DS Programming 101: From Zero to Hero!

Table Of Contents

Written by Foxi4

Revised by Pomegrenade

Preface

Hello and welcome! If you are reading this then it's likely that you're interested in getting to know more about programming for the Nintendo DS! If you are not, then you likely took the wrong turn, but let's not get into that.

Let's also start with establishing one important thing – as the title suggests, this is a "From Zero to Hero" guide. If you are an experienced programmer then it is likely that you will not benefit from it much, if at all. It is going to introduce the very basics to users who have never even seen a compiler before and never coded in their life – stuff that you probably already know and aren't interested in anymore. You are however still welcome as this is my first tutorial and will likely require a certain degree of proof-reading, plus, you may of course have useful suggestions! Keep in mind the target audience though; I'm doing my best not to introduce complicated concepts early on. If you're not an experienced programmer or never programmed at all, this is a great place to start!

I've seen many guides approaching this subject – some were more helpful, some were rather vague, but there is one thing that was common in all of them, and it became apparent to me that something has to be done about it.

The guides I've seen so-far are dedicated to users who are familiar with programming and only require an introduction to the DS environment; none of them are actually "tutorials" from the ground up. Does this mean that a non-experienced user simply cannot program for the DS or should not begin his adventure with programming on this exact platform? No, it does not! In fact, the DS is likely the easiest platform to program for when it comes to consoles – libnds is really not that hard to wrap your mind around and there are numerous libraries out there that facilitate programming for it even further.

You probably want to ask: "If it's so easy, why do you think it requires some sort of an explanation? The libraries are well-documented, do you expect the readers to be dill-wits who can't follow simple examples?" and the answer to that is "No, in fact, I do believe that everybody is capable of programming, however one has to learn and acquire some basic programming habits and have some practice in C to be successful at it" and this is exactly the main goal of this tutorial. Depending on the interest shown by users and my workload at Uni this may or may not be a full-featured guide, however I promise that I will at least try to keep it up-to-date and expand upon it from time to time.

Now that the purpose is established, let's move on to the juicy parts! I hope you will enjoy learning together and in case of any questions or suggestions, do write! Dear readers, keep in mind that the first few tutorials will be an incredibly rapid course in C, applicable to any type of programming, not just for the DS! We won't be compiling much until this material is covered and thoroughly understood! So… Let's get it on!

Chapter 0: Preparing the Environment

Alright then! We've established our goal; it is time to make the first steps towards it! For the duration of this course we will be using the **C language** which is versatile and simple enough to understand. We'll program in the **Visual C++ Express 2008** and we'll be using **NFLib by NightFox and Co.** as our graphics library so that graphics do not impede our progress, but we're miles before getting there. First and foremost, we need to start up our download queue, and we'll need plenty-fancy things!

Firstly, we of course need the **devKit Pro toolchain**, available here:
http://sourceforge.n...ed%20Installer/
Next, we need to get our hands on **the code editor**:
http://msdn.microsof...future/bb421473
We will also need **Drunken Coders's Wizard** to add a Nintendo DS application (among others!) as a project type:
http://pern.drunkenc...udio_wizard.zip
Finally, we'll download **NightFoxLib**:
http://sourceforge.n...ts/nflib/files/

Once all the files are downloaded, proceed to installing them, starting from devKit Pro (preferably on C:\), next the Visual C++ Studio Express 2008, the Drunken Coders's Template Wizard and finally NightFox Lib (preferably in the devKit Pro directory).

Now we have our programming environment ready! Exciting, isn't it? Well, not quite. Why? Because now we have to move onto the Introductory chapters of this guide – they may be boring and taxing but without them we will not be able to really program anything as everything you will be reading from now on will sound like black magic before this knowledge sinks into you, at least partially. Thus, without further ado, we move onto…

Introductory

Chapter 1: Variables!

To even have a chance at programming something, we need to know what we'll be dealing with. **Introductory Chapter 1** will introduce the first important subject – **Variables**, your main weapons when it comes to programming.

So, what **is** a Variable? Well, in layman's terms, a variable can be just about "anything" – **it can be a number, a character, literally anything,** but to simplify it, we'll say it is a **value**, a **piece of data** that is kept in the console's memory for later use. In most tutorials you would see a link to a Wikipedia article or a reference to a book that will introduce you to the subject – not here. We want to get from Zero to Hero and we want to get there fast, don't we? As far as we're concerned, we could possess all the knowledge about the variables and we still won't be able to use them anyways! I will only pass onto you the knowledge that will actually be relevant to you, so bear with me. 

Let's start with describing how to declare a Variable:

```
Type Name = Value;
```

Your typical variable has four identifying features – **a scope, a type, a name and a value**. First and foremost we will tackle **types**, as they determine how we will actually use a given variable.

We can divide our variables into 3 main groups – **Integers, Booleans and Chars**.
Those can be **Signed** or **Unsigned** (except Booleans), plus Integers may **range from 8 to 32 bit** ones (even 64 on PC's, but let's not go ahead of ourselves). Sounds like black magic so-far? Good, cause we're going to explain what all that means, starting from types and their properties:

Signedness is the first property we will discuss. One could write a whole elaborate essay on the differences between signed and unsigned variables. What you need to know is that Signed variables range from negative values throughout 0 and positive values while unsigned ones range from 0 upwards. That's it, really. That's all you need to know!

Next, we'll tackle the lovely **Integers**. As I said, Integers on the DS may consist of **8, 16 or 32 bits**, but **what does that mean**, exactly? Well, it means what **range of values a variable will be able to carry!** For example, an **8-bit integer** will have a length of **255**, meaning it will range from **-128 to 127** when signed and from **0 to 255** when it's unsigned. A **16-bit** one will have a length of **65,535** while a **32-bit one**, a length of **4,294,967,295**.

To use them, we will have to learn how to **declare** them! The symbols used to refer to those integers are u for Unsigned, s for Signed, followed by the length in bits, for example "**u8**" or "**s16**".

Next we have **Booleans**. There's really not much to say about them other then they may hold only two possible values – "**true**" (1) or "**false**" (0). We will mostly be using them for simple switches and there's no point in dedicating any more time to them. The symbol used to declare Booleans is "**bool**".

Finally, we have **chars**… **Chars** will mostly be used to hold **characters or numbers**, but they are much more versatile then that. By "**holding anything**" I really meant that you can use those variable to hold "**anything**". From **text** to **graphics** and **sound**, arrays of chars **will be your tool of choice**. The symbol to declare chars is "**char**".

Now that types have been discussed, we can move to **Scopes**, and we'll discuss only two: **Global** and **Local**.
A **Global Variable** is declared **outside** the main block of the program, thus can be called anywhere within it while a **Local Variable** is declared **within** a function, thus may only be used within that function. I'm sounding vague, huh? Functions? Blocks? **What am I talking about?** Well, let's get right onto that in the **Introductory Chapter 2**, as I don't think the "name" part needs any explanation other than "don't use special signs or spaces when naming a Variable".

Chapter 2: Functions!

So, **what is a Function**, you ask? Well, a Function in layman's terms **is a set of instructions that the program is supposed to carry out**. How does a function look like, how do we declare it you ask? Let's see the declaration for ourselves:

Type Name(TypeOf Argument, TypeOf Argument2){ … };

That… looks incredibly complex, doesn't it? Let's cut it into pieces, shall we?

Where "**Type**" you put either "**void**" if you don't really want the function to return any value for you, just carry out the instructions. If you do want it to return some sort of a value or data, you can specify it using the **previously described Types** – that way, you will be able to assign the value returned to a given variable! Snazzy!

The **Name** can be just about anything, but without spaces and really fancy special signs.

As far as **Arguments** are concerned, **there can be as many as you like** and they can be **any type of a variable you want**.

How do we call our declared function? Well, we simply do this:

NameOfTheFunction(Value1, Value2);

Simple!
Just to practice, let's build our very first simple function, shall we?

CODE

```
s16 Adding(s16 Number1, s16 Number2){
s16 Result=Number1+Number2;
return(Result);
}
```

So, **what does this function do?** Well, it adds the value specified as Number1 to the value specified in Number2 and returns its number as shown by Result!

So, if we do this:

CODE

```
s16 Number3;
Number3=Adding(2, 2);
```

Our function will assign the value "4" to the declared variable Number3. Neato, huh? Of course we're not going to build functions with *that* simple functionality but hey! We're just starting, right?

Before we can actually create any fun functions, we'll have to learn a bit about operators and built-in C functions… **but I'm feeling like you guys are yawning at this point** – so much new-found knowledge and we can't use it anywhere! Annoying, isn't it? Well…
Let's do something else, shall we? Let's **build** our **very first application!** Oh, the **excitement!** First and foremost though, we will analyze the template we're given from libnds so that we know where to put what, hmm? Your minds open? You're fingers ready? This is the part you were all looking forward to, so **let's do this!**

Exercise #1 - Your first program!

Open up Visual Studio C++ Express, start a new project and select the DS as your platform of choice via the wizard – the default settings will do. Once you're done, you will be able to browse the contents of your Solution using the table of contents situated on the left-hand part of the screen. Right now you only have the Source file called **"template.c"**. Let's open it! You will likely see this as the result:

CODE

```
 /*--------------------------------------------------------------------------------

        Basic template code for starting a DS app

--------------------------------------------------------------------------------*/
#include <nds.h>
#include <stdio.h>
//--------------------------------------------------------------------------------
int main(void) {
//--------------------------------------------------------------------------------
        consoleDemoInit();
        iprintf("Hello World!");
        while(1) {
                swiWaitForVBlank();
        }

}
```

Wow wow wow, that's alot and sooo fast! Let's look through this quickly, shall we?

Every program is essentially divided into three main sections: the **Includes**, the **Declarations and Defines** and finally the **main();** Function. We will discuss them all briefly. I will also explain the three functions that appear in this code so that they're **crystal clear**.

#include is a directive referring to any files that may be included in the project. It's vital to know that #include files will *always* be in our RAM – their content can be accessed anywhere throughout our program! This will turn out to be very useful when organizing your work and resources later-on, so remember it! The two files included are nds.h, which is the libnds library and stdio.h, which is the Standard Input/Output library of C and C++. The Angle-Brackets <> indicate that the compiler will search for the file using it's specified INCLUDE path as well as the parent directory of your program, were we using simple quotation marks "", the compiler would attempt to find it only in the parent directory, but more on that later.

Normally after the #include we would start putting our **Global declarations**, every variable declared there will be treated as Global. Keep that in mind!
This basic template has no variables, but let's add one, just for the heck of it! Scroll up and put your declaration right before the main(); function. Let's say…

char MyName**[25]**="Put your name here, silly!";

Can you tell me what kind of a variable that is? **Yep, you are absolutely right!** Great attention span! It's indeed a char, its name is "**MyName**" and its value is… well, whatever your name is! What's with the weird number in the brackets, you ask? Well, one char can only hold one character – with this number, our variable is now an **Array**, otherwise known as **String**, of the **length of 25 characters** – that way, we can put more characters in, but more on that in the later chapters about Arrays. In any case, **good job!** Remember about the "**;**", without putting it at the end, the compiler will not know when the line of input ends and will return an error, so **declare carefuly!** Let's move on.

Lines starting with "**//**" or enclosed between "**/***" "***/**" are called **Comments** – the compiler ignores them, but they are helpful to mark important parts of your code.

Finally we move on to the **int main();** the most interesting part we'll tackle today.
main(); is effectively the main function of the program – whatever is supposed to happen is placed in it. It is divided into two main parts – the part before **while(1)** and after it. We'll talk about the while loop later, for now, all we need to know is that whatever is before while happens once, whatever comes within its brackets happens once each frame (the DS works at circa**60 frames per second**, so the maths here are clear).

Before while(1) we have two functions, let's have a closer look to know what they are:

consoleDemoInit(); is a function built-in libnds, it will launch a DOS-like command line to which you will be able to output text. Nothing more, nothing less, really.

iprintf("Hello World!"); is a function used for printing text. It prints it directly into the console, the text printed is placed between the quotation marks. Simple!

Let's head to while, shall we?

It contains only one function - **swiWaitForVBlank();** which, as the name implies, wait for a VBlank, meaning the end of a frame. You will have to put this function at the end of your main, always!
So, from what we've read so-far, this program should:

1. Initialize the console.
2. Print Hello World! On the screen.

Let's see if it does that, shall we? Select "**Build**" from the context menu at the top of the screen, then hit "**Build Solution**", alternatively just hit "**F7**". This will begin the makefile operations and compile your very first .nds file. **Excited? You should be!**

Once Building Operations are finished, you will be able to read where the file was saved at the bottom part of the screen. **No surprise there** – it's in the folder you specified when creating the Project.

Open that folder – the .nds file should be there. Provided you have an .nds emulator, you will be able to launch it, and… **magic!** It does what it's supposed to!
I can hear you nagging "but what about my Variable?!? I want it to do something that wasn't in the template, you're a lame teacher!" and that **breaks my heart.** Would I desert you? Would I? **Never!**

Let's **spice up the code** a little bit, hmm? Let's make this program specifically about *__you__*.
Scroll back up where you declared your first variable and declare another one, like this:

```
char Text[100];
```

We already know it's a string of characters with a specified length, but why No value? Well, it means that the string is **empty** – we can put data into it later-on though. This is called a "**buffer**" – a place dedicated in RAM for us to input data into.

Let's move onto the function though, shall we? Before the **iprintf()**, we'll add another fun function that is commonly used with strings:

```
sprintf(Text, "Hello World! My name is %s!", MyName);
```

Okay, **confusing**, I know. Explaining now. **sprintf();** is a function used for formatting strings – you select a **destination buffer**, then you write the **format** within the brackets, finally you write**which variables should be used to format it**. In this case, we are inserting the string MyName using "%s". The final result should be "Hello World! My name is Whatever you specified as your name". That's **neat**, isn't it?

Now, let's modify the iprintf to use our text. All you need to do is this:

```
iprintf(Text);
```

Now, rather then Hello World!, the text printed will be whatever has been put in the **buffer Text** (hence the lack of brackets). Now, what you should have is this:

CODE

```
#include <nds.h>
#include <stdio.h>
char Text[100];
char MyName[25]="Foxi4";
int main(void) {

        consoleDemoInit();
        sprintf(Text, "Hello World! My name is %s!", MyName);
        iprintf(Text);
        while(1) {
                swiWaitForVBlank();
        }
}
```

Ready? Steady? **BUILD!** So… does it work? Well, why shouldn't it?

You've just built your very first program for the DS, **congratulations!** Admittedly it's simple, but with some dedication, soon enough **you'll be a pro at this!** One thing is for sure, **you're no longer a Zero**, you moved onto... **0,1**… So **there's still a road ahead of you!**

Chapter 3: Operators in C

So far, we've learned the basic structure of an application in C, the types of Variables and looked at some functions - I hope everything so-far sunk in because we're about to look at **if statements** and **switches** with various **conditions!** Before we get there though, we need to know how to build those **conditions**, and for that, we'll need to have a look at **Operators**.

Without further ado, let's get to it, starting from explaining what exactly they are.

Operators are very simple elements which tell the program **how to combine, compare or modify Values**. By putting them in-between of two **Operands** we create **Expressions**. We can also use some of them **connect Expressions together**, designing more advanced **Logic**.

Let's use an example that's known from our daily life and math - an **expression** is for example "3+4" where "+" is the **Operator**. **Operators** are divided into **5 groups**, each of them with different uses.

Those groups are **Arithmetic, Comparison, Logical, Assignment and the Ternary Operator**. First, we'll have a look at **Arithmetic Operators** as we've already used a few. They are used exactly the same as you'd use them in everyday math tasks, so their function should not pose any problems to anyone. Those Operators are:

ARITMETHIC OPERATORS

+ addition
- substraction (- when placed before numbers negates them)
***** multiplication
/ division
(and **)** define precedence (to define in what order the calculations happen)

++ incrementation (adding 1)
-- decrementation (substracting 1)
>> bit shift right
<< bit shift left
% modulus (calculating the remainder of dividing two numbers)

Example of use: *AVariable=3**+**3;*
(Assign the result from adding 3 to 3 to AVariable)

As of now, only the first 5 (well, maybe 7 ) are of our interest, the rest will be covered in more detail in subsequent chapters if they pop up, for now you just need to know they exist.

Next up are operators which we also used, called **Assignment Operators**. We use them to **assign Values** to a given **Variable**. Those operators are:

ASSIGNMENT OPERATORS

= Simple assignment. The Value of the right operand is assigned to the Variable on the left.
***=** Multiplication assignment. The Value of the left operand is multiplied by the Value of the right operand and the result is assigned to the left one.
/= Division assignment. The Value of the left operand is divided by the Value of the right operand and the result is assigned to the left one.
%= Remainder assignment. The remainder of dividing the Value of the left operand by the right operand is assigned to the left operand.
+= Addition assignment. The Values of operands are added and the result is assigned to the left one.
−= Subtraction assignment. The Value of the right operand is substracted from the Value of the left operand and the result is assigned to the left one.

<<= Left-shift assignment. The Value of the left operand is shifted left by the ammount of bits specified by the right operand and the result is assigned to the left one.
>>= Right-shift assignment. The Value of the left operand is shifted right by the ammount of bits specified by the right operand and the result is assigned to the left one.
&= Bitwise-AND assignment. Obtains the bitwise AND of the left and right operands and assigns the result in the left operand.

^= Bitwise-exclusive-OR assignment. Obtains the bitwise exclusive OR of the left and right operands and assigns the result in the left operand.
|= Bitwise-inclusive-OR assignment Obtains the bitwise inclusive OR of the left and right operands and assigns the result in the left operand.

Example of use: *AVariable=AnotherVariableOrAValue;*
(Assign the value of AnotherVariableOrValue to AVariable)

Once again, I've divided them into two groups - the ones you should be concerned with firstly and the ones that we'll be using much, much later.

Now we can move on to what interests us the most today, which would be **Comparison Operators.** These will be the basis of building conditions and you need to know them by heart. Their purpose is to compare two **Values** and return the result of this comparison. Those operators are:

COMPARISON OPERATORS

== Checks if the Value on the left is equal to the Value on the right.
!= Checks if the Value is not equal to the Value on the right.
< Checks if the Value on the left is smaller then the Value on the right.
> Checks if the Value on the left if greater then the Value on the right.
<= Checks if the Value on the left is smaller or equal to the Value on the right.
>= Check if the Value on the left is greater or equal to the Value on the right.

Example of use: *if(Variable<3) Variable++;*
(if the Variable's value is less then three, increment it)

ALL **Comparison Operators** are essential; there is literally nothing we could leave for later.
Finally we can move on to another useful kind of **Operators** which we will frequently use to join together different expressions. Those operators are called **Logical Operators** and are relatively simple to remember since there are only three of them.

LOGICAL OPERATORS

! Is used to negate a given expression, this operator is called NOT.
&& If both parts of the expression are true, the whole expression is true. If either part if false, the expression will be false. This operator is called AND.
|| If either part of the expression is true, the whole expression is true. This operator is called OR.

Example of use: *if(AVariable==3* **&&** *AnotherVariable==3) AVariable++;*
(if both variables are equal 3, increment the first one)

That'd be all we need to know now - the **Ternary Operator** will be introduced together with if statements. That's all for this chapter and tune in next time to learn how to build Conditions with our new-found knowledge, alongside some fun examples!

Chapter 4: Conditions

Now that we know **Operators**, we are pretty much set to start learning about creating various **Conditions in C**. What is a **Condition**? Well, it is a **Statement** that specifies **how our program is going to react when a given event happens**. We can divide those statements into two basic groups, **if Statements** and **switches**. Firstly we'll have a look at both, then we're going to compare them as to know exactly when to use them efficiently.

We've actually seen a few **if statements** already in the previous Chapter, I'm sure you've noticed! These were however very short - they were the shortest possible form of that kind:

```
if (Condition) Result;
```

With that kind of a statement we can only specify **one Result** and **one Condition**, so naturally this is a **really limited** statement. To fully utilize **if Statements** we need to be familiar with their entire structure, which is as follows:

CODE

```
if (Condition){
   Statements;
  }
else if(Condition2){
   Statements;
  }
else{
   Statements;
  }
```

We can see that this is a **much more flexible** implementation of an **if Statement**, we can not only specify as many **Conditions** and **Results** as we want using **else if**'s, we can also set a **Result**that will occour **if neither of the Conditions are met!** It is divided into three parts, our first **if**, a number of following **else if's** and finishes with **else** - the latter two parts are **entirely optional**and you can ommit either of them if you don't need them.

To show how this works in a more practical way, let's have a look at this example:

CODE

```
if (Age>=18){
   iprintf("You're an adult!");
  }
else if(Age<18 && Age>=0){
   iprintf("You're a minor!");
  }
else{
   iprintf("Very funny. You were supposed to input your age!");
  }
```

Here, if the **Variable** Age's **Value** is **greater or equal** to 18, the program will inform the user that he is an adult by printing a message on the screen. If the Value is **less then** 18, but more then 0, the program will state that the user is a minor. If the Variable has a value that is **incorrect**, such as less then 0, the program snarkly remarks that the user inputted an incorrect value.

Of course we could just do this:

CODE

```
if (Age>=18){
   iprintf("You're an adult!");
  }
else{
   iprintf("You're a minor!");
  }
```

But this presents a **degree of ambiguity** - if the user would input a value that is less then 0, the program would **still** refer to him as a minor **despite** the fact that the value is **incorrect**. This could be prevented by doing this:

CODE

```
if (Age>=18){
   iprintf("You're an adult!");
  }
else if(Age<18 && Age>0){
   iprintf("You're a minor!");
  }
```

However again, the program would be confused if the inputted value was of **incorrect type**, for example a character rather then a digit and this would result in a **glitch** - the character would be transcribed into a numerical value.

Why am I saying all this? You're going to say *"we're not brain-dead, this is obvious!"*. Well, let me tell you, **it's not.** I'm doing this to teach you to **avoid being ambiguous at all costs,** this will save you time when **debugging** your applications.

Keeping these issues in mind will help you **avoid glitches** in your program and prevent it from acting in a manner that you would not expect. **Computers are not intelligent,** they will not do your work for you.
You have to imagine **you're giving orders to a complete simpleton** who requires really specific instructions as to **not make a mistake.**

Simplify your code **when it is possible, by all means,** but **where there is room for error,** specify your intentions **in detail.**

To conclude the section about **if Statements**, we'll have a look at the **Ternary Operator** as promised last time. Sometimes our conditions **can be thoroughly simplified** and we can ommit the use of **if Statements** or **switches** altogether by using the **Ternary Operator "?".**

Let's consider this snippet of code:

CODE

```
if (Variable==10){
        Result = 1;
}
else{
        Result = 0;
}
```

We created an **if Statement** which checks whether the **Variable** is equal to 10. If it is, the program will assign the value 1 (true) to **Result**, otherwise it will assign 0 (false). We can simplify this with a simple **Ternary Operation**, like this:

CODE

```
Result = Variable==10 ? 1 : 0;
```

This will automatically assign the values to **Result** depending on whether the **Condition** will turn out to be **true** or **false!** Much quicker, **isn't it?**

Now that we are accustomed with **if Statements,** we can learn about **switches**, which are very similar in function however may come in handy simply because they are more **see-through** and**easier to debug.** Let's have a look, shall we?

CODE

```
switch(Variable){
  case 1:
        Statements;
        break;
  case 2:
        Statements;
        break;
  default:
        Statements;
        break;
}
```

In this example **switch**, we have two **cases** and the **default case**.
If the **Value** of **Variable** is **equal to 1**, it performs the **Statements** specified in **case 1**, if it is **equal to 2**, it performs the **Statements** specified in **case 2,** if it is **neither,** it performs the **Statements** specified in the **default case,** which is **optional.** Each case is separated by **break;** as to **conclude the case,** however it is **optional** if you want to receive the same result in **several cases,** like here:

CODE

```
switch(Variable){
  case 0:
        iprintf("Variable is equal to 0");
                break;
  case 1:
  case 2:
        iprintf("Variable is equal to either 1 or 2");
        break;
  default:
        iprintf("Variable is neither 0 nor 1 or 2");
        break;
}
```

By not adding the **break;** after **case 0,** we specified **the same Statement** for both **case 1** and **case 2,** thus saving time and **simplifying** the **switch**.

The **default;** case again is optional, **however** I'm reminding you about possible **ambiguity of code.** This is a **disadvantage** of using **switches** - if the **Variable will not fall in either case specified** and there will be **no default case present,** the program will simply **ignore** the input altogether! Another disadvantage of **switches** is that the **Variable** used must be a **simple Variable** - either an **Integer** or a **char**, thus **we cannot use it with Strings or Structures**.

This concludes **Introductory Chapter 4**, in the next and final introductory chapter we will have a look at **Loops**, specifically **for Loops** and **while Loops.** The rest of C-related material will be discussed as we progress **if needs-be.** Thank you for reading and **see you next time!**

Chapter 5: Loops

Today we'll be having a look at **Loops**, to be more specific, the **for()** and **while()**. In programming, we use **Loops** to create **cyclical action** - we put **functions** we want to use **repeatedly** in them to save time. As always, we'll have a look at both and compare their features to know when to use each of them. Let's start with the **for()** loop and its syntax, shall we?

for(Starting Expression; Testing Expression; Count Expression){...}

Starting? Testing? Counting? What's all that? Well, let's have a look at a more practical example:

CODE

```
u8 i=0;
for(i=0;i<5;i++){
iprintf("This is a line of text.\n");
}
```

As we can see, we created a **Variable** to be used in the **Loop** called **i**, this **Variable** is **incremented** once each cycle as long as the **Testing expression** is **true**, which here means as long as it is less than 5. Until it reaches this state though, every **Statement** between the **{ }** brackets **will be executed once each cycle** aswell! Thus, the end effect is:

CODE

```
This is a line of text.
This is a line of text.
This is a line of text.
This is a line of text.
This is a line of text.
```

You're going to ask "where's the weird "\n" you've ended the String with, huh? Well, this is the **Newline Sign**! Using it you can make sure that whatever new string you'll be inputting into the console will be printed **in the next line**! Let's compare this **for()** to a loop we already know, the **while() Loop**.

while(Condition){...}

Looks a whole lot more simple, huh? Well, it also works **slightly differently**. As you can see, there is no place to put in the **Start Condition**, nor do we have a place where we could **add or substract** from our controlling **Variable**... How do we use it then?

The while **Loop** will execute the actions nested between its **{ }** brackets as long as the **Condition specified is true**. For example...

CODE

```
u8 i=0;
while(i!=5){
iprintf("This is a line of text.\n");
i++;
}
```

The idea here is clear - we have our **Variable i** which we use as a controller. The **Loop** prints text into the console with each cycle and then increments the **Variable i**. After 5 cycles, **i** reaches 5 and the **Looped functions** are no longer executed... Yes, **you guessed it!** The result is:

CODE

```
This is a line of text.
This is a line of text.
This is a line of text.
This is a line of text.
This is a line of text.
```

As you've seen, we can in fact utilize both **Loops** to do the exact same thing, but you will also notice that the **for() Loop** is very much independent - it sets the value for **i** itself and it also sets how this value will change each cycle.

With **while()**, you have to plot the activation and deactivation of the **Loop** yourself.

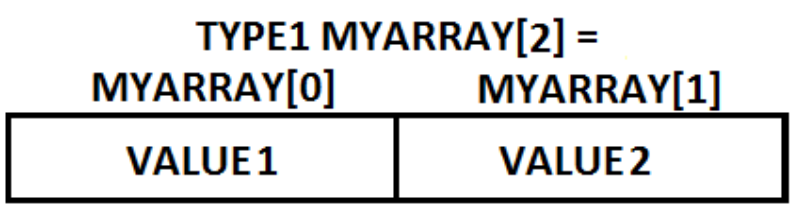
Both cases have their advantages and disadvantages so all I'm really going to say is that we will use **for() Loops** whenever we want a certain function to be repeated a specified number of times and leave the **while() Loop** for functions that only need to be executed repeatedly when a given event occurs, for example when constructing A.I's, but more on that in the Chapter about game logic later on.

I hope you enjoyed this chapter, even if it's a little bit short. There really isn't much to say about **Loop**, so bare with me. I'd also like to inform that there's been a slight change in the schedule - we'll jam a chapter about **Arrays** and **Structures** before we proceed to strictly DS-specific programming as I don't want to run into unintroduced concepts while we're at it. I'll do my best to introduce both briefly and clearly. Stay tuned and thanks again for reading, see you next time!

Chapter 6. Containers of Variables - Structures and Arrays

Structures and **Arrays** are nifty **containers** which will greatly improve our overall work. Rather then working on one **Variable** at a time, we can construct **Arrays** and **Structures** to work on **multiple Variables** at the same time. We've already seen a few **Arrays**, but we never really discussed how they work so we'll start with them.

In essence, an **Array** is a **uniform set of numerous Variables of the same Type** placed next to each other **that can be referred to using the same name**



We already know how to **declare** an **Array** from the previous chapters:

```
Type ArrayName[NumberOfUnits];
```

But this is only one of possibilities. An Array declared in this fashion will have **no values assigned**; just **empty slots** ready to receive them. You can just as well do this:

```
Type ArrayName[]= 1,2,3 ;
```

Here, we are not assigning a **"total size"** at all - the compiler deduces on its own that initially the Array has 4 slots - for the 1, 2, 3 and the NULL - there are no empty slots.

What's interesting is that **you don t have to use all the values in an Array immediately** You may assign values to it later until you reach its maximum **Size**. For example, you may declare your **Array** as one with 255 slots while in reality you will be using less then that and add values further as the program progresses:

```
Type ArrayName[255] = {1,2,3};
```

Here, we **assign** values - 1, 2 and 3 to corresponding slots 0, 1 and 2, however, the number of slots is higher than that, thus **the rest is left empty** and a **NULL** is appended to them. Keep in mind though that going past the specified **Size** of the Array will result in assigning **out-of-bounds values!** You **don t want that to happen!**

All that said... why are **Arrays** useful? After all, we could just as well declare the exact number of **Variables** we need, surely declaring them in an **Array** has some advantages? Well, **sometimes you may hold numerous values that refer to pretty much the same thing** - having the capability to call them by the same **Name** is pretty useful. Earlier we saw **Arrays** of char's - the **Array** was a **collection of signs** that were later called as a whole String. Another use for Arrays is that their Values can easily be changed using Looping, like this:

CODE

```
for(i=0; i=5; i++){
MyArray[i]++;
}
```

With only a few lines of code **we ve incremented 6 Variables** - **MyArray 0** up to **MyArray 5** . Normally it would take us **6 lines of code**, here we're just looping through an **Array** and calling its elements using the **Variable i**.

This shows us how to **call specific units** within the **Array** - all you have to do to call up a given **element** is putting **the appropriate number** corresponding to the unit into the **brackets** . As you can see, an **Array** can be **called as a whole** as previously shown, but at the same time **you are not losing the capability to call its specific elements** like you would normally call Variables, which is a big plus.

Arrays have another interesting feature. What we declared was a **single-dimensional Array** - a single **row of Variables**. **We are not limited** to this though - you can declare multi-dimensional Arrays easily:

MyArray[**Rows**][**Columns**];

This is for example a **two-dimensional Array**. Try to imagine how the **Variables** are situated in it. Why of course, on a **two-dimensional grid**. This kind of an **Array** could be used for example to track two values referring to the same object - for example the height and width of a sprite or its position on the screen.

I won't dwell into the details here - **the possibilities in design are endless**. What I will mention is the **limitation of Arrays**.

As I previously said, **every element of an Array is of the exact same type**. This can be quite a hindrance as even though **Arrays** are really versatile for holding data, all the data within them has to follow its strict **Type rules**. What can we do when we require a specific variable that will hold data of two or more types? Why, **we create a Structure!**

There are numerous ways to create a **Structure**, but I will only cover my favourite and most see-through method - **creating a new type**. It is only logical that once no available Type can suit your needs, you create a new one, **isn 't it?** How do we **declare a new type?** Quite easily, really:

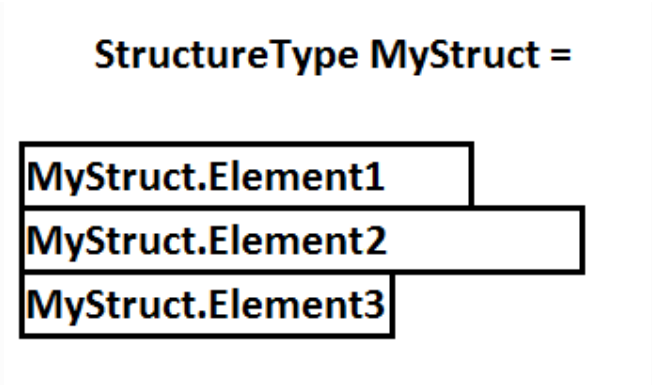
CODE

```
typedef struct{
type1 Element1;
type2 Element2;
type3 Element3;
} NameOfType;
```

As you can see, **the syntax is quite clear** - we create a **definition of a type**, and it is going to be **a structure**. The structure type in this template has three elements and its **name** is **NameOfType**... but this alone is just a type, how do we declare this structure? Easier than you think:

NameOfType **MyStructure**;

Yep - you use your newly-defined **Type** as you would with any other. **How do you call the elements** Also rather easily - you call them using the **name** of the new **Structure** and the **name** of its **element**, here it would be for example:



MyStruct Element1, **MyStruct Element2** or **MyStruct Element3**

By using a **Structure**, you free yourself from the **limitation of the Array** - you can now use any **type** for any **element** freely. Unfortunately, you also **lose** some of the privileges.

You **cannot** for example freely **cycle through elements** with a Loop as shown earlier as they are not numbered - they have **separate names**.

We'll end our adventure with C here for now... why?
Because **what s coming up is libnds Input!** I honestly believe that the information we've discussed allows us to jump into DS Programming head first without worrying of getting lost in it. The C introduction will still be expanded as we tackle new subjects, but our knowledge is sufficient for now. Can't wait? Me neither! Stay tuned and remember - do comment! **Foxi over and out!**

Practical use of libnds

Chapter 1 Input – Keys and Stylus

With the basic introduction of C behind us, I believe we're **ready** for what you've all been waiting for – the basics of **libnds**.

Until now, even with everything we've learned so-far, we were **unable** to utilize **any** user input whatsoever, as we didn't know **how to use the Keys** or the **Touchscreen** – these are platform-specific after all. Today we're going to learn **how to detect the state of Keys and the Touchscreen and how to use it**

Let's start with **Keys**. First and foremost, to use **Key Input**, we need to **scan** for their use. To do that, we place the function:

scanKeys();

Within the **main while(); loop** of the program. This will **initialize the Key scanner** and thus allow us to use them as a method of input.

Their use is divided into two portions – **Key Detection** and **State Detection**. These are of equal importance – **one cannot work without the other**, so keep that in mind!

Keys are neatly put into an easy-to-remember structure of **Enumerators**. We're not going to tackle what exactly Enumerators are; all you need to know that **each Key corresponds to a value**, the **Lid** and **Touchscreen taps** are **also** treated as a **Key**. This is the list of all Keys you can use:

CODE

```
KEY_A = BIT(0)
KEY_B = BIT(1)
KEY_SELECT = BIT(2)
KEY_START = BIT(3)
KEY_RIGHT = BIT(4)
KEY_LEFT = BIT(5)
KEY_UP = BIT(6)
KEY_DOWN = BIT(7)
KEY_R = BIT(8)
KEY_L = BIT(9)
KEY_X = BIT(10)
KEY_Y = BIT(11)
KEY_TOUCH = BIT(12) (Refers to touchscreen tapping)
KEY_LID = BIT(13)
```

As you can see, each **Key** has its respective **Name**. Using those we can create **Statements** that will depend on their values, but not just them alone. We will also require their **State**, and for that we have a set of useful functions:

CODE

```
keysDown();
keysHeld();
keysUp();
keysCurrent();
```

Down, **Held** and **Up** are relatively self-explanatory – Down is the State occurring **directly after a button is pressed**, Held is a State that occurs **when the key is pressed and kept down** and Up is a State that occurs **when the key is released**. All three return different values when different keys enter the specified state. **Current** is an interesting State, **as rather then Returning a value to a specific State immediately, it first detects which state it is**.

Now that we know that, we're able to compose our first condition based on **Key input!**

if(**KEY_A & keysHeld()**){
Statements;
}

This if statement is quite clear – **if** the **A button** is **held down**, the Statements within the **{ }** will be executed. The **Statements** will be executed until the **State** of the **Key** changes, in this case, when it is **released**. You can do the exact same thing with any combination of button + state. To make it easier and more sensible, it's worth to declare some special Variables before you begin your program

CODE

```
u16 Pressed;
u16 Held;
u16 Released;
```

...and update them at the beginning of every while(1); cycle.

CODE

```
Pressed = keysDown();
Held = keysHeld();
Released = keysUp();
```

This way we save valuable calculation power by checking the **States** of buttons once each frame at the beginning of the program rather then with each button press.

```
if(KEY_A & Held){
Statements;
}
```

Now that we've covered **Keys** we can safely progress to the **Touchscreen** which is also quite simple to use. We'll start by declaring a **Variable** to hold all the data recovered from the screen:

```
touchPosition TouchStructure;
```

We are yet unfamiliar with the type touchPosition - it's a type declared within libnds and platform-specific. It's a Structure type and holds the data on numerous stylus readings:

CODE

```
u16 px (Pixel X value)
u16 py (Pixel Y value)
u16 rawx (Raw X value)
u16 rawy (Raw Y value)
u16 z1 (Raw cross panel resistance)
u16 z2 (Raw cross panel resistance)
```

rawx and **rawy** are exactly what their names imply - raw values read from the screen. What makes them raw? Well, they're not exactly user-friendly as they are not translated to **pixel positions** on the screen, they're large and require further calculations to be used in pointing and clicking, however they are read much quicker then other values, thus are useful at for example implementing **swiping and dragging** with the stylus. **px** and **py** values are calculated from raw values and refer to specific pixels on the screen. The screen is **192 pixels high** and**256 pixels wide**, with the top left-hand corner marked as **pixel 0,0** - knowing this alone will help you use the **px** and **py** values successfully. As for **z1** and **z2** values, they refer to the panel resistance and are useful (from what I know) at reading the pressure, however it's not something you'd normally use and thus we won't talk much about them.

You can use any name you feel comfortable with in this declaration rather then "**TouchStructure**", it doesn't in any way influence the readouts... which we're not getting yet anyways.

To receive readouts of the stylus position, we need to initialize a screen scan function at the beginning of our **main while() loop** as we did with the keys. In case of the touchscreen, we use:

```
touchRead(&TouchStructure);
```

Sigh, I know what you're going to ask, oh-ever-so-inquisitive reader. **"What's that & doing there? That shouldn't be there, right?"**. Well, yeah, it should. This "&" refers to a **Pointer**, you don't know what those are yet and for now you don't need to. What I will tell you though is that to save space and calculation power, rather then using the whole structure inside this function, we just **Point** at the location of the structure in memory with a **Pointer**.

After using this function, all the readouts from the touchscreen will be immediately sent to the **TouchVariable** structure, out of which we'll be able to draw elements as we normally would with a **Structure**, as we already know its elements. For example, the position of the Stylus Horizontal-wise and pixel-wise will be kept under **TouchVariable px**.

This concludes this Chapter, I hope it was a pleasant read. I'll do my best to include a nice exercise concerning **Key** and **Touchscreen** use soon, for now, experiment on your own and if you have any questions or if you'd like to boast a bit with your newly-written code, go ahead and post! It'll **only get better** from now on!

Foxi over and out!

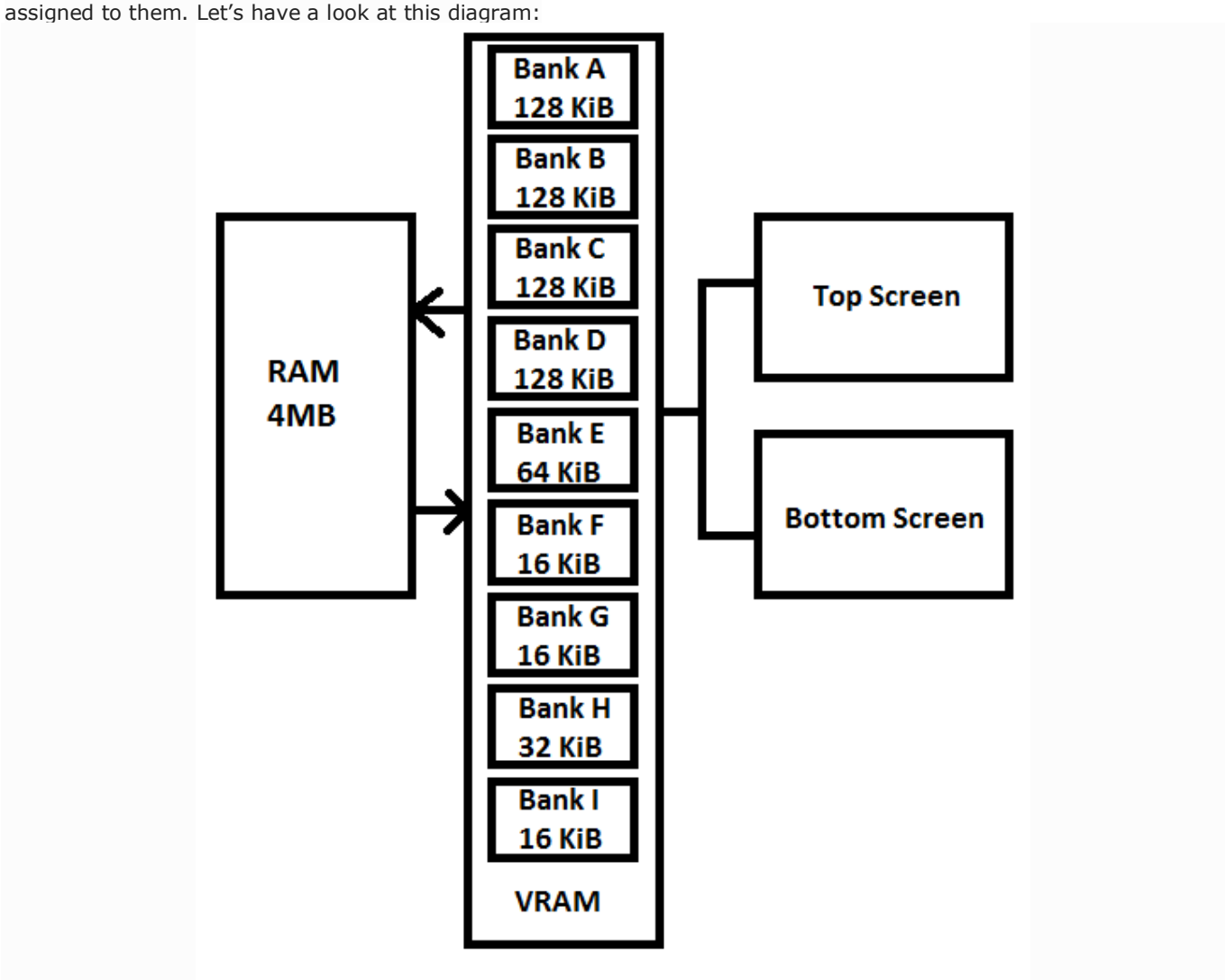
To re-cap, so-far we've learned everything we need to create conditions in our programs and to use both the buttons and the touchscreen as means of input. That's all great, but other than console text, we still don't know how to include graphics in our projects!

The next few chapters will be dedicated to both the basic theory behind graphics on the DS and practical use of NightFoxLib, which will be our library of choice to facilitate the use of backgrounds and sprites. Before we get there though, we need to learn a little bit about the DS itself once more – to be exact, we have to review how exactly the memory of the DS works and how to use it properly. Without further ado, let us dig in to our next chapter, dedicated to RAM, VRAM and Screen Layers.

Introduction to DS Hardware:
Chapter 1 – RAM and VRAM

Before we properly dive into the use of **graphics** and various other **resources**, first we need to learn a bit about how the DS **manages memory**. As you may or may not know, **the DS has 4MB of Main RAM memory**, shared between its two processors – **the ARM9 and the ARM7**. **Main RAM** is where your entire **Binary** will be stored during its execution and naturally it has to be used efficiently, as it isn't a whole lot of space. For a long time, it was a huge bottleneck for homebrew programmers, but then came libraries such as **FATlib**, **NitroFS** and other file systems that facilitated **streaming resources into RAM**, nullifying this problem by the use of buffering. We won't tackle that just yet, seeing that conveniently, **NightFoxLib** does it for us automatically, however we will talk about their use later-on when introducing **Saving and Loading**. Let's move on to the next important part of the DS's memory – the **VRAM**.

The fact that your resources are stored in **RAM** isn't always enough to utilize them. **Graphics** have to **fit within Video RAM** before they are displayed. The size of VRAM is only **656KiB** altogether, which is quite a bottleneck, considering the way it is structured. The memory is divided into several **banks**, all of which have different functions assigned to them. Let's have a look at this diagram:



This should help you imagine how the system works. Again, conveniently, **NightFoxLib** will deal with the **VRAM banks** for us, but I figured that it's best if you have a look at it to understand why sometimes you run out of **VRAM blocks** when using **graphics**.

To combat this inheritent problem of lack of space, **Tiled graphics** have been introduced back in the olden days. They take much less space than standard backgrounds, and these will be the first for us to tackle – specifically, tiled backgrounds converted with the GRIT (GBA Raster Image Transmogrifier) tool.

I specifically picked **NightFox Lib** as the library of choice for this guide, simply because its use is incredibly simple and because the library is still in active developed, not to mention that it's well-documented. I believe that we will all get the quickest and best results if we focus on it as our graphics library, so let's get to it.

Practical Use of NightFox Lib

Chapter 1 – NightFox Lib Integration Part 1

Integrating the library with your **Projects** is incredibly easy – all you really have to do is copy/paste the **Template** provided in the lib's folder into Your project… and that's it! There are further integration options that you may choose that will be discussed in the second part of this chapter. As you've already noticed, a few new folder have appeared in your **Project Folder** – those will be used to store various **Resources**. **Visual Studio** is capable of **linking** those folders to particular elements of your **Solutions**. By doing that, you will always see the contents of your folders which will make coding this one bit easier. Including those folders in your solution is pretty straight-forward, but we'll leave that for later as I already know you all can't possibly wait any longer to get to the juicy parts.

For now, attempt to compile the Template. If everything goes well and without hiccups, you've just successfully included NightFox Lib in your Project – **well done!**

Additional Utilities - Chapter 1 - GRIT

Now that we've included NightFox Lib in our project and we're all acustomed with how **RAM** and **VRAM** works, we can finally move on to what we're really interested in – using **Graphics** in our games and applications. As I already mentioned, the DS mostly uses **Tiled Graphics** – they're much easier to render and their size is lower, thus they don't clutter the RAM and VRAM as much. To convert an image to that kind of a format, we will be using GRIT. The name stands for GBA RASTER IMAGE TRANSMOGRIFIER and it has been the tool of choice for both GBA and NDS development due to the variety of functions it offers. It's been created and maintained by Jasper Vijn and it's available at http://www.coranac.com/projects/grit/ however you already have it as it's the default tool used by NightFox Lib – you can find it in the Tools directory at nflib/tools.

There, you will find several Batch files that we will be using to convert our Bitmaps with. Those include:

Convert_Affine - this Batch file will convert all the graphics in the BMP folder into Tiled Affine Backgrounds and copy them into the affine folder.
Convert_Backgrounds - this Batch file will convert all the graphics in the BMP folder into Tiled Backgrounds and copy them into the backgrounds folder.
Convert_bitmap8 - this Batch file will convert all the graphics in the BMP folder into 8-bit Bitmaps and copy them into the bitmap8 folder.
Convert_bitmap8_shared - this Batch file will convert all the graphics in the BMP folder into 8-bit Bitmaps with a shared palette and copy them into the bitmap8 folder.
Convert_bitmap16 - this Batch file will convert all the graphics in the BMP folder into 16-bit Bitmaps and copy them into the bitmap16 folder.
Convert_CMaps - this Batch file will convert all the graphics in the BMP folder into Collision Maps and copy them into the cmaps folder.
Convert_Fonts - this Batch file will convert fonts and copy them to the fonts folder.
Convert_Sprites - this Batch file will convert all the graphics in the BMP folder into DS-Compatible Sprites and copy them into the sprites folder.
Convert_Sprites_autopal - this Batch file will convert all the graphics in the BMP folder into DS-Compatible Sprites with a shared palette and copy them into the sprites folder.

As you can see, with NightFox Lib's pre-prepared Batch converter files, converting resources will be a breeze! Remember, by default, all Tiled Graphics have a transparent colour selected, and this colour is Magenta (RGB 255,0,255) – use it whenever you want parts of your graphics to be transparent as it will not appear on the hardware.
Another thing worth noting is that our **Backgrounds must have sizes divisible by 256 pixels** and our **2D Sprites must have sizes divisible by 8 pixels, 64 pixels being the maximum width/heigh**, so keep that in mind before converting to avoid errors.

Chapter 2 – NightFox Lib 2D MODE-0 Part 1 - Tiled Backgrounds

I didn't quite know how to start off the subject of **Graphics** in **NightFox Lib** to make it approachable to anyone, so I figured that the best way to do so would be to do it methodically. In the following few chapters I'm going to introduce the idea behind **MODE's** that the **DS's 2D Engine** uses, what they do and how to take the best advantage of them. We're going to start with **MODE 0** and climb upwards until we've discussed them all. Just so that nobody's bored during these chapters, I'm also going to introduce the types of graphics most commonly used in the **MODE** in question, so that we get a little bit of practical and theoretical knowledge. This chapter will be dedicated to **MODE 0** of the **2D engine in NightFox**. This mode offers **4 layers of Tiled Graphics** on the screen in question, and it's likely going to be the **MODE** you use most commonly. To initialize a **MODE**, we use the command:

```
NF_Set2D(Screen, MODE);
```

Simply substitute Screen with use 0 or 1 to select the **Top** or **Bottom** one, and substitute **MODE** with **0, 2 or 5** – the default **MODE's** of **NightFox Lib**.

Now, before we do anything else, we have to define the default Folder for our Resources, using this function:

```
NF_SetRootFolder("ROOT");
```

Now, to use the Internal filesystem, we substitute the **ROOT** with **NITROFS**, in all other cases, we simply input the folder name and the function will enable FAT instead.

With that out of the way, we can start having fun with Tiled Graphics in our projects. We'll start with Backgrounds. Before we can use any, we have to initialize the buffers for backgrounds, as NightFox streams them from NitroFS or FAT by default. To do so, we use these functions:

```
NF_InitTiledBgBuffers();
NF_InitTiledBgSys(Screen);
```

And again, we substitute the word **Screen** with the screen we are interested in – **0 or 1**.
With that out of the way, we're ready to select a background to load into **VRAM**. Our backgrounds have to be placed in the **"nitrofiles"** folder of our project, once they're there, we can use this function:

```
NF_LoadTiledBg("Path", "Name", Width, Height);
```

It will open the background in question, load it into **RAM**, attach a name to it and store the parameters of its **Width** and **Height (divisible by 256!)** so that we don't have to remember them later on as we code. It's pretty convenient, but our **Tiled Background** is not ready to be displayed yet. As we learned from the previous chapter about RAM and VRAM, we still have to copy it over to VRAM to display it. To do so, we simply use this command:

```
NF_CreateTiledBg(Screen, Layer, "Name");
```

As you can see, NightFox will deal with RAM-VRAM communications for us and all we really have to do is to select the **Screen** and **Layer** on which we'd like to display out background. We won't tackle the idea of Layers for now beyond the fact that there are **4 Layers** on each **Screen**, numbered **from 0 to 3**, 0 being the **top-most** and 3 being the **bottom-most Layer**.

Okay, so we have our **Background** ready to be displayed – it's going to appear on our screen right after the next **swiWaitForVBlank();** - perfect!

Just to make sure, let's review our sample code, shall we?

CODE

```
/*
##############################################
##DS Programming Guide - From Zero To Hero!##
####Example #2 - MODE 0 Tiled Backgrounds####
##############################################
*/

/*
############
##Includes##
############
*/

// Include C
#include <stdio.h>

// Include Libnds
#include <nds.h>

// Include NFLib
#include <nf_lib.h>

/*
###############
##Main(){...}##
###############
*/

int main(int argc, char **argv) {

// Turn on MODE 0 on the Top Screen
NF_Set2D(0, 0);

// Set the Root Folder
NF SetRootFolder("NITROFS");

// Initialize the Tiled Backgrounds System on the Top Screen
NF InitTiledBgBuffers();
NF_InitTiledBgSys(0);

// Load the Tiled Background
NF LoadTiledBg("Background", "Background", 256, 256);
NF CreateTiledBg(0, 3, "Background");

while(1){
swiWaitForVBlank();
}
return 0;
}
```

Let's compile this... and... **SUCCESS!** We've just learned how to create, load and display **Tiled Backgrounds!** Just to make sure that everything is understandable, I'll attach the Example compile folder so that everyone can review the source: http://www.mediafire.com/?lg6tvxd78gw3kly

Introduction to DS Hardware:
Chapter 2 – OAM and 2D Sprites

Before we jump head-first into using Sprites, it's worth to know a little bit about how the Sprite system works on the DS. The name **OAM** stands for **Object Attribute Memory** – it's a dedicated space in memory that the DS uses to track and control all Sprites (their current Frame or all of the Frames, depending on the code used) and their **Attributes** (Mosaic, Rotation, Double Size etc.). All in all, we have space for up to **128** individual Sprite data per screen, **32** of which can be freely rotated using the built-in rotation and scaling engine. The Sprite's Graphics Data (GFX) is stored in VRAM Banks, the Sprite sheets are cut up into 8 by 8 squares, meaning tiles, similarily to the previously discussed Backgrounds. Each engine **(Main and Sub, Main for the Top Screen, Sub for the Bottom one)** can support up to **1024 tiles** at a time. Sprites can use **16 or 256 colours** palettes - do note that all sprites use the same tile data, however each individual Sprite may technically use a different palette, so Sprites created from the same tiles may have entirely different colour schemes to save space rather than add additional tiles. OAM has to be refreshed each frame so that all the on-screen Sprites (their frame, position etc.) are displayed properly. Note the word "individual" – you can in fact display more than 128 Sprites at a time! What's meant by that is that within memory, you can store **128 *different* Sprites** using different attributes – you can re-use the same Sprite as many times as your own code or the library used allows you to. In case of NightFox Lib which we're using, it's **256 Sprites per Screen.**

Much like in the case of backgrounds, we will be using GRIT to convert our Sprites. Sprites are a composition of frames of pre-defined size and shape on one Sprite Sheet – the shapes allowed are **Square** (width is the same as height), **Wide** (width is larger than height) and **Tall** (height is larger than width), while the sizes allowed are **8x8, 16x16, 32x32, 64x64, 16x8, 32x8, 32x16, 64x32, 8x16, 8x32, 16x32, 32x64 pixels**. As you probably already noticed, those **sizes are all divisible by 8**. The way tiles are cut up is pretty simple – as with backgrounds, starting from the upper left-hand corner, **8x8 pixel** big parts of the original image are converted into **tiles**, the conversion takes place to the right from that point, and once it reaches the edge of the image, it goes a row lower until the entire image is converted. Knowing that, you can deduce that the easiest image to convert would be a Strip with the width of the original frame.

Once we have our Sprite Strip created, just like with Tiled Backgrounds, we put it in GRIT's bmp folder. Once our Sprite or Sprites are in it, we run the Convert_Sprites batch file, and after the process is complete, the resulting binary files representing the Tiles and the Palette data can be recovered from the sprites folder. Alternatively, you can convert Sprites using **Joint Palettes** – this way, you can use several Sprites using just one Palette. Keep in mind though that it limits the total number of colours you can use to 256 rather than 256 per Sprite, however when using a large number of them, you may end up having to resort to that.

You probably already noticed that the **Sprites do not have a Map file** – this is because the hardware itself calculates which Tiles are supposed to be used at any given time, copies them into VRAM if needed and OAM updates the displayed image. With the size and shape parameters pre-set, it can do it with relative ease, hence a Map is not necessary at all.
Certain limitations mentioned above, such as the size of Sprites can be overcome by using the 3D hardware of the DS, we'll eventually get there, but for now we're going to focus on using the 2D hardware.

Practical Use of NightFox Lib
Chapter 3 – NightFox Lib 2D MODE-0 Part 2 - Tiled Sprites

Now that we have some basic knowledge about how Sprites work in general, we can have a look at how the system works in NightFox Lib. By the end of this Tutorial, we will modify the previously written code to include a Sprite. First and foremost, we place our GRIT-converted Sprite in the "nitrofiles" folder. From here, we can move on to the coding. Seeing that in NightFox, Sprites are loaded from NitroFS, just like with Backgrounds, we start by Initializing the Buffers used by the library.

```
NF_InitSpriteBuffers();
NF_InitSpriteSys(Screen);
```

These two are pretty self-explainatory – the first function Initializes the Buffers, the second Initializes the system that controls them for the Screen of your choice – **0 for Top Screen, 1 for the Bottom one**.

NF_LoadSpriteGfx("Sprite", RAM_Slot, Width, Height);
NF_LoadSpritePal("Sprite", RAM_Slot);

These two functions load the Gfx data and the Palette of our Sprite into a selected **Slot in RAM memor**y. We have **256 (0-255) Slots** for Sprites and **64 (0-63) for Palettes**. We're not ready to display it though – as we learned earlier, **our resources have to be in VRAM to be displayed**. To transfer them there, we will use these functions:

NF_VramSpriteGfx(Screen, RAM_Slot, VRAM_Slot, Transfer_Flag);
NF_VramSpritePal(Screen, RAM_Slot, VRAM_Slot);

The Screen is self-explainatory and we already know of the **RAM Slots** used in NFLib, but we need to discuss the **VRAM Slots** and the **Transfer Flag**. As we already know, we can simultainously have up to **128 (0-127)** Sprites in VRAM – that's the value we are looking for then. As for Palettes, we can use one of **16 (0-15)** Palette Slots for the Sprite to use. "Wait!" – you're going to say, "You said each Sprite can use a dedicated Palette!" – that still applies, however one has to keep in mind that Palettes take space in their Bank – you can't use more simply because you'd run out of space – this is why I mentioned that it's often useful to use **Joint Palettes** for sprites with similar colour schemes. Finally, we have the Transfer_Flag – as I said, **you can either transfer one Frame at a time, or you can transfer all frames into VRAM at once** (the arguments are **true** and **false** depending on whether we want to transfer all the frames or not). Both options have their pro's and con's – using just one Frame at a time conserves more space, however this makes our Gfx Individual. **If you re-use the same Gfx, all the Sprites using that Gfx will animate to the Frame currently in VRAM**. If you copy all the frames into VRAM, **each Sprite using that Gfx will be able to animate separately, meaning that the Gfx can be Shared**. When to use which then, you ask? Well, it's really up to you, and it's highly-dependant on the design of your game. The general rule should be that if a given Sprite has more than one frame and the number of its occurances on-screen is higher than the number of its Frames, one should transfer all of the Frames to VRAM. It's relatively easy to calculate why – say, you have a Sprite that has 15 Frames and it occurs on-screen 50 times. If each of these 50 instances were to be Individual, you'd use up 50 Slots in VRAM. If you transfer the entirety of Gfx data at once, regardless of whether you use it 5 or 250 times, it will use up the same amount of VRAM. On the other hand though, if you have a Sprite that has 100 Frames and it occurs 10 times on-screen, there's no good reason as to why you'd transfer all 100 Frames – you can just as well create 10 Individual Gfx for them and use less Memory all-in-all. Always have memory in-mind – you have 128kb of VRAM for your Sprites per screen – use it wisely and strive towards conserving it.
Now that our data is in VRAM, we're ready to display it properly.

NF_CreateSprite(Screen, ID, VRAM_Gfx_Slot, VRAM_Palette_Slot, X, Y);

By now, all of those Arguments should be clear to us. This function will spawn our Sprite at the position indicated by X and Y.

This concludes the basic tutorial – now, it's time for some practice. Let's have a look at some code.

Exercise #3 - MODE-0 2D Sprites

Let's observe our Sample Code:

CODE

```
 /*
##########################################
##DS Programming Guide - From Zero To Hero!##
####Example #3 - MODE 0 Tiled Sprites    ####
##########################################
*/

/*
############
##Includes##
############
*/

// Include C
#include <stdio.h>

// Include Libnds
#include <nds.h>

// Include NFLib
#include <nf_lib.h>

/*
###############
##Main(){...}##
###############
*/

int main(int argc, char **argv) {

// Turn on MODE 0 on the Top Screen
NF_Set2D(0, 0);

// Set the Root Folder
NF_SetRootFolder("NITROFS");

// Initialize the Tiled Backgrounds System on the Top Screen
NF_InitTiledBgBuffers();
NF_InitTiledBgSys(0);

// Initialize the Tiled Sprites System on the Bottom Screen
NF_InitSpriteBuffers();
NF_InitSpriteSys(0);


// Load and Create the Tiled Background
NF_LoadTiledBg("Background", "Background", 256, 256);
NF_CreateTiledBg(0, 3, "Background");

// Load our Tiled Sprite
NF_LoadSpriteGfx("Sprite", 0, 64, 64);// Tempy!
NF_LoadSpritePal("Sprite", 0);

// Transfer our sprite to VRAM
NF_VramSpriteGfx(0, 0, 0, false);
NF_VramSpritePal(0, 0, 0);

// Create the Sprite!
NF_CreateSprite(0, 0, 0, 0, 0, 0);

while(1){
swiWaitForVBlank();
}
return 0;
}
```

We compile the code and… the Sprite didn’t spawn! Why…? What have we forgotten? Let’s come back to OAM. To properly display our Sprites, **we need to refresh its state**. To do so, we will have to update our OAM contents **in-between of our VBlanks**:

CODE

```
//Update NF OAM Settings
NF_SpriteOamSet(0);
swiWaitForVBlank();
//Update OAM!
oamUpdate(&oamMain);
```

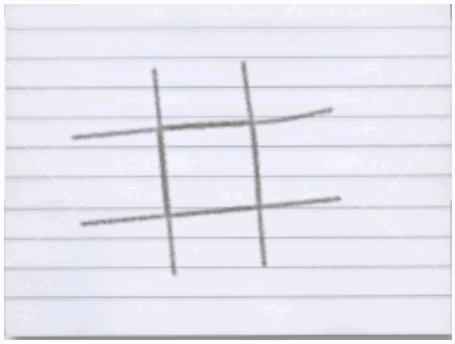
With these lines altered at the end of our applications, it should start displaying our Sprites properly!

EDIT: I've been informed (on good authority) that libnds has been modified to include 16 Ext.Palettes for backgrounds stock, which didn't use to be the case and I did not know that at the time of writing this guide. It used to be handled by means of setting the start adress of the Palette, now it's done by stating the Slot number. With that in mind, the sentence stating that this is a NightFox Lib limitation has been removed.

Excercises: Our very first game - Tic Tac Toe

Even with our very limited knowledge of programming, we're already capable of coding our very first game! I specifically chose Tic Tac Toe because the logic rules for this game are incredibly simple. Keep in mind that the code was created with the intention of being as straight-forward as possible - it may not be up to all "coding standards" for the sake of simplicity. ;)

In our game, we'll use these base resources:



When starting to code a game, one has to think about what rules will govern it. Our example concerns Tic Tac Toe, so let's list the rules:

1. There are 2 players and 9 spaces for either crosses or circles
2. Players take turns, each player can place one sign on one field per turn, then the turn changes
3. Players cannot put their sign on the other player's sign - the space has to be blank
4. To facilitate multiple games as well as quitting, bind a button (we will use A) to clear the playing field and reset the game

We will concern ourselves with those two for now and as we learn more about video game logic, we will improve upon this example. We know that there are 9 spaces on the game field - this makes it convenient for us to create 9 sprites - one on each space and change their frames depending on who clicks on them. We can use a loop to facilitate even spacing of the sprites.

Touching the screen within the boundries of the sprite will change the frame, but only when the frame is equal to 0 (blank frame, as seen above). The frame to which we change will be dependant on the turn - turns that are not divisible by 2 (1, 3, 5 and 7) belong to one player, the rest (0, 2, 4, 6, 8) to the other one.

We know everything! Now we can begin coding, do note the Comments in code:

CODE

```
/*
-----------------------------------------------
       Include block
-----------------------------------------------
*/

// Includes C
#include <stdio.h>

// Includes libnds
#include <nds.h>

// Includes NightFox Lib
#include <nf_lib.h>

// Create a Sprite data structure to hold important information
typedef struct{
       u8 ID, X, Y, Frame;
} Sprite_Data;

Sprite_Data Sprite[9];


/*
-----------------------------------------------
       Main() function block
-----------------------------------------------
*/

int main(int argc, char **argv) {

       NF_Set2D(0, 0);             //Set 2D MODE-0 to both Screens
       NF_Set2D(1, 0);

       NF_SetRootFolder("NITROFS");              // Set the Root Directory to NITRO FS

       NF_InitTiledBgBuffers();    // Initialize Background Buffers
       NF_InitTiledBgSys(0);               // Initialize Top and Bottom Screen BgSystems
       NF_InitTiledBgSys(1);

       NF_InitSpriteBuffers();           // Initialize Sprite Buffers
       NF_InitSpriteSys(1);        // Initialize Bottom Screen SpriteSystem

       NF_LoadTiledBg("TopScreen", "Top", 256, 256);    // Load a Background into RAM for the Top Screen
       NF_LoadTiledBg("BottomScreen", "Bottom", 256, 256); // Load a Background into RAM for the Bottom Screen

       NF_LoadSpriteGfx("Sprite_TicTacToe", 0, 32, 32); // Load our Sprite for the circle, cross and blank
       NF_LoadSpritePal("Sprite_TicTacToe", 0);

       NF_VramSpriteGfx(1, 0, 0, false);  // Load the Gfx into VRAM - transfer all Sprites
       NF_VramSpritePal(1, 0, 0);         // Load the Palette into VRAM

       NF_CreateTiledBg(0, 3, "Top");             // Create the Top Background
       NF_CreateTiledBg(1, 3, "Bottom");          // Create the Bottom Background


       u8 X, Y, ID = 0;            // Prepare Variables necessary for the field-generating Loop

       for(X=55; X<190; X+=45){           // Loop the X value between 55 and 190, add 45 each Loop
              for(Y=32; Y<167; Y+=45){           // Loop the Y value between 32 and 167, add 45 each Loop
                     NF_CreateSprite(1, ID, 0, 0, X, Y);              // Create a Sprite in the designated spot
                     NF_SpriteFrame(1, 3, 0);           // Set its Frame to a blank one
                     Sprite[ID].X = X; // Remember all the important variables for the Sprite in our Data struct
                     Sprite[ID].Y = Y;
                     Sprite[ID].ID = ID;
                     Sprite[ID].Frame = 0;
                     ID++;         // Next Sprite...
              }
       }
```

CODE (*continued*)

```
	touchPosition Stylus;		// Prepare a variable for Stylus data
	u8 Turn = 0;		// Create a variable to count turns

	while(1) {

		scanKeys();		// Scan for Input
		touchRead(&Stylus);		// Read Stylus data

		if(KEY_TOUCH & keysDown()){ // If the Touchscreen is touched...
			for(ID=0; ID<9; ID++){		// Cycle through all Sprites...
				if(Stylus.px >= Sprite[ID].X && Stylus.px <= Sprite[ID].X+32){
	// If the Stylus position is within the bounds of a Sprite in question...
					if(Stylus.py >= Sprite[ID].Y && Stylus.py <= Sprite[ID].Y+32){
	// Between X and X+Width; Between Y and Y+Height...
						if(Sprite[ID].Frame == 0){
	// If the Sprite's Frame is 0 - meaning it is neither a cross nor a circle
							if(Turn%2){		// If it's the cross's turn...
								NF_SpriteFrame(1, ID, 2);
	// Change the Frame of the Sprite to Frame 2 (cross)
								Sprite[ID].Frame = 2;
			// Save the change in our Data Struct for future reference
							}
							else{		// Otherwise...
								NF_SpriteFrame(1, ID, 1);
				// Change the Frame of the Sprite to Frame 1 (circle)
								Sprite[ID].Frame = 1;
	// Save the change in our Data Struct for future reference
							}
							Turn++;		// Next turn
						}
					}
				}
			}

		}

		if(KEY_A & keysDown()){		// If the A button is pressed, clear the playing field
			for(ID=0; ID<9; ID++){
				Sprite[ID].Frame=0;		// Clear Sprite Data
				NF_SpriteFrame(1, ID, 0);		// Change Frames to blank ones.
				Turn = 0;		// Reset the game
			}
		}

		NF_SpriteOamSet(1);		// Update NFLib's Sprite OAM System
		swiWaitForVBlank();		// Wait for the Vertical Blank
		oamUpdate(&oamSub);		// Update the OAM of the Bottom Screen engine

	}

	return 0;

}
```

In this code, we learn one new function, which will be useful to us once we learn how to Animate Sprites:

NF_SpriteFrame(Screen, Sprite_ID, Frame);

It's relatively self-explainatory - it changes the Frame of a given Sprite on a given Screen to the one selected - easy-peasy!

Now, analyze the code - **here's the Example itself, alongside its source code**, if you feel it is necessary to clarify anything, do post!