GBA/DS/Gamecube (GC)/Wii rom hacking 101.

This document aims to be the first proper release of the rom hacking documentation started by FAST6191 for gbatemp.net and sosuke.com, it is largely rewritten from earlier versions you may have seen before. This initial release will probably lack nice formatting but that should be rectified in short order while writing style may vary from informal to very bland technical document depending on what I was writing the original paragraph for/when I was writing it. Gamecube and Wii documentation may be a bit lacking when compared to the GBA/DS as hacking is still in its infancy on the systems but the techniques are good and the file formats/concepts used are very similar, identical to the GBA/DS or covered in the docs.

There is another document connected to this, it details the methods and though process behind hacking the hardware in the first place.

Preamble by FAST6191.

Back when this project/document was started the GBA was only just starting to be hacked, the DS was limited to a very small group of people for anything beyond rudimentary file system hacks, the GC was split across several sites and the Wii was still known as the revolution (and naturally did not figure into these documents).
Today as this paragraph is written the GBA release scene is all but dead and has several very high profile projects released and in progress, the DS still has many releases and also has some very high profile projects with tens of people in the teams, the GC release scene is long dead but the hacking scene has solidified (and is helped by the success of the wii) and the wii (which can run GC code) still has releases and not only has the file system decrypted but methods by which to run custom code other than homebrew built from the ground up. On all those consoles simple graphical tools (or even game specific tools/info) do not really exist at this time for all but a handful of games on all the consoles.
The overtly technical nature of rom hacking coupled with the tendency towards high level coding being taught elsewhere as well as the somewhat legally and ethically dubious nature of it makes people wanting to take up this fascinating subject can face a steep learning curve.
This document aims to help people come in “cold” (you know little of computers but have a desire to learn) as well as “retrain” (you can already code but this hacking thing is something relatively new) and although it is not explicitly aimed at such people it should hopefully be of some use for those already versed in rom hacking to use as a reference manual (the end of the document will serve this purpose). This relatively broad range of targets means some areas will repeat things. Other times things which have not be detailed extensively yet will be referred to, to some extent this is unavoidable but the document should allow you to skip backwards and forwards.
The original reasons for writing this and to some extent it still exists is that if you visit any sites with a focus on ROM hacking you will generally get told to learn to hack nes/SNES/Megadrive (Genesis to those in the US)/Master System ROMs and then move on to “harder” systems like the GBA/DS/GC and Wii.
Should you ask specifically how to hack GBA/DS/GC/Wii you will be told to look at the general/NES/SNES… documents to learn as it applies to “harder”/newer systems. While true it is generally nicer to read a document geared exactly towards what you want.

License stuff:

Thankyous. Rather than place them at the tail end of the document the people directly responsible are featured here.
Thanks from FAST6191 to:
People at gbatemp.net and sosuke.com, original hosts of this and extremely active discussion boards on GBA, DS, GC and Wii hacking.
Romhacking.net the people there have helped more than they probably know with this.
Deufeufeu, rom hacker, spec writer and sounding board for a lot this.
Martin Korth, author of no$gba and the awesome technical document on the GBA and DS (there would not be this document without it).
All team members of the original and forked Jump Ultimate Stars translation project.
Cracker, author of DSATM and countless other cheat tools, guides and codes for all manner of systems as well as discussion on this.
Slade, cheats guides, cheats and discussion.
All regulars of #gbatemp.net on efnet (now irc.gbatemp.net) and all regulars of #ezflash on irchighway.
Any and all authors of tools/guides/posts that have been linked.
Tom Waits, Korpiklaani and the other bands in my music collection. Music is good while writing technical documents.

That over the real fun begins now.

General layout and theory behind this document. This document will be split into several sections and things will be repeated and much like conventional education each time will have some more information added on.
There will be discussion of the fundamentals, an example application and a writeup of the specification (if one exists) for a given concept, if possible an external file/page will be linked in an attempt to display/describe the concept or a closely related one in another way.

If you lack an internet connection it is advised you download a copy of gbatek (http://nocash.emubase.de/gbatek.htm) beforehand as it will be linked repeatedly throughout this text.

A full list of contents is below but there will be three main sections
An introduction
Discussion of the areas concerned with hacking itself.
- underlying/core logic.
- graphics
- text
- sound/multimedia

Outlying areas (legality, project management and suggestions for teams)
List of links in document (unless specifically requested not to all links will have a copy retained and mirrored if necessary to help avoid dead links).

Reference guide for file formats and technical concepts (OAM layout and other relevant memory addresses). [incomplete as of preview release]

Contents.

Introduction.

This will take the form of a FAQ followed by a section of normal text. Some terms may be unfamiliar to readers but will be explained later in the document and will hopefully be sufficiently clear from context.

What is rom hacking?

Hacking has several definitions depending on who you speak to but here is means the action of figuring out how something is made and remaking it according to how you want it to act.
Rom is a shortened form of rom image and is it a copy of all the code needed to run a program, there is a distinction used between code from a chip and code from a piece of optical media with the former being a rom image and the latter being an iso image (usually shortened to iso).
For the purposes of rom hacking (what it is and what it does) though a rom and an iso are effectively the same thing and iso hacking is not really a term. The distinction is still observed though and talk of playstation roms will tend not to endear you to people.
Essentially rom hacking is taking code (usually from a game) from someone else, figuring out how it works and changing it in a useful manner (useful is in there as anyone can destroy something but altering while maintaining a working “device” is a skill).

What makes a good rom hacker?

There are many answers but the three dominant qualities and the ability to think logically, desire to figure out how things work and patience. Maths skills and computer knowledge tend to come with such qualities but as far as age, place in the world and work/knowledge backgrounds are concerned they are all represented in the rom hacking world.
Some advocate experience and while it is useful the following analogy serves a good example. It concerns normal human language;
How many people might you have met who have been speaking/writing a language for 50 years yet what they speak/write is awful with regard to what the language actually is: experience is not all powerful.
Likewise how many of you have met foreigners speaking your language who probably

possess a greater knowledge of the the implementations of irregular verbs and are far more able to communicate (even if it is their own language) what a pronoun is than you might be yet due to them only knowing 70 odd words they might as well not have bothered: technical knowledge is not all powerful.
On the subject of language English is probably the most commonly used language for this sort of thing (technical/scientific/engineering discussion) so it is probably best to become acquainted with it, Japanese is probably the most common language other than English (a lot of games get releases only in Japan) so it probably would not hurt to know something there either (a quick intro is provided in this document from the perspective of a programmer/hacker).
While this document aims to reduce some of need to search through thousands of sources of information the ability to do that never hurts either (you will see links to documents dealing with all aspects of computing and beyond later in this document), for future reference hacking/reverse engineering are often given the politically correct term forensic engineering, assembly programming is an invaluable source of information, programmers are people with a limited time and modern computers are complex so code will be reused, in much the same way hardware developers will not design a new processor from the ground up when an existing one works for the task (which feeds back into the programmers lacking time to learn an entirely new machine), homebrew coders will tend to have a lot of information/tools that are very useful to hacking (as an aside rom hacking is generally considered but a step away from piracy which is generally frowned upon by homebrew coders so pick your words carefully). The concept that links all those examples is the so called duality of knowledge.

Some thoughts though. Some study how computers work from the ground up and how the specific platform works and go from there.
Modern consoles (the GBA, DS, GC and Wii all count here) however do not tend to use assembly coding as much (just quickly assembly is the type of coding that revolves around changing the hardware manually, it is only different to altering the raw data the game uses by abstracting it to a more human readable form) owing to it be far more complex than it may need to be and the console makers should provide extensive software development kits (SDK) to developers. This means games often share features (such as methods of storing and displaying text/images) and this can be abused by rom hackers.
However the mere fact rom hacking exists should say that someone can do something better (or in a manner perceived to be better) than someone else. This means that purely relying on “SDK based” hacking can fall flat on occasions developers decide to change or write additions to (or even badly implement) the SDK or attempt to obscure their code (normally against cheat makers but this does have a knockon effect for rom hacking).

What do I need to do it?
Aside from some way to run the code (dealt with quickly later in the text but covered extensively elsewhere) you will need a computer, while there are custom/hacking oriented tools for unix like systems and beyond most are written for windows (and for those wondering WINE and co may not cut it). Aside from some aspects of emulation the machine does not need to be that high spec either, hex editors (the bread and butter of any hacker), spreadsheets (useful for manipulation of lots of hexadecimal where there is no tool that will do) and tile editors (although perhaps not the nicest ones) come as part of homebrew software development kits (which tend to feature the lower level stuff too) and those have all existed for many years and consequently there are versions for slower systems. If you can though get a system with a fair bit of memory and maybe two screens as you will often be doing a lot at once. Most tools used in this guide are free or there are sufficiently good freeware versions/alternatives.

What can be done?
Absolutely anything, the trouble comes in the difficulty in pulling it off.

What are the main sorts of rom hacks?
There are two main ways to categorise rom hacks and those are usually based on what is changed in the game and what is changed for the overall experience.

There are 4 main areas to a game
text
graphics
sound/multimedia
underlying/core logic.

What is changed in the game
Likewise there are 4 main types of hack
Improvement
Translation
Alteration
“spoof”

Improvement can be things from bug fixing to alterations of stats (to increase underpowered parts or decrease overpowered parts).
Uncensoring and removing regional changes is a popular thing to do here.
Censors will allow different things in different countries (violence is not such a problem for Japanese censors but US and European censor boards do not tend to like it so various parts can be cut or altered depending on region). Removing the blocks/alterations is frequently performed (the Wii version of manhunt 2 providing a good example), theoretically it is possible to censor a game but there is not likely to be much of an audience for such a work.
On the DS the process of undubbing is fairly popular, here people figure out how to swap the sound files (usually in a US or European release) from the dubbed version to the original Japanese (detailed extensively later but often a simple file swap).
Other things can include reworking of text as such things are usually altered to attract the broadest audience but can lose some of the “charm” to fans (mainly for anime/manga based games and “period” RPGs).
Another twist on this is backporting. Here later versions of the game may have some bugs fixed but may end up censoring something:
http://desertcolossus.com/compendium2/index.php?All%20About%20the%20Gerudo%20Symbol

Translation is exactly that.
The most common hacks in this category are Japanese to English translations although Japanese to Chinese (normally simplified) and English to mainstream European languages are fairly common. Slavic and other Asian languages (mainly Thai and Korean) are also represented.

Alteration:
This can range from new levels to new weapons to new stats to whole new games. Sure it sounds a lot like improvement in a lot of cases but improvement hacks tend to focus on making the original game better while alteration tends to encompass a broader range of changes/intentions.

Spoof:
Technically this sort of hack is an alteration but they are usually done as jokes, before dismissing them though note that they can get quite intricate.
They run the gamut from sprite replacement (putting Sonic into a Mario game was popular prior to things like Sonic and Mario Olympics and Super Smash Brothers Brawl both on the wii) to text alteration (one of the most well known hacks of this sort would be the GB hack of Zelda Wind Waker in a hack entitled link gets laid which changed the text to innuendo).

Regarding the areas of a game, hacking methods will be detailed later but
Text is the graphical representation of a spoken language and forms the backbone of rom hacking (all four areas listed above have prominent examples with extensive text hacking).

Graphics:
The graphical representation of a concept. 2D and 3D are the two most common methods of representing graphics in games.

Sound/multimedia
Music, sound effects and speech are the three categories of sound. In days past it was considered one of the harder areas of hacking (the GBA is a good example) but the DS, GC and Wii have a fairly consistent method of storing/playing sound which makes for easier manipulation.
Other multimedia (generally non-interactive) has become more common as computing power and space available has increased.

Core logic: the universe has physics to govern what happens and games have their logic. It is possible to alter the way the game calculates things and get it to do whatever you would like it to do (jump higher, fire faster, turn an RPG into a driving game).

How do I play rom hacks?

This question can and has formed entire documents and sites in and of itself so this will only cover the basics. Basic information on this is fairly easily obtained via the internet anyhow and there is some level of overlap with tools for manipulating games (hacking them) and those used to simply get the games running.

For information on how to extract and manipulate the saves of games see the cryptography section.

The two forms are original hardware and emulation.

Emulation

Emulation is the act of running code designed for another system on a system other than that for which it is intended.
The GBA is incredibly well emulated on PC for nearly every operating system going and the wii has a playable emulator (other consoles also feature GBA emulation but that would send this document far off track).
It will be covered in great detail later for as well as playing a game it forms an incredibly powerful tool for rom hacking (it should be said that hacking oriented emulators may not provide the best experience when compared to game playing emulators).

The DS has playable emulators on the PC (most major operating systems) but hardware requirements can be fairly steep.

The gamecube.
Early stage emulators for the PC exist but playability is very low.

The Wii.
No emulators or even projects confirmed to be attempting to emulate it exist yet (not to mention hardware documentation is still ongoing)

On the hardware.
All systems covered in this document feature methods of running the altered code on the actual hardware.
The whole area of running code on a device (especially ones the manufacturers do not allow “out of the box”) is a fast moving area and owing the tendency for most features for devices to appear well after launch up to the minute info can be hard to come by. This means research is worth doing and asking questions is important, some advice though; forums, irc channels and the like are frequently asked “what is best” so try to show that some research has been done and make sure to add in any conditions required into that (no soldering, price range, support in a language you know...). There are also open source/highly customisable methods (aside from some aspects of the GBA and gamecube) so if something really does not meet your requirements then consider using one of these methods.

GBA:
There are three main ways to run GBA carts. The GBA itself, the DS(see note) and the GC via a device called the GB player.
ROM is an acronym of read only memory which as it implies can not have new code written to it. Therefore carts originally sporting flash memory were made which allowed code to be written and run from them.
Note: the DS has two slots in it. Slot 1 aka the DS slot and slot 2 aka the GBA slot. The DS slot is too slow to run GBA code directly so only GBA slot devices have the capability.
As a sidenote the GBA requires code to be readily available and will generally crash upon all but the shortest of delays so originally carts used very fast memory (usually very expensive and with some awkward limitations), however some people decided ram (which is very fast but gets erased when power is lost) could also work.
Today the carts are still available and come in two main forms.
Memory+ram arrangements. The memory is usually either large bank of NAND memory or a memory card of the CF or SD (including mini and micro but not SDHC) variety.

Expansion packs (usually with ram and/or some other form of memory onboard).
These are geared towards DS owners and require a DS slot cart to get new code written to them. Note also that a lot of these devices are aimed at the DS lite which has a smaller GBA slot (unable to encase carts the size of GBA originals) and by virtue of that they are unable to be inserted in original GBA devices or the original models of the DS and the GB player, at least without some modification of the case.

Compatibility is nearly 100% (no real showstoppers) for the good carts and it is usually trivial to get code running in the first place. In the case it is not quite so easy there are many patches available which can be easily applied to fix problems.
The most common memory+ram carts are the EZ3,EZ4, M3 (various versions) and g6. In the case of the EZ4 and m3 there are versions that have limited GBA capability (the EZ4

lite compact and m3 professional) so it would be ill advised to get those if GBA code is something you want to do.
Aside from the EZ3 they all have DS lite sized versions as well. There is another popular line called supercard but the GBA compatibility of the whole range is not very good.

The expansion pack market changes frequently but the three main cards are
EZ5 3 in 1 aka 3 in 1 aka EZ 3 in1.
M3 real expansion pack (be sure to get the correct version)
Ewin expansion pack.
Consult the internet for advice on choosing an expansion pack.

DS:
Presently only the DS can run DS code but there are two main options.

GBA slot/slot 2 devices.
DS slot/slot 1 devices.

The GBA slot was the first method by which code was run. The major downside is that it will not work out of the box so you need to either get a device for the DS slot to trick your DS into running the code (see nopass and most DS slot carts feature this ability) or modify your firmware (see flashme, catch is that it requires at least one use of a nopass).
Today GBA carts have been overshadowed by DS slot carts although compatibility is still very high for those still being updated (mainly the supercard line and the EZ4 line) and with some effort few of the features DS slot carts have over GBA slot carts (cheats mainly) are still possible to be used. The major problem is no GBA slot cart supports SDHC (an improvement on the SD spec to allow for larger and faster carts) whereas several DS slot carts do but they do support GBA (EZ4 and M3) and have ram able to be used for certain DS homebrew applications.

DS slot carts.
Again the market changes quickly, the most common cart (and most commonly heard of, to the point where the brand name became a synonym) is the R4/m3 ds simply and while it works it is not considered a good purchase these days. Others include the SC-DS one, M3 real, cycloDS, Acekard RPG, Acekard 2, EZ5/EZ5+ and many more.
Currently they are the most popular way to run code. All that is needed is a method to transfer the code to the memory used (usually microSD and SDHC is supported for most of the carts above, read specifications though before purchase as some have earlier models without SDHC).
Simple to use cheats (cheats are available for GBA slot carts via third party tools like DSATM (more on cheats later) but DS slot carts are easier to use for this), soft reset, download play support (without flashme), drag and drop ease of use, extensive and current support from third and first parties, high compatibility (not necessarily higher than GBA slot devices but it is less likely a new release will not work on a DS slot device) and more exotic features like savestates (far from emulator grade savestates), slowdown (can be done via DSATM) and in game menus (for example to bring up a guide) mean they are the most popular option.
Downsides are that there is no way to run GBA code (without an expansion pack or similar, often supported on cart or via software) and no extra ram for homebrew that can use/needs it (again fixed by having an expansion pack or another GBA slot cart).

Note. An additional version of the DS called the DSi was released in late 2008. These blocked existing flash carts but various manufacturers brought out new cards that work on

the DSi, they are usually just tweaked versions of the current versions of the DS slot cards though.

GC:
While there exists methods by which to load code from a memory card (SD adapter or the so called USB gecko) + action replay combo a modchip is considered the best way to run code.
There is also a choice between miniDVD and full DVD with case replacement. Debates are plentiful but the present price of miniDVD and the general lack of a decent library mean miniDVD is still very viable.

Wii:
The wii can support Wii and gamecube games, unlike the GC the wii can read a whole DVD and with the appropriate hardware this also applies to the gamecube.
Modchips, specifically drivechips, are the best way to execute commercial code at present.
There are several drive chips available and they include:
early DMS, D2A and early D2B
Simple to mod with readily available and even fully open source chips.

Cut pin D2B. Here three essential pins were cut from the chip meaning people have to cut into the chip itself or use one of the later chips that use different points that were not cut.

D2C and beyond. Highly complex chips which entail extensive soldering (to pin or reverse of PCB) or a clip on device (fairly costly and often of dubious quality).

Some of the latest consoles also have the drive controller merged with other chips on the board making chips unworkable.

Drive replacement: you can replace the drive from any wii with any other wii drive but this refers to chips like the flatmii which sit on the drive ribbon and can load games from a PC.

GC code can also be run from a wii and most chips allow for perfect play but some have issues with audio streaming titles (which unfortunately include some of the best the GC has to offer) and workarounds may need to be found.
The action replay trick worked for earlier wii firmware versions but was blocked by a header check in an update along with a few choice pieces or GC homebrew (easily fixed).

Softmod: in conjunction with the methods below a handful of softmods exist for both wii games and GC games. At this time modchips will give a better playability but many games can still be played via softmod.

However while the chips allow copies the actual code itself is signed (more on cryptography later in the guide) meaning only 1:1 copies could be run (aside from some very rudimentary header hacks to dupe some region checking and update checking). In earlier firmwares there exists a bug with the signing (also explained later in cryptography) which allowed for people to dupe the wii into thinking the disc was signed.
The word earlier is the key word there. The bug was blocked in the otherwise fairly minor 3.3 update (along with an older homebrew method explained in a second) but all is not lost.
The people behind the USB gecko mentioned in the GC section made a channel to launch discs using an earlier IOS (think of it as group of operations available for a game) for their device (which also works without it although it loses features). Here an earlier IOS is used

and the bug is still available to exploit. Others have made/are continuing to hack the IOS updates themselves but this is very much bleeding edge work.

These hacks however require a method of running homebrew which is where the twilight hack comes in.
Although blocked in the 3.3 update there were still bugs (http://hackmii.com/2008/06/wii-menu-tp-hack-killer-analysis/) and the twilight hack was reborn.
The hack involves a buffer overflow with Zelda Twilight Princess (which you can run from a copy if you have a chip) and a tweaked save which was then made into an ELF loader and then allows you to both run code and install custom channels (the geckoOS (formerly gecko region free) channel being the one of choice here).
The signing bug (also known as the trucha bug in some circles) allows you to also gain complete region free, bypass chip checks (some older chips have not been/can not be updated) and of course hack the isos.
A small sidenote is the wii scrubber from Dack. Wii isos use a full DVD disc (most are single layer but some are dual layer) but code is not necessarily a full disc so there is junk data between the files.
This junk is not signed and thus can be altered freely (the key is still needed to determine where the junk is) which is good as the junk is essentially random for most games which means can not be readily compressed. Replacement of the junk with simple 00/FF's means conventional compression works again.

Sidenote. The wii features so called wiiware (games that run from the NAND memory section of the wii) and emulators called virtual console, a twilight hack or equivalent is required for hacking these sorts of games.

If you are interested in the theory behind all of the wii hacking a presentation was made at 25c3, you can download a copy http://www.4shared.com/file/78190880/6856af6f/25c3wiifail.html Format is H264 (high profile unlimited level) AAC (He-AAC+PS) in MKV. Other versions should exist in multiple other places.

Is it legal?

A discussion on the legality of rom hacking is included after the hacking section of the document but the whole area falls under intellectual property law and “complete” answer to such a question is again a document in and of itself. Some aspects of rom hacking can also run up against cryptography law which is another rats nest of law.
However provided the original rom is not distributed (why rom hacks are normally distributed as patches) and the work does not represent a loss to the company involved (translating a game due out in a month in the language/region covered by a hack would not go over well) companies do not tend to mind.

How hard is it to hack/what is the hardest type of hack?

Simple answer is that there is none. Generally speaking text and graphics are the easiest with sound and core hacking being harder.
Contrast however the difficulty of figuring out what something like the following means:
pmr fsu O es;lrf fpem yjr dytrry (“one day I walked down the street” with all characters shifted by 1 to the right on a standard UK keyboard)
Add in some compression (common in text) and it gets even harder.
Now compare it to the fictional (but not unrealistic) example of longsword1stats.txt for a

core logic type hack
[all values max=255]
attack=180
speed=150
mass=10
defence=40
[end of text]

Which would be easier to edit?

In terms of what sorts of games make for the hardest/most time consuming hacks projects aiming at translating RPGs tend to be the longest and be the most involved while puzzle games tend to be the simplest. Again there are examples for and against that generalisation from the point of the hacker (the actual translation is usually what takes the time though). It also depends on the level of polish a project has (general text may be found in simple files but menu text can be hidden all over the place).

Rom hacking.

The key to all rom hacking is to ask “what does it do?/what does it represent?”. The reason you are able to read this document is over the course of several thousand years a series of lines and the sounds they represent has been taken by a group of people to mean a certain thing.
Computers however are not so lucky and any single part of the code that makes up a rom/iso can mean something (or nothing).

However there is a fundamental concept shared with all current computers* and that is called the bit.

On and off, 1 and 0, i and o, high and low are just some of the names used to describe the concept.

Binary is the word and binary means “two states”, therefore a bit being a binary entity can represent two things *(2). What those two things are do not necessarily have to be related but generally it is related.
This document will aim to cover methods of finding out what something means and because what a particular bit represents is not always “random” (computers are made by people and people generally like to communicate/be able to follow what is going without learning something new).

*Anybody mentioning quantum computing can probably skip most of this document.
*(2)There are some methods of cryptography and the principle of lossy compression that can skip parts and then get others to fill in the blanks but that is getting ahead of things.

The words on and off, high and low imply a signal of some form and that is the original basis for binary (as an aside in 99% of cases you will meet 1 = high and 0 = low but there is the case of negative logic where the situation is reversed: http://www.tpub.com/content/neets/14185/css/14185_85.htm). Two however is rather limited given the complex nature of things so things have to change; stick two binary digits together and you have 2 times 2 aka 4 things able to be represented. Stick three together and it is 2x2x2 or 8 things.
1110101100101010100101010101010101 is a pain to write, read, say out loud and

manipulate in a persons head.
A stopgap measure is to break it up like large numbers are:
1110,1011,0010,1010,1001,0101,0101,0101,0101
This however still faces a problem with larger numbers so a new way of representing it is required. Step in hexadecimal.
Before getting to that some groundwork has to be done, early in life people are taught about thousands, hundreds, tens and units.
That is to say the number 1234 can be written as
1 thousand, 2 hundreds, 3 tens and 4 units. Awkward but accurate. To aid in the example in a few lines 1000 is 10^3 (ten to the power three), 100 is 10^2, 10 is 10^1 and 1 is 10^0 (any number raised to the power 0 is one))
Humans have ten fingers and ten is fairly good number in that it can easily divide 2,5 and 10. As such basic numbers are known as base ten (it will be further delved into a few paragraphs from now but it is signified by a subscript 10 after the number although it is almost never used except in examples such as this).
Looking at binary it is possible to see that the two options available to be represented can be represented like conventional numbers
1011 is 1 multiplied by 2^3, 0 multiplied by 2^2, 1 multiplied by 2^1 and 1 multiplied by 2^0
Calculate that and in decimal this would be 11.
Binary then is known as base 2. This has not solved the problem of it being awkward however so another base is used.

10 is not a power of 2 though so binary can not be easily represented by 10 digits so another base is needed. Two options are fairly simple and those are base 8 (8=2^3) and base 16 (16=2^4). Base 8 is known as octal and uses the characters 0,1,2,3,4,5,6 and 7 while base 16 is known as hexadecimal.
Hexadecimal is the most common throughout computing. Some among will notice that 6 is hex (in languages of days gone by) and 10 is decimal which put together (with an “a” for some reason) become hexadecimal.
However there are only 10 numbers (0 through 9) and to display 16 variations with them would be impossible so someone somewhere at some point decided why not use letters.
Now we have 0 in decimal being equal to 0 in hexadecimal with hexadecimal=decimal numbers up to 9, when decimal rolls back around and gains another digit hexadecimal starts on the letters. That is to say A = 10 in decimal, B= 11, C=12, D= 13 E=14 and F=15).
Here is a table for easy conversion although you would do well to learn it (at the very least the decimal, hexadecimal part).

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000

9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

After you get to F in hexadecimal it simply becomes 10 (pronounced one zero, not ten), you convert in a similar manner as binary to decimal but you use a 16^x instead of the 2^x you would use in binary I.E.:
F3 is 15 lots of 16^1 and 3 lots of 16^0 (aka1) which gives you 15x16 + 3 or 243.

With the concept of the base of a number now understood the “joke” as follows should make sense :
“there are 10 kinds of people in the world: those that understand binary and those that do not” (10 is the binary number for 2 in decimal for those not keeping score).

The question that then arises is “how do I tell binary and decimal (or any other base) apart then”?

Answer: well the first clue is that binary is all 1's and 0's but as the “joke” above clearly shows it is not always that easy so strictly speaking there is a need to put something to signify it.

As mentioned already in general maths it takes the form of a subscript of the number system you are working with the subscript number being the decimal version of whatever number system you happen to be using (2 for binary (also called base 2), 10 for decimal (base 10), 7 for base 7….).

A problem arises from the awkwardness of putting a subscript value where computers with simple input methods (most of them) are concerned.

The most common of these being as follows:

0xUUUUUUUU where the “U” characters after the x are any hexadecimal number (if more than 8 hexadecimal digits are used the 0x part is sometimes repeated)

hXXX or XXXh where the XXX can be any length and is made up of hexadecimal numbers

%XXX or $XXX: (same idea as hXXX, most often used in assembly language although you may recognise it from your web browser where it uses %20 to signify as space key (20 hexadecimal is the ASCII method of putting a space key)

The method used in this guide (most people tending to use the following or the hXXX form) being XXXXXX (hexadecimal) or XXXXXX (hex) where XXXXXX is any hexadecimal number of choosing.

You can rest assured you will also see hexadecimal called hex on many occasions (forums, instant messaging, newsgroups, documents (the author has had to resist calling it hex nearly every time so far) and just about everywhere else where such things will be discussed).

Strictly speaking hex means 6 but unless it is one of those ever frequent conversations about computers and witchcraft (“I made a hex in raw hex before my enemy could make a

hex from the raw hex lying around “etc) you will not be mistaken when saying hex instead of hexadecimal (a good example of this is a program that can view files as raw hexadecimal and edit them are known as hex editors and they will probably never be known as a hexadecimal editor although the term binary editor have been used on occasion).

As this document is ultimately about games it is about time something was referenced on the subject: you will notice that FF(hexadecimal)=255(decimal), guess why the largest amount of items you can pick up or stats you can attain is often 255?

Answer: the programmers used a hexadecimal to decimal conversion for the number they wished to represent. (To a lesser extent it is also why other stats are limited to 511,127 although the reason for this generally lies a bit further into the computer side of things and it can get slightly more complex, enough to be skipped for now).

Before going on (and it will be explained fully later along with a few key concepts) computing largely deals with small sets of data (the processor in the computer this is being displayed on is perhaps limited to operations with numbers no larger than 64 bit, more likely 32 or even 16 bit for certain things). This feature is a proverbial two edged sword in that it provides both a limit that is very complex/costly to work around and causes the relative ease by which a computer can be operated/programmed. Secondly while you can use numbers/addresses that are not a power of two (bits) (and smaller than the “bit” of your processor) in a computer it is quite cumbersome so it is not usually the done thing now that memory is relatively inexpensive and plentiful.

For more on hexadecimal (especially about maths and or just another persons take thereupon) consider reading chapter 3 of The Art of Assembly http://webster.cs.ucr.edu/AoA/Windows/index.html .

Much like the complexity of putting a base as a subscript in normal text it is also not easy to represent negative or fractional numbers in the raw binary/hex as the guide has presently defined it. However this is a relatively special case and it is not usually needed for entry level hacking (although certainly worth taking the time to learn) so it will be presented after a more important section.

Binary and Hexadecimal wording:

If you have understood it all so far you are doing well, it will probably take a week or so of using it to sink in but once it is there it will become second nature.

Now for some of the more tricky stuff regarding binary and hexadecimal numbers that is important to ROM hacking.

A single binary digit is known as a bit, it can be used to represent 2 different things (what these are actually does not matter provided you realise it can only be two things: it can be true or false, right or wrong or even yellow and an apple). (A simple application of thought would tell you decimal can hold ten values and hexadecimal 16)

As you may recall stick two together and you get 4 possible variations: 00, 01, 10 and 11.

As an aside it is the largest number of bits that a processor can store in one of its internal memory sections called a register that determines what “bit” you machine is.

In practice though 1, 2 or 3 bit(s) is rarely ever used as numbering. 4 bits as already

discussed is what is represented in a hexadecimal number, 4 bits stuck together is called a nibble (sometimes spelled nybble).

Stick 8 bits together (2 nibbles remember) and you have a that ever present computing term the “byte”, this is usually the smallest value a computer can address and are very important in ROM hacking as they often give the location of various pieces of useful information, represent useful information and loads of other stuff.

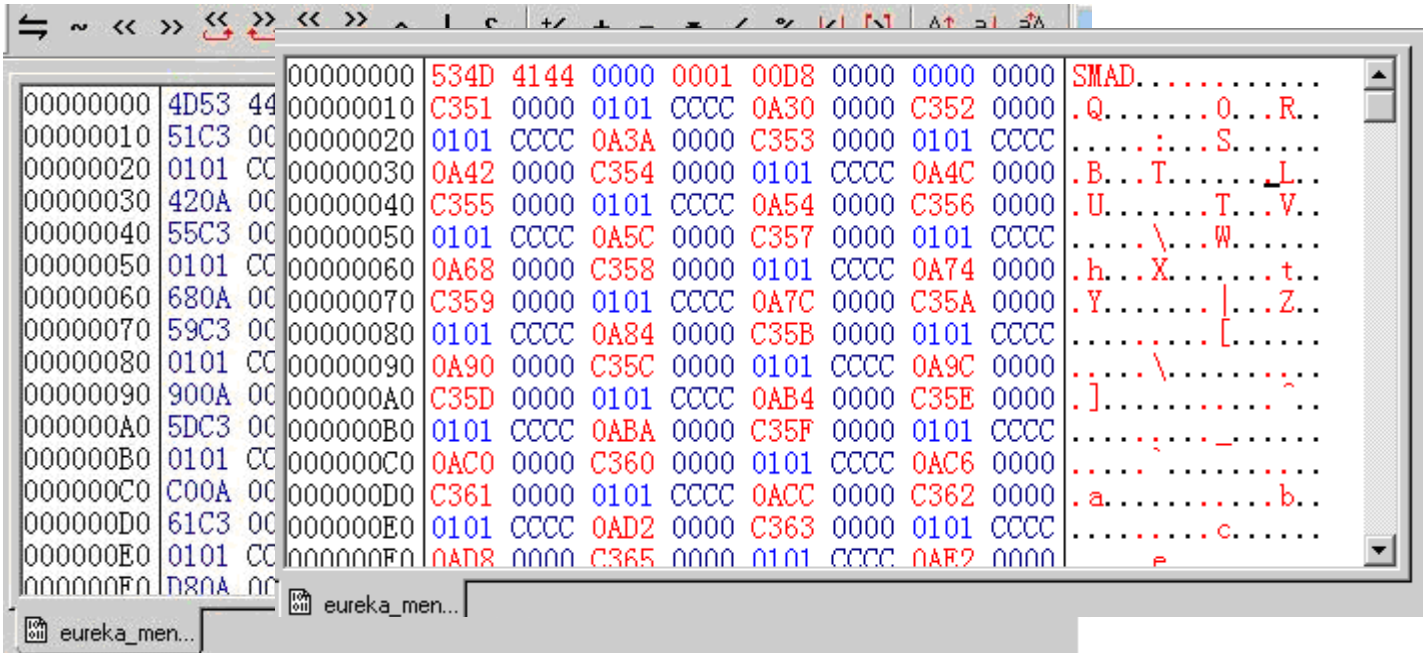
Stick two bytes together and you have a word (as a ROM hacking example pointers in the .msd files found in the Japanese release of Final Fantasy III DS are the byte addresses of the text the files contain).

This is where it becomes important on the DS and GBA (and most other consoles): these bytes like in the pointer example above are flipped around meaning you either have to flip them manually (your hex editor should have a provision for this) or you can think around the problem. In technical terms the two bytes you stick together are called the high order (HO) and the low order (LO) byte with the high order byte being the part that would for the largest section of the number:

e.g.1234 (hexadecimal) is a number with the 12 being the high order byte and 34 being the low order byte, swapping the bytes gives 3412. As already mentioned this procedure is useful for certain files or sections within for the DS.

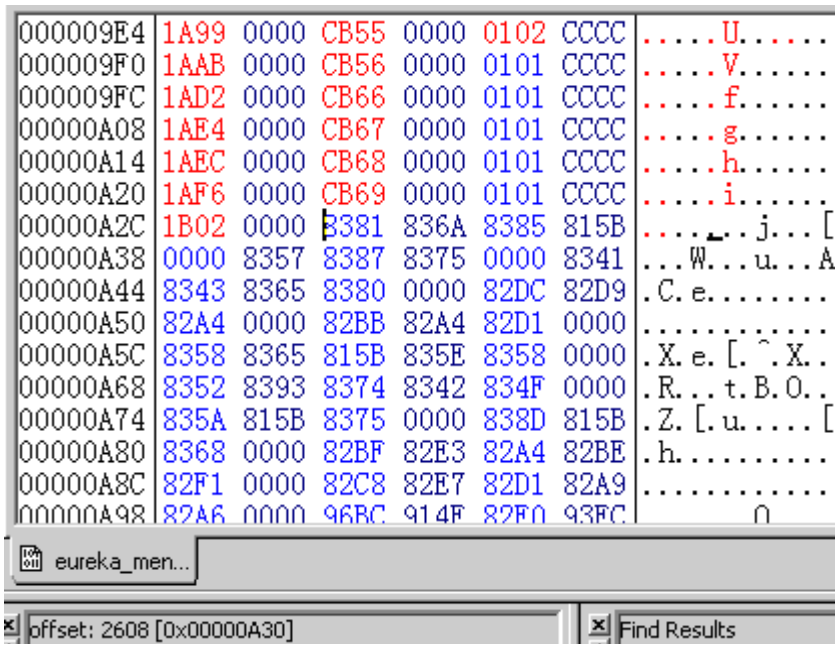
Not that important for us but if you stick two words together you have a double word (dword) and stick 4 words together to get a quad word (qword), should you ever play around with large files (e.g. sound files on the DS) you may have to be swap LO and HO words about to get an easily readable address/file length

For example:



The first picture shows a the original file while the second one shows what happened after the contents of the byte order was flipped (a 16 bit unsigned short byte flip was used in this

case), while it probably makes little sense here if we look at offset 0A30 you will see the text starting (the text in this case is shift_JIS so the inbuilt ASCII converter of the hex editor is not much use), now if you look at offset A3A (count on 10 spaces) you will see the next text section.:



Do not worry though if you did not follow what was done exactly with respect to the editing of a file as it will all become apparent later, this example was just to show the importance of byte order.

If you do not have the ability to flip bytes in your editor then the following is what occurs:

A number:

114F 45AD

You give a letter to each byte

A B C D

You rearrange to read*

D C B A

AD45 4F11

*you can sometimes get away with only flipping 16 bits/2 bytes but it only works for small files. Large files (which are common on the DS and up) will not work.

The DS and GBA make frequent use of a so called 4 bit per pixel imagery, in this case the nibbles are flipped around compared to how they are displayed (there is an example involving a DS icon later in the text).

Chapter 3.3 of the Art of Assembly is a good alternative to this and is good for further reference:

http://webster.cs.ucr.edu/AoA/Windows/index.html

Logical Operations:
This is more often the domain of electrical engineering but such operations are used in ROM hacking on occasion and they are especially prevalent at hardware level, they are also useful for the manipulation of numbers. Such concepts will not be heavily dwelled upon though and any interested parties are advised to find a resource dealing with electrical engineering (a high school electronics book will suffice).

NOT (also known as inverse)
The basic operation, also called an inverter it changes the signal to the other sort: 1 becomes 0 and 0 becomes 1. Signified by a bar on top of the number to be inverted i.e. $\bar{1}$

AND:
Takes a minimum of two inputs and outputs a 1/high/on if and all inputs are 1. Signified with a fullstop between numbers i.e. A.B

NAND:
Takes a minimum of two inputs and outputs a 1 if just one of the numbers is 0/off/low. Think of it as a AND operation followed by a NOT. Signified by a bar on top of a AND arrangement.

OR: Takes a minimum of 2 inputs and if one of both is 1/on/high outputs a 1/on/high. Signified by a plus symbol i.e. A+B

NOR: Much like NAND is an AND followed by a NOT a NOR is an OR followed by a NOT.

XOR: exclusive OR, This one is more complex, only the two input case is dealt with first. Takes a minimum of two inputs and only outputs a 1 if only one of the inputs is a 1/high/on none on or two on and it outputs a zero. Signified by a plus with a circle around it or occasionally a ^.
Multiple inputs, two “implementations” exist. The first variant requires that one and only one input is a 1/on/high. The second variant is where multiple two input gates are connected in series with the output of one feeding the input of a following gate.

XNOR: XOR followed by a NOT.
http://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/10-gates/00-gates/xor.html

Not strictly logic operations but important none the less.
Be sure to pay attention to the order of the bytes.

Bit shift.
In decimal maths you can quickly multiply or divide by 10 by shifting the decimal point right or left respectively. A shift can be done to binary digits in much the same way. Note though that shifts are limited to a given number of bits so for a 4 bit number shifting 1 bit will lose one of the bits (shifting back will make the number that was lost become a 0)
This operation is used for all manner of things including encryption/obscuring data (usually if there is enough space to shift without something important being lost), getting the number in a suitable format for later operations (removing the high order bit from a NARC file to get the address can be done by shifting one way and then the other).
Some operations do it only upon a given section (for example 8 bits) while others do it for

the entire section so make sure you use the correct one.

Bit rotate.
Acts in much the same way as a bit shit but instead of losing a value it returns it to the other side. 1001 1000 shifted to the left would become 0011 0001
Used in much the same way as bit shift but for situations where losing data is not desirable.

Binary coded decimal aka BCD.
Often a tool to help get the fairly abstract nature of non-base 10 numbers it occurs in rom hacking. The DS in fact uses it in the firmware
http://nocash.emubase.de/gbatek.htm#dsfirmwareheader
Similar to the playground game with the jugs of water (get exactly ? volume of water using given jugs) you can try to get a decimal number using binary.
The most obvious start is the binary power system or 8, 4, 2 and 1
Here any number from 0 to 9 can be created by using the numbers above.
Other methods include
7,3,2 and -1
6,3,2,1
5,4,2,1
Most however will use the 8,4,2,1 method and it comes into play when multiple digits are displayed where it each digit is literally the binary interpretation.
0011 0001
It is not 49 but 31
0011 = 3 and 0001 = 1
Times and dates are often encoded in this manner and it is an older method of fractional numbers (numbers after the “decimal” point).

Hexadecimal maths and other important notes on hexadecimal.

A few of you may have taken computing 101/electronics 101 type modules or classes over the course of your education and are now shuddering at the thought of this. Others who have just learned hexadecimal are maybe starting to think about the consequences of that last sentence.
Well do not worry, while there is much joy and frustration to be found working with hexadecimal and binary when it comes to even simple maths (hexadecimal and binary division is not for the feint of heart) 90% of the time ROM hacking needs only basic addition and subtraction to be done in hexadecimal (pointer tables, offsets etc) and as far as 90% of ROM hacking goes provided you can do basic maths it is the same as ordinary decimal maths, remembering to take into account the different number system. Of course it will not hurt to be able to multiply and divide and be able to tell someone what a signed byte is.

So right off the bat what is 10 – 1 (numbers in hexadecimal)?

The answer is F of course but it is all but guaranteed many reading this will have said 9. This is as we are all taught to think in decimal since we are young (the author will refrain from any social commentary regarding the merits of teaching your kids to think in hexadecimal) and in a panic/rush/moment of excitement you will almost certainly revert back to decimal (in ROM hacking this will come when you have found a table, extracted the text, translated it, reinserted it (maybe having done some other stuff too) and are now just altering pointers to get it all working properly again).

For this reason a double check is needed or better yet you do the conversions beforehand on a calculator, hexadecimal calculators do exist although you can expect to pay a little bit more for them. Windows has a hexadecimal and binary calculator built in (click the view pulldown menu then choose scientific) and there exist many freeware alternatives that are as usual far better than windows inbuilt stuff. Can it also be suggested that for something like a pointer table where there might be more than 90 calculations that need to be done a spreadsheet (Microsoft office supports hex with the Analysis ToolPak (you will likely need the CD) while open office supports it out of the box).

Fractional and negative (signed) numbers.

An important concept, even if only in some of the higher levels of hacking it is easier to learn them now than trying to wing it later.

Negative/signed numbers.

Much like the problem with signifying the base of a number above there is difficulty in representing a negative symbol with only binary. Unfortunately there are multiple methods in common use and much like the much maligned “10 kinds of people” joke it is not certain what kind of system unless it is stated (one fortunate aspect of rom hacking is you can experiment to find it out if it comes down to it).

First if the situations that only deal (as in negative numbers do not exist not merely do not occur in the sum being done at the time) with positive numbers the are called unsigned whereas negative numbers are called signed. While the twos complement method is the dominant method in most of computing rom hackers are not so lucky.

Here 4 methods are common.

Sign and magnitude also known as a negative flag (although the term can be used more generally when dealing with signed numbers) and signed magnitude.

Here the first bit is given over to being a sign with the rest of the numbers being interpreted as usual. It is the most similar method to conventional counting/maths. 0 means positive while 1 means negative.

0000 0001

Equals 1

1000 0001

Equals -1

Second is ones complement.

Here a bitwise NOT operation is applied to a positive number to generate the negative counterpart.

0001 becomes 1110.

0010 becomes 1101

0011 becomes 1100

Next is twos complement

This one is the more complex than the other two but does not have the pitfalls of the other numbers (two representations of zero and simple maths is possible).

Here the ones complement is made (bitwise not to all the digits) and then 1 is added to the result using conventional binary addition*

The example of -1

0001

ones complement

1110

add 0001

1111

Example -3

0011

ones complement

1100

add 0001

1101

*0000 becomes 1111 and then 0001 0000 but the other part is ignored.

Similarly 2 decimal 0010 added to -1 1111 (which gives 0001 0001) the leftmost “spillover” is ignored.

“Excess 7”. Strictly speaking this is known as excess 2^(m-1) where m is the number or binary digits you have to work with. It becomes more complex (and important) when fractional numbers are involved (see section below).

Almost a hybrid of Signed magnitude and twos compliment.

This example will be using 8 bits but it should be easy enough to apply to other areas.

Here the first bit is used to signify a sign (note not the sign of the number per se) giving you an extra 7 to play with or an excess 7 if you will.

Several years ago you may have used the make it large method of avoiding signed numbers.

For instance

7 -6 -6 +5 -4 +9 = 5

It is simple enough but in a rush it is possible to miss a sign and consequently get an incorrect solution.

Adding 10 to all the numbers though

17 + 4 + 4 + 15 +6 +19 =65

-6 x10 or -60

=5

A similar principle is used here and a number known as the bias value is chosen to make sure the numbers are all above 0.

In the case of the excess 7 (also called 8 bit excess 127) this is usually* 01111111 or 127 in decimal.

Here the value you want to encode is either taken (positive numbers) or added to (negative numbers).

*in theory (and so probably in some game somewhere it holds true) it does not have to be 127.

Another way of looking at it could be to start counting with the lowest number possible (-1 +2^m) or for 8 bits -(-1 + 2^7)=-127 and call that 0. 0 (as in nought) is then the bias value.

It has the advantage of being easily compared with other numbers so it is worth knowing about.

Examples

Number to encode -7 or (-) 0000 0111

0111 1111 – 0000 0111 = 0111 1000

Number to encode 18 or 0001 0010

0111 1111 + 0001 0010 = 1001 0001

Examples

```
0000 0000 -127
0000 0001 -126
0000 0010 -125
0111 1111 0
1000 0000 +1
1000 0001
1111 1111 +128
```

In the real world (namely IEEE 754 which is the de facto standard for floating point numbers) 32bits or more can be used with the first being the sign, the next being the bias value and the rest being the encoded number in question.

For extra reading, some real world examples and some history behind it
http://www.math.grin.edu/~rebelsky/Courses/152/97F/Readings/IEEE-reals.html

http://www.psc.edu/general/software/packages/ieee/ieee.php

Signed numbers are also one of the reasons for some stats ending at 511 or 127 or similar. The other (assuming the first bit is not simply ignored) being the first bit is used for some other flag (the DS file format NARC uses it so signify a subdirectory for example).

For another source on the subject

http://www.cs.grinnell.edu/~rebelsky/Courses/CS152/97F/Readings/student-binary.html

Fractional numbers/real numbers.

Fractional numbers are usually done using a so called “floating point”, it is the subject of quite complex so the basics will be covered here and for more you will be referred to http://docs.sun.com/source/806-3568/ncg_goldberg.html for a more in depth discussion.

Two important concepts come into rounding numbers (especially useful due the inaccuracies created when working with floating point numbers), normal rounding of numbers tends to involve picking the middle number between the highest precision number you care about (for a “round to the nearest hundred” type question 100 is the precision so above 50 and it goes up, below it goes down). Here the concepts of ceiling and floor are introduced, ceiling takes a number and rounds up regardless while floor rounds down regardless. The C function “int” simply ignores the fractional part.

As you can probably imagine calculations using such numbers is a bit more complex than basic adding and subtraction and while most software development kits have this sort of thing inbuilt it is not used unless it is truly necessary.

Other methods of fractional numbers mimic some of the alternative methods of negative numbers. A common method is a fixed point (4 bits “natural number” 12 bits “fractional” is the method used in a lot of the DS 3d hardware). That is to say the computer assumes all numbers after a given binary digit are fractional, this method is used fairly often for timers with fractional parts, a twist on this is a variable length version created as it is needed. What the numbers mean is usually either the logical extension of the binary “powers” (2^-1=0.5 decimal, 2^-2 = 0.25 decimal) or they count again and the number after the point is effectively assumed to be a normal number (binary coded decimal (almost invariably 8421 arrangement) can also appear here).

A final method is more interesting although usually it is only a special case and not much good for general purpose work, representation (in the decimal world) of the value “a third” is usually done 1/3 (as opposed to 0.33333333..... or 0.3 with a dot above the 3) and other times (such as in trigonometry*) a number is “left” and only calculated when the need becomes pressing.

*trigonometry will not be covered in depth here other than a bit of stuff in the 3d hacking chapter but there are functions built into consoles which can be used and other times any trigonometry work is usually done by lookup tables and a method of approximation (linear interpolation for example). It is also one of the special cases (think of things like sin^2(theta) + cos^2(theta)==1).

Curiosities:
The following will also be in the data representation section but it is worth mentioning here.

Quite frequently 7 bits are used as a number with the highest/”most significant” bit being used to signify something, a good example of this is the DS narc file which uses a 1 in the highest bit to signify a subdirectory with the rest being a location within the file (read as though the first bit was a 0).

Numbers are occasionally abstracted a bit, a good example would that of relative pointers where the pointer is the number at an address plus the number of the address itself.

Following on from the “most significant bit” concept above while the byte is the main piece of data sometimes multiple pieces of information (the OAM of the GBA and DS is a good example of this). This means it is a good idea to be able to manipulate binary and convert into hexadecimal (the numbering method used in the memory viewer).

Hacking proper.

As covered in the introduction there are 4 main areas in hacking:

core hacking: how the game works. Attacking takes atk stat adds a random number and takes it from health, sticking 014f in a certain memory block means a certain sprite will appear.....

graphics: for years games have relied on graphics to display different worlds.

Text: This can be a game generated representation of a language and a developer created picture which has text in.

multimedia: This is audio and video.

There is no one hardest area: stats can be simple XML files and text can be compressed and encoded with the actual binary code that is run on the processor(s).

2 things now arise: what does a string of 1 and 0's mean and I know what I want so how do I find it.
There are many methods available and this guide aims to teach some of them and the theory behind it all.

core: there are three components
the binary
extraneous files
file system (not present in the GBA).

Graphics: again there are three components:
2d tiles
2d bitmaps
3d graphics.

text: 3 sections.
tables
pointers
fonts

multimedia: 3 sections again.
audio
video
ingame cutscenes.

Tools of the trade

Hacking involves manipulating files which means it should go without saying (although it will be said) that taking frequent backups of your work is a wise idea.
Computers break and discs/sticks get lost so it is also an idea to get a free email account and send yourself some attachments.

There are many tools of trade although they usually fall into the two main categories of game specific and general purpose.
This section will contain minimal information on usage (although a fairly in depth description will be present on what they are used for) which is to be detailed in the respective sections.
Some of the more specialist tools are left for the respective sections as well.

Game specific tools can be anything from a pointer table calculator to some of the tools aimed at pokemon games that are almost up there with the game creation tools. The former sort are usually created by a hacker/hacking team working on the game and released to save others some time should they decide to take up hacking the game.
The likes of pokemon* hacking tools are aimed at those who do not want to hack the game yet have ideas for new stories/maps and so forth, while they are quite powerful they tend to lack the ability to fine tune things/operate outside the scope of the tool and you will not really learn anything worthwhile about hacking when using them.
That is not to say they are all bad as some truly impressive works have been made using them, just remember using one of the tools (exclusively) will not/does not make you a hacker and as most hacking forums are intended for discussion other than how to use such tools meaning your questions will at best be redirected to somewhere that does deal with them and at worst ignored.
There are several tools for dealing with file formats but owing to the ubiquitous nature of the formats and that fact they do not do much by themselves they are included in the next section.

*pokemon games are usually very hard to hack owing the developers making them fairly hard to hack so you may wish to pick another game to learn with.

Regardless of what tools you use it is strongly suggested you familiarise yourself extremely well with the tool you are using.

Game specific.

Owing to the amount of effort it takes to make high level (“easy” to use) graphical tools as seen in the case of pokemon and the power possessed by modern scripting languages (like python, autoit and even bash/dos with a few helping tools) the former do not tend to appear much these days while the latter are usually distributed in script form.

Still some tools exist:

Golden Sun.
http://sourceforge.net/projects/gstoolkit
http://z9.invisionfree.com/Golden_Sun_Hacking/
Pokemon (almost all versions have some tools)
http://www.romhacking.net/?category=&Platform=10&game=&author=&os=&level=&perpage=30&page=utilities&utilsea

rch=Go&title=&desc=
http://gbatemp.net/index.php?showtopic=94499&hl=
http://gbatemp.net/index.php?showtopic=99737&hl=
Romhacking.net collection of applications:
http://www.romhacking.net/?category=&Platform=10&game=&author=&os=&level=&perpage=30&page=utilities&utilsearch=Go&title=&desc=

Some guides/sites to make things simple using more general purpose tools exist too:
Advance Wars.
http://forums.warsworldnews.com/viewforum.php?f=11
Sonic Advance
http://www.romhacking.net/docs/335/
Metroid Zero mission
http://www.romhacking.net/docs/184/
Final fantasy 4 (GBA)
http://www.romhacking.net/forum/index.php/topic,4449.0.html

Should you look around some of the various translation/hacking projects there is usually some good information available about games and occasionally even tools to help edit.

General purpose.

The two main tools of any hacker are the hex editor and the tile editor but there are many others that can make life easier.
A spreadsheet is useful, open office (http://www.openoffice.org/) provides a spreadsheet program (calc) that also functions well with hexadecimal so it is worth a look.
Those involved in translation hacking will usually find a Japanese text editor a useful tool JWPCE is a freeware tool used by many in the hacking community.
http://www.tanos.co.uk/jlpt/jwpce/

Development of new programs is split across many languages but C,C++, C# and scripting languages (python, autoit, visual basic....) are the most common. They key aspect as you might imagine in manipulation of files with a nod to file conversion (images to/from PNG/some other lossless format, text to/from a common encoding, music to/from a common format) so it is advised you pick a language that deals with it (or at least is able to interface with tools able to do it). 3D files and the programs that deal with them will often use existing libraries (mainly directX and opengl). A language that can readily make a GUI is also an idea.

Hex editors

Anything can be done with a hex editor as any aspect of the files in question can be manipulated but for a lot of things it gets quite difficult to do so a hex editor is usually used during the reverse engineering of file formats, for small tweaks to things and the manipulation of files (copy and paste, insert blank section, file corruption.....).
Being a valuable tool in almost all aspects of computing there are many hex editors, some of the things done in rom hacking are not so common though so editors geared towards rom hacking (but generally lacking in some of the general purpose functions) are available. Most hackers have a collection of editors used for different purposes for this reason.
Hex Workshop
http://www.hexworkshop.com/
A very powerful payware editor although it lacks rom hacking specific features. Most hex editor screenshots in this guide are from it.

Xvi32:
http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm
Simple, effective freeware editor. Again lacks rom hacking specific features.

Winhex
http://www.x-ways.net/winhex/index-m.html
Freeware (a more featured paid version exists), lacks rom hacking specific features but comparable to Hex Workshop.

HxD:
http://mh-nexus.de/en/hxd/
Freeware hex editor, Simple but effective. Lacks rom hacking specific features.

Notepad++.
http://notepad-plus.sourceforge.net/uk/site.htm
Programming oriented text editor, contains a basic hex editor.

Free hex editor neo
http://www.hhdsoftware.com/Products/compare/hex-editor-free.html
General purpose editor (multiple versions including free). Lacks rom hacking features.

Crystaltile2
http://gbatemp.net/index.php?showtopic=81767&hl=
Features a hex editor amongst other things. Rom hacking oriented features like relative search, decompression, custom character sets (table support).

Romhacking.net list of hex editors. Contains links to a whole host of hacking specific emulators.
http://www.romhacking.net/?category=13&Platform=&game=&author=&os=&level=&perpage=50&page=utilities&utilsearch=Go&title=&desc=

Note a lot of the rom hacking oriented hex editors lack support for the bitwise operations and other things a lot of the originally listed editors have. The main draw is table support and other features useful for text hacking.

To cover a few
Translhextion
http://www.romhacking.net/utils/219/
Table support is the main draw along with relative search capabilities.

Windhex32
http://www.romhacking.net/utils/291/
Lots of rom hacking oriented features.

Most hackers have several editors in their toolkit. Usually a powerful general purpose editor and a hacking related editor.

Rom corruption.
The ease by which you can test code out (via emulation/flash cart) and the fact you can easily backtrack on a computer means you can indulge in a bit of “what does this button

do?". Rom corruption is where you alter a section and then test the game out to see what was broken/if the bit you want to edit is broken. Hex editors have the ability to do this but dedicated programs also exist. A sidenote, colours are limited to only 15bits (3 primary colours at 5 bits each or 0 through to E in hexadecimal)) for the GBA and DS so be careful.
http://www.romhacking.net/?category=1&Platform=&game=&author=&os=&level=&perpage=20&page=utilities&utilsearch=Go&title=&desc=

Hex editors feature the likes of bit shifting, bit inversion, logic operations and much more which can be used for corruption with the bonus of them being easily reversible.

Graphics editing

Tile editor.

There are few text based games these days so you will need to edit images at some point, a full discussion on what images are made of is present later. A tile editor makes for easier editing environment than a hex editor so it is vital bit of kit. Quickly scanning over a whole file with a tile editor is also possible and can reveal things that may not be as evident to a quick scan with a hex editor.

GBA

Note there is 4bpp (bits per pixel) and 8bpp tile formats, Some tools only include 4bpp which is OK as not that much 8bpp work is done by the GBA courtesy of CPU and space limitations. For newer systems is it fairly essential.

tiled2002

http://home.arcor.de/minako.aino/TilEd2002/

Crystaltile2

http://gbatemp.net/index.php?showtopic=81767&hl=

NDS

tiled2002 also works for DS owing to the same (as far as this is concerned) graphics hardware. The DS makes extensive use of 8bpp graphics so it really is a must for a tile editor.

Crystaltile2

http://gbatemp.net/index.php?showtopic=81767&hl=

GC/Wii. Use variations on the GBA/NDS and a lot of 3d work so tile editors are not as useful.

http://hitmen.c02.at/files/yagcd/yagcd/chap17.html#sec17

Unknown formats can often require reverse engineering.

Feidian is the tool of choice here

http://feidian.sourceforge.net/

Text hacking

jwpce is a good tool for Japanese text editing.

http://www.tanos.co.uk/jlpt/jwpce/

romjuice is good for dumping text and converting it into a more workable format (converting binary flags to whatever is necessary). Simpler tools exist but often fall flat on more complex projects.

http://www.romhacking.net/utils/234/

Hex editors have replace functions though and knowing 0d0a is the usual method for "start a new line" is good to know.

File format specific.

There exists simple decompression tools for given formats (sometimes game specific) to general tools for a given format to full blown kits (the two most common for the GBA/DS being tahaxan and crystaltile2).
Tahaxan:
http://tahaxan.arcnor.com/
Crystaltile2, originally Chinese language but there are now translations and the application is fairly simple to use.
http://gbatemp.net/index.php?showtopic=135321&hl=
SDAT files, NARC files, NFTR files and several other formats (graphical in nature) are supported.

Sometimes they are distributed as application specific addons (tahaxan has xml modules) or hex workshop formats while others are distributed as scripts (python is popular here although dos/bash script also exist simple programs like file trimmers and readers), sometimes as simple command line programs (NARC files have a widely forked program) called narctool.
Alternative specifications are available at the end of this document.
Warning reverse engineering specifications is an ongoing work and different/bad implementations (or even the use of options not used in the originally reverse engineered files) can make existing tools not work. Notably in the case of pokemon the subdirectories of the NARC files are not named which causes early versions of narctool not to work.

GBA files are binary only have been covered in game specific tools above. More general tools to aid decompression and other things are available from the links above too.

DS.
NFTRedit (font files used in a lot of DS games)
http://gbatemp.net/index.php?showtopic=105060&st=0&start=0
Includes specification.
Alternative specification (Chinese language)
http://www.ndsbbs.com/read.php?tid=146954

NARCtool. Narc is a common file format used to store other files, also comes in compressed format called carc and occasionally seen with the extension arc.
http://members.cox.net/dexter0/DSTools/narctool.shtml
narctool 0.1p (for pokemon roms with no folder/file name)
http://www.pipian.com/stuffforchat/narctool-0.1-p.zip (file format here http://www.pipian.com/ierukana/hacking/ds_narc.html).

SDAT
tool for ripping: NDS sound extractor
http://nintendon.s6.xrea.com/
Second link down on the left or ”ダウンロード"
Mirror (source included):
http://www.4shared.com/file/68276816/8092229e/ndssndext_v04.html

SDAT spec:
http://kiwi.ds.googlepages.com/sdat.html
See also the source code for the various tools linked.

Multiple specs for graphics and sound
http://tahaxan.arcnor.com/index.php?option=com_content&task=section&id=7&Itemid=36

Start of NBSMD:
http://kiwi.ds.googlepages.com/nsbmd.html

Jump Ultimate stars:
http://gbatemp.net/index.php?showtopic=60809
http://deufeufeu.free.fr/wiki/index.php?title=Jump_Ultimate_stars

Gamecube. Multiple formats are available.
http://forum.xentax.com/viewtopic.php?t=2105
http://wiki.xentax.com/index.php?title=Just_Cause_ARC
http://hitmen.c02.at/files/yagcd/yagcd/chap14.html#sec14
http://www.amnoid.de/gc/
http://www.emutalk.net/forumdisplay.php?s=&f=107&page=1&pp=20&sort=lastpost&order=desc&daysprune=-1
http://www.hcs64.com/in_cube.html (a winamp plugin for multiple formats including some GC (and even some wii) formats. Source is available).

Assembly tools.
There are three main tools here.
Strictly speaking assembler is the human readable form of machines language, assembly is the action of turning assembly code into machine code, an assembler is the program used for the process of assembly..
Disassembling is the action of turning machine code into assembly code, a disassembler is the program used for disassembly.
Compiling is the action of turning high level code (like C, java) into assembly (or more commonly machine code).
Decompiling is the turning of machine code (or assembler) into a high level language. It is very rarely done by a machine unless the language allows for it (java and scripting languages like autoit are common targets) owing to it being incredibly difficult (most companies rely on this fact to prevent hacking by anyone who can code), as an aside dynamic recompilation is the technique used in several emulators where instructions are turned directly from code for one system into code for another as opposed to running a mockup of the hardware and running the code through that (conventional emulation), in much the same way decompilation exists for common operating systems, compilers and languages like C# that rely on a system of libraries used by all users of a language to do common task. None exist for any of the systems covered by this guide though.
A decompiler is then a program that decompiles code. Some decompilers act as a halfway house between disassembly and true decompilation and will decompile things that the decompiler can recognise (as complexity increases people turn to premade libraries to do things, decompilers can be made to spot these) and leave the rest as raw hexadecimal or assembly.

In practice assembler and assembly are synonyms along with ASM (to do that will require an ASM hack, I programmed it in ASM). Searching for disassembly will tend to find you guides on how to pull apart a console* meaning disassembler is is good search term.

*going back to the sanitised names part for hacking related terms from the introduction. Disassembly of the console is generally considered part of hacking/modding and will be taken down by the more overzealous anti-piracy types, console repair on the other hand uses exactly the same methods.
Similarly machine repair will not involve assembly language so adding (program)

disassembly specific terms like opcode to the search with disassembly will tend to narrow the results to things that you might want.
Tile (or some other) format editors are also somewhat afflicted by the same problem, tile or graphics viewers on the other hand will usually have some good info/source code that can be twisted to editing purposes (or be straight up editors as well). Not to mention that viewers can usually render something in a far more useful manner for the non hackers on a team than most tile editors.

Assemblers: turns assembly code (text based) to machine code (binary/hexadecimal based). Normally you will only author a small extension to a binary with the assembler, (for example to change a game from displaying each character 8 pixels apart to checking the width and placing each character so there only 2 pixels between each edge of the visible character).
ARM:
http://common-lisp.net/project/armish/
http://labmaster.bios.net.nz/pyga/
http://www.romhacking.net/utils/343/ (ARM7 only)

See also LIARDS by the same author if you plan on doing DS homebrew using assembly:
http://common-lisp.net/project/liards/

For the gc and wii powerpc processors (as well as the GBA and DS processors) the homebrew developers kit known as devkitpro is suggested.
http://www.devkitpro.org/

Disassemblers:
The tools turn machine code (binary/hexadecimal based) to assembly code (text based). Note this is not an exact science so freshly disassembled code may well not be able to simply assemble. There exists a free disassembler for all the consoles but a bit of paid software called ida (interactive disassembler) is commonly used and is geared towards reverse engineering.
There are two main types of disassembler: raw disassemblers and format specific. The former blindly disassembles files while the latter is geared towards a known file type (like exe, elf or DOL in the case of the GC/Wii) or more simply disassembles at a given location (the GBA binary starts after the header with the actual code also being referenced after the I/O routines) or automatically attempts to switch to ARM to THUMB code for the likes of the GBA and DS (there are two instruction sets).

Links to basic assembly code/instruction/opcode lists included as well.

GBA
Emulators feature the option. Nearly all versions of VBA include a basic disassembler.

Crystaltile2
http://gbatemp.net/index.php?showtopic=81767&hl=

gbadsm (java and DOS versions available)
http://www.postbox.javamaster.co.uk/downloads.html

Assembly code.
http://nocash.emubase.de/gbatek.htm#cpuoverview

http://doc.kodewerx.net/documents.html

NDS
ndsdis2 (parses the header: you can feed it a raw rom)
http://hp.vector.co.jp/authors/VA018359/nds/ndshack.html
Some emulators have the option, the most advanced is no$gba paid version (freeware version lacks the debug features of the paid version although some extra programs/techniques can be used to allow cheat making).
IDA plugin:
http://www.openrce.org/downloads/details/56/NDSLDR

Assembly code:
http://nocash.emubase.de/gbatek.htm#cpuoverview
http://doc.kodewerx.net/documents.html

Gamecube
vdappc (windows, mac and linux versions and alterations to those).
IDA modules and more
http://hitmen.c02.at/html/gc_tools.html

Opcodes and the like
http://hitmen.c02.at/html/gc_docs.html

Wii:
Some gamecube tools work (the “broadway” processor is upwards compatible with the gamecubes) but most is done with IDA. Work is ongoing.
http://wiibrew.org/wiki/Broadway/Registers
http://wiibrew.org/wiki/Wii_Hardware

http://www.backerstreet.com/rec/rec.htm#Features
Aimed at general powerpc processors but worth a look.
IDA plugin:
http://www.openrce.org/downloads/details/15/Nintendo_GameCube_DOL_Loader_Plugin_v0.1

Tracing tools:
Invaluable on the GBA, less so for newer systems. These log when an area of memory is read and what command was doing it. This means if you determine some text is in a given memory section (usually by corruption) you can run the emulator again and wait until the text gets written there and trace it back to where it came from (which hopefully is a location in the rom, if not a few extra steps are necessary). This is also a fairly simple (but very powerful) assembly level technique. Simpler versions of these methods are used to make cheats and these can be turned to hacking purposes.
File systems make tracing incredibly difficult to do (all consoles lack a virtual file system at this time), the best technique is usually to try a search within the rom/file in question with data pulled from the memory, with information being compressed and dynamically generated this obviously falls far short of true tracing.
GBA. Usually this is done with a specially made emulator, the most common here being vba-sdl-h but the functionality has been added to other versions of VBA.
http://labmaster.bios.net.nz/vba-sdl-h/

Emulators.
Emulation is invaluable in software (and to some extent hardware) creation and debugging (read rom hacking) as it allows quick checks on changes and the ability to change things in real time. More importantly a lot of emulators feature save states (also known as instant saves, real time saves, snapshots and several other similar names) which can mean a quick way of getting to the required places.
Little in the way of hardware level debugging exists so it will not be covered.

GBA.
The GBA has several emulators although for hacking the list tends to then shrink to PC only emulators.
VBA-sdl-h
http://labmaster.bios.net.nz/vba-sdl-h/

“vanilla” VBA, Not as full featured.
http://vba.ngemu.com/
See also VBAlink (allows emulation of the link cable)
http://vbalink.wz.cz/
And VBA-m (a combination of all of the forks of VBA, work is ongoing)
http://vba-m.ngemu.com/

no$gba. The debugging sections cost money (fairly cheap for hobby users) but it is generally regarded as one of the best tools for the job.
http://nocash.emubase.de/gba.htm

DS:
While emulation may not be at a “fully playable” level the emulators are certainly good for hacking (being slightly slow/juddery in graphics does not matter if you can eventually see what you need).
There are many emulators for the GBA with most featuring some level of debugging.
no$gba
http://nocash.emubase.de/gba.htm
Desmume (limited hacking features)
http://www.desmume.com/
iDeaS
http://www.ideasemu.org/

The GC and Wii lack workable emulators of any kind, none the less some of the ongoing projects for the gamecube:
http://www.gekko-emu.com/forums/viewtopic.php?t=385&highlight=
http://www.nyleveia.com/daco/
http://www.dolphin-emu.com/
http://www.tuxemu.se.nu/

Strictly speaking file/font viewers are also types of emulation but this will be covered in the relevant sections.

Patching tools.

Unlike the other tools this section will have a discussion as it is applicable to all areas and types of hacking.
Developers tend to frown upon distribution of rom/iso images and doing so is not also that

nice on bandwidth for the end user. Patches however can be made that only contain the changed data and so people are free to distribute them (technically it can fall under the derived work aspect of copyright law but it is a relatively unexplored area, for more see the discussion on rom hacking and the law at the end of this document).

The are many ways of making patches however. Methods of making patches and tools aimed at end users (the “complexity” of some patching methods are often a source of frustration for new users unfamiliar with patches) will be detailed.

IPS
Very common on the earlier systems but there is a 16 megabyte* file size limit (there is a hack to allow the patch to start at an offset but it is not widely supported).
The coupled with the fact it is a “dumb” patcher in that it sees changes in the original file and makes a patch accordingly. This is not a problem for systems up to and including the GBA (assuming it is 16 megabytes or under in size) where shifting bytes would mean a large amount of work but newer systems where files can be shifted (which happens often) and also signed/encrypted (looking mainly at the wii) mean a change is seen and consequently encoded.
*actually it is 16 megabytes – the bytes to represent EOF (end of file). Most patchers will also support changes within the first 16 megabytes which led to the use of IPS for some trainers and simple patches.
However it is widely used and supported so it is still worth mentioning.
Patch making.
IPSXP
http://home.arcor.de/minako.aino/ipsXP/

Patch applying
GBATA:
http://www.no-intro.org/tools.htm
http://ezflash.sosuke.com/viewtopic.php?f=8&t=5047&start=0

BSDiff
Used by many of the trainers released by “scene” groups and on several other occasions.
Patching and application
http://sites.inka.de/tesla/f_others.html#bsdiff
See also
http://ezflash.sosuke.com/viewtopic.php?f=8&t=5047&start=0

Xdelta
The main rival to BSDiff. Can achieve similar levels of usability to BSDiff.
http://evanjones.ca/software/xdelta-win32.html (older but still widely used port)
http://code.google.com/p/xdelta/ (google code pages)
Patch applying can be done with:
http://ezflash.sosuke.com/viewtopic.php?f=8&t=5047&start=0

PPF
Playstation patching format. Originally designed for use with the original playstation there were several often mutually incompatible implementations. Today it is mainly used by some hackers of wii isos.
http://www.romhacking.net/?category=2&Platform=&game=&author=&os=&level=&perpage=40&page=utilities&utilsearch=Go&title=&desc=ppf

UPS
Designed as a replacement for IPS and to unite the different formats it has not achieved a large foothold yet.
http://www.romhacking.net/utils/519/

Workaround and custom/patching implementation.
Courtesy of wii isos being signed and no signing key (the existence of the verifcation key and certain bugs are what is used, more under file systems) most patches will patch the decrypted file and get the user to rebuild the ISO thus avoiding the problem with lots of changed bytes. Several patching methods mentioned above are used and it is the main reason for the inclusion of PPF in this document.

Some patches for DS roms use a similar technique although as the file system is not signed the whole process can be done by one program (or more commonly batch file). Here the DS rom is pulled apart and the individual files patched before the rom is rebuilt. This makes smaller patches than even the “smart” patching methods of xdelta and BSDiff as well as allowing different grades of patch to be applied (for example in the case of multiple cheats or for sound hacking where different amounts of change are called for the users of the patch).
Deconstruction of the file system type methods also make it slightly easier to implement multiple hacks from different groups if done correctly.
A few others release patching “programs” based on their own or an existing method.

More patching methods and alternatives to the tools above
http://www.romhacking.net/?category=2&Platform=&game=&author=&os=&level=&perpage=40&page=utilities&utilsearch=Go&title=&desc=

An aside on stacking patches
You will have to understand how things work with the DS (as well as GC and wii) and how patching mechanisms work.
All known commercial DS roms use the nitrorom file system and all use their own file types underneath that. These file types are what makes DS rom hacking easier and harder at the same time (easier as you do not have to worry about space and harder as tracing (a method by which you can determine what does what with extreme accuracy) is nearly impossible.

Patching mechanisms also vary, some of the more modern ones like deufeufeu's patching system and the earlier hacks with batch files and rom ripping rely on the fact the DS file system exists.

Others like IPS, bsdiff and xdelta work simply by having an address (or three) and replacing the information at the address with the data they hold. It gets far more complex in that IPS has a limit of 16 megs unless you use the hack (which hardly anything supports) to change the starting address giving you a window, it is also a dumb patcher in that if you change the position of something position in a file it sees a "change" rather than a shift. Xdelta and bsdiff do not have the size limit and for the most part shifts do not affect them either which is why they are used for the DS.
Another problem comes with the newer patching types that implement hashing into a patch (it will check to make sure the file you are putting is good and the file coming out is also good), you can usually tell it to ignore the hashing though although it may take going to the

command line to do it.

However many hacks work on the internal file system of the files within the DS rom: small patches like the basic sound hacks (tetris, pokemon, Castlevania POR), cracker's early trainers and some the scene trainers only change files within the rom and so should not need to rebuild it or change the location of anything.
If the hack is simple (as in the case of the simple sound hacks) the the length of the file will not change and if the hacker knows what they are doing they will reinsert the file with something like ndsts instead of rebuilding the entire rom. Note you can usually pad smaller files out to match sizes and then insert using this method.
This is not always the case though and some choose the rebuild the rom (which has another set of problems as various rom/flashcart combination do not get along with ndstool which DSLazy and DSbuff use to manipulate the file systems of DS roms), mario kart is the classic example and 99% of the time when the basic undub procedure fails (basic procedure is grab sound files from Japanese rom, replace sound file from US/Eu release with Japanese one and rebuild) it is because of this problem.

If it is a basic patch using something like ndsts or ndshv such as the sound hacks you can usually stack as many of these as you please. Note that just because you can does not mean you should: trainers and sound hacks could well use the same things and interfere with each other (for example changing a sound file to play 1 song and then changing it to play another in another patch will mean you “lose” the effects of the first patch).

There is also another method whereby you extend the size of the rom beyond what it normally is (trimming roms just gets rid of junk space and you can reclaim this junk for your hack). For the most part this sidesteps the problems with ndstool based operations (at the cost of wasted space) and allows you to change file sizes without rebuilding the rom. Aside from some internal testing and that of a few others nobody has yet done this to an actual rom hack released to the "general public" on the DS but it has been seen a few times for internal files, on the GBA there was a problem with earlier scene intros that did just this which meant roms could not be trimmed (a 32 megabyte card could well cost over $100 back when: roms were up to 32 megabytes, average was 8 to 16).

Translation patches and bigger changes may well change the file sizes (pointers are simple and space is near enough unlimited so people tend not to have space constraints like in the older systems) and so the file system will be rebuilt.

Knowing this you can bypass some of the problems or work around it. If for example you want to apply a trainer to a rom and a translation patch apply the trainer first as it should not touch the rest of the rom and it will likely be a dumb patch of some form and when the translation rebuilds the rom you can get the best of both worlds.
On the other hand you can reverse engineer the rom hack and find which files are changed before reassembling a hack of your own.

Adding into this is the patching used by GBA slot cards when running DS roms, often they expect clean roms and so you have to be very careful when patching them usually this patching is done last though and hackers will know this. It also has the effect of changing positions for some things which can make the dumb patching methods (IPS and the like) fail.

Core hacking

The following section deals with hacking the executable code and the implementations of the systems concerned as well as other important aspects of the hardware and finally on data representation.

The binary: if you have made it this far you will have heard of computer languages. Simply put computers blindly follow strings of 1's and 0's but trying to get the hardware turned on to display, shape and colour an image of a box is a hard task and machines can be different so people abstract this and then write compilers and runtime environments for the code. Strictly speaking the term “the binary” refers to the file that contains the code that is run upon a machine but in the “real world” it almost always refers to the actual code that is run upon the processor.
The most fundamental computer language is called assembler/assembly and is simply a more human readable form of binary (granted there are some niceties for some assembly environments regarding memory locations and other such things). As you might imagine it differs from machine to machine and even operating system to operating system.

Implementations in the systems concerned.
Info on these has also been included in the assembly tools section.

The GBA has only one processor (an ARM7TDMI processor) and as mentioned above has only big file. The binary (as in the code run by the system) is relatively easy to track down though and will be detailed later.
http://nocash.emubase.de/gbatek.htm#gbatechnicaldata

The DS has a file system and has two processors (an ARM7 at a higher clockspeed than the GBA and an ARM9 at a higher speed still).
DS homebrew code often makes extensive use of both processors but commercial games almost invariably stick to the ARM9 and the ARM7 is used for simple things that the nitroSDK (the official SDK for the DS) provides. This has the effect that the ARM7 binaries can frequently be swapped between roms without issue (this led to the so called ARM7 fix for the running of roms on flash carts as some copy protection/saving is often delegated to the ARM7). Secondly there is the concept of overlays present for the DS, it is a rather old way of doing things (today dynamic linked libraries (DLL files) and similar are used instead) but here a small section of memory is reserved and a function that is not often needed is then called when it is needed and given that memory section (hence the term overlay). Such things are often only the domain of the ARM9 and there is little to predict what game will use them (big RPGs often have none while small puzzle games can make extensive use of them).
http://nocash.emubase.de/gbatek.htm#dstechnicaldata

The gamecube has a powerpc gekko processor (a custom version of the powerpc750) that has does not have that much in common with the powerpc in the old style apple mac computers. There are also other processors for more specific things
http://hitmen.c02.at/files/yagcd/yagcd/chap2.html#sec2
http://hitmen.c02.at/html/gc.html (navigation options on the left).

The wii has two processors, a main powerpc core that is a tweaked and faster version of the gamecubes (upwards compatible with the gamecube processor and known colloquially by the codename of broadway) and an ARM9 processor embedded within the “Hollywood” graphics unit (not especially well understood at time of writing and known colloquially as the starlet, used a lot to organise the wii hardware (to the extent that the wii was once dubbed gamecube 1.5 and the starlet could reasonably be considered the 0.5).

http://wiibrew.org/wiki/Wii_Hardware
http://wiibrew.org/wiki/Starlet
http://hitmen.c02.at/html/wii_docs.html

extraneous files: you can include stats for your monsters/players/weapons/whatever in the binary or you can do it in files/sections distinct to it. In actuality a large chunk of rom hacking comes down to this but a lot of those sections are common enough across games and even systems.

file system: you can lump all your code, images, multimedia and text into one file and systems up to and including the GBA did this very successfully*.
However as time goes on you will likely want to make an index of files and be able to call upon said index to get files. Most disc based systems (wii and GC in this documents case) do this and so does the DS. Files can have their own file system as well. Common ones include (N)arc, SDAT, NCER, NANR, NCLR, NTFP, NTFT, NTFS and NSBMD
http://tahaxan.arcnor.com/index.php?option=com_content&task=category&sectionid=7&id=28&Itemid=36
http://kiwi.ds.googlepages.com/nsbmd.html

Warning reverse engineering specifications is an ongoing work and different/bad implementations (or even the use of options not used in the originally reverse engineered files) can make existing tools not work. Notably in the case of pokemon the subdirectories of the NARC files are not named which causes early versions of narctool not to work.

*the GBA bundling it all together means the whole GBA rom is technically speaking a binary. However the shift from code for the CPU and data for it to play with is usually fairly noticeable and it will be treated like the other systems.

Introduction to assembly.
Assembly is the subject of many books/modules/education courses and is quite rightly considered a hard subject to pick up, those who are able to work in assembler are generally considered good coders (even if their other coding abilities are lacking) and it truly is as powerful as people say. However while a C/java coder could probably work on any system that can be compiled for with the tools available and a bit of effort assembly is not as portable. The payoff is the fastest code can be made with assembly code (modern compilers are pretty good though and can usually beat a poor bit of assembly) and you can reverse engineer and thus alter code in a profound manner. It is advised you not skip over this section entirely even if you do not plan to do any assembly work as some of the concepts introduced and explained here are important to other areas.

Cheats. Originally a section added for completeness the tools and methods used combined with what can be done and the widespread use of cheats means this makes for a good introduction topic to assembly.
These are not the cheats the developers added by manipulations of the memory that allow you do thing the game otherwise would not allow, modern machines tend to be limited to only the writeable memory but older machines also had so called game genie codes which as far as the game was concerned altered the game pak itself.
This section will give a step by step breakdown of the method used to make cheats. The system of choice will be the GBA but the techniques apply to all systems past and present. An excellent resource for cheating (as far as creating new cheats is concerned) is http://doc.kodewerx.net/index.html , This guide will focus more on the theory behind cheats than actual implementations used for given emulators/carts/consoles.

Some links beforehand
http://doc.kodewerx.net/index.html
http://etk.scener.org/?op=tutorial
http://ezflash.sosuke.com/viewtopic.php?f=3&t=686
http://nocash.emubase.de/gbatek.htm#dscartcheatactionreplayds
http://nocash.emubase.de/gbatek.htm#gbacheatdevices

As an aside cheats can be implemented in many ways. The most complex is hacking them into the rom/iso and running that, assuming you have a method of running them it is probably the simplest method to use once all is said and done.
Trainers are popular implementations of these (some observe a distinction that trainers need a menu or at the very least a method to turn (all) the cheats on and off to be a trainer and not a hack but this is not observed for this guide).
Links to GBA trainers:
http://gba.dellicious.de/trainer.php?s=n&o=asc&d=
http://bubbz.pocketheaven.com/?system=gba&section=patch

Sometimes this is done by hand, sometimes there are game specific tools and sometimes there are general purpose tools (the DS has DSATM and the older program project dipstar, the GBA GBAATM and an older tool called GABSharky).
DSATM:
http://gbatemp.net/index.php?showtopic=80540

GBAATM:
http://www.gbatemp.net/index.php?showtopic=99334

Gabsharky:
http://gathering.tweakers.net/forum/list_messages/942567/26 (click on Steven's cheating guide)

Following this is emulator cheats.
Usually they will implement the most common cheat types (usually commercial cheating systems) and maybe their own (commercial cheating methods have restrictions such as the need to hook a game, emulators are not bound by such restrictions).

Next is cheat devices. They will be split into three categories here.
On cart/on chip.
The addition of support for cheat codes has long been considered a valuable feature for flash carts and the like. Restrictions on cheats that are derived “by piracy” by certain sites as well as the fact carts often face the same restrictions as commercial devices means commercial cheats (or converters to and from them) are the usual methods of storage/distribution.

External debugger
Common in the PS1 era owing to it being a method to run copied games and homebrew it has returned for the likes of the DS in tools like nitrohax (uses a flash cart to enable cheats on commercial games) and to a lesser extent project dipstar (one of the first cheating methods and very hardware specific).
The gamecube and wii have the USB gecko.

Commercial debugger.*

These are the cheating tools you can buy in shops.
Some will feature methods of creating your own cheats (often the most valuable/rare versions), others will have lists of cheats inbuilt.

*note some of these are not cheating tools as will follow but hacked save games or tools that enable you to hack (or better still backup) the save games, others may be more like the tools that will follow but will only have a limited library of cheats and no capability to create new cheats.
Save games are as close as people get to user code and improperly implemented save handling is a very common method to load user code (the xbox has the softmod from this, the wii has the twilight hack, the passme2 for the DS also made use of savegames to an extent).

It is explained in more detail in the “assembly proper” section below but it is assumed that by now you know data is stored as a sequence of (hexadecimal) numbers somewhere in the machine. This guide details methods for figuring out what they are and how cheats can work to implement them. It will also cover the problems that can come when cheating and how to work around them.
While cheating may not be as powerful as hacking there are some cheats that can rival hacks for complexity and what they achieve. Cheating toolkits (usually created by the likes of Datel) are often extremely valuable tools when reverse engineering a new system or trying to get user code to work owing to the fact they are one of the few companies that can and do bypass media protection (very hard to do with end-user hardware).

When testing new things incremental changes are often called for, determining what a given value is often requires the same method.
Key to this is the ability to view the memory and ultimately manipulate it (hence why emulators are so valuable).
The basic principle is as follows
You determine what you want to change in a game (in this case we will go after infinite bullets)
You run the game and get to a point where you have the weapon you want
You take a snapshot of the memory
Being careful to change as little as possible (namely you have no enemies around, you do not move, you do not restore any health.....*) you shoot a bullet.
You take a snapshot of the memory again.
You compare what has changed.
Rinse and repeat until you either find the value/location you want or you have sufficiently few options that you can try them all out.

*sometimes things do change (time for example) so it can be worth attempting to isolate time beforehand by taking snapshots where you have done nothing.

Next comes determining the scale of the value, while some may argue it is a problem and thus needs to be under the problem section below it is sufficiently common to warrant being here.
Going back to the gun example some guns may use 30 bullets to fire a grenade, a whole bunch of energy (say 100 points).
You may have figured out the ammo value in 7 goes however (we are assuming a basic counting system where the number or bullets directly corresponds to the value in the memory) and thus all you would have seen is the last few bits changing.

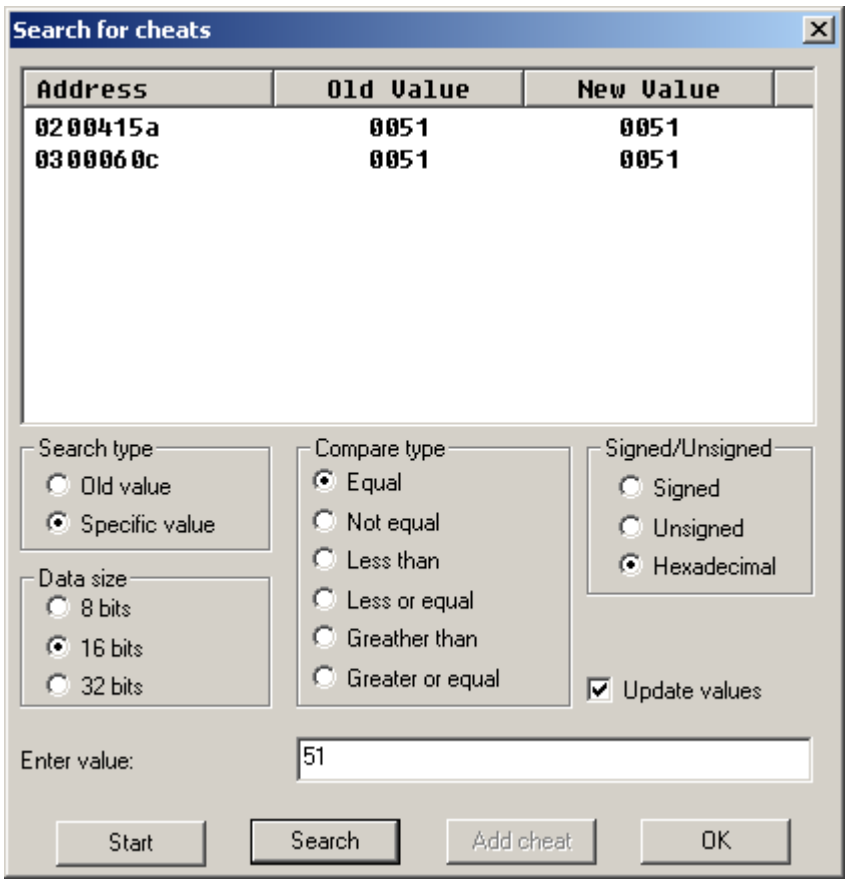
Should you assume only those last few bits matter and hold just them you may end up running out of “infinite” ammo when the larger number gets take from the section that is not held. Game makers themselves have even been known to fall foul of this one when implementing “conventional” cheats.

As an aside it is worth noting that while hardware has memory locations for buttons most games copy this value to another section every so often (usually a v-blank) and then use said copy to do work on as it avoids “sticky” buttons/troublesome reads of the sections and other problems with slow hardware/requiring access using more low level commands. It also means you can make crude turbo fire/button held cheats a bit more easily. Such information is especially valuable when you are dealing with controller addons or trying to hack a game to use them instead. For more on controller hacks:
http://crackerscrap.com/docs/sfchacktut.html

Worked example.
Would be makers of DS cheats can go here for a more specific example:
http://gbatemp.net/index.php?showtopic=116242&st=0&start=0

Summon Night - Swordcraft Story 2 (USA) for the GBA.
Infinite HP is the aim.
In a battle.

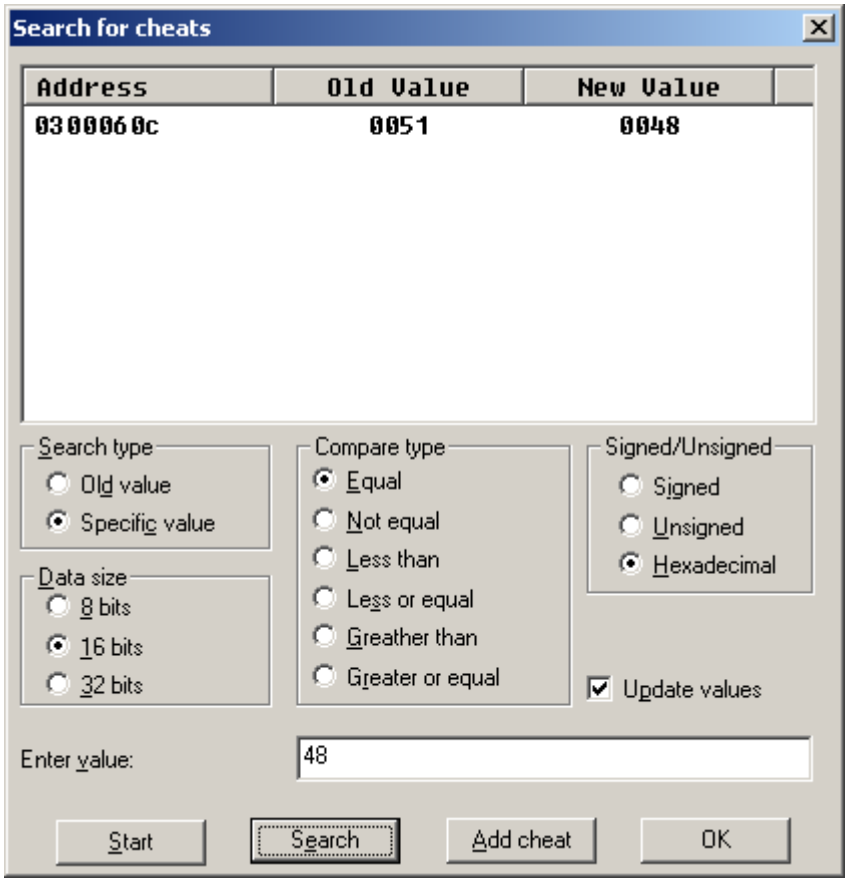
81 is the starting value.
81 in hexadecimal is 51. 8 bit means 256 unsigned and from knowledge of the game the health can go far beyond that so a 16 bit value is chosen.



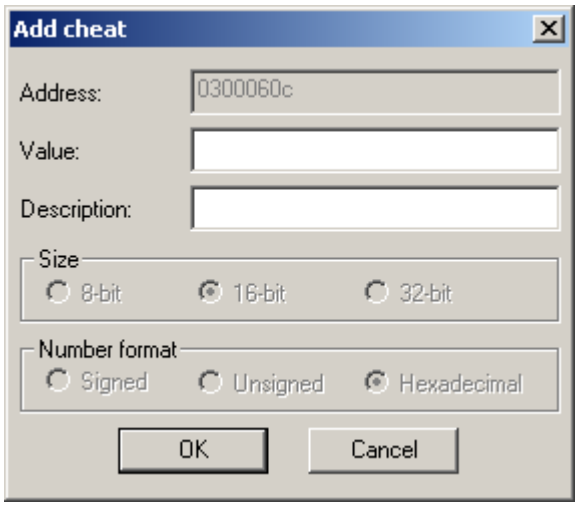
This is enough to start experimenting but a further iteration will be done to confirm. You can also use the other compare types on other occasions.

Now switch to the game and lose some health.

72 is 48 is hexadecimal. So the search of the previous values for numbers that are now 48 is conducted



Highlight the value and click add cheat.



Here the value 90 is decimal is wanted. This is 5A, this is added in the Value box and so as to remember what it is a description is also made (infinite HP). Click OK and go back to the game. After a change it resets to 90 and further attempts to change it do not work.

More complex uses of cheats.
Games are not books or films, you play them and cause changes. These changes need to be recorded and the memory is as good a place as any to do so. Cheats alter the memory.....
Should a game give you a double jump of sorts it will probably record whether you have done a jump somewhere (for the sake of the example it will be a simple 1 or 0 with 1 for “second jump done” and 0 for the opposite)
Should you hold the value at 0 (no second jump done) it should then be possible to fly into the sky.
Moving on a string will probably be in the memory to record what is in your inventory
Likewise a string may exist for what is available in a shop.
Your attack value may be a sum of the weapon, your character, the given conditions....
This will be stored in the ram.
You generally can not pass through walls. The location of the walls need not be determined by the graphical systems but by a table of info that happens to correspond to the graphics.
Further still is the fact that pointers to locations of data can be in the ram, this can mean you can load different sprites, backgrounds, text. This method is often used to uncensor games which merely have the offending pixels moved elsewhere in the file.

One of the more interesting types of hack involves the AI of a game, true artificial intelligence is a very hard thing to make (so far nobody has done it) but approximations can be used to achieve good results in the limited environments that exist in games. Often the easier levels of the AI will be held back or prevented from doing certain things, cheats can then unlock the AI (or indeed lock it down), like most other areas of hacking there is no magic bullet for this especially as every game is different and so every AI is too (usually, sequels and games from the same company may reuse things).

The main problem facing the creation of such cheats is that it is quite difficult to change the values enough to get a good reading and often it is hard to keep other things the same while doing it (you may have to open menus and do a whole host of other things to change it). Disassembly is good here and savestates can help minimise the changes done.
Other problems include that the harder hacks involve a lot of changes and with the tools geared towards simple codes (often only 16 bits as a maximum per code) a lot of codes may be needed (which some carts/systems can not handle).

Problems with cheating.
Developers will sometimes make cheating harder for people.

Common methods include.
Mirrored values.
Here a variable for life (or indeed the variable for life) will be written several times across the memory. Sometimes these mirrored values are dummy values and holding them will even break the game (holding the number for a life bar may not hold the life and prevent the use of things that increase the life bar).
You have to find all the values and operate on them all.

Obscured values.
Cheaters often expect the values to represent what is there, namely 15 bullets will mean an “F” hexadecimal somewhere in the ram. Many methods can be used to obscure this. Such methods include
Addition of a value, simple but serves to make searching directly for numbers quite hard.
Subtraction of a value or from a value. A bit more complex but the same principle as above.
Not technically a method of obscuring a value but sometimes floating numbers are used even if conventional numbers will do,
Bitwise operations on the value. Inversing the bits is a simple but effective way, shifting the bits so an 0F00 (0000 1111 0000 0000) reads (0000 0011 1100 0000) or 03C0, XORing a string against another one is extremely effective (using XOR like this is actually a form of encryption that is quite effective and cheap as far as resources are concerned). Encryption has the problem of consuming resources so such methods are not usually used for rapidly changing values but for things like money it is often used.
Here you have to find the operation performed and mimic it in your cheats (which is why cheats often hold a value rather than anything more elaborate. Usually it will involve searching for differences rather than exact values. As an aside shifts and XOR (repeated very often) are often used as very crude forms of encryption.

Game checks. Time passes and there is little that can be done to stop it. A game knows this so it may read a value for time and then read it again in a few cycles, assuming the game does not have a “freeze time” parameter (which the developers would know) then the time should not be the same.

Dummy values.
They have been mentioned already but timers (and health bars) seen on a screen may be nothing (or have very little) to do with the time being used in a game. Testing is then necessary.

Values can be both mirrored and obscured too. Sometimes only the mirror is obscured to make life even more difficult.

Multiple points of entry.
As you see above mirrored values can be used. Here multiple entries can be used (that is to say every given number of clock pulses the value is moved or a different location is used).

Pointers. These will pop up again and again in hacking but these are values that hold the location of the data you need, such things are often used for multi level games so finding a cheat may only work for a handful of levels. You will need to find the pointer and then change that to do what you need or change all the values. Beware pointers are used for more than just annoying would be cheaters and setting values outside what the game expects can have interesting results.

Multiple use values.
It was already mentioned elsewhere (and it will be returned to again and again) but a given by may also be used for more than one purpose. This makes determining what section used or what a value means is difficult. This is quite taxing on the CPU though so it is not often used. Setting a value that does not work or overwrites the other piece of data in the section can lead to a game not working. Testing or disassembly of the whole program is the only way to get around this.
Returning to dummy values for a moment they can be combined with this to form things like “every 100 cycles copy actual timer to screen timer” and then have two distinct processes dealing with timers.

Broken gameplay.
Here something may have to happen to allow the game to continue, for RPGs this could be the typical impossible boss (which will then either join your team/side or be beaten by you an hour later after you train/see a master). Infinite health will prevent this.
Similarly time may need to be lower than the maximum/initial value or a calculation based upon it will fail/return a bogus result.
Falling down a hole may have a routine that happens when you do, preventing this may mean you are stuck down the hole. Likewise a “moon jump” cheat may send you out of bounds which owing to the fact it should not happen in normal play.....

These are usually avoided by setting appropriate values, turning the cheats off (you may have to make special provisions for this), changing something to refill a health bar as opposed to providing infinite health.

Hashing.
It is covered in cryptography and is mainly a problem with savegames but here a value is taken for the data in question and then a number generated from that value. In simplistic terms this can be odd and even, in practice sums of the bytes in question and a whole host of other methods can be used.

http://etk.scener.org/?op=tutorial has a worked example dealing with encrypted money on the GBA.

Assembly proper.

Layout of a machine.
The following is a grossly simplified layout (architecture) of a computer. The are several variants but all computers will have some memory, some method to output data, some method to input data, a computing device (read processor) to perform operations on the

data and usually some hardware to tie it all together.
Most modern machines will have several processing devices, each tuned to do a slightly different thing (graphics, underlying processes, maths (although now such things are usually onboard the actual CPU) and audio are common things for extra processing devices to do).
It is sometimes assumed that as the main processor has the ability to access/manipulate most areas of the system it is then responsible for all manipulations of the memory. This is not true, quite in fact there are usually areas the processor can not reach (usually done for security, simplification or stability purposes and consequently the ability to manipulate these areas is one of the main aims of hacking any new system).

CPU overview.
The following is a simplified layout of modern CPUs. It is aimed at those unfamiliar with such things and as such contains minimal useful data on the CPUs used by the consoles covered in this guide.

Registers.
Each processor will feature a series of small memory sections of extremely fast memory called registers. These can be different sizes and be general purpose or have a specific purpose. When talking about the “bit” of a system (nintendo 64 (bit), megadrive 16 bit) the maximum number of bits an CPU can have in a register is what gives this number.
Often for marketing purposes some companies use other things like graphics chip and there is a distinction if a processor contains one or two large registers with the majority being smaller ones.

Instructions.
Detailed in a following section but these are the operations a processor can do to to data. Often a processor will have more abstract commands in addition to the basic ones to aid in the programming process.
Broadly speaking there are 4 stages
fetch. Before the instructions are done they are stored in a stack, getting them from the stack is naturally the first thing to do.
Decode. The instruction will contain the operation to be performed, where the data is located that needs to have the operation performed and where it needs to go after it is done.
Execute. The operation is done to the data in question.
Writeback. The data is written to where it needs to be for whatever needs to happen next.

Most processors are so called serial processors, that is to say they process one instruction after the other. Nowadays there are multiple processors in machines and so called hyperthreading.
A thread is a single list of commands that will be performed one after the other. This need not be a single aspect of a program, quite in fact most programs ever written will only have a single thread for the entire program.
Multiple thread programs (usually abbreviated multithreaded programs) are programs that have 2 or more threads for a given task. For example you could have one thread to take care of the GUI and another to take care of the underlying processes.
Multiple processors (or multiple cores per processor) can mean each thread can be run upon a different processor/core while hyperthreading is an attempt to emulate multiple cores by picking and choosing what thread to run at a given time. As you might imagine this is quite difficult to pull off effectively and coupled the fact it is a fairly recent development in mainstream computing such things are not used as much as they might

be.

The clock.
A clock in electronics terms is a pulse generator, the speed of the pulses (pulses per second measured in Hertz (Hz)) is then called the clockspeed. A lot of machines have one clock for the entire system (even if the pulses are subsequently reduced or increased in frequency). At each pulse of the clock part of the instruction is performed.
If you read on about this RISC or reduced instruction set computing (read “reduced instruction” set computing not reduced “instruction set” computing) is where these instructions are limited to only a few clock cycles whereas CISC (Complex Instruction Set Computer) tends to favour a larger amount of clock cycles with the payoff of more work being theoretically able to be achieved.
A very crude example follows: if you can only add one number to another (RISC computing) it may be quick to do and simple to store but if you can add a set of numbers to another using a few tricks (CISC computing) even if it takes a bit longer to do and more to store far more can be achieved for a given instruction. This is a crude example as RISC processors can have complex instructions and vice versa, not to mention the lines have been well blurred over the years. Importantly it has nothing to do with the amount of instructions available but rather the ethos behind the instructions. The ARM processors (ARM stands for Advanced RISC machines) and powerPC processors favoured by the consoles covered in this guide are generally considered RISC processors while the (presumably X86 or x64 based) machine the hacking work takes place on is generally considered a CISC processor.

Increasing the clockspeed is known as overclocking and has advantages (instructions get performed faster) and disadvantages (increased heat leads to lower stability and equipment lifetime as well as more power usage). It is a fairly popular technique used in high end PCs so you are encouraged to look to guides to overclocking should you want more information on this.
Consoles are occasionally overclocked but it is outside the scope of this section (it can make for better hacks* as more data can be processed and more data can mean more detail, more enemies on the screen, larger levels.....). As an aside the speed hacks usually do not overclock a processor but remove limiters put in place or flood the CPU with extra instructions/loops which slow the processing of the actual data (and thus the game).
*Emulators tend to (read always) require faster computers than the consoles them emulate, as computers get faster than is required to emulate the extra speed can also be used to surpass the hardware limitations (see high res textures for N64 games: http://www.racketboy.com/retro/nintendo/n64/2008/03/enhance-n64-graphics-with-emulation-plugins-texture-packs.html).

Memory overview.
This section deals with RAM or system memory, long term memory (rom images/disc isos) is mentioned but not covered in detail.
The CPU will perhaps have a few tens of registers and a bit of cache all with but a few bits to their name meaning they are no good for storage.
Registers being fast memory makes them expensive so sections of slower memory (but with much larger capacity) is used as the main memory (usually referred to as RAM even if the use of the technical term is not as accurate as it might be).
Modern computers are often tasked with a lot to do all at once (all on top of an operating system) and computer languages tend to abstract things at the cost of memory space, memory is cheap though so people do not tend to mind quite so much.
Consoles on the other hand (up until the likes of the latest generation: PS3, xbox 360 and

to some extent the wii which tend to run the “dashboard”/menu in the background) only run one program at a time (your game) so the memory requirements could be considerably less than your PC building/buying days may lead you to think is viable.
Although not entirely relevant to the consoles in question the following site has some nice electrical diagrams dealing with ram and some of the other aspects of computer hardware: http://www.comp.nus.edu.sg/~bleong/6.371_project/main_datapath.html
Addressing the memory (having some data is not much good if you can not look it up) is dealt with in the next section but the memory is usually given an address where it can be found (think page in a book, be careful though as page is also a technical term dealing with multiple memory sections to dodge issues with large numbers to address ram).

Long term storage is what is used to hold the games (or it should be anyway) as ram loses the data it holds if it loses power (there many types (invariably slower) of memory that do not and the developments in the memristors http://www.theregister.co.uk/2008/05/01/hp_labs_unveils_memsistor/ may change that, those however do not really apply so much to hacking other than in some aspects of saving of games). These are covered in the file systems section as they can not be written to, information on accessing these sections should be contained within the technical documents or if necessary in the same section.

Other hardware.
Dealing with this is not only what makes computing possible but also one of the main aspects that makes assembly programming as difficult as it is (the others being assembly has little to nothing in the way of checks to stop you doing something silly and the fact you are the one tasked with defining sections/formats and remembering what they are used for).
Each piece of “other hardware” will usually have some area of memory/line of communication available to them (sometimes able to be changed/tweaked but not always), accessing them in assembly programming is usually done by so called I/O routines and is one of the reasons assembly programming is quite hard to do well (I/O routines are occasionally gifted to programmers (see the ARM7 binary for the DS commercial games and the likes of HLA for the PC) and can be a useful target when trying to track down information.

Introduction to assembly programming.
The two most important commands are generally considered push and pop. As the processor only has a limited number or registers trying to accomplish too much at once will just end up causing a hang. Knowing this the processors have the option temporarily free a register by “push”ing the register to a stack and later “pop”ping it back in. As far as rom hacking is concerned and with some of the more interesting aspects of processors http://www.geek.com/images/geeknews/2006Jan/core_duo_errata__2006_01_21__full.gif there is also nop (no-operation) that will do nothing for one cycle, it is used to either slow down a program (by putting lots of nop commands in) or as it is as long as any other instruction it can be used to skip over commands you do not want (for instance taking lives if making a trainer: “if [death] then sub lives” becomes “if [death] then nop”).

Next there are the navigation (j(u)mp and processor manipulation (the command mov (common across most processors) copies data from one place to another (see explanation on command information sources below), notably it does not destroy the original as mov(e) might imply) while the ARM processors can shift mode to a 16 bit mode (with 32 bit data processing capabilities) called THUMB).

After this there is basic arithmetic, add, sub(tract), multiply and divide*

After this is the comparing features (usually jump if higher, if equal or if lower). While not the same as the archetypal IF command of many languages they are close enough.

Most operations above this provide simpler methods of doing common or more complex tasks (multimedia was the primary reason behind the introduction of MMX/3d now instructions for the PC and one of the reasons why multimedia is so hard to do on the lower powered systems).

*division is quite hard so it is not usually used as often as it might be when compared to general maths.

Commands have 4 main types/sources of information.
Register: “take value of a1 and add it to the current register”
memory: “take value of [memory address] and add it to the current register”
immediate “take value from this instruction and add it to the given register”
implied “instruction adds a given value that no programmer can influence”

An equally important concept to the processor commands is the concept of interrupts. An interrupt as the name implies is where a process is “stopped” so as to enable something else to happen.
There are many types of interrupts (often with a different priority over each other and/or requiring special conditions) but broadly there are software and hardware versions. Crucially you can set up interrupts to do things that would otherwise take space, power (electrical and resource) and extra effort to do, in the example of games rather than check every few operations if your character has died you can instead set up a method whereby dying triggers and interrupt and as a consequence a given routine for dying. As an aside games have often been known to have multiple ways of “dying” (loss of health, falling in a hole, drinking a poison potion, running out of time.....) so you may need to find all of them if you plan on preventing death.

Direct memory access (DMA).
As a continuation to the “processor is not all powerful” section above it is important to note that you do not have to run all data through the processor, DMA or direct memory access allows you to transfer files/data directly to the RAM from other sections and similarly hardware will often have write access independent of the CPU (and often only readable by the CPU), an example can be found in the case of the DS touch screen:
http://nocash.emubase.de/gbatek.htm#dsserialperipheralinterfacebusspi

For GBA DMA see
http://nocash.emubase.de/gbatek.htm#gbadmatransfers

A couple of example hacks will be included but a conventional guide to assembly is suggested instead. This will hopefully decrease the size of the document as well as keep it more current in the case of the systems that continue to be reverse engineered.
General/x86 based assembly
http://webster.cs.ucr.edu/AoA/index.html
http://burks.brighton.ac.uk/burks/language/asm/#tutorial
http://www.drpaulcarter.com/pcasm/

GBA assembly programming (quite nice to begin with worth a look if you plan on assembly

programming on the DS (and to a lesser extent Wii). Contains useful links to ARM assembly.
http://www.coranac.com/tonc/text/asm.htm
http://patater.com/gbaguy/gbaasm.htm
http://gbadev.org/docs.php
http://nocash.emubase.de/gbatek.htm

note that unlike some earlier systems the GBA has a lot of conditional flags (instructions will only perform (or perform as expected) if a condition (usually a specialist register/memory location) has been met). If you are coming from x86 the ARM processors and to a lesser extent the hardware are almost all in “real” mode as opposed to protected.

DS
http://gbadev.org/docs.php
http://nocash.emubase.de/gbatek.htm
http://patater.com (a general DS programming guide but worth a read).
https://oopsilon.com/The-Smallest-NDS-File (not a guide per se but worth a read if you are new to assembly on the DS)

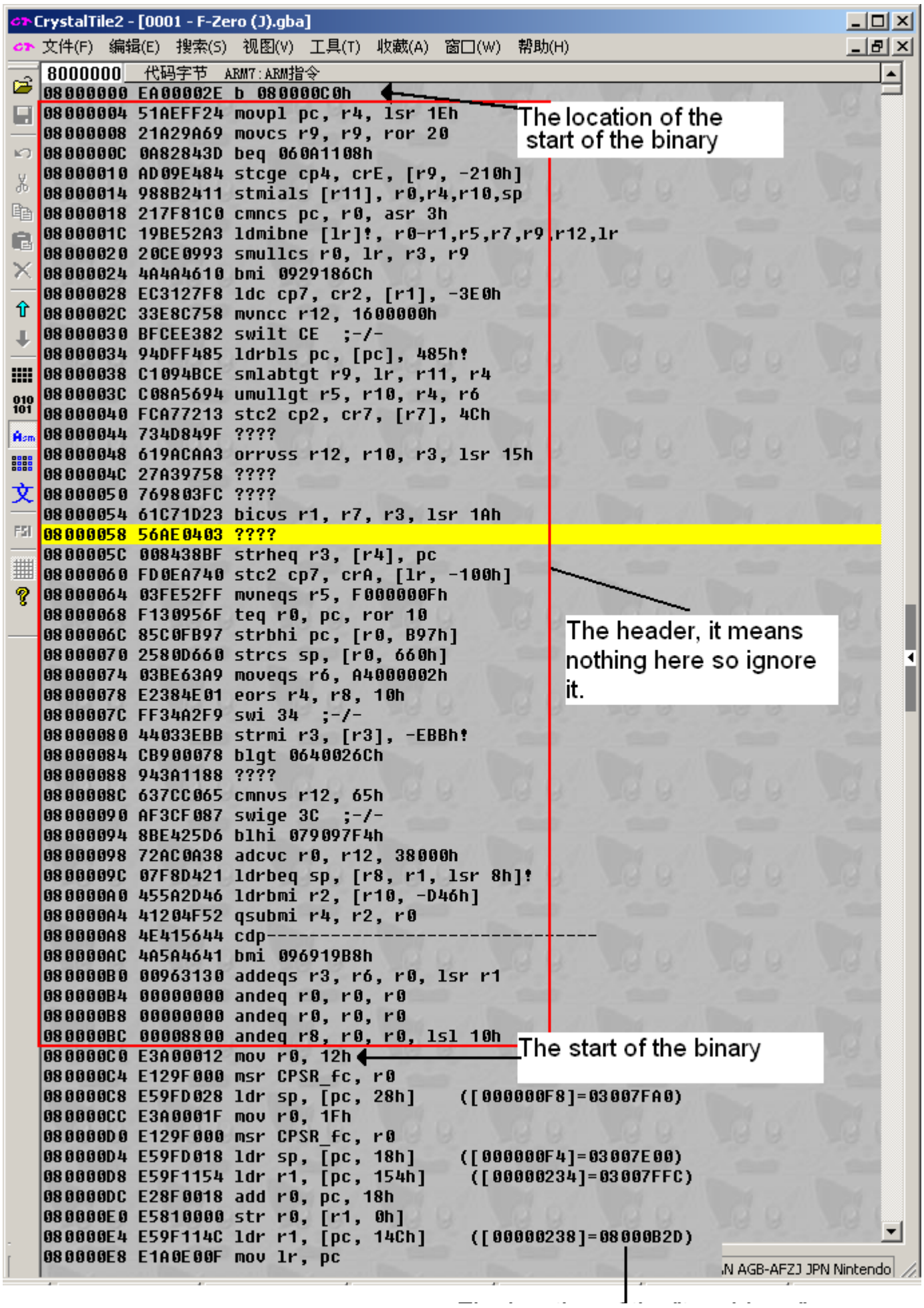
GC/Wii
Little in the way of gamecube specific stuff exists so it is probably worth a look at some of the general purpose PPC stuff as the documentation for the devkitppc (part of devkitarm)
http://www.lightsoft.co.uk/Fantasm/Beginners/begin1.html
http://timestocome.com/wordpress2/assembly-on-the-ppc (a nice collection of links)
http://www.devkitpro.org/downloads/ (devkitpro)
http://freelink.org/gcdev

Hacks and techniques.

The following section details a few common hacks of varying difficulty and discusses the implementations of various pieces of hardware used by the systems.

GBA binary location.
GBA is slightly more complex than the other systems (which tend to use little more than well known pointers) but it goes as follows.
The very first instruction is where the binary starts, more often than not it will be around 0C (i.e. the end of the header).
Example from F-Zero (J) (the very first GBA rom to be properly dumped):

A picture is below that details an example, the following paragraph however details the general case.
The first 4 bytes are an instruction detailing where the binary starts, all valid GBA roms have this. As a disassembler is generally a “dumb” tool it converts the following section into assembly language as well, it is however entirely useless for this purpose so ignore it.
At the start of the binary there will be the so called I/O routines (stack location, memory definitions and other such stuff), they will not change much from rom to rom either as it is provided by the tools the developers will use.
The next number you see that “points” to 08?????? (i.e. in the ROM memory area) is the developers binary, if you find a +1 there or the pointer is out by 1 (some disassemblers parse the code slightly incorrectly) you will need to switch to THUMB mode.
Note this is a good example of the relative addressing used by most assembly (after you assemble it in the first place), here the location is set to 08000000 (the start of the GBA rom in the memory)


CrystalTile2 - [0001 - F-Zero (J).gba]
文件(F) 编辑(E) 搜索(S) 视图(V) 工具(T) 收藏(A) 窗口(W) 帮助(H)
8000000 代码字节 ARM7:ARM指令
08000000 EA00002E b 080000C0h
08000004 51AEFF24 movpl pc, r4, lsr 1Eh
08000008 21A29A69 movcs r9, r9, ror 20
0800000C 0A82843D beq 060A1108h
08000010 AD09E484 stcge cp4, crE, [r9, -210h]
08000014 988B2411 stmials [r11], r0,r4,r10,sp
08000018 217F81C0 cmncs pc, r0, asr 3h
0800001C 19BE52A3 ldmibne [lr]!, r0-r1,r5,r7,r9,r12,lr
08000020 20CE0993 smullcs r0, lr, r3, r9
08000024 4A4A4610 bmi 0929186Ch
08000028 EC3127F8 ldc cp7, cr2, [r1], -3E0h
0800002C 33E8C758 mvncc r12, 1600000h
08000030 BFCEE382 swilt CE ;-/-
08000034 94DFF485 ldrbls pc, [pc], 485h!
08000038 C1094BCE smlabtgt r9, lr, r11, r4
0800003C C08A5694 umullgt r5, r10, r4, r6
08000040 FCA77213 stc2 cp2, cr7, [r7], 4Ch
08000044 734D849F ????
08000048 619ACAA3 orrvss r12, r10, r3, lsr 15h
0800004C 27A39758 ????
08000050 769803FC ????
08000054 61C71D23 bicvs r1, r7, r3, lsr 1Ah
08000058 56AE0403 ????
0800005C 008438BF strheq r3, [r4], pc
08000060 FD0EA740 stc2 cp7, crA, [lr, -100h]
08000064 03FE52FF mvneqs r5, F000000Fh
08000068 F130956F teq r0, pc, ror 10
0800006C 85C0FB97 strbhi pc, [r0, B97h]
08000070 2580D660 strcs sp, [r0, 660h]
08000074 03BE63A9 moveqs r6, A4000002h
08000078 E2384E01 eors r4, r8, 10h
0800007C FF34A2F9 swi 34 ;-/-
08000080 44033EBB strmi r3, [r3], -EBBh!
08000084 CB900078 blgt 0640026Ch
08000088 943A1188 ????
0800008C 637CC065 cmnvs r12, 65h
08000090 AF3CF087 swige 3C ;-/-
08000094 8BE425D6 blhi 079097F4h
08000098 72AC0A38 adcvc r0, r12, 38000h
0800009C 07F8D421 ldrbeq sp, [r8, r1, lsr 8h]!
080000A0 455A2D46 ldrbmi r2, [r10, -D46h]
080000A4 41204F52 qsubmi r4, r2, r0
080000A8 4E415644 cdp----------
080000AC 4A5A4641 bmi 096919B8h
080000B0 00963130 addeqs r3, r6, r0, lsr r1
080000B4 00000000 andeq r0, r0, r0
080000B8 00000000 andeq r0, r0, r0
080000BC 00008800 andeq r8, r0, r0, lsl 10h
080000C0 E3A00012 mov r0, 12h
080000C4 E129F000 msr CPSR_fc, r0
080000C8 E59FD028 ldr sp, [pc, 28h] ([000000F8]=03007FA0)
080000CC E3A0001F mov r0, 1Fh
080000D0 E129F000 msr CPSR_fc, r0
080000D4 E59FD018 ldr sp, [pc, 18h] ([000000F4]=03007E00)
080000D8 E59F1154 ldr r1, [pc, 154h] ([00000234]=03007FFC)
080000DC E28F0018 add r0, pc, 18h
080000E0 E5810000 str r0, [r1, 0h]
080000E4 E59F114C ldr r1, [pc, 14Ch] ([00000238]=08000B2D)
080000E8 E1A0E00F mov lr, pc
The location of the start of the binary
The header, it means nothing here so ignore it.
The start of the binary
N AGB-AFZJ JPN Nintendo

Startup hacking and Cheat to ASM hack
Following on from the GBA binary location and because it is most common on the GBA is startup hacking. Here you can alter the place the rom starts at to do various things. These things can include

padding the memory with something. Not so useful when emulators exist but the idea goes if you pad the memory with something other than the initial values (usually 00 or FF) you can determine what is free to be used for whatever purposes. If you read elsewhere this padding will tend to be “DEADBEEF” or “DEADFACE” or some other word that can be spelled with hexadecimal letters.

Making an intro/hack selection screen. Here you can jump the game to another point (the first intros often used the junk section at the end of the rom which did not go over well in some circles as it prevented trimming (removal of the junk section) of the roms; when you only have 32 megabytes to use and junk can amount of a megabyte or more per rom...) Note there are far more complex methods of adding an intro to a game (many of which were used to prevent the removal of intros).

The cheating section above gave you a method of determining what does what. This method (although surpassed and implemented in other tools like DSATM) is quite a useful thought exercise. It also provides a basis for other assembly hacks.

Note the value used need not be the same as the “scale of the value” as mentioned in the cheating section (while the value may only be 8 bit the instruction/available space may be 16 and be better for it).

Here we already have a cheat with the memory location and the format used so instead of having a cheat which may not be a viable option on real hardware/given emulators it will be hardcoded into the game.
The general idea is to find the instructions that deal with the memory section in question and alter them (rather than deducting a value it can be made to add a number or perhaps more wisely store a value/deduct 0. This also provides the basis for rebalancing the game and more advanced hacks.

Set up a tracing session with the ram section being the target. Halt on write to allow the active instructions to be analysed.
Another method is to disassemble the program and look for an instruction that deals with the memory location in question but such things can become a bit more complex depending on what is going on with the game.
As mentioned tools like DSATM have surpassed the specifics of this method but it is worth knowing:
http://gbatemp.net/index.php?showtopic=44410&st=0&start=0

They methods have been surpassed as they rely on altering the binary by way of removing/rewriting a “junk” section or some other function, for cheating purposes it is also considered better to “hook” a game (by setting up an interrupt for example) as opposed to rewriting a function like this.
A better method is available from the documentation section of
http://crackerscrap.com/docs/dshooking.html

Tracing

An incredibly powerful technique and one that this guide suggests to all hackers even if assembly is not something you plan on using.

These log when an area of memory is read and what command was doing it (you can also expand this to halt the emulation when writing it, halt on last write, halt when a given section of data appears....), you can also use the logs with a lot of compression tools to make life easier when you have compression (mainly limited to BIOS decompression methods although you can expand the halt on break method and work back through the assembly of the decompression method).

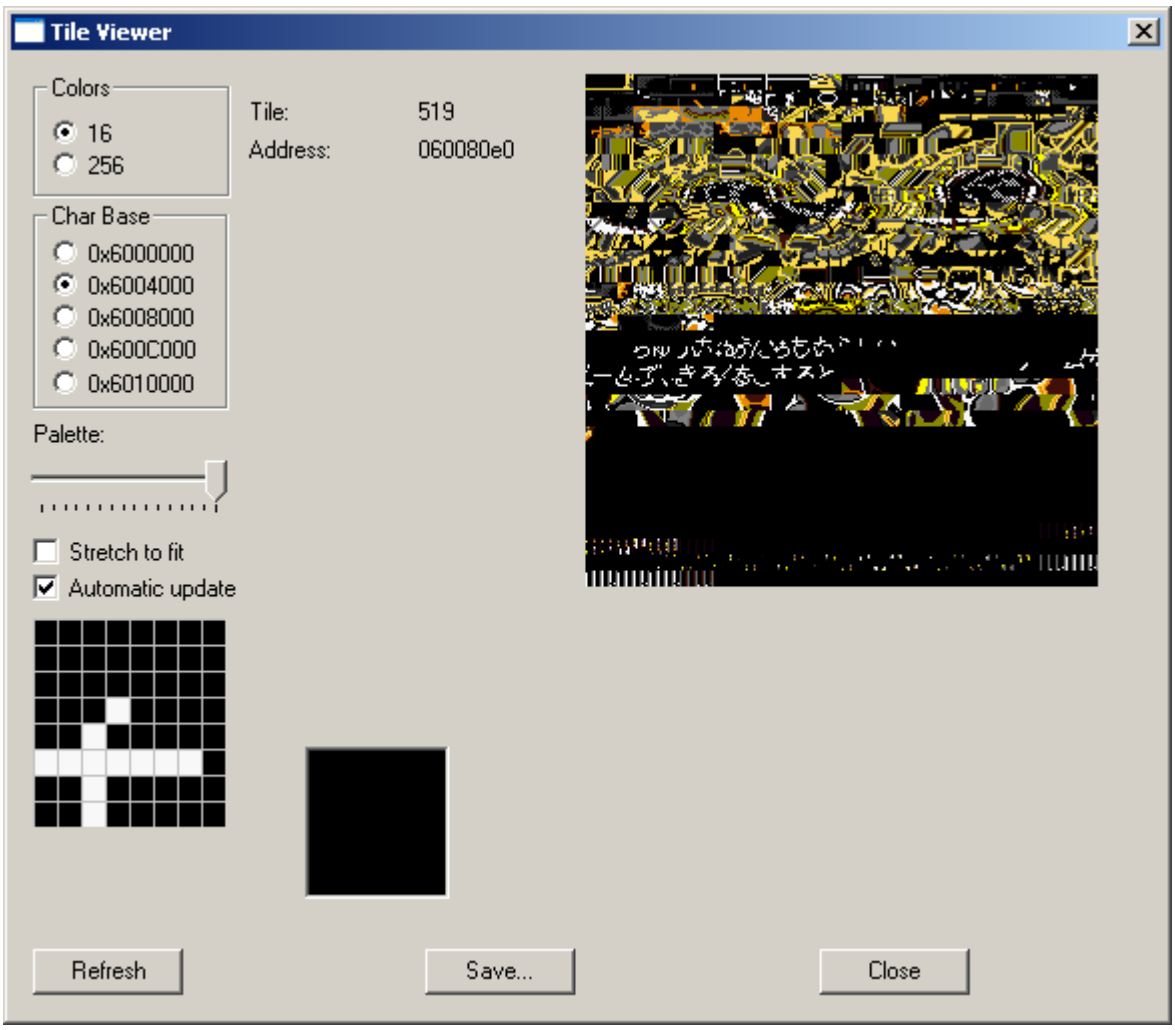
This means if you determine some text is in a given memory section (usually by corruption) you can run the emulator again and wait until the text gets written there and trace it back to where it came from (which hopefully is a location in the rom, if not a few extra steps are necessary). This is also a fairly simple (but very powerful) assembly level technique.

Simpler versions of these methods are used to make cheats and these can be turned to hacking purposes.
File systems make tracing incredibly difficult to do (all consoles lack a virtual file system at this time) owing the slightly differing read methods and things like copying/parsing a header from a file into the ram and then using that with subsequent calls. The best technique is to try a search within the rom/file in question with data pulled from the memory, with information being compressed and dynamically generated this obviously falls far short of true tracing*.
GBA. Usually this is done with a specially made emulator, the most common here being vba-sdl-h but the functionality has been added to other versions of VBA.
http://labmaster.bios.net.nz/vba-sdl-h/

*all is not lost in these cases, often things will be decompressed and manipulated beforehand in the memory so you can go further back and find compressed/original forms of what is in the ROM).

As usual an alternative guide to to tracing on the GBA (from Labmaster, the creator of the vba-sdl-h emulator)
http://www.romhacking.net/docs/361/

Tracing example
Aim of this example: Find the font on Magi Nation (Japanese release) on the GBA.
This game features an English translation (in ASCII) although you need to hack the game to get it. Although this hack is not covered here this would be one of the first steps to translating the game.
A normal emulator gives the location of the text at 060080C0 (it is suggested you scroll through a couple of lines to make sure it is not text moved to the middle of the screen for effect) and using the autoupdate option you see it gets written there as the text appears on screen. Highlighted text is done by changing the VRAM as opposed to the palette.



It is found in the BG0 layer (see video hardware below) if you are interested in that.

This 060080E0 is the target for the tracing emulator (the tile starting at 060080C0 has minimal data and while it would work this is a good time to have large changes for the sake of example).
As the section gets repeatedly written to the font (or something that spawns it) is sure to be among the source locations for instructions that write to that area.
Running VBA-SDL-h we get to the point where a new game can be started (before we walk through the door though). From experience we know the section is blank now.
Pressing F11 (to enter debug mode)
“bpw 060080E0 32” (no quotes)
this means the game will halt when any of the 32 bytes (we know is it 4bpp which means 32 bytes per tile) following 060080E0 are written to.
The command “c” allows the game to continue.
Upon walking through the door the game is halted and the command prompt appears again. The entire area has been filled with 00's (part of the title screen was still left in the ram) so we have to press c to continue with the tracing session.
Eventually (and by looking at the memory output in a copy regular copy of VBA we can see that 060080D6 becomes 40 (as opposed to 00)
waiting for the instruction that causes this.
Several instructions occur (all written in 16 bit writes)

There are several other functions available rather than just the breakpoint on write (bpw) method.
They include
Execution breakpoints.
Here when a specific command is executed a break will occur.
ARM and THUMB versions exist.

Memory breakpoints.
Can be made for when a section is read from or written to.

Memory/Register editing.
A bit like the cheating section but more complex. Here a given register or a given memory section can be altered.

Raw data
You can either dump data from a section or you can write it to a section at a given point.

Searching.
The memory can be searched for a string (hexadecimal or ASCII)

For a full list of the commands and their syntax see the readme-sdl-h.html file included in the download.
If you have understood the guide so far the immense power this technique has should not be apparent.

Video hardware overview.
The following section gives an overview of the way the GBA (and subsequently DS) video hardware works from a hacking perspective. It is intended both to allow the assembly hacks to make more sense and to supplement the graphics and font hacking (under text hacking) sections.
Note this section while going into some detail does not delve into some of the quirks of the video hardware such as the bits other than those used for priority being ignored when data from the DS 3d hardware using the bg0 layer. GBAtek (http://nocash.emubase.de/gbatek.htm) and the various source code and developers guides/libraries are suggested reading for this sort of thing.

There are
The registers often called/referred to as the background controllers (although some control aspects of the OAM)
the sprites aka the objects/objs controller (the Object Area Memory aka OAM).
the video ram itself
the palettes
The DS has the above and some 3d hardware, this is covered in 3d hacking.

The text is usually part of the background unless it is encoded on a sprite, the characters (as in player character and enemies not graphical representations of spoken words) on the screen are usually sprites/objects.

Another source of information
http://www.coranac.com/tonc/text/video.htm

http://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm#Graphics%20Hardware%20Overview
http://www.coranac.com/tonc/text/video.htm (good for display timings)

The video ram.
The GBA has 96 kilobytes of video ram found at 06000000- 06017FFF hex, while this may not seem a great deal (compared to the hundreds of megabytes sported by modern PC graphics cards) much can be accomplished using it. See http://www.pineight.com/gba/managing-sprite-vram.txt for more on this (the link is aimed at sprites but the logic carries over to the backgrounds).
http://nocash.emubase.de/gbatek.htm#lcdvramoverview
What each subsection is used for depends on the video mode (see http://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm#Graphics%20Hardware%20Overview and http://nocash.emubase.de/gbatek.htm#lcdiodisplaycontrol).

Registers/Background controllers
There are several registers, each with a function.
The dominant register is DISPCNT (LCD control) located at 04000000 hex.
It controls the video mode and which layers are displayed.

Backgrounds as you might imagine are used for the backdrop and the sprites “run” around on top of this.
Each of the 4 background layers (BG0 through BG3) have their own register controlling several aspects and there are several other registers controllings
http://nocash.emubase.de/gbatek.htm#lcdiobgcontrol
When you see a background “scrolling” as seen in the tetris worlds OAM hack below it is usually these registers being manipulated, sometimes to create a “random effect” several trigonometry operations or similar are stacked/added and then written to the BG controller.

A note on the “window” feature. Here different layers can be changed in priority for different areas on the screen, it is often used for menus and other effects where a menu will appear but leave a section of the screen visible.

The OAM layout
Much like the background controllers there is a section for sprites. In games many more sprites are likely to be in use that backgrounds so rather than registers there is the OAM>
The OAM memory sections starts at 07000000 hex for both the GBA and DS (the DS uses the ARM9 memory), the GBA is one kiloybyte while the DS is 2 kilobytes.
Each sprite/OBJ/object gets 8 bytes (first sprite is at 07000000, second is 07000008, third is 07000010). Remember the numbers get flipped by the time they end up from the explanation here. That is to say the number (each character represents 4 bits/1 nibble) 1234 would actually read 3412.
The OAM room for 128 sprites (referred to as OBJ0 through to 0BJ127) with each sprite's attributes having 6 bytes. The OAM is read and write (unlike the scrolling registers for the backgrounds) and consequently can be used for calculations in a game (a game may not “know” a wall is at a given point but by comparing the sprite location to something (located in the binary for instance) it is possible to determine if a wall is there.
OBJ0 has the highest priority over the other sprites in any instance. OBJ1 is higher than everything but OBJ0.....
Note however that the background can have a higher priority.

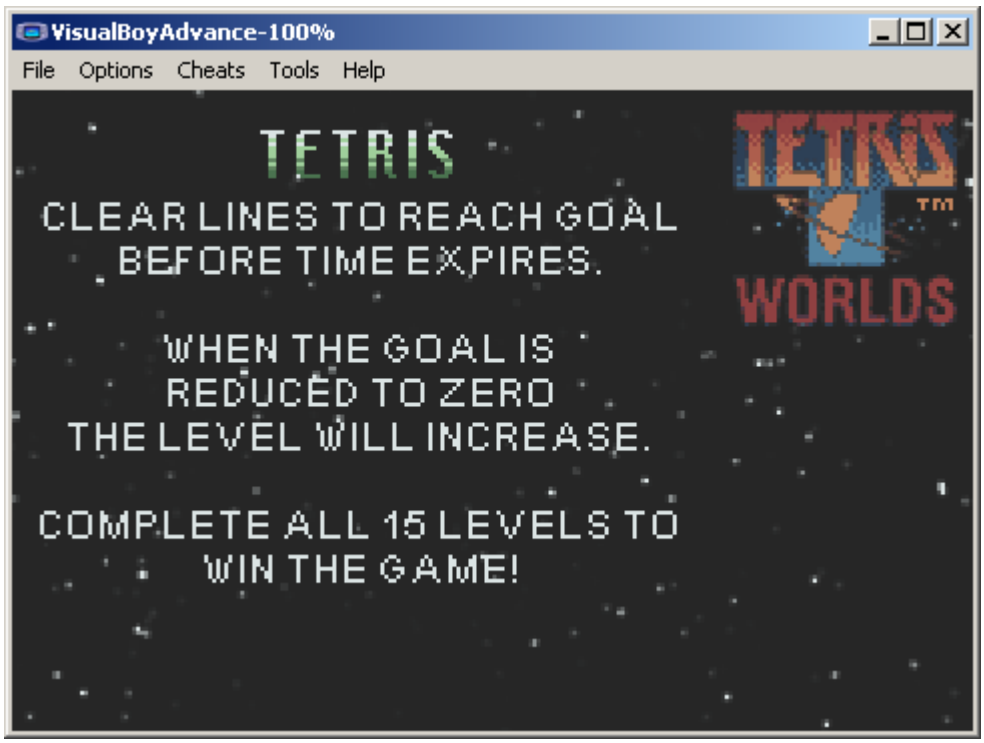
It is entirely possible to have sprites “offscreen”, such a method is often used to “store” sprites for later use.
Note when viewing the ram through a tile viewer or similar you may also see sprites/backgrounds that are no longer used (a good example is the title screen may still be visible soon after “starting” a game, while this may be there in case the player wishes to reset to the title screen it is often there as there is little point in wasting cycles/resources erasing something. It is mentioned as it is somewhat counter intuitive for those coming from a PC coding background where memory is freed as much as is possible.
Layout
http://nocash.emubase.de/gbatek.htm#lcdobjoamattributes

The palettes.
There are two palettes on the GBA able to store 256 15-bit colours (the GBA/DS colour method is detailed below in the graphics hacking section). One palette for the background and one for the objs/sprites. The palette can be used as either a single 256 colour palette (used for 8 bit per pixel) or a region of 16 palettes with each storing 16 colours (used for or a
The BG palette is located at 05000000-050001FF hex (512 bytes, 256 colours)
The OBJ/sprite palette are is located at 05000200-050003FF hex (512 bytes, 256 colours)
The DS has 4 palettes: 2 for each screen with the same arrangement as the GBA.

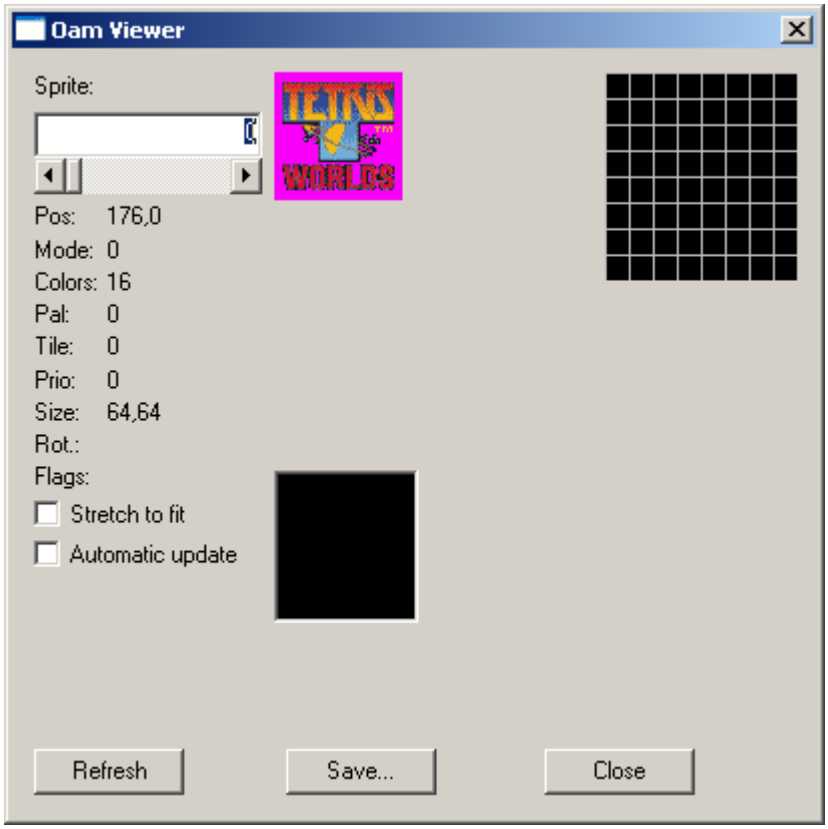
Note it is also possible for the graphics themselves to be colour information (the so called bitmap mode) where each “pixel” is also the 15 bit colour, hardware demands of this mode mean it is used sparingly (see the “modes” links above).
More reading:
http://nocash.emubase.de/gbatek.htm#lcdcolorpalettes
http://br.geocities.com/erikjs_br/geral/gbapal.htm

OAM hacking
OAM is covered in the graphics hacking section but simply put the OAM is the section of ram that controls where and what is on screen at any given time.
Here a simple moving of a sprite according the whims of the hacker will be done. Text is usually a background feature and will be covered below.

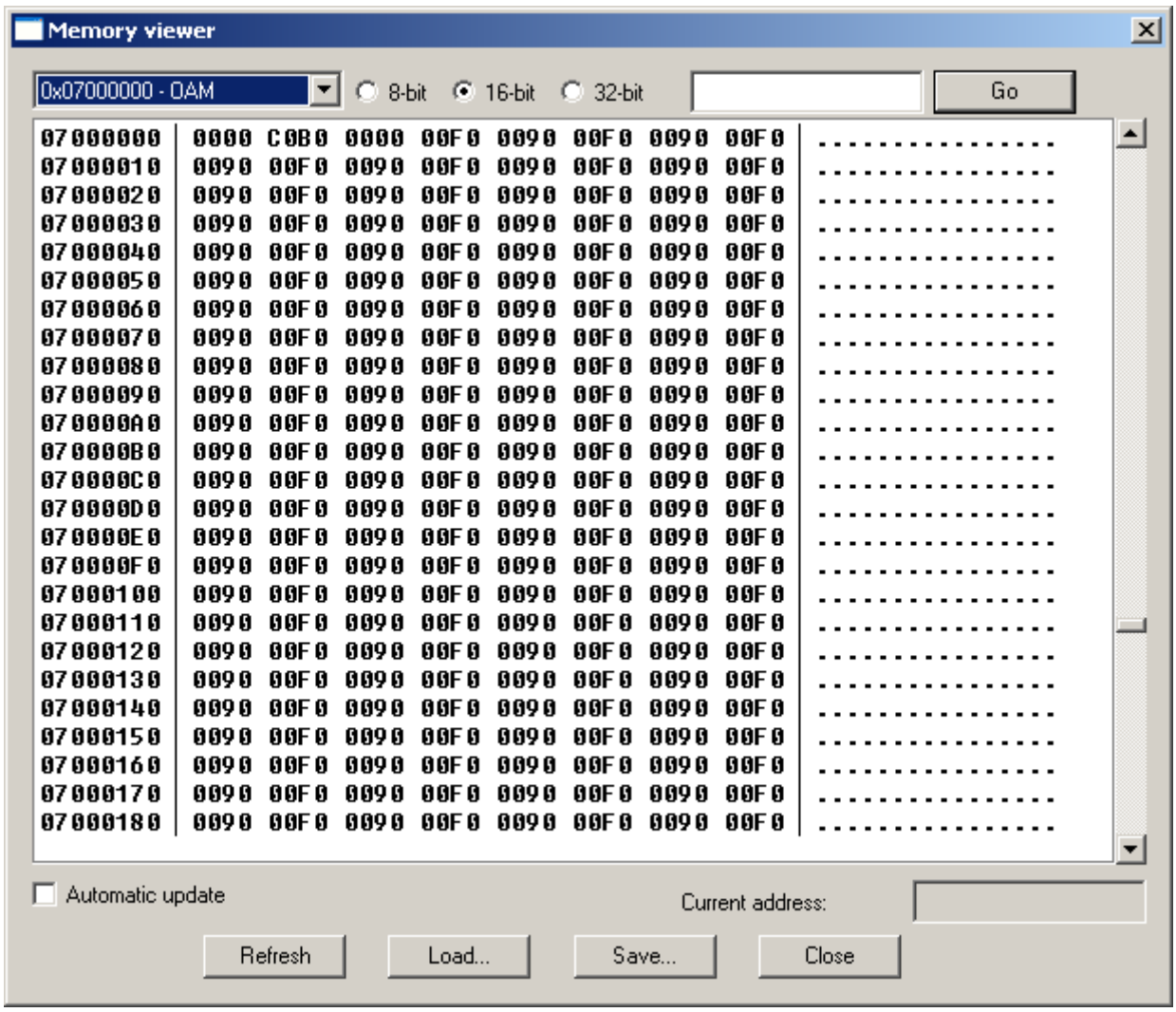
The logo sprite in the top right of the visible display is the target.



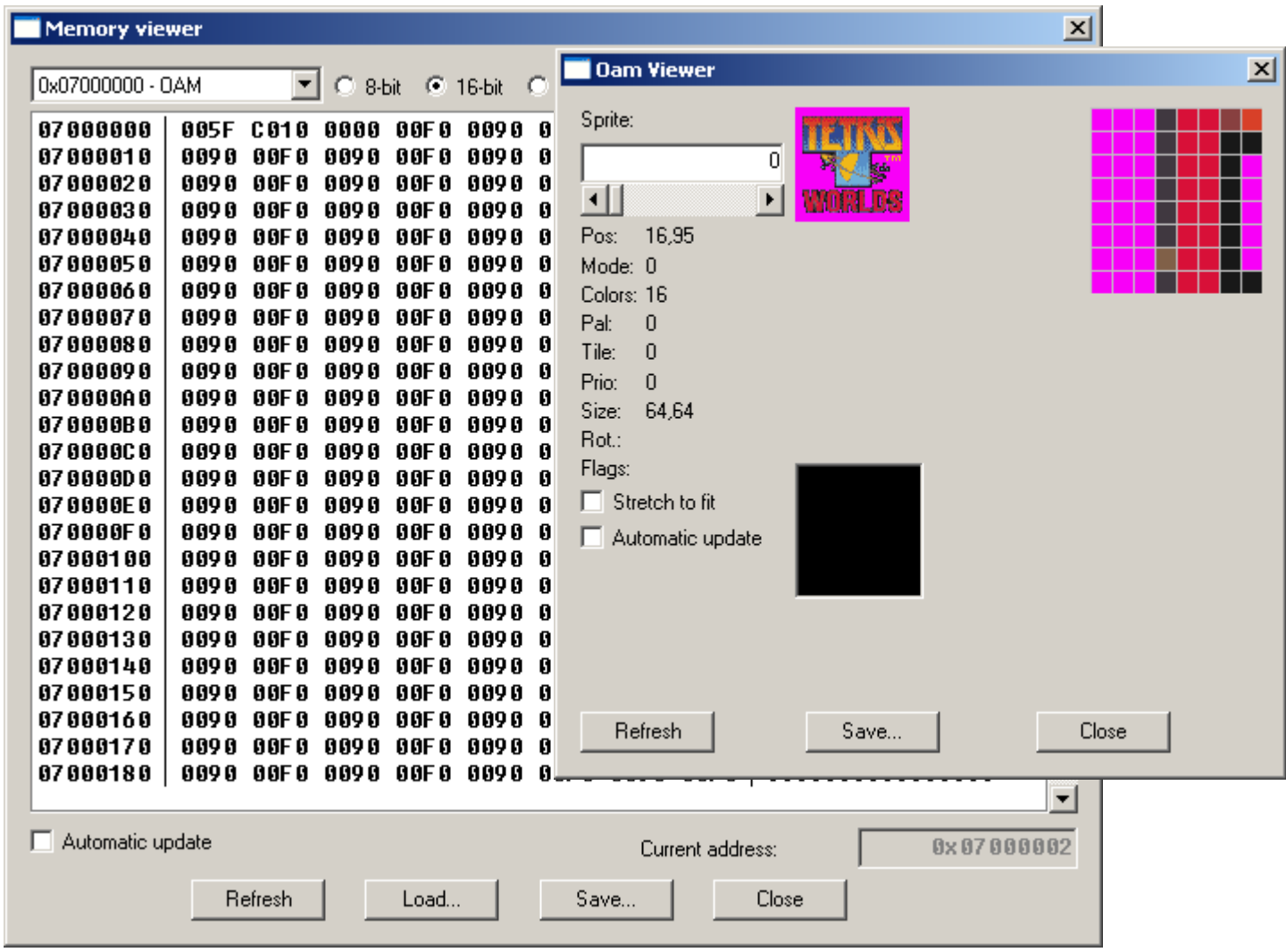
There are multiple methods that can be used now.
The simplest is to look at the location of the sprite an act accordingly. There OAM is a well known memory section and the emulator features nice information on the OAM and such things:



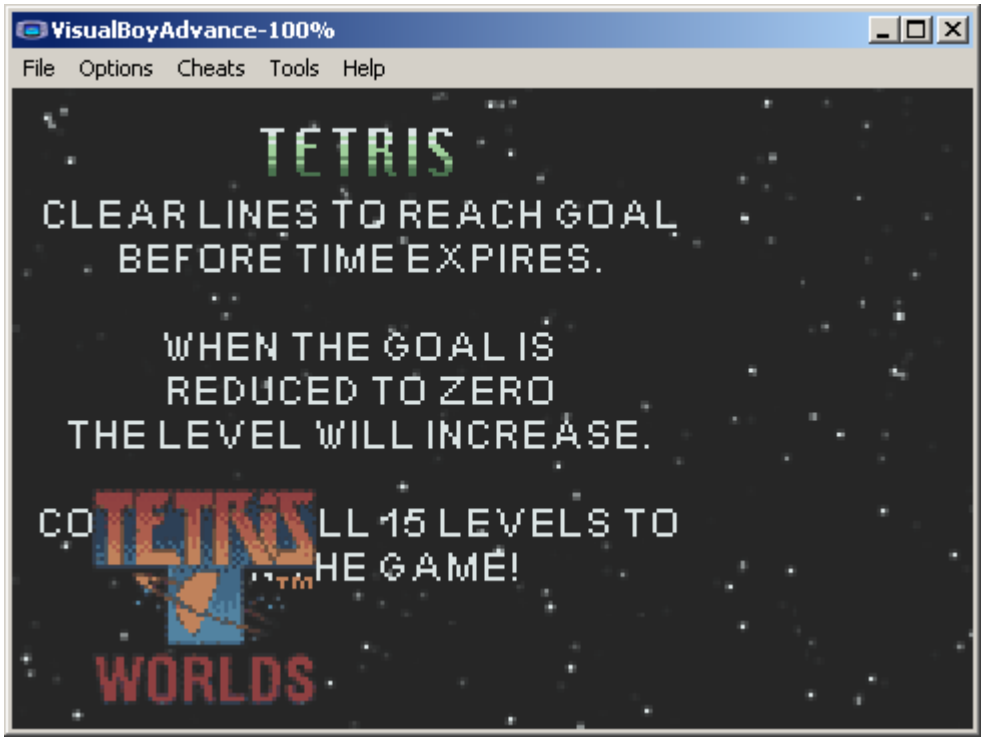
Even if the emulator did not have this option there is always the memory viewer (available in most emulators)



There are 128 possible sprites (given the numbers 0-127) with the lower numbers having higher priority. Either experimentation or knowledge from the OAM viewer tells us that the sprite it located at coordinate 176,0 (the y coordinate counts down from the top of the screen).
Changing some values


Memory viewer
0x07000000 - OAM
8-bit
16-bit
07000000 005F C010 0000 00F0 0090 0
07000010 0090 00F0 0090 00F0 0090 0
07000020 0090 00F0 0090 00F0 0090 0
07000030 0090 00F0 0090 00F0 0090 0
07000040 0090 00F0 0090 00F0 0090 0
07000050 0090 00F0 0090 00F0 0090 0
07000060 0090 00F0 0090 00F0 0090 0
07000070 0090 00F0 0090 00F0 0090 0
07000080 0090 00F0 0090 00F0 0090 0
07000090 0090 00F0 0090 00F0 0090 0
070000A0 0090 00F0 0090 00F0 0090 0
070000B0 0090 00F0 0090 00F0 0090 0
070000C0 0090 00F0 0090 00F0 0090 0
070000D0 0090 00F0 0090 00F0 0090 0
070000E0 0090 00F0 0090 00F0 0090 0
070000F0 0090 00F0 0090 00F0 0090 0
07000100 0090 00F0 0090 00F0 0090 0
07000110 0090 00F0 0090 00F0 0090 0
07000120 0090 00F0 0090 00F0 0090 0
07000130 0090 00F0 0090 00F0 0090 0
07000140 0090 00F0 0090 00F0 0090 0
07000150 0090 00F0 0090 00F0 0090 0
07000160 0090 00F0 0090 00F0 0090 0
07000170 0090 00F0 0090 00F0 0090 0
07000180 0090 00F0 0090 00F0 0090 0
Automatic update
Current address:
0x07000002
Refresh
Load...
Save...
Close
Oam Viewer
Sprite:
0
Pos: 16,95
Mode: 0
Colors: 16
Pal: 0
Tile: 0
Prio: 0
Size: 64,64
Rot.:
Flags:
Stretch to fit
Automatic update
Refresh
Save...
Close

Gives


VisualBoyAdvance-100%
File Options Cheats Tools Help
TETRIS
CLEAR LINES TO REACH GOAL
BEFORE TIME EXPIRES.
WHEN THE GOAL IS
REDUCED TO ZERO
THE LEVEL WILL INCREASE.
TETRIS
WORLDS

From here you can use the cheating methods detailed above to do what you will.

Note some games automatically redo the OAM every so often (usually at a every Vblank) so this may not work as well as you might like. You could however make a cheat to force this value to be what you want. This is a useful hack as some sprites prevent you from moving somewhere in a game and moving the sprite allows for sections not previously open to become open.

Background hacking
The GBA as mentioned features background and sprite options. Use of the background is not preferred if the choice between that and sprites is available but it is used in games and is worth knowing.

Here a “skinny font” hack will be attempted.
Font hacking will be covered in text hacking (which follows graphics hacking) but just quickly most Japanese symbols are a fixed width while other languages will tend to squash smaller characters together:
i.j.l.1.;|
That is the normal representation.
Now with a fixed width (both 12pt sizing)
`i.j.l.1.;|`
even with the little “flicks” on the characters and the slightly bolder appearance it does not look so nice to most people.
Hacking a game to display characters in a different manner is quite an advanced hack.
Reducing the size though is quite simple, assuming it is not done externally by another file (nitroSDK font files often used in DS are called NFTR and have the width included) it will be in the binary itself.
This will then be determined by the background at some stage.
Game. GBA game, 0804 - Maji Nation (J). While this never saw a Western release the game rather nicely has a fully translated script contained within the game, this will not be covered here though.
We already have the font

Step 1. Find the text control in the memory.

We will use a combination of methods today.

Screenshot.

There are a given number of characters on the screen so it is then possible to try messing around with things.

.....some messing around later.....

Bingo. We have the text section.

Now we run a tracing session on the Initially everything will be logged and disassembled.

.....some messing around later.....

We have the text routine.

Rather nicely it adds a tile width every time, a simple tweak on the opcode and

screenshot

One skinny font.

Level hacking
Appearances and such are all well and good but what about the levels you play?
Of course it is possible to hack the levels but it can get quite complex as there are many different ways of making a level for a game and many different ways of saying what can be done in the level.
Broadly speaking there are three types of level

All in one. Here everything (levels, limits, enemies...) are all part of one file.

Overlay.
The backgound is one “layer” (not necessarily a layer in the video hardware sense)
The level/platform is another
The enemies another.
The pickups (occasionally part of the “enemies” section)
The collision data (if present at all) another.

Dynamic.
A example serves here. In a game like GTA there is no set group of NPCs, they are generated by the game code (in this case in real time but games like diablo may generate in a “loading” sequence) and placed randomly (usually within limits).
In older systems (especially in platform shooting games) enemies would be “endless” and simply come from a spawn point.
Good news is such things are usually able to be altered by changing a byte or two in the ram (well within cheat range).

Calculations/operations are done on any number of things (location via the OAM, location determined every v-blank and added to a position in general memory, referencing the OAM or referencing a premade “collision” table so unlike some of the other methods in this guide there is little chance of a general guide beyond what is below.

Have a read of some examples (unlike other sections there is no “level” hardware so the

following sections have examples in other platforms):
http://treeki.googlepages.com/
http://www.romhacking.net/?category=&Platform=&game=&author=&os=&level=&perpage=50&page=utilities&utilsearch=Go&title=&desc=level (utilities but many have source code or details)
http://www.romhacking.net/?category=&Platform=&game=&author=&perpage=50&page=documents&level=&docsearch=Go&title=&desc=level

Procedural generation/ dynamic generation.
Not an example hack per se but worth knowing about. It is quite an old technique (legendary space sim Elite released in 1984 used it to make the galaxies/worlds in the game rather than store the lot) and still a fairly common technique (all games are technically a form of it when compared to static media like books, music and film) but it is experiencing a renaissance of sorts lately. Procedural generation involves using a computer or to a lesser extent the gamer to make a level, character design, colour scheme..... How it does it usually involves some algorithms or similar (note that algorithms can have a definite output and as such are compression not procedural generation) or manipulation of a basic layout by the user/random number generator. It is worth noting as it has the potential to create a lot of different things (far beyond the capabilities of a modestly sized design team) and can be used to diminish the effects of repetition (how many people have the same “face” in a game, add a bit of random generation in there and you have thousands of “different” faces) for a fairly modest space increase. It is not a very common hack to add such things in but theoretically it is quite simple; expanding the space is not hard (it is often unavoidable) and tweaking a binary to do extra work is called for in all but the simplest of assembly hacks.

Data representation (file formats and theory)

It is assumed you have read the section on binary and hexadecimal above or are familiar with it.
While a given bit can mean anything in practice methods are used that mean many files have things in common. In the world of PCs this would include things like exe files, elf files, dll files. Where rom hacking is concerned it is not necessarily quite as easy but it is worth knowing about. The GBA not having a file system makes this a bit harder but there is still something to be gained, to this end it will be covered last.
You are probably familiar from reading textbooks and similar things with the concept even if you did not know it.
Most files have a header describing the file to come (where it is and occasionally some advice on how to treat it) followed by the actual data, similar to a contents section and main text of a book.

Addressing

Data is all well and good but with no idea how to process it/find what you want there is no use in having it, this means you need to find a method by which it is possible to obtain the data required for a given task. Usually this means making lists of the locations that things can be found at or addresses if you will.
There are two main methods of addressing, these being direct and sector based. Either way lists of addresses are called pointers and are a fundamental part of text hacking and retain a fair level of importance elsewhere. This is a two edged sword as far as rom hacking is concerned, on the one hand it can accurately determine where and how big a file is and on the other it means another file format.

Direct is what is says and is a direct link to the file in question. However this can be complicated by a few things. Note, rarely are the pointers obscured (they may be compressed though) but the obscuring methods mentioned above in the cheat making section are useful to know.

Offset. Here where the address counts from is not necessarily the start of the file, in the case of text files the start of the text section is sometimes used as the start of the numbering and the game counts from there.

Relative. A hybrid of the two methods above, here the pointer counts from itself (that is to say if the pointer had a value of 1F and was located at 4C in the file the data would be at 6B), it may sound a bit complicated but in practice it can save resources as a game will likely know the address it is already reading and can thus add the number located there. A lot of CPUs work with relative addresses so such a concept is important for assembly work.

Sector based
Has the same pitfalls as above but as space increases (looking mainly at the wii and gamecube) it becomes unwieldy to directly address files so instead a section (usually a fixed length) of data is given a value and the value is listed as a pointer. Going back to the book analogy this would be chapters.
A simple example.
A file is FF long,
Each section of 16 bytes (0-F, 10-1F, 20-2F.....) is called a sector.
Sector 3 (assuming no “sector 0”) is then 20-2F, not a huge gain for such a small file but when addresses can end up being 8 or more digits long a small number can save a substantial amount of space.
Most of the time a sector will only be used for one file/subsection (file 1 takes sectors 010-11F, file 2 120-14B....) and the remaining space will be unused. This is the reason the NTFS file system has a “file size” and “file size on disk” property when you look up a file size.
The two methods can however be combined as well and “sector 1A, relative address 1B” type addresses can exist.
As an aside the direct addressing mentioned above is often called byte addressing and sometimes words/double words or quad words can be address values instead (mimicking sector addressing), usually when people talk about sectors though the size of the sectors is in the kilobyte range as word addressing does not carry much of an advantage (if any).
A technique like this used in memory is called paging. Here memory sections are given names (page numbers if you will) and then normal addressing occurs with those pages. Sometimes it is “seamlessly” integrated to the hardware, other times it requires some fiddling with the software and other times it is done manually by the user (in the case of some memory cards by physically switching it with a button).
The NES was notorious for the use of mappers to extend the abilities of the hardware (hacking a game to use a different mapper is considered one of the hardest hacks for a NES hacker):
http://bripro.com/low/mappers/index.php
It is not so important for the GBA and DS (some save related stuff) as far as hacking goes although some flash cards and consequently homebrew did/do use such techniques to allow things like 128 megabyte carts (memory address limit is 32 megabytes) and extend the RAM of the GBA.

Calculated/signified/fixed width addressing

Mentioned mainly to stop you searching for non existent pointers.
Calculated
Here the text is scanned and new lines generated and ends of sections determined by various methods. As you can imagine this is very resource intensive and is not done very often. The main reasons for use are for dynamic text, that is text that changes size/font/shape*
*There are simple methods that do not affect the overall line length so changing text in a game does not necessarily indicate this method.

Signified.
These have the most resemblance to human language, here the end of the line is signalled by a given character (almost invariably the same but occasionally there are numbers counting up, most notably in the likes of some compression methods)

Fixed width.
In the case of game menus (most notably Japanese RPGs) a lot of the time a fixed length of text serves as a pointer, that is to say at a given point a new line will start regardless. Working around such a limit can be costly in terms of time too. Fortunately there are fairly simple workarounds that can be employed (see pseudo variable width fonts) and you can often simply alter the value that the game reads (will almost always involve altering things at opcode level).

Returning to assembly hacking for a moment. The GBA (and the GBA cart slot in the DS*) is mapped to a memory section as follows
08000000-09FFFFFF Wait State 0 (usually only 08 owing to most GBA carts being less than 16 megabytes)
0A000000-0BFFFFFF Wait State 1 (mirror)
0C000000-0DFFFFFF Wait State 2 (mirror)
*the DS in DS mode only recognises 08XXXXXX hex and 09XXXXXX hex and the mirrors have no meaning.

This means that sections with a lot of these values close together (but generally equally separated; 08 can be in an address too) and is thus a quick and easy method of finding data, be warned it is used for all the data types so it may not necessarily lead you to the data you want.

Data alignment. While technically coming under addressing it is important enough to merit a distinct section.
While it is entirely possible to put data wherever you choose processors and the supporting hardware* is often limited to calling data at given points (namely 8,16,32,64,,,,, bits depending on the function/method/hardware) or boundaries if you will. The terms are usually byte aligned (8 bits), word aligned (16 bits),double word aligned (32 bits), quad word (64 bits) aligned.
When you call for data from somewhere other than these boundaries the data will have to be treated further (see bit shifting) and this is an extra cycle or more which is not a wise idea if you are already planning to do operations on top of this.
*your hardware/language of choice may have the ability to get data from wherever but in reality it is only doing the above method (or similar) for you.

This becomes especially important for things where resources are already taxed. In rom hacking this is sound/multimedia and (de)compression (ever more popular as games get bigger and systems powerful enough to deal with useful compression methods).

File formats

The DS and a lot of other systems have a fairly common method of storing data. Naturally it varies with the type of data being stored and the usage but there are consistencies.

Magic stamp. This is 4 characters (usually big endian unlike the consoles themselves) and can usually be tied to the file type (SDAT: sound data. NARC: Nintendo/nitro archive. SSEQ: sound sequence to name but a few).

Header

This starts immediately after the magic stamp and will contain info on the file to come. This is where it varies but usually something like the following will occur, all sizes and offsets/pointers will be little endian.

Length of header

length of file (total)

[files are often contained within other files when it comes to the DS, information on these files will now usually follow if it does. Note the DS will blindly parse this data so it is quite possible to trick it into loading other files, many hacks have done this to great effect).]

Compare this to a BMG (used for text)

Game in question is New Super Mario Brothers European release.

File is course.bmg File ends and 0840 hex.

```
0000000 | 4D45 5347 626D 6731 8004 0000 0200 0000 |
MESGbmg1........
0000010 | 0200 0000 0000 0000 0000 0000 0000 0000
| ................
0000020 | 494E 4631 6000 0000 1000 0400 0000 0000 |
INF1`...........
0000030 | 0200 0000 0A00 0000 1000 0000 2200 0000
| ............"...
0000040 | 2C00 0000 3C00 0000 4600 0000 F800 0000 |
,...<...F.......
0000050 | 0401 0000 6401 0000 C801 0000 7402 0000 |
....d.......t...
0000060 | 8802 0000 9602 0000 CC02 0000 F802 0000
| ................
0000070 | 0000 0000 0000 0000 0000 0000 0000 0000
| ................
0000080 | 4441 5431 0004 0000 0000 5900 6500 7300 |
DAT1......Y.e.s.
0000090 | 0000 4E00 6F00 0000 4300 6F00 6E00 7400 |
..N.o...C.o.n.t.
00000A0 | 6900 6E00 7500 6500 0000 5300 6100 7600 |
i.n.u.e...S.a.v.
00000B0 | 6500 0000 4F00 7000 7400 6900 6F00 6E00 |
e...O.p.t.i.o.n.
00000C0 | 7300 0000 5100 7500 6900 7400 0000 5900 |
s...Q.u.i.t...Y.
00000D0 | 6F00 7500 2000 6E00 6500 6500 6400 2000 | o.u. .n.e.e.d.
.
00000E0 | 1A00 08FF 0000 0100 1A00 0601 0100 2000 | ..............
.
00000F0 | 1A00 08FF 0000 0000 5300 7400 6100 7200 |
........S.t.a.r.
0000100 | 2000 4300 6F00 6900 6E00 7300 2000 7400 |
.C.o.i.n.s. .t.
```

```
0000110 | 6F00 0A00 6700 6F00 2000 7400 6800 7200 | o...g.o.
.t.h.r.
0000120 | 6F00 7500 6700 6800 2000 6800 6500 7200 | o.u.g.h.
.h.e.r.
0000130 | 6500 2E00 2000 4400 6F00 2000 7900 6F00 | e... .D.o.
.y.o.
```

And to a SDAT (used for sound)
Game in question is Yoshi touch and go (chosen for the small sound file), this file is the smaller sound_data_vs.sdat. This is by no means a full specification, it is designed to draw your attention to the concepts used to make files in general.

The file ends at 000595c0 the FAT location is 083c and the size of the FAT section is CC the INFO (which follows the SYMB one you can see) section starts at 0664 (0624+40). Again you can see the “magic stamp” in action. Notably this magic stamp can and does also appear for unknown files, even those made by other companies.

```
0000000 | 5344 4154 FFFE 0001 C095 0500 4000 0400 |
SDAT........@...
0000010 | 4000 0000 2106 0000 6406 0000 D801 0000 |
@...!...d.......
0000020 | 3C08 0000 CC00 0000 0809 0000 B88C 0500 |
<...............
0000030 | 0000 0000 0000 0000 0000 0000 0000 0000
| ................
0000040 | 5359 4D42 2406 0000 4000 0000 4C00 0000 | SYMB
$...@...L...
0000050 | 0801 0000 1C01 0000 3001 0000 7401 0000 | ........
0...t...
0000060 | 8001 0000 8401 0000 0000 0000 0000 0000
| ................
0000070 | 0000 0000 0000 0000 0000 0000 0000 0000
| ................
0000080 | 0200 0000 8801 0000 8F01 0000 0200 0000
| ................
0000090 | 9E01 0000 6000 0000 E101 0000 7000 0000 |
.... `.......p...
00000A0 | 0300 0000 A801 0000 B701 0000 CB01 0000
| ................
00000B0 | 2500 0000 F201 0000 0702 0000 1C02 0000 |
%...............
00000C0 | 3302 0000 4A02 0000 5802 0000 6902 0000 |
3...J...X...i...
00000D0 | 7702 0000 8602 0000 9502 0000 A802 0000 |
w...............
00000E0 | BD02 0000 CB02 0000 DB02 0000 EC02 0000
| ................
00000F0 | FF02 0000 1003 0000 1E03 0000 2B03 0000 | ............
+...
0000100 | 3C03 0000 5303 0000 6C03 0000 8303 0000 |
<...S...l.......
0000110 | 9903 0000 AB03 0000 C403 0000 DE03 0000
| ................
0000120 | F403 0000 0B04 0000 2404 0000 3A04 0000 | ........
$...:...
0000130 | 4B04 0000 5F04 0000 6F04 0000 8404 0000 |
```

```
K..._...o.......
0000140 | 9204 0000 A404 0000 0400 0000 B304 0000
| ................
0000150 | BE04 0000 D004 0000 DC04 0000 0400 0000
| ................
0000160 | EC04 0000 F704 0000 0905 0000 1505 0000
| ................
0000170 | 1000 0000 2505 0000 0000 0000 3305 0000 | ....%.......
3...
0000180 | 4305 0000 5205 0000 6105 0000 7005 0000 |
C...R...a...p...
0000190 | 7F05 0000 8E05 0000 9E05 0000 AB05 0000
| ................
00001A0 | B805 0000 C605 0000 D405 0000 E405 0000
| ................
00001B0 | F405 0000 0200 0000 0206 0000 0E06 0000
| ................
00001C0 | 0000 0000 0000 0000 4247 4D5F 5653 0042 |
........BGM_VS.B
00001D0 | 474D 5F56 535F 5749 4E5F 4641 4E00 5341 |
GM_VS_WIN_FAN.SA
00001E0 | 525F 5653 5F53 4500 5653 5345 5F53 5953 |
R_VS_SE.VSSE_SYS
```

The Wii/GC will tend to use a similar method but any values are likely to be larger (many more bits for a pointer/file size) and/or use sector based addressing. Data contained within is likely to be larger too owing to the higher levels of detail. Some example specifications are available in the links in the tools section.
For more on game file formats in general (with numerous examples) have a read of the following site:
http://wiki.xentax.com/index.php?title=Game_File_Format_Central
And a guide from said site:
http://wiki.xentax.com/index.php?title=DGTEFF

Cryptography and things preventing hacking.

It has already been shown the rom hacking is not a trivial exercise but sometimes there are extra steps that can make life even harder. This section covers cryptography, developers tricks and compression.

Cryptography

Warning, the following section may seriously impair your ability to enjoy cheesy TV shows/films (although if you are reading this you probably already laugh the depiction of computers in such cases).

For various reasons people try to obscure something they have created from others or prevent others from altering it. Rom hacking being concerned primarily with finding out how something was made and altering it will often run up against it. A more in depth discussion on compression will be included here too.

Space issues, resource issues/assumptions, the law and the fact such matters are not given the attention they need during a great deal of educational courses all tend to combine make it possible to bypass these issues.

While not strictly related to this guide the following video/presentation provides a great deal of insight:
http://uk.youtube.com/watch?v=uxjpmc8ZIxM

Cryptography deals mainly with 3 areas
Hashing
Signing/signatures
Encryption

Hashing
The three main uses for hashing are error detection, file verification (making sure it has not been tampered with) and with a tweak on the concept sender authentication. This is also the order of increasing complexity.

Now hashing in the most simplistic form is the simple odd or even.
Example: I send you a number (46 and tell you it is even), you however get 45 which is odd and obviously not the number I sent you.
Of course there just as many odd as even or a 50% collision (a collision is where two files have the same hash) rate which is completely useless for just about everything. However the idea can be expanded to the sums of numbers (4+6) and beyond.

Common hashes are bytesums (used in a lot of DS save games), crc 16 and crc 32 (used just about everywhere but possible to fake quite easily so is falling out of use for anything beyond file checking where there is not much chance of someone making a matching file (this actually happened to some GBA roms late in the life of the GBA)), md5 (used in bit torrent and loads of other places) and sha1 (stronger than md5 and also used in several places).

Sidenote rather than store the actual password for something (which several cases have demonstrated can be easily read off a disc) a computer will usually just store a hash of the password so what you enter merely has to have the same hash as the original password to allow access (failing that if for example multiple hashes are used it will serve to rapidly shrink the list of possibilities). Naturally people made lists of all the hashes and passwords that made them (called rainbow tables if you fancy reading up) for the more common/simple hashing algorithms so your password may not be as safe as you think.
Signatures are just a tweak on encryption to output a "hash" instead of an encrypted file (if the hash is known anyone can generate a hash and send the new hash but to generate a new signature with the data having changed should not be very easy). Whether the signature is a hash of sorts or a form of encryption is used on the hash depends on the method used but the principle is the same.
An important note, sometimes what appears to be a signature may in fact just be a hash using a different method to the standard/what was used already (whether this is deliberate or not is up to debate). For example the cyclic redundancy check (crc) can be performed using several different polynomials or keys to the commonly used values.

A nice page describing CRC from a mathematical perspective:
http://www.mathpages.com/home/kmath458.htm

An interesting aside comes from error correction, the most common method (used in the binaries part of usenet in the form of par2 files) is parity, It works by oversampling a selection of data (extra par2 files increase the amount of samples available and so can repair more and more damage). For example you can define a straight line on a graph with two points but if one is out then your line is wrong, given three or more points and you can still get the correct line (assuming you have a means to tell if a point is wrong, for instance a hash table of the points.
From error correction also comes error prevention, redundancy is the usual method which involves padding the file with otherwise useless data (gambling on any errors appearing in this junk) or more commonly replicating important data several times throughout the file (DVDs have three main file types: .vob (the video, audio, subs, menus....), the .ifo (all the data on the locations of everything in the vob files and data on how to make the menu up) and the bup (a byte for byte copy of the ifo file, note your decryption app may leave these alone so do not be surprised if they are different following decryption).
This oversampling concept returns in lossy compression used in audio and other multimedia (sample a sound often enough and even if you lose data your listener will not be able to tell).

Encryption
Encryption is designed to obscure data and it is usually based on so called "hard maths" (that is something that is hard to work backwards from a result to get the method/initial values rather than being a concept that is not usually seen (most people have probably never encountered the error function (http://planetmath.org/encyclopedia/ErrorFunction.html) but the maths behind it is simple enough)).

“Hard maths” usually stems from prime numbers (a number that can only be divided by 1 and itself to leave a natural (counting 1,2,3, etc) number) but ellipses are good as well and there are various other methods like random* numbers used like primes (factored together) and XOR (based on the boolean logic principle of XOR).

The RSA algorithm example is good here for something slightly beyond the truly elementary examples being given out here:
http://mathcircle.berkeley.edu/BMC3/rsa/node4.html
There are two terms/types of encryption to know that are very useful when dealing with encryption. They are symmetric encryption (the same passkey is used to encrypt and decrypt) and asymmetric where one key is used to encode and another to decode), both have their advantages and disadvantages but generally symmetric is less demanding on resources but has more in the way of “flaws” and asymmetric is more demanding but less susceptible to compromise (but as you will see that means very little in the real world). Some methods combine the two, notably most aspects of web browser security (SSL encryption) whereby a asymmetric key is used to generate a window of time to transmit a symmetric key (the session key).

*you may have seen the fairly recent Debian linux problem where the numbers generated were not quite random and thus could be guessed (http://www.theregister.co.uk/2008/05/13/debian_openssl_bug/)

The simple example would be a "multiply every 4th number (starting with the first) by 2"

example
original message, call it my phone number I do not want anyone but the person I am sending it to having.
1111,1111,1111
"encrypted"
2111,2111,2111

a hash/signature could be to encrypt and then take every 4th number and send that giving
2222

when the person gets my message as
2111,1111,1111
and the signature generated reads
4222 something is obviously amiss.
Naturally I have to keep my "times 2" method secret (a concept known as security by obscurity and widely panned by cryptographers) but people could know my every 4th number hash if they liked. The holes in my scheme are blatant (1234,1234,1234 would be the same signature and what happens if a number is 5 or above) but you should get the idea.

Decryption
This is the subject of thousands of documents, sites, journals, education courses, careers..... and is an extremely complex subject to boot. Secondly there are laws (generally enshrined high up (as in national/continent wide) in the law) generally aimed at preventing people decrypting files they do not have the right to decrypt (as well as possessing too strong a level of encryption capability) and they generally read that if someone has implemented a method properly (as in the method is used and not just sitting there, a broken implementation does not count as not properly implemented) it is illegal to try and decrypt it (regardless of however weak the encryption is).
However games are fairly unique in that they join the ranks of viewable but encrypted data and the general consensus is that if you allow anyone to see your work (especially using readily available equipment) it will not be safe.
Encryption takes time, CPU power and a healthy knowledge of the concepts involved to implement. There is also a time limit making it whereas a would be hacker does not have such a limit.
One of the nicest tools in the arsenal of someone looking to crack encryption is an emulator. As you are probably aware by now signals are transmitted around a computer to the different areas, while setting up probes at junctions is possible (and often necessary) it is a pain to do (especially if the signals are contained within a chip). Emulators also provide a nice model to poke and prod in an attempt to get something to work in a different way to what was intended as well as being able to do things real hardware can not; the DS for example has several read only/write only sections but an emulator can be happily programmed to ignore this which is doubly useful as encryption often relies on just these sections to work properly (be effective) and sidestep some of the resource issues mentioned above (if you can not read a section then you do not have to use resources obscuring the value). As symmetric keys are vulnerable if they are found out they are often stored in these “write only”/process only sections of memory.
Quite in fact a lot of the hacks to real hardware aim to add such abilities (replacing memory with an equivalent but readable section is a common one) or in the case of computers with multi process operating systems at the heart of it attach such capabilities to the operating system (such tools are very useful for debugging so the duality of knowledge appears once again).

Another interesting aspect of emulation is that it can be used in a timing attack, encryption requires some operations to be performed on the “plain text”, if the hardware is well known (and thus is able to be emulated, conversely consoles are almost identical when it matters) by measuring the timings for certain things to happen things can be inferred about the encryption process and/or keys. An example for timing would be to cause the number to be high enough that “spillover” from multiplying needs to be tidied up.

Ignoring social engineering (whereby the human side of the encryption problem is targeted and often extremely successfully, mounting a operation to infiltrate a multinational company with the aim of grabbing their keys is a bit beyond this guide though) there are many methods to work around encryption.

Brute force is the method by which all the possible passwords/keys are tried in an attempt to get the right one, methods by which the time required or the amount of keys that need to be tried counts as a failing. However nobody minds if the time is still far longer than the shelf life of the data (a common example is credit card details: an encryption that takes 12 years to break is no good when the credit card expires before that time is up).

Many attacks on encryption require new encrypted data (either in general or more commonly a known source) to be generated, rom being a “read only memory” tends to preclude this method. Save games on the other hand are new data and with some effort can be manipulated.

Other methods rely on statistics and the language (think scrabble tiles: the less common characters are worth more points because they are less common, similarly if you transpose a character with another you can often guess what the original character is because of the context/surrounding text). Do not be so quick to write off language based attacks for while they may not be as useful in the machine/pure data encryption world table making (generally considered the basis/starting point of rom hacking) can and does make extensive use of it.

Just to get a sense of the scale of the numbers involved a kilobyte is 8 x 1024 bits (or 8 x 1000 if you are a hard drive company). 8 is the number of bits in a byte which means 8192 bits is 2^8192 possible combinations (which is a huge number of combinations: there are just under 2^25 seconds in a year so at the rate of say 1 a second you are looking at 2^8167 years, length of time the universe has been around is approximately 4.6 x 10^17 seconds or about 2^56 seconds: http://www.umich.edu/~gs265/bigbang.htm). This is why encryption can use what appear to be tiny numbers (what else other than processors and bus width is still measured in bits).

Workarounds. Your chip/softmod in your console most likely bypasses the inbuilt encryption and/or encryption checking that comes as part of modern machines. This does not help you get at the data if it is encrypted though.
The big one and most important as far as rom hacking is concerned is failure of implementation with two main components of this being the passkey and the actual implementation. The wii (as far as the disc encryption is concerned) provided excellent examples of failures in both departments and as such will be used for real world examples.

Failure of passkey. It is wise to keep your keys safe. In the case of the wii the common key (allowing decryption but not encryption so technically not a failure as such but certainly embarrassing and it facilitated further hacks) was stored in the ram that could not be accessed from gamecube mode software, a hardware addon was made which allowed the

upper area to be read and thus the key was extracted. Similarly the first HDDVD key was extracted from the ram when it was there for a brief period.
Brute force is rightly considered a slow method and if encryption is handled properly it is the only method (although there are a few generally unwieldy examples where even that will not work). Anything that reduces the amount of attempts needed counts as an exploit (ignoring certain mathematical things such as even numbers (other that 2) not being primes).
A simple example of a bad encryption is the use of ASCII letters to make a "random" string for encryption purposes. For those that have come directly to this section and missed the section on text encoding you count up in binary and you give letters as you go up similar to how you did as a kid with a=1, b=2, c=3..... (ASCII codepage: http://www.asciitable.co.uk/). Now computers are only (easily) able use 8 bits which means 2^8 or 256 combinations, 127 characters however is more than is really needed for the roman alphabet, arabic numbers and hangers on. This means every eighth bit is guaranteed to be a 0 or in other words I can drop half* the possible combinations for a single character (or divide by 2 for every character in the password) which serves to rapidly reduce the amount of time needed to work out the password. You also can take some more as the first 31 (0 to 19 hexadecimal) are control characters (not printable) and thus not used.

*although halving a number is usually a nice way to reduce the size remember we are working with exponential numbers so half of 10x10^6 is only 5x10^6 at at the rate of one a second is still in the millions of seconds.

Failure of implementation.
Much like other areas of programming cryptography is complex and time consuming (and quite easy to get wrong), this means developers will often use “off the shelf” cryptography methods and often they will fail to implement them properly.
The wii common key mentioned above has aspects of failure of implementation but the so called trucha hack has a far more clear cut example.
You may have a good algorithm but when writing it into a program it is possible to mess it up. Likewise several hacks at various points in time have been made possible when people attempted to speed things up and added in an exploit (the debian bug linked above is a good example).

You may also have read one of the older xbox hacks where MS used a rather bad choice of signature generating algorithm (http://www.xbox-linux.org/wiki/17_Mistakes_Microsoft_Made_in_the_Xbox_Security_System#Encryption_and_Hashing) which had a problem being used in this way (any encryption can be used to make a signature and any signature making tool can become encryption but as this and the crude “multiply by 4” example above shows it is not always a good idea).

So Nintendo have their signatures (one for what it is supposed to be and what it is now you have messed around with it, obviously a signature is required as it is trivial to change a hash table).
In the case of the wii the Nintendo managed to pick a nice algorithm to use (RSA, again detailed nicely here: http://mathcircle.berkeley.edu/BMC3/rsa/node4.html) but alas nintendo messed it up (tmbinc explained it all in far more detail in a rather nice post on his blog http://debugmo.de/?p=61) and used a strncmp() where they should have used a memcmp() (they actually made a few other mistakes all explained in the link but it boils down to that).
For those not up to date on their C? (http://www.cplusplus.com/reference/clibrary/cstring/memcmp.html) the former terminates when it reaches a 00 (which in this case is usually

fairly quickly owing to some nice circumstances with the use of RSA) while the latter checks the entire sum.
This had the effect of making terminating the check before it was complete (i.e. only a few bits in) possible.
Now guessing a signature having used a long private key is very hard but owing to some clever maths you can get a match to the first few bits (you have the key which can check it but not sign it remember) fairly easily on semi powerful modern machine (yours machine) and as only the first few bits are needed courtesy of the end at 00 thing.....
If you are interested in reading more on cracking RSA the following link may be of some interest
http://www.woodmann.com/crackz/Tutorials/Rsa.htm

Another example seen in a few programs is simple checking with data already in the game. “downloadable content*” for the DS is often done in this manner.
Here a password is entered and compared (often to the password buried within the rom itself or maybe a hash if the developers got a bit creative), all that is then required is to alter the check from “if equals” to a “if does not equal” (which can be as simple as changing a single opcode or even less).
*as the DS (when using official carts anyhow) lacks a place to write a substantial amount of data, let alone store it long term, this is often the only way to do it, some games like Daigassou! Band Brothers DX however have large save sections giving an appreciable amount of space to play with.

Analysis of encrypted files. If you still have not read the text section go and do it as the techniques listed there stem from cryptography attacks. However while encryption originated with text many thousands of years ago today binary code is very common and it serves to drastically reduce the effectiveness of a fair few of those techniques.
In the case of text aspects of social engineering (thinking like a human is relatively easy to do)/linguistics (one of the most common if not the most common character in a section of text is likely to be a space “character”). Computers can also have repeated sections or predictable sections which can then be abused to either figure out the required passkey. For example an encryption key will have a “bit” value, this means that assuming the file is long enough sections will be encrypted with the same key (not necessarily that simple mind).
Should you know what a section is and you have a copy of the encrypted data (and hopefully the encryption method) it is then possible to determine the encryption key for some algorithms.
An example.
Ignoring for a moment data snatched from the ram (again why emulation is so good for this sort of thing) you may recall from the data representation/file formats section the tendency of files to use “magic stamps” at the start of the file. It is then possible to assume the first few bytes will decrypt as that magic stamp.
In this example bitwise XOR encryption will be broken for an SDAT file.

The first characters are SDAT or 5344 4154 in hexadecimal.

```
5344 4154
0101 0011 0100 0100 0100 0001 0101 0100
```

XOR encryption here is 8 bit (easily broken by normal methods but it will be used here). Key is BB (1011 1011). Remember for a “2 pin” XOR the output is only 1 if one of the inputs is 1 (none or both means 0). XOR encryption works in the principle that for every binary digit there are two possibilities meaning very quickly you are dealing with powers

2^1024 or 1.797693134862315907729305190789e+308 combinations for a single 128 byte file).

0101 0011 0100 0100 0100 0001 0101 0100
XOR encryption
1110 1000 1111 1111 1111 1010 1110 1111

E8FF FAEF
1110 1000 1111 1111 1111 1010 1110 1111

We know the first part is to be SDAT,
0101 0011 0100 0100 0100 0001 0101 0100
Running XOR against them
0 generates 1. Therefore 1 is the first binary digit.
1 generates 1. Therefore 0 is the second binary digit.
0 generates 1. Therefore 1 is the third binary digit.
1 generates 0. Therefore 1 is the fourth binary digit.

0 generates 1. Therefore 1 is the fifth binary digit.
0 generates 0. Therefore 0 is the sixth binary digit.
1 generates 0. Therefore 1 is the seventh binary digit.
1 generates 0. Therefore 1 is the eighth binary digit.

1011 1011 or BB is the key. This is an example of a “known-plaintext attack” and given the low power of systems is likely to be a very useful attack method. XOR was dealt with not because it is that common by itself but because it forms a building block for other algorithms and is often used as part of a larger series of shifts and the like to obscure data.

In practice any cryptographer would be fired for suggesting such a small key but the principle is there. It also serves to weaken longer keys by reducing the amount of time needed to brute force a solution.
Likewise if a file has not been compressed then all the junk data (which is usually the same character repeated) provides a nice demo, compression (which eliminates repeated sections if done properly) can render this method useless.

Another twist on the known plaintext attack involves write only sections. Here mathematical operations may be able to be performed using such a section. By carefully choosing the data to be operated upon it is possible to then gather data on the encryption and ultimately break it.
A somewhat unrealistic example would involve the XOR operation. By XORing the unknown string with a string of all 1's or all 0's the hidden string could then be determined from the result. In practice such an operation would not be allowed but less revealing attacks like adding* or other maths can be used as the people responsible for encryption often assume the locked down “original” system rather than the enhanced systems of the hackers (the revealing of the wii public key being a good example).

*going back to the timing attack where the operations are carefully timed and things inferred from how long things take adding is likely to cause some spillover. How it handles addition (namely if spillover is detected) can tell various things about the number being stored.

On final twist on known plaintext involves algorithms based on maths (most of them), they often rely on powers to numbers and getting the modulus of the result. Here an engineered

result when encrypted effectively gives the key. Going back to the XOR encryption XORing 00000000.....0000.....0000 (binary) will yield the key used to encrypt it. Similarly 1 to any power is still 1, encryption makers know this so padding is used to prevent this (although the wii ignores this).

Dictionary attack.
A halfway house between the social engineering aspect and true cryptanalysis. How many of you have passwords based on your pets name or similar (or more broadly words in general or something able to be pronounced), now while there are many millions of words available there are fewer words than combinations of characters. This is the reason why "strong passwords" (where you have a number or some punctuation in the password) are so frequently advocated.
A broader definition can include potential keys chosen for their likelihood to come up for a given algorithm (the example of not using ASCII to generate a key detailed above is a good one),
In rom hacking the end stage of the cheat making where there are a handful of possibilities out of thousands caused by monitoring locations for changes.

Regional Differences
It becomes more useful in the case of text hacking but regional differences are useful. These can highlight changes and prove useful when working with encryption that has a minimal amount of data available on it.

Circumvention take 2.
Hacking is all about changing things and this can include encryption. Rather than crack the encryption keys it is often possible to change the encryption value (one of the fundamental "rules" of encryption is that it only works if the parties using it know what is being done). This is obviously useful in the case of asymmetric encryption (where the decryption key is different to the encryption key) as it is then possible to change the keys (by generating and implanting a new set) while avoiding a massive rework* of the code to get around it.
*The amount of work may not actually be quite so big (data obviously ending up where it needs to be) so you can sometimes replace a whole encryption section with a "copy to" command.

You can also seek to introduce bugs like those that provided the basis for the trucha bug although this tends to come into things where hacking firmware is involved or additional checks on encryption are present and a simple workaround is not possible.

More reading:
http://www.schneier.com/essay-028.html

Save games.
This section will deal mainly with the DS for now although information should correspond across systems.
Extraction and manipulation of saves for games is usually one of the first aims of hackers when dealing with a new system. For one it allows backup of the saves and secondly saves can usually contain anything and this provides an avenue for attack (a large number of hacks have relied at some level on save games at some point).

Most save games are hashed/have a checksum and some systems take this a step further and tie a save to a console.

Save extraction on the consoles.
Some of the gameshark/action replay devices are able to back up saves.

GBA.
Older flash cart linkers supported extraction of SRAM and Flash save games. Alas EEPROM (the most common) was not supported.

There are methods using a cable (sometimes for a flash cart, other times a homemade cable):
http://chishm.drunkencoders.com/SendSave/index.html
DLDI (needs to be able to write) capable GBA slot carts (which run on the GBA as opposed to DS “mode”) of which there are several can use
http://chishm.drunkencoders.com/SendSave/index.html#cart_save (not it involves swapping so some supercards may not work)

Some DS slot carts are able to extract saves too using tools like rudolph's GBA rom dumper
http://kotei.blog.so-net.ne.jp/2008-09-27 (Japanese language but it is a popular tool throughout the GBA/DS “scene” so instructions should not be hard to come by).
In the same manner SRAM saves can be dumped with EEPINATOR (remember to do it twice):
http://blog.davr.org/2007/03/31/eepinator/

DS saves.
There are countless DS save dumping tools.
The save type for DS games is not specified simply in the rom unlike some other systems so most DS slot flash carts use a 512 kilobyte save (most saves are far smaller) and aside from the padding it is untouched from what the game makes. Several conversion tools exist but the following site is used by many:
http://www.shunyweb.info/ (nicely it also supports no$gba save format)

Broadly they fall into three categories.
SRAM based.
Designed for early carts that either lacked write support.
http://nds.cmamod.com/download.html
http://www.pineight.com/dx/RAC (designed for animal crossing and sports tools to interact with it but it also supports other 2Mbit save using games).

GBA slot based
The above SRAM based methods were largely eliminated with the introduction of the ability to write to the memory of the flash cart being used and ultimately forgotten about by the time DLDI arrived and issues with cart writing ceased to be an issue.
http://www009.upp.so-net.ne.jp/rudolph/nds/Backup/

DS slot based.
These tend to use the wifi capabilities of the DS or occasionally swapping and the ram.
http://www009.upp.so-net.ne.jp/rudolph/nds/Backup/

addendum. Expansion packs served to replace the full blown GBA slot carts for most people but as write code was adapted to them there are tools that can use them.
http://gbatemp.net/index.php?download=964

GC saves.
Possibly the hardest of all the save types to work with. The easiest way is with a wii and one of the following wii homebrew applications:
http://wiibrew.org/wiki/Homebrew_apps/Gamecube_Saver
http://gbatemp.net/index.php?showtopic=126834

Save file specifications
http://members.iinet.net.au/~theimp/gci/GameCube%20GCI%20Memory%20Card%20Save%20File%20Format%20Specifications.pdf

Wii saves.
A lot of games are simply able to have the game transferred to the onboard SD slot. Some games are unable to be transferred however as there is a flag (literally a couple of bits) in the save that tell the console not to do it. Several homebrew game launchers and tools exist that allowing copying
http://wiibrew.org/wiki/Homebrew_apps/Gecko_Region_Free

Save games.
For any hash generating mechanism worth anything the hash will change when the data changes. The GBA, NDS and GC files can be swapped at will (although “friend codes” may interfere in the case of the DS) while the wii makes use of console dependent hashing. Wii save hashing is slightly more complex as unlike the others it is console based:
http://wiibrew.org/wiki/Wii_Security#Savegames_on_SD_cards
A note, games are tied to a given save but you can change this:
http://gbatemp.net/index.php?showtopic=128215&hl=

All the consoles have example of games that hash saves so as to detect unwanted changes (the company will probably say it is to prevent corruption but it has the side effect of preventing editing). There are two main ways around this
1) patch the game to ignore the hash. Example Final Fantasy 3 trainer:
http://gbatemp.net/index.php?showtopic=40825&st=0&start=0 (do a page search to find it).
2) find the hash and replicate it. In keeping with the “corruption detection” excuse the hash of choice is usually fairly simple (bytesums are used in Final Fantasy 3 DS for instance) so it can be detected, assembly level hacking is still a distinct possibility though.
The usual method mimics that of finding cheats, here the game is repeatedly saved with minor changes (one step, one bit of money lost) and that way the section that is the hash section is determined as it goes against the cryptography principle of send your secret key in a separate transmission.

Sidenote. Hashing can be worked around by using conventional cheats and altering the ram which is usually simple to edit (see the cheating section). Savestates made just before save points coupled with cheats can also be used to produce multiple save games with more precision and/or dodge save limits (once every five minutes or similar).

Second to this comes save editing.
Again it mimics cheat making in that small changes and trying to determine what changed is the method of choice. Sometimes it is a psuedo save-state and is just a dump of the memory sections responsible for things.
Nicely values used to store things are usually “relative” in some way (a small dagger is 01, a medium dagger is 02, a large dagger is 03.....) so it is possible to experiment and obtain

values for extremely rare items.
Of course asm level hacks and tracing can be applied.

Many people have done this for several games and made extensive editors with the information.
GBA:
Several pokemon editors exist. Usually require saves to be in gameshark or some similar device format. VBA is able to convert them.
http://www.pokecommunity.com/index.php
Doom
http://chishm.drunkencoders.com/GBA_SaveMod/
Super Mario Advance
http://chishm.drunkencoders.com/GBA_SaveMod/
Savestate editors (not save editors per se)
http://www.zophar.net/utilities/gbacheat.html

DS
Pokesav is perhaps the most well known:
http://gbatemp.net/index.php?showtopic=50498&st=0&start=0
Final Fantasy 3 has a selection of editors. Thundaga is the most full featured (it also fixes saves negating the need for the hack).
http://gbatemp.net/index.php?download=358
Animal Crossing:
http://www.aibohack.com/nds/
Nintendogs
http://www.aibohack.com/nds/
yu-Gi-Oh (several exist, usually from the same author):
http://gbatemp.net/index.php?download=1028
Mario Kart:
http://www.caitsith2.com/

GC:
Save file specifications (header info and the like)
http://members.iinet.net.au/~theimp/gci/GameCube%20GCI%20Memory%20Card%20Save%20File%20Format%20Specifications.pdf

Wii.
Various Mii extraction/editing tools exist.
See also http://wiibrew.org/wiki/Wii_Security#Savegames_on_SD_cards
See also partition editor: wii save games are tied to the ticket ID, changing this allows many different saves and so many different hacks.
http://gbatemp.net/index.php?s=&showtopic=128215&view=findpost&p=1693866

Developers tricks.

In addition to the built in security developers can add their own features that serve to prevent hacking in one way or another. This can range from custom controls (think guitar hero/rock band or the solar sensors from boktai, tilt sensors from GBA warioware twisted) to dongles/addons to a system (think the commercial DS browser with the ram pack) to more sneaky things like Final Fantasy Crystal Chronicles on the DS where the code had the side effect of detecting a flash cart and setting a timer (in effect making a time limited demo*) or the cheat prevention method or mirroring (and possibly obscuring) a value.

The first two options are usually dealt with by cart makers or occasionally other hackers (quickly the addon carts will have some place in the ram and that will be used in some way) but timers and “cheat” prevention is a hacking problem. Cheats and their associated problems were covered in the cheating section above.

*there are a few other examples where nothing so overt was used and things like extra enemies, shorter time and other frustrations have been used.

Compression.

Even though space is fairly cheap and plentiful on the systems contained within this guide wasting it where it is not needed (and can make for a smaller* (cheaper) memory chip being used to store the game) is generally frowned upon by those controlling the purse strings. Often considered the bane of rom hackers it can pose a problem but with a bit of patience and knowledge it can not only be easily worked with but become a helpful tool.
*While the GBA supports roms up to 32 megabytes a lot of carts (or the owners) prefer roms to be 16 megabytes or less. Similarly using a dual layer disc for the wii where a single layer will do will not serve to win you many friends.

Like encryption above compression places a significant demand on system resources and such has really only started to become apparent as the CPU power of systems has increased making it more viable (compressed data has to be decompressed to run after all) and with players demanding ever longer, more detailed and more complex games it is a viable way of cramming more in there. This means data alignment (another drain on resources) is very important here and unaligned decompression is rarely, if ever, performed.

Compression tends to be used on less vital components to bring space usage down (although compression of the binaries/parts thereof is far from unheard of), unfortunately this usually means graphics and text (probably the two main areas of ROM hacking). The GBA and DS for instance have various decompression routines built in to their bios, it should be said that while said routines (or compatible ones) are commonly used the compression is not limited to these built in routines.

Compression works on the principle of repeated sections being replaced with shorter unique sections, a crude example follows:

Today for breakfast I had several pieces of toast.

My friend had several bowls of cereal for breakfast.

As you can see the words for, breakfast, had, of and several are repeated meaning a form of compression could be to replace these repeated characters with something shorter:

Today 1 2 I 3 4 pieces 5 toast.

My friend 3 4 bowls 5 cereal 1 2.

The first phrase is 102 characters in length whereas the second is 64 and they both say the same thing, of course elsewhere 1,2,3,4 and 5 would have to be defined (a slightly better scheme would replace 1 and 2 with a single value) but it is easy to see why compression is used. The definitions of the compressed elements is called a dictionary.

As storing data using fewer bits than required is at best computationally expensive (you

guess/iterate an answer, also a fairly good if infrequently used encryption method) and at worst impossible it will be assumed a sufficient amount will be used here. An aside is lossy compression, used most often when the difference will not be noticed but makes the overall file smaller (this is usually video where an error on a single frame is near impossible to detect or in audio when the playback frequency also makes detection difficult. Computer code is a precise thing though so such techniques are not detailed here.

Before going on it is worth making a distinction between binary compression and “word” compression. The former deals with the binary while the latter deals with the text itself (much like the example above), text compression has fallen out of favour though in recent times but it is present on older systems.

Without question the most common compression on all the systems is LZ aka lempel ziv based (usually LZSS but called LZ77 even though they are slightly different, more reading http://michael.dipperstein.com/lzss/).

Next up is Huffman and run length encoding (each about as common as the other).

Custom variations of these are next with different compression methods/algorithms being next.

Compression proper.

Much like bypassing cryptography above you can also bypass encryption by replacing a “decompress” section with a copy to. A nice worked example exists courtesy of Labmaster: http://www.romhacking.net/docs/253/

GBA/DS tools. Several tools that deal with compression exist.
Crystaltile2 has some very good LZ features (searching for compressed sections, decompression from file system and sub files (NARC), compression).
Other editors (mainly LZ), check the various features on offer. Some deal mainly with graphics, others can use VBA SWI logs.
http://www.romhacking.net/?category=&Platform=10&game=&author=&os=&level=&perpage=20&page=utilities&utilsearch=Go&title=&desc=compr
http://www.gbadev.org/tools.php?showinfo=56 (a useful command line tool with source code).

BIOS functions (SWI - SoftWare Interrupts) represent a trade-off (which is usually adequate for game purposes) between speed, usefulness and size and because of this games can and do implement their own compression functions. Ignoring custom compression for a moment the BIOS decompression functions mean compression can often be determined with a simple tracing session looking for the relevant SWI calls.
For a discussion on the low level hardware aspects (what registers do what) of the SWI calls see http://nocash.emubase.de/gbatek.htm#biosdecompressionfunctions
This section deals mainly with the theory and how the data is manipulated and how the SWI calls expect the data to be formatted.

Packing techniques.

Compression gains are often enhanced with packing techniques and “filters”. The GBA and DS provide decompression functions as part of the BIOS and those are available at

http://nocash.emubase.de/gbatek.htm#biosdecompressionfunctions

Most of the text you will read (both here and in general) will be black text on a white/single coloured background. This means that for 4 bit per pixel GBA image fonts there will be only two characters (out of a possible 16) present. You can then simply use a single binary digit to represent 4 and expand them when the time comes (white or 0000 is represented as a 0, black or 0001 is represented as a 1).

As an aside if the “white background” is set to colourless the background of another section can show through meaning a font could still be packed even if it does not appear to be. Likewise bitpacking is a relatively simple operation so a game may well implement a custom version in place of the SWI.
GBA implementation
It is almost the cousin of binary coded decimal. Remember to account for endian ness.
For a tile each row is given a byte (two hex numbers). As each pixel can only be one of two values binary can be used.
Now the lines are read off and a byte is generated with each 4 binary digits making one hexadecimal one.
For example
0110 0111 would read out 67.
The BCD style example is is used here as opposed to simply telling you to convert directly from binary to hexadecimal so as to help when dealing with 8bpp imagery*

Another worked example can be found at http://www.coranac.com/tonc/text/text.htm do a page search for “Bitpacking”.

*don't ask, it has been known though.

Filters
`abcdefghijklmnopqrstuvwxyz`
Instead of all that you could write
`a11111111111111111111111111`
2 distinct characters versus 26, both the same length but guess which will compress more easily. Useful for some bitmaps and wave files.
Note in this example 1 represents the “increase” to the next character. It will be detailed in the table making section but most (roman character sets) text encoding will have a given character one hexadecimal value up from the last (a=1, b=2, c=3 and so on). You can then use a so called relative search and feed it a pattern, for example cab (be careful when dealing with capitals) could be 3,-2,+1, the game could then be searched for an hexadecimal values with a character with the next two less and the following one more than that.
GBA/DS implementation
http://nocash.emubase.de/gbatek.htm#biosdecompressionfunctions

LZ compression.
This is a so called sliding window compression. It works by having a window and then parsing it for repeated sections. Should a repeat come up it will create a flag for the decompression function and a pointer to the original uncompressed section and finally a length of the repeat.

[to be finished]

Alternative document:

http://www.romhacking.net/docs/281/
http://oldwww.rasip.fer.hr/research/compress/algorithms/fund/lz/index.html

In the case of the GBA much like the other decompression functions there are two main methods. VRAM safe and WRAM safe. VRAM is slower but writes in 16 bit lengths while wram is faster but writes in 8 bit lengths. Games can and do implement their own versions some of which are incompatible with the BIOS implementations meaning tracing by way of logging SWI calls is not going to work and meaning you may need to reverse engineer the method used*.
*It may be easier to recompress the lot with a known method and implement a BIOS function or a simple copy instead.

http://nocash.emubase.de/gbatek.htm#biosdecompressionfunctions

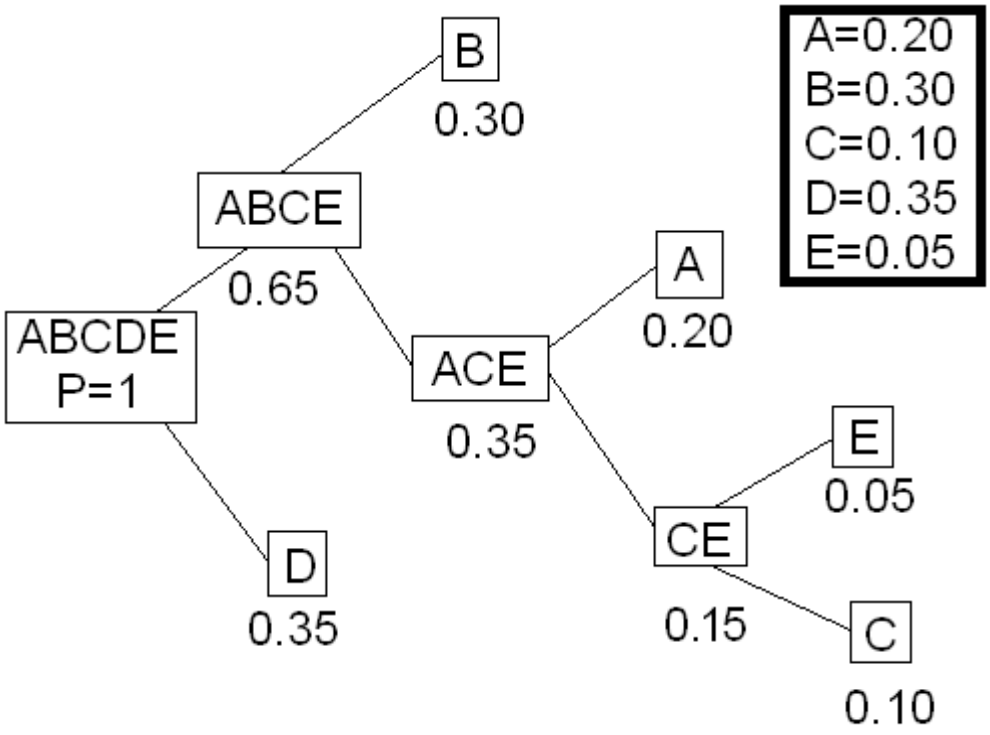
Huffman compression.
In simplistic terms huffman makes strings of various lengths according to how common they are, common ones get short identifying characters while longer ones get longer identifying characters.
The usual method of explanation comes from mathematical probability where the Huffman concept stems from.
In mathematical probability circles a probability tree is a common method examining simple probabilities. It is probably the closest method to the elementary example detailed above.

In probability trees an event is given a probabilities and then events following this are also given probabilities. A simple model may have two outcomes (tossing a coin for example) with each subsequent event also have two outcomes. Each outcome has two possible outcomes and each “branches” out to form a “tree”. It is then possible to read along the tree and calculate probabilities for a series of events.

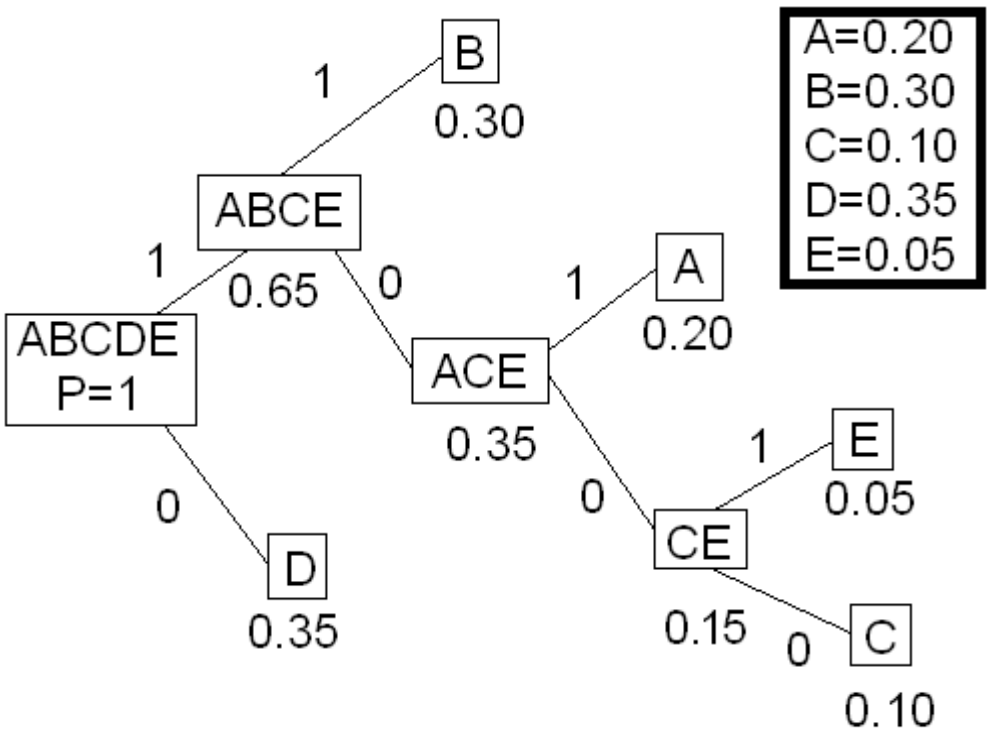
Huffman works along the same lines and uses the end probabilities to assign a value.
Worked example.
5 characters.
A,B,C,D,E with respective probabilities 0.20, 0.30, 0.10, 0.35, 0.05. As frequency (number or occurrences) is directly related to probabilities here it can be used instead (useful as probabilities do not tend to feature in hex editors).

As there are only 5 characters the probability of one them coming up in a section is 1, this is used as the so called root node.

ABCDE (p=1)



Above is a probability tree
Assigning binary digits to the branches



Now to get a encoding value for a letter you simply need to read along the branches noting the binary digit at each step

D=0

B=11
A=101
E=1001
C=1000

In practice strings are used in place of letters. The limit on length of the strings depends on your source file (if it is fairly random it will get more complex) and your decoder (most systems do not handle large 256 bit values all that well).

GBA/DS huffman.
There are several tweaks upon huffman compression. As mention compression taxes resources so data needs to be aligned to a 4 byte boundary (starts at address ?0, ?4, ?8 or ?C)
http://nocash.emubase.de/gbatek.htm#biosdecompressionfunctions
Alternative document
http://michael.dipperstein.com/huffman/index.html
http://www.si.umich.edu/Classes/540/Readings/Encoding%20-%20Huffman%20Coding.htm
http://oldwww.rasip.fer.hr/research/compress/algorithms/fund/huffman/index.html

Run Length Encoding (RLE)
This is the method filters deal with very well although fonts and other images are equally likely work well with this compression. Here a string is parsed (along the length it runs) and the lengths of repeated characters are measured and given a repeat value.
Such a method has limited usefulness as relative simple patterns other methods work very well with will be missed but it is very fast if done properly.
Text example

RRRRRRRRRRRRRRRRRRRRRRRRRRRRLLLLLLLLLLLLLLLLRRRRRRLLLLLRRRRRRRRRRRRR

70 characters.

28R16L6R5L15R

13 characters.

Text is not likely to provide a good example but two colour fonts are frequently mainly “whitespace” with the occasional bit of black for the character/glyph. Moreover with tiles being a certain size a character may not fill all of it and there can be blank lines.
A common method of increasing variety in colours used by a game for example in grass is to have two slightly different colours alternating to form a patchwork effect. Run length would not work well here but LZ which works on sections would work very well.

GBA/DS implementation.
Like other decompression functions there is a work ram (WRAM) version and a video ram (VRAM) version. The destination for the VRAM version needs to be halfword aligned (byte). Usual needs for the compressed data is not a multiple of 4 and padding with 00 if not as well as the source address to a 4Byte boundary (starts at address ?0, ?4, ?8 or ?C).
http://nocash.emubase.de/gbatek.htm#biosdecompressionfunctions

Alternative document
http://oldwww.rasip.fer.hr/research/compress/algorithms/fund/rl/index.html
http://wiki.xentax.com/index.php?title=RLE

The file systems and searching for data within them.
Reference documentation is available at the end of the document, this section deals with the tools and methods useful for hacking.

The GBA has all the data in the binary so “conventional” methods apply for finding files/locations of data. However the binary (code that runs the game rather than the resources) is relatively easy to find (it was detailed above in the example hacks). The sections that deal with the text/pictures often follows the header (with the info on what is coming up) and then data (what the game displays) concept common across computing and related files (pictures and the palette they use for example) are not usually spread “randomly” through the rom.
GBA Rom expansion, When compared to things like nes and SNES expansion this expansion of GBA roms is trivial.
Again the GBA rom is mapped directly to the memory as follows:
08000000-09FFFFFF Wait State 0 (usually only 08 owing to most GBA carts being less than 16 megabytes)
0A000000-0BFFFFFF Wait State 1 (mirror)
0C000000-0DFFFFFF Wait State 2 (mirror)

A lot of rom hackers choose to add things to the end of the rom instead of fiddling around with pointers during text replacement and other such things, of course if you can add more content to a game which also requires space.
To this end you sometimes need more space than is available in the trimmed (if you were unaware memory modules come in powers of 2 sizes (4,8,16,32.... megabytes) but code is under no such limits, a dump of the whole memory chip is used though and is what is used when you obtain images of the memory chips, this junk can then be trimmed from the file).
The more astute among you may have noticed that 08XXXXXX only allows for a 16megabyte file, simply replace 08 with 09 and then start from scratch again (i.e. if you have a file 8 bytes in from the start of the 16 megabyte limit you get 09000008).

The DS file system.
While not required by the DS the nitrorom file system aka nitroFS is used for all commercial roms as far as the community is aware. Most games have their own file formats (either custom, a tweak on the nitroSDK or an implementation of the nitroSDK) although some games (notably the first phoenix wright/gyakuten saiban title and some of the tony hawks games) have only a couple of files in the format with the main files being large files contained within the nitroFS (think zip file within a zip file).
Expanding the DS file should be taken care of by the tools but if doing it by hand remember to also alter the size of the rom (4 bytes located at 80 hex) as well as the file allocation table.
DS roms are compilations of the files developers make and as such can often have interesting extras within. For example an early mario kart hack allowed for some of the beta levels to be used, the SDAT format used by most DS games for sound had substantial help when they were first being reverse engineered thanks to a smap file left in the a game (Zoids saga DS was the game in question although other games have had similar files).

The GC file system.
Unlike the wii below gamecube discs are not signed and can be freely edited. Owing to several games being developed for multiple systems at the same time and the fairly similar hardware in the three systems (GC, xbox and PS2) the file formats are similar between them as well as the DS and Wii.

Wii iso stuff
As mentioned Wii formats are largely the same as GC and DS in layout and concept (although larger/more complex owing to the faster hardware and more memory).
The wii uses isos which at present occupy the whole of a DVD (mainly DVD5 but there are some dual layer discs).

Important terms concerning wii isos:
Signing bug aka trucha: (blocked in 3.3 update along with the original twilight [princess] hack aka TP hack: http://wiibrew.org/wiki/Twilight_Hack) but several easy to use workarounds exist*)
When the wii checks(ed?) the disc it compares the signature embedded in the disc to the signature generated from the files. If they match all is good but should they not match means disc is rejected.

Signatures are what happens when cryptography meets hashing and you are directed to the cryptography section for more on that.
At present (November 2008) the main method is the gecko region free channel but hacked versions of the IOS introduced in 3.3 exist and there is sure to be other methods as time goes on.

Scrubbing: The term whole disc encryption/signing you may have seen when the wii is described is somewhat of a misnomer as the junk (containing nothing of interest) space between useful files is not signed. This is replaced as this unsigned data is in all but a very small handful of cases essentially random uncompressible (by conventional/popular algorithms/techniques anyhow) garbage. In essence scrubbing is just replacing the garbage with very easy to compress 00/FF's (it gets more complex once you add in headers and whatnot but that is the basic principle).
The key is needed to decrypt the filesystem to figure out what to send to silicon hell and as no signing is broken all should be good still.

Methods of manipulation.

While there is a file system or similar the game will not tend to arrive to you as the hacker as a bunch of files for editing (the original xbox is one of the few systems where that might happen) and while it may be a known format (looking at iso files) they will still need to be pulled apart or at least parsed for locations of files to do any serious work. It is equally important to be able to put it back together once all is said and done.

GBA.
It comes as a .gba (occasionally .bin) binary only. The file is not signed and is all regions.
Support for multiple games per flash cart is done at flash cart level.
Size limit for commercial roms is 32 megabytes (16 megabytes is preferred by cart users)*.
GBA addressing has been covered already (08XXXXXX up to 16 megabytes and 09XXXXXX for space after adjusting for higher wait states if need be).
Techniques for finding the computer code (as opposed to the resources such as pictures

and text) is detailed in the hacks and techniques section.

*certain homebrew applications used with early NOR based carts that feature large memory sections can bypass this using bankswitching, presently no hack to a commercial rom has attempted to use it and any hack would be cart specific. There is also rumoured to be a larger commercial rom (GBA video: shrek) but it is undumped.

DS
Comes as a .nds file. The file (aside from some wireless multiplayer files*) is not signed and is all regions.
Support for multiple games per flash cart is done at flash cart level. Size limit is considered 4 gigabit/ 512 megabytes but higher may be possible, largest commercial rom to date is 2 gigabit/256 megabytes.
Note the nitrorom file system is not mandated by the DS (all known commercial roms use it however) and some roms also use another file system on top of the nitrorom file system.

*some early carts do not support signing of multiplayer titles (download play) but having the firmware modification known as flashme on the DS receiving the file tends to bypass this which is useful for streaming things from the DS Download station roms.

Many tools exist to operate on DS files. The most popular are
ndsts (nds top system).
http://www.no-intro.org/tools.htm
Only allows replacement of files with the same size as the original (you can pad smaller files). Several other header viewers/tools such as ndshv from vinpire have similar functionality.

Ndstool. Originally built for homebrew (it is part of devkitpro/devkitarm) but works on commercial roms too. Falling out of favour as it does not work properly on some roms, while rebuilding is a problem the extraction side of things is very very well tested.
The actual program is a command line tool but there are frontends in DSlazy and DSbuff (both frontends require .NET)
Command line batch files.
Deconstruction command
“ndstool -x x.nds -9 arm9.bin -7 arm7.bin -y9 y9.bin -y7 y7.bin -d data -y overlay -t banner.bin -h header.bin”

Reconstruction command
“ndstool -c xmod.nds -9 arm9.bin -7 arm7.bin -y9 y9.bin -y7 y7.bin -d data -y overlay -t banner.bin -h header.bin “
Downloads (other mirrors are a quick search away):
http://wiki.pocketheaven.com/Ndstool

DSlazy
http://blog.dev-scene.com/ratx/archives/category/dslazy
DSbuff
http://www.gbatemp.net/index.php?showtopic=56602
Nitroexplorer (requires .net).
http://treeki.googlepages.com/NitroExplorer2b.zip
Crystaltile2:
http://gbatemp.net/index.php?showtopic=81767&hl=
Custom tools. Usually built for a project.

Nitrorom file system specification:
http://nocash.emubase.de/gbatek.htm#dscartridgenitroromfilesystem

GC

Comes as a .gcm file (often renamed to .iso) file. It is not signed for the GC or the Wii, files are region locked but a there are tools and most chips (GC or wii) should bypass this. Support for multiple games per flash cart is done at iso level with several tools able to do it. Size limit is 1.4 gigabytes (miniDVD) for gamecube and DVD size (4.35 gigabytes) for Wii games if making a multiple game disc.
Gctool:
http://gbatemp.net/index.php?download=894
GCMtool is good for unix like operating systems (X86 and ppc versions exist):

http://www.sadistech.com/gcmtool/tutorial.php
Multiboot iso creation:
http://gbatemp.net/index.php?download=633
http://gbatemp.net/index.php?download=559

There are many other tools for nearly every common OS if these do not suit your needs.

Wii
Comes as a .iso file. Actual data is signed (junk/padding is not hence the exception for "scrubbing" the iso), the decryption key is known and various bugs (see trucha bug in encryption above) allow for data to pass signing checks.
Multiple games per disc is possible but not viable for play (save games do not work properly) at this time.
Size limit is DVD9 at 8.7 gigabytes (DVD5 at 4.35 gigabytes is the usual standard).

Region free.
There is section in the unsigned part of the header the some games use to determine region while others used additional checks. Wii chips claiming to be region free will use this method and it is far from flawless.
When the trucha method appeared newer hacks also followed to deregion games (at the cost of a fair amount of complexity), from these were also born language changing hacks so English text would appear on Japanese consoles.
Today custom launching channels, firmware/IOS hacks and even region changing of the wii allow true multiregion gaming.

File extraction was possible before the possibility to "resign" data was widespread so some early tools may only be able to extract. Tools that can perform both operations are the only ones detailed here.
Trucha application.
http://gbatemp.net/index.php?download=1909
Fails to work on some ISOs so not always used.

Wiiscrubber (should be more accepting of unusual isos):
http://gbatemp.net/index.php?download=3012

Finding files/data required

In the eyes of most hackers the ability to do this along with the ability to determine what a file means is difference between a good hacker and a bad one. Most hackers will use a combination of all the techniques below in varying amounts depending on the files/rom in question.

Tracing (worked example above) is the most certain method but it is not so easy with other systems meaning it is mainly a GBA technique. Monitoring known disc/cart calling methods is still possible but not flawless, a virtual file system for the DS has been discussed but nothing exists at present.
Some of the techniques such as searching for data in the rom with data grabbed from the memory* can still be used though.
*compression, encryption, dynamic data (a changing palette is used to rapidly switch colours with a good example found in Mr Driller 2 (GBA version) in the rainbow blocks), likewise variables (amount of money means a text dump may not be viable)), composite data (a picture will not be stored in the rom but made up of sprites, backgrounds and

whatever else).
You can however use a tracing/logging feature to figure out what happens and improve your odds and wildcards in the case of dynamic data.

Pointers
Pointers have been explained already but on the GBA they will tend to point to the rom section. Long sections of the (especially approximately equidistant occurrences: “08” can be part of the pointer too).
A similar technique can be applied to the DS and other systems where a string likely to be pointer is found and followed but it is usually a check once the file has been found and reverse engineered or it is used for large files in conjunction with other techniques.

File names
Looking back at the social engineering part of cryptography (things like dictionary attacks based on words/birthdays) most files on the DS and other systems will have file or directory names that reflect the contents.
Alongside this is the SDK based hacking concept where the magic stamp of a given file type (which is also unlikely to be untouched by LZ compression) or the extension of the file from the file system is used to help pinpoint what does what.
Note .bin is a general extension used throughout computing and can be anything. Also while an extension may correspond to a known format do not expect it to conform to the standards (the fact the GC and wii games come as a ISO file which is unable to be opened by standard iso extraction tools is a good example).

File sizes
Space is not usually wasted or more importantly files with the same function will have the same amount of space. For example a palette will usually be a small file and there will usually be several of those for one game.
Similarly the size of the file itself can provide an indication (media files (video/audio) are usually quite large while a puzzle game that lacks a great amount of text will tend to have small file used for it.
File sizes are also part of pointers

File occurrences
There is usually only one or maybe two sound files for the DS but graphics/text and so on is likely a different story.

Corruption
It was said at the start the strings of binary digits tend to represent something.
By changing these and then playing the game it can be possible to determine what a given section does.
Naturally there are major pitfalls* (in the 5:5:5 colour format of the GBA/DS at “16th” colour can have unexpected effects, a character outside the normal encoding range can also change what happens to a piece of text) . Likewise achieving precision when things need to be aligned is tricky.
Corruption can be done to both the file itself and the memory. Memory corruption is quite useful for cases where making up a new file (read burning a disc) is not viable and in the case of emulators (with savestates and/or rewind functions) can be extremely useful.
Generally there is reversible corruption and irreversible corruption, reversible is something like inverting all the bits or rotating them) while irreversible will be something like filling a memory section/file with random data (or predetermined but otherwise different data).
Inverting and single bit shifting can fall foul of the 5:5:5 example meaning some

modifications can be done to avoid this.
Another common corruption method pertains to the file system itself. If a file format is used once there is a good chance it will be used somewhere else, by switching files around (level data is a good one here) it is possible to determine what file does what.

*this can sometimes trigger an error handling scenario which may not be good for the end result but for those dealing with hacking/reverse engineering a system it can be very useful.

Disassembly
Working through disassembled code will tend to yield what does what. Naturally this is incredibly difficult (such things are usually abstracted too) which is why tracing became an important technique. It can however be used to great effect when a opcode dealing with a memory location/range is needed for a cheat.

“Brute force” and file analysis
Related to corruption. Here everything is looked at until the data required comes up. Usually it is best to limit use of this and/or eliminate sections with other methods first (the file .sdat is likely sound so checking through the space (which in the case of sdat is often considerable) it occupies in a rom is a waste of time when looking for text).

Similarly while a tile viewer may not reveal pictures it is generally easier to see patterns with colours* that by examining text/numbers.
On the subject of hex editors and aligned data the ability to shift the start of new lines to different boundaries is useful (viewing strings 8 bytes long it is possible to miss patterns that are blatantly obvious when the new lines start every 11 bytes), even if the ASCII readout has nothing to do with the data required we are taught to see patterns in letters so it can also reveal interesting things, not to mention even if magic stamps are not present ASCII text (especially for 3d files) often is.
*replacing a section of text with the same value or otherwise highlighting it can be useful.
Another trick when using a tile viewer is to tar or otherwise archive files without compressing them to save time opening files or work around palettes being in separate files.
Finally there are multiple tools that can search for the common occurrences with compressed data.

Different region/same series/same file type
File formats and so forth will often be reused for different regions or sequels. Should knowledge of one of these titles exist it can be used as a guide/starting point. Such a method is often extremely useful for reverse engineering a file format as multiple examples can be found.

Other hacks
Many times a hacker will find themselves finishing a hack started by someone else or a different hack (a spoof hack may have all the data required for a translation) or especially with the DS when games are often translated very quickly by other hackers into Chinese.
On the file system games it is quite easy as you can hash all the files (compression formats like ZIP often require it as part of the specification), comparing files for GBA games and for files with their own file system is also possible with most hex editors and there are standalone tools for it as well.

Graphics:

These days pixels (small quadrilaterals that can have their colour changed) are ultimately used to represent anything visual regardless of the way it is stored/generated.

2d hacking.

It is covered in detail in the video overview part of the assembly hacking section but generally there are sprites (sometimes objects/objs) and backgrounds. Sprites do not need to be a single tile and backgrounds can either be generated from tiles (a compression method: multiple black tiles for example would be a single tile repeated and then loaded accordingly) or stored as complete “pictures”.
Both have their own memory sections exclusively dealing with them with sprites being the OAM and backgrounds being the background controller.

For the most part the GBA uses a standard method of displaying graphics which you will see called GBA format, there are 2 types you see called 4bpp and 8bpp (4 and 8 bits per pixel respectively).

2d tiles

Think paint by numbers but with pixels. Tiles are as the name implies small sections (usually square) that can be used the generate an image.
The suggested tile editor for ROM hacking on the GBA/DS is called TileED 2002: http://home.arcor.de/minako.aino/TilEd2002/ or the multi purpose tool crystaltile2 comes in handy where oddly shaped tiles (for fonts mainly) are used. In reality any tile editor that supports 4 and 8 bpp GBA format will do (although it does not support 8bpp GBA the tile editor YY-CHR is a good choice too).

If you are hacking a GBA ROM take it slowly when you are scrolling through the file so as not to miss anything.
If you have split the DS ROM apart and find it tedious to open files only to scroll through the few bytes (some files are often very small (a few kilobytes perhaps) but can contain important graphical data) something to do is put all the files in a tar file but making sure not to use any compression (it will be the store option under compression) and then opening this in the tile editor. You may well get a bit of junk displayed for your trouble but it enables a quick scan through. Note crystaltile2 is able to parse the DS file system and also supports various formats used by the DS meaning you do not necessarily have to do this.

It is important to note that as well as note requiring a single tile for a sprite the illusion of movement is achieved by having many slightly different sprites and playing them fast enough to trick the eye.

Tile editing theory.

As most consoles rely heavily on 2D tiles are very important (the newer consoles feature some substantial 3d capability but 2d is still important and used all the time).

You question now is probably something like: what pray tell are tiles?

Simply put they are small “paint by numbers” style pictures (or more often sections thereof) which can be used for sprites (the best way to visualise a sprite is as one of the characters or enemies on a game although they are also used for fonts) or backgrounds.

Any self respecting games console/SDK will provide various facilities including the ability to flip, scale, mirror and rotate (by 90,180 or 270 degrees) these tiles and the consoles

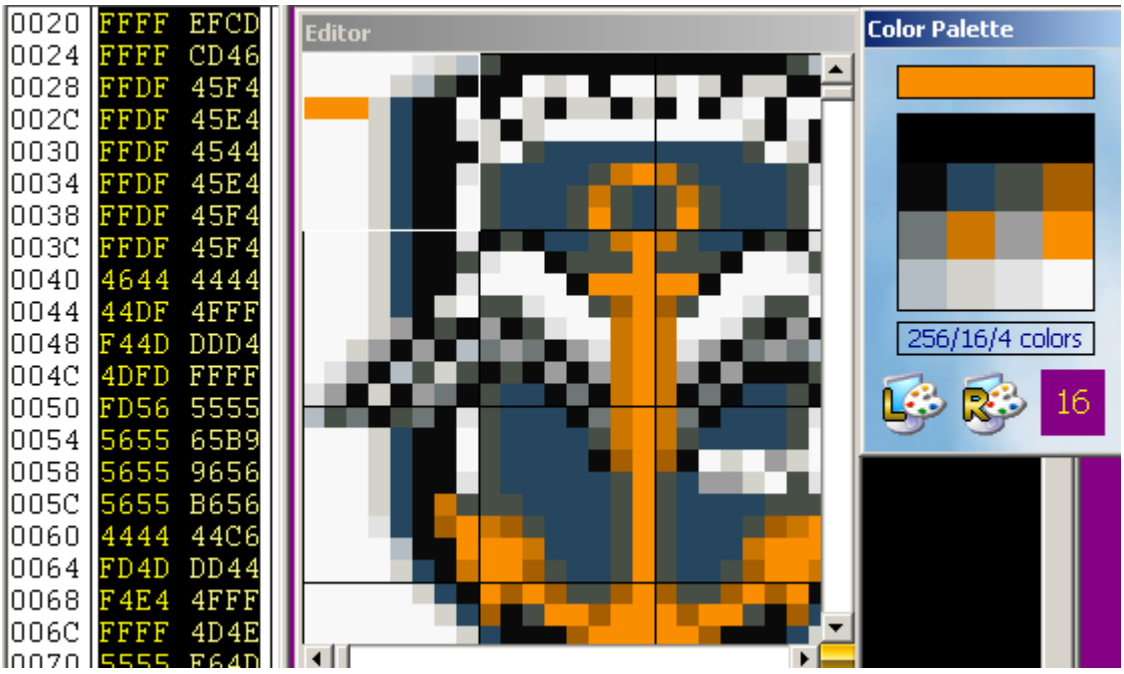
covered here are no exception.

This may not be of great interest unless you are coding your own app/plugin but the following section details GBA graphics “from the ground up”

Remember the GBA/DS is little endian, basically this means you need to flip the palette: ff7f becomes 7fff etc (an important distinction for when you grab a palette from the memory and search the rom for it). Discard the first byte as it is transparent.
Read the remaining 15 bytes and place them in the order they come out.
The palette's numbers merely reflect the colour and intensity of the colour the represent (detailed below).

read 16 bytes of the bitmap and call it a tile, repeat in rows of 4 tiles until all 512 bytes are used.
4bpp is 4 bits per pixel so every byte is 2 pixels and each tile has 32 i.e. 8x8 pixels.
The arrangement is known to the DS/GBA so these bytes merely define what colours are used, this is where it gets a bit tricky though as the nibbles will need flipping. Either way each nibble points to the colour at that place in the palette (in the example below F points to the 15th colour which is white and you can see it repeated)
Here is a picture of the super robot taisen icon with the palette in hex form as follows (first line is 8 colours):

0000	0000	0000	0421	2D04	2128	0174	39CD
01D9	4E73	023F	62F6	675A	739C	7FFF	



Each line of hex corresponds to 8 pixels and the three orange pixels on the top left tile were added in for this guide so you can try and get your bearings.

An 8bit per pixel uses the same principle but each byte gives a location, now however 256 (2^8) colours can be used.

Palette editing

Palettes were already mentioned but picture this: you are playing Final Fantasy (an early one VI or below) and you have just started, chances are you are facing an enemy like G. Rat where your opponent happens to be a Giant Rat probably coloured brown and black or something similar.
Fast forward a few hours and you have saved a princess/town/planet or two already but now you are somewhere new and now you are in a new random battle and up pops your old friend G. Rat although it is probably a putrid blue/green colour and is sporting a new M. Rat moniker (and probably has more HP, defence and some new abilities) so you proceed to kick the snot out of the new mutant rat and continue on (probably to face Plague. Rat (P. Rat) later on).
What happens is the same tile(s) is/are used on both occasions but a different colour scheme is applied for each of those occasions.
Why?
It is a simple (and widely accepted) way to increase variety without having to sacrifice the limited space available (remember the maximum is 32 Mbytes on the GBA with the average being 8 or 16 Mbytes). Note that because of this editing sprites can have some unintended effects at later points in the game.

The paint by numbers analogy was used when attempting to explain tiles and it serves again here: if you imagine a tile as a blank paint by numbers picture the palette serves as the key.
Of course you can edit this key should you so desire but first there is a need to explain the GBA colour system (and probably some colours in general).
Early on we learn that mixing paint and ink of different colours produces another coloured paint/ink, we then learn the same applies to light and using red green and blue in varying amounts to change its colour.
We also know about hex(adecimal), hexadecimal has 16 possible combinations.
Hex has been used to represent just about everything else so why not colours?
Alarm bells may be ringing now: there are only 3 colours and 16 combinations: 16 does not divide by 3 to give a whole number.
Assuming we use one byte to represent colour 2 choices remain: give one colour 6 bits and the other two colours five bits, a choice often used in computers (red is usually given the honour here as blue is harder for human eyes and machines to detect: this is why green light is used in some microscopes where blue would be better and one reason why some of the first/cheaper Blue ray discs have issues).
*As an aside video files are almost always not in RGB but in another form of colour representation (called a colourspace) such as YV12 or YUY2 where there is no longer Red Green and Blue but Luma, Chroma and brightness from which the original colours can be determined. The GBA and DS do not really need this but similar concepts can appear when 3d work is attempted.

The other option is to forget about the 16th bit and instead only use the first 15 (5 bits per colour), this is what the GBA and DS use (the use the same scheme too). The colours are as follows.
See also the cowbite virtual hardware spec:
http://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm#Graphics%20Hardware%20Overview
http://br.geocities.com/erikjs_br/geral/gbapal.htm (has a method to convert RGB 16-bit into GBA color)
http://www.coranac.com/tonc/text/video.htm#sec-colors

All colours (both paletted and bitmapped) are represented as a 16 bit value, using 5 bits

for red, green, and blue, and ignoring bit 15. In the case of paletted memory, pixels in an image are represented as 8 bit or 4 bit indices into the palette RAM starting at 0x5000000 for backgrounds and 0x5000200 for sprites. Each palette is a table consisting of 256 16-bit colour entries. In the case of the bitmapped backgrounds in modes 3 and 4, pixels are represented as the 16-bit colour values themselves.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
x	B	B	B	B	B	G	G	G	G	G	R	R	R	R	R

0-4 (R) = Red
5-9 (G) = Green
A-E (B) = Blue

There is an undocumented red-green swap at 04000002 hex in GBA mode (0 is normal, 1 triggers it). http://nocash.emubase.de/gbatek.htm#lcdiodisplaycontrol Being undocumented some emulators lack support for it so using it is not advised.

Sprites get one palette, and background layers another.
On the GBA there are 2 palettes active at any one time,
On the DS there are 4 palettes: two per display.

Your emulator will probably have provisions for changing the palette although it will probably not be able to save it into the ROM.
The suggested GBA emulator for this text is VBA (Visual Boy Advance) or one of the many forks (VBALink, vba-m and kode54's builds being some of the best, VBA- SDL is a command line version you can use for serious ROM hacking as it included extensive debugging features (useful for determining compression amongst other things)).
You can find the address of a palette in the ram at a particular point in the game by hitting -> tools -> palette viewer although this is of little use for ROM hacking.

As that probably did not make much sense TileED2002 has the facilities to construct your own palette as well as rip one from a savestate or rom, you can then edit it as you will and insert it where ever you like.
Where the palette is located in the GBA rom you ask? That is the hard part, there is a method whereby you grab the palette from the emulator and use the data to find it in the rom, a rather nice guide for doing this for super mario advance is located at http://etk.scener.org/?op=showtutorial&st=3

Otherwise corruption, brute force or ASM are pretty much your remaining options here (palettes are usually found kicking around the picture files or all together in one block).

DS palette editing in some ways is not so easy, DS graphics can either have their own palette inbuilt (garbage at the start of the graphics file if viewed in a tile editor) or as a separate file (various extensions usually with the same name as the graphics).
Realistically a hex editor is the only way forward here.

Notes on palette editing:
A palette may not only be used once in the game so take great care when editing them as you may accidentally change the colour of another section of the game.
Palettes are not always static, that is to say they are not simply dragged up from the rom in some way and stuck in the memory. A good example of this is Mr Driller 2 for the GBA, if you open the game in an emulator such as VBA get into a level and view the palette you will see one colour changing fairly rapidly (it corresponds to the rainbow blocks in the

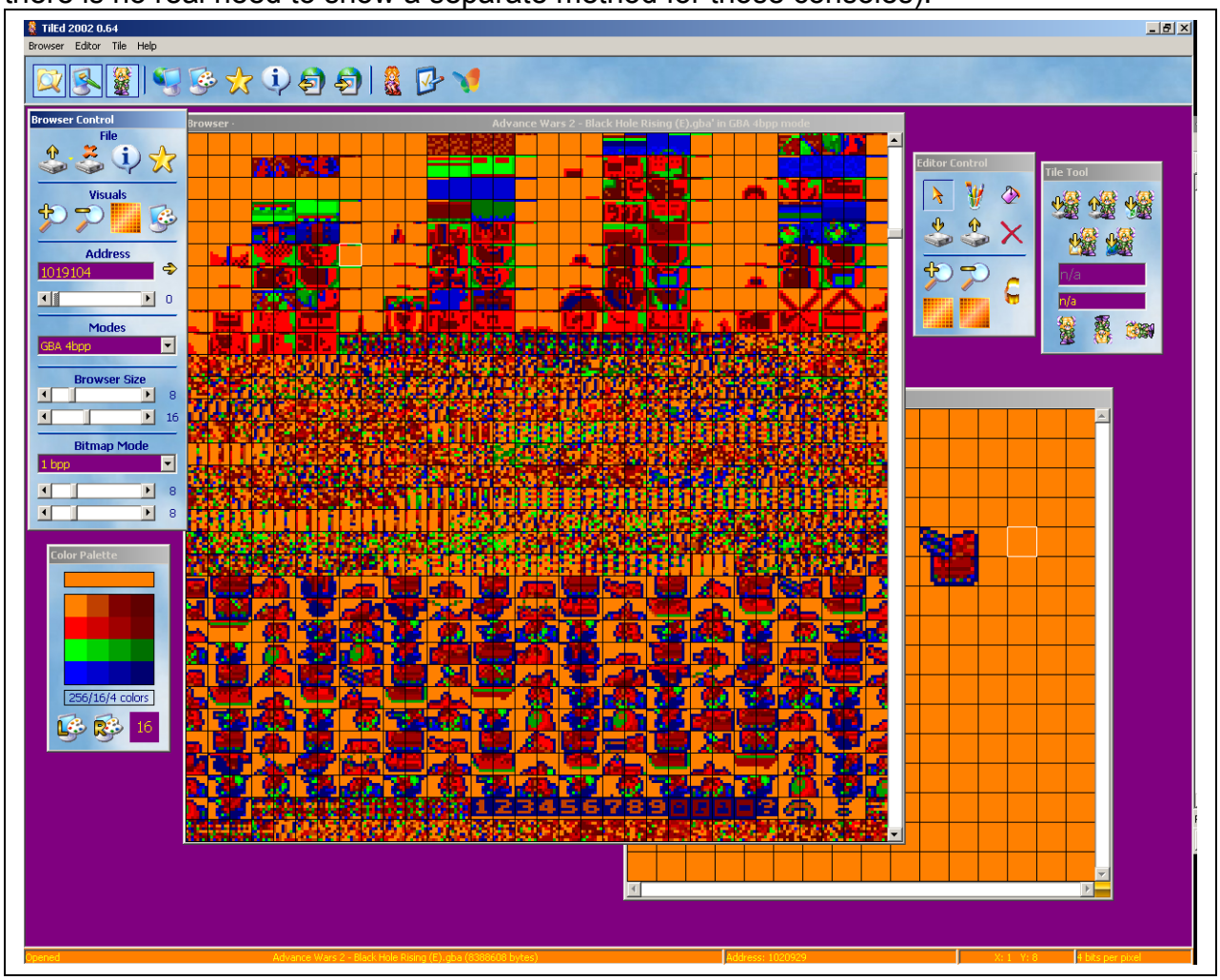
levels).
This can pose two problems, above I suggested getting the palette, reversing it and then searching for it in the rom. Depending on the game that uses a palette such as this a few things can happen, some code can be placed in the middle of the palette which will naturally disrupt your search or a unused value can be put in and then the game is told to change it during operation.
In such cases search for a shorter string of characters that come before it and you will usually be OK.

2d bitmaps

For various reasons 2d tiles are not suited for a given application (photos often have millions of colours which the palette hardware can not support) so each pixel is given a colour by the numbers it is made of. Bitmaps are usually used for whole images (backgrounds especially) or very highly coloured images (a palette is the “key” to the paint by numbers and can often pose a limit when dealing with photographs with millions of different colours in the same image).
See http://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm#Graphics%20Hardware%20Overview for more on the various modes and their limitations.

Tile editing proper.

The concepts of tiles and palettes have now been explained but the concept/theory and the reality are not always the same. For tile editing to be explained the best way is to probably to jump on in, here is a typical shot of a tile editor (the game being edited is Advance Wars 2 (Euro version) for the GBA, the DS/GC and Wii are much the same so there is no real need to show a separate method for those consoles):

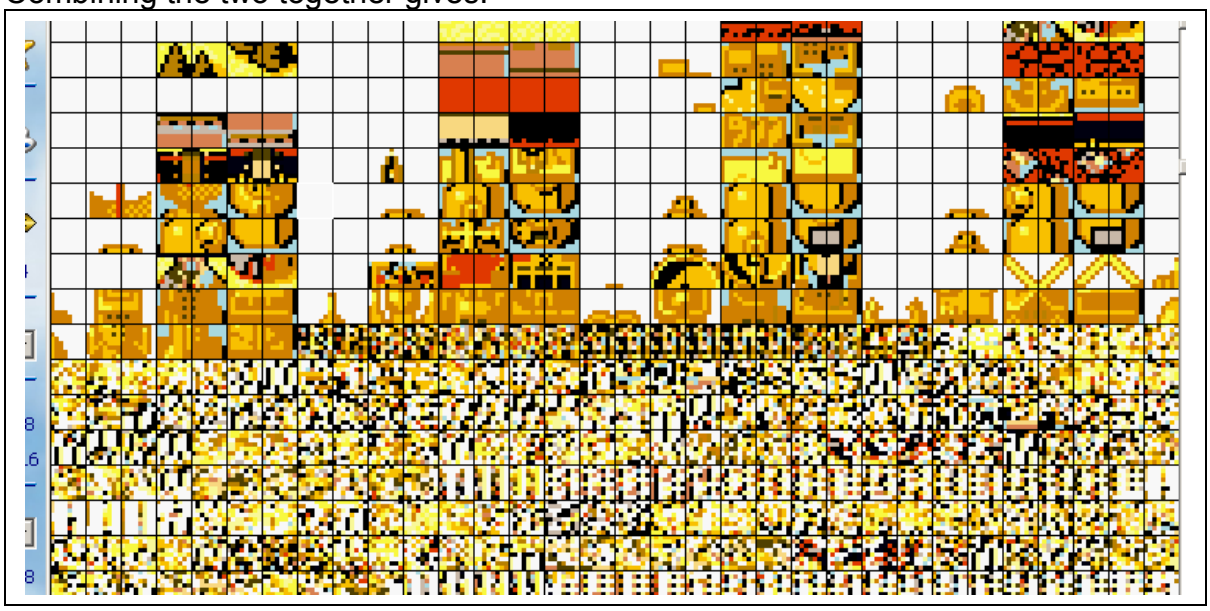


You are probably thinking two things right now:
1) That sure looks like game imagery but the colour is way off.
2) What is up with the tiles at the top of the screen?, they are leaking into the adjacent tile.

The reply:
1)The reason for the paint by numbers analogy now becomes clear, games use a palette to determine the colour to apply to a pixel and these may indeed change throughout the game (Advance Wars 2 in the picture above actually uses a change in palette to differentiate between different players characters/troops). Palette editing is covered in much more detail below.
To this end you can either make you own for the purposes of the exercise or grab one from an emulator's savestate/the ROM. TileEd2002 supports VBA save states (rename the sg? extension to .sgm first though). There may be multiple palettes per area and the GBA has 2 going on at any one time (one for the background and one for the sprites). The DS has 4 palettes active at any one time (background and sprites for each screen).

2) The tile editor is not smart: it just reads the data from the given address and displays it as a picture.
To make it look correct use the little slider bar under the address box so as to tell the tile editor to start from an address a bit further on, it will be obvious when you succeed.

Combining the two together gives:



TileEd2002 has basic facilities for picture editing inbuilt which are probably suited to most tasks including "text" editing and basic sprite/background editing.

While I said colours are assigned by the palette, if you should change one of the "colours" in a tile editor that change will be reflected in the final ROM (remember to think of the image you see in a tile editor as a paint by numbers affair and the palette responsible for the colours to go with the numbers)

Notes on tile editing:
Where you see movement on a game the chances are you are you witnessing multiple separate tiles being displayed (sometimes a single tile is flipped or rotated but that does not happen so often).

TileED2002 is hard coded to only look for files of certain extension, this can mess up things with the DS files so to get around this simply type *.* in the box to view the files.
It was already mentioned but if you are dealing with the DS/GC/Wii file system and have broken it apart then you may not care to open several hundred files to scan them so to make things quicker you can put all the files into an archive with no compression and then open that in the tile editor

For things like the title screen and menus there may be repeated sections where the same colour/"tile" is used. As a form of compression when the data is read from the game itself the code will be told to repeat sections, this usually entails an assembly hack but it is usually a simple one (run a tracing session and adjust the commands accordingly). Some take this a step further and assign a single pixel to represent a background colour for the entire screen.

3d graphics.

The DS, GC and wii have 3d hardware and it would take someone very foolhardy not to use it and try to do it all in software*.

The following section applies mainly to the DS but the concepts are common to 3d in general (in fact some concepts introduced are too resource intensive for the DS and do no exist on it). The links dealing the various systems have both information on the 3d implementations of those systems and some of the formats used.

*some games (and the concept is used a great deal on the GBA) may use pseudo 3d techniques in an attempt to fool the viewer into thinking something is 3d when in practice it is nothing but prerendered backgrounds and sprites. Dungeon crawling games like the megadrive/genesis title shining in the darkness provide a good example.

The easiest type of hack for this sort of thing is a nitrorom file system hack where one 3d section is replaced with another. This is followed by internal file system hacks where the pointers are changed to use different characters.
Example of the latter for Soma Bringer on the DS.
http://gbatemp.net/index.php?showtopic=109587
Similar techniques have been done for the Wii and Super Smash Brothers Brawl:
http://gbatemp.net/index.php?showtopic=97324&st=0&start=0

3D basics.

In short the 3d hardware creates/allows the creation of a mathematical model of a situation (based on 3d coordinates often with different origins) and then generates (the technical term is renders) the according 2d image while the usual scaling, rotation and translation methods or movement occur to the objects depicted. Contrast this to a sprite which is totally defined (albeit at a fairly low resolution*) and relies on the art team to create enough sprites and the hardware and the programming to switch sprites fast enough to give the illusion of movement.

*there is a technique that is essentially a hybrid of old style tile editing and 3d concepts called vector graphics where a mathematical model of a 2d "sprite" is made.

From the ground up.
Every schoolboy is taught how to draw a 2d graph and more importantly the idea of

coordinate systems and it is assumed you are aware of this (http://chortle.ccsu.edu/VectorLessons/vectorIndex.html has some stuff if you want a refresher). 3D just adds a third dimension (usually called z in the beginning).

A secondary concept is that of multiple points of origin.
Here another set of coordinates will be made and the the origin of those coordinates will be referenced from another set of coordinates, the main set will be called the/a parent and the set that is referenced from the parent will be called the/a child. There can be multiple children (what fun is a game with only one character) and children can have children (even if it is part of a model the ability to simply move an arm/leg/wheel/claw by itself is one worth having).
Such a concept is useful as it allows the programmer to move an object defined by the child without having to perform a lot of operations. If the concept still eludes you think back to the sprites of 2d and then what would have to happen if every pixel that made up a sprite had to be moved with individual commands as opposed to the simple move obj to x,y.
At this point it is worth mentioning the term coordinates when talking about points of reference for shapes will tend to be replaced with vertices (vertices are the “corners” of objects and it makes more sense to talk about those).

The end of coordinate system.
Trigonometry provides a useful method by which to replace some aspects of the coordinate system and simplify (in as much as there are fewer operations) some maths. Instead of coordinates an angle to the origin is defined and a length of a line is defined from there. Conversion is simple matter and can be done using trigonometry and Pythagoras' theorem (for a right angled triangle the square of the hypotenuse (the longest side) is the sum of the squares of the remaining sides). Coordinates are still likely to be used for storage but this allows for simplification of some of the maths (rather than generating a new matrix with which to multiply another you can multiply an existing matrix by a number (generated using this coordinate method) and then use the newly devised matrix.

Terminology.
Point. A position in space.

Line. An infinite path that intersects two points.

Line segment. What is usually called a line, It is the visualisation of the direct path between two points and bounded by those two points.

Alpha- used in blending colours and allowing translucency through to transparency of objects. Often incorrectly assumed to just be transparency.

Vector: can be used to show a displacement or a direction (for something like light), note that although it has a magnitude and a direction it has no position. Think of vectors more as instructions than objects.

Viewport. The camera position.

Axis (plural axes). The w,u,v are usually designed relative to the screen rather that the “world” the 3d hardware is depicting. The w-axis is like an arrow stuck into screen (normal to the plane of the screen in mathematical parlance).

Of course not everything has only a few points of reference. For example a cube has 8 points of reference* (each of the corners) but more elaborate constructions can be made with multiple cubes or different shapes. Even for 8 corners though the maths can get hectic if done individually.
This necessitates matrix maths or matrices. It is an area of maths that many dislike but the advantages it confers are too great to ignore.

*there are a couple of 3d concepts. One of which is that of skeletal 3d where a skeleton is made (think wireframe) and then the “muscles” are added on the outside almost making each bar of the skeleton a parent for the “muscles”. Others deal exclusively with the corners of the muscles (or larger constructions from them) with the edges generated.

There are still 3 primary operations that can be performed:
translation
The movement of an item

scaling
The change in size of an item

rotation.
A point is taken and the coordinates for the points of reference are rotated about that point. While rotation about a point is possible it is usually the axes that are used.

Note when it comes to maths here (trigonometry and the like) fractional numbers will obviously be present. The consoles themselves will be detailed but the DS uses fixed point, what value is used depends on the type of 3d operation being used.

Basic matrix maths.
Note this is sometimes called vector maths depending on your location in the world.
There are many guides in addition to what follows.
http://chortle.ccsu.edu/VectorLessons/vectorIndex.html
It is mentioned for when changing the shape of things the matrix format allows for “simpler” ways of changing things.

Matrices.
They are used for definition of location coordinates and for the operations that can be done upon them. The above link deals with the maths (from basics through to advanced) but usually scaling is done by multiplying a value with a matrix and translations are done by matrix addition. Elementary rotation (about axes) can be performed using a certain set of matrices but more advanced rotation can be performed by combining them. As the coordinates are defined by a single matrix it is also possible to scale, rotate and translate with a single operation.

Translations.
The simplest form. Here a like matrix is added to the coordinate matrix and the hardware then interprets this.

Scaling.
Here a matrix will either be multiplied by a single number (normally if one of the reference points is the origin: 0 multiplied by any number is 0) or another matrix is used.
Using another matrix is useful as it aids scaling in a given direction.

Rotation, The most complex of the options. Most of the time it is performed about an axis (one of the major reasons why having the ability to define a child is useful)

3d files themselves.
There are multiple things to consider when dealing with 3d.
These are
The objects.
What the object is probably evident by the name. The section defines the shape of the article that the 3d engine is to represent. There are many methods by which to to do this Skeleton. Usually an animation term. Here a section of line sections are defined (the bones if you will) and the skin is generated using the line section and/or it's endpoints as points of reference. This method makes extensive use of parent and children concept.

The other side of this (and one that is used by the DS) is to define only the skin and then animate those points responsible for it.

A further method is similar to the vector model of 2d image making. Here equations are formed to represent an item rather than a series of points, it has the advantage of being able to make a truly smooth curve. Usage of this method is rare however owing to the tendency for lots of calculations to be needed for little gain (by increasing the polygon count it is possible to reduce imperfections seen by the viewer to scales where it is nearly impossible to tell).

A note on solid and shell 3d modelling. Solid defines a volume of space (quite useful for calculations and so used often in computer aided design), shell merely defines the outside of the item (why you sometimes see “air” or can see through walls if get into an interesting position in a game). As “seeing through” things implies most games are shell based.

Their textures
These are the palettes of of the 3d world. Occasionally they do not exist and the plain colours (called material colours) of an object are used. Usually textures are image files or some form but they can be mathematically generated too.

Light
Light is a fundamental concept in physics and the maths involved can get very complex (http://www.rp-photonics.com/encyclopedia.html with a random example: http://www.rp-photonics.com/hermite_gaussian_modes.html) but the consoles make use of simplified models. Light is no good without a source and the three main sources are spherical (think ball of light), spotlight (think directional light source) and laser type (a direct pinpoint of light that does not “widen” over distance (lasers actually do but this is an example of simplification. A further example of simplification is that even though the sun is a sphere the planet is far enough away that is may as well be a laser).
Interactions of various light sources is the usual area of interest, it gets more complex once colours of the light are involved too.
Fog is an important concept when dealing with light. The usual purpose is either to reduce light intensity or prevent something from appearing (the so called draw distance can determined by this or something more local).

Reflection of light. Two main types exist, they are known as specular and diffuse. Specular is similar to that of a mirror while diffuse is more the colour you “see” things as and is associated with rough surfaces(shine a light at a red ball on a table/wall and the light while

not a true reflection the table/wall will have a red coloured light).
While both are governed by the usual laws of reflection specular is enough to model. Rough surfaces will have light hit it at all sorts of angles which makes calculating the reflection difficult, this means an approximation is the typical method of calculating diffuse reflection.
For more see http://www.glenbrook.k12.il.us/gbssci/phys/class/refln/u13l1d.html

DS 3d basics

On the DS the nitroSDK format used for storing is NSBMD, when working with it developers use the so called nitro intermediate file (given the extension imd) which is an xml based format with plugins for many of the common 3d editing applications. However it is not especially common outside of the first party games and developers owned by Nintendo. Regardless of the format though they all work with the 3d hardware:
http://nocash.emubase.de/gbatek.htm#ds3dvideo
Using this and knowing that 3d is already likely to be drain (and obscuring the value is not likely*) it is possible to determine what a 3d format is much like you can for 2d and formats in general.

* While 3d is resource draining there is usually some compression (typically LZ) involved.

GBAtek is pretty good for the 3d internals and well worth a read if you plan on doing any serious work (the DS 3d is fairly simple so you can usually see how things match up or use a bit of brute force to aid you in that area). While GBAtek is the suggested source for 3d information the basics will be covered.
The following will be a bullet point format with a bit of explanation and workarounds if needs be.

Note the screen coordinates for the 3d system are different to that of the 2d hardware, the origin (0,0) is at the bottom left (like a basic graph) as opposed to the top left.

The 3d hardware has a 32 bit register known as DISP3DCNT mapped to memory location 4000060 hex. While not the only thing responsible for the 3d hardware it plays a very important role and is responsible for most of the major abilities of the 3d hardware. There are many functions inbuilt to the 3d hardware (analogous to SWIs rather than opcodes)
http://nocash.emubase.de/gbatek.htm#ds3diomap
The 3d hardware is governed by the ARM7 so clockspeed for cycles is 33.51MHz

4 axes are used for the DS, x,y,z and w (w is an axis that at all times is normal to the plane of the screen (if you put the DS on the floor the w axis would be straight up and down). The w axis helps simplify maths when x,y and z are not aligned with the “expected (think standard graph)” x,y and z axis positions and provides a destination for the reflected light.

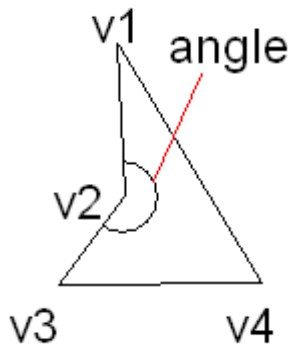
Matrix maths is used, all matrices are 4 by 4 in the hardware although simplifications are made for matrices that can use it (the remaining values are set to 1 or 0 depending of the matrix used).
See http://nocash.emubase.de/gbatek.htm#ds3dmatrixtypes

The DS has the ability to change viewport (where the “camera” is located), you can often do this with cheats alone rather than editing the rom (pokemon cheats provide a good

example).

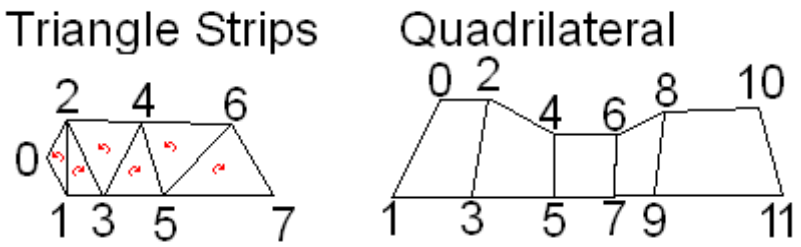
DS 3d is polygon based with the edges calculated from the values for the vertices (“corners” if you prefer) and much like the arrangement of hex digits for tiles the arrangement of vertices is known to the 3d hardware.
The DS 3d polygons are either 3 or 4 sided (triangles or quadrilaterals) and there are forbidden arrangements. You can also make line segments by giving two identical vertices to a triangle.

The forbidden arrangements are polygons with internal angles over 180 degrees and those with ”crossed sides” (examples adapted from gbatek):



Note Both shapes can be made with triangles.

Several polygons (they have to be all triangles or all quadrilaterals, they can not be mixed*) are able to be joined together into “strips” (which then get made into things like http://www.cubeecraft.com/). Single polygons are usually used for levels (some are 2d though with 3d “sprites”) while strips are used to make characters and other objects the player is likely to see.
Owing to shared vertices in the strips is it then possible to make a shape. This is also where it gets a bit complex.
Single shapes label vertices in an anticlockwise manner (as seen in the “greater than 180 degrees shape forbidden) but every other shape but the arrangement of vertices for strips are different depending on the type of primitives used.
*nothing stopping you from making a very small edge on a quadrilaterals to use in place of a triangle.
Again GBAtek provides the suggested reading here
http://nocash.emubase.de/gbatek.htm#ds3dpolygondefinitionsbyvertices
In short triangle strips count anticlockwise and clockwise alternately while quadrilateral based strips alternate “up and down” (as gbatek says though rotation changes up and down so do not expect it to always be elementary)
Examples:



Textures

Warning changing the texture format can have unintended consequences such as filling up the (limited) memory for textures. Make sure to check the memory beforehand.

Textures are optional (material colour is available) and there are 7 formats (that the DS can use, there can be any number of formats used by developers). Alpha blending is possible if using the correct texture format.
Texels are to textures what pixels are to still images, the distinction arises as the pixels are usually the final work while the texel will usually have more work done upon it before being turned into a screen pixel and it will form part of a greater work while pixels are the material from which graphics are made.
DS texture hardware:
http://nocash.emubase.de/gbatek.htm#ds3dtextureattributes

The number beforehand is the value of the bits 26-28 in the set texture parameters command.
0 No Texture (material colour) Note use of this method is the only way to use only the material colour. All others will be a blend of the texture and material.
1 A3I5 Translucent Texture
2 4-Color Palette Texture
3 16-Color Palette Texture
4 256-Color Palette Texture
5 4x4-Texel Compressed Texture
6 A5I3 Translucent Texture
7 Direct Colour Texture (this would be textures where the values of the texels are the colours themselves like the bitmap modes of 2d)
The hardware has provisions to flip and repeat as well as the ability to repeat textures too.
Texture formats 1 and 7 both use 5 bit alpha values but format 1 expands the 3 bit value to 5 in the following manner
$Alpha_{final} = (Alpha_{3bit}*4)+(Alpha_{3bit}/2)$. Naturally it is not as precise as format 7 but it affords more colours, round down if you need to interact with other alpha elsewhere.

For the layout of the textures you are directed to
http://nocash.emubase.de/gbatek.htm#ds3dtextureformats
Remember to account for endian-ness (which courtesy of some of the formats gets quite difficult).

Alpha blending is possible.
Values for alpha ultimately end up between 0 and 31 (decimal), 0 is transparent while 31 is solid colour.

There are 4 light sources available (given the numbers 0 through 3) and these should have different vectors. All light is of the parallel “laser beam” type of light. Spherical and cone are not available.

Reflection is the only effect able to be achieved with light (no shadows).

Shadows are separate concepts and no shadow is cast by light. Light colours use the same 5:5:5 format used by the 2d palettes.
Shadows can be dynamically calculated but for fixed sections they are generally calculated by the developer. In simplistic terms they are another form of alpha-blending.

Fog.

Fog is used for many reasons including hiding the edge of the 3d object if the level does not (clip plane is the technical term) and to change the interactions of objects with the light. Interestingly you can disable the effects of fog on a per item basis (shining sword in the dark, “torch” in the distance).
Much like light you can set the colour so while grey/black may be the most common there are options.
For more see http://nocash.emubase.de/gbatek.htm#ds3dtoonedgefog
It is a memory location similar to the OAM so similar techniques will need to be applied.

Along with fog you can also create a wireframe like effect by setting the edge colour assuming it is enabled in DISP3DCNT (bit 5, 1=enabled, 0=disabled).

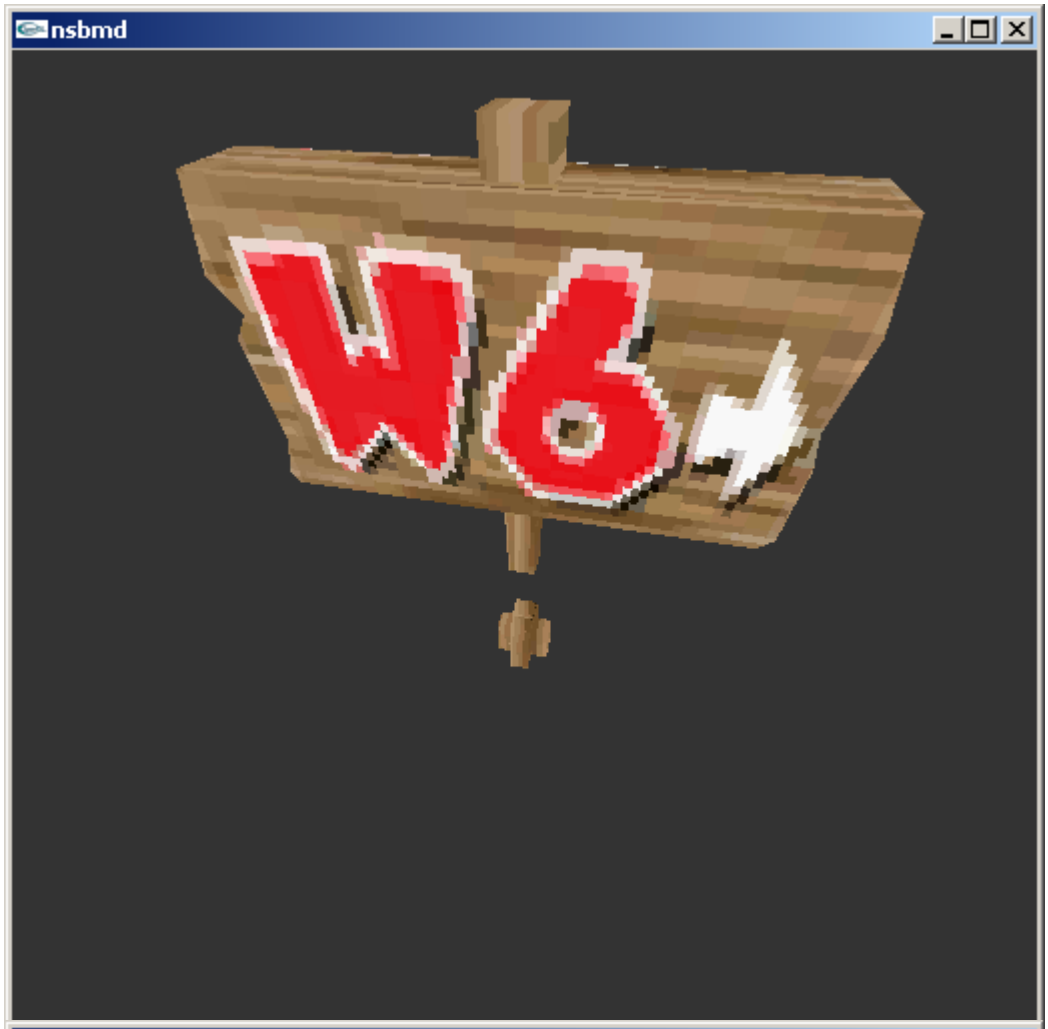
After all is said and done the 3d image is rendered and either sent to the 2d engine for display as layer BG0 or to the capture engine for additional work. An aside, you can also set the rear plane to transparent by setting the alpha to 0 and you also have the option of the Window Feature where different priorities can be set to different areas of the screen.

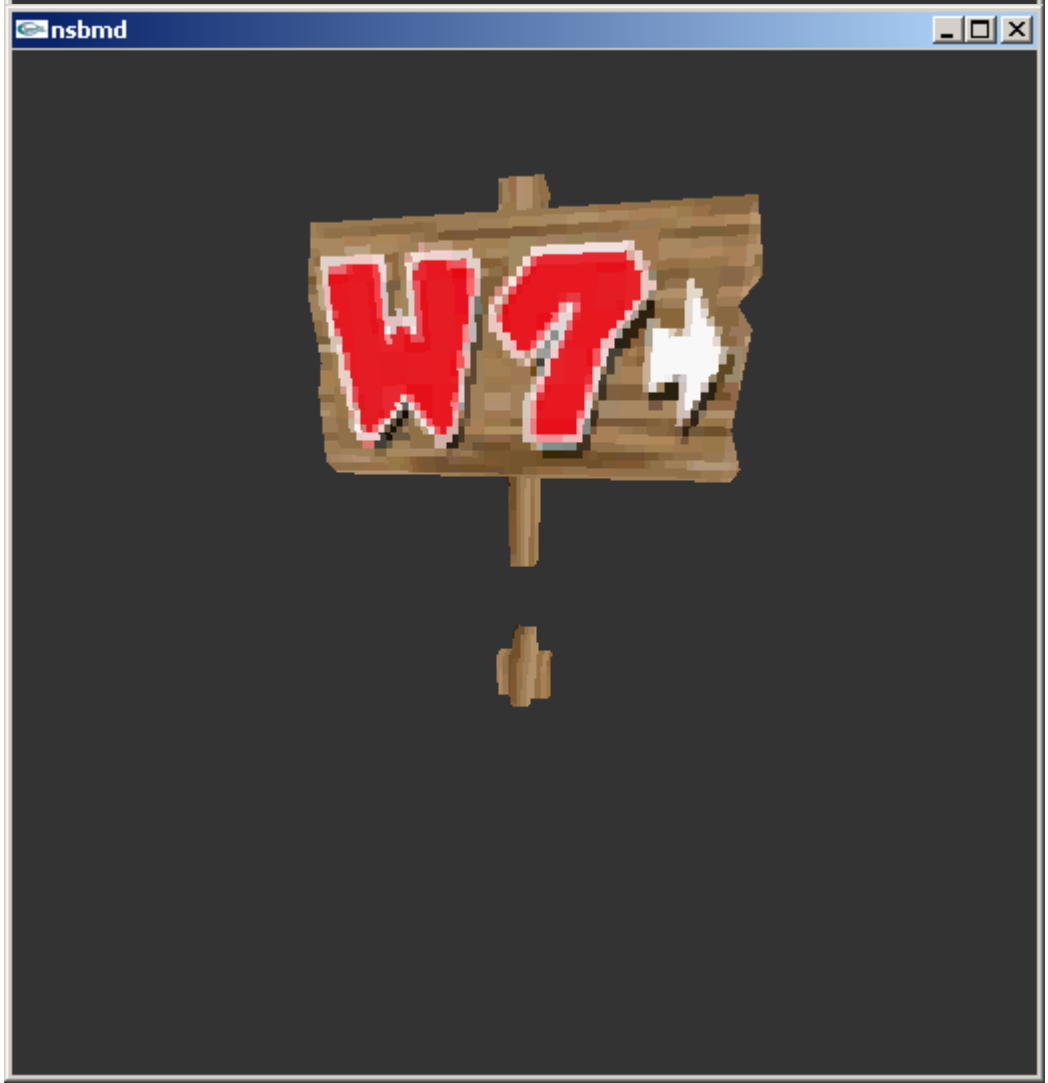
Hacking 3d formats.
A while back there was a metroid viewer called Dsgraph from mike260 (http://gchack.free.fr/DS/utilsDS.htm) and some pokemon hackers also did some work (http://pokeguide.filb.de/ and some more information should be available from the site listed in the pokemon section further up)

Bear in mind 3d is fairly big and usually fairly compressible (bios compatible LZ mainly) so it is not likely to be simple hack. Running a tracing session or viewing the memory sections responsible and working backwards is the best route for unknown 3d formats.

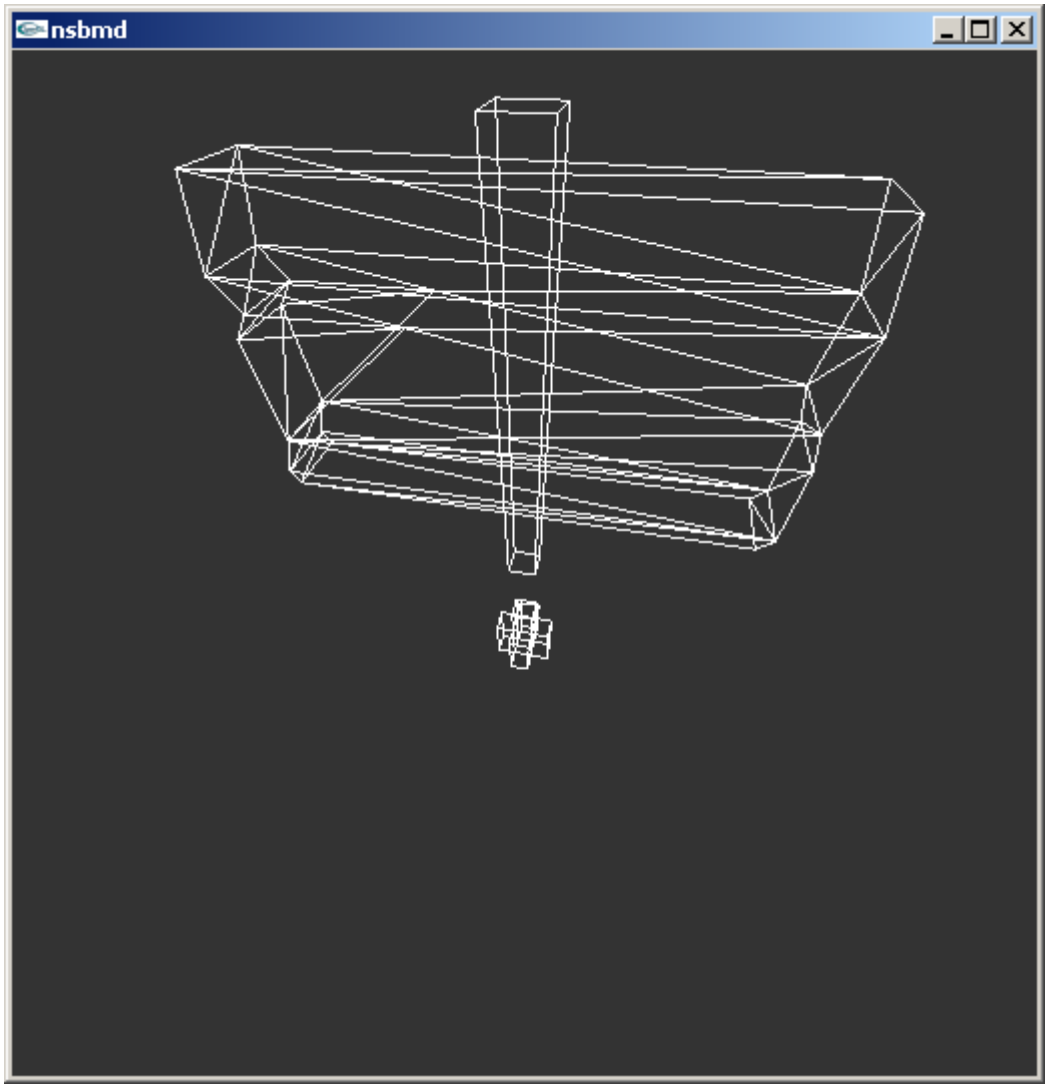
Hacking NSMBD.
kiwi.DS also found time to reverse engineer NSBMD files, this is the nitroSDK format for 3d like SDAT is to sound but nowhere near as common (mainly first party or captive devs hence the maths and 3d lesson).
http://kiwi.ds.googlepages.com/nsbmd.html
http://tahaxan.arcnor.com/index.php?option=com_smf&Itemid=29&topic=34.45
http://tahaxan.arcnor.com/index.php?option=com_smf&Itemid=29&topic=39.15

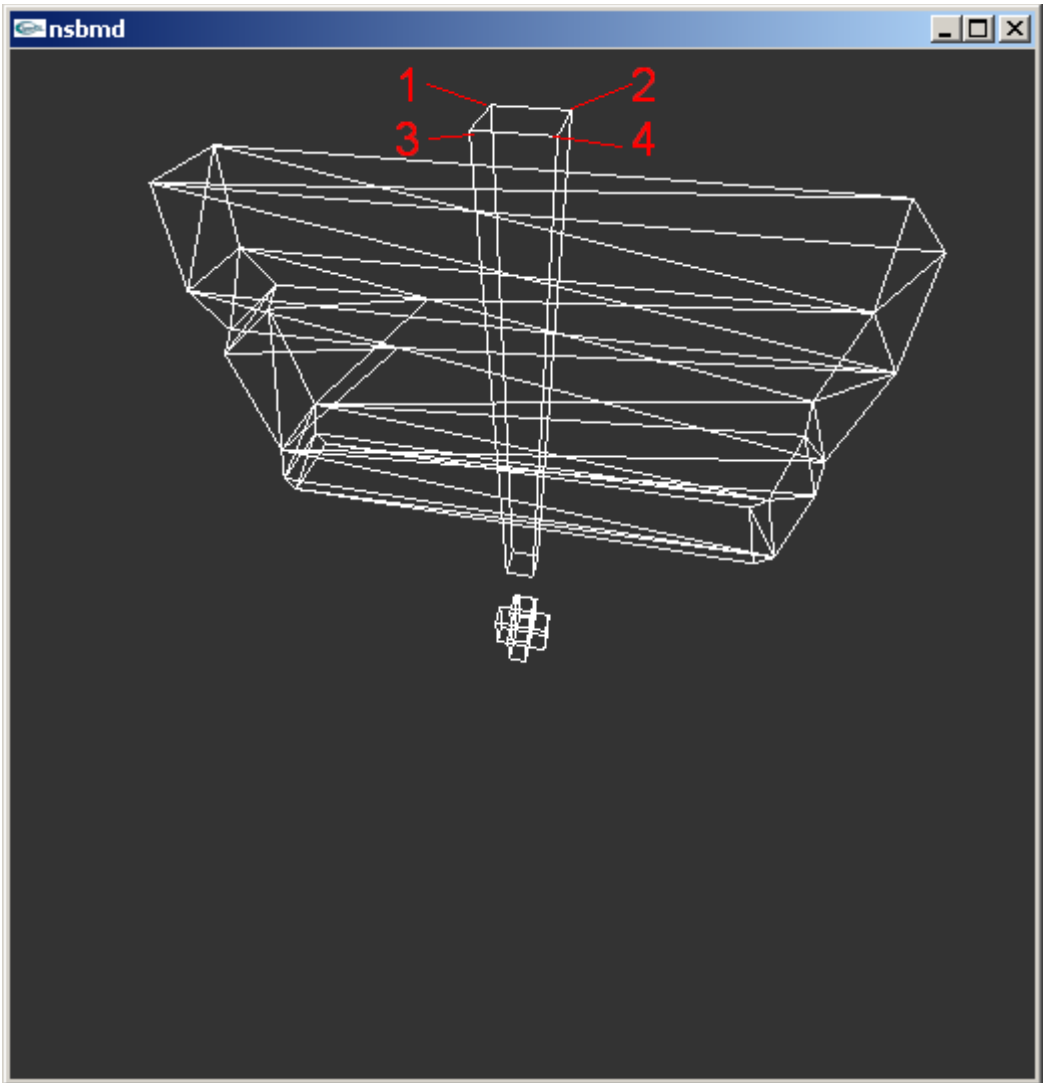
Aim, change polygons of w6_sign.nsbmd found in the data\enemy\ directory of New Super Mario Brothers (European release).
Original, picture made with kiwi.ds's NSBMD tool. Note if you spin the sign around there is a W7 picture on the back (see the notch on the left is now at the right and vice versa for the other side). In the game you do not see this so it can be done (which is presumably why the based is not “fixed” to the pole), it is a kind of throwback to the multiple palettes for the same sprite or the pointer driven compression mentioned in the text section below and worth noting. Even on modern systems (especially games and CAD) wasting 3d resources is not considered good coding/use of resources.


nsbmd
W6


nsbmd
W7

Aim, Change the “height” of one of pole holding up of the sign.
Wireframe shot





Now the file has been analysed it is a matter of finding the vertices numbered (note the numbers are arbitrary at this time) on the picture above and editing their values.
Usefully all those points should share a point in common with their neighbour and all should share the same “height” which in all likelihood is the largest value in the whole image,

[To be finished]

---

text

One of the fundamentals of rom hacking.
The written word or text is a dominant part of human society and a such games would not be much without it. Sometimes games display text as part of a picture, however lots of text stored as a picture would mean lots of space required so such techniques are usually reserved for puzzle games and other low text count games and other common places (if a building in a game has text on it and/or it looks different to the normal text the chances are

it is graphical.
So for the longest time text has been generated on the fly from data stored within a rom and the hacking of such text is of great importance to hackers and the hacks they make.

tables

Many years ago you may have made a "code" along the lines of 01=a, 02=b, 03=c and so on (if you were especially drawn to secrecy you may have set the numbers in reverse z=1 y=2 x=3 etc or added some to the base number (a=10, b=11 etc). Nothing has changed now computers are there other than hexadecimal is used and normally more than just 2 digits are used (26 characters in the current roman alphabet used by the English language x 2 for 2 cases, 10 numbers, some punctuation and a few other terms and it starts adding up. Account for French, Spanish, Slavic, Asian languages, Greek, ancient Greek and you end up with thousands of characters.

Speaking of computers most of the time a conversion is done from the raw hex/binary to what is known as ASCII (American Standard Code for Information Interchange, definition here http://www.lookuptables.com/asciitables.html). To keep this current though ASCII is an 8 bit code which only affords 256 combinations thereby eliminating several languages (Russian, Greek (to a limited extent), Chinese (both simplified and definitely Traditional), Korean, Japanese, Arabic, Thai and the list goes on and on).
To this end ASCII is slowly being replaced by Unicode (Unicode is 16 bit aka 65536 combinations which should be enough for most people, Traditional Chinese (generally considered the language with largest number of characters) only really makes use of 20,000 symbols although many more are defined (have a look for the Kangxi Dictionary if you are interested) and they are shared somewhat between other Asian languages), if you leave the first the 2 bytes blank (I.E. 0033) in a Unicode system you get back to ASCII* too (0033 becomes 33 as most text editors ignore 00's)).

*It will be mentioned (even used in an example) later but only the first 128 combinations are used in ASCII with the last 128 being called the extended set. Over the course of history many additions have been proposed to ASCII but none have become the accepted standard. One such set however became the de facto standard for a lot of things (http://www.cdrummond.qc.ca/cegep/informat/Professeurs/Alain/files/ascii.htm), this is not the same as the "extended" set used in unicode.

One and two byte (8 and 16 bit) character systems were mentioned above now is probably a good time to explain it, while you can assign one byte to a value that only gives 256 combinations. While that may be more than enough for every European language (probably including Greek) with enough space left over for a few symbols it is not nearly enough for say Japanese (a language pretty much everyone in ROM hacking for any length of time will end up dealing with).

Japanese from the perspective of rom hacking.
Go go Japanese lesson. It is generally a good idea for all the members of a Japanese translation ROM hacking team (translation is so much nicer if it is done as a team effort) to know what follows at least:
Japanese uses four methods to display characters: Hiragana, Katakana, Kanji and Romaji. The Hiragana and Katakana are collectively known as the kana and are used for everyday words mainly (Hiragana are typically reserved for Japanese words with the Katakana used for foreign sounding/originating ones).
Kanji are the elaborate symbols and are also used (although there are unique symbols not

in said languages and the meaning is not always common across them) in several other Asian languages (the fact simplified Chinese translations of Japanese games are very common is not entirely unrelated to this fact). They are considered one of the more complex aspects* of the Japanese language (both by native speakers and those learning it) and there are cases where they are not used (certain aspects of medical terminology is a good example) and there are also pronunciation guides included in some games (called furigana, a good example is Zelda, Phantom hourglass)

*This is occasionally referred to in games or indeed it is played upon in some text (characters who are not so good at the language will avoid using them to the point where the phrasing is awkward or be confused by them. Naturally to those without knowledge of Japanese it means nothing and is one of the first things to be dropped in translation for a mainstream audience, the general audience of a translation hack may well have some knowledge of Japanese (even if they do not know it well enough to play the original) and thus this can be accommodated in a hack or even restored for a mainstream/commercial release. Plays on words are common in Japanese (much like other languages) and is yet another reason why machine translation is not a wise idea at all and why it can be so difficult to do a good translation.

Romaji are Japanese words spelt phonetically using the Roman alphabet (what this document is in) and are not in strictest sense Japanese characters but you will see Japanese written this way on the internet as it is easier to type, you will probably never see it in a ROM mind (save perhaps the names of files in Japanese games on the systems the use filesystems).
There are also multiple romaji conversions methods: some aid writing while others aid pronunciation but courtesy of the use of accents and such make it far harder to write on a computer.
Numbers in Japanese can be either Arabic numbers (0,1,2,3,4,5,6,7,8,9) or Japanese http://sp.cis.iwate-u.ac.jp/sp/lesson/j/doc/numbers.html .

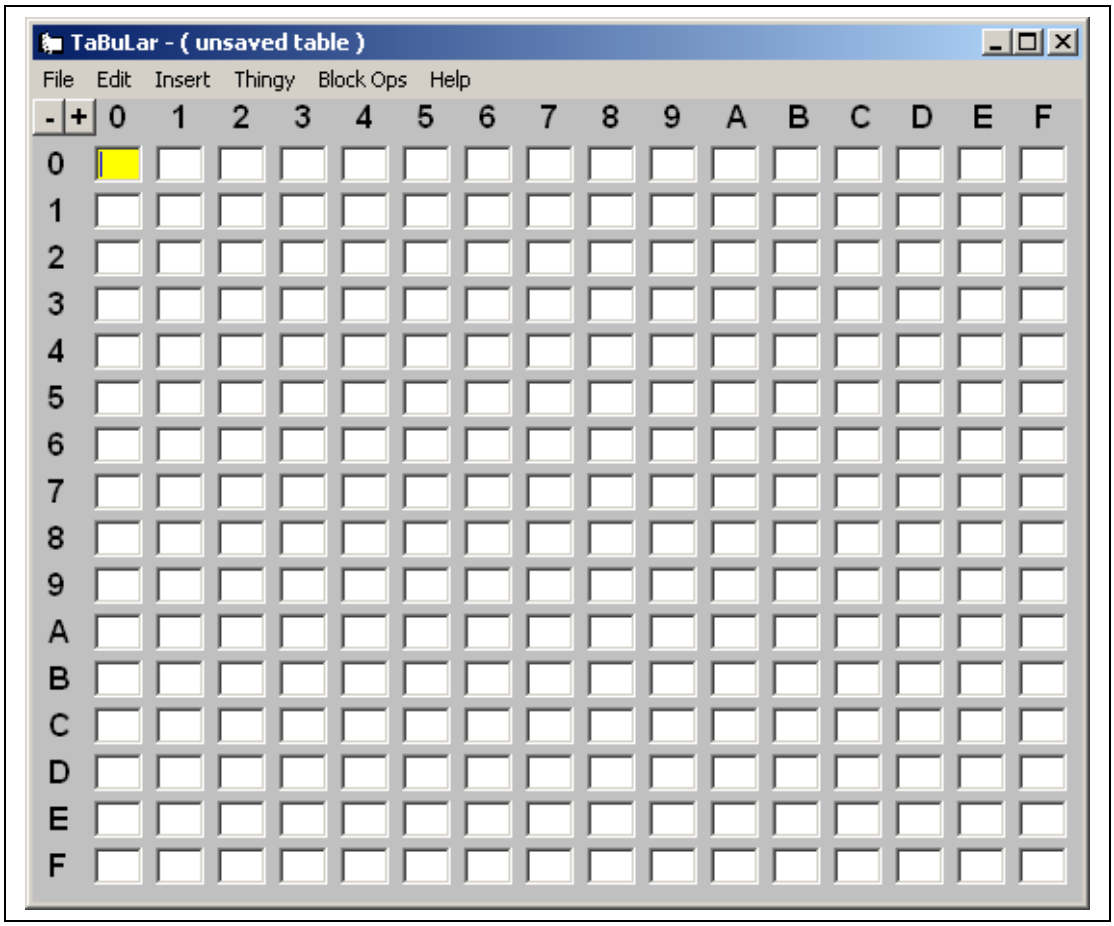
In the fonts section below I said that characters are defined and then pictured made and referenced by the ROM during usage (to go with the multiple tables and sometimes just for the hell of it there may be multiple character sets (normal text and cutscene text for example).
Now the kana have but a few characters so that is nothing much but the Kanji number in the thousands, obviously these are not all going to be used on one game so only those that are needed are defined.
If you are lucky the Roman alphabet is included within the font (more than 50% of the time it is) but sometimes it is not. On these occasions it is wise to replace the Kanji with the Roman Characters and then go from there.
Obviously this lack of continuity between characters (sometimes they are ordered by amount of strokes but do not count on this) makes finding a table a pain and consequently dumping the script that much more difficult. Reasonably often the order they appear in the font is also the order they are defined but equally do not count on this (I speak about fonts later but they will be able to be viewed in your tile editor usually).

Table making

So you are probably thinking it is all well and good knowing that in ASCII 20 (hex) equates to a space and 33 (hex) equates to 3 and so on but you need some way of putting it all together and converting a lot of hex at once.

You will never guess what a table is.
Table is the term given to the list of numbers and their definitions. It is the most common term used by hackers but codepage, character encoding, character definitions, character sets, character scheme and several similar terms have been and are used in different areas of computing. Table and codepage should suffice if nearly all occasions.

Like many things in life and rom hacking it is not always that easy as many different file formats exist for tables. Perhaps the most common format (and the suggestion of this guide) is to use Thingy tables (they are supported just about everywhere and there is no real downside to them).
Tabular is a good way to make tables:
http://www.romhacking.net/?Category=8&Console=0&Game=0&Author=0&Recm=&Level=&utilsearch=++Go++&title=&dsearch=&page=utils&action=utillist



The accompanying TaBuLar.txt contains all the usage guides you will need and there is an example later on.
After you have your .tbl file created you can load it into Translhextion (under the script menu) and fire away.

Note there is nothing stopping a game from having more than one table.

It probably all sounds very simple and well it is very simple when you think about it.

Unfortunately as you may have gathered by now not all ROMs use ASCII/Unicode, some

like FF3 DS (Japanese) use Shift_JIS though (another common hex to letter system (especially in DS roms) and a good example of a 16bit one). Note also the even if something appears to be in a known encoding “extensions” and tweaks can be present.

Another thing to note is the one byte (or two bytes) is not always used to represent one letter: The classic example of this would be in Final Fantasy 2 (USA rom version, platform NES, Japanese numbering) where the common characters would have two bytes assigned to their name as opposed to the 6 or 7 it would take to spell it out every time (it also helps in games where you can choose characters names or you have other variables you want to display in the text):
0400 (hex)=Cecil
0401 (hex)=Kain
0402 (hex)=Rydia
0403 (hex)=Tellah
0404 (hex)=Edward
0405 (hex)=Rosa
0406 (hex)=Yang
0407 (hex)=Palom
0408 (hex)=Porom
0409 (hex)=Cid
040A (hex)=Edge
040B (hex)=FuSoYa
040C (hex)=Golbez
040D (hex)=Anna
This concept is known as dual tile encoding/multiple tile encoding (DTE/MTE*), implementing one tends to involve finding the text grabbing routine, finding the encoding setup and adding in your dual/multiple encoded text as an unused character (be it a control character/variable like bold, amount of money x item costs, a space or small or a normal character). Dealing with one can be a bit harder and will usually result in you having to use an emulator/debugger and some brute force to check what the character represents.
*The term is also used when speaking of fonts and the tendency of some games to have a word defined across multiple tiles. Most of the time the distinction should be obvious from the context of the post/document/conversation or defined if it it not.

Be warned though that when you see a “missing” section in your hex editor it can also be an indication of compression (testing and common sense is usually the best method to tell this).

Very often you will see shorthand/reference notation inside DS and GBA language files but for things like cost of hotel or number of coins needed (examples later). If you want a more modern example like the Final Fantasy 2 one above Advance Wars 2 on the GBA has a byte assigned to the Yes/No option.
Sidenote. If you are after Advance Wars info you could do far worse than look at http://forums.warsworldnews.com/viewforum.php?f=11

It is also worth mentioning at that this point ROMs do not need to have just one table, some ROMs are especially fun and have an 8bit table and a 16bit one. 12 and 20 bit encodings are extremely rare (goes to the alignment problem) but have been known.
Most character encoding methods are 8 or 16 bit but there are two important cases amongst them.
The two cases are

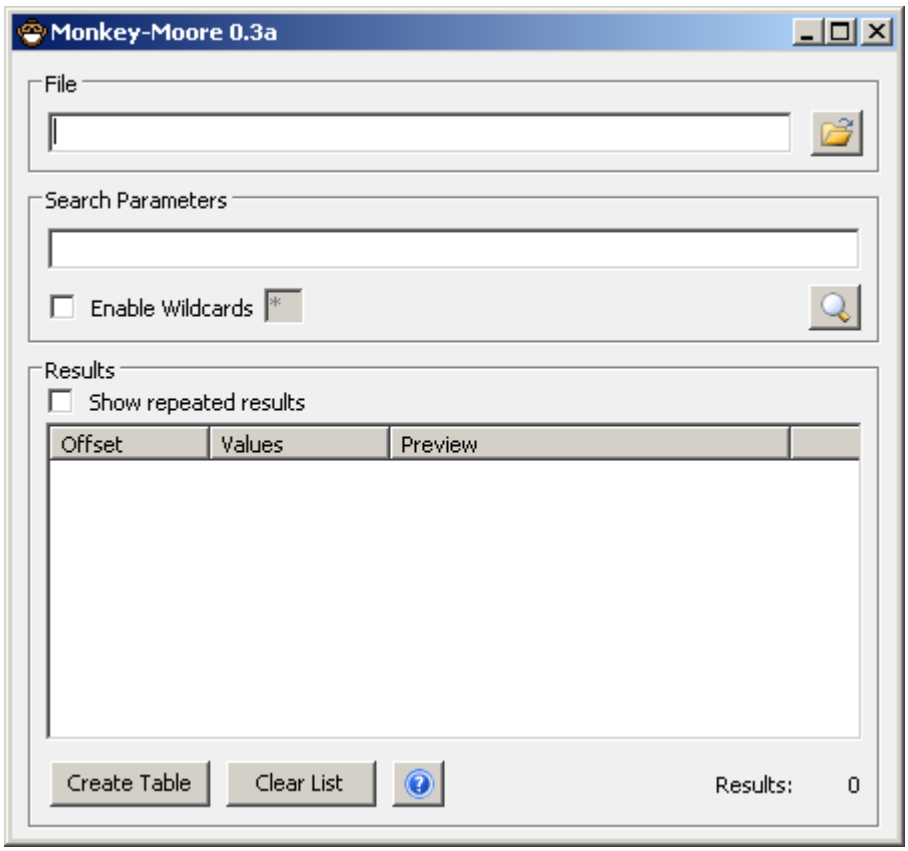
Half width, more of a font issue but some encodings have half a Japanese character in a single 8 bit section and the other half in another, this differs from conventional 16 bit in that half a 16 bit encoded character will tend to display nothing (or worse break the decoder). It is fairly rare these days (older consoles and older PCs before unicode and the like became standards) but worth knowing.
16 bit flag. Much like the most significant bit being used in signed characters it can be also be used to signify a 16 bit (actually 15 bit+1 signal) character. Very rare owing to the increased processing power needed for little gain in space.
Now a table then as you have likely gathered is a method of collecting all the conversion data and then applying it, unfortunately hex workshop does not support any tables (you can define your own character set but it is limited to single characters). So it is for this reason I linked Translhextion (Translhextion is a hex editor built with ROM hacking in mind unlike most others). Sometimes a 16 bit encoding will have an 8 bit “punctuation” (end of line, new line, end of section) too.

Most documents will kind of skip out how to find a table or explain what one is mutter a few times and assume you now have one but this document will detail and explain the basic methods. Most stem from earlier codebreaking techniques and are fairly similar to word puzzles:

Luck:
ASCII/Unicode/some other common type like shift_JIS will be used thus enabling you to simply and easily locate text. Unfortunately this is not that common, especially for GBA roms (Advance Wars and homebrew stuff like ROM loaders are all that really uses it). ASCII and Unicode are used frequently throughout the ROM and the files it contains as reference points/ “magic stamps”. As an aside even if a rom is based on ASCII/unicode some symbols (normally unique symbols/constructions) can appear and what said symbols are will have to be determined. Other quirks include such things as although roman characters appear in the Japanese portion of shiftJIS they are lacking from the game itself, this is more of a font problem than a table making problem but worth knowing about.

Relative searching:
Assuming the nice events described above do not happen most fonts are relative, that is to say the letter B is one value after A (case is important here) and C is 2 after A.

You can get a very useful tool called relative searcher that enables you to search for a phrase but do it relatively for example the word BAD
You would type this into the box and it would search for every value the has the second value one less than the first and the second two more, for such a short string you can expect multiple hits but with repeated searching and testing you can get the correct one. After you have the correct values you can then extrapolate the rest of the table or use some brute force tactics.
Many tools exist
http://www.romhacking.net/?category=&Platform=&game=&author=&os=&level=&perpage=20&page=utilities&utilsearch=Go&title=&desc=relative
crystaltile2, several of the hex editors and many others have the ability.
Here a relative search tool called monkey moore will be used.
http://www.romhacking.net/utils/513/
Screenshot of the basic application



A problem arises with Japanese, as you have already seen Japanese has a series of elaborate symbols called kanji, these have many ways by which to order them and more importantly there are many of them which means only those used in the game will tend to appear (or a select handful and whatever else is needed).
Other games may use an order than is not relative (for example the order of which the characters first appear in the game has been seen) or simply a random order. Whether this is done to prevent hacking is up for debate but the fact remains it happens and prevents relative searching.
See the example table finding below for a worked example of a basic relative search.

Difficult relative searches.
Sometimes the difficulty comes not from the game but the tool, a lot of relative search tools are 8 bit only and with games increasingly likely to use 16 bits per character..... Fortunately a relative search tool should provide a wildcard option. Simply add a wildcard for every other character, it also helps when the upper 8 bits may not be in a “logical” order.
Sometimes the upper case and lower case are not distinct sections but may come one after the other. The font is still relative but the “gap” is 2 per character, if the searching tools does not give a option you can usually game it by adjusting your input character string
Example assuming lower case upper case arrangement:
`(aAbBcCdD.....)`
`(abcdefgh.....)`
Searching to give “cab” would become a search for “bec” or the equivalent for a numerical based relative search.

Remember punctuation has no set order so relative searching will not necessarily give you the entire table. Other times symbols may be defined by never used in the game.

Extrapolation

Languages have various rules (there must be a vowel or “y” in every word for English) or the meaning/word can be inferred from the rest of the sentence/word itself. Likewise given that you can see the decoded text in a game you can simply copy the correct character across.

An example of extrapolation would in the European release of New Super Mario Brothers, while it used ASCII (technically it is Unicode but still) to display text for the most part we are now dealing with European characters (many European languages have characters like Æ, Ç, È, É, Ê, Ë, Ì ,Í, Î and Ï). In the strictest sense of definitions ASCII is only a seven bit system for describing numbers (in the initial table you will find on ASCII the character 7F as the last character (del) and in binary it comes out as a zero with seven ones after it: that is to say this character and none of the ones before it contain anything other than a zero in the highest bit) this of course leaves a whole other set of numbers able to be displayed.

This “other” set is called the extended set and there is in actuality no true extended set , a sort of standard one is used in Windows (and consequently 90% of the programs it uses). However this is not the case here.

Open up the course.bmg.efr file in your hex editor and scan down, at 214 (hex) in the ASCII window you will see L.e.s. .d.o.n.n...e.s., either from knowledge of French or from the game we know it should read L.e.s. .d.o.n.n.é.e.s so we look at the hex used to represent it: E9 from this we guess the E9 represents é, in the conventional extended table this would be 88 (OK so this was a bad example for anyone that knows the Unicode encoding method but it works to illustrate the point).

The other side of this is determining the sentence/section length and referring back to the game to determine what a section is (how many sections will have the same length).

A final note is to pay attention to what is said by a character. A good example can be found in Final Fantasy titles, the creatures known as moogles will often end a text section with kupo (or even better repeat it twice), using this as a starting point you can perform the various other table making techniques.

Distribution

Not a method per se but it can certainly help pinpoint certain values (think how many times the space key is used in a section of text compared to the other characters or how few the “x” character is).

Not all editors may have this feature but Hex Workshop has it under tools, you could always import data into a spreadsheet and then look too.

Corruption

You can get programs (or use your hex editor) to scramble your ROM thus enabling you to ultimately pinpoint what does what. (It is suggested you only use this as a last resort).

In the case of table making you can corrupt a section and then view it and correspond what is seen to the corrupted section.

Brute force

The name is taken from a password cracking method of trying every combination until you get it right.

Used in conjunction with corruption or if you luck out and find one of the text carrying files on the DS you can then proceed to replace the text with other values and then trying it out. I suggest counting upwards in hex with certain “break character” used every so often to enable you to tell what section you are in. In doing this you can determine what values are assigned to which character (often a good way if you find your values are non sequential or you need to determine Kanji values (not all Kanji are included in a ROM so they will probably not be in much of an order).
It is also a viable method when it comes to 16bit characters.
I often use this method to seek out control characters and use when I encounter dual tile encoding (one byte is used to represent two letters or even more).
A bit of advice GBA pointers are of the 08XXXXXX (hex) variety (I will explain this below) so doing a search for 08 and then picking the section with many close results (better yet many close results all equal in distance from one another) can help you find the pointer table in no time and as text and pointers are related (pointers “point” to text, no) you lose the need for corruption.

Related to Brute force is another cryptography originated method called chosen ciphertext and adaptive chosen ciphertext. As you should be aware by now game dynamically generate text, a rom hacker can then change the text section to a given string (in codebreaking this would be a known text section probably including all the characters or a carefully selected bunch of them but in rom hacking a repeated string or a string counting up (0102 0304 0506....) works best). In reality this is usually used at the table checking stage but it can be used when finding a table.
Chosen ciphertext is where a known text is encoded using an algorithm you wish to attack. The encrypted text is then checked against the original and things are determined from this.
Adaptive chosen ciphertext takes this a stage further and then from theories posited from analysis of the original original chosen ciphertext and tries them out (coincidences happen) or chooses a new text with which to fill in gaps from the first attempt.

<u>ASM</u>
The potential ability of ASM/assembly was detailed above and there is not much need to explain it again. While it can give you a table (simply reverse engineer the text decoding/display routine) it can also redefine the text for you (many an ASM hacker has made the game use ASCII/Unicode text purely to make life easy for all concerned).
Linked with assembly is font based table making, regardless of what a game uses it with be a fairly simple relationship between the decoder and the font (the order is most likely the same and especially for a font in the ram if you run a tile viewer over a memory dump).

<u>Regional differences</u>
Not a method but another potentially useful piece of knowledge/source of information.
Text will often be tweaked between regions, such changes (or indeed similarities) can be detected by a simple file compare.
Note it not uncommon for the encoding to change entirely between regions.
Not so useful for early stage translations (Japanese to something else) as there will usually be something else but for improvement hacks and total conversions as well as translations to the less commonly supported languages (which are usually done some time after the fact).

Dumping the text
After you have a table, have sorted decompression and whatever else you then have to distribute it. There are many ways but a tool known as romjuice will do if you lack the time

to edit it manually:
http://www.romhacking.net/utils/235/
Manually.
There are many schools of thought here, your translation team may not be rom hackers and so seeing flags and such in the code may not be ideal. Likewise using a hex editor however nice it may be in place of a text editor may not be ideal either. Many then code game specific editors which can get quite intricate if features like text viewing and the ability for the user to use control codes (bold, italic, colour and so forth) while others attempt to beat the text dump into a more workable shape.

Some thoughts

the “windows” method to start a new line is for the hexadecimal characters 0d0a to appear. Unix systems (and text viewers/editors that can) tend to use 0a but are able to parse 0d0a and a single new line.

ASCII also has control/escape codes (everything below 20 (20 = space key and is a normal character)), some editors do not like these and they may have to lost. 2E is a fullstop (which can get confusing as the fullstop character is used to represent an unknown character too).

If using a 16 bit encoding it may have to be aligned to a 2 byte (word) address or the decoder could break, roms are often not bound by this (which is a problem if 8 bit control codes are used).

00 is often used to end a section (although the game does not parse it) as well as for other purposes (space), this means replacing the character may not work. Finding the pointers (see pointer section below) and replacing accordingly is then necessary.

For common encodings like shiftJIS a web browser makes for a good “test viewer” and it is a common tool across many platforms.

Problems
It should be obvious by this point that rom hacking is not always a trivial exercise and there are problems/complexities. Some have already been mentioned but some mentioned for the first time.

Text formatting.
Changing the amount of text/space for it is a matter of changing the pointers (detailed below), when to start a new line is usually left to the programmer (which in this case is the hacker). Find the character responsible and use it when a new line needs to happen. The following “problems” in this section are interesting cases, this however is very common.

Implied text
“in the 80's”, when referring to a time that phrase strictly means in the 80's ce/AD but most people would interpret that as “in the 1980's”. Similar things are occasionally used in games where a new line (think speech bubble) would start with something and instead of repeating it that would be a string located elsewhere. Think back to the repeated tile problem often seen in title screen/menu hacking.

pointer driven compression
See below for an example but here a repeated phrase (normally a magic spell naming

scheme like firebolt megafirebolt great megafirebolt and so on). Pointers need not point to the start of a text section and may instead point to the section of the phrase required as opposed the . Naturally this can interrupt readability but you can either replicate the compression or you can simply alter the pointers to destroy it.

Parsed punctuation and variables

Hotels in game cost money and this amount may be different throughout a game, the text may then have a variable contained within it. Similarly if text is required to be bold/italic/sparkly then a flag may be placed within the text itself.
The main dilemma is whether to leave it, remove it or to change it to something else (naturally telling your translation people), naturally this will depend on the team but if it is a binary flag then a search and replace with the result being a human readable string that is not likely to come into the final translation (square brackets and excessive punctuation is good for this) is an idea and also helps basic text editors work (in ASCII the characters 1F and below may not have a graphical representation but they do have a function).

Multiple tables

Simple enough and normally used for different sections but it has been known for different characters to use different tables (usually to allow for a different font).

Binary located text

In the case of some DS games the actual text (or worse for things like implied text) can be found within ARM9 binary (the ARM7 is not usually used for anything the developer would create remember). Changing is not all that hard but changing the binary size can affect some carts and/or their patching methods. Likewise the pointers will become a bit more complex depending on what is happening (text can either be as it comes or in one big lump).
Simple, if tedious, to deal with for a hacker who can work with assembly but not so nice for someone who is not familiar with such techniques.

Example table finding.

This example will be tailored to show all the techniques described above (aside from ASM). In practice there are usually far fewer “stages”.
The table in question is the capcom table, this table is used by capcom in most of their games.
Game in question. Megaman ZX, US release.
File used \data\m_back_en.bin

Sample of the file.

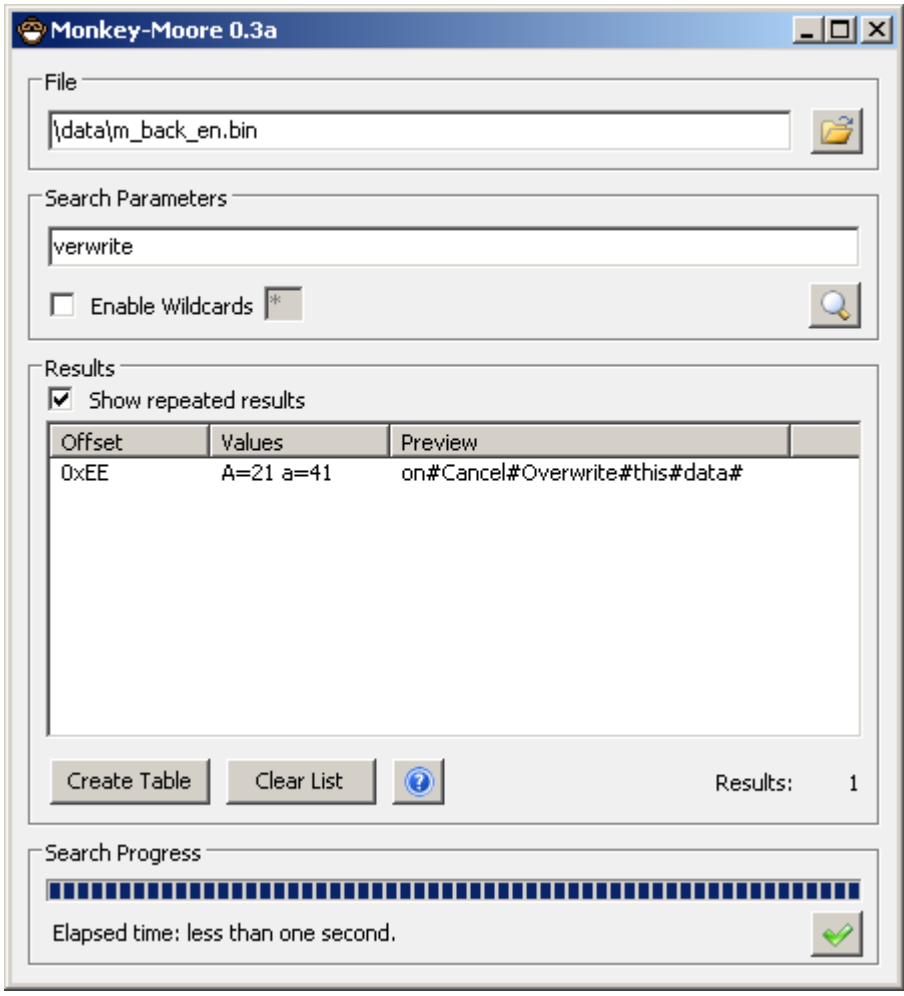
```
0000000 | BD01 1C00 0000 0E00 1D00 5400 9000 CD00 |
..........T.....
0000010 | EE00 0A01 2801 3301 3F01 4C01 6001 8A01 | ....(.
3.?.L.`...
0000020 | 2E4F 0044 4154 4100 464F 554E 44FE 2441 | .O.DATA.FOUND.
$A
0000030 | 5441 0043 4F52 5255 5054 4544 FEE0 E123 |
TA.CORRUPTED...#
0000040 | 4F4E 5452 4F4C 0030 4144 00EE 0FEF 1A33 | ONTROL.
0AD.....3
0000050 | 454C 4543 54FC E2E3 2255 5454 4F4E 1A33 |
ELECT..."UTTON.3
0000060 | 4156 4500 E4E5 2255 5454 4F4E 1A23 414E |
AVE..."UTTON.#AN
```

```
0000070 | 4345 4CFE E0E1 234F 4E54 524F 4C00 3041 | CEL...#ONTROL.
0A
0000080 | 4400 EE0F EF1A 3345 4C45 4354 0024 4154 | D.....3ELECT.
$AT
0000090 | 41FC E2E3 2255 5454 4F4E 1A2C 4F41 4400 |
A..."UTTON.,OAD.
00000A0 | E4E5 2255 5454 4F4E 1A23 414E 4345 4CFE
| ..."UTTON.#ANCEL.
00000B0 | E0E1 234F 4E54 524F 4C00 3041 4400 EE0F | ..#ONTROL.
0AD...
00000C0 | EF1A 3345 4C45 4354 0024 4154 41FC E2E3 | ..3ELECT.
$ATA...
00000D0 | 2255 5454 4F4E 1A24 454C 4554 45E4 E522 | "UTTON.
$ELETE.."
00000E0 | 5554 544F 4E1A 2341 4E43 454C FE2F 5645 | UTTON.#ANCEL./
VE
00000F0 | 5257 5249 5445 0054 4849 5300 4441 5441 |
RWRITE.THIS.DATA
0000100 | 1FFC 0000 3945 5300 0000 002E 4FFE 2C4F | ....
9ES.....O.,O
0000110 | 4144 0054 4849 5300 4441 5441 1FFC 0000 |
AD.THIS.DATA....
```

Luck:
It is not anything common but if you look at the ASCII output of the text editor the lower case characters match up with the upper case ASCII.

Extrapolation.
Whilst possible to do with other methods this is done for the sake of example.
The "UTTON section is fairly sure to mean Button. From this it is possible to infer 22 is used to represent the B character.

Relative searching
In this case it will follow on from extrapolation, in the real word you are likely to encounter several failures before you find the correct version (in the example above searching for “utton” would give several results).
The 22 character is a capital B and the # character in the ASCII section is obviously a capital C (#ancel #ontrol).
However a relative search will be conducted in an attempt to find “verwrite” (the capital O is omitted as it could be anything which would not be of much use when searching at the start of trying to find a table).

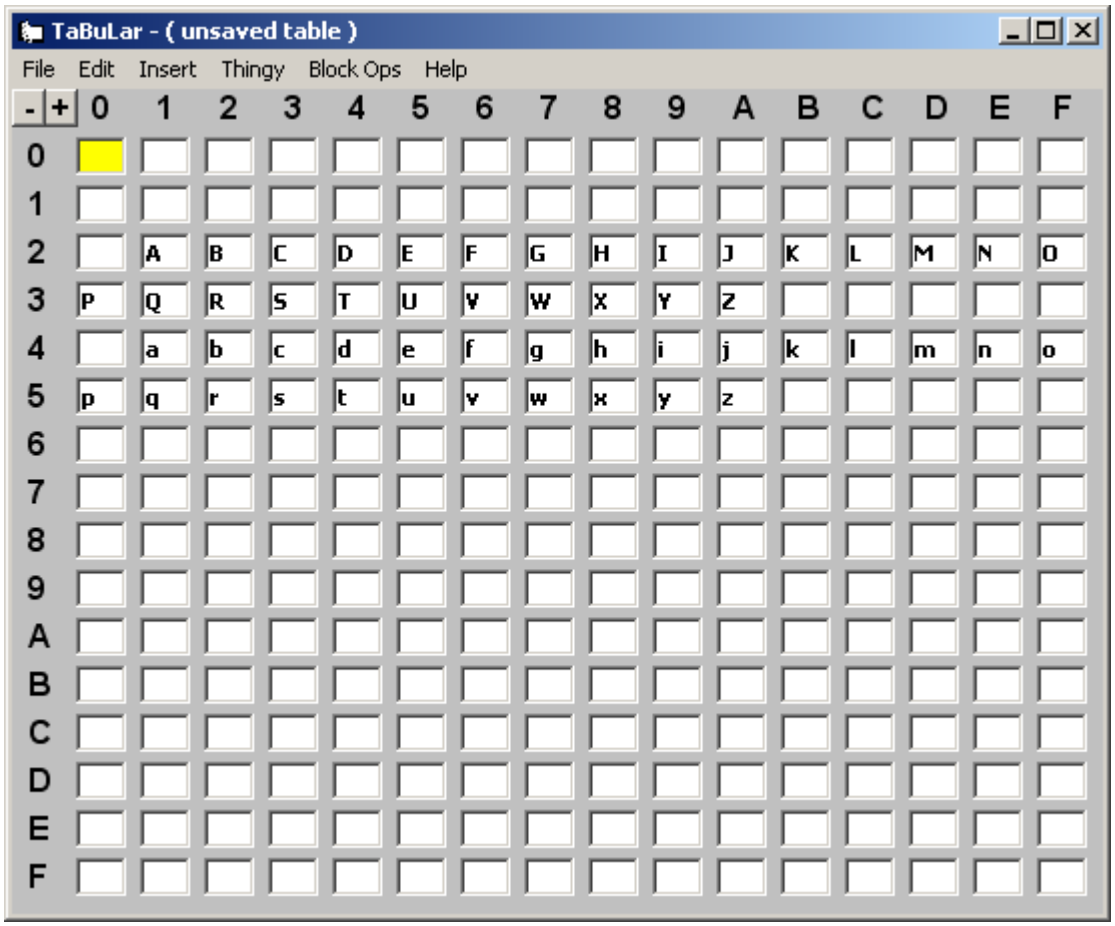


Distribution.
Here the space character will be determined. It is obvious thanks the things above but this is an example.

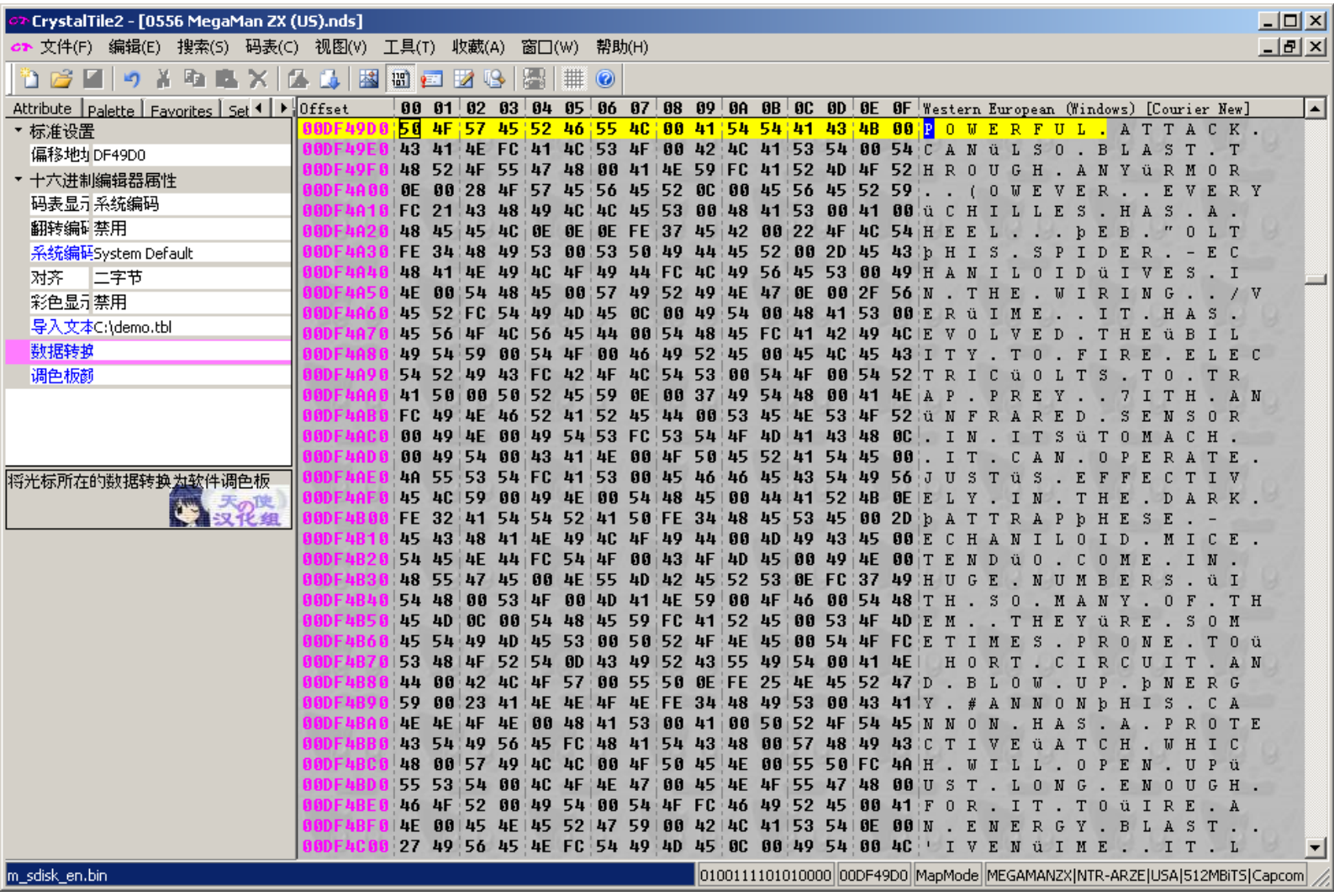
Dec	Hex	Char	Count	Percent
0	0x00		51	13.53
84	0x54	T	45	11.94
65	0x41	A	41	10.88
69	0x45	E	35	9.28
79	0x4F	O	26	6.90
68	0x44	D	22	5.84
76	0x4C	L	16	4.24
78	0x4E	N	16	4.24
254	0xFE		13	3.45
26	0x1A		9	2.39
36	0x24	$	9	2.39
82	0x52	R	9	2.39
85	0x55	U	9	2.39
83	0x53	S	8	2.12
67	0x43	C	7	1.86

Making a table.
Tabular is the table making application of choice here.

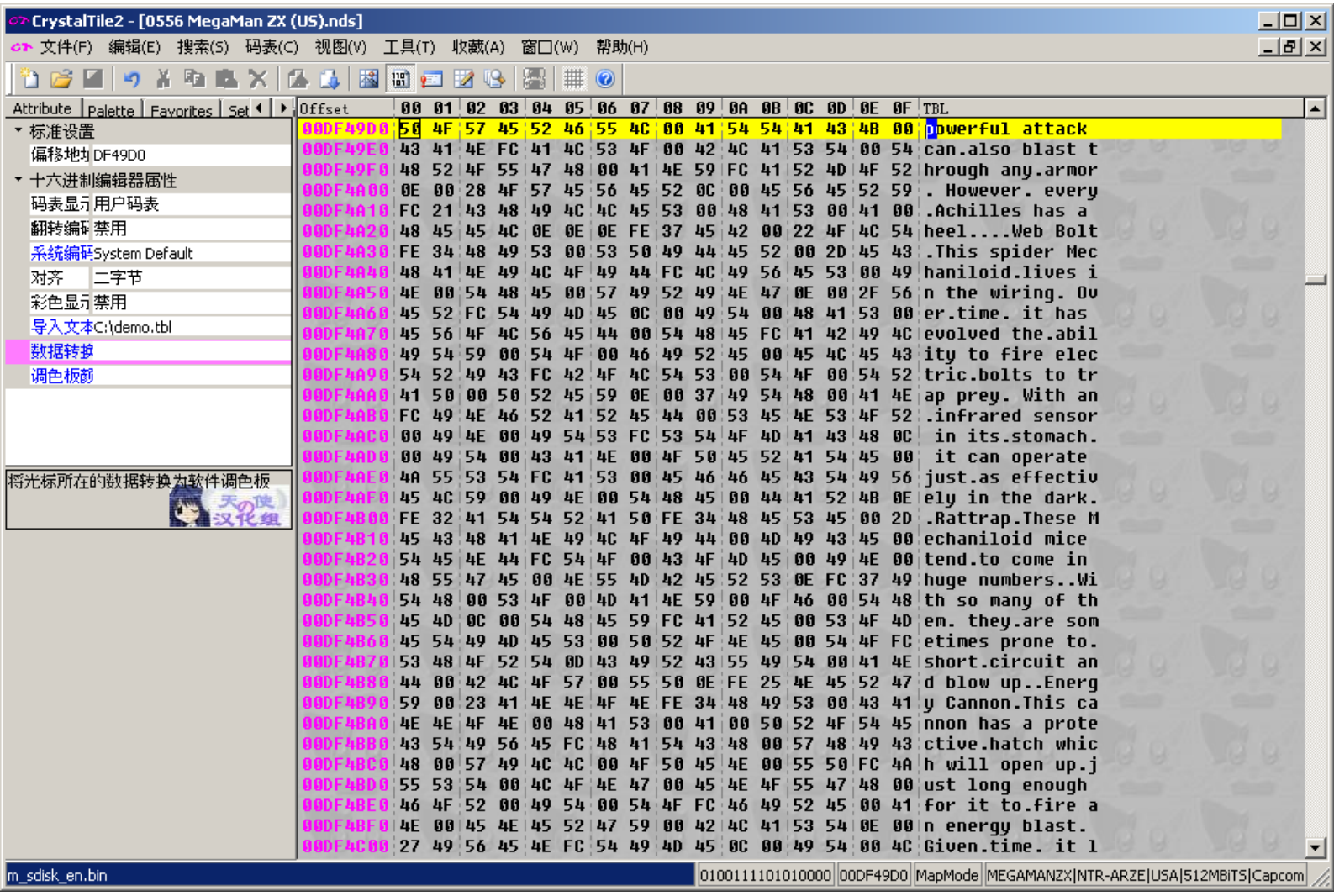
Remember the option to add a full alphabet of a given case is available and it is ordered column, row



Demo from crystaltile2:
In a real hack the numbers, punctuation and other things would be determined via guesswork/corruption and brute force methods.
Without table


CrystalTile2 - [0556 MegaMan ZX (US).nds]
文件(F) 编辑(E) 搜索(S) 码表(C) 视图(V) 工具(T) 收藏(A) 窗口(W) 帮助(H)
Attribute Palette Favorites Set
标准设置
偏移地址 DF49D0
十六进制编辑器属性
码表显示 系统编码
翻转编码 禁用
系统编码 System Default
对齐 二字节
彩色显示 禁用
导入文本 C:\demo.tbl
数据转换
调色板颜
将光标所在的数据转换为软件调色板
Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Western European (Windows) [Courier New]
00DF49D0 50 4F 57 45 52 46 55 4C 00 41 54 54 41 43 4B 00 POWERFUL.ATTACK.
00DF49E0 43 41 4E FC 41 4C 53 4F 00 42 4C 41 53 54 00 54 CANüLSO.BLAST.T
00DF49F0 48 52 4F 55 47 48 00 41 4E 59 FC 41 52 4D 4F 52 HROUGH.ANYüRMOR
00DF4A00 0E 00 28 4F 57 45 56 45 52 0C 00 45 56 45 52 59 ..(OWEVER..EVERY
00DF4A10 FC 21 43 48 49 4C 4C 45 53 00 48 41 53 00 41 00 ü!CHILLES.HAS.A.
00DF4A20 48 45 45 4C 0E 0E 0E FE 37 45 42 00 22 4F 4C 54 HEEL...þ7EB."OLT
00DF4A30 FE 34 48 49 53 00 53 50 49 44 45 52 00 2D 45 43 þ4HIS.SPIDER.-EC
00DF4A40 48 41 4E 49 4C 4F 49 44 FC 4C 49 56 45 53 00 49 HANILOIDüIVES.I
00DF4A50 4E 00 54 48 45 00 57 49 52 49 4E 47 0E 00 2F 56 N.THE.WIRING../V
00DF4A60 45 52 FC 54 49 4D 45 0C 00 49 54 00 48 41 53 00 ERüIME..IT.HAS.
00DF4A70 45 56 4F 4C 56 45 44 00 54 48 45 FC 41 42 49 4C EVOLVED.THEüBIL
00DF4A80 49 54 59 00 54 4F 00 46 49 52 45 00 45 4C 45 43 ITY.TO.FIRE.ELEC
00DF4A90 54 52 49 43 FC 42 4F 4C 54 53 00 54 4F 00 54 52 TRICüOLTS.TO.TR
00DF4AA0 41 50 00 50 52 45 59 0E 00 37 49 54 48 00 41 4E AP.PREY..7ITH.AN
00DF4AB0 FC 49 4E 46 52 41 52 45 44 00 53 45 4E 53 4F 52 üNFRARED.SENSOR
00DF4AC0 00 49 4E 00 49 54 53 FC 53 54 4F 4D 41 43 48 0C .IN.ITSüTOMACH.
00DF4AD0 00 49 54 00 43 41 4E 00 4F 50 45 52 41 54 45 00 .IT.CAN.OPERATE.
00DF4AE0 4A 55 53 54 FC 41 53 00 45 46 46 45 43 54 49 56 JUSTüS.EFFECTIV
00DF4AF0 45 4C 59 00 49 4E 00 54 48 45 00 44 41 52 4B 0E ELY.IN.THE.DARK.
00DF4B00 FE 32 41 54 54 52 41 50 FE 34 48 45 53 45 00 2D þ2ATTRAPþ4HESE.-
00DF4B10 45 43 48 41 4E 49 4C 4F 49 44 00 4D 49 43 45 00 ECHANILOID.MICE.
00DF4B20 54 45 4E 44 FC 54 4F 00 43 4F 4D 45 00 49 4E 00 TENDüO.COME.IN.
00DF4B30 48 55 47 45 00 4E 55 4D 42 45 52 53 0E FC 37 49 HUGE.NUMBERS.ü7I
00DF4B40 54 48 00 53 4F 00 4D 41 4E 59 00 4F 46 00 54 48 TH.SO.MANY.OF.TH
00DF4B50 45 4D 0C 00 54 48 45 59 FC 41 52 45 00 53 4F 4D EM..THEYüRE.SOM
00DF4B60 45 54 49 4D 45 53 00 50 52 4F 4E 45 00 54 4F FC ETIMES.PRONE.TOü
00DF4B70 53 48 4F 52 54 0D 43 49 52 43 55 49 54 00 41 4E SHORT.CIRCUIT.AN
00DF4B80 44 00 42 4C 4F 57 00 55 50 0E FE 25 4E 45 52 47 D.BLOW.UP.þ%NERG
00DF4B90 59 00 23 41 4E 4E 4F 4E FE 34 48 49 53 00 43 41 Y.#ANNONþ4HIS.CA
00DF4BA0 4E 4E 4F 4E 00 48 41 53 00 41 00 50 52 4F 54 45 NNON.HAS.A.PROTE
00DF4BB0 43 54 49 56 45 FC 48 41 54 43 48 00 57 48 49 43 CTIVEüATCH.WHIC
00DF4BC0 48 00 57 49 4C 4C 00 4F 50 45 4E 00 55 50 FC 4A H.WILL.OPEN.UPü
00DF4BD0 55 53 54 00 4C 4F 4E 47 00 45 4E 4F 55 47 48 00 UST.LONG.ENOUGH.
00DF4BE0 46 4F 52 00 49 54 00 54 4F FC 46 49 52 45 00 41 FOR.IT.TOüIRE.A
00DF4BF0 4E 00 45 4E 45 52 47 59 00 42 4C 41 53 54 0E 00 N.ENERGY.BLAST..
00DF4C00 27 49 56 45 4E FC 54 49 4D 45 0C 00 49 54 00 4C 'IVENüIME..IT.L
m_sdisk_en.bin
0100111101010000
00DF49D0
MapMode
MEGAMANZX|NTR-ARZE|USA|512MBiTS|Capcom

With table:



pointers

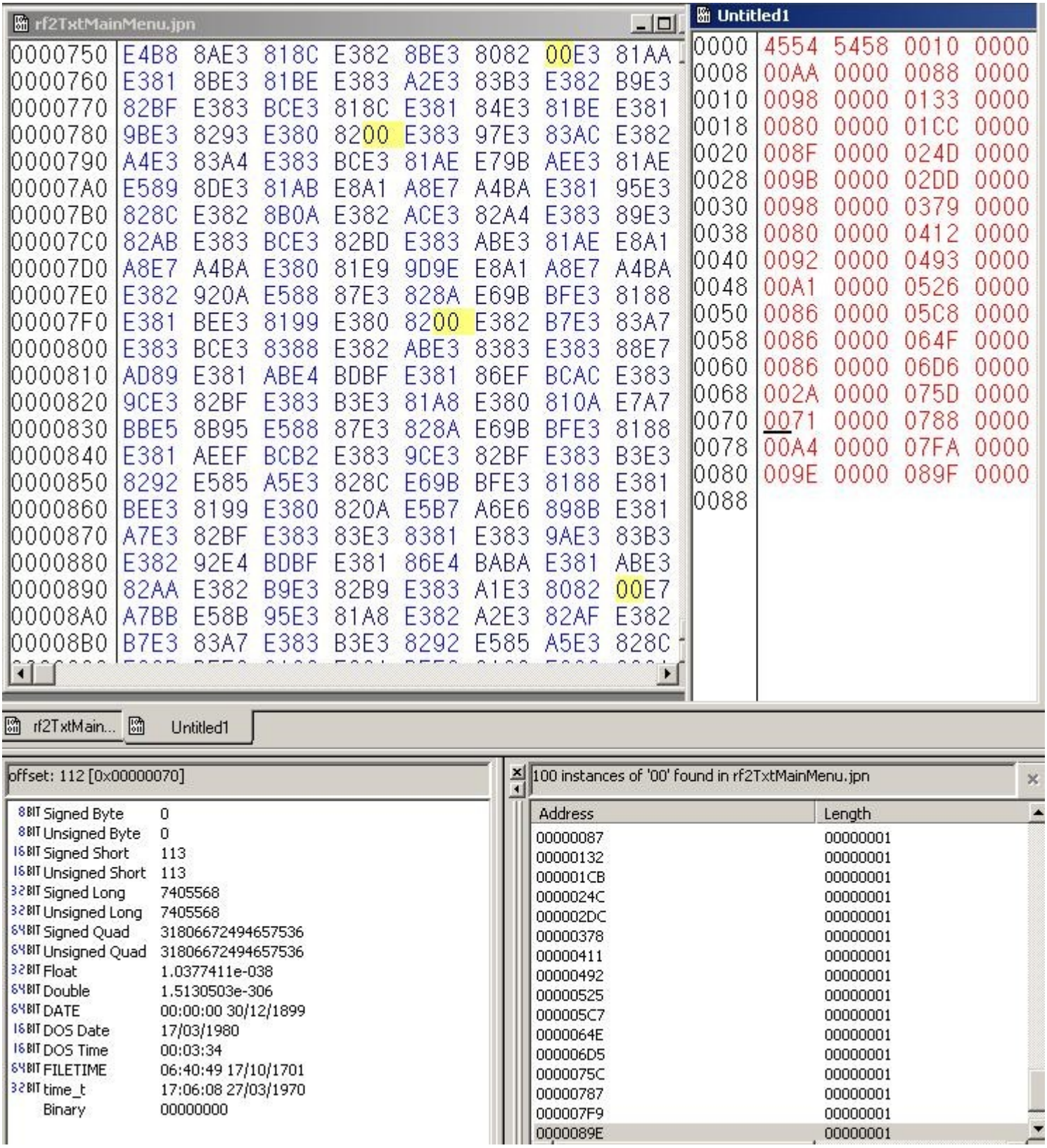
You know a sentence ends because of the fullstop, a computer however does not so it can either parse the text or more commonly use and index/contents file/section known as a pointer table.
This was touched upon back in the addressing section but there are some subtle differences.
Assuming your text is not graphical (a good chunk of puzzle games have a very small amount of text so they eschew encoding a font in place drawing a picture) and the very rare cases of end to end and set limit text (I will quickly summarise them at the end) what you want to read about is pointers.

In short they are like a contents to a book as trying to get the computer to calculate the end of a work is a pain and a waste of resources.

If you run over the length without changing the pointers to match the worst that will happen is you will crash and at best your text will spill over into other areas.

They come in three main forms:
Standard: you will get a list of hexadecimal numbers

These numbers count from the start of the file to the text.
Example from rune factory 2 (original Japanese release) and it deals with the endianness problem. Note the nicely named file (rf2TxtMainMenu.jpn).



Offset:

The numbers in the standard form above usually come at the start of the file and are called a header. These work in the exact same way as standard but count from the start of the text.

Relative:
A hybrid of offset and standard types and in some ways the easiest on the system (although not on your head). Basically they count on from their place in the file. For example if the pointer is at address 5c in the file and reads 10 you add the two numbers together to get where you need to be at (6c in that example).

The exceptions and a few problems
GBA: As has been mentioned several times now the GBA cart is in the memory so pointers will usually start with a 08 for roms 16 megabytes and under and can have 09 for some roms 32 megabytes in size. DS roms tend to be little files which is nice and keeps things simple.
Remember the GBA also has mirrors at higher wait states as follows:
08000000-09FFFFFF Wait State 0 (usually only 08 owing to most GBA carts being less than 16 megabytes)
0A000000-0BFFFFFF Wait State 1 (mirror)
0C000000-0DFFFFFF Wait State 2 (mirror)
The DS only recognises 08XXXXXX hex and 09XXXXXX hex and the mirrors have no meaning if you are hacking a DS rom (or changing the use of pointers for the times the cart cart is referenced).
Pointers being made in this fashion can be utilised to find information to, a search which reveals a group of 08 characters equidistant from one another may well be a pointer map (which leads directly to the files being pointed at)

End to end: I know a sentence is finished when you put a fullstop: some games (mainly nes and snes era) do the same with a 00 or something. This method is frequently seen in file formats for names of the files contained within.

Fixed length:
ever wondered why certain final fantasy titles have names of spells that look horrible when translated. Basically the devs decided 12 (or some other number of) characters would do for names (and it usually does in Japanese) but when it comes to the west things get a bit more difficult. Rather than redo it the localisation team just scrunched the names up. If you encounter this it is usually wise to continue at the next section i.e. for 4 characters ACBD|EFGH|HI__|JK__

Endianness:
Already covered back in the section of hexadecimal but the concept is as follows:
I know the number 7000000000 is bigger than 6999999999 despite most of the numbers in the one starting with 7 are smaller than the one with 6 at the start. This is called big endian and the GBA/DS read numbers the other way* i.e. 0745 would actually be read 4507. Simple enough really once you know the trick.

*That is 16 bit, some use 32 bit for larger files. Here 0745 1234 would be read 3412 4507.

Sector based:
Not so important for text but as pointers are used in packing formats (see ARC for wii:

http://gbatemp.net/index.php?showtopic=72013&st=0). Here a sector is defined as opposed to being the start of a sentence/paragraph in most text. In short a sector of say 256 bytes is defined, now instead of saying go 1024 bytes in the application takes a 4 and multiplies it by 256 to get there. 4 is a much smaller number and can be represented far more easily allowing for a larger space to be occupied. There is a problem with wasted space but compression or simply the fact a disc can hold a large amount of data with large tracts unused renders this point somewhat moot.

Use in/as compression.
Perhaps not as useful for the GBA, DS, GC and Wii where space is plentiful but worth learning/considering.

Spells in some RPGs have a naming scheme referring to how powerful they are.
E.g.
firebolt
great firebolt
mega great firebolt

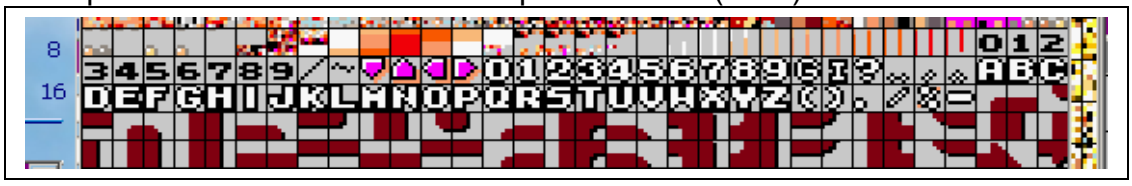
A nasty use of language perhaps but it is just an example. Imagine if you will that the end of the text has a 00 or some other end of section character.
Now instead of having firebolt00great firebolt00mega great firebolt00 with the pointers going to each one you can point to the start of the word you want (i.e. for the basic firebolt spell you would just point towards the end of the text where firebolt starts and so forth) leaving you with just "mega great firebolt00" for your text section.

fonts

binary code is just electrical signals and try as hard as you might but you will probably not be able to interpret them in any meaningful way. This means a number then calls a representation to be displayed on the screen as a glyph, character, rune or whatever your language of choice chooses to call the pictorial representation of spoken word.

Before going on it is worth understanding the ways graphics work on the systems concerned, this being said the font is often just a 2d tile or two for a character but it can be more complex than that (see deufeufeu's jump ultimate stars project: http://deufeufeu.free.fr/wiki/index.php?title=Jump_Ultimate_stars#AFT_file_format_.28font.29)
While the link above details another format a common format for fonts on the DS is NFTR. Rather nicely a couple of tools exist that are able to edit files using it:
http://gbatemp.net/index.php?showtopic=105060&st=0&start=0
crystaltile2 has basic nftr editing abilities.

As mentioned normal fonts (sometimes there are multiple fonts for a particular ROM) are usually in GBA graphics format like so for the GBA and DS, they are used often and are fairly small so you will usually be lucky and find them uncompressed, common compressions are LZ and bit packing owing to the tendency for fonts to be 2 colours only):
Example from Advance Wars 2 European release (GBA)



For those with experience of PC based fonts vector based fonts do not really exist for the consoles outside of tech demos, all fonts are ultimately (see above for two examples where this is not so simple) bitmap based.

Two words/acronyms a good to know here too: CFW and VFW. They stand for Constant font width and Variable font width respectively.
CFW, while getting rarer, is normally what you see when you convert a Japanese game to a European language.
What happens it characters like i or l or even punctuation like . (fullstop) will be assigned their own “box” to sit in whereas in normal use they will be squashed up against other characters. As most Japanese characters are about the same width this is not an issue for them really.
These gaps make for some fairly ugly looking text and also wastes a lot of space on screen (something you do not want really).
The other thing is that lowercase roman characters do not always stay in the lines like Japanese ones (examples being j,q,p,y and g), these characters will either sit at the same height as the other characters making for a rather ugly mess. The are several methods (and you can even combine a few of them)

The replacement of characters however can lead to mojibake or more commonly in rom hacking circles moonspeak (not to be confused with the semi-pejorative term for Japanese) or cavespeak. Here when a font (or the font parsing routine*) is changed any existing text is liable to be rendered incorrectly. It is most often associated with partial translations on the earlier systems.

*a common hack when translating from Japanese is to change the character encoding from an 16 bit one to an 8 bit one with an eye to saving space.

Method 1 (best but hardest)
Brushing up on your assembly skills is the best way to deal with this but there are methods you can use to achieve good results without needing to resort to assembly coding.
Some concepts for implementing a VFW font, the most common way is to first hijack the text display routine, jump it to another area and take a deep breath.
You have a few options here, one is to use the collision detection with a specially engineered font (add a pixel in a similar colour to the background and “stop” when the letters collide. This poses problems mainly with speed.
The other is to assign widths (the lookup table returns) to each character and have it calculate the width of a current “set” of characters and then start a new tile if necessary, a good guide via some very well commented assembly source to this can be found here:
http://www.romhacking.net/docs/337/
Problems can come if you want to change the font (although this normally just entails tweaking the table of widths).

Method 2 (easier and quite effective) Skinny font.
It is not quite as nice as a true VFW hack but it is far easier, find the code that deals with font width (a Japanese game may only have a few lines of code) and it is sure to deal with the OAM or the bitmap background (both of which has a nicely defined memory section for the GBA, DS, GC and Wii) at some stage. Edit this to a lower value and build your font around this. An example exists in the assembly hacking section above.

Method 3
A method to work around the ugliness that can occur with wide characters (w, m, Q and

more depending on how thin you want to go) also provides another method to make a font look nice in general. It is also nice to use on the occasions that you have a fixed number of characters (fixed width for menus is a good example, text located in the binary another) and altering the amount is not so easy.
Dubbed here as the pseudo variable width font it is a take on the half width problem briefly mentioned in text hacking above. What you do is spread long characters (or multiple characters) across several tiles (assuming tiles are used to store character representations), a good example to start with is the common phishing scam trick of using a vv (two lower case V characters) in place of a w (W as in wolf). It may not work for your hack (for example if the game forces a pixel or two between what it perceives as “characters” but it provides a thinking/starting point.

Nicely Japanese has thousands of characters (way more than the hundred or so in common usage in European languages and even if a game does not provide them all there is usually enough for this sort of thing) so you can quite easily repurpose some characters you otherwise would not be using.

Method dealing with lower case characters and the lines.
Again assembly is the best route. Shifting the characters up by one line and by extension giving a section for the “tails” of the characters to sit is a simple and effective way of dealing with things. It also helps alleviate some of the aesthetic issues with having a narrow font.

Multimedia

While games are primarily and interactive medium it is occasionally useful to have “traditional” media in the game. Audio and video are these and are detailed in their respective sections below.

Audio:

Sound appears in waves for those that recall science lessons. The most basic form of storing sound is to sample the waveform at intervals (hence sample-rate) and play back the amplitude of the wave at the intervals they were taken from. Do this well enough and you replicate the sound as far as people can detect (this is usually considered to be around 2 times the highest frequency someone can hear or around 44KHz for "transparent" quality, this is an example of oversampling mentioned in the cryptography section). The DS tends to muddle around from around 11KHz (about telephone quality) to around 48000 Hz ("DVD" quality) and come in three main formats:
instrument:
a cymbal hit, a piano sequence.....
"midi" (midi is a computer “language” aimed at hardware, the SSEQ files which act like them are programming oriented though)
an arrangement of instruments
full blown wave format.
whole songs or voice samples.
There are some subtle tweaks on this and they are often interrelated but that is the main concept.

44000 samples a second each with 16 bits as you can imagine takes a lot of space up in very short order but rather nicely most sounds do not alternate that much so you can then assume one millisecond is the same as the next (it gets far more complex than that of

course) which means you can drop the size of the file. Now with the sound not being a waveform you have to make it into one which takes CPU time and other resources.

The rest of this section is going to focus on the formats used and how to manipulate them.
GBA sound hardware:
http://belogic.com/gba/

For information on the DS sound hardware GBAtek has a lot of information
http://nocash.emubase.de/gbatek.htm#dssound

What sound files are.
GBA:
The audio on the GBA can be in any format and is usually an ASM level hack. There is however GSF which is an emulated format (much like PSF and similar)
http://gsf.caitsith2.net/
Recently there has been some activity in reverse engineering on of the common sound formats from a hacking perspective. Work is ongoing however:
http://z9.invisionfree.com/Golden_Sun_Hacking/index.php?showtopic=9

DS:
The most common format is the SDAT format. Used in all but around 15 games (unfortunately those 15 games are somewhat noteworthy).
Others use common formats (electroplankton uses ordinary wave files) and not so common in general audio circles but common in games world (the world ends with you http://gbatemp.net/index.php?showtopic=86998 and N+ http://gbatemp.net/index.php?showtopic=102645&st=0). Such “new” formats are almost always another way to get a compressed wave file playing rather than a new take on midi style arrangements.

One or two use tweaks on a "common" format meaning you will likely have to make new tools/alter existing ones to do it (lower bitrate/only mono audio/lower sample frequency/lower bit depth.....) but this is fairly rare and most tools allow for this even if they were not intended to be used for it). Owing to the difficulty in creating a new format from scratch and the lack incentive to do it (it would be a lot of work for little, if any, gain) few companies ever make their own format here. Should they make a new format they will usually either patent it (if it is allowed) or publish it to try and spread it far and wide (a format nobody uses is doomed to failure and obscurity).
While audio is not exactly low demand there is enough power possessed by the consoles that any unknown sound format is not likely to correspond closely to the hardware.

As mentioned most DS roms use the SDAT format and this section will now focus on it. On rare occasions files usually contained within the SDAT format (STRM is found in Tony Hawk's American Skateland).

SDAT format stuff is fairly easy to work with.
Specs here
http://loveemu.yh.land.to/page/NDS/SDAT.html
http://kiwi.ds.googlepages.com/sdat.html
http://tahaxan.arcnor.com/index.php?option=com_content&task=view&id=38&Itemid=36
Tahaxan forums have a early stage repacker but a hex editor and a spreadsheet are generally nicer to work with.

The techniques used often mirror the concepts used in 3d hacking detailed above in that

files either replaced wholesale or the internal file structure is manipulated to play different songs at different times.

SDAT/sound replacement:
Originally done to shrink roms (goldeneye has an especially small file) but these days is done mainly for undubbing games.
The simple undub merely takes the Japanese sound files and replaces the ones from the European or US release and it works most of the time. Other times there are extra or fewer sounds or it is done in a different order (the DS can use name of a file name or ordinal (numbers) to call a sound) leading either to odd sounds being played or to the game crashing. Here you have to alter the files to match with techniques detailed in the next paragraph.

SDAT tweaking: various people have tweaked tetris to play different songs. The author did it to play the classic tetris theme (http://gbatemp.net/index.php?showtopic=36870) all the time (method detailed below) and mufunyo removed the BG music entirely (http://gbatemp.net/index.php?showtopic=69603).
Other times this can be done to reduce loading times/increase stability. Various people did it for the Castlevania portrait of ruin to reduce/help with crashing for DS flash carts and/or their memory cards with slow read speeds.

The names and their extensions/types will usually make the file you want to change obvious but if not there are many decoders available to test the sound out with and the usual methods of finding files should be relatively easy owing to the low amount of files involved.

For the BG music tweaking the file system is normally tweaked to read a different file (or none at all, normally by "reading" outside the actual file/setting the file to 0k).

Example hack. Tetris Sound tweaking.
Here the aim is to make the game play the classic tetris music normally only available to higher levels in the single player game (and not at all in the vs computer which leaves you with the mario themetune). Note this means the game will not play the correct songs when playing through the music “test” in the options menu.
Unfortunately the file names and their locations are not in the same sections. However they should be in the same order.
The location can be found in the symb block are located at offset 48 hex. Flipped it reads 064C hex (adding 40 hex (the length of the header)) it reads 068C. Failing that search until you find a long list of ASCII file names.

Locate the offsets. These are found in the FAT section. Either search for FAT (in ascii) in the file or read the number at 20 hex in the header
In this case it is 2198 hex.
Align the names to the offsets.

Alternative method. Read the offsets with something like crystaltile2. You still have to find all this within the file so time saved in negligible.
In the file system the desired song is called the BGM_STD_KARINKA (in crystaltile2 it is Sequences\018_BGM_STD_KARINKA.SSEQ). The properly romanised name is the Kalinka although the song itself is frequently confused with another very similar sounding Russian folk song (also used in various versions of tetris) called the Korobeiniki (the Kalinka increases in speed and cuts out while the Korobeiniki repeats some more).

Enough of that though, this is hacking document not Russian traditional music discussion.

Now the FAT locations for the other files (in this case the targets are the BGM_STD files (08 through 019 in crystaltile2))
The locations were given when the files were aligned so all that is needed is for the string giving the location and size to be replaced for each of the strings for the other BGM_SDT files.
If for some reason you have flipped the bytes (it will give readable offsets) remember to flip them back at some point so as not to mess up the game.

The string in question is 605E 0100 780F (as an aside it means the file is located at relative address 00015E60 and is 0F78 large).
If you preferred you could copy an entire section between entries but as such a hack will probably not be as simple when you come to do it only the data will be used rather than the padding.

Sample from the original (section in question has been highlighted). Note that the 0000 between the early entries disappears as the locations get further away.

```
0002190 | 0000 0000 0000 0000 4641 5420 7C08 0000 | ........FAT
|...
00021A0 | 8700 0000 202A 0000 4403 0000 0000 0000 | ....
*..D.......
00021B0 | 0000 0000 802D 0000 8404 0000 0000 0000
| .....-..........
00021C0 | 0000 0000 2032 0000 B403 0000 0000 0000 | ....
2...........
00021D0 | 0000 0000 E035 0000 E003 0000 0000 0000 | .....
5...........
00021E0 | 0000 0000 C039 0000 7405 0000 0000 0000 | .....
9..t.......
00021F0 | 0000 0000 403F 0000 0402 0000 0000 0000 |
....@?..........
0002200 | 0000 0000 6041 0000 DC03 0000 0000 0000 |
....`A..........
0002210 | 0000 0000 4045 0000 4407 0000 0000 0000 |
....@E..D.......
0002220 | 0000 0000 A04C 0000 0C53 0000 0000 0000 |
.....L...S......
0002230 | 0000 0000 C09F 0000 FC11 0000 0000 0000
| ................
0002240 | 0000 0000 C0B1 0000 2816 0000 0000 0000 | ........
(.......
0002250 | 0000 0000 00C8 0000 2C19 0000 0000 0000
| ........,.......
0002260 | 0000 0000 40E1 0000 5C1E 0000 0000 0000 |
....@...\.......
0002270 | 0000 0000 A0FF 0000 7027 0000 0000 0000 |
........p'......
0002280 | 0000 0000 2027 0100 8C04 0000 0000 0000 | ....
'...........
0002290 | 0000 0000 C02B 0100 8016 0000 0000 0000 | .....
+...........
00022A0 | 0000 0000 4042 0100 640E 0000 0000 0000 |
....@B..d.......
00022B0 | 0000 0000 C050 0100 900D 0000 0000 0000 |
.....P..........
00022C0 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
```

```
00022D0 | 0000 0000 E06D 0100 DC08 0000 0000 0000 |
.....m..........
00022E0 | 0000 0000 C076 0100 F81F 0000 0000 0000 |
.....v..........
00022F0 | 0000 0000 C096 0100 F002 0000 0000 0000
| ................
0002300 | 0000 0000 C099 0100 4404 0000 0000 0000 |
........D.......
0002310 | 0000 0000 209E 0100 AC5F 0000 0000 0000 | ....
....._.......
```

Hacked version

```
0002190 | 0000 0000 0000 0000 4641 5420 7C08 0000 | ........FAT
|...
00021A0 | 8700 0000 202A 0000 4403 0000 0000 0000 | ....
*..D.......
00021B0 | 0000 0000 802D 0000 8404 0000 0000 0000
| .....-..........
00021C0 | 0000 0000 2032 0000 B403 0000 0000 0000 | ....
2...........
00021D0 | 0000 0000 E035 0000 E003 0000 0000 0000 | .....
5...........
00021E0 | 0000 0000 C039 0000 7405 0000 0000 0000 | .....
9..t.......
00021F0 | 0000 0000 403F 0000 0402 0000 0000 0000 |
....@?..........
0002200 | 0000 0000 6041 0000 DC03 0000 0000 0000 |
....`A..........
0002210 | 0000 0000 4045 0000 4407 0000 0000 0000 |
....@E..D.......
0002220 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
0002230 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
0002240 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
0002250 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
0002260 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
0002270 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
0002280 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
0002290 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
00022A0 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
00022B0 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
00022C0 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
00022D0 | 0000 0000 605E 0100 780F 0000 0000 0000 |
....`^..x.......
00022E0 | 0000 0000 C076 0100 F81F 0000 0000 0000 |
.....v..........
00022F0 | 0000 0000 C096 0100 F002 0000 0000 0000
| ................
0002300 | 0000 0000 C099 0100 4404 0000 0000 0000 |
```

```
........D.......
0002310 | 0000 0000 209E 0100 AC5F 0000 0000 0000 | ....
....._.......
```

If you extracted the file to work upon it can be inserted with ndsts as the file has not been resized which bypasses the issues some carts/hacks and so forth have with rebuilding tools and also makes the patch a bit smaller (in this case it would allow IPS to be used).

Complex undub.
Occasionally the sound file layout design is changed from the original release which has the effect of “incorrect” sounds being played or the game crashing. The task here then is to make them match up.
Note most crashes of this type are actually due problems with the commonly used ndstool (some will refer to the frontends DSLazy and DSBuff) which can be fixed by using another tool/method (preferably something like NDSTS).

Other areas
Next is simple replacement/swapping. A repacker was already mentioned and it is easy enough to swap from rom to rom but the formats used within SDAT are not quite common PC formats which makes converting to the format a bit harder. The formats are detailed in the links above and kiwi.ds also made midi2sseq:
http://kiwi.ds.googlepages.com/midi2sseq.exe

You have simple sounds: drum hits, a gun shot in sound bite format (normally some form of PCM/IMA-ADPCM). SWAR is the extension of choice here (they are a collection of mono wave samples) although there is another similar type called SWAV (one of the samples from the SWAR archive) also exists.

You have midi like files (not actual midi but close enough for conversion to and from) with their instruments (banks) in distinct sections. These sequences have the extension SSEQ (Sound SEQuence) and they can be combined into SSAR (Sound Sequence ARchive). Their sound banks have the extension SBNK.
Note some games mix and match what they use the above type for, some will play a SSEQ for a single gunshot or flourish.

Lastly you have full blown tracks or streams (Tony Hawks games are the most common here). PCM or IMA-ADPCM (also detailed above) are the most common with STRM being the extension of choice.

Provided you observe the file format (you give a bank to the sseq), you account for compression type, headers and so forth files can be swapped.

SSEQ tweaking, despite SSEQ format sounds being fairly well understood it is not that commonly done. Other file types being wave related it is simply a matter of accounting for format requirements (same samplerate, bit depth.....) and headers and adjusting the file locations and sizes (extending a file is usually the easiest route) so will not be detailed in this version.

http://kiwi.ds.googlepages.com/sdat.html#sseq
The above link details specifications for the sseq file, if you are familiar with midi type music the following section may seem a bit simple however it is assumed more readers will be coming in “cold” and as such it will be geared towards them.

Note where it says “refer to loveemu's sseq2mid” it means this file
http://loveemu.yh.land.to/wiki/NDS/SDAT.html

Strictly speaking SSEQ and midi have some substantial differences (midi is hardware based and spreads instructions over several stages while sseq is aimed at software and is far more contained) but in as much as they are both “sheet music” for electrical hardware the analogy stands.
For more on midi
http://www.midi.org/aboutmidi/tut_midimusicsynth.php

The link on sseq above details the “events” and the timing of the operation.
In games you may notice a change in pace according to what is happening at the time (low health/time or almost at game over in tetris), you can influence this timing and change it.

GC sound formats.
Several sound formats are used, most are able to be played back and with knowledge of this edit them too
http://hitmen.c02.at/files/yagcd/yagcd/chap15.html#sec15
http://www.hcs64.com/in_cube.html

Wii sound formats.
Much like the GC formats for the Wii, there is also sound format resembling the DS sdat format that is not especially well understood at this time.

Video:

Much like audio with samplerate if you play enough pictures quickly enough you create the illusion of movement (12 frames per second (fps) is about the limit with 17 being acceptable (older films used this) 20 fps being comfortable and 24-30 being pretty good (24 is film and 25 is PAL 30 is technically NTSC although video is usually converted from 24/25 to 30).

Not much happens from frame to frame and not much changes between one pixel and the neighbour to it so you can call them similar and save space (again it gets far far more complex with everything from psychology, electrical engineering, physics to fields of maths many have never heard of being involved), again this incurs the need for more CPU and resources to decode.
While intellectual pursuits are likely what brings rom hackers to the table reverse engineering all but the most basic of formats is a monumental task (video, even for low powered systems like these, is usually anything but basic).
In fact it was due to a smap file (part of the SDAT format not present in the end product) being left in a rom (0235 - Zoids Saga DS - Legend of Arcadia (Japan) if you want to have a look at an official one) that the SDAT format got reverse engineered as quickly as it did.
As an aside developers have been known to leave very interesting things within roms from both a hackers perspective (header files/source code) to things of interest to fans (discussion on the game, pictures and so on).
Should you desire and example here is the an introductory explanation of MPEG1 encoding:
http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/mpeg1/
Today MPEG1 is regarded as an antiquated video encoding standard and is not really used outside of low performance/legacy systems and software. The link above is just the specification too, a would be hacker has to not only figure out the specification but make a

decoder too.

The two most common hacks are
file system: make one video play by tricking the game into loading it by altering the file system or pointers that the game uses. Usually done with an eye towards saving/making space but making videos play at a given time is possible too.

and the lucky occasions where you have a known format like rad tools BINK: http://www.radgametools.com/bnkdown.htm It is very common in home consoles (the GC and Wii included), the PC market and has a presence in the DS world.

Ingame cutscene:
the controls tend to fall dead and the sprites on the screen or the 3d images get moved normally according to a list somewhere. Saves on space used (a sequence of numbers for a 5 minute cutscene like this is not likely be any more than that used for a 30 second video clip). This is usually a tracing hack on the OAM or if you are lucky a file/simple section with coordinate data. Harder versions would be if the programmers have the data in the binary (although it is a good place for an overlay).

Outlying areas.

With a few exceptions up until now the document has been little more than a basic computer science introduction and a nicely commented specification on the file formats. This next section aims to be a “notes from the front lines”. It will probably not make much sense if you read it without knowing the thought process behind it but it should offer some insight. This is only based my experiences and those I have had immediate contact with during the course of things so I encourage you to find other sources of info on this.

Example translation hack

Adapted from a private message on the subject detailing the thought process behind what is done and how it should progress.
The game was a harvest moon title on the DS aiming for translation from Japanese to Spanish.

First stage is find the text encoding and dump the script ready to be translated. The most common type of encoding for Japanese games on the DS is shiftJIS, a rather horrible abortion of a standard as far as things go but it works.
The rune factory series uses it so there is a good chance Harvest moon will also use it. Hopefully it will not be compressed.
ShiftJIS:
http://www.rikai.com/library/kanjitables/kanji_codes.sjis.shtml
Simple Japanese text editor:
http://www.tanos.co.uk/jlpt/jwpce/

Before you get to all that though you will have to pull the rom apart, DS roms are much like CDs in that unlike the other systems files are distinct from each other making rom hacking easier once you have figured out what they do and makes it easier to get at them but harder than other systems until you do find out what is what.
A topic covering the basics of it:
[See the file systems section above.]

The script will likely be in several files and if you are lucky they will have names (English

based) to indicate what they do: something like NPC_name for the names of the non player characters is not uncommon (and if the harvest moon and rune factory games are anything to go by then you should be OK here as well).

You now have a choice: if you want to recruit someone I would make a demo showing you can do something, rather basic but here is something made by pulling apart rune factory 2: http://gbatemp.net/index.php?showtopic=70192&st=41

Putting it back together: text files have an index (see "pointers" section). Once you have this you will probably want to alter the pointers to make the text read more easily (or even to work at all: if the game has something outside boundaries it may crash). This stage takes a while.

In short there will probably be the following stages:

pulling the rom apart (takes a minute at most once you have everything set up)

finding the text (could be very quick or it could take days or more)

decoding the text (if you have an emulator it should be fairly quick)

reworking the text: If you hand your translators a nasty looking script with control codes and the like they will probably not like you much). If it is something common like shiftJIS leaving how you got it in is an idea unless you really really have to change it.

Translating. This one is depends on how you and/or your team work: my personal preference for translations from Japanese is not to assume your readers are Japanophiles but assume they are at least knowledgeable (unlike most official translations).

reinserting the text: encoding it is not bad, but redoing the pointers may be. I suggest a spreadsheet unless you want to code your own app.
ShiftJIS is nice as it has ASCII built in (although the extra Spanish characters may prove a problem). The font on the other hand can cause problems and I suggest leaving it until you have got the translation "playable" unless you have a good coder on the team as it can get messy.

reworking the text: you may have to trim lines/sentences to make it appear on screen better or fix bugs.

If you get a team the usual team rules should apply:
everyone should be able to do the job of everyone else, maybe not as well as the person but it should be able to be done.
everybody has tasks/jobs.
in this case everybody should know the basic Japanese text: what are the kana (and the subsets), kanji and romaji and what are they for (you could probably leave out stuff like kana in place of kanji when medical terms are needed).

Thoughts from the frontlines.

Before this document ends with a discussion on the law, the list of links and file formats for reference. Some thoughts from someone who has spent time hacking (me aka FAST6191).
The question is often asked what is useful to be a hacker and what should I know to take up GBA/DS/GC/Wii hacking.

Hacking from the ground up
How to hack.

1) Every feat of human scientific/technological endeavour has gone so far as to require something to be written down. This then becomes science and technology (how far would you have got without being taught things; I would count it lucky if any of us were are laws of motion stage should we have to go from scratch).

2) Ultimately these become standards (hence the learn programming, electronics, engineering).

3) Few people become programmers/engineers for any reason other than they think they can do things better than others (either aiming for monetary rewards, prestige, to help themselves out.....).

4) See 1), everything being so difficult/expensive to make means it is ultimately easier to pervert something to your own ends.

5) See 2) Being so difficult it has to be based on something that is known, secrets do not become known and thus are not so important. Even if it is kept secret from the general public it will ultimately be based on a standard somewhere along the line (see GC memory card being a rebadged SD slot).

6) See 3) people take 2)/5) and use it to achieve 3).

Methods by which to do this.
Electronics. Signals carry information, you have to work out what form this information is in and if/how it derives from that which already exists.
Popular methods include:
patents. Patents contrary to some level of popular belief do not keep things secret but disallow others to use the setup/methods for monetary gain.
Adverts/company portfolios. A company will usually shout from the rooftops that it has a given contract to make things and those that have it made will usually do the same.
Reverse engineering. If you see something in action you can tell how it works if you know where and how to look. Knowing where and how courtesy of the duality of knowledge is necessary to create new things so those that can create can also reverse engineer (although it is by no means a simple thing).
Leaked documents. Usually saves a lot of effort digging around the first three.
Other programmers. See 2) in the list above.

Programming: Everything starts at the electrical level but that is damn hard so it is abstracted to assembler/assembly language.
Assembler is different for every chip and/or setup so that is abstracted again. Usually this means C of some form.
C? is also difficult so it is abstracted again.

Rinse and repeat several times.

At every stage up you lose some level of control to an "intelligent" program and also some of the electronics will be hidden from the programming level. You then tweak the electronics (this would be the "chip" you add to a wii) to allow programming OR you emulate the electronics and conveniently forget to add in the restrictions of the real hardware.

Most hackers have a considerable level of mastery of at least the first 2 (electronics and assembler) and these days being able to pull apart a higher level language from the perspective of assembly is also a damn useful skill (and where I tend to fall down a bit).

Knowing these is how the hackers can do it.
Some links
Electronics:
http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/index.htm

Programming:
Assembly (this is for x86 but it is one of the best guides out there):
http://webster.cs.ucr.edu/

C
http://www.cplusplus.com/doc/tutorial/
http://www.physics.drexel.edu/students/courses/Comp_Phys/General/C_basics/

Eventually the knowledge gets passed on/written down which allows conventional programmers a go at doing things.

There is a reason the general layout of the file formats was included. Nearly every new format or twist upon an existing one uses the principles there.

It was mentioned many times in the document but programmers are people and will reuse code or think about the easiest way to do it (even cryptography tends to almost be an afterthought). This means

ShiftJIS is common enough in games these days to be worth being able to recognise on sight (usually by the abundance of 89 and 90's in the text).

Data alignment is important, even though you can deal with it quite easily it is not worth the effort most of the time. This means programmers will tend to align data to boundaries.

It is worth being able to do addition and subtraction in hexadecimal in your head, likewise endianess is a problem that will not go away so being able to shift it in your head is a worthwhile.

I hack many games (often as they are released) just to see how they are put together, often nothing other than the occasional example comes from it but the experience is invaluable when it comes time to hack a game “for real”.

Information comes from everywhere, being able to search and scan through it is a useful ability to have.

Along with text encoding and file formats compression is useful to be able to see. LZ is without a doubt the most common method.

Law and rom hacking.

Should you find yourself on the receiving end of a bit of legal hassle it is advised you find someone to deal with it, this section is just a discussion.

The whole issue of rom hacking and the law (or even just intellectual property law) is very convoluted (the fact that there are entire law firms dealing solely with such matters should be enough to demonstrate that) with lots of holes and loops for all parties concerned. To this end this section serves as little more than a thought exercise but hopefully it will be a valuable one.
This section will generally be European law contrasted with US law.

The current climate means people will tend to try to avoid a court case as a legal defence/ attack is costly to mount (how many reading know how to address a judge and what your rights of appeal, evidence law or even know the grades of witness let alone anything more than that and with the fact it changes country by country and state by state).
Broadly speaking however there are two areas of law applicable here, to use the US terminology these are civil and criminal law.
The former deals with two (or more with two groups being formed from the people concerned) parties trying to settle something which is ultimately enforced by the countries law enforcement while the latter is law according to the country you are in (and there are various levels of that, criminal law in the US being above misdemeanour but below federal). The most common in the circles dealt with here is civil law and is usually what is used for the likes of copyright violations and DMCA notices while criminal law can appear to start proceedings (as an example in the UK civil cases have less evidence gathering ability (for instance they can not compel and ISP to match an IP address to name) so a criminal case can be started (which can match IP to names) and then dropped. Criminal law will however appear when large sums of money are involved or with matters of cryptography, security (financial or physical) of an area, institution or the country.

There are also two types of law in as much as there is "book law" (what the government puts out in whatever method is used in the country/continent of choice) and case law (what a judge can interpret the law as for book law can be a bit vague, plain incorrect (how many politicians know electronics and computing to the levels required) or behind the times, a matter of great importance in the computing/electronics world where things like moore's "law" exists: old cryptography law* said no more than 56bit encryption in certain cases which was fine for the time but a modern home computer can crack it in seconds or less which makes it no use whatsoever and when new law can (quite rightly as a rushed law is usually full of holes) take some time to be debated). On the flip side case law also affords a measure of protection and there have been notable cases where people like the RIAA have dropped a case so as to prevent case law from being formed on an issue that would later complicate their activities. The word most likely to be found when researching case law is "precedent" which is usually found where a similar (or the same) situation has previously arisen yet is not detailed in "book law", however (and it gets complex) a judge can choose to follow/act on precedent or ignore it (and being accountable to people means there should be justification for the actions of a judge).

*more on cryptography later but suffice it to say cryptography is considered munitions (http://rechten.uvt.nl/koops/cryptolaw/cls2.htm#co) and is afforded a certain position in the

law. For a crash course in cryptography there is a section above.

So some definitions:

Copyright:
A very broad array of things are covered by this so some clarification is needed. Ostensibly it deals with a specific implementation of something. The intended purpose is to grant a monopoly to the copyright holder for a period of time (the length of which is a matter of extensive debate) so as to enable them to gain funds from it and make further works/retire/whatever. After this time people are free to do what they like (within the law anyhow) as it falls into public domain. There is an exception for personal use ("if I can build it") but it gets complicated when giving away or finances are involved.
In computing world however there are only so many ways to do something effectively (it is entirely possible to make a 5000 megabyte program to add 2 single digit numbers stored in the program but nobody would do it as it is trivial to do simply) so you are allowed to have the same implementation provided you did not copy source code from someone else (the usual method to tell is to look at the mistakes made, people match code but rarely mistakes and as a code copier is usually lazy.....).
Another interesting note is levels of rights available. Here a company can give/sell rights to part of something for various reasons (although thought not strictly copyright the original people behind the making of the transistor allowed hearing aid companies to not have to pay) and is the reason for the "all rights reserved" line of text you may see in places. Normally all rights are reserved but there are many notable exceptions as far as hacking and old games are concerned (stuff like half life mods and a lot of the spectrum stuff http://www.worldofspectrum.org/permits/publishers.html as simple examples).
A term that can apply here is the so called "orphaned work". If you were to look at your job/university contract (for something that requires a creative output) you will probably see a clause giving your company the rights to things you make (sometimes even in your own time). However companies go bust or simply sell the rights to something. Eventually it may to pass that nobody has any claim to the works and thus things can happen. Like most areas of copyright law this is a hotly debated area so read up (this is not so important for DS roms but for older systems someone may like to port a game from (length of time for software copyright is multiple decades in most places).
Sidenote/talking point: Antigua recently suspended all copyright for US concerns and had it upheld as justified:
http://www.nytimes.com/2007/12/22/business/worldbusiness/22gambling.html?_r=2&oref=slogin&oref=slogin .
The real matter here is the word specific implementation, you may argue that a rom hack is your own work and thus you are free to do what you will with it. However the copyright law creators thought of this and in actuality a rom hack is most likely a "derived work" which is as the name implies a work built from a/upon a copyrighted source. However companies tend to ignore such efforts from this standpoint (compare it to time shifting mainly in VHS era in the UK) assuming no money is charged but ripping sprites and using them again will tend to get you noticed. Similarly if the hack is likely to damage the company/brand image then they may act but this usually comes under the auspices of trademark law (see below). A known example of derived works being subject to such things is microsoft against the DOA volleyball title hackers on the original xbox.
Companies naturally dislike the spread of roms so patches tend to be distributed containing only the data from the hack (saves on bandwidth too). The interesting part comes when data from one rom is swapping into the other as is the case in undubbing games (where a different (but official) soundtrack is swapped for the same game (usually a different region or for censorship purposes).

One more point is the exceptions permitted, education and criticism purposes mean copyright can be ignored (reverse engineering is usually taught as part of engineering) and there is also fair use. This is a minefield but it basically boils down to rights given to make backups of data and certain other things being able to overrule copyright. The DMCA analysis at http://www.doom9.org/ is a fair starting point (the whole site is based around the concept of fair use when it comes to copying/altering video and as such is not too far removed from rom hacking and purposes of this guide).

Patents

First this section will only be defining software patents at any significant depth, general patents are a different legal entity. The definition is a method of doing something (that is non obvious and original as well as having to work: see perpetual motion machines and patents) will be given a monopoly for reasons much like copyright above. The difference is method being granted a monopoly rather than a specific implementation although there is some overlap in the case of drug companies (patents are given to concoctions which is close to a specific implementation).

US patents are generally he who makes it first (and can go some way to proving it) while European patents are first past the post into the patent office (see early telephone patents). The mere fact the the US and Europe is mentioned should indicate that there is a regional difference although there is a fair amount of similarity owing to international meetings and agreements.

More on this:

http://www.iusmentis.com/patents/uspto-epodiff/

Software patents are a matter of huge debate, the main argument against them is software is maths and to patent maths is not the done thing (http://eupat.ffii.org/log/intro/).

Arguments for are copyright as mentioned is for a very specific thing; it is well known google the big search engine/ad broker uses a method by which the number of links to a site determines what rank (how far up the list it appears) it gets. The actual algorithm used is not able to be used by others but there is nothing stopping anyone else using the same concept and thus robbing them of business "just as good as google". The US has software patents while the EU does not (perhaps overly broad, more info here: http://www.law.duke.edu/journals/dltr/articles/2003dltr0006.html and http://eupat.ffii.org/log/intro/).

As far as rom hacking is concerned the only real interest comes with things like patented formats for multimedia but little has arisen in this matter (a media format lives and dies according to how many people support it so a patent is usually only enforced to stop what is perceived as rogue use by applicable parties.

Sidenote many cases arise where overly broad patents are defined ("I have a new invention for a circular thing upon which things can be moved for great distances") and then used to cause hassle. Such an action is generally known as patent trolling and an example could perhaps be found in the various controller lawsuits over the years (of course a small company holding a patent could be doing so legitimately).

Patent pending and patent granted (both usually with a number so it is possible to look it up) are used to refer to processes still having their patents checked and those given a patent. Both are afforded approximately the same legal protection but the former is given while the patent application is being checked (there are millions of existing patents, new patents and not so many staff).

Licensing

In computing this is an agreement between the person wanting to do something and person "selling". There are millions (or more) possible licenses (often minor tweaks on existing licenses) and the only thing they have in common really is to restrict what you can

do. Broadly speaking there are two sorts of license the "copyright" license and the "copyleft" license. Copyright tend to be for closed source applications people try to charge money for while copyleft tend to be associated with free products but there are countless examples of things in both camps and neither.
The interesting part is whether they are applicable to situations and how they are enforced (that is to say are they overstepping their mark: a license can not demand your head if you do something wrong). The capital punishment example is a bit extreme but in reality an example could be a license prohibits reverse engineering but the DCMA (usually given as an example of a badly thought out copyright law, it is a US law although many places in the work have similar laws these days) takes precedence and has a clause saying reverse engineering OK if it is for compatibility purposes. Other examples include right of resale/transfer of ownership:
http://www.channelregister.co.uk/2008/05/23/ebay_autodesk_ruling/ (this is recent but in the past people paid for companies thus gaining them but keeping the original company on paper only meaning keeping the software from that company was possible).
When it comes to copyright licenses the law is very unclear as the above examples begin to detail and in the case of copyleft there is almost no law or cases of prosecution (successful or otherwise) for failing to comply with it (see also the software as a service aka SaaS debate).
Another interesting fact is that in many places a license is invalid if you present it after the point of purchase (and the 30 minutes between the shop and your house....).
With most licenses prohibiting reverse engineering it is fairly obvious where rom hacking appears in that (rom hacking being applied reverse engineering). Other aspects include licensing for distribution which is one of the headaches facing people bringing games out of Japan if they feature multiple anime/manga stars.

Trademark/servicemark
This is the name and general likeness of a matter (trademark is physical while servicemark deals with a service). With regards to this the tetris company has been fairly active in this arena (to the point where even alterations of the name are gone after, a preemptive action but see tepples' tetnus on drugs: see may 25th 2008 posting: http://www.pineight.com/) but there are precedents in things like the giana sisters: http://www.worldofspectrum.org/infoseekid.cgi?id=0009800 and more recently blizzard and the starcraft homebrew app for the DS). Outside of computer games the superbowl (the final match in the American football NFL season) has been known to use trademark law in a manner many would regard as overzealous.

Cryptography and reverse engineering
As mentioned cryptography is considered munitions by most countries in the world (http://rechten.uvt.nl/koops/cryptolaw/cls2.htm#co) and is afforded a certain position in the law different to the copyright talked about up to now. For a crash course in cryptography see the cryptography section in this guide.
It stems from governments facing challenges policing the people (conspiracy theorists would say tracking and perhaps that is not entirely incorrect) but these days laws exist to force decryption keys to be given up at the behest of law enforcement (http://www.theregister.co.uk/2008/10/14/ripa_self_incrimination_ruling/).
At the more extreme levels rom hacking deals with matters of cryptography and thus run afoul of the various regulations in place concerning matters of it. Most commonly this would be laws saying decryption of stuff that is not yours/or you have the permission of the people concerned is not OK.
In the case of the wii this is the so called common key and is in actuality the product of two very large prime numbers. Owing to bugs in the decryption it is possible to fake a signed

(signing is a cousin in both the physical and legal sense of cryptography) file and thus execute unsigned code.
The key is an example of the so called public key part of the cryptography, the usual line given to stop the sharing of keys is decryption being illegal under the laws.
However much like the DMCA and reverse engineering above (which has been used to argue for cryptography related matters in the case of the deCSS code) laws may allow for ways around it. For example the US has a law regarding free speech (with certain provisos) with speech also including certain actions and consequently code for computers (and freedom to do legal things on computers like say a homebrew channel).

Reverse engineering is the action of taking a finished product (or something beyond the fundamental plans stage) and figuring out how it works or deriving the specification if you prefer. Some however (and these can include those very well noted in the field) define it as merely taking the shape (or in the case of computers input data and how it manipulates it) and then anything beyond that as re-engineering or something else. This is usually prohibited under license conditions and some copyright law but the importance of interoperability (see standards/ISO organisation) and so do certain laws so that is one exception. Academia is another exception and it can again get very complex. Generally speaking there are no cases from rom hacking specifically where reverse engineering has been the angle of choice for which something has been gone after. Firmware/bios reverse engineering has been (usually as a means to an end) and so have some "chips" for consoles.

Warranty
Most readers will probably not care all that much about such things but suffice it so there are two types of warranty. Warranty as afforded by law that protects you against defects in workmanship (note this can exclude design to some extent) and the warranty given by the manufacturer. There is some overlap but the law based one is generally shorter and lacks provisos like the “disassemble this and warranty is void”.

List of links.

The following is a list of links that have appeared in this document an a sentence on them if necessary. Order is approximately what order they appeared in.

File specifications.

To be added in final version. Existing specifications (external links) are available in the tools section and throughout this document.