes in the future, you should always make a skeleton framework class to work from. It makes implementation straightforward and you do not have to worry about the semantics of setting up a class so much.

The Constructor

I have provided you with a simple constructor, copy constructor, = operator, and private init method. These are often mundane things to make. Feel free to modify the default values to try out different effects of changing the ship properties.

Acceleration

Acceleration is probably one of the most important things your ships can do. To accelerate, we simply increase out velocity by a certain increment, that being the thrust capability of the ship, in the angle we are headed. Here is where some simple trig comes into play. Since our velocity is stored as an x and y component, we have to shadow our thrust vector onto each direction. We do this we multiply the thrust by sin(angle) for our x component, and by -cos(angle) for the y direction. Next, after we have computed the increment for both x and y, we add them onto our current velocity, making sure we don't go over the ship's maximum speed.

```
void Ship::accelerate() {

      float incX = thrust * sin(angle);
      float incY = -(thrust * cos(angle));

      velocity.x += incX;

      //make sure can't go too fast in x direction
      if (velocity.x > maxSpeed) {
            velocity.x = maxSpeed;
      }
      if (velocity.x < -maxSpeed) {
            velocity.x = -maxSpeed;
      }

      velocity.y += incY;

      //make sure can't go too fast in y direction
      if (velocity.y > maxSpeed) {
            velocity.y = maxSpeed;
      }
      if (velocity.y < -maxSpeed) {
            velocity.y = -maxSpeed;
      }
```

```
}
```

Moving the Ship

This one is incredibly easy thanks to the Nintendo DS hardware. All we have to do is increment by our velocity. The hardware takes care of any wrapping or offscreen issues.

```
void Ship::moveShip() {
	//move the ship
	position.x += velocity.x;
	position.y += velocity.y;

	//hw does wrap around for us
}
```

Reversing the Ship's Direction

This one took me a while to figure out, even though it's just one line, but it's very useful. We can turn the ship around, not a 180 per se, but simply pointing into the opposite direction of our current velocity. This will get the angle of our velocity with respect to 0 degrees, and then will do a 180 from that angle.

```
void Ship::reverseTurn() {
	angle = (2 * PI) - atan2(velocity.x, velocity.y);
}
```

Rotating the Ship

Rotating the ship is also quite simple. We just increment or by ship's turning speed depending on which direction we wish to turn.

```
void Ship::turnClockwise() {
	angle += turnSpeed;
}

void Ship::turnCounterClockwise() {
	angle -= turnSpeed;
}
```

Getting the Ship's Position

Return the ship's position.

```
Coordinate Ship::getPosition() {
	return position;
}
```

Getting the Ship's Angle

This one is a bit more tricky and involved. I suppose I should start by explaining that a Nintendo DS circle has 512 degrees. It doesn't actually have 512 degrees, nor does a Nintendo DS even know what a circle is, but it is easy to understand the hardware a bit better when we think of it this way. I will say, however, that the reason for the 512 degrees is due to libnds's built-in look up tables for the sin and cos functions. In order for the Nintendo DS to know how to rotate our sprites, we have to convert the internally stored radian angle value to a 512 degree system. This is an easy conversion.

The first step is to convert to a 360 degree system, as you must have learned in junior high school. This is done by multiplying the radian value by 180/π. The 180 part is half the number of degrees in a circle. So, in a 512 degree system we can convert by multiplying the radian value by 256/π. Lastly, just return that value as an integer (the hardware does not have any floating point, so when rotating our sprites, must use a fixed point value disguised as an integer).

Then, we make a function to return a converted angle value, for whenever we need it.

```
int Ship::radToDeg512(float rad) {
      return rad * (256/PI);
}

int Ship::getAngleDeg512() {
      return radToDeg512(angle);
}
```

Linking the Ship into Our Program

First off, we need to modify our initSprites function to use our ship class to keep track of where to draw the sprite and how.

```
void initSprites(Ship * ship, SpriteEntry * spriteEntry, SpriteRotation *
spriteRotation) {
      //init OAM
      initOAM(spriteEntry, spriteRotation);

      //get the ship's initial position
      Coordinate position = ship->getPosition();

      //create the ship sprite
      int orangeShipGfxID = 64;

      spriteEntry[0].attribute[0] = ATTR0_COLOR_256 |
                                    ATTR0_ROTSCALE_DOUBLE | //able to
                                                            rotscale
                                    (int)position.y;

      spriteEntry[0].attribute[1] = ATTR1_ROTDATA(0) |
                                    ATTR1_SIZE_64 | //size 64x64
                                    (int)position.x;

      spriteEntry[0].attribute[2] = orangeShipGfxID;

      //set initial rotation attributes
      rotateSprite(&spriteRotation[0], ship->getAngleDeg512());
```

```
    //copy in the sprite palettes
    dmaCopy(OrangeShuttlePalette_bin, //from address
            (uint16 *)SPRITE_PALETTE, //to address
            OrangeShuttlePalette_bin_size); //size of data to copy

    //copy the sprite graphics in obj graphics mem
    dmaCopy(OrangeShuttle_bin,          //from address
            &SPRITE_GFX[orangeShipGfxID * 16], //to address
            OrangeShuttle_bin_size);    //size of data to copy

    //update the OAM
    updateOAM(spriteEntry);
}
```

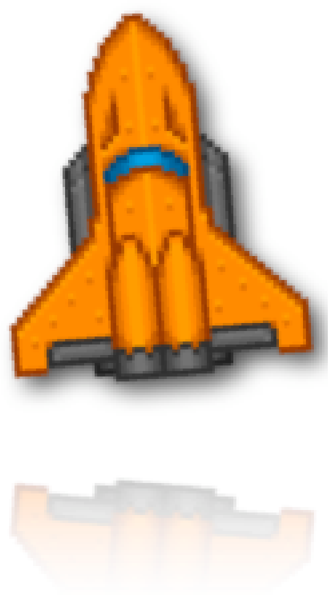
We now need to create an instance of the ship in our main function. Creating an instance of a class, known as an object, is quite simple.

```
#include <nds.h>

//gfx
#include "StarField_bin.h"
#include "Splash_bin.h"
#include “OrangeShuttle_bin.h”
#include "OrangeShuttlePalette_bin.h"

#include “Ship.h”

//other functions we made go here

int main () {
    //turn on the 2D core
    powerON(POWER_ALL_2D);

    //turn on the 2D core
    irqInit();
    irqSet(IRQ_VBLANK, 0);

    initVideo();
    initBackgrounds();

    //display backgrounds
    displayStarField();
    displaySplash();

    //make the ship of size 64x64 pixels
    Ship * ship = new Ship(64, 64);

    //create the sprite entry table
    SpriteEntry * spritesMain = new SpriteEntry[128];

    //create the sprite rotation table, assigning it to the same location
    //as spritesMain because the attributes overlap in memory
    SpriteRotation * spriteRotationsMain = (SpriteRotation *)spritesMain;

    //load and init the sprites (just one sprite actually)
    //pass in the ship this time
    initSprites(ship, spritesMain, spriteRotationsMain);
```

```
	return 0;
}
```

COMPILING

Everything should compile for you fine at this point if you wish to play around with your new class. However, in the next chapter we will cover how to get Nintendo DS input to affect the Ship. Be ready for it, we're going to have some major fun.



CHAPTER EIGHT

Nintendo DS Input Systems

Overview

The Nintendo DS has many different user input systems, including buttons, touch screen, and a microphone. Most video game systems only have buttons and an analog stick or two. While the Nintendo DS does not have an analog stick, it does have an amazing touch screen which has millions of different creative uses. In the first edition of this manual, I will only cover buttons though. If you wish to learn more about the touch screen and the microphone, I'd recommend reading
http://www.bottledlight.com/ds/index.php/Misc/TouchScreen and although a bit outdated, the only microphone resource I know of
http://www.double.co.nz/nintendo_ds/nds_develop9.html.

Key Input

Libnds provides us with a very nice abstraction for key input. Instead of having to AND registers with cryptic masks to discover which keys we are pressing, we simply call scanKeys(), then check one of three input functions, keysDown(), keysHeld(), or keysUp(). In order to see which keys have been recently pressed, use keysDown(). To see which keys are currently held, use keysHeld(). To see which keys have just been released, use keysUp(). Libnds provides us with defines for some key masks as well. How they are set up is explained in Figure 8.0.

Key Define	Mask Bit	Associated Input
KEY_A	1 << 0	A Button
KEY_B	1 << 1	B Button
KEY_SELECT	1 << 2	Select Button
KEY_START	1 << 3	Start Button
KEY_RIGHT	1 << 4	Right D-pad
KEY_LEFT	1 << 5	Left D-pad
KEY_UP	1 << 6	Up D-pad
KEY_DOWN	1 << 7	Down D-pad
KEY_R	1 << 8	R Button

Key Define	Mask Bit	Associated Input
KEY_L	1 << 9	L Button
KEY_X	1 << 10	X Button
KEY_Y	1 << 11	Y Button
KEY_TOUCH	1 << 12	Pen Touching Screen (no coordinates)
KEY_LID	1 << 13	Lid shutting (useful for sleeping)

Figure 8.0 Libnds key defines.

Writing an Input Updating Function

Know that we know a bit about how input is laid out on the Nintendo DS, let's write a function in our main.cpp to handle the input. We'll call it handleInput. First, we want the ship to accelerate when we press up. Do do this, we detect when the Nintendo DS has the Up Key on the D-pad held (which included the initial down press) and accelerate the ship if so. The Up key will constantly read as held, so long as it is held. Reading the input does not affect the keys register. We'll do similar things for each of the other keys. See if you can tell what each key does from the code listing below.

```
void handleInput(Ship * ship) {

      //up
      if (keysHeld() & KEY_UP) {
            //accelerate ship
            ship->accelerate();
      }

      //down
      if (keysHeld() & KEY_DOWN) {
            //reverse ship direction
            ship->reverseTurn();
      }

      //left
      if (keysHeld() & KEY_LEFT) {
            //rotate counter clockwise
            ship->turnCounterClockwise();
      }

      //right
      if (keysHeld() & KEY_RIGHT) {
            //rotate clockwise
            ship->turnClockwise();
      }

      return;
```

```
}
```

As you've noticed, having that Ship class made input handling extremely easy. Our keys will directly affect various properties of the ship as we press them. This is really amazing, but the true miracle is yet to come.

Creating the Main Game Loop

Let's check back in on our main function now. It's time for use to create an infinite loop to run our program. The first thing we want to happen in our game loop is for the key registers to get updated. We make a call to scanKeys and it all happens for us. Next, we handle the input we just received by calling our recently created handleInput function, passing in our ship object that it can change our ship for us. Next, we tell our ship to move at it's current velocity. This will change the ship's position. Then we update the sprite attributes with new information about our ship, as handleInput most likely changed some properties about the ship. Finally, we call a function that will make sure our program does not exceed 60fps (speed of the graphics on the Nintendo DS), and update the OAM, telling it that we changed some attributes on the sprites and it need to handle that.

```
//former functions above here

int main() {

      //our former code here

      for (;;) {
            scanKeys();

            handleInput(ship);

            ship->moveShip();

            //update sprite attributes
            Coordinate position = ship->getPosition();
            moveSprite(&spritesMain[0], (int)position.x, (int)position.y);
            rotateSprite(&spriteRotationsMain[0], ship->getAngleDeg512());

            swiWaitForVBlank();

            updateOAM(spritesMain);
      }
}
```

The OAM really shines through here. The all powerful Nintendo DS hardware, an incredible masterpiece, will rotate and move our ship with very little effort on our part. In hindsight, all we have done is flip a few bits in a few registers in a structured manner, and our ship comes to life. Incredible.

COMPILING

Now we can control our ship with the D-Pad. What fun! The game should now appear as in Figure 8.1. Now if only we had some aliens to kill...

Figure 8.1 Flying around in the Orange Shuttle.

CHAPTER NINE

What about the sounds?

A Sound Theory

Sounds are essential in any game. Our little project should be no exception. Sounds bring life to various elements in the game, such as space ships, weapon systems, rain, sword clashing, car engines, and so forth.

Many games don't need music. For instance, a game with the programmer's or game designer's favorite song may annoy many players who have different tastes of music. Instead, give the player the option to listen to their own music, on their own audio equipment. Not to mention, music is costly in data size.

Sound effects on the other hand, are quite useful. A word of warning, however, a game with bad sounds (scratchy, annoying, too repetitive, etc.) is worse than a silent game. Take care when engineering sounds for your game.

The Hardware

The Nintendo DS has amazing sound hardware. We will only be able to scratch the surface of it, and even still, we won't leave any scratch marks. The Nintendo DS has 16 channels to play sounds, numbered 0 through 15. Channels 8-15 are special channels. All channels can play ADPCM or PCM sounds, while 8-13 are the only channels that can play PSG sounds and 14-15 the only that can generate noise. We will only be experimenting with PCM sounds.

Making the Sounds

To play our sounds, we need to convert them into a PCM format that the DS will understand. The format we will use is 8-bit signed 22050Hz raw audio. You can experiment with other combinations if you'd like, but in the interest of keeping this manual simple we'll just be using the aforementioned format.

There are a myriad of programs our there to do sound conversion for us, but I will present the most universal method of sound conversion: the open source program *SoX*. SoX stands for Sound eXchange. It is available for Linux, Macintosh OS X, and Windows. I have included pre-compiled versions for Mac OS X PPC, Mac OS X x86, and Windows. For Linux, just compile the included SoX source.

To use SoX, simple convert your file as shown below. You can practice with my included audio file "thrust.aif". The relevant command line options work as follows: -b 8-bit, -s signed, -r 22050 22050Hz sample rate. SoX knows to turn your sound into a .raw by reading the file extension on the output filename. Keep in mind since raw sounds have no headers, to somehow label them with their formatting information. I use the filename to do this, as you can see.

```
      sox sounds/thrust.aif -bsr 22050 data/thrust_1_22050_8_s.raw
```

Using the Sounds

The makefile I've included with my manual has a custom bin2o rule for creating .o files from .raw files. There is really nothing special about it. It does the same thing that the .bin bin2o rule does. The libnds default makefiles require you to explicitly specify the file types you want to be processed.

After the makefile processes the raw sound file, the header for it will look like so "thrust_1_22050_8_s_raw.h" and will be located in the build directory.

Getting Down with the Code

Now it's finally time for some code. Libnds simplifies sound playing quite a bit. It provides a struct called "TransferSoundData" that we can use to encapsulate our sound. In a way, it acts as a header to let the DS know how to handle it. Libnds will handle the transfer of the sound to the ARM7 (the only processor that can control the sound hardware), although it is a bit inefficient, frumpy (and yes, for those of you who don't know, libnds is a girl), and limited. It will serve our purposes for now, as a springboard into more advanced forms of sound. Enough with the chatter, here's some code for our main.cpp.

```
//... previous code here
//snd
#include "thrust_1_22050_8_s_raw.h"
//... previous code here

void handleInput(Ship * ship, TransferSoundData * soundData) {

      //up
      if (keysDown() & KEY_UP) {
            //play our sound only when the button is initially pressed
            playSound(soundData);
      }
      if (keysHeld() & KEY_UP) {
            //accelerate ship
            ship->accelerate();
      }
      //... previous code here
}
//... previous code here
int main() {
      //... previous code here

      //immediately following sprite setup
```

```
    /* Sound Data setup */
    TransferSoundData * thrust_sound = new TransferSoundData();
    thrust_sound->data = thrust_1_22050_8_s_raw;
    thrust_sound->len = thrust_1_22050_8_s_raw_size;
    thrust_sound->rate = 22050; //22050Hz Sample Rate
    thrust_sound->vol = 127; //Volume can be from 0 to 127
    thrust_sound->pan = 64; //Sound comes equally from both speakers
                            // (0-127, left to right)
    thrust_sound->format = 1; //1 - 8bit, 0 - 16bit
    for (;;) {
        updateInput();

        handleInput(ship, thrust_sound);
        //... previous code here
    }
}
```

In summary, we simply set up our sound in the main function, modified the handleInput function to accept a sound pointer, and had the handleInput function play our sound whenever the up key is held.

Wait, What's Going on Here?

Now, for an explanation of what libnds is doing behind the scenes. A pointer to your sound data in shared memory (iwram?) is given to the arm7 from the arm9 via the IPC. If the pointer is null, then no sound will be played. If the pointer is not null, the arm7 finds an open sound channel. If any channels are open, it reads the header information from your TransferSoundData struct and then calls an internal function, startSound, which plays the sound for us.

The startSound function is not very flexible (nor is the TransferSoundData struct). The sound can only be 8-bit or 16-bit. We have no way of signaling the arm7 that we want the sound to loop, unfortunately. We also have no way of telling the sound to stop playing (which is extremely unfortunate). If we wanted these very useful capabilities, we'd have to create a nice inter-processor communication system.

The Need for Inter-processor Communication

The need for a standard inter-processor communication system is real. Dekutree has perhaps written the first one for the Nintendo DS. Chris Double used Dekutree's inter-processor communication model in his tutorials. Tobias Weyand, the creator of NitroTracker and the DS Sampling Keyboard, also used Dekutree's interprocessor communication model. Many programmers prefer to use the FIFO (a special queue for inter-processor communication built into the Nintendo DS hardware). I will address both of these methods and present a nice abstraction that will allow the programmer to choose either method (according to their needs and opinions) in a later edition of this manual.

COMPILING

This is the final iteration of the Orange Spaceship demo that we will cover in this edition of the manual. Compile it and enjoy the fruits of your labors. Mmm, tasty. You should hear a nice sound when you press the thrust button. The game output should now look like the screen shots in Figure 9.1.

Figure 9.1 Flying around in the Orange Shuttle, with sound!

APPENDIX A
VRAM

OVERVIEW

I compiled these tables so that it would be easy to see what banks of VRAM could be used for which purposes. I hope this will be of help to those of you designing your 2D graphic engines or are wanting a more in depth coverage of the Nintendo DS VRAM. I hope they will be as useful to you as they continue to be for me.

Usually, using this appendix is just a two step process. Based on what you want to do, use the usage tables to figure out which VRAM banks are suited to the task. Next, check the layout tables to see if there exist any conflicts between bank memory locations and assignments. If there are no conflicts, you have successfully come up with a VRAM layout for what you want to do. Otherwise, simply go back to the usage charts to see if there are any alternative ways of doing what you want to do. If what you want to do still seems impossible, consider using the 3D hardware.

VRAM Usage

NOTES AND PRECAUTIONS

In the tables, you'll see some numbers. These numbers are called "usage index numbers." The best usage index number is one: it matches perfectly for that particular task. Greater than one means a wasteful design. Less than one means that you won't be able to complete the whole task. Individual VRAM banks may have a rating of less than one for any particular task, but when used in conjunction with another bank, those two banks both might be perfectly suited for that task.

For example, if we wanted to store tiles for 128 unique 16 color sprites, the best choice is to use two banks that each have a rating of less than one (.5), F and G. These two banks together sum to one, a perfect match for the task.

There are exceptions to the one is best rule, however. Keep in mind that if you want to double a task, as in you want two 16-bit color bitmaps for instance, you'll want to sum your usage index number to two. Likewise, if you wanted 10 256 color external backgrounds (for whatever crazy reasons you may have), you'll want to sum your usage index number to ten. Also note that if assigning leftover banks to auxiliary memory (LCD or Work RAM), a larger number simply gives you more memory, so the one is best rule doesn't really apply for that table's purposes.

Each bitmap in the table has a size of 256x256 and is at the specified color depth. If you need a larger bitmap, do the appropriate math to linearly transform the usage index numbers. For instance, a 512x512 16-bit color bitmap will take up 4 times as much memory as a 256x256 sized one. Thus, we multiply our usage numbers by .25 for that particular row, the 16-bit color bitmap row. We then see that it will take VRAM banks A, B, C, and D to meet our purpose. Likewise, if we wanted to do a smaller bitmap, we would multiply by a number larger than 1 for that particular row.

Just because the usage charts say that it is possible, make sure that you check the layout charts to be sure that using your specific combination of the banks is legal.

Main Screen Usage

VRAM Bank	A	B	C	D	E	F	G	H	I
16-Bit Color Bitmap	1	1	1	1	.5	.125	.125	N/A	N/A
256 Color Paletted Bitmap	2.66	2.66	2.66	2.66	1.33	.66	.66	N/A	N/A
128 Unique 256 Color Sprites	2	2	N/A	N/A	1	.25	.25	N/A	N/A

VRAM Bank	A	B	C	D	E	F	G	H	I
128 Unique 16 Color Sprites	4	4	N/A	N/A	2	.5	.5	N/A	N/A
Background Extended Palette	N/A	N/A	N/A	N/A	1	.5	.5	N/A	N/A
Object Extended Palette	N/A	N/A	N/A	N/A	N/A	1	1	N/A	N/A

Sub Screen Usage

VRAM Bank	A	B	C	D	E	F	G	H	I
16-Bit Color Bitmap	N/A	N/A	1	N/A	N/A	N/A	N/A	.5	.125
256 Color Paletted Bitmap	N/A	N/A	2.66	N/A	N/A	N/A	N/A	1.33	.66

VRAM Bank	A	B	C	D	E	F	G	H	I
128 Unique 256 Color Sprites	N/A	N/A	N/A	2	N/A	N/A	N/A	N/A	.25
128 Unique 16 Color Sprites	N/A	N/A	N/A	4	N/A	N/A	N/A	N/A	.5
Background Extended Palette	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1	N/A
Object Extended Palette	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1

Auxiliary Memory Usage

VRAM Bank	A	B	C	D	E	F	G	H	I
LCD (ARM9)	1	1	1	1	1	1	1	1	1
Work RAM (ARM7)	N/A	N/A	1	1	N/A	N/A	N/A	N/A	N/A

Texture Usage

VRAM Bank	A	B	C	D	E	F	G	H	I
Texture Image	1	1	1	1	N/A	N/A	N/A	N/A	N/A
Texture Palette	N/A	N/A	N/A	N/A	1	.25	.25	N/A	N/A

VRAM Layout

Notes and Precautions

To use the layout charts, look for the VRAM bank you want to map and select a memory location to map it to. Depending on what you want to do, you'll want to refer to the numbers listed under the usages. These numbers represent the offset you'll need to use to access your VRAM bank.

For example, say we decided to use VRAM banks A and B to display a 16-bit color bitmap. We would look for VRAM banks A and B under the VRAM Banks column on the "Main Screen Background" table. After finding them, we notice that we can assign them to memory locations 0x0600:000, 0x0602:0000, 0x0604:0000, and 0x0606:0000. We then choose to map VRAM bank A to 0x0600:000 and VRAM bank B to 0x0602:0000. To access our 16-bit color bitmaps we use the offset of 0 for the first and 8 for the second. In the case of backgrounds, libnds provides the BG_BMP_BASE macro for use to use when setting up our backgrounds. In this case, some code we might use to set up BG2 and BG3 might look like the following.

```
BG2_CR = BG_BMP16_256x256 | BG_BMP_BASE(0) | BG_PRIORITY(3); //on the bottom
BG3_CR = BG_BMP16_256x256 | BG_BMP_BASE(8) | BG_PRIORITY(0); //on the top
```

Slots are not readable or writable by the ARM9 or the ARM7. No usage is necessarily implied on these tables, only layout in comparison to the sizes of relevant data.

Each bitmap in the table has a size of 256x256 and is at the specified color depth.

Main Screen Background

Memory Location	16-Bit Color Bit-map	256 Color Paletted Bitmap	VRAM Banks
0x0600:0000	0	0	A,B,C,D,E,F,G
0x0600:4000	0	0	F,G
0x0600:8000	0	0	F,G
0x0600:C000	0	3	F,G
0x0601:0000	0	3	F,G
0x0601:4000	0	3	F,G
0x0601:8000	0	6	F,G
0x0601:C000	0	6	F,G
0x0602:0000	8	6	A,B,C,D,F,G
0x0602:4000	8	9	F,G
0x0602:8000	8	9	F,G
0x0602:C000	8	9	F,G
0x0603:0000	8	12	F,G
0x0603:4000	8	12	F,G
0x0603:8000	8	12	F,G
0x0603:C000	8	15	F,G
0x0604:0000	16	15	A,B,C,D
0x0604:4000	16	15	
0x0604:8000	16	18	
0x0604:C000	16	18	
0x0605:0000	16	18	
0x0605:4000	16	21	
0x0605:8000	16	21	
0x0605:C000	16	21	
0x0606:0000	24	24	A,B,C,D
0x0606:4000	24	24	
0x0606:8000	24	24	
0x0606:C000	24	28	
0x0607:0000	24	28	
0x0607:4000	24	28	
0x0607:8000	24	N/A	
0x0607:C000	24	N/A	

Main Screen Object

Memory Location	128 Unique 256 Color Sprites	128 Unique 16 Color Sprites	VRAM Banks
0x0640:0000	0	0	A,B,E,F,G
0x0640:4000	0	0	F,G
0x0640:8000	0	2	F,G
0x0640:C000	0	2	F,G
0x0641:0000	4	4	F,G
0x0641:4000	4	4	F,G
0x0641:8000	4	6	F,G
0x0641:C000	4	6	F,G
0x0642:0000	8	8	A,B,F,G
0x0642:4000	8	8	F,G
0x0642:8000	8	10	F,G
0x0642:C000	8	10	F,G
0x0643:0000	12	12	F,G
0x0643:4000	12	12	F,G
0x0643:8000	12	14	F,G
0x0643:C000	12	14	F,G

Main Screen Background Extended Palette

Background extended palette slots are 32KB each, despite which VRAM bank is assigned to them.

Slot	VRAM Banks
0	E,F,G
1	E,F,G
2	E,F,G
3	E,F,G

Main Screen Object Extended Palette

Object extended palette slots are 8KB each, despite which VRAM bank is assigned to them.

Slot	VRAM Banks
0	F,G

Sub Screen Background

Memory Location	16-Bit Color Bitmap	256 Color Paletted Bitmap	VRAM Banks
0x0620:0000	0	0	C,H
0x0620:4000	0	0	
0x0620:8000	0	0	I
0x0620:C000	0	3	
0x0621:0000	0	3	
0x0621:4000	0	3	
0x0621:8000	0	N/A	
0x0621:C000	0	N/A	

Sub Screen Object

Memory Location	128 Unique 256 Color Sprites	128 Unique 16 Color Sprites	VRAM Banks
0x0660:0000	0	0	D,I
0x0660:4000	0	0	
0x0660:8000	0	2	
0x0660:C000	0	2	
0x0661:0000	4	4	
0x0661:4000	4	4	
0x0661:8000	4	6	
0x0661:C000	4	6	

Sub Screen Background Extended Palette

Background extended palette slots are 32KB each, despite which VRAM bank is assigned to them.

Slot	VRAM Banks
0	H
1	H
2	H
3	H

Sub Screen Object Extended Palette

Object extended palette slots are 8KB each, despite which VRAM bank is assigned to them.

Slot	VRAM Banks
0	I

Texture Palette

Texture palette slots are 64KB each.

Slot	VRAM Banks
0	E,F,G
1	E,F,G
2	E,F,G
3	E
4	F,G
5	F,G

Texture Image

Texture image slots are 128KB each.

Slot	VRAM Banks
0	A,B,C,D
1	A,B,C,D
2	A,B,C,D
3	A,B,C,D

ARM9 Access (LCD)

Memory Location	VRAM Banks
0x0680:0000	A
0x0682:0000	B
0x0684:0000	C
0x0686:0000	D
0x0688:0000	E
0x0689:0000	F
0x0689:4000	G
0x0689:8000	H

Memory Location	VRAM Banks
0x068A:0000	I

ARM7 Access (Work RAM)

Memory Location	VRAM Banks
0x0600:0000	C,D
0x0602:0000	C,D

Postface

I hope you've enjoyed reading this manual as much as I've enjoyed writing it. I hope it has helped you to learn the basics of Nintendo DS programming, and if not, that it has pointed you in the correct direction. I wish you luck in all your future projects and endeavors.

Feel free to contact me jaeder@patatersoft.info, if you need any help or have corrections or suggestions.

Acknowledgments

I'd like to offer special thanks to all those who have taught me this past year about Nintendo DS programming. I apologize if I've left anyone off the list who has helped me. Special thanks to

John Haugeland

Liran Nuna

Thorsten Lemke

Tobias Weyand

Dave Murphy

Shaun Taylor

Bryant Poffenberger

Hollis Jacobsen

Jeff Katz

James Zawacki

Michael Noland

Jason Rogers

Christopher Double

Matt Luckett

Mathias Onisseit