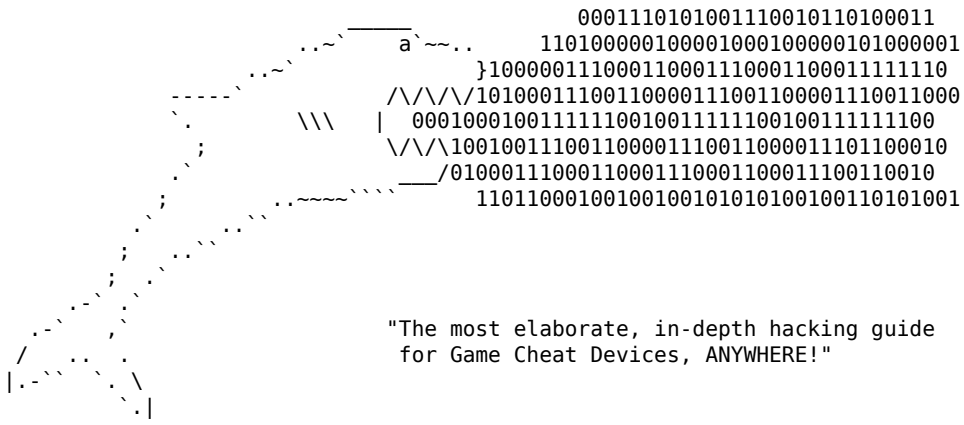
```
 ____    ______   ______      THE      ______   ______   ____
/    \  |   ___| |      |   _____   |   ___| |_    _| /    \
|    |__| |  |__  |    |__| |     \  |  |__      |  |  |    |__|
 \   \    |   __| |    |    |   |  | |   __|     |  |  |    \   \
 _\   \   |  |    |    |__  |      /  |  |        |  |  _\    \   \
|  |    | |  |___ |    |  | |    | \  |  |___     |  |  |  |    |
 \____/  |______| |______| |___|__\ |______|    |__|   \____/
                Of Professional Gameshark(tm) Hacking
```



[Originally By: Kong K Rool* and Macrox]
with additions by [Tolos, DGenerateKane,
HyperHacker, Viper187, and Kenobi]

Table Of Contents

Section 1 : Foreword

I) This Version

```
    Version Number:    5.00c
    Release Date:      03-25-2003
    Edited by Tolos, Viper and macrox
```

* = aka Parasyte

II) What's New

03-02-04

minor update:
- -Hacking "unhackable" GBA master codes
- -Bitwise Operations info
- -A little basic info on using COP1 instructions (N64 ASM)
- -IcyGuy revamped his Image Mods and GBA Size Mods info
- -MAX Crypt, GCN Crypt, and GCN Code Type Helper downloads added

06-27-03

"Secrets..." goes solely HTML!
- -XPloder 7K decryption algorithm
- -New AR Crypt info added
- -Added PS2 encryptor program (javascript)
- -Added Z64 and V64 info in the GameShark FAQ
- -Added N64 Emulator Based Hacking guide
- -Updated N64 ASM Hacking
- -Updated the Downloads section
- -More Downloads Added
- -Image Modifiers

05-31-03

Tons of new info added:
- -codetypes updated/added for all systems
- -new N64 enabler information
- -More complete Button Activator/Joker info
- -New method of hacking N64 Button Activators
- -Hacking Timer codes
- -Quickstart/Skip Into codes
- -CPU <> P2 Control Modifier Codes
- -New Moon Jump Method
- -X/Y/Z Coordinate Modifiers
- -Floating Points info
- -Encryption information PS2
- -AR Crypt for GBA: Program, V3 code types and guide.
- -PS2 hacking
- -Info on DC hacking
- -Huge N64 ASM Guide
- -GBA ASM Hacking tutorial
- -GBA Size Mods
- -Brought the GS FAQ up-to-date
- -Greatly expanded the TOC
- -Added a few mini-TOCs in places they'll be useful

05-21-03

Tolos finally manages to pull an update together.
- -Added PS2 code type info
- -GSA and CBA info
- -Added N64 control stick activator info.
- -Added links in the Table of Contents for easy refernce (HTML version).

01-01-03

Macrox fixed the errata of missing code types for GS and CB GBA.

05-20-02

Tolos assigned new acting editor by macrox.
- -Hacking GameBoy Advance enabler codes
- -GameBoy Advance code types

-Added a tidbit on hacking debug codes
-Added a section on hacking GameBoy Advance codes.

11-27-01

Interact Accessories affiliation in legal section removed. Reference to Gameshark removed from work title. The reader is to Infer the term gameshark, codebreaker and xplorer to mean cheat device where it occurs in this work. See legal section for proper credit.

04-21-01

-Revised section on N64 and XP code types
-DC code types and buttons.
-Comparison of GameBoy hacking devices
-Reflashing a GameBoy Shark.
-More on hacking enabler codes and forcing high and low mode res
-DC code types: CodeBreaker, Xplorer and Gameshark.
-How to Hack Speed modifiers.
-Update on Keycodes list
-Revised FAQ section.

11-11-00

-Reflashing a corrupted GameShark
-Info on Pelican's new hack device for Game Boy, "Code Breaker"
-Gameboy hacking info courtesy of Curly9od

05-06-00

-How to hack specific codes section appended
-Using the memory editor has been appended
-How to install the pc hacking utilities, hooking the GameShark to a PC and upgrading (flashing) the GameShark Rom added.
-How to hack enabler codes
-Hacking walk through walls (WTW) codes
-Link to online code porter (when text viewed online).

07-22-99

-New chapter added - Chapter 4 - GameShark / GameShark Pro FAQ.
-New "How-to" added, for the Big Time Hackers.
-Decimal/Hexadecimal conversion formula added.
-"This Version", "What's New" and "Coming Soon" sections added.
-Many new code type prefixes added.

III) Coming Soon

Gamecube Hacking
Caetla Code Types
XP64 Code Types
Saturn Emulator Hacking
CodeBreaker2 Code Types
TBA - To be announced topics.

IV) Dedication - by Kong K. Rool

This document, in all its entirety is dedicated to my dad. A very brilliant man in the computer/science fields. He passed away July 9th of 1999 at the age of 36. May he rest in peace.

- Kong K. Rool (aka Parasyte)

V) Preface

First off I want to say, Macrox has semi-retired from the hacking scene. That left no one to tend this marvelous document. Then Interact suddenly stopped hosting GSCentral when they learned that their GameBoy Advance encryption code had been cracked, and the people at GSCentral were creating codes that worked with the GameBoy Advance GameShark. After GSCentral came back, we learned we could not use the word GameShark (TM), or have the Hacking Text displayed.

So I volunteered to host this document. I hope I can do a good job; as well as Macrox and Parasyte: that is my aim. I also want to thank macrox, HyperHacker, and DgenerateKane for helping me along.

Happy hacking,
Tolos (Assigned Editor in Chief)

P.S. The Hacking Text has been re-instated on GSCentral. A mirror copy of this document will be kept on Tolos web sites. These are the only officially endorsed sites by macrox on the most up to date versions of the text.

- macrox 1-1-2003 (Editor - retired)

VI) Acknowledgments

Many talented people have contributed to this work over time either directly or indirectly. To those people we say thank you for all your contributions to the world of video games and for hacking codes and sharing ideas on how to hack codes.

People whom have shared ideas and contributed information for this document:

Kong K. Rool (aka Parasyte)
Macrox macrox_the_sage@yahoo.com
Tolos tolos_magician@yahoo.com
HyperHacker
DGenerateKane
ShadowKnight
Jim Reinhart (GSCentral Founder)
Code Master
Kamek
Freeza
Subdrag
Viper666(187) viper@gscentral.com
Sutaz
james007
Gold64007
Stinky613
Crocc
Zap2
CodeBoy
Savior
Charizard
Dr. Ian
Curly9od
Bleeding Gums Murphy
Kola
FoxDie
ARHQ - our AR PRO replay affiliates.
Kenobi
Icy Guy
Goldenboy

This list goes on and on, we apologize for any omissions of people who gave of their time to advance the art. The authors want to thank everyone at GSCentral, Game Shark Zone, Game Shark (Software) Code Creators Club, Gamasutra, Dextrose,

Interact, MadCatz, and Datel for fruitful discussions.

Section 2 : Hacking

VII) Introduction

Welcome to the GameShark World. In this document, you will learn several different ways to hack your own codes. These methods range in degree of difficulty from easy to difficult and yield various results. As you read you are encouraged to practice the methods that are described in this document. This way, you will learn by your activities.

There is more than one way to hack codes. This ranges from guessing, to a secret hacking system, which not everyone will understand. Again, as you read, try the ideas mentioned. It has been proven that people learn easier and faster when they are active in a project.

It is the authors wish that the material presented here meets the anticipated needs of the reader's wishes to learn to hack GameShark codes. You might even come up with another variation of these methods to hack codes. If you do, let us know and we will consider including it in future versions of this document.

VIII) Hacking Basics

A) Know the Code

Offsets

Offsets(or RAM addresses) are typically found by examination of the game memory by using advanced hacking equipment such as a GS Pro, Shark Link or hex viewer on a ROM. Basically, an offset is a "memory holder" in which it (usually) holds a byte of memory(a two digit hexadecimal code).

If you find an offset that holds the health digits when using a ROM and hex viewer, you can be certain that it isn't the GS code(if you find the health at offset "012203" the GS code usually won't be "80012203 FFFF".) There is less than a 1% chance of finding an offset and it actually being the code. The reason the offset and the offset digits in a GS code are not the same is this - There are MANY, MANY offsets which are used to tell the platform what type of game it is(size, language, title, checksum values, etc.), and other operation codes which will assign all the offsets to do what they are meant to do. There are offsets that hold the hex values that make up the pictures you see in the game, the coordination's of the character you control, mathematical operations... The list goes on and on... The header(tells the machine what type of game you're are booting) might take up all offsets past "012203" itself!

There is a block of info that tells where the quantifier-offsets(the byte of memory which you change through GS codes begin and which are usually things such as number of an item you have or level of health you have). This block is called RAM(Random Access Memory), which does exactly what it says. RAM is memory that will be changed all throughout its processing. Score and health are good examples of RAM, the values for both will be changed while you're game is running. So think of GameShark as a RAM Editor.

More information about offsets is beyond the scope of this document and will not be included in future editions of this text.

B) Systems of counting or number base

B-1) Decimal

Decimal Notation, based on ten digits, is something you already know. Count to 50 like you normally count. You can count using decimal notation.

B-2) Binary

Binary, or dual counting, is based on two digits. It's really easy to understand and use. You'll need to know the following -

There are two characters used in binary - 0,1
(Think of it as a switch).
A "1" means the switch is turned ON.
A "0" means the switch is turned OFF.
That's what binary is, a bunch of switches. I won't go into any more detail about switches now, but will return to this topic later in the document.
A four-digit string of code written in binary is called a "word".(this is also the same in hex[1-digit])
Four Binary Digits(bits - 'BInary digiTS') equals 1 digit hex.
Three bits equals 1 digit octal.

Now that you know that, hex and octal should seem easier to learn. In this document, we will refer to any and all hex values with "-h" and decimal values with "-d". So value "100" decimal will read like this - "100-d" and "64-h". How do you convert from bits to hex and back? Look at this chart -

Hex		Binary	Hex		Binary
0	-	0000	8	-	1000
1	-	0001	9	-	1001
2	-	0010	A	-	1010
3	-	0011	B	-	1011
4	-	0100	C	-	1100
5	-	0101	D	-	1101
6	-	0110	E	-	1110
7	-	0111	F	-	1111

If you notice, there are no more 4-digit combinations of "0,1" left. Now for the conversion part. Look at the 4 bits, each of the numbers have a value assigned to them. We will call these values, "Bit Values".

Number in Hex	6
Number in Binary	0110
Bit Value	8421

(The Bit Value will ALWAYS be this! So remember it!)

You are going to learn to convert by using multiplication. Math is a great tool to use when working with the GameShark.
You can represent the binary word by letting "0110"(8421) = "IJKL" and thus you get "1xL + 2xK + 4xJ + 8xI" = "L+2K+4J+8I"(in algebraic terms). Now substitute the binary back in, you would get "1x0 + 2x1 + 4x1 + 8x0" = "0+2+4+0" which adds up to six. Six is what the hex value was in the beginning.
To convert back to binary, use the formula "L+2K+4J+8K", find the numbers, which add up to six. In this case, "4 and 2". Remember, "IJKL" = the bit value. Then substitute the binary back in - "1x0 + 2x1 + 4x1 + 8x0" = "0110".
Why do that when there's an easier way? Because there is no use in converting when you don't understand why it is done in that way. You will learn an easier way soon, in fact, make one up!

Octal conversions are the same as hex-to-bit. Only, octal goes up to "7". So the bit value looks like this -

```
Octal     3

Binary    011
Bit value 421
```

The Bit Value will NEVER change. The bit value is actually the value assigned for each bit. If you have an 8-bit value, the bit value would look like this -

```
(128)(64)(32)(16)(8)(4)(2)(1)
```

Notice that every time a new bit is added(to the beginning, no doubt), the last bits' value will double. Further explanation is beyond the scope of this text.

To convert between hex and decimal, use this formula -

```
yz = 2-digit value hex
(when) z = "A-hex", A = "10-dec"
(when) z = "B-hex", B = "11-dec"
(when) z = "C-hex", C = "12-dec"
(when) z = "D-hex", D = "13-dec"
(when) z = "E-hex", E = "14-dec"
(when) z = "F-hex", F = "15-dec"
(if) z = #, skip next step
z-hex = z-dec, z-dec = q
(if) y = #, skip next step
y-hex = y-dec, y-dec = r
y*6 = s
yz+s = yz-dec
```

This looks confusing, I know, but I'll explain it as if I were talk to a 10-year-old child.

First, "yz" represents a 2-digit hex value. Our value will be "64"(y=6, z=4). When "z" is an "A", A equals "10-d". Understand that so far? If "z"(in the 'yz' hex value) is a number, skip the next step(4 is a number, so we skip this next step). Transfer "z" to decimal(look at the "when's"). If "y" is a number, skip the next step(6 is also a number, we skip the next step). Transfer "y" to decimal(look at the "when's"). Multiply value "y" by 6, the factor is "s". 6*6 = 36, s=36. Add value yz and value s. 64+36 = 100 64-h = 100-d.

Now let's do it short-hand -

```
"yz" = C8
 C = 12, yz = (12)8
 12*6 = 72
 (12)8 + 72      [7]2  --  72-d
               +(12)8  --  C8-h
               ------
                [20]0  --  200-d

 C8 = 200
```

If this doesn't make sense, I didn't explain it well enough. It is important to understand how to do the number base conversions before continuing. If you do not understand, the reader is encouraged to review the material already presented.

B-3) Bitwise Operations

You may hear about "Bitwise Operators" and wonder what some of them actually do. They're used for doing binary math, for lack of a better explanation.

& (AND)

The AND operation can best be understood like addition.. only there's no adding or carrying involved... really the only similarity is that you work with each digit the same as you do with addition...

Like this:

```
 1000
+0001
------
1001
```

Just add each digit downward. Well, AND requires you to work with each digit downward as well. Here are the rules:

```
1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0
```

That means the result will be a 1 ONLY if both comparing digits are 1
Think of it as true and false.. if TRUE & TRUE, then TRUE.

```
 1100
&1010
------
1000
```

Starting from the left-most digits, 0 & 0 = 0, 0 & 1 = 0, 1 & 0 = 0,
1 & 1 = 1.... and there's your result.

You can use bitwise AND for a technique called MASKING. Masking allows you to strip certain bits, while saving others. Say you wanted to strip the upper nybble of a byte, and save only the lower nybble... Well, you AND that byte with 00001111b. When you do that, the upper 4 bits will be completely stripped, because 0 & anything is always equal to 0 and the lower four bits will copied over directly, because 1 & anything = bits that were set. This can be useful when dealing with hex numbers as well. Say you have AC1B02FF and you want the lower four bits for some reason. AC1B32ED AND 0000FFFF = 000032ED

| (OR)

When using OR, TRUE or anything = TRUE

The rules:

```
1 OR 1 = 1
1 OR 0 = 1
0 OR 1 = 1
0 OR 0 = 0
```

OR is used to set bits... whereas AND is used to clear. So, if you wanted to set the least significant bit, your could do "BYTE OR 00000001"

XOR

XOR means EXCLUSIVE OR. it's purpose is to inverse bits.

The rules:

```
1 XOR 1 = 0
1 XOR 0 = 1
0 XOR 1 = 1
```

0 XOR 0 = 0

Pretty simple here... it works just like OR, with the exception that 1 XOR 1 = 0. Say you have a flag, and you want to toggle it on and off. You can do "VAR XOR 00000001" and it will inverse it; it will turn it on if it's off, or it will turn it off if it's on. That's much easier than doing "if VAR = 0 then VAR = 1, else if VAR = 1 then VAR = 0." Just a simple XOR operation and you're done. Much faster.

NOT

NOT works EXACTLY like XOR with the mask completely filled with 1's NOT will inverse the variable completely. All 1's are changed to 0 in the result, and all 0's are changed to 1 in the result
10101010 XOR 11111111 = 01010101
NOT 10101010 = 01010101

What's the difference?

As you can see, NOT does not have a mask... so you just say "NOT VAR" and you know it's the same as "VAR XOR 11111111"

The difference could be speed. Especially in assembly, where you might have to load 0xFFFFFFFF into a register to perform the XOR; you could just do a simple NOT instead.

Inversing is great because you can negate numbers with it NOT VAR + 1 = -VAR

Take for example...

NOT 00000001 = 11111110
11111110 + 1 = 11111111
and of course, 11111111 = FF, FF is -1

<< (Left Shift)

Shifting left works like multiplication when you shift left, all right-most bits get shifted over to the left, and 0's get shifted into the blank spaces.

Here are some examples:

00111111 << 2 = 11111100
00000001 << 3 = 00001000
10000000 << 1 = 00000000

Just shifting digit places. So, shift left by 1 is the same as multiplying by 2, left shift by 2 is the same as multiplying by 4. shift left by 3 is the same as multiplying by 8, etc. Left shit works best as a means to multiply by a power of 2. 2^1 = 2, 2^2 = 4, 2^3 = 8, etc. Shift left x is the same as multiply by 2^x.

>> (Right Shift)

Right shift works the same, only opposite, so it's like dividing. When right shifting, bits shifted off of the right side are completely lost, and bits shifted in from the left come in as 0. Well, they usually shift in as 0 when right shifting. In MIPS, you may have noticed "Shift Right Arithmetic (SRA)" and "Shift Right Logical (SRL)." Shifting right logical will ALWAYS shift 0's into the new spaces. Shifting right arithmetic will shift the MSB (most signifigant bit) into the new spaces. The MSB is treated as a sign bit with arithmetic right shift. That's a way to preserve the sign when you divide a negative number.

Here are some examples...

SRA = Shift Right Arithmetic, SRL = Shift Right Logical:

10000000,00000000,00000000,00000000 SRA 4 =
11111000,00000000,00000000,00000000

10000000,00000000,00000000,00000000 SRL 4 =
00001000,00000000,00000000,00000000

B-4) Hexadecimal

Hexadecimal is a programming 'language' you must know in order to hack GameShark codes. So, what is it? Hex is what your GameShark codes are written in. There are sixteen characters used in a GS code. The characters are as follows -

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

Let's learn to count in hex. If you look above, you'll see how to count to fifteen in hex. What's sixteen? "10-h" is sixteen. By the way, don't say "ten," say, "one, zero" for '10-h'. Let's see what you've learned.

What comes after 19?
What comes after 3F?

If you said, "1A" for question #1, that is correct.
If you said, "40" for question #2, you should know how to count in hex!

B-5) Octal

Octal is just another way to write binary(like hex), but octal words are longer than hex words. What you've already read is enough to know about octal. You do not need to know octal to hack GS codes.

B-6) ASCII

ASCII is what you are looking at right now. ANYTHING that can be typed on the keyboard is ASCII. ASCII is useful to know when hacking in my secret way. It is also useful for using the text editor search option in hacking devices such as the GameShark PRO. You don't need to know the assignments for ASCII characters(although you might need to know them when hacking text editing codes).

B-7) Floating Points

Floating Points are a hexidecimal representation of "real numbers", usually following the IEEE-754 standard. This could be considered an advanced topic, if nothing else, because even some long time hackers I've mentioned it to have never heard of them. However, even those who haven't heard of them, have most likely dealt with them in one game or another. To put it in the simplest terms possible: Floating Points are numbers with decimal points. 100.0 and 100 are common values used by games to represent your max health. The difference is 100 is 64 in hex; 100.0 is 42C80000 in hex.

In hacking terms, Floating Points can make it difficult to find some things, if you're only using 8-Bit comparisons. I'm not saying 32-Bit comparisons are required though. Most things that use Floating Points are still found by 16-bit searches.

So how can you convert those hex values like 42C80000 to their decimal form? This is a question I've been asking for a while now. I'm told it involves advanced math functions like Shifts and XORs. Don't get worried though. As with most things that require much thinking, somebody wrote a program to do the conversions for us. You can get FloatConvert here. Nobody really knows who wrote this, but I thank that

person whoever they are.

It's not required you know anything about Floating Points to hack most codes, but they are a major part of games and certain, more advanced, code types can be harder to find if don't have a little understanding of this.

C) About Most GameShark Hackers

Most hackers use more than one way to hack. Most know programming languages such as binary/hex/octal, HTML and Perl, scripting languages, R300 Instruction sets, etc. HTML and Perl is included here because many hackers want to use this language to create a website that has all their codes displayed. You don't need to learn many of the things other than binary and hexadecimal to "hack better than the Pros."

Here are some ways you can hack(1 star[*] by the name is easy, 2 is harder, etc.) -

Guessing****
This isn't easy, because you don't always find a code this way. It's not only troublesome, but risky at some times. Some guessed codes can corrupt game data and corrupt hacking devices.

Modifying Codes*
One of the easiest things to do. Change a number on an existing code, you make a new code. (Only works when you have a basis[base code] to work with.) This technique was and still is widely used.

Looking At The Source[Code]***
Hard, but most effective. Worth a shot. This technique requires the reader to have a working knowledge of dissembler programs and is currently beyond the scope of this work.

Using Hacking Equipment* - ***(Code Generators; i.e. GS PRO)
Another way to get codes. It might not be very easy, but it doesn't take much time. Plus, it's the second most effective way to hack. Can be used to get easy to hard code types.

Porting**
Porting is taking a code from one version of a game, and making it work on another version of that same game. This does not always work. The reason is the same reason that they make more than one version. Possibly to fix a minor bug. So the offsets will be in a higher or lower position, or even moved to a totally different location. You can use the "GS Code Porter"(available at GameShark Central) to port any code for you. Hence you can make a code before anyone else gets the chance! See FAQ section.

Combination Hacking* - ***
All you need to do is hack using two or more methods at once, for a greater chance of finding a code.

(!!!!!There is a slight risk of loosing saved data on your GS when turning the system on and off while guessing or modifying codes.)

IX) How-to Guide - Getting Started

A) What do I need?

You need a Game Platform(N64, PSX, Dreamcast, Game Boy, etc.), you need a game to hack, and you need a GameShark,GameShark PRO or GameShark CDX and/or other hacking devices such as the Blaze Xplorer/Xploder or Pelican CodeBreaker. You will also need other things to hack in other ways such as a hex editor and calculator can do hex math or perhaps an emulator that can capture memory dumps or can do game saves all of which can be analyzed later by some hex editor

program. A will to learn, basic math skills, patience, endurance, practice and this using this document are the things you need.

B) What do I need to Know?

If you've managed to read everything above, then you know about basis of everything you'll need to know about hacking. The remainder is learning to use hacking equipment and the software for it and to master the basics as presented here and then from there to be resolved to try as best you can to hack the harder code types.

C) Learn Your "Shark" and code types

There are a few things you'll need to know about your GameShark before we begin hacking. You'll discover how the GameShark codes work.

This is the layout of all N64/PSX 'Shark codes -

XXYYYYYY ZZZZ

This is the layout of all GB 'Shark codes -

XXZZYY-YY This format is called endian. Most Memory ram dumps are in this format. The GB Gameshark rewrites the code to reversed endian format as XXZZYYY. The Pelican CodeBreaker uses unencrypted endian format. The Xploder uses both encrypted and decoded reverse endian formats.

"X" digits is the 'prefix', used to tell the GS what kind of code you are using.
"Y" digits are the 'Offset Digits', used to locate the offset digits you want to change.
"Z" digits are the 'Quantity Digits', used to change the quantity of the chosen offset.

The reason there is a hyphen between the Y's in the GB code layout is because the Y's are actually set up backwards. The first two Y's should be behind the second two. Of course, you only need to know this if you are going to use my hacking method... You ARE going to try it, right?

Note:
GameShark and Action Replay code types are the same.

Nintendo 64 Code Types		Compatibility
8-bit Constant Write	Constantly writes the value specified by YY to address XXXXXX. This and its 16-Bit counterpart below are the most used code types on N64. You'll probably be making most of your new codes with them.	GS/XP64
80XXXXXX 00YY		
16-bit Constant Write	Constantly writes the 16-Bit value	GS/XP64

81XXXXXX YYYY	specified by YYYY to address XXXXXX.	
8-bit Uncached Write	Constantly writes the value specified by YY to the uncached address XXXXXX.	GS/XP64
A0XXXXXX 00YY		
16-bit Uncached Write	Constantly writes the 16-Bit value specified by YYYY to the uncached address XXXXXX.	GS/XP64
A1XXXXXX YYYY		
8-bit GS Button	Writes the value YY to address XXXXXX ONLY when the GS Button is pressed.	GS/XP64
88XXXXXX 00YY		
16-bit GS Button	16bit version of the above. Writes the value YYYY to address XXXXXX ONLY when the GS Button is pressed.	GS/XP64
89XXXXXX YYYY		
8-Bit Equal To Activator	Execute the following code (ZZZZZZZZ ZZZZ) ONLY when the value stored in address XXXXXX is equal to YY.	GS/XP64
D0XXXXXX 00YY ZZZZZZZZ ZZZZ		
16-Bit Equal To Activator	Same as above,	GS/XP64

	only it reads a 16bit value. GS Pro 3.0+ ONLY!	
D1XXXXXX YYYY ZZZZZZZZ ZZZZ		
8-Bit Different To Activator	Execute the following code (ZZZZZZZZ ZZZZ) ONLY when the value stored in address XXXXXX is NOT equal to YY.	GS 3.0+
D2XXXXXX 00YY ZZZZZZZZ ZZZZ		
16-Bit Different To Activator	Same as above, only it reads a 16bit value.	GS 3.0+
D3XXXXXX YYYY ZZZZZZZZ ZZZZ		
Disable Expansion Pack	Keeps the game from using the expansion pack if it is present. Also used on some older non-expansion pack games to increase compatibility with the code generator.	GS Pro 3.2+
EE000000 0000		
Disable Expansion Pack	Disabled the expansion pack (if present) using a secondary method.	GS Pro 3.2+
DD000000 0000		
Disable Expansion Pack	Disabled the expansion pack (if present) using a 3rd method.	GS Pro 3.2+
CC000000 0000		
Enable Code	Tells the GameShark where the value "3C0880" is in the RAM. This type of code	GS 1.08+

	does not write a value to the given address. It sets the entry point which the GS will use to start the game. Games which require that code have a specific protection chip which will set the entry point upon booting the N64.	
DEXXXXXX 0000		
Enable Code / 8-Bit Write Once	Tells the GameShark what address is causing malfunction with it, and writes the supplied value to that address. Writes the value YY to the address XXXXXX once on boot. F0\F1 codes write to RAM before starting the game. This way, the codes take effect before the code handler is executed.	GS Pro 3.0+ / XP64
F0XXXXXX 00YY		
Enable Code / 16-Bit Write Once	16-Bit version of the above.	GS Pro 3.0+ / XP64
F1XXXXXX YYYY		
Set Store Location For Active Codes	Sets the location in RAM where active codes are stored. Usually only used on games that utilize the expansion pack.	GS Pro 3.3+
FFXXXXXX 0000		

Enable Code - Xploder64	The same as an F1 enabler on GS Pro.	XP64
2AXXXXXX YYYY		
Enable Code - Xploder64	The exact use of this code type is unknown as of now.	XP64
3CXXXXXX YYYY		
Patch Code	Patch codes, aka Serial Repeaters, are used to make a code string shorter. EG, You have five codes put together to give you "all weapons." Use the patch to shorten it to two codes. XX is the number of addresses to write; YY is the amount (offset) to add to each address; ZZ is the amount to add to each value.	GS Pro 3.3+
5000XXYY 00ZZ TTTTTTTT VVVV		

Playstation Code Types		Compatibility
GameShark		
8-bit Constant Write	Constantly writes the value specified by YY to address XXXXXX. This and its 16-bit counterpart below are the most used code types on PSX. You'll probably be making most of your new codes with them.	GS x.x
30XXXXXX 00YY		

16-bit Constant Write	Constantly writes the value specified by YY to address XXXXXX. This and its 16-bit counterpart below are the most used code types on PSX. You'll probably be making most of your new codes with them.	GS x.x
80XXXXXX 00YY		
8-bit Equal To Activator	When the value for the given address is equal to the supplied value, activate the following code.	GS 2.2+
E0XXXXXX 00YY		
8-bit Different To Activator	When the value for the given address is different to the supplied value, activate the following code.	GS 2.2+
E1XXXXXX 00YY		
8-bit Less Than Activator	When the value for the given address is less than the supplied value, activate the following code.	GS 2.2+
E2XXXXXX 00YY		
8-bit Greater Than Activator	When the value for the given address is greater than the supplied value, activate the following code.	GS 2.2+
E3XXXXXX 00YY		
16-bit Equal To Activator	When the value for the given address is equal to the supplied value, activate the following code.	GS x.x
D0XXXXXX 00YY		

16-bit Different To Activator	When the value for the given address is different to the supplied value, activate the following code.	GS 2.2+
D1XXXXXX 00YY		
16-bit Less Than Activator	When the value for the given address is less than the supplied value, activate the following code.	GS 2.2+
D2XXXXXX 00YY		
16-bit Greater Than Activator	When the value for the given address is greater than the supplied value, activate the following code.	GS 2.2+
D3XXXXXX 00YY		
16-bit Universal Activator	Same as D0 except not RAM dependent. More easily used as a button activator.	GS 2.41+
D4000000 YYYY		
16-bit All-code Button Activator	When buttons pressed equal YYYY then activate all codes.	GS 2.41+
D5000000 YYYY		
16-bit Universal De-Activator	When buttons pressed equal YYYY then de-activate all codes.	GS 2.41+
D6000000 YYYY		
16-bit Increment Value	Add value(16-bit) code. Use with D/E activators. Example - (adds value "1007" to address "001221" when address "110012" equals value "5") D0110012 0005 10001221 1007	GS 2.2+
10XXXXXX 00YY		

16-bit Decrement Value	Subtract value(16-bit) code. Use only with D/E activators. Example - (subtracts value "102" from address "001221" when address "110012" equals value "6") D0110012 0006 11001221 0102	GS 2.2+
11XXXXXX 00YY		
8-bit Increment Value	Add value(8-bit) code. Use with D/E activators. Example - (adds value "7" to address "001221" when address "110012" equals value "5") D0110012 0005 20001221 0007	GS 2.2+
20XXXXXX 00YY		
8-bit Decrement Value	Subtract value(8-bit) code. Use only with D/E activators. Example - (subtracts value "2" from address "001221" when address "110012" equals value "6") D0110012 0006 21001221 0002	GS 2.2+
21XXXXXX 00YY		
Patch Code	Patch codes, aka Serial Repeaters, are used to make a code string shorter. EG, You have five codes put together to give you "all weapons." Use the patch to shorten it to two codes. XX is the number of addresses to write; YY is the amount (offset) to add to each address;	GS 2.41+
5000XXYY 00ZZ TTTTTTTT VVVV		

	ZZ is the amount to add to each value.	
Activate All Codes	Works like the D0/E0 code type, but affects ALL codes. Use as an (M) Must Be On if the game won't load with codes turned on.	GS 2.2+
C0XXXXXX YYYY		
Activate All Codes On Delay	Works like the D0/E0 code type, but affects ALL codes. This is like a timer. A value of around 4000 or 5000 will usually give you a good 20-30 second delay before codes are activated. Use as an (M) Must Be On if the game won't load with codes turned on.	GS 2.41+
C1000000 YYYY		
Copy Bytes	Copy's YYYY bytes from location XXXXXX to location ZZZZZZ. Example use would be: C2040450 0008 80040680 0000 That would copy 8 bytes from 40450 to 40680.	GS 2.41+
C2XXXXXX YYYY 80ZZZZZZ 0000		
Xplorer/Xploder -- see explorer FAQ for more info		
8-bit Constant Write	Writes value YY to address XXXXXX.	Xplorer
30XX XXXX 00YY		
16-bit Constant Write	Writes value YYYY to address XXXXXX.	Xplorer
80XX XXXX YYYY		

Slow Motion Code 4000 0000 000X	Delays CPU by X per cycle. Best used with activator.	Xplorer
Text Replace Code 50XX XXXX YYYY ZZZZ ZZZZ ZZZZ ZZZZ ZZ..	Writes any number of bytes ZZ to address XXXXXX. YYYY is the number of bytes to write.	Xplorer
Do on Event Code 6000 0000 YYYY AAAA AAAA CCCC FFFF FFFF XXXX XXXX XXXX XXXX	CPU breaks at address AAAAAAAA; YYYY is number of bytes used (XX's); FFFFFFFF is the break point mask; CCCC is the type of break point, which can be E180 (instruction gotton by CPU but not yet implemented), EE80 (data to be read or written), E680 (data to be read), EA80 (data to be wrtten) or EF80 (instruction).	Xplorer
Do-if-True Code 70XX XXXX YYYY	If address XXXXXX is equal to value YYYY execute following code.	Xplorer
Do-if-False Code 90XX XXXX YYYY	If address XXXXXX is not equal to value YYYY execute following code.	Xplorer
Patch Code B0nn AAAA BBBB 10XX XXXX YYYY	nn is the number of repetitions (plus one); AAAA is the size of the address step; BBBB is the increase in the data	Xplorer

	value per step; XXXXXX is the initial address; YYYY is the inital value.	
Do-if-True Code (c-code)	Same as 7-code, but only functions from 0010 0000 to 01FF FFFF.	Xplorer
C0XX XXXX YYYY		
Do-if-True Code (d-code)	Same as 7-code. but only functions from 0000 0000 to 000F FFFF.	Xplorer
D0XX XXXX YYYY		
Auto-Activating Code	Automatically activates other selected codes if address XXXXXX is equal to YYYY.	Xplorer
F0XX XXXX YYYY		
32-Bit Constant Write	32-bit constant write to XXXXXX address (0000YYYY)	Xplorer
00XX XXXX YYYY		

Playstation 2 Code Types - Courtesy hellion (hellion00.thegfcc.com)	
Note that all the code types below are in RAW form. RAW codes must be encrypted to work on the Gameshark for Playstation 2.	
8-bit Constant Write	This command will constantly write the value specified by dd to the address specified by aaaaaaa.
0aaaaaaa 000000dd	
16-bit Constant Write	This command will constantly

	write the value
1aaaaaaa 0000dddd	specified by dddd to the address specified by aaaaaaa.
32-bit Constant Write	This command will constantly write the value specified by dddddddd to the address specified by aaaaaaa.
2aaaaaaa dddddddd	
Increment/Decrement Commands	
8-bit Increment	This command adds the value specified by nn to the value stored at the address aaaaaaaa.
301000nn aaaaaaaa	
8-bit Decrement	This command subtracts the value specified by nn to the value stored at the address aaaaaaaa.
302000nn aaaaaaaa	
16-bit Increment	This command adds the value specified by nnnn to the value stored at the address aaaaaaaa.
3030nnnn aaaaaaaa	
16-bit Decrement	This command subtracts the value specified by nnnn to the value stored at the address aaaaaaaa.
3040nnnn aaaaaaaa	
32-bit Increment	This command adds the value

	specified by nnnnnnnn to the value stored at the address aaaaaaaa.
30500000 aaaaaaaa nnnnnnnn 00000000	
32-bit Decrement	This command subtracts the value specified by nnnnnnnn to the value stored at the address aaaaaaaa.
30600000 aaaaaaaa nnnnnnnn 00000000	
Test Commands	
16-bit Equal	Only when the value at the address specified by aaaaaaa is equal to the value specified by dddd will the next line of code be executed.
Daaaaaaa 0000dddd	
16-bit Not Equal	Only when the value at the address specified by aaaaaaa is not equal to the value specified by dddd will the next line of code be executed.
Daaaaaaa 0010dddd	
16-bit Less Than	Only when the value at the address specified by aaaaaaa is less than the value specified by dddd will the next line of code be executed.
Daaaaaaa 0020dddd	
16-bit Greater Than	Only when the value at the address specified by aaaaaaa is greater than the value specified by dddd will the next line of code be executed.
Daaaaaaa 0030dddd	

16-bit Equal : Multiple Skip	Only when the value at the address specified by aaaaaaa is equal to the value specified by dddd will the next nnn lines of code be executed. Otherwise, they will be skipped.
Ennndddd 0aaaaaaa	
16-bit Not Equal : Multiple Skip	Only when the value at the address specified by aaaaaaa is not equal to the value specified by dddd will the next nnn lines of code be executed. Otherwise, they will be skipped.
Ennndddd 1aaaaaaa	
16-bit Less Than : Multiple Skip	Only when the value at the address specified by aaaaaaa is less than the value specified by dddd will the next nnn lines of code be executed. Otherwise, they will be skipped.
Ennndddd 2aaaaaaa	
16-bit Greater Than : Multiple Skip	Only when the value at the address specified by aaaaaaa is greater than the value specified by dddd will the next nnn lines of code be executed. Otherwise, they will be skipped.
Ennndddd 3aaaaaaa	
Miscellaneous Commands	
Copy Bytes (GS2 v2.0 or higher)	a = Address to copy from b = Address to copy to n = Number of bytes to copy
5aaaaaaa nnnnnnnn bbbbbbbb 00000000	

32-bit Multi-Address Write 4aaaaaaa xxxxyyyy dddddddd 00000000	Starting with the address specified by aaaaaaa, this code will write to xxxx addresses. The next address is determined by incrementing the current address by (yyyy * 4). The value specified by dddddddd is written to each calculated address. Also known as a "Patch Code."
Untested Commands	
3000nnnn dddddddd aaaaaaaax(n-1)	32-bit Multiple Address Write?
8aaaaaaa bbbbbbbb cccccccc 00000000	Master Command
Aaaaaaaa dddddddd	32-bit Write Once?
B0000000 nnnnnnnn	Timer Command
Caaaaaaa dddddddd	32-bit Equal?
Faaaaaaa bbbbbbbb	Master Command
DEADFACE xxxxxxxx	"DEADFACE" Master Command - changes encryption seeds

Sega Dreamcast Code Types	
The following are what decrypted or raw code types look like. Xploder and Codebreaker use this format. Gameshark uses an encrypted format for the first line of the code (the address) while all use the second line as is (offset or quantifier). Dreamcast has 32 bit codes. These codes will require 8 digits for the offset and 8 digits for the quantifier, 2^32= FFFFFFFF in hex. Example, XXYYYYYY ZZZZZZZZ. ***Note:*** *It is not unusual for manufacturers of cheating devices to encrypt their codes. Fire International (Blaze USA) has encrypted codes in both its N64 and Game Boy Xploder/Xplorer, while Interact has employed encryption in its DC Shark, GameBoy Advance Shark, and Playstation 2 Shark. If the code begins with a '0', then it is in decrypted format. As in any encryption there always is a crack to defeat it. Codebreaker and Xploder both have the built in ability to accept DC gameshark codes in decipher them. It is left to the reader to explore the decryption further.*	
Code Value	**Description**

00xxxxxx 000000vv	Write 8bit (byte) value "vv" to memory address 8cxxxxxx. That is, 8 bit constant write.
01xxxxxx 0000vvvv	Write 16bit (2byte) value "vvvv" to memory address 8cxxxxxx. That is, 16 bit constant write.
02xxxxxx vvvvvvvv	Write 32bit (4byte) value "vvvvvvvv" to memory address 8cxxxxxx. That is, 32 bit constant write.
0300nnnn aaaaaaaa	Group write code. nn specifies how many 32 bit values follow. aaaaaaaa is the addrsss to write to. The values following this code are written to address aaaaaaaa. E.g: 03000004 8c012000 11111111 22222222 33333333 44444444 The effect is as follows: With a count of 00000004 codes, to address 8c012000: 8c012000 = 11111111 8c012004 = 22222222 8c012008 = 33333333 8c01200c = 44444444
030100vv aaaaaaaa	Increment code. Add the 8bit value vv to the value at address aaaaaaaa
030200vv aaaaaaaa	Decrement code. Subtract the 8bit value vv from the value at address aaaaaaaa
0303vvvv aaaaaaaa	Increment code. Add the 16bit value vvvv to the value at address aaaaaaaa
0304vvvv aaaaaaaa	Decrement code. Subtract the 16bit value vvvv from the value at address aaaaaaaa

03050000 aaaaaaaa vvvvvvvv	Increment code. Add the 32bit value vvvvvvvv to the value at address aaaaaaaa Note that this code is 3 lines long and so will require an 0exxxxxx condition (not a 0dxxxxxx) if you're using it with a condition code.
03060000 aaaaaaaa vvvvvvvv	Decrement code. Subtract the 32bit value vvvvvvvv from the value at address aaaaaaaa Note that this code is 3 lines long and so will require an 0exxxxxx condition (not a 0dxxxxxx) if you're using it with a condition code.
04xxxxxx rrrrssss vvvvvvvv	Repeat/Filler code. Writes to address 8Cxxxxxx. Writes the 32bit value vvvvvvvv. Repeats this rrrr time, each time increasing the address by ssss (actually ssss x 4). That is, 32-Bit Constant Serial Write E.g: 04007a30 00030001 12345678 Effect: 8c007a30 = 12345678 8c007a34 = 12345678 8c007a38 = 12345678
05xxxxxx dddddddd nnnnnnnn	Copy bytes code. Copy nnnnnnnn bytes from the address 8cxxxxxx to the address dddddddd. That is, constant copy bytes
071000XX	Change Decryption Type
0b0xxxxx	Delay putting on codes for xxxxx cycles. Default 1000 (0x3e7)
0cxxxxxx vvvvvvvv	If the value at address 8Cxxxxxx is equal to vvvvvvvv, execute ALL codes; otherwise no codes are executed. Useful for waiting until game has loaded.

0dxxxxxx 0000vvvv	If the value at address 8Cxxxxxx is equal to vvvv, execute the following code. Can be used with code types 00, 01 and 02 only. To use this type of control with other codes use an 0e code.
0dxxxxxx 0001vvvv	If the value at address 8Cxxxxxx is different to vvvv, execute the following code. Can be used with code types 00, 01 and 02 only. To use this type of control with other codes use an 0e code.
0dxxxxxx 0002vvvv	If the value at address 8Cxxxxxx is less than vvvv (unsigned), execute the following code. Can be used with code types 00, 01 and 02 only. To use this type of control with other codes use an 0e code.
0dxxxxxx 0003vvvv	If the value at address 8Cxxxxxx is greater than vvvv (unsigned), execute the following code. Can be used with code types 00, 01 and 02 only. To use this type of control with other codes use an 0e code.
0ennvvvv 00aaaaaa	If the value at address 8caaaaaa is equal to vvvv, execute the following nnnn lines of codes. E.g: 0e04abcd 00012000 02300040 ffffffff 02300050 eeeeeeee if address 8c012000==abcd, execute the 04 lines of codes following. The 4 lines of codes being two "02xxxxxx" codes "02300040=ffffffff" and "02300050=eeeeeeee".
0ennvvvv 01aaaaaa	If the value at address 8caaaaaa is different to vvvv, execute the following nnnn lines of codes.
0ennvvvv 02aaaaaa	If the value at address 8caaaaaa is less than vvvv (unsigned), execute the following nnnn lines of codes.

0ennvvvv 03aaaaaa	If the value at address 8caaaaaa is greater than vvvv (unsigned), execute the following nnnn lines of codes.
0F-XXXXXX 0000YYYY	16-Bit Write Once Immediately. (Activator code)

Sega Saturn Code Types - Courtesy Leo/AGSCC and CodeMaster	
16-bit Constant Write	Just what it implies. Continuously writes YYYY value to XXXXXXX address.
1XXXXXXX YYYY	
8-bit Constant Write	Continuous write of YY value to address XXXXXXX.
3XXXXXXX 00YY	
16-bit Write Once	Writes YYYY value to XXXXXXX address once on boot up. Same as F0/F1 on N64
0XXXXXXX YYYY	
16-bit Equal To Activator	Activates the code on the line directly beneath it ONLY when XXXXXXX address is YYYY value.
DXXXXXXX YYYY	
16-Bit Enable Code	Enable Code
FXXXXXXX YYYY	

Gameboy/Gameboy Color Code Types

8-bit Constant Write	The most common GS code prefix for Gameboy is "01". This means the code resides in the first bank of the address line. Codebreaker users will find that 00 and 01 are equally used code types. The "00" simply means the code resides in the zero bank and "01" as above, the first bank. There are no known other code types for gameboy as there are for N64, Playstation and Dreamcast. Z is the data bank; XXXX is the address; YY is the value.
0ZYYXXXX	

Gameboy Advance Gameshark V1/V2 Code Types - by Parasyte (Additions by DGenerateKane)	
Note that all the code types below are in RAW form. RAW codes must be encrypted to work on the Gameshark for Gameboy Advance.	
8-bit Constant Write	Continuously writes the value xx to the RAM address aaaaaaa.
0aaaaaaaa 000000xx	
16-bit Constant Write	Continuously writes the 16-Bit value xxxx to the RAM address aaaaaaa. Address must be aligned to 2 (must end with one of the following digits - 0,2,4,6,8,A,C,E).
1aaaaaaaa 0000xxxx	
32-bit Constant Write	Continuously writes the 32-Bit value xxxxxxxx to the RAM address aaaaaaa. Address must be aligned to 4 (must end with one of the following digits - 0,4,8,C).
2aaaaaaaa xxxxxxxx	
32-bit Group Write	Writes data to the following "count" (cccc) addresses. (xxxxxxxx value is also concidered an address, not really a problem, just a very stupid bug -- thanks

	Datel!!) Many addresses can follow. Example: 30000004 01010101 03001FF0 03001FF4 03001FF8 00000000 (write 01010101 to 3 addresses - 01010101, 03001FF0, 03001FF4, and 03001FF8. '00000000' is used for padding, to ensure the last code encrypts correctly)
3000cccc xxxxxxxx aaaaaaaa	
16-bit ROM Patch	This type allows GSA to intercept ROM reads and returns the value xxxx. The address is shifted to the right by 1 (divided by 2). You can either manually shift the address left by 1, or multiply by 2 to get the real address. GSAcrypt (Win32 version) has an option to automatically shift the address for you. **Note:** V1\V2 hardware can only have up to 1 user-defined rom patch max. V3 can have up to 4. some enable code types can shorten the amount of user-defined rom patches available.
6aaaaaaa 0000xxxx	
16-bit ROM Patch	Similar to first ROM patch code, except patch is enabled before the game starts, instead of waiting for the code handler to enable the patch. (address >> 1)
6aaaaaaa 1000xxxx	
16-bit ROM Patch	16-bit ROM Patch ? (address >> 1)
6aaaaaaa 2000xxxx	
8-bit GS Button Code	8-Bit RAM write only when

	the GS Button is pressed.
8a1aaaaa 000000xx	
16-bit GS Button Code	16-Bit RAM write only when the GS Button is pressed.
8a2aaaaa 000000xx	
Slowdown On GS Button	Slow down on GS Button. This type will put the GBA into a loop for "xxxx" number of times, each time the code handler is run. This slows the game down.
80F00000 0000xxxx	
16-Bit 'If Equal To' Activator	Activate the code on the next line ONLY when the value of address 'aaaaaaa' is Equal To xxxx.
Daaaaaaa 0000xxxx	
16-Bit 'If Equal To' Activator (Multi-Line)	16-Bit activate the multi lines if-true. If the value at address is equal to xxxx, execute following 'zz' lines.
E0zzxxxx aaaaaaaa	
Hook Routine (For Enablers)	Used to insert the GS code handler routine where it will be executed at least 20 times per second. Without this code, GSA can not write to RAM. xxxx: 0001 - Executes code handler without backing up the $lr register. Must turn GSA off before loading game. 0002 - Executes code handler and backs up the $lr register. Must turn GSA off before loading game.
Faaaaaaa 0000xxxx	

	0003 - Replaces a 32-bit pointer used for long-branches. Must turn GSA off before loading game. 0101 - Executes code handler without backing up the $lr register. 0102 - Executes code handler and backs up the $lr register. 0103 - Replaces a 32-bit pointer used for long-branches.
ID Code (For Enablers)	Used by GSA only for auto-detecting the inserted game.
xxxxxxxx 001DC0DE	
DEADFACE - Change Encryption Seeds	"Deadface" is used to change the encryption seeds. It's original intent was probably to re- encrypt codes if someone figured out the normal encryption. (Very similiar to the CBA's '9' code type.)
DEADFACE 0000xxxx	

Gameboy Advance Codebreaker Code Types - by Parasyte (Additions by DGenerateKane)	
Note that all the code types below are in RAW form. RAW codes must be encrypted to work on the Codebreaker for Gameboy Advance.	
Master Code #1	xxxx is the CRC value (the "Game ID" converted to hex) Flags ("yyyy"): 0008 - CRC Exists (CRC is used to autodetect the inserted game) 0002 - Disable Interupts
0000xxxx yyyy	
Master Code #2	'y' is the CBA Code Handler Store Address (0-7) [address = ((d << 0x16) + 0x08000100)]

	1000 - 32-bit Long-Branch Type (Thumb) 2000 - 32-bit Long-Branch Type (ARM) 3000 - 8-bit(?) Long-Branch Type (Thumb) 4000 - 8-bit(?) Long-Branch Type (ARM) 0020 - Unknown (Odd Effect)
1aaaaaaa xxxy	
8-Bit Constant RAM Write	Continuosly writes the 8-Bit value specified by 'yy' to address aaaaaaa.
3aaaaaaa 00yy	
Slide Code	This is one of those two-line codes. The "yyyy" set is the data to store at the address (aaaaaaa), with xxxxxxxx being the number of addresses to store to, and iiii being the value to increment the addresses by. The codetype is usually use to fill memory with a certain value.
4aaaaaaa yyyy xxxxxxxx iiii	
16-Bit Logical AND	Performs the AND function on the address provided with the value provided. I'm not going to explain what AND does, so if you'd like to know I suggest you see the instruction manual for a graphing calculator. This is another advanced code type you'll probably never need to use.
6aaaaaaa yyyy	
16-Bit 'If Equal To' Activator	If the value at the specified RAM address (aaaaaaa) is equal to yyyy value, active the code on the next line.
7aaaaaaa yyyy	
16-Bit Constant RAM Write	Continuosly writes yyyy values to the specified RAM address (aaaaaaa).
8aaaaaaa yyyy	
Change Encryption Seeds (When 1st Code Only!)	Works like the DEADFACE on GSA. Changes the encryption seeds used for

	the rest of the codes.
9yyyyyyy yyyy	
16-Bit 'If Not Equal' Activator	Basicly the opposite of an 'If Equal To' Activator. Activates the code on the next line if address xxxxxxx is NOT equal to yyyy
Axxxxxxx yyyy	
16-Bit Conditional RAM Write	No Description available at this time.
D00000xx yyyy	

AR V3 Codes Types	
About the Code Types Numbers	
Let's take for exemple : Type E3 3.0.3.1.x : 00XXXXXX : (00000130 -> C7000130) ZZZZZZZZ : Write the Word ZZZZZZZZ to the address $4XXXXXX 3.0.3.1.x : 1st number = 3 = Data size (0 to 3) 2nd number = 0 = Code Type (0 to 7) 3rd number = 3 = Cude subtype (0 to 3) 4th number = 1 = Special bit (0 to 1) 5th number : x = Unused bit (0 to 3) 3.0.3.1.0 gives these (bit speaking) = 11.000.11.1.00 reverse it : 00.1.11.000.11 = 0011100011 = E3 = The code type. I choose to take this numbering to make it that the Ram 8bits write (Type 0), Ram 16bits write Type 1 andRam 32bits write (Type 2) have the same type number than for AR/GS V1/2. Moreover, If I didn't "reverse" the numbers, we've gotten almost only even code type number, which, IMHO, sounds really strange...	
1) Normal RAM Write Codes	
Type 00 0.0.0.x.x	(02024EA4 -> 00224EA4) Fill area (XXXXXXXX) to (XXXXXXXX+YYYYYY) with Byte ZZ.
XXXXXXXX YYYYYYZZ	
Type 01 1.0.0.x.x	(02024EA4 -> 02224EA4) Fill area (XXXXXXXX) to (XXXXXXXX+YYYY*2) with Halfword ZZZZ.
XXXXXXXX YYYYZZZZ	

Type 02 2.0.0.x.x	(02024EA4 -> 04224EA4) Write the Word ZZZZZZZZ to address XXXXXXXX.
XXXXXXXX ZZZZZZZZ	
2) Pointer RAM Write Codes	
Type 20 0.0.1.x.x	(02024EA4 -> 40224EA4) Writes Byte ZZ to ([the address kept in XXXXXXXX]+[YYYYYY]).
XXXXXXXX YYYYYYZZ	
Type 21 1.0.1.x.x	(02024EA4 -> 4224EA4) Writes Halfword ZZZZ ([the address kept in XXXXXXXX]+[YYYY*2]).
XXXXXXXX YYYYZZZZ	
Type 22 2.0.1.x.x	(02024EA4 -> 4424EA4) Writes the Word ZZZZZZZZ to [the address kept in XXXXXXXX].
XXXXXXXX ZZZZZZZZ	
3) Add Codes	
Type 40 0.0.2.x.x	(02024EA4 -> 80224EA4) Add the Byte ZZ to the Byte stored in XXXXXXXX.
XXXXXXXX 000000ZZ	
Type 41 1.0.2.x.x	(02024EA4 -> 82224EA4) Add the Halfword ZZZZ to the Halfword stored in XXXXXXXX.
XXXXXXXX 0000ZZZZ	
Type 42 2.0.2.x.x	(02024EA4 -> 84224EA4) Add the Word ZZZZ to the Halfword stored in XXXXXXXX.
XXXXXXXX ZZZZZZZZ	
4) Write to $4000000 (IO Registers!)	
Type 63 3.0.3.0.x	(00000130 -> C6000130) Write the Halfword ZZZZ to the address $4XXXXXX
00XXXXXX 0000ZZZZ	
Type E3 3.0.3.1.x	(00000130 -> C7000130) Write the Word ZZZZZZZZ to the address $4XXXXXX

00XXXXXX ZZZZZZZZ	
5) If Equal Code (= Joker Code)	
Type 04 0.1.0.x.x	(02024EA4 -> 08224EA4) If Byte at XXXXXXXX = ZZ then execute next code.
XXXXXXXX 000000ZZ	
Type 24 0.1.1.x.x	(02024EA4 -> 48224EA4) If Byte at XXXXXXXX = ZZ then execute next 2 codes.
XXXXXXXX 000000ZZ	
Type 44 0.1.2.x.x	(02024EA4 -> 88224EA4) If Byte at XXXXXXXX = ZZ execute all the codes below this one in the same row (else execute none of the codes below).
XXXXXXXX 000000ZZ	
Type 64 0.1.3.x.x	(02024EA4 -> C8224EA4) While Byte at XXXXXXXX <> ZZ turn off all codes.
XXXXXXXX 000000ZZ	
Type 05 1.1.0.x.x	(02024EA4 -> 0A224EA4) If Halfword at XXXXXXXX = ZZZZ then execute next code.
XXXXXXXX 0000ZZZZ	
Type 05 1.1.0.x.x	(02024EA4 -> 0A224EA4) If Halfword at XXXXXXXX = ZZZZ then execute next code.
XXXXXXXX 0000ZZZZ	
Type 25 1.1.1.x.x	(02024EA4 -> 4A224EA4) If Halfword at XXXXXXXX = ZZZZ then execute next 2 codes.
XXXXXXXX 0000ZZZZ	
Type 45 1.1.2.x.x	(02024EA4 -> 8A224EA4) If Halfword at XXXXXXXX = ZZZZ execute all the codes below this one in the same row (else execute none of the codes below).
XXXXXXXX 0000ZZZZ	
Type 65 1.1.3.x.x	(02024EA4 -> CA224EA4) While Halfword at XXXXXXXX <> ZZZZ turn off all

XXXXXXXX 0000ZZZZ	codes.
Type 06 2.1.0.x.x	(02024EA4 -> 0C224EA4) If Word at XXXXXXXX = ZZZZZZZZ then execute next code.
XXXXXXXX ZZZZZZZZ	
Type 26 2.1.1.x.x	(02024EA4 -> 4C224EA4) If Word at XXXXXXXX = ZZZZZZZZ then execute next 2 codes.
XXXXXXXX ZZZZZZZZ	
Type 46 2.1.2.x.x	(02024EA4 -> 8C224EA4) If Word at XXXXXXXX = ZZZZZZZZ execute all the codes below this one in the same row (else execute none of the codes below).
XXXXXXXX ZZZZZZZZ	
Type 66 2.1.3.x.x	(02024EA4 -> CC224EA4) While Word at XXXXXXXX <> ZZZZZZZZ turn off all codes.
XXXXXXXX ZZZZZZZZ	
6) If Different Code	
Type 08 0.2.0.x.x	(02024EA4 -> 10224EA4) If Byte at XXXXXXXX <> ZZ then execute next code.
XXXXXXXX 000000ZZ	
Type 28 0.2.1.x.x	(02024EA4 -> 50224EA4) If Byte at XXXXXXXX <> ZZ then execute next 2 codes.
XXXXXXXX 000000ZZ	
Type 48 0.2.2.x.x	(02024EA4 -> 90224EA4) If Byte at XXXXXXXX <> ZZ execute all the codes below this one in the same row (else execute none of the codes below).
XXXXXXXX 000000ZZ	
Type 68 0.2.3.x.x	(02024EA4 -> D0224EA4) While Byte at XXXXXXXX = ZZ turn off all codes.
XXXXXXXX 000000ZZ	
Type 09 1.2.0.x.x	(02024EA4 -> 12224EA4) If Halfword at XXXXXXXX <> ZZZZ then execute

XXXXXXXX 0000ZZZZ	next code.
Type 29 1.2.1.x.x	(02024EA4 -> 52224EA4) If Halfword at XXXXXXXX <> ZZZZ then execute next 2 codes.
XXXXXXXX 0000ZZZZ	
Type 49 1.2.2.x.x	(02024EA4 -> 92224EA4) If Halfword at XXXXXXXX <> ZZZZ disable all the codes below this one.
XXXXXXXX 0000ZZZZ	
Type 69 1.2.3.x.x	(02024EA4 -> D2224EA4) While Halfword at XXXXXXXX = ZZZZ turn off all codes.
XXXXXXXX 0000ZZZZ	
Type 0A 2.2.0.x.x	(02024EA4 -> 14224EA4) If Word at XXXXXXXX <> ZZZZZZZZ then execute next code.
XXXXXXXX ZZZZZZZZ	
Type 2A 2.2.1.x.x	(02024EA4 -> 54224EA4) If Word at XXXXXXXX <> ZZZZZZZZ then execute next 2 codes.
XXXXXXXX ZZZZZZZZ	
Type 4A 2.2.2.x.x	(02024EA4 -> 94224EA4) If Word at XXXXXXXX <> ZZZZZZZZ disable all the codes below this one.
XXXXXXXX ZZZZZZZZ	
Type 6A 2.2.3.x.x	(02024EA4 -> D4224EA4) While Word at XXXXXXXX = ZZZZZZZZ turn off all codes.
XXXXXXXX ZZZZZZZZ	
7) [If Byte at address XXXXXXXX is lower than ZZ] (signed) Code	
Signed means : For bytes : values go from -128 to +127. For Halfword : values go from -32768/+32767. For Words : values go from -2147483648 to 2147483647. For exemple, for the Byte comparison, 7F (127) will be > to FF (-1).	
Type 0C 0.3.0.x.x	(02024EA4 -> 18224EA4 or 28224EA4) If ZZ > Byte at XXXXXXXX then execute next

XXXXXXXX 000000ZZ	code.
Type 2C 0.3.1.x.x	(02024EA4 -> 58224EA4 or 68224EA4) If ZZ > Byte at XXXXXXXX then execute next 2 codes.
XXXXXXXX 000000ZZ	
Type 4C 0.3.2.x.x	(02024EA4 -> 98224EA4 or A8224EA4) If ZZ > Byte at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
XXXXXXXX 000000ZZ	
Type 6C 0.3.3.x.x	(02024EA4 -> D8224EA4 or E8224EA4) While ZZ <= Byte at XXXXXXXX turn off all codes.
XXXXXXXX 000000ZZ	
Type 0D 1.3.0.x.x	(02024EA4 -> 1A224EA4 or 2A224EA4) If ZZZZ > Halfword at XXXXXXXX then execute next line.
XXXXXXXX 0000ZZZZ	
Type 2D 1.3.1.x.x	(02024EA4 -> 5A224EA4) If ZZZZ > Halfword at XXXXXXXX then execute next 2 lines.
XXXXXXXX 0000ZZZZ	
Type 4D 1.3.2.x.x	(02024EA4 -> 9A224EA4) If ZZZZ > Halfword at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
XXXXXXXX 0000ZZZZ	
Type 6D 1.3.3.x.x	(02024EA4 -> DA224EA4) While ZZZZ <= Halfword at XXXXXXXX turn off all codes.
XXXXXXXX 0000ZZZZ	
Type 0E 2.3.0.x.x	(02024EA4 -> 1C224EA4) If ZZZZZZZZ > Word at XXXXXXXX then execute next line.
XXXXXXXX ZZZZZZZZ	
Type 2E 2.3.1.x.x	(02024EA4 -> 5C224EA4) If ZZZZZZZZ > Word at XXXXXXXX then execute next 2 lines.
XXXXXXXX ZZZZZZZZ	
Type 4E 2.3.2.x.x	(02024EA4 -> 9C224EA4) If ZZZZZZZZ > HWord at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
XXXXXXXX ZZZZZZZZ	

Type 6E 2.3.3.x.x	(02024EA4 -> DC224EA4) While ZZZZZZZZ <= Word at XXXXXXXX turn off all codes.
XXXXXXXX ZZZZZZZZ	
8) [If Byte at address XXXXXXXX is higher than ZZ] (signed) Code	
Signed means : For bytes : values go from -128 to +127. For Halfword : values go from -32768/+32767. For Words : values go from -2147483648 to 2147483647. For exemple, for the Byte comparison, 7F (127) will be > to FF (-1).	
Type 10 0.4.0.x.x , 0.6.0.x.x	(02024EA4 -> 20224EA4 or 30224EA4) If ZZ < Byte at XXXXXXXX then execute next code.
XXXXXXXX 000000ZZ	
Type 30 0.4.1.x.x	(02024EA4 -> 60224EA4 or 70224EA4) If ZZ < Byte at XXXXXXXX then execute next 2 codes.
XXXXXXXX 000000ZZ	
Type 50 0.4.2.x.x , 0.6.2.x.x	(02024EA4 -> A0224EA4 or B0224EA4) If ZZ < Byte at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
XXXXXXXX 000000ZZ	
Type 70 0.4.3.x.x	(02024EA4 -> E0224EA4 or F0224EA4) While ZZ => Byte at XXXXXXXX turn off all codes.
XXXXXXXX 000000ZZ	
Type 11 1.4.0.x.x, 1.6.0.x.x	(02024EA4 -> 22224EA4 or 32224EA4) If ZZZZ < Halfword at XXXXXXXX then execute next line.
XXXXXXXX 0000ZZZZ	
Type 31 1.4.1.x.x	(02024EA4 -> 62224EA4) If ZZZZ < Halfword at XXXXXXXX then execute next 2 lines.
XXXXXXXX 0000ZZZZ	
Type 51 1.4.2.x.x, 1.6.2.x.x	(02024EA4 -> A2224EA4 or B2224EA4) If ZZZZ < Halfword at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
XXXXXXXX 0000ZZZZ	
Type 71 1.4.3.x.x	(02024EA4 -> E2224EA4) While ZZZZ => Halfword at XXXXXXXX turn off all codes.
XXXXXXXX 0000ZZZZ	
Type 12 2.4.0.x.x, 2.6.0.x.x	(02024EA4 -> 24224EA4 or 34224EA4) If ZZZZ < Halfword at XXXXXXXX then execute next line.

XXXXXXXX 0000ZZZZ	
Type 32 2.4.1.x.x	(02024EA4 -> 64224EA4) If ZZZZ < Halfword at XXXXXXXX then execute next 2 lines.
XXXXXXXX 0000ZZZZ	
Type 52 2.4.2.x.x,2.6.2.x.x	(02024EA4 -> A4224EA4 or B4224EA4) If ZZZZ < Halfword at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
XXXXXXXX 0000ZZZZ	
Type 72 2.4.3.x.x	(02024EA4 -> E4224EA4) While ZZZZ => Halfword at XXXXXXXX turn off all codes.
XXXXXXXX 0000ZZZZ	
9) [If Value at adress XXXXXXXX is lower than...] (unsigned) Code	
Unsigned means : For bytes : values go from 0 to +255. For Halfword : values go from 0 to +65535. For Words : values go from 0 to 4294967295. For exemple, for the Byte comparison, 7F (127) will be < to FF (255).	
Type 14 0.5.0.x.x	(02024EA4 -> 28224EA4) If ZZZZZZZZ > Byte at XXXXXXXX then execute next line.
XXXXXXXX ZZZZZZZZ	
Type 34 0.5.1.x.x	(02024EA4 -> 68224EA4) If ZZZZZZZZ > Byte at XXXXXXXX then execute next 2 lines.
XXXXXXXX ZZZZZZZZ	
Type 54 0.5.2.x.x	(02024EA4 -> A8224EA4) If ZZZZZZZZ > Byte at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
XXXXXXXX ZZZZZZZZ	
Type 74 0.5.3.x.x	(02024EA4 -> E8224EA4) While ZZ <= Byte at XXXXXXXX turn off all codes.
XXXXXXXX ZZZZZZZZ	
Type 15 1.5.0.x.x	(02024EA4 -> 2A224EA4) If ZZZZZZZZ > Halfword at XXXXXXXX then execute next line.
XXXXXXXX ZZZZZZZZ	
Type 35 1.5.1.x.x	(02024EA4 -> 6A224EA4) If ZZZZZZZZ > Halfword at XXXXXXXX then execute next 2 lines.
XXXXXXXX ZZZZZZZZ	
Type 55 1.5.2.x.x	(02024EA4 -> AA224EA4) If ZZZZZZZZ > Halfword at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
XXXXXXXX ZZZZZZZZ	
Type 75 1.5.3.x.x	(02024EA4 -> EA224EA4) While ZZZZZZZZ <= Halfword at XXXXXXXX turn off all codes.
XXXXXXXX ZZZZZZZZ	
Type 16 2.5.0.x.x	(02024EA4 -> 2C224EA4) If ZZZZZZZZ > Word at

XXXXXXXX ZZZZZZZZ	XXXXXXXX then execute next line.
Type 36 2.5.1.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> 6C224EA4) If ZZZZZZZZ > Word at XXXXXXXX then execute next 2 lines.
Type 56 2.5.2.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> AC224EA4) If ZZZZZZZZ > Word at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
Type 76 2.5.3.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> EC224EA4) While ZZZZZZZZ <= Word at XXXXXXXX turn off all codes.
10) [If Value at adress XXXXXXXX is higher than...] (unsigned) Code	
Unsigned means For bytes : values go from 0 to +255. For Halfword : values go from 0 to +65535. For Words : values go from 0 to 4294967295. For exemple, for the Byte comparison, 7F (127) will be < to FF (255).	
Type 18 0.6.0.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> 30224EA4) If ZZZZZZZZ < Byte at XXXXXXXX then execute next line..
Type 38 0.6.1.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> 70224EA4) If ZZZZZZZZ < Byte at XXXXXXXX then execute next 2 lines.
Type 58 0.6.2.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> B0224EA4) If ZZZZZZZZ < Byte at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
Type 78 0.6.3.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> F0224EA4) While ZZZZZZZZ => Byte at XXXXXXXX turn off all codes.
Type 19 1.6.0.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> 32224EA4) If ZZZZZZZZ < Halfword at XXXXXXXX then execute next line.
Type 39 1.6.1.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> 72224EA4) If ZZZZZZZZ < Halfword at XXXXXXXX then execute next 2 lines.
Type 59 1.6.2.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> B2224EA4) If ZZZZZZZZ < Halfword at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
Type 79 1.6.3.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> F2224EA4) While ZZZZZZZZ => Halfword at XXXXXXXX turn off all codes.

Type 1A 2.6.0.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> 34224EA4) If ZZZZZZZZ < Halfword at XXXXXXXX then execute next line.
Type 3A 2.6.1.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> 74224EA4) If ZZZZZZZZ < Halfword at XXXXXXXX then execute next 2 lines.
Type 5A 2.6.2.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> B4224EA4) If ZZZZZZZZ < Halfword at XXXXXXXX then execute all following codes in the same row (else execute none of the codes below).
Type 7A 2.6.3.x.x XXXXXXXX ZZZZZZZZ	(02024EA4 -> F4224EA4) While ZZZZZZZZ => Halfword at XXXXXXXX turn off all codes.
11) If AND Code	
Type 1C 0.7.0.x.x XXXXXXXX 000000ZZ	(02024EA4 -> 38224EA4) If ZZ AND Byte at XXXXXXXX <> 0 (= True) then execute next code.
Type 3C 0.7.1.x.x XXXXXXXX 000000ZZ	(02024EA4 -> 78224EA4) If ZZ AND Byte at XXXXXXXX <> 0 (= True) then execute next 2 codes.
Type 5C 0.7.2.x.x XXXXXXXX 000000ZZ	(02024EA4 -> B8224EA4) If ZZ AND Byte at XXXXXXXX <> 0 (= True) then execute all following codes in the same row (else execute none of the codes below).
Type 7C 0.7.3.x.x XXXXXXXX 000000ZZ	(02024EA4 -> F8224EA4) While ZZ AND Byte at XXXXXXXX = 0 (= False) then turn off all codes.
Type 1D 1.7.0.x.x XXXXXXXX 0000ZZZZ	(02024EA4 -> 3A224EA4) If ZZZZ AND Halfword at XXXXXXXX <> 0 (= True) then execute next code.
Type 3D 1.7.1.x.x XXXXXXXX 0000ZZZZ	(02024EA4 -> 7A224EA4) If ZZZZ AND Halfword at XXXXXXXX <> 0 (= True) then execute next 2 codes.
Type 5D 1.7.2.x.x	(02024EA4 -> BA224EA4) If ZZZZ AND Halfword at XXXXXXXX <> 0 (= True)

XXXXXXXX 0000ZZZZ	then execute all following codes in the same row (else execute none of the codes below).
Type 7D 1.7.3.x.x	(02024EA4 -> FA224EA4) While ZZZZ AND Halfword at XXXXXXXX = 0 (= False) then turn off all codes.
XXXXXXXX 0000ZZZZ	
Type 1E 2.7.0.x.x	(02024EA4 -> 3C224EA4) If ZZZZZZZZ AND Word at XXXXXXXX <> 0 (= True) then execute next code.
XXXXXXXX ZZZZZZZZ	
Type 3E 2.7.1.x.x	(02024EA4 -> 7C224EA4) If ZZZZZZZZ AND Word at XXXXXXXX <> 0 (= True) then execute next 2 codes.
XXXXXXXX ZZZZZZZZ	
Type 5E 2.7.2.x.x	(02024EA4 -> BC224EA4) If ZZZZZZZZ AND Word at XXXXXXXX <> 0 (= True) then execute all following codes in the same row (else execute none of the codes below).
XXXXXXXX ZZZZZZZZ	
Type 7E 2.7.3.x.x	(02024EA4 -> FC224EA4) While ZZZZZZZZ AND Word at XXXXXXXX = 0 (= False) then turn off all codes.
XXXXXXXX ZZZZZZZZ	
12) "Always..." Codes	
For the "Always..." codes: -XXXXXXXX can be any authorised address BUT 00000000 (use 02000000 if you don't know what to choose). -ZZZZZZZZ can be anything. -The "y" in the code data must be in the [1-7] range (which means not 0).	
Type 07 3.y.0.x.x	(02024EA4 -> 0E224EA4) Always skip next line.
XXXXXXXX ZZZZZZZZ	
Type 27 3.y.1.x.x	(02024EA4 -> 4E24EA4) Always skip next 2 lines.
XXXXXXXX ZZZZZZZZ	
Type 47 3.y.2.x.x	(02024EA4 -> 8E224EA4) Always Stops executing all the codes below.
XXXXXXXX ZZZZZZZZ	
Type 67 3.y.3.x.x	(02024EA4 -> CE224EA4) Always turn off all codes.
XXXXXXXX ZZZZZZZZ	

13) 1 Line Special Codes (= starting with "00000000")	
Type z00 0.0.0.0.0	End of the code list (even if you put values in the 2nd line).
00000000	
Type z04 x.1.0.x.x	AR Slowdown : loops the AR XX times
0800XX00	
14) 2 Lines Special Codes (= starting with '00000000' and padded (if needed) with "00000000")	
Note: You have to add the 0es manually, after clicking the "create" button.	
Type z08 0.2.0.x.x	(02024EA4 -> 10224EA4) Writes Byte ZZ to address XXXXXXXX when AR button is pushed.
XXXXXXXX 000000ZZ	
Type z09 1.2.0.x.x	(02024EA4 -> 12224EA4) Writes Halfword ZZZZ to address XXXXXXXX.
XXXXXXXX 0000ZZZZ	
Type z0A 2.2.0.x.x	(02024EA4 -> 14224EA4) Writes Word ZZZZZZZZ to address XXXXXXXX.
XXXXXXXX ZZZZZZZZ	
Type z0C 0.3.0.x.x	(02024EA4 -> 18224EA4) Patches ROM address (XXXXXXXX << 1) with Halfword ZZZZ.
XXXXXXXX 0000ZZZZ	
Type z0D 1.3.0.x.x	(02024EA4 -> 1A224EA4) Patches ROM address (XXXXXXXX << 1) with Halfword ZZZZ. Does not work on V1/2 upgraded to V3. Only for a real V3 Hardware?
XXXXXXXX 0000ZZZZ	
Type z0E 2.3.0.x.x	(02024EA4 -> 1C224EA4) Patches ROM address (XXXXXXXX << 1) with Halfword ZZZZ. Does not work on V1/2 upgraded to V3. Only for a real V3 Hardware?
XXXXXXXX 0000ZZZZ	
Type z0F 3.3.0.x.x	(02024EA4 -> 1E224EA4) Patches ROM address (XXXXXXXX << 1) with

XXXXXXXX 0000ZZZZ	Halfword ZZZZ. Does not work on V1/2 upgraded to V3. Only for a real V3 Hardware?
Type z20 x.0.1.x.x	(00000000 -> 40000000) (SP = 0) (means : stops the "then execute all following codes in the same row" and stops the "else execute none of the codes below)".
00000000 00000000	
Type z30 x.4.1.x.x	(00000000 -> 60000000) (If SP <> 2 -> SP = 1) (means : start to execute all codes until end of codes or SP = 0). (bypass the number of codes to executes set by the master code).
00000000 00000000	
Type z40 0.0.2.x.x	(02024EA4 -> 80224EA4) Writes Byte YY at address XXXXXXXX. Then makes YY = YY + Z1, XXXXXXXX = XXXXXXXX + Z3Z3, Z2 = Z2 - 1, and repeats until Z2 < 0.
XXXXXXXX 000000YY Z1Z2Z3Z3	
Type z41 1.0.2.x.x	(02024EA4 -> 82224EA4) Writes Halfword YYYY at address XXXXXXXX. Then makes YYYY = YYYY + Z1, XXXXXXXX = XXXXXXXX + Z3Z3*2,
XXXXXXXX 0000YYYY Z1Z2Z3Z3	
Type z42 2.0.2.x.x	(02024EA4 -> 84224EA4) Writes Word YYYYYYYY at address XXXXXXXX. Then makes YYYYYYYY = YYYYYYYY + Z1, XXXXXXXX = XXXXXXXX + Z3Z3*4, Z2 = Z2 - 1, and repeats until Z2<0.
XXXXXXXX YYYYYYYY Z1Z2Z3Z3	
WARNING: There is a BUG on the REAL AR (v2 upgraded to v3, and maybe on real v3) with the 32Bits Increment Slide code. You HAVE to add a code (best choice is 80000000 00000000 : add 0 to value at address 0) right after it, else the AR will erase the 2 last 8 digits lines of the 32 Bits Inc. Slide code when you enter it !!!	
15) Special Codes	
-Master Code- Type 62 2.0.3.x.x	(address to patch -> address to patch AND $1FFFFFE) Master Code settings.
XXXXXXXX 0000YYYY	
-ID Code- Type 62 2.0.3.x.x	word at address 080000AC Must always be 001DC0DE

XXXXXXXX 001DC0DE	
-DEADFACE-	Must always be DEADFACE New Encryption seed.
DEADFACE 0000XXXX	

Final Notes

```
SP = 0 : normal (execute n codes before giving back the hand to the game).
SP = 1 : execute all codes until SP <>1 (or end of codes).
SP = 2 : don't execute anymore codes.

Each time the GSA starts to execute codes (= each time "it has the hand"), SP = 0.

The 'execute all the codes below this one in the same row' makes SP = 1.
The '(else execute none of the codes below)' makes SP = 2.
The 'turn off all codes' makes an infinite loop (that can't be broken, unless
the condition becomes True).
```

D) Button Activators

What is a Button Activator?

Most, if not all systems have the ability to use Button Activators. A Button Activator is a code that will activate another code when you a button on the controller. These codes use the D0/D1 prefix.
NOTE: Activators *usually* only activate only the code directly beneath them. There are some special cases though. The digits and such for these codes vary by system...

What is a Control Stick Activator?

A Control Stick Activator is the same as a Button Activator, but it is unknown at this time if they can be used in the same way on the console itself (only tested on emulator). Based on N64 testing, the reason for this difference is the range of values that the control stick can have (e.g. pushing the stick all the way left on the emulator makes the activator value 82, but odds are it's slightly different on the console). I know at the very least, these will work on the console to activate codes when the stick is moved any direction if you change, for example, an 'Equal To Activator' to a 'Not Equal To Activator' and use 0 for the value. This could be useful for the THQ wrestling games where the only thing the stick does is cause your wrestler to taunt; you could use the control stick to activate the Always Special or Full Spirit code for your player whenever you taunt. I invite whatever hackers we have left here to experiment with these on the console and post their findings on the messageboards.

What good are they?

Activators are used when you only want to have a certain code active at a certain time, rather than it always being active.

Activators on Nintendo 64

Activators on N64 come in a few varieties: 8-Bit and 16-Bit. This is also the only system to see Control Stick Activators thus far.

- Button Activator 1 (8-Bit) values:

```
00 - No Buttons
01 - D-Pad Right
02 - D-Pad Lefft
04 - D-Pad Down
08 - D-Pad Up
10 - Start
20 - Z
```

```
        40 - B
        80 - A

   You can combine these for multiple buttons. I.E. D-Pad Left and
     B would be 42.
```

- Button Activator 2 (8-Bit) values:

```
        00 - No Buttons
        01 - C-Right
        02 - C-Lefft
        04 - C-Down
        08 - C-Up
        10 - R
        20 - L

  You can combine these for multiple buttons. I.E. C-Left and R would be 12.
```

- Dual (16-Bit) Activator values:

```
       Dual Activators are nothing more than the above 2 Activators together.
They have 4 digits instead of 2. the first 2 digits use Activator 1 values
and the 2nd 2 use Activator 2 values. This allows for more button combos for
activating codes without having to use both Activators seperately.
I.E. L + R + Z would be 2030.
```

- Control Stick Activator 1 (8-Bit) values (Nemu 64 only):

```
00 - Nothing/Centered
45 - 55% Right
7E - 100% Right
BB - 55% Left
82 - 100% Left
```

- Control Stick Activator 2 (8-Bit) values (Nemu 64 only):

```
00 - Nothing/Centered 45 - 55% Up
7E - 100% Up
BB - 55% Down
82 - 100% Down
```

- Control Stick Activator 1 (8-Bit) values (Project64 only):

```
00 - Nothing/Centered
7E - 100% Right
81 - 100% Left
50 - 100% Right on Axispad
B0 - 100% Left on Axispad
```

- Control Stick Activator 2 (8-Bit) values (Project64 only):

```
00 - Nothing/Centered 7E - 100% Up
81 - 100% Down
50 - 100% Up on Axispad
B0 - 100% Down on Axispad
```

- Dual (16-Bit) Control Stick Activator values:

I would hope you can figure this one out. ;-)
(Same as Dual Button Activators)

Gameboy Advance Codebreaker Activator Digits:

```
START         0x0008
SELECT        0x0004
A             0x0001
B             0x0002
UP            0x0040
DOWN          0x0080
LEFT          0x0020
RIGHT         0x0010
Left Trigger  0x0200
Right Trigger 0x0100
```

Playstation Joker Commands

Button Activators on Playstation are known as "Joker Commands"; don't ask me why. You'll see Playstation Jokers in four forms (all are 16-Bit). The reason there are 4 forms is that, apparetly, game makers never agreed on a common way to store values when keeping track of what buttons are being pressed. Digits here were provided by GSCCC

- Normal Joker Command Digits:

```
0000 - No Buttons
0001 - L2 Button
0002 - R2 Button
0004 - L1 Button
0008 - R1 Button
0010 - Triangle Button
0020 - Circle Button
0040 - X Button
0080 - Square Button
0100 - Select Button
0800 - Start Button
1000 - Up Direction
2000 - Right Direction
4000 - Down Direction
8000 - Left Direction
Multi Buttons - Just combine (add) the values. i.e. Up + R1 would
be 1008.
```

- Reverse Joker Command Digits:

```
0000 - No Buttons
0100 - L2 Button
0200 - R2 Button
0400 - L1 Button
0800 - R1 Button
1000 - Triangle Button
2000 - Circle Button
4000 - X Button
8000 - Square Button
0001 - Select Button
0008 - Start Button
0010 - Up Direction
0020 - Right Direction
0040 - Down Direction
0080 - Left Direction
Multi Buttons - Just combine (add) the values. i.e. Up + R1 would
be 0810.
```

- Max Normal Joker Command Digits:

FFFF - No Buttons
FFFE - L2 Button
FFFD - R2 Button
FFFB - L1 Button
FFF7 - R1 Button
FFEF - Triangle Button
FFDF - Circle Button
FFBF - X Button
FF7F - Square Button
FEFF - Select Button
F7FF - Start Button
EFFF - Up Direction
DFFF - Right Direction
BFFF - Down Direction
7FFF - Left Direction
Multi Buttons - Combining these is a little harder. L1 + Select would be FEFB.

- Max Reverse Joker Command Digits:

FFFF - No Buttons
FEFF - L2 Button
FDFF - R2 Button
FBFF - L1 Button
F7FF - R1 Button
EFFF - Triangle Button
DFFF - Circle Button
BFFF - X Button
7FFF - Square Button
FFFE - Select Button
FFF7 - Start Button
FFEF - Up Direction
FFDF - Right Direction
FFBF - Down Direction
FF7F - Left Direction
Multi Buttons - Combining these is a little harder. L1 + Select would be FBFE.

Playstayion 2 Joker Command Digits - Courtesy hellion (hellion00.thegfcc.com):

0001 - Select
0002 - L3
0004 - R3
0008 - Start
0010 - Up
0020 - Right
0040 - Down
0080 - Left
0100 - L2
0200 - R2
0400 - L1
0800 - R1
1000 - Triangle
2000 - Circle
4000 - X
8000 - Square
Multi Buttons - Add the values. 8200 - Square + R2

Dreamcast: Indices For DC Pad Commands - By UL1

The 'good' thing is: for the DC there are commands available that allow you
to activate a code in the game by pressing one or more buttons on your pad.
The 'bad' thing is : It isn't as easy as it is for the PSX...

The *BUTTON COMMAND* is easy to use because you just insert the values and
that's it. You can combine different buttons - just subtract the given
values from FFFF.

With the *L / R COMMANDS* it's a bit difficult...

There are two types :

- The ANALOG-L/R COMMAND

  For the *A-L/R COMMAND* the values for the buttons differ depending on how strong you push the button(s). So just the value for the default position of the buttons can be given - you can activate the code by saying :

  Code active if button isn't moved (use '0' instead of the question mark) or
  Code active if button is moved (use '1' instead of the question mark).

  Alternatively, if you want to activate the code(s) when pressing button(s) to the max, use FF00 for 'L', 00FF for 'R' or FFFF for both.

- The DIGITAL L/R COMMAND

  This one is easy to use because you use it the same way as the *BUTTON COMMANDS*. The *ANALOG STICK COMMAND* is similar to the *A-L/R COMMAND* but here you have two different values for the default position of the stick : 0000 or 8080 and there is no value for a max position. XXXXXXXX always represents the code address. Most of times the *BUTTON* and the *L/R COMMAND* addresses are placed consecutive in the memory, e.g. :

  BUTTON COMMAND
  01000000
  0000FFFB

  L/R COMMAND
  01000002
  0000FFFD

  so you also could use a combination of the two commands. That would read like this :
  01000000
  FFFDFFFB

NOTE : Because SEGA / the developers use almost all buttons for game play it's sometimes difficult to find a combination that doesn't affect other functions in the game. So I suggest to use the second Pad for activating codes in such cases.

- BUTTON COMMAND

  XXXXXXXX
  0000**????**

  A = FFFB
  B = FFFD
  X = FBFF
  Y = FDFF
  START = FFF7
  D-UP = FFEF
  D-DOWN = FFDF
  D-LEFT = FFBF
  D-RIGHT = FF7F

- A-L/R COMMAND

  XXXXXXXX
  000**?**0000

  0 = active when none of the two buttons is pressed
  1 = active when one or both buttons are pressed

```
L = FF00
R = 00FF
```

- D-L/R COMMAND

```
XXXXXXXX
0000????

L = FFFD
R = FFFE
```

- A-STICK COMMAND

```
XXXXXXXX
000?0000
or
XXXXXXXX
000?8080

0 = active when A-Stick isn't moved
1 = active when A-Stick is moved
```

E) Patch Codes

How do Patch Codes Work?

Patch codes are used to make a code string shorter. EG, You have five codes put together to give you "all weapons." Use the patch to shorten it to two codes. This is how it works -

```
50000A02 0000 +
80844CF0 FF5F       EXAMPLE ONLY! NOT A REAL CODE!
```

The first code is the patch, the second is the first code of the expanded STRING(a string is where the offsets will go up only a few digits for each code, EG. 100000, 100002, 100004, etc). The patch does not use an address, yet it is an instruction for the GameShark. The seventh and eighth digit in the patch tells how many numbers the second code will raise to get to the next code in the string. In this case, "2" is used, meaning the next code in the string must be "80844CF2 FF5F." Then the next code would also go up by two. So would the next one, and so on. The fifth and sixth digits of the patch are the digits that tell the GameShark how many codes are in the string. "0A" is used in the example, so ten codes are being used at once with only two codes! Also note, the codes within the string MUST have the same quantity digits!!!! It's possible to have as many as 255 codes used at once using this format. Maybe even more in the future. You can also change the quantity digits in the patch to make the values of each code in the string raise by a certain value.

This is what the above code would look like with out a patch -

```
80844CF0 FF5F
80844CF2 FF5F
80844CF4 FF5F
80844CF6 FF5F
80844CF8 FF5F        EXAMPLE ONLY! NOT A REAL CODE!
80844CFA FF5F
80844CFC FF5F
80844CFE FF5F
80844D00 FF5F
80844D02 FF5F
```

Which would you want to use?

Okay now to put a new twist on the Patch Codes. I'll use the materia modifier for Final Fantasy 7 as an example.
slot 1: 3009CE60 FFxx
slot 2: 3009CE64 FFxx

There are 200 slots but there are only 90 different types materia so to get them all that's how many codes you'll need. the quantity digits for those range from 00-5A and 5A equals 90 in decimal. The codes go up by 4 so we have:
50005A04 0000 +
3009CE60 FFxx

Is that the code to give you all the materia? NO! That code will give you the same materia in 90 different slots. To get all the materia we need to make the XX go up by one with each code. We do that the same way we make the first half of the code.
So here it is:
50005A04 0001 +
3009CE60 FF00

We just did the same thing with the last four digits as we did for the rest of the code. So which is better 2 codes or 90?

F) Encryption

F-1) Playstation 2 Encryption - Courtesy hellion & kpdavatar (hellion00.thegfcc.com)

There are 3 types of encryption that the PS2 Gameshark (and AR) V1/V2 use. The encryption type is determined by the (M) Must Be On code for each game. You may have noticed in looking through some codes that some games have a 1 line (M) code and some games have a 2 line. That extra line sets different encryption seeds. After seeing "DEADFACE" on GBA, I would've expected this extra line to have more than just 2 ways of encrypting. I'm guessing we'll see that in the V3 shark if anyone gets around to studying it.

The Codebreaker for PS2 only has 1 encryption type that we're aware of. No detailed information about the encryption itself is available at this time, but kpdavatar's encryptor/decryptor supports them.

- The "1456E7A5" Encryption Scheme

This was the original encryption method -- your 1 line (M) code. You may have seen codes like "3CA2B610 1456B00C" on some sites. The "1456E7A5" name was derived from the fact that it's commonly used in the values on encrypted codes. 1456E7A5, when used in that part of a code, decrypts to 00000000. The other 2 encryption types are the same wway. Here are the equations that are used for the encryption. The input to the encryption is denoted as a0a1a2a3 d0d1d2d3, and the output is denoted as A0A1A2A3 D0D1D2D3. Each a0, a1, etc is a two digit hex number. In the equations below, the '$' means that the following number is a hexidecimal number. XOR is a binary operation that can be performed on hex numbers using Windows Calculator in Scientific Mode. *Note: If you're not into equations, there's an encryptor created by kpdavatar. :)*

A0 = (a0 XOR $A6) - $6A
A1 = (a1 XOR $96) - $FF
A2 = (a2 XOR $01) - $7E
A3 = (a3 XOR $82) - $5A
D0 = (d0 XOR $D9) - $C5
D1 = (d1 XOR $3B) - $E5
D2 = (d2 XOR $1B) - $34
D3 = (d3 XOR $CC) - $27

- The "BCA99B83" Encryption Scheme

This encryption is used for games that have certain two-line (M) Must Be On codes. The first line of the (M) code should be "0E3C7DF2 1853E59E". Here are the equations that are used for the encryption. The input to the encryption is denoted as

a0a1a2a3 d0d1d2d3, and the output is denoted as A0A1A2A3 D0D1D2D3 . Each a0, a1, etc is a two digit hex number. In the equations below, the '$' means that the following number is a hexidecimal number. XOR is a binary operation that can be performed on hex numbers using Windows Calculator in Scientific Mode.

```
A0 = (a0 - $69) XOR $69   or   A0 = (a0 - $E9) XOR $E9
A1 = (a1 - $4F) XOR $4F   or   A1 = (a1 - $CF) XOR $CF
A2 = (a2 - $7B) XOR $7B   or   A2 = (a2 - $FB) XOR $FB
A3 = (a3 - $45) XOR $45   or   A3 = (a3 - $C5) XOR $C5
D0 = d0 + $BC             or   D0 = d0 - $43
D1 = d1 + $A9             or   D1 = d1 - $56
D2 = d2 + $9B             or   D2 = d2 - $64
D3 = d3 + $83             or   D3 = d3 - $7C
```

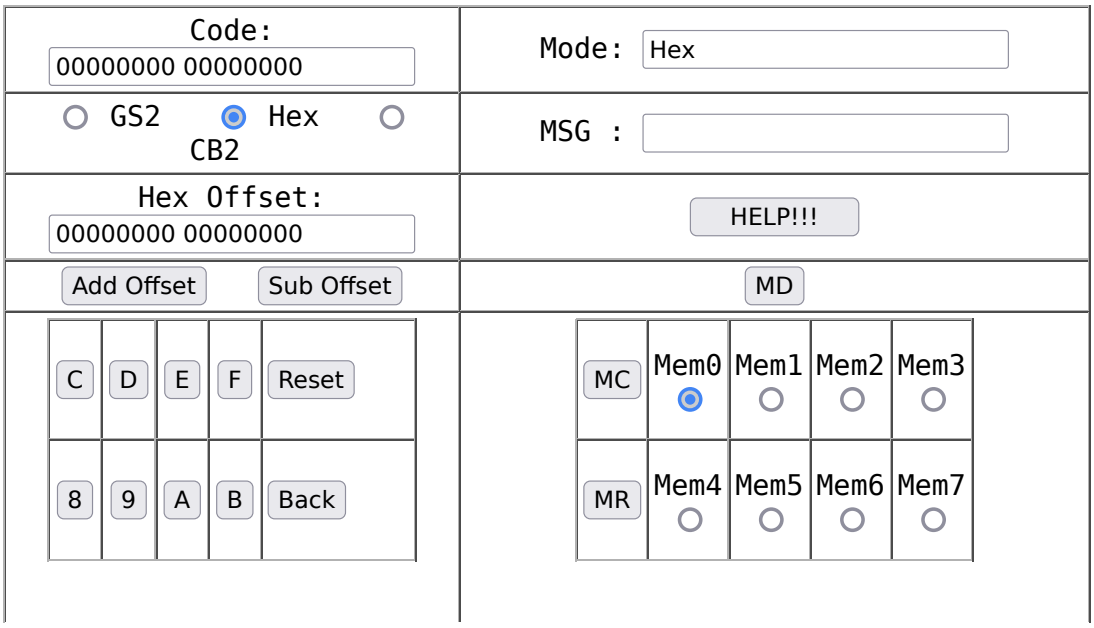
- The "F8FCFEFE" Encryption Scheme

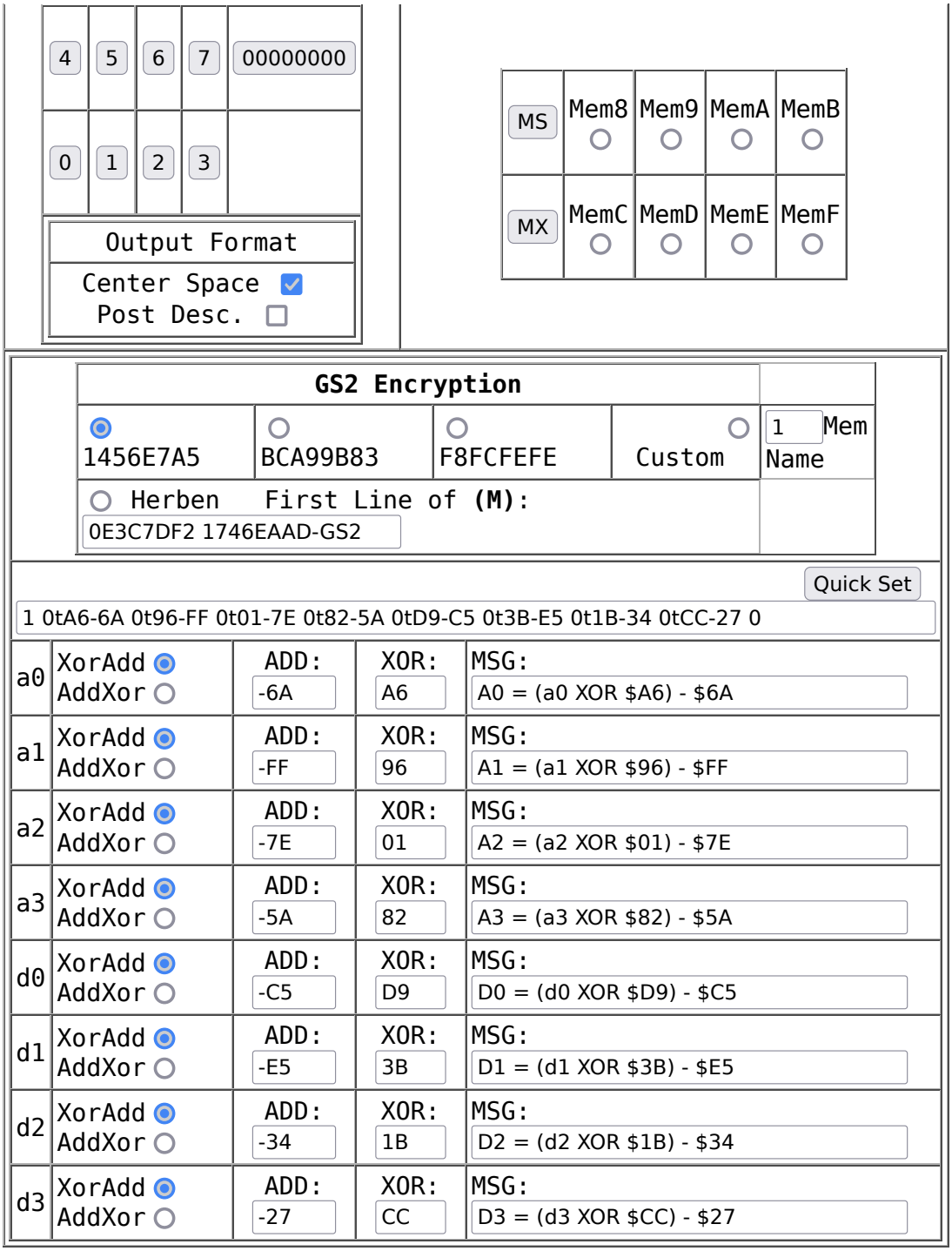
This encryption is used for games that have certain two-line (M) Must Be On codes. The first line of the (M) code should be "0E3C7DF2 1645EBB3". Here are the equations that are used for the encryption. The input to the encryption is denoted as a0a1a2a3 d0d1d2d3, and the output is denoted as A0A1A2A3 D0D1D2D3 . Each a0, a1, etc is a two digit hex number. In the equations below, the '$' means that the following number is a hexidecimal number. XOR is a binary operation that can be performed on hex numbers using Windows Calculator in Scientific Mode.

```
A0 = a0 + $1C                 or   A0 = a0 - $E3
A1 = a1 + $F7                 or   A1 = a1 - $08
A2 = a2 + $4E                 or   A2 = a2 - $B1
A3 = a3 + $CF                 or   A3 = a3 - $30
D0 = (d0 - $44) XOR $44   or    D0 = (d0 - $C4) XOR $C4
D1 = (d1 - $42) XOR $42   or    D1 = (d1 - $C2) XOR $C2
D2 = (d2 - $27) XOR $27   or    D2 = (d2 - $A7) XOR $A7
D3 = (d3 - $09) XOR $09   or    D3 = (d3 - $89) XOR $89
```

- Playstation 2 Code Encryptor by kpdavatar

You didn't actully want to do all those equations, did you? Well if you do, knock yourself out. Otherwise, check out kpdavatar's 1337 encryptor below :)





F-2) Sega Dreamcast Encryption

The Gameshark for Sega Dreamcast only accepts codes in their encypted form. However, Xploder and Codebreaker DC accept both RAW (decrypted) codes and Gameshark (encrypted) codes. If you have a DC shark and want to encrypt codes for use on it, then use DCCrypt. There's also a CD image of the Gameshark floating around that's been modded to accept RAW codes.

F-3) Converting between GSA and CBA - by DGenerateKane

Well, I've seen many people asking how, and I'm sure we will get more n00b's who will ask. So I whipped up this guide on how to. I did it in about 5 minutes, there

may be some mistakes and typos, correct them if you please. And I don't mind feedback or questions.

GBA Code Conversion Guide Version 1.1
By DGenerateKane
Last Updated: 03-03-03
Code Type List originally by Parasyte, edited by DGenerateKane
CBACrypt and GSACrypt by Parasyte

Table Of Contents

GameShark to CodeBreaker

1. Get a GS code you want converted. Example: Breath of Fire II Infinite Zenny 4C051DA2 3F3075BB
2. Put it in the left panel of GSACrypt, making sure to seperate the first and second 8 digits with a space.
3. Press Decrypt. The new code in the right panel is the same code now in RAW format. The example code: 12006870 0000FFFF
4. Now, refer to the code types to see what the name of the code type is, then check for the same code type for CB, and change the first digit accordingly. If there is no matching code type, the code cannot be converted. Example: 82006870 0000FFFF
5. Now, go to www.cmgsccc.com/gba (I think) for gsccc's CB codes, find the game that you are converting codes for, and copy the first line of the (M) code, it starts with a 9. If it doesn't have one, you don't need to convert it any further, you're done. The BoF II Seed code is: 9171AA80 F67E
6. Paste the 9 code into the left panel of CBACrypt.
7. Paste the RAW code you decrypted into CBACrypt UNDER the 9 code. Delete the first 4 digits from the second line of 8 digits. EXAMPLE: XXXXXXXX YYYYXXXX: Delete the four digits in the Y positions. Remember to seperate the first 8 digits and the last 4 digits with a space. The example code is: 82006870 FFFF
8. Press Encrypt. The new code in the right panel is the code now in CB encrypted format. (Ignore the first line, its the 9 code) Example Code: 0AD6BC51 CF46
9. Fine! (End)

CodeBreaker to GameShark

1. First, look at the codes for the game at GSCCC and check if it has a 9 code in the (M) code. If not, the code is already in RAW format, skip to step 5 Example: Breath of Fire II Infinite Zenny 0AD6BC51 CF46
2. Take the 9 code from the (M) and paste it into the left panel of CBACrypt. 9 Code is 9171AA80 F67E
3. Paste the CB code you want converted under the 9 code.
4. Press Decrypt. The new code in the right panel is the same code now in RAW format. (Ignore the first line, its the 9 code) Example Code: 82006870 FFFF
5. Now, refer to the code types to see what the name of the code type is, then check for the same code type for GS, and change the first digit accordingly. If there is no matching code type, the code cannot be converted. Example Code: 12006870 FFFF
6. Paste the RAW code you decrypted into GSACrypt's left panel. Add 4 0's to the begining of the second line of 8 digits. EXAMPLE: XXXXXXXX YYYY: Add four 0's in the Y positions. Remember to seperate the first 8 digits and the last 8 digits with a space. Excample Code: 12006870 0000FFFF
7. Press Encrypt. The new code in the right panel is the code now in GS encrypted format. Example Code: 4C051DA2 3F3075BB
8. Fine! (End)

Tools

Location of GSACrypt: ARCrypt Final 2_2
Location of CBACrypt: http://www.gscentral.com/lib/downloads/CBAcrypt.exe

Note for ARCrypt: You must have a program that will extract .rar files, such as WinRAR.

Info

Unconvertable Codes:
Codes that have a code type in one format but not the other.
(M) Codes. (Different Format)
ID Codes. (Different Format)
GSA 32bit codes. (CB only supports 8 and 16)
GSA GS Button Codes. (The CB for some reason doesn't have one this time around.)
Encryption Seed Code. (Different encryption obviously)
Since I know you are lazy, I have pasted below all the known GS and CB code types.

(see above section - Tolos)

F-4) Xploder/Xplorer N64 & PSX Encryption
-by Parasyte & Misfire

Some Xploder/Xplorer codes are encrypted, and some aren't. We're guessing that maybe only the "official" ones are encrypted. Perhaps we'll find out more about this later, but at least we can encrypt/decrypt them. :)

Encryption	Decryption
A0A1A2A3 D0D1	a0a1a2a3 d0d1
A0 = (a0 XOR $06) on PSX A0 = (a0 XOR $68) on N64 A1 = (a1 XOR $81) - $2B A2 = (a2 XOR $82) - $2B A3 = (a3 XOR $83) - $2B D0 = (d0 XOR $84) - $2B D1 = (d1 XOR $85) - $2B Alternate: A0 = (a0 XOR $06) on PSX A0 = (a0 XOR $68) on N64 A1 = (a1 XOR $01) - $AB A2 = (a2 XOR $02) - $AB A3 = (a3 XOR $03) - $AB D0 = (d0 XOR $04) - $AB D1 = (d1 XOR $05) - $AB	a0 = (A0 XOR $06) on PSX a0 = (A0 XOR $68) on N64 a1 = (A1 + $2B) XOR $81 a2 = (A2 + $2B) XOR $82 a3 = (A3 + $2B) XOR $83 d0 = (D0 + $2B) XOR $84 d1 = (D1 + $2B) XOR $85 Alternate: a0 = (A0 XOR $06) on PSX a0 = (A0 XOR $68) on N64 a1 = (A1 + $AB) XOR $01 a2 = (A2 + $AB) XOR $02 a3 = (A3 + $AB) XOR $03 d0 = (D0 + $AB) XOR $04 d1 = (D1 + $AB) XOR $05
Newer PSX 7K Encryption (PSX Only)	
A0A1A2A3 D0D1	
A0 = (a0 XOR $07) A1 = (a1 - (a2 AND $73)) + ((a3 XOR $90) - $F5) - d0 - d1 A2 = (a2 - (a3 AND $73)) + ((d0 XOR $90) - $16) - d1 A3 = (a3 - (d0 AND $73)) + ((d1 XOR $90) - $5A) D0 = (d0 - (d1 AND $73)) + $35 D1 = (d1 + $35)	

Newer PSX 7K Decryption (PSX Only)
a0a1a2a3 d0d1
d1 = (D1 - $35) d0 = (D0 + (D1 AND $73)) - $35 a3 = (A3 + (D0 AND $73)) - ((D1 XOR $90) - $5A) a2 = (A2 + (A3 AND $73)) - ((D0 XOR $90) - $16) + D1 a1 = (A1 + (A2 AND $73)) - ((A3 XOR $90) - $F5) + D0 + D1 a0 = (A0 XOR $07)

X) How-to Guide - The Hacking Begins

Are you ready to start hacking? Of course you are. If you have skipped directly to this section hoping you can read just this part and think you can be a GameShark code hacker, think again. It is HIGHLY recommended, indeed required that you read everything that has already been presented in this document. If you do not, you won't understand what most of the following states, and you will most likely not be successful in being able to hack GameShark codes.

This is the untouched, unchanged GB Hacking Guide written by DaRfUs. We include it here to honor a hacker who first desired to teach others his trade.

==========

GB Hacking guide Written by DaRfUs

First you start off by choosing what you want the code to do (start with something simple like an unlimited life code). Now go to "Game Trainer" in the GS menu. It will start the game. Walk around for a little bit WITHOUT getting hit. Now press the little button at the top of the GameShark. It will take you to the GS menu once more. Again go to Game Trainer. Now walk around this time I want you to get hit. Press the GS button as soon as the life goes down. You will be taken back to the menu once again. Go to the trainer, and select "Continue Trainer" there will be 4 boxes now here is the most important part... What happened to your life the 2nd time (the time you got hit? it went down right?) so you would pick the down arrow, because your life went down! Wasn't that simple! Now it will search for some possible codes(if you get a big # of possible codes then keep repeating step 2)

==========

What this says is "compare and contrast", which is what the guys like CodeBoy and CodeMaster do when hacking! They use the same method the GB GS trainer uses. In the text above, DaRfUs explains that you choose the "down arrow". This is not only because your life went down, but so the trainer will compare the two RAM Dumps(which were taken when you pushed the little button on top), and display only the codes that had the values lowered in the second dump. From what I understand, the GameShark Pros' trainer will work similar to the one on the standard GB 'Shark and the SharkLink/Comms Links.

With all this information, you should be cutting the edges in your mind. You should be gathering ideas right now. Thinking of what these trainers actually do. Discovering for yourself what you want to know about GS hacking.

A) One Small Step For Man...

Nintendo 64 -

This part discusses important information about things like key codes, and "Enable Codes". A key code is a code which is used to bypass the lock-out code set into the games header. When a key code is activated in the 'Shark, it will use that key code as a default for a one-time shot. There are currently four different key codes available. Check GameShark Central (www.gscentral.com) for the key codes or check the FAQ section of this document.

Most "locked" games need the use of an "Enable Code". There are also many games that use enablers that do not require Key codes. They simply allow codes to function and the code generator which is the software newer Game Sharks have function. This code is a code that is placed in with the others(usually named, "MUST BE ON" or "M" code).
The Enable code used the most is "DE000400 0000" but there are others.
Most recently, Donkey Kong 64 had a multi line enabler code in the 3.2 version to allow codes to work with the Game shark. The 3.3 Game shark now uses only one line of code. Enablers also allow the gamer to use what is called hi resolution codes which in order for a game to run would normally use the memory expansion pack and thus render the GameShark Code Generator unable to run since it requires this same memory expansion pack.

Any code beginning with "DE" in GS versions 1.08 to 2.1 will cause the 'Shark to freeze up when not used with a key code and needing game.
In my version 2.2, the code "DE000400 0000" will NOT freeze the Shark. Yet almost any other "DE Code" will freeze it.

As mentioned earlier, there is one key coded game that DOESN'T need an enable code. The Legend of Zelda: Ocarina of Time and when using the GameShark 2.2, you see a new addition to the key codes. It's a 32-bit block(aka instruction code). Now, three of the four key codes use the block "80 20 10 00". BUT, Zelda uses the block "80 19 00 00". Does this suggest that using a block different from the default, will make the use of an Enable Code nullified? I think not. There must be some defect that The GS has when using the "80 20 10 00" code. This calls for the 'DE' enable code to make the GameShark work! The GS 3.2 and up does however require an enabler code or M codes in order to work without freezing.

The DE is used to tell the GameShark where the value "3C0880" is located. Why "3C0880"? I don't know for sure, but my guess is that the value is where the RAM begins. So the RAM actually starts at RAM address "000400" and not "000000"(which also has the value of "3C0880"). The DE code does not overwrite the value of the given address.

PlayStation -

There are a few Enable Codes needed in PSX games(Crash Bandicoot for example). But I don't understand if the enablers are used for a lock-out, or to fix a small bug. This part of the section is in need of reference. I know a few people that could submit contributions, and I'm sure everyone will greatly appreciate it.

B) The Methods

Here are a few of the many methods used to hack GS codes -

Game Trainers - MadCatz and Datel have their own trainers. You can, however, get a trainer called "PC Comms Link" which will connect your platform to your computer where you can use a hex editor to view the ROM/RAM of your game. You will only be editing the RAM part of your games memory though. Comms Links may still be available from Datel.

Game Backup Devices - Some devices like "Doctor V64" and "CD64" have a built-in hex editor and GameShark function that allows you to view the ROM/RAM and create GameShark codes. There are many LEGAL issues regarding

the use of these type of hacking devices and as such the authors of this document have declined to include them within the scope of this work.

ROM and Hex Editor - Finally! Kong's secret hacking method is revealed! It is, at sometimes, seemingly impossible to hack using this primitive method, but it is well worth the trouble of it all! I was able to hack 54 Harvest Moon GB codes using this. When no one else could even get one code! How about those results?! I've also hacked numerous other codes this way. Definitely something to try out! Be Aware, using ROMs(they can be found on the internet) is ILLEGAL when used for purposes of your own entertainment, this means you can't use them just to play them! It may be possibly illegal just because most of the Nintendo game instruction booklets say, "WARNING: Copying of any Nintendo game is illegal and is strictly prohibited by domestic and international copyright laws. "Back-up" or "archival" copies are not authorized and are not necessary to protect your software. Violators will be prosecuted." The author does not endorse nor will support any information or the acquisition of obtaining game ROMs for legal reasons, and their use is explained here for comparison and illustrative purposes only. The author waives all responsibility for those readers that possess or attempt to possess such material, and the sole responsibility, accountability and legal consequences rest solely with the reader.

If none of this scares you, read on. You can hack this way by doing similar to what DaRfUs explains. If looking for an infinite health code, start a ROM with an emulator and DO NOT lose any health. Save the current state(check with your emulators readme file to see how to save a state or a snapshot), which will be like a ROM Dump. Then restart the ROM, go to a different spot you were at last time, lose health, and as quickly as you can, save the state/snapshot! Now load up a hex editor such as "Hex Workshop". Open the saves that you made(check with your emulators' readme file to find out what the extension will be). Choose Tools/Compare(in Hex WorkShop). You will get various comparisons. When you find a comparison that looks like it could be the one, change it in the second save that you made. Change the quantifier to something like "00"(this way, you will die if you have the right offset). Now load your emulator and ROM, load the SECOND save that you made(with the changed quantifier), if you die, or have less energy, you found the right offset! Now you need to find the beginning of the RAM. This might be impossible w/o previous training skills.

A really easy way to hack the beginning of the "valid" RAM, You can activate certain GameShark codes, and save a state, then find the values used in the codes, in the hex editor! EG- for Game Boy hacking, I use these codes together on any game-

```
013C00C0
013D01C0
014902C0
012203C0
```

Now I'd save a snapshot. Then Id enter hex workshop and find value "3C3D4922" which is nearly impossible to see twice(although on some games, the GS wrights to two different address locations with one code). When I find the FIRST string with that value, I write the beginning offset on a piece of paper. then I subtract that offset from "C000". Let's say the first "3C3D4922" value was at offset "000066D0". I would use this math problem-

```
  C000
- 66D0
  ----
  5930
```

I then take my value of "5930" and insert that many bytes(hex) into the BEGINNING of the file. Now search for value "3C3D4922" again, if you come to offset "C000", then you know that your GS code will modify offset "C000". And you can now go on hacking like that. The beginning of the valid RAM is "A000"(GS code "01xx00A0") and it ends at "DFFF"(01xxFFDF).

Other Methods - New methods of hacking will be included in future issues of this document as they become known and created.

B-1) Using Game Trainers

Using these Trainers is rather easy and usually brings up some codes. You will learn how to use the trainers by reading this section. You will learn to hack codes like "Infinite health", "Infinite Ammo", Debug Menus and even "activators".

Emulators such as "ULTRA HLE" for N64 ROMs and "Bleem" that run PSX games on the PC, are in the purest sense PC based programs that allow you to run an image (ROM) of a console game like Zelda64 on the PC. The emulators for the newest platforms require state of the art video and sound cards to be able to get a close enough effect as that of the real console, otherwise they result in poor sound and graphics display and game performance. The legal debate of having the game image (ROM) is ongoing but is clearly considered by some companies such as Nintendo as illegal. ROMS can be obtained via the web but please don't ask for any sites. The emu or emulator really is used for playing the game. Now the emu coupled with a hex editor can bring a hacking element into the scene by simply changing addresses in a "save" file on the PC which acts as the RAM which would be in the console cartridge. Loading these files after changing them can be the same as loading saved games. Using the Hex editor with the emu at the sametime can be considered like using a GS Pro and Game with the console. Using the hex editor requires no less patience or experience in looking for codes than using a GS Pro. By the way...a very good Hex editor I use is called the HEX Workshop and can be found on line at several web sites.

Trainers are earlier, PC based programs similar to the GS Pro. They required a PC Comms Links card for the PSX console where the Comms Link card allowed communication of the PC with the console. The N64 required an additional adapter that was in reality a watered down GameShark. This adapter was needed because the N64 GameShark never had the 25 pin port on it the way the PSX did. The legality of using a trainer is transparent in that you never made a copy of the game ROM. You were just scanning through the RAM addresses looking for active codes just as you do with the GS Pro. There are also several proprietary new trainers that are used by the pros that have not been released to the general public.

B-2) Hacking With GameShark Pro

Here we'll cover:

Hacking The Easy Stuff: Some examples.

The easy stuff is codes that are easily obtained. Such as inf. money, health, and ammo. A note to beginners, ***DO NOT SKIP THIS SECTION!*** Practice makes perfect and is everything to getting the feel of what hacking codes is about. You must crawl before you walk and find you will even run in a much shorter time if you heed this advice.

"One learns by doing the thing; for though you think you know it, you have no certainty until you try" - Sophocles

"Lo...try not, do or do not, there is no try" - Yoda

We will begin:

Infinite Cash-

To hack an Infinite Money code, first you must start up the trainer process with a Known Value search. While you have 0 money, search for value "0". Start the second part of the process, go back into the game. Now earn some cash, it doesn't need to be a large sum. Then do another search(for the value that you now have). Now, repeat the process until you find the code.

Infinite Health-

For a code like infinite health, you go through the same steps above, only you will search for an Unknown Value and you character will start with full health in step one, and less health in step two. So you choose the "less than" option. And when summarizing the quantifiers, notice that MOST(not all) games with a health bar will recognize full health at "0064"(100) or "00C8"(200). With this in mind, you can find the health code a lot easier!

Infinite Lives-

Now on to infinite lives. Do the same thing you did in the first example. The first step will be the default amount of lives, and the second will be one life less. if you started with 3 lives, the first quantifier should be "0003" OR "0002". Since some games start counting at 0 instead of 1, it may be possible that "2" = "3" But this is not usually done.

Infinite Ammo-

Another code is Infinite Ammo. This one might be a little trickier. If you do the first search with 100 bullets(search for 100), and the second with 50(search for 50). The code should show up.

I hope you know how to hack the easy codes now. So far so good.

Hacking The Intermediate Stuff

The intermediate stuff is like "In-Game Cheats", Have weapons/medals/items codes.

Have Weapon/Item codes-

If you wanted a code that gave you a weapon or an item, you would first take one readout without having the weapon/item, then a second with it. When you compare, choose "greater than". The first quantifier SHOULD be "0000" and the second "0001". If not, the second could be "0002/0004/0008/0010/

0020/0040/0080/0100/0200/0400/0800/1000/2000/4000/8000". The reason it would have so many is the game uses the switches(described earlier). Only the switches are represented in hex. You need a little luck when hacking a code like this.

Activate In-Game Cheats-

For a code like "Activate In-Game Cheats", you will take one readout without the cheat active, and the second with the cheat active. The quantifiers should read, "0000" & "0001". Good Luck

Character Modifiers- (submitted by Gold64007)

1. Select a character, start the game.
2. Push the freeze button and start an Unknown Value search.
3. Change to a different character, start the game.
4. Now do a "Different to" search.
5. Change some other aspects of the game (other than your character) (example, health, place you are in, depending on what kind of game it is).
6. Now search for "equal to".
7. Repeat steps 3 through 6 until you get a manageable amount of results (20 or less).
8. After you have a manageable amount of results, get into the view valid cheats screen. Now activate the codes one at a time until you find the character modifier.
9. Now in the view valid cheats screen highlight the Character Modifier code on the right side of the screen and push the Right C button to edit the value of the code put in "00" which is probably the first value that will work (though it could be higher).
10. See which character "00" corresponds with, write it down.
11. Now change the value to "01," see which character you have.
12. Repeat step 10, adding 01 each time until there is no effect.

NOTE: Exit to the main menu of the game before editing the code to keep the game from freezing.

Using this method, you may find two types of Character Modifiers, The actual character mod(select any character, always play as certain character). Or the character selected mod(your 'cursor' will always be on a certain character at character select screen).

Hacking P2 Controls CPU Codes (by HyperHacker)

This only works for a game where P2 can normally play and in 1 Player mode the CPU takes control of them.

1) Start up a game with Player 2 playing, search for 1.
2) Go to the same level in 1 player mode, search for 0.
3) Keep doing this until you find a few, set to 1 and the computer should stop (or not do anything in the next round). Set to 0 and in 2 player the computer should take over.

If that doesn't do it, switch the 1 and 0, and if it still doesn't work just do Equal/Different searches. some games like to use funny numbering/flags. (ahem, Mario Kart)

Hacking the harder stuff

The hard stuff is like activators, debug menus, cut scenes, walk thru walls, level modifiers, moon jumps, size modifiers, speed modifiers and Enabler codes.

Activators/Joker Commands-

An activator/joker command is a code that will activate another code at the touch of a button on the controller. These codes use the D/E prefixes. To hack an activator/joker, start a known value search on the game title screen. Take this search when NOT pushing any buttons on any controller. Next, hold the R Button for N64 or Tringle button on PSX, and push the freeze button while holding the button on the controller. Now search for the

following values-

N64 - "16-d"
PSX - "16-d" NORMAL
PSX - "4096-d" REVERSE

The two PSX versions are listed because some games use activator quantifiers that are different from the rest. The N64 activators are usually always the same. So if you do not find the Triangle button with value "32-d", try value "4096-d". When you find the code that you think it is, put the prefix to "D0" on the code. Then test it by putting any code after it.

Here is a quick example. Start the game but do not touch any buttons after coming to the opening screen. Press and hold the "R" button and press the GS Button to freeze the game. Release the R button. Do a known search for the value 16 in decimal, which is the equivalent of 10 in hex base. After the search resume the game and then press and hold L while pressing the GS button again. Release the L button and do a known search for 32 in decimal, which is the equivalent of 20 in hex. You should be down to 4 or 5 possibilities. Now let us say we try one of them to test. Let's pretend the code is 80012345 0010 you choose first. First Change the first digit "8" to a "D" which tells the Game Shark to execute the code that follows this new code you made, D0012345 0010 once and only once each time you press the R button. Now we pretend the code we want to use is for selecting a certain weapon, gun, 80022222 0001. So by combining the two codes;
D0012345 0010
80022222 0001

We get a code that turns the gun on every time we should press R. Now of course we must test the code to see if it works. If by chance you choose wrong, fear not, a matter of elimination of the other few possible codes will reveal the correct one. Once tested and proven you can read on for the next step.

Now we illustrate how we could de-select gun.
Well let us choose the 'L" button to do this. First, again we select the proper code to use as the activator as before. If you'd like to turn the gun code off, you can usually put another activator over the code you want turned off.
Example -
D0012345 0020
80022222 0000

Now by putting the two pieces together you can make a double activator let Let's you turn the gun on and off.
D0012345 0010
80022222 0001
D0012345 0020
80022222 0000
Now, the 'd0' codes are the activators. the one on top is the R button, and the one on bottom is the L button. The '80' codes are 8-bit gun code. When you push R button you get the gun and when you push the L you loss the gun.

This type of code can be used for many types of use. Image transformers, Size modifiers, Cut scenes, level modifiers, health and status modifiers, items modifiers... the list goes on and on. Activators come in 3 basic types. Odd or Normal type 1, even or Normal type 2 for the 8 bit type, and even or dual for the 16 bit type. In an odd 8-bit type the activator will always have the 8th digit as odd, where it will have the 8th digit for an even value. All 16-bit activators are even by definition simply by inspection that any two odd 8-bit values create a even number. The only difference in a 16 bit activator is the 2nd digit will be a "1", e.g., D1012346. A complete list of quantifiers concludes this section
on activators.

Normal Activator 1 Quantity Digits Button Pressed to Activate
00 No Buttons
01 Right Directional Pad
02 Left Directional Pad
04 Down Directional Pad
08 Up Directional Pad
10 Start Button
20 Z Button
40 B Button

80 A Button
Multi Buttons to use any combination of buttons, like, press Right Directional and Z to enable the codes. Just add the two digits up for Right Directional and Z. 01 plus 21 equal 21, so 21 would enable the codes when you push Right Directional and Z on the Controller 1

Normal Activator 2 Quantity Digits Quantity Digits Button Pressed to Activate
00 No Buttons
01 C Right Button
02 C Left Button
04 C Down Button
08 C Up Button
10 R Button
20 L Button

N64 Activators - Quick Tips ~by Viper187

Here's 2 tips for finding activators in a hurry:

1. Use more than 1 controller. Press the same button(s) on 2 or even all 4 controllers. Then when you view possibilities after after searching, you should see a group (2-4 depending on how many controllers you used) of results that are 6 or 8 hex apart. In the 200 or so games I've hacked so far, all but maybe 1 has had both a set that was 8 apart and at least one set that was 6 apart. Most times, these even use the same numbers as the last digit of the address. i.e. in the the typical '8 apart' set P1-P4 would be D1022204, D102220**C**, D1022214, D102221**C** and in the typical '6 apart' set P1-P4 would be D1022200, D1022206, D102220**C**, D1022212. The last digit on each address being 4,C,4,C or 0,6,C,2 respectively.

2. Use a 32-Bit Known Value search (PC Utils, GSCC2k2, etc). If press, for example, L + R + Z (value 2030), on controller 1 then search for that value as a 32-Bit value by adding 0000 to the end (20300000) you'll usually end up with only activators as your results; there can be some junk results depending on the value (button combo) use use though. I've found that 60200000 (Z + B + L) tend to work rather well, as does 4030 (B + L + R). Don't move the control stick when doing this though. That's what the 0000 is. :)

N64 Activators - A Slightly More Techy Method ~by Viper187

If you haven't noticed this yet, pretty much every N64 game has multiple sets of Button Activators in different areas of the RAM. There is 1 set that I refer to as the "good set" because each controllers' activator is seperated by a 32-Bit value, FF010401. This is how it's setup in respec tto addresses:

81055840 FF01
81055842 0401
81055844 ???? - P1 Button Activator (44 would be Activator 1 and 45 Activator 2)
81055846 ???? - P1 Control Stick Activator
81055848 FF01
8105584A 0401
8105584C ???? - P2 Button Activator (44 would be Activator 1 and 45 Activator 2)
8105584E ???? - P2 Control Stick Activator
81055850 FF01
81055852 0401
81055854 ???? - P3 Button Activator (44 would be Activator 1 and 45 Activator 2)
81055856 ???? - P3 Control Stick Activator
81055858 FF01
8105585A 0401
8105585C ???? - P4 Button Activator (44 would be Activator 1 and 45 Activator 2)
8105585E ???? - P4 Control Stick Activator
81055860 FE00
81055862 0000

Notice the repeating value? if you're using the PC utils or anything that allows 32-Bit searches, search for "FF010401". The only possibilities you get should be a nice set like that that are 8 hex apart, and the activators for each of the controllers plugged in are between them. :)

Curious as to what those 'FF010401's actually are? It appears the game uses them

to keep track of what controllers are plugged in and such.

Debug Menus-

This is NOT easy at all. You will need to search through MANY, MANY codes if you have no base to go on. If you are looking for a debug menu that is already known, just take a readout with the menu NOT activated then the second with it activated choose "greater than". The quantifiers should read "0000" & "0001". That wasn't so bad was it?

Now it's time to hack the Debug Menus that have never been seen before! To pull this off, you must start playing the game and do a few million "equal to" searches. You will see MANY codes that have the values as "0000". Well, ONE off those "0000" codes MIGHT be a debug menu. That is, if there is a menu in the game, IN THAT SECTION OF THE GAME. This must be the hardest code there is to hack. Are you up to the challenge?

Cut-Scene Modifiers- (Submitted by SubDrag and macrox)

-Press the freeze button while a cut-scene is playing and start an Unknown Value search.
-Play the same cut-scene, do an "equal-to" search.
-Play a different cut-scene, search for "different-to" values.
-THROW OUT ALL CODES that have value "FF"... Look for everything else, turn all codes that are nor "FF" on(some of them at a time). Play a cut-scene, if you play the same cut-scene no matter where you are, turn off a few codes(WRITE THEM DOWN), until you get the activate codes that do not play the same cut scene. Eliminate those codes.
-Turn on one code at a time(that you've written down) until the cut-scene never changes.

OK, now just play all the cut-scenes, press freeze button, write down digits, repeat until you get all the cut-scenes... If you have been through all cut-scenes and notice missing digits, try those missing digits in your new GameShark code. Sometimes the game will freeze with the missing digits, but you may find a hidden, or 'unfinished' cut-scene.

Cut-Scene Modifiers - Code Generator and Memory Editor Method -(submitted by Macrox)

- Press the GS button while the opening cut scene of game is playing.
- Do an unknown search.
- Wait till next cut scene and do greater than search
- Restart the game. Wait for opening cut scene. Do a less than search.
- Again wait until next cut scene and do greater than search.
- Wait for another new cut scene and do another greater than search.

You may have to do this for quite some time to get the possibilities down. If you feel you are not getting anywhere try the following.

- At the opening cut scene do a Known search of 0 or 1. Make an assumption.
- Wait until the next cut scene and search for values equal to 1 or 2 depending what you choose the step before this.
- Restart game and wait for opening cut scene. Do known search for 0 or 1.
- Again wait for next cut scene and do a search for 1 or 2.

Level Modifiers and Cut Scene Modifiers - by macrox

Now for my secret. Many games are programmed where the level modifier is addressed close to the cut scene modifier. This is how Castlevania and Castlevania LOD level and cut scene modifiers were hacked.

- Start game and take initial unknown search on opening level.
- Either wait for interlevel or new level and take greater than search.
- Wait for next interlevel or new level and again take a greater than search.
- Play that level awhile and take an equal to search.
- Restart the game and take less than search on opening level.
- Codes should be dropping in number of possibilities now.
- Once you get the codes possibilities down to 20 or so write them down.
- Start the memory editor and look for the codes you have on your list. The codes you have should have values no greater than 3 or 4.
- Try changing the code value in the memory editor to something smaller or larger. For instance, if the value is 3 try changing it to 6. This should

take you to a different level. Once it works you have the level modifier and interlevel modifier.

Now that you have the level modifier you can find the cut scene modifier with the memory editor by inspection. Inspection means looking around by going up and down the address screen while a cut scene is playing for that level. Since most new levels run a cut scene this makes things easier. You should find the cut scene modifier rather quickly. Try changing the value after the current cut scene is done. WARNING! As I have often stated when submitting cut scene codes: Running cut scenes out of their natural location or level might cause the game to freeze or do weird things like have your character fall into empty space once the game resumes.

I will say more about the memory editor again later.

Moon Jump codes- GLEE method

Be sure that you have an activator/joker code before you begin.

1) Start the game, while your character is ON THE GROUND, push the freeze button and start a 16-bit unknown search.
2) Move your character to a different location. With character on the ground, push the freeze button search for equal-to values.
3) Repeat above steps a couple more times.
4) Push the jump button(if you have no jump button, fall off of something really high). While your character is in the air, falling, push the freeze button and search for greater-than values.
5) When your character lands, and is on the ground, push the freeze button and search for less than values.
6) Repeat from step one until you narrow down the possibilities.
7) To test the codes available as a moon jump code, jump up in the air and push the freeze button while the character is going up(if you have no jump button, fall off of something really high, as your character falls, push the freeze button).
8) View possibilities. Disregard any codes with value "0". Activate all possible codes(you should have less than five before doing this). If your game doesn't have a jump button and you had to fall, activate all codes and change the values to "0".
9) Return to the game, If you character continues to go up(or is not falling, but is in the air), then one of those codes is the Moon Jump.
10) Test codes individually. When you have the one you want, find the best digits for it by making it a few numbers lower, or higher.(for the fallers, Check to see if the code has value of around "Cxxx" when falling. If so, use a value of around "4xxx".
11) A value too high will cause the character to fly into the sky too fast and may cause the game to freeze. A value too low will cause the character to rise unnoticeably slow.
12) To finish the code- Put an activator in front of the code, set the activator digits to a button you would like. Now test the code, hold the button you set it for, if your character doesn't move, but you have the right code, push the jump button or fall off of something while holding your Moon Jump button. The character should rise!

    Skip these last steps if your code works with out falling off of something.

13) Try setting the activator to the jump button, this may help.
14) If it doesn't work, or you have no jump button, you need to find out what codes tell the game that the character is in the air. These codes are REALLY tricky.
15) You'll need to start an unknown value search while stationary, then one stationary, elsewhere(perform equal-to search), then fall from something and do a different-to search. Land, do different-to, move to another place, equal-to.
16) You get the idea. When you find the code that instructs that the character is in the air, use it on the Moon Jump code you made before, and put that on an activator as well.
17) Later we will show you how to use the memory editor to hasten the finding of the moon jump code and testing it. Which ever way you prefer always remember GLEE - , Greater, Less than, Equal, Equal. Good Luck!!

Moon Jumps - 32-Bit Signed Method - by Viper187

If you're using GSCC2k2, you might've noticed the 32-Bit Signed search options. I'm going to show you a way to find Moon Jumps using this. I can't say if this method is faster than GLEE. It is a different, slightly advanced approach to Moon Jumps.

Note that "Signed" searches read hex values somewhat differently. Values are compared ranging from 80000000 to 7FFFFFFF. Why start in the middle like that? Because it reads 800000000 as -2147483647, not 2147483648. That's right. Negative values. FFFFFFFF is now seen as -1, FFFFFFFE as -2, and so on. This is useful here, since most games use a negative value when you're falling and/or on the ground, and a positive value when you are going up -- typicly 4xxxxxxx at the peak of your jump. Now the actual searching method. Reason I wrote this is because of games like Banjo-Kazzoie. I noticed that the Moon Jump address on there is the usual 4xxxxxxxx when in the air, but when you're on the ground it's BF800000. Try doing a normal Greater/Less when you're in the air and on the ground with that one. I don't think it'd work out that well; someone apparently found a way to make it work though. ;)

If you can jump:
1) Get hooked up, start game, etc and start a 32-Bit Unknown search, Signed.
2) Now jump and as you start to go up, do Greater Than.
3) Then quickly resume and do Less Than as you start to fall.
4) You can then continue to do Less Than another time or 2 til you hit the ground.
5) Repeat until you get the results down a ways and you should know what to look for (Cxxxxxxx going down, 4xxxxxxx going up).
6) Freeze or set to activator to test using 4xxx. The first 16-Bit half of the 32-Bit value should be enough. Say it found 800448A0. To modify it as a 32-Bit code you'd have to use both 810448A0 and 810448A2. But with Moon Jumps, the first half (810448A0) is enough.
7) You'll notice I didn't say to do any Equal To searches there. If for some reason you just can't get the results down, you could try it, but it can also eliminate the result we're looking for. This is because some games have been known to change the MJ address when your character is resting. i.e. you feet are tapping, or your can't does some kind of bounce type thing while standing. This may not seem like a big deal, but an Equal To at the wrong time and you'll be starting from scratch.

If you can't jump:
1) Get hooked up, start game, etc and start a 32-Bit Unknown search, Signed.
2) Find the highest place you can and try to commit suicide (walk off). The point is to get as much time in free fall as you can.
3) As soon as you start moving downward (aaaah!) start doing Less Than over and over til you finally hit the ground. If you get the possibilities down enough to pick out the right one, good for you; otherwise head back up for some more frequent flyer miles.
5) Repeat until you get the results down a ways and you should know what to look for (Cxxxxxxx going down, 4xxxxxxx going up).
6) Freeze or set to activator to test using 4xxx. The first 16-Bit half of the 32-Bit value should be enough. Say it found 800448A0. To modify it as a 32-Bit code you'd have to use both 810448A0 and 810448A2. But with Moon Jumps, the first half (810448A0) is enough.

Size Modifiers -

This type of code can be considered moderate to hard to hack depending on whether or not the object in question normally changes size, e.g., flattens, or whether the object in question normally stays a constant size, e.g., gates, trees, rocks, doors etc. The former description would be intermediate difficult to hack while the latter hard to hard. Let us break up the two types of difficulty:

Objects can normally change shape: By Subdrag

- Start the game and identify the object you want to hack a size mod for.
- Do an unknown search
- Move around the screen, and even change the area you are in if possible.
- Do an equal to search to unload a lot of junk codes from the search.
- Now do whatever you have to do to get the object to normally change shape. Get hit, fall, flattened or flatten, squeeze...you get the idea.
- Before the object regains it normal shape do a less than search.

- Once the object regains it normal shape, take a greater than search.
- Repeat this several times for whatever time it takes to get the codes down reasonable number. Inspect the possibilities and look for those codes that have quantifiers (in hex base) in the range of 3C00 - 4200, which is the typical values used in N64 and PSX games. Another big hint is to look for groups of 3 codes close to each other and having close to if not the same quantifiers. The reason this works is simple. The N64 and PSX are using 3D objects, hence you need 3 codes, one for each dimension.

Objects that do not normally change shape: by macrox

This type of code is harder than above to hack. You have very little to go on regarding using the code generator (search engine) with the Game Shark alone. We will also in conjunction with the PC hacking utilities use the memory editor further aid and organize our search. In fact our example will show how using the memory editor is a very powerful tool to use and will be our second example in using the memory editor to hack a code.

- Start your PC and start the hacking utilities. Insert the Game shark and game into N64 attached the cable. Turn the N64 on and do a detect on your PC.
- Start the game and identify the object you want to hack a size mod for. We will use an example from Castlevania and Castlevania LOD for this.
- We have identified we want to hack a size mod for Cornell the Werewolf. (We will be pleasantly surprised to find this will pan out for the other characters as well later.)
- Ok. Now. We already told you that N64 usually uses several types of values for size variables in their games. Typical values of 3F80 and 3C00 or 3DCC are not uncommon to see.
- Making an assumption that the value is one of the above may or may not pan out for the hacker. Patience, endurance and luck are the rule of the day here.
- Using the PC hacking utilities perform a known search for your value. We suggest you break down the search by ranges, such as 800XXXXX - 8010XXXX,
  then 8010XXXX - 8020XXXX, etc... ALSO use the range value 3C00 - 4200. Doing this will shorten your search time and minimize guess work.
- By inspection, look over the possibilities and observe is you see the group of 3 we mentioned before. If the results are not too large you might want to try changing 1 of 3 codes you see on your list. Observe if the object on the screen has changed by doing this. Using the process of elimination you will either exhaust your list or hit pay dirt. No pay dirt? Perform another search but change the address range in the search.
- Again, by inspection check for the group of 3.
- Eventually, you will find the code. NOW, the question is this, Is this code valid for any level in the game? If not you will have to repeat this process for each and every level and interlevel too perhaps. Knowing whether a constant offset exists for your game is VERY helpful and will save you tons of work. We discovered that we did not have constant offsets. So our search routine took
  us quite a while in a monotony that would have quite easily discouraged a beginner hacker. That is why we stress patience and endurance so much. Anything worthwhile is worth work to coin a phrase.
- Luck was with us in some regard doing this hack. We found that even though the size moderator code changed for our character for each level and interlevel...it was in fact the same value for all the other characters in the game as well. We only wish we had such luck hacking with moon jumps as they were different for all level/interlevel and characters as well.
- Finally, there is one more secret we would like to share. We have often noticed that size modifiers are usually close to location modifiers. Location modifiers being the spatial coordinates the object has while having its own size coordinates or modifier. This being the case one could hack first the location modifier for an object and once knowing that use the memory editor to inspect "look around" at nearby addresses for other groups of 3.
  Those without access to a PC or use of the hacking utilities can none-the-less still carry out the search but should be warned that this will usually be a rather longer process and might not yield expected results. Many hackers that use the memory editor have found many an unexpected code by simply looking around for "patterns" such as groupings of 3 for size mods,

location mods etc. The memory editor has been very useful to inspect addresses close to known codes to see if that code could be further enhanced or whether a new code could be found nearby. We encourage the beginner to not give up in frustration but understand the hacking is not always going to hit pay dirt each time you search. Indeed, you will learn to appreciate the workings of game programming and find as time passes you will "know" the game code as it you wrote it yourself. A feeling for the game is a valuable but truly subjective experience for any hacker.

Finding X, Y, and Z Coordinates - by Viper187

Coordinates are what the game uses to tell where exactly something is (i.e. Player 1). These are almost always found in a group of 3 32-Bit values, that we refer to as X,Y and Z. X and Z are for left/right & up/down location and Y is your vertical location. Y is NOT how high off the ground you are, but how high you are in general. Thus, if you go up a hill, you are at a higher.

My method here requires hooking your shark to your PC and using 32-Bit Unknown Value searches with either Interact's old program or GSCC2k2. However, this should work about the same way with a 16-Bit Unknown search on the in-game code generator -- just might take longer to get the results down, and there's a question of which 16-Bits of the coordinate is changing as you move. If you're going to do this, just make sure you move a good couple feet at a time. This should keep the upper half of the value changing. If you only move a tiny step then you may end up rolling over the lower 16-Bits so it becomes less when you're still searching greater. Confused? by that? Ok. Take an X coordinate for example. It's 32-Bits, we'll say 3DE93376, now as you move, that increases or decreases depending on the direction you're going. Now what I mean by the 16-Bit thing, is when you're searching you want to move enough to change the 3DE9 (upper) half because as that changes the 3376 (lower half) could be anything from 1818 to A2B4. If it was 3ED9E02A when you started, it could end up 3ED9F5CC when you move, or 3F882046. Now you can see why I say we want to be careful of how much we're are moving, and why comparing the whole 32_bit value with the PC programs is easier than comparing the 16-Bit halves.

1) Get hooked up and into the game as usual.
2) Decide which coordinate you want to search for. Remember they're almost always found in that group of 3 anyway, so you should get them all from finding one. It's just a matter of what you're more comfortable searching for. I prefer X or Z, so I'll use that here...
3) Start a 32-Bit Unknown Value search
4) Now pick a direction (I'd go forward) and move that way a few steps or so, depending on how much room you have before you hit a wall. We want to be able to move the same direction at least a couple times, then change. Now once you've moved a few steps, search Greater Than. Next, move a few more and do Greater Than again. Repeat this as desired, then go to step 5.
5) After you've done a few Greater Thans, start moving in the opposite direction and do Less Than searches.
6) You should be able to get the possibilities down enough that way. If you need to though, you *could* stay in 1 place and do an Equal To or 2. I must warn you though, that you coordinates can change on a game even when you're not actually moving. Ever notice how your character on some games with sort of shuffle side to side or something? The coords could be changing as the character does this. Just a heads up :)
7) Once you ge tthe results down, test them. Either freeze the value and see if you can still move, or change it a lot and see if it moves you somewhere else. Some games may write the coords to fast for you to just change them in the mem editor, so if for some reason you find what you think should be the group of coords and they just keep changing back when you try to change them, this is way.
8) No coords? Do the same thing again but search the opposite directions (do Less Than where you did Greater and vice versa). This should take care of it. If for some reason you still can't find them, then you could try 16-Bit searches (maybe the game only uses 16-Bit -- like games with a real short range of movement), or you could try the same 32-Bit searches, but do them Signed.

Coords are fairly easy to find on most games, once you get used to it, but as always there are problem games. If you find another strategy, don't hesitate to let us know. :)

How do I use the Memory Editor? (Submitted by Kamek)

The most common question I see on the hacking boards is, "How do I hack with the memory editor?" I'm not the least bit surprised -- when you enter the Memory Editor, it looks like some kind of programmer's screen.

First off, let me explain the big myth about the Memory Editor. Some people think that just by doing a text search, they can unlock special areas in the game. THIS IS NOT TRUE! The Memory Editor just doesn't work that way. Suppose you loaded up Zelda and searched for "Master Sword". You probably wouldn't find it, even though the Master Sword is in the game. A text search is just that -- a text search. Some games encode their text, so you might not always be able to come up with results in the text search. HOWEVER, you CAN use the Memory Editor to possibly unlock extra levels, characters, and other stuff. I'll explain more about that later.

Here is what the Memory Editor CAN do:

* You can use the Memory Editor to unlock game secrets that are accessible.
If you have unlocked a certain character, then you can use an unknown search to find the memory address that tells the game you have that character. By changing nearby values, you can attempt to convince the game that you've found other characters that you haven't really found. BUT, you must have a memory address to start from. Plus, the character MUST be accessible. You probably won't be able to play as the Magikoopa in Mario Kart 64, because he was removed completely from the game. (You can't have a 4-player Grand Prix either, because the game wasn't programmed that way.)

* You can use the Memory Editor to find secret passwords.
If you know a cheat password for a game, you might be able to find other codes. For example, the cheat "NOYELLOWSTUFF" in Diddy Kong Racing removes all the bananas. If you use the Memory Editor's text search to look for "NOYELLOWSTUFF" in memory, you might find other passwords nearby!

* You can use the Memory Editor to find weird stuff.
Load up Zelda OOT and open the Memory Editor. Search for the word "light". You'll be amazed at what you find! Solid proof that Zelda 64 was originally designed for the 64DD!

Breakdown of the Memory Editor

```
80010000: 00 00 00 00 00 00 00 00 ........
80010008: 00 00 00 00 00 00 00 00 ........
80010010: 00 00 00 00 00 00 00 00 ........
etc..
```

(Note: On some TV screens, the left and right parts of the Memory Editor are chopped off. Unless your TV has a Horizontal Size knob or an equivalent, there is no way to fix this.)

So what is all this mumbo-jumbo? Let's break it down..

80010000:

This first part of the line is the memory address you are viewing. Actually, you can chop off the "80"

if you want, and you'll get the real memory address. (The 80 is just for reference)

```
          00 00 00 00 00 00 00 00
```

These are the values in the memory. If the number on the left was 80010000, then the first value is the value in memory location 80010000, the second value is what's at 80010001, and so on.

```
                                  ........
```

You won't see dots, they'll probably be blanks, or strange characters. This is only useful when looking for text, and you can ignore it unless you're looking for text.

The N64 memory starts at 000000 and goes up to 3FFFFF, for your information.

Now, first of all, I would not recommend using the Memory Editor to find codes from scratch. It's practically impossible to find a code from scratch. Here are some things you can do with the memory editor

Suppose you were going to hack a code to get all the characters for Super Smash Brothers. (It's been done, but this is just an example.) Let's say that after several unknown value searches, you find out that 800A4938 0008 gives you Ness. Open up the Memory Editor and go to location 800A4938. You'll get something that looks like this:

```
800A4900 xx xx xx xx xx xx xx 00
800A4938 08 00 xx xx xx xx xx xx
800A4940 00 00 00 00 00 00 00 00
```

When searching for "got character" modifiers, common values are usually powers of 2. (2, 4, 8, hex 10, etc.) Since the game just takes the power of 2 out of whatever the value in memory is, you can usually set the value to FF and possibly get more characters. So make sure all codes are off, then change the 08 to FF:

```
800A4900 xx xx xx xx xx xx xx 00
800A4938 FF 00 xx xx xx xx xx xx
800A4940 00 00 00 00 00 00 00 00
```

Now, go back to the game and enter the Character Select screen. Wow, you've got Jigglypuff! (If you didn't already have Jigglypuff, and got Ness first, then you play too much SSB!!) Let's try nearby values to get Captain Falcon and Luigi as well.

```
800A4900 xx xx xx xx xx xx xx 00
800A4938 FF FF xx xx xx xx xx xx
800A4940 00 00 00 00 00 00 00 00
```

Change the value at 80AA4939 to FF and return to the game. The character select screen will need to be reloaded for the changes to take place, so exit and return. They're all there!

Now that is how you find extra characters. Hmm... But what if you change other values? Perhaps you could unlock a certain classic stage of the Mushroom Kingdom. Let's try some fiddling around in the memory editor.

```
800A4900 FF FF FF FF FF FF FF FF
800A4938 FF FF FF FF FF FF FF FF
800A4940 FF FF FF FF FF FF FF FF
```

Change them ALL to FF. This way, you'll know that if you find something, one of those values unlocks it! Quit the Memory Editor and check out other areas of the game. Go to Options -- the Sound Test is there! One of the values you changed is definitely a modifier to get the Sound Test!

Now, go back to the Memory Editor, and change everything back to 00. Then, start from the Ness/Jigglypuff Modifier and work your way backwards, changing each 00 to FF and reloading the options screen to see if you got the Sound Test. Or, you could start at 800A4939 and go upward. However, it just so happens that the Sound Test, Item Switch, and Mushroom Kingdom modifier is right before the Ness/Jigglypuff modifier!

800A4900 00 00 00 00 00 00 00 FF
800A4938 08 00 00 00 00 00 00 00
800A4940 00 00 00 00 00 00 00 00

Congratulations! You hacked your first Memory Editor code!

What else can the Memory Editor do? I'll explain how I found the Saffron City Pokemon Anti-Modifier and other codes.

I was playing with the Yoshi's Island "Stay on cloud forever" code on one day. I forgot and left it on, then played a game at Sector Z. Suddenly, the game froze. Somehow, the Yoshi cloud modifier was causing something to happen in Sector Z! Not all game freezes are bad -- sometimes it means you've stumbled upon something big!!

I wrote down the cloud modifier location (80131400 region), restarted without the Yoshi cloud code, then watched a CPU battle at Sector Z, going into the Memory Editor every now and then. Values in the Memory Editor changed as the Arwing entered and left the arena. I couldn't find any pattern, so I tried poking around that same area during a battle at Saffron City. That's when I noticed the timer.

Every time I went into the memory editor, there was this one memory location that kept decreasing. Then, when a Pokemon came out, it suddenly jumped to a higher value, then started over again. I found the Pokemon Timer! To test my theory, I changed the memory value at 13140E to 00 01, then resumed the game. A Pokemon came out!

You might not be as lucky when looking for codes, but if a code for a certain stage freezes the game on another stage, check out that area in the Memory Editor. You might find something good, or you might not. It all depends on how the game is programmed.

If you remember only one thing from this document, let it be that the Memory Editor is not magic. The magic is in you, and with some skill and a little bit of luck, you may stumble upon some great codes! In summary:

* You can only unlock areas that were left in the game.
* Things that were either partially or completely deleted from the source code of the game probably can't be accessed.
* You can quickly find neighboring codes to unlock more characters than what you searched for in the

Code Generator.
* You can find cheat codes, on some games.

One final warning. You must be careful with the Memory Editor. It's a ticking time bomb in some cases. If you put the wrong value in the wrong memory location, you could end up freezing up your game. In other cases, you might accidentally overwrite part of your game's SRAM and either lose saved games, or ruin the cart. But usually, if you stick to the areas of memory you find using the Code Generator, you won't have any problems.

Happy hacking!

How do I use the Text Editor? (Submitted by Sutaz)

I shall use the game Mortal Kombat 4 for an example.

I am playing with the fighter Fujin. When I win a round, it says on the screen "Fujin Wins". Well, what if I want it to say "Sutaz Wins"? Easy.

Press the GS button to get to the in-game menu. Choose memory editor. Press A to get the search menu. Choose Text Search. Type in the name Fujin. Press search. We come upon the first set of digits that spell the word Fujin but this text is in his biography screen. We don't want to alter that text now, we want the "Fujin Wins" text. So tap A again to continue searching for Fujin text. After 2 more searches, we come upon the group texts for all the characters that say "x person wins". Looking at the right of the screen and search for the term "Fujin Wins". Now we look to the left of the screen and see the location code for "Fujin Wins" which is 8004C0E8. In the middle of the screen, we see a set of 8 airs of numbers. What are these? ASCII digits. These are the numerical code for the letters in the text. There are 8 pairs per line of code. So, the line for "Fujin Wins" looks like this:

8004C0E8 - 46 75 6A 69 6E 20 57 69
8004C0F0 - 6E 73

46=F
75=U
6A=J
69=I
6E=N
20=(space)
57=W
69=I
6E=N
73=S

So, with the text editor, just look at the ASCII digit menu and choose the numbers to replace Fujin with Sutaz. Press Up-C or Down-C to cycle through the digits. With the text editor, the new phrase "Sutaz Wins" would look like this now:

8004C0E8 - 73 75 74 41 7A 20 57 69
8004C0F0 - 6E 73

73=S
75=U
74=T
41=A
7A=Z
20=(space)
57=W
69=I
6E=N
73=S

That would be the line to change the NS if you wanted. Now how to put this in your GameShark. Each line of code ends in a string numbering by 8's; 0,1,2,3,4,5,6,7 or 8,9,A,B,C,D,E,F. Each letter in the text "Fujin Wins" has its own separate code. Example;

To put "Sutaz Wins" as a code in your GameShark to replace "Fujin Wins":

```
8004C0E8 0073 = S
8004C0E9 0075 = U
8004C0EA 0074 = T
8004C0EB 0041 = A
8004C0EC 007A = Z
```

No need to enter the blank space for this example, or the term "Wins".

But let's say your name is longer than 5 digits and you need more space. You got two options; look at the last letter, which in this case is S, and see if there are more "free" or "empty" digits after it. Example: The line of code for the NS in "Fujin Wins" looks like this in the text editor:

```
8004C0F0 6E 73 00 00 53 68 69 6E
```

You see that you have two slots that are 00 00. Those are "text breaks". 00 or 20 are used to separate text. You could now use the 00 right after the 73 slot to spell a 6 letter word like Macrox. But if you use the second slot 00, you will run into the 53. Look at the text to the right and we see to conserve space, the programmers put another character's Wins text on this same line which is Shinnok. Look at the code above again; 53 68 69 6E are S H I N, obviously for Shinnok. Now if you want to, you could do option 2 for adding more space for bigger words by "bleeding" or running over into Shinnok's text. You could take all of his digits and spell bigger words like Darth Maul Wins. Here is the catch, if you choose to fight with Shinnok, his Wins text will not appear because Fujin has stolen it. Get it? You must turn all excess digits to 20 to create a blank space and you must leave at least one 00 between each character's beginning codes or you will bleed into their text.

Here is a table of ASCII - numerals used in the text editor:

A = 41/61	B = 42/62	C = 43/63	D = 44/64	E = 45/65	F = 46/66
G = 47/67	H = 48/68	I = 49/69	J = 4A/6A	K = 4B/6B	L = 4C/6C
M = 4D/6D	N = 4E/6E	O = 4F/6F	P = 50/70	Q = 51/71	R = 52/72
S = 53/73	T = 54/74	U = 55/75	V = 56/76	W = 57/77	X = 58/78
Y = 59/79	Z = 5A/7A	0 = 30	1 = 31	2 = 32	3 = 33
4 = 34	5 = 35	6 = 36	7 = 37	8 = 38	9 = 39

Other Symbols:

@ = 40	" = 22	# = 23	$ = 24	% = 25	& = 26
' = 27	(= 28	* = 2A	+ = 2B	, = 2C	- = 2D
. = 2E	/ = 2F	: = 3A	; = 3B	= = 3D	? = EF
[= 5B	\ = 5C				

How to hack Walk through walls (WTWs):
(Used by special permission from Code Master)

Information That Is Needed To Know Before Tackling This Code

While your character is walking, or while they are not walking, there is a value in RAM that tells the CPU whether or not you are able to walk at all. This value is used to tell the CPU whether or not to move your character on the screen. I call it the 'Collision Detection Address', because basically, the game checked to see if there was any structures or walls that you would 'collide' with and stores a value to the Collision Detection Address based on if there was or wasn't an object to collide with in the direction you are trying to go. The value of this address could be almost anything the developer wants to use. Most common is 0 (Cant Walk), and 1 (Can Walk). Although with the release of this FAQ, developers are going to get smarter and use different values.. :(But oh well, we can deal with it. Now that you know this info, on to the actual searching methods.

The Actual Searching Method!!

There are 2 ways to go about this, you can do the 'Known' value search method. Or the 'Unknown' value search method.

1) The 'Known' Value Search Method:
This method, you are assuming the value is 0 for when you cant walk, and 1 when you can walk. Now, here's the critical part. This value changes based on whether or not you are moving, or if you are not moving. So, while you character is currently walking in a certain direction. (Character HAS to be moving!) Do a search for values Equal to 1. After that has finished the game will restart, keep walking in the same direction until you run into an object/wall. While holding down the direction the object is. (character will be moving their legs, but not going anywhere) Do a search for values Equal to 0. You should have a pretty good size of possibilities. So you do some more searches. Walk to the opposite direction that you are facing. While your character is 'Walking', do a search for values Equal to 1. Walk till you run into an object/wall, and while holding the direction of the object/wall, do a search for values Equal to 0. Then walk in the opposite direction till you run into an object/wall, do a search for values Equal to 0. (Why 0, because the value should be the same as before, meaning you couldn't walk then, and you cant walk now.) Keep repeating these methods, until you get a respectable amount of possibilities to search for. Once you start going thru the possibilities, your Walk Thru Walls code, will be the address and the last 4 digits of 0001. And once you put it in shark, it should work.

2) The 'Unknown' Value Search Method:
This method is for use with games that Method 1 did not work in. This method, you don't assume what the values are, you just know that they are different based on whether you can walk, or can not walk. Same as before, do an Initial search when you are currently walking. (Character HAS to be moving!) Then when you run into an object/wall, do a different to search. Then walk in the opposite direction, do a different to search. Walk till you hit an object/wall, do another different to search. Then walk back in the opposite direction till you hit an object/wall, do an equal to search. (Because the value should be the same as before, meaning you couldn't walk then, and you cant walk now.) the previous search you took where you couldn't walk.) Keep repeating these methods, until you get a respectable amount of possibilities to search for. Once you start going thru the possibilities, your Walk Thru Walls code, will be the address and the last 4 digits will be the value the code was when you were walking. And once you put it in shark, it should work.

3D Game Specification

Due to the nature of this method, it may or may not be possible to do with a 3D game. The logic behind it all is this: In a 2D game, you can move at the most in 4 directions, up/down/left/right. In a 3D game, you can move in 360 Degrees, so the developer has to cope with this as well, and probably will use different routines. The above method should still work, provided that when you are running into an object/wall, you are running into it 'Head On'. 'Head On' means, that while holding the direction of the object/ wall, you're character doesn't turn, face, or go in any other direction. And in all reality, there's not many places you can do this. Sometimes, you can just hold 'Up' when you are directly facing a wall, and sometimes you cant. When you cant, I suggest trying a corner in a room. :)

Tips & Tricks Of The Trade

Now, there are some key tips to finding this type of code.
1) You found the code that lets you walk thru walls, but only in left/right or up/down directions!! DOH!!! Here's what you need to do. Make a Joker Code for the game, and lets just say the code you made works in the 'Right' direction. Lets say, the Joker Code for this game was a normal Joker Code and the code was.
D0123456 ????
And the Walk Thru Walls code you made only worked in the 'Left/Right' directions. If you put in these codes:
D0123456 2000
80XXXXXX XXXX << Walk Thru Walls Code You Made Goes Here.

```
    D0123456 8000
    80XXXXXX XXXX << Walk Thru Walls Code You Made Goes Here.
  It will now work in the 'Left/Right' directions, with no lock
  ups. And now, you need to find out the Walk Thru Walls code
  for the 'Up/Down' directions. So, repeat the searching method
  used to find the 'Left/Right' code, but this time, walk 'Up'
  and 'Down'.
2) You cant get it down to a lower amount of possibilities. What
  I would do, is walk in one direction till I hit an object/wall,
  and do the appropriate search, then walk in the other direction
  till I hit an object/wall and do an Equal to search. Or even do
  an Equal to Search while your character is walking in opposite
  directions. Also, another key way to reduce possibilities, is
  to do an equal to search in different parts of the map. I.E.
  If you walked 'Right' till you hit an object/wall, did the
  appropriate search. Walk up some more on the map, and walk
  'Right' till you hit an object/wall up there, and do an Equal
  to search. (This helps quite a bit usually. Try and make it not
  be part of the same object/wall though. If you are walking into
  a house, walk around in the town till you get to a different
  house, etc..)
3) The above 2 methods didn't work. This means the game probably
  changes the value based on what direction you are heading. So,
  you need to use search for it in the same direction only. Like
  so: Initial search while walking 'Right', do an Initial search.
  Walk till you hit an object/wall, while holding 'Right', do a
  Different to search. Then walk to the left, and then back to the
  right till you hit an object/wall, while holding 'Right', do an
  Equal to search. Then walk to the left, and then back to the
  right, and while walking 'Right', do a Different to search.
  Keep repeating these steps only in the 'Right' direction.
4) Last but not least, the code you found only works in Towns, or
  only works in the Overworld Map. Plain and simple almost ALL
  Walk Thru Walls codes will be like this. You just have to search
  again in the area where the code doesn't work.
```

How do you hack Speed Modifiers? (Submitted by Bleeding Gums Murphy)

```
You start by doing a time trial or something you don't have a time limit on.
1) Once you start, keep your speed at 0 and take an Unknown 16-bit search.
2) Now when it's finished, speed up a little bit.
3) Take a Greater than last search.
4) Now speed up a little more and take a greater than last search again.
5) Now speed up completely and take a greater than last.
6) Now slow down to about half speed and take a less than search.
7) Next speed up a little bit and take a greater than search.
8) Then go back to 0 speed(stop) and take a less than.

Repeat steps 2-8 until you get around 10-30 codes remaining margin.
Speed up to the fullest and press the freeze button(the button on your
gameshark itself).  Try out each code one at a time until you do not slow
down when you break.  This is the speed mod.

Typically, for most racers, a value of around the 4800's is top speed you
can control, but games like Rush 2047 isn't like that.

If one of the codes freezes the game, don't get angry, like I said before,
Speed Mods are reletively easy, you'll just have to start again.

You may also like to know that speed mods can be hacked for non-racing
games, but they don't help you as much as they should.
```

Hacking Timers (by Viper187)

```
     Timers can be the easiest thing there is to hack, or the biggest pain
in your @$$ since the school bully. It pretty much depends on the game, as
to how much trouble a timer can be, since different games store and read their
timers in different ways.

     The easiest timers are usually the ones that are just a set of digits on
the screen counting up or down. I'll give you the steps for a typical countdown
```

timer. For counting up, just reverse the searches (Less <> Greater).
This can also work for *some* clock type timers (i.e. "4:36:55"); in that case you'd most likely find one of the left two numbers in that time.

1) Get to the point in the game where the timer is running (duh).
2) Start an Unknown search (16-Bit recommended).
3) Return to the game and let the timer count down just a bit (2 seconds should be sufficient).
4) Now search Less Than.
5) Return to the game and let the timer countdown a bit more.
6) Search Less Than again.
7) If you can restart the timer somehow, do so, then search Greater Than. Otherwise just keep goin Less Than -- your call. When you get the possibilities down to a managable amount, start freezing them and see if your timer freezes. Note that you may want to write down the possibilities in case you freeze the entire game instead of just the timer. ;)
8) Did any of the possibilies work? If they didn't, or for some reason you ended up with none, try this method reversed (search Greater Than where you did Less THan before).
9) Still nothing? This is where things get intersting.

If you couldn't find the timer with the method above, then it may be a 32-Bit timer. These appear a lot on Racing games, but they are seen on other game genres as well. The only way to search for them using the in-game code generator is by using the method above with a small twist. Only do your searches in accordance with 1 part of the timer -- minutes or seconds, never milliseconds. There's a good chance you can find the timer like this. When you get the possibilies narrowed down in this case, look for ones ending in 0,4,8, or C and try them first. If one works then you found the upper 16-Bits of the timer. The part for milliseconds would be the next 16-Bit address (i.e. if 8104440C freezes the minutes/seconds, 8104440E would be for the milliseconds).

Now the easier way to find 32-Bit timers is by using the PC utils/GSCC2k2. What's the advantage? You can actually do 32-Bit searches. You'd just start a 32-Bit Unknown value search and do Greater/Less Than in accordance with the -whole timer-. By that I mean, "4:44:50" is LESS than "4:45:20" even though that last part (milliseconds) appears to be Greater.

Think you found your timer code when you stopped the numbers on the screen from counting down? In a good deal of games, this is the case, but there are times when the on screen timer and "real" timer aren't the same thing. When this happens, you basicly have to start at square one and search for the "real timer" using the any and all methods necessary. Also, don't take for granted that the "real" timer will be the same type (32-Bit/16-Bit) or even count in the same direction!! The "real" timer on Starfox 64 eluded me for years because the on screen timer was 32-Bit and counted down, while the "real" timer was only 16-Bit and counted UP! Even the most seasoned hackers need to learn new tricks once in a while, I guess.

If all else fails, you could get creative and start tracing ASM. :-P

Good luck :-)

Hacking Quickstart/Skip Intro Codes (by HyperHacker)

As with certain other code types, the search methods for these are not set in stone. They should work for a lot of games, but do'nt be suprised if you have to get inventive once in a while.

The easy way is to look for a timer (usually 16-Bit) that counts during the intro. Keep doing Less Than or Greater Than until it restarts, then do the opposite once (if you were doing Less Than do Greater Than, then Less Than again).

The hard way (that works on a lot more games, mind you) only works if the game uses the same address to tell if it's in the intro or title you need to find the byte(s) and do the same as the timer method.

Once you find the timer/flag, you could do a simple ASM trace to find where it writes the intro's value and instead have it do the title, which probably would work a lot better if you could do it.

If all else fails, and you're into ASM, you could find the intro ASM and Jump over it.

Hacking Color Modifiers (by Goldenboy)

Method 1:

Let's say you wanted to modify the crosshair color. Take a screenshot of the game with your GS. Open the pic up in Paint (or whatever) and get the value of the color of the crosshair. Now search for the value with a 32 bit search.

Method 2:

In a game that has the color palette in front of you, like No Mercy... Put your cursor on the Black color and do a 16-bit search for 0000h. Now put your cursor on the white color and search for FFFFh. Then turn it into a 3-line 24 bit code (3 8-Bit codes for Red, Green, and Blue), since most games use 24-Bits for color.

Hacking Image Modifiers (Alternate) - by Icy Guy

This method of hacking image modifiers is used when you have the quantity digits for any image modifier in the game available.

Did that make sense?

If it didn't, read on. If it did, read on anyway.

There are two ways to hack image modifiers - with Unknown Value searches and with Known Value searches. The Unknown Value method is used when the case is just that - you don't know what the quantity digits for an image modifier are. The Known Value search is used when you know the quantity digits for ANY object's image modifier and you want to hack an image modifier for something else.

The Known Value method is covered in this tutorial, since I'm not entirely sure about how to hack image modifiers with the Unknown Value method. I know it involves searching when the image changes, but I'm not up on the specifics. Plus, all of the image mods I've hacked involve the Known Value method.

For this tutorial, I'll be using Banjo-Tooie as an example and I'll be hacking an image mod for Heggy on the Isle 'O Hags.

Here we go...

-Find a list of image modifier values for any object, preferably a complete list. (You can find a complete list starting here) Look at the list until you find the value that will turn an object into Heggy, which is 914 (the list says 0914, but same thing). Punch this number (914) into a calculator capable of converting between hex and decimal (e.g. Windows Calculator, in Scientific Mode), and convert it to decimal. You'll get 2324.

-When you've loaded up 'Tooie go to the Wooded Hollow on the Isle 'O Hags. Locate Heggy's Egg Shed and march on in.

-When Heggy appears on the screen, hit the GS Button to go to the In-Game Menu. Select the Code Generator. Do a 16-bit, Known Value search for 2324 (914 in decimal).

-Check your results. One of these codes SHOULD be the right one. If not, then the game is doing something crazy, like using random addresses for images. If that's the case, I can't help you.

-In this case, the code that turns out to be the correct one is 81132F44 ????.

So, to recap:

-Get a list (preferably complete) of image mod quantity digits for any object.
Find the digit that would change the original object to the object you want
to modify. Convert this digit to decimal.

-When the object that you want to modify shows up on the screen, do a 16-bit,
Known Value search for its value in decimal.

-Check/test your results until you find a code that works.

A few parting notes...

-If you know the general location (in the game's memory) of other image
modifiers, you can use those as guides as to where you might find your new
code(s). Knowing this really comes in handy when you check your results, as
it may eliminate almost all of the guesswork involved when you're trying to
find the right code. You should test codes in that general area of memory
first - you may not need to test any others after that.

-The locations of image mods differ between games, even if the games are in
the same series or are made by the same developers. For example:
Banjo-Kazooie stores images in the 803XXXXX area (usually 8035XXXX-8039XXXX),
Perfect Dark stores them in the 801DXXXX (maybe even 801C?) area and up (in
low-res, at least), and Banjo-Tooie stores them in the 8013XXXX area. Due to
this fact, it may be harder to hack a code because the number of results is
above 100, and the area of memory that has your code is too high to be
displayed until more results are eliminated. Unless of course, you're using
an emulator or PC utilities. ;)

-Some objects just can't have their images changed (like the Flibbits in
Banjo-Kazooie). Period.

-You may need to exit and reenter a level to see a code's effect if you altered
a memory address' value in the results screen.

-Sometimes a code won't work if you try to modify a 3D object's image with a 2D
sprite. The reverse also applies.

C) Finding N64 Enable Codes

How To Hack MTC0 Enabler Codes (Used with special permission of Code Master)

Information That Is Needed To Know Before Tackling This Code

ASM - aka Assembly Language knowledge is good to know, because
these codes that are being written are in fact ASM codes.
Well, let's cut to the chase, what could Nintendo possibly be doing
to cause a problem with the Cheat Device. Not only does it disable the
Code Generator's Button to go to the In-Game Menu, but no codes will
work without an Enable Code either. This means that they are doing
something that disables how the Cheat Device works. Well, here's some
information that not many know, and I have figured this info out
over my time with hacking N64. Cheat Device Pro (Version 3.0 & Higher)
uses WatchLo & WatchHi Co-Processor break points to know when to
activate the codes that have been turned on. Well, obviously, if
they changed the WatchLo & WatchHi Co-Processor registers, it would
make Cheat Device not be able to set its own 'Break Points' to know
when to activate the codes, thus disabling the Cheat Device.

The Actual Searching Method!!
Well, obviously if you can't use the In-Game Code Generator, then
you cant dump memory to the PC, so you ask, how do I search for
this? Well, simple, you must have an Illegal Rom copy of the
cartridge :P (Well, having a 'Backup' copy of an N64 rom is not
illegal in all places if you own the original cartridge, but for
the sake of argument we'll just say it is.) In the rom file, you
need to get yourself a Dissembler, to disassemble the N64 Rom
into R4300i coding. (aka ASM) I recommend either N64PSX.EXE by
Nagr, or Niew by Titanik. Both are good, and can be used to help
find this command. Well, now what is the command that we are
looking for? Obviously, if they want to change a Co-Processor
register, you would just set the register equal to something else

right? Exactly, there is a Co-Processor opcode called 'mtc0',
which means, 'Move To Co-Processor 0'. Well, you need to find an
line that is something like, 'mtc0 $v0,$s2' or 'mtc0 $v0,$s3'.
In this opcode, the first register can be 'anything', $at, $v0,
$v1, $a0, etc.. But the 2nd register must be $s2 or $s3. Why must
it be that register? Simple, the opcode is moving a register to
the Co-Processor registers, which have different names as well.
So, in essence, $s2 is equal to WatchLo, and $s3 is equal to
WatchHi. Make sense now? They are setting WatchLo or WatchHi
with this 'mtc0 xxx,$s2/$s3' line. It also happens that the
Cheat Device needs both WatchLo & WatchHi to function properly. So
they can change either WatchLo or WatchHi to make the Cheat Device
not set its own break point to turn on codes, or check to see if
the GS Button was pressed, etc..
OK, now I know the 'mtc0 xxx,$s2/$s3' line in the ROM, how do I
make that into a Cheat Device code? Simple enough. Write down the rom
address of the line, lets say the line was 0005A278h. Now you need
to go to the rom address 00000008h in your dissembler, this is
where the game's 'Entry Point' is stored. For most games, it will
have 80000400h at this line. So, take this address you just got
from 00000008h in the rom, and add the 0005A278h to it, and your
total is 8005A678h and then subtract 1000h from it and the address
becomes 80059678h. And to make the Final Enable Code, change the
beginning of the code to F1, so you have F1059678h. And last, but
not least add 2400 to the end of that and there you have it,
F1059678 2400, your final 'Enable Code'.

Tips & Tricks Of The Trade
Now, there are some key tips to finding this type of code.
1) None really here, unless u make your own program to disassemble
the rom yourself and check for the 'mtc0 xxx,$s2/$s3' line. If you
can and do this, it becomes very simple to make an Enable Code.
There you have it!! The first EVER N64 Enable Code FAQ on how to
make them. Note that the codes will only work on Cheat Device 3.0
Or Higher because of the F1 at the beginning of the code means
that the Cheat Device activates before the game even starts to run.

An Easier Way Of Finding "MTC0" Type Enablers via Nemu 0.8
by Viper187

This can be much easier than the original method by CodeMaster, thanks to
a recently released update of Nemu 64 that includes some rather useful stuff.
To do this, as with CM's method, you need to have the ROM for the game you want
to hack the enabler for and Nemu 0.8.

1) Open Nemu and load the game (ROM) you want to hack the enabler for.
2) Once the game begins to run, and you see the intro logo crap, you can Pause
it. That should be far enough for our purposes.
3) Now, open Nemu's "Dump Memory" window. Plugins > Debugger: Dump Memory
4) Set the address range to dump. Start at 80000000 and end at 80400000. Ending
at 80800000 may be needed for hi-res games, but I doubt it.
5) Pick a filename and location for this memory dump.
6) Set the Format to "TEXT -Disassembled + PC"
7) Click "Dump" and it will create a somewhat large text file. Open this file
in the text editor of your choice (I prefer Textpad 3.2.0).
8) Upon loading the file, you'll see all kinds of ASM stuff.
9) Use the Find/Search feature of your text editor to search for "MTC0".
Now, the particular MTC0 line we're looking for is "MTC0 $xx, reserved". I
realize that CM said "MTC0 xx,$s2/$s3" above, but this is how it appears to
work with Nemu's disassembler and the actual RAM. I've looked at the enablers
for several games and each one has been the same so far, right down to the reg.
tip: You may actually just want to search for "reserved"; it might appear less
than "MTC0" itself.
10) Once you've found the MTC0 line, simply take the address beside it, for
example "80023240", and make it into your enabler. Do this by changing the
first 2 digits to F1 and using the value 2400 (hex).
11) Test the enabler on the game and start cheating! :_

Hacking Enabler codes that do not conform to the "mtc0 xxx" search
- by Parasyte

To hack most of the newer enable codes for N64, follow the ASM from the entry point and look for something that reads or writes to an address below 80000400. If you find something, KILL IT! You must do that by either nulling, that is setting the offset to zero or finding some other suitable offset. You must do this for every instance you can find. the new GS protection put into games will read/write to the following addresses -

80000080
80000100
80000180
80000190

go through the entry point and find all instances of "ori/addiu xxxx" where 'xxxx' is either '0080', '0100', '0180', or '0190'. then search above those opcodes for an "lui 8000" opcode if you find that lui, just change the 'xxxx' in the ori/addiu opcodes to '0120' and the protection will be flawed. some games will also force 8MB mode before actually checking for the expansion pak. To force 8MB mode, addresses around 80000300 will be written to. many times, you can find the ASM by searching for "addiu/ori 013x" where 'x' is a value 0-F. you may find some "sb/sh/sw 013x($yy)" opcodes. if $yy is $sp, ignore that entire opcode. Otherwise, look above that address for "lui $yy,x000" where 'x' is either '8' or 'A'.

An example of one common enabler that can be used to force either high or low resolution modes and is useful in getting a game to boot and running in either mode is as follows:
F0000319 00??
?? =
40 - low rez
80 - full hi-rez
78 - trick the game into thinking it's hi-rez, but there's actually 1/8th of the RAM free... Which the GS uses for it's trainer routines and such.

that address simply specifies the highest address the game is allowed to use. just don't go above 80.. Since that RAM is reserved and unusable, the game will crash. But just play around with the values.. you might come up with better values.

It works for every game. BUT it does NOT force screen resolution!!!!! It forces the game to use any amount of RAM you want. It does not make the games any faster, does not give it a higher screen resolution. It's mainly used to hack games that run with the expansion pak. You can use the PC Utils. Addresses 80000000 - 80780000 are available in this mode. So the codes you hack with it may not work in the full 8MB mode.

When modifying the 80000318 address, and it tells you there's no expansion pak, that generally means the game is reading that address to ensure it's value is 80800000. This may be defeated by watching reads of 80000318 using Nemu 0.8 or possibly another method of finding them.

Finding FF Enable Codes (For Hi-Res Games)
by Viper187 & Parasyte

FF enable codes tell the GS where to store your active codes. This is sometimes needed because the shark's default store location is used by a game, which causes it to crash. FF enable codes can fix this. The only catch is, they weren't supported til GS Pro version 3.3.

Ok, to do this you need a hi-res RAM dump of the game that needs the enabler. You can get this in a few ways. The easiest way I've found is Nemu 0.8. When you have the game running in it, and have started a new game and such, go to Plugins > Memory. This looks a lot like the Memory Editor in GS Pro. Now that you're able to view the RAM, either with Nemu as stated or another way, look for a large block of 00s, 256 bytes minimum. The trick here is finding a block of 00s that actually stays 00s while playing, meaning the game doesn't use that area. ever! The address where this block of 00s starts can be your FF code (i.e. FF266610 0000). You'll just have to try it and see if the game will load with that and a code or two on.

D) N64 Emulator Based Hacking - by Viper187

Hacking With Nemu

Nemu is most useful, as it has a full debugger, memory editor, etc. but it only has Known Value search and Text search, which come in handy at times. Luckily, it also has some nice RAM dump options. Now the downside is that a lot of games have problems on Nemu or just don't run. You can sometimes get stubborn games to run by trying an alternate INI file or simply changing an option somewhere. One of the dumbest little tricks I've found to getting some games running, was disabling Audio HLE in Nemu's Debug Menu. Another point of interest is the expansion pack. Some expansion pack games won't seem to run without the pack enabled, even though they do on the console, so try switching the pack on (Options > Settings). There are also other Audio/Video plugins that you can use with Nemu to get games running better. The best ones I've seen so far are Rice's Daedalous 4.6.0 (5.0.0 and 5.0.1 don't seem as good) and Jabo's D3D 1.30. To dump RAM using Nemu 0.8 or higher, go to the Plugins Menu, and locate "Dump Memory". Here you can specify how much of the RAM to dump, where to save it to, and what format. "RAW - Big Endian (N64)" is the normal N64 RAM dump. This is the format you use with Phantom's program, Cheater64, etc. Just take a few RAM dumps where you would normally do your searches with the shark, and plug them into RAM Compare to find some codes.

Hint - Goldeneye 007: I know it's one of those games that everyone just has to try on here. Well, it's really only worth fiddling with on Nemu if you want to try to ASM hack it; otherwise use Project64. The only way I've gotten it to run on Nemu is with that alternate INI file and Jabo's D3D 1.30.

Hint - Perfect Dark: Only seems to boot in hi-res. If you can get it booted, you'll see a good deal of graphics glitching and such, but with a little luck you can get into the game. I've noticed it freezes a lot at the Start Mission screen; getting into Combat Sim was a little easier though.

Hint - Jeopardy: Audio HLE might need to be off (I forget), but once you load the game it may seem like your controller doesn't work. Turn off Controllers 2-4 before loading and it should be fine. :)

Hint - Turok 2: Runs pretty good, but don't press the L button on your controller because it'll cause the game to screw up.

Hint - Turok Rage Wars: I know I had this thing run before, but I haven't been able to get it running since then. If anyone figure out why it keeps crashing Nemu on me, let me know.

Hacking With Project 64

Project 64 is about the best N64 emulator there is, at least for playing games. I've found very few games that won't run on this. "Where's the RAM dump thing??" There isn't an actual RAM dumper, memory editor, or code generator. Now here's the good news. The Save States are basicly RAM dumps when uncompressed; PJ64 zips them by default, but you can turn that off to make things easier. Now a .PJ file is a RAM dump that has a header and footer, and each 32-Bit value is reversed. Use Phantom's Byteswap program to take care of this. Then you can either plug your RAM dumps into the comparer program/hex editor of your choice and go to work.

PJ64 Save States - Specific differences from a RAM dump:
1. The First 75C of the file is an extra header. Beyond that is the RAM.
2. Byteswapping: Every 32-bits word in a PJ file is reversed.
i.e. 80012344: 78 56 34 12 is really 80012344: 12 34 56 78.
3. The last 2000 (hex) of the file is a footer.

Phantom's Comparer

Phantom's little RAM Compare program has the basic search options, plus it can search for Floating Point values, to an extent. It only supports normal RAM dumps. To use with PJ64 save states, you'll need Phantom's Byteswap Program.

Cheater64

Cheater64 is Viper's newest toy. It has all the usual options, with much needed improvements. It accepts RAM dumps and PJ64 save states (no need to fix them).

VTR Compare

VTR Compare is the mother of all search programs, as far as options go, but it lacks horribly in speed. When dumping RAM from Nemu, you need to select "TEXT - 32 Bit HEX Data" (or 16-Bit) to use it with VTR Compare. Once you've got a few dumps, crack open VTR Compare and go to work. It should be pretty straight forward, but don't forget to check the ReadMe.

E) N64 Assembly ("ASM") Hacking - by Viper187

Covered In This Section:

All my ASM knowledge came from Parasyte, The Phantom, CodeMaster, and just playing around with it. I'd like to thank Parasyte in particular for putting up with all the questions I'm always asking him. :-)

Introduction

This guide was written to teach you some basic and intermediate ASM hacking. This is only a taste of what you can do. ASM, which is short for Assembly Language, is a low-level programming language. The standard codes we all know how to hack are merely RAM addresses that hold a value. You'll soon see that you can do more than just freeze those values.

It may seem like there's a lot of information here. I try to make it as easy to understand as I can, but I, like Parasyte have been doing this stuff for so long that it becomes a little hard to write for total beginners. It should be evident that this isn't necessarily a beginners kind of hacking in the first place. Don't be overwhelmed; learn this at your own pace.

The best piece of advice I can probably give you is to get to try setting Breakpoints, and look at the ASM before you just NOP something for Infinite Somethin. Get a feel for how it actually works and what you are doing by NOPing it. See what other ways you could go about getting the same effect.

You can also take apart existing ASM codes that were hacked by people like Parasyte and myself. By dumping the RAM of a game, opening it in Niew, and setting the values of some ASM codes that are laying around, you can see what they change and what that does.

If there's something in here you still don't get, or are having trouble getting the effect you want, just can't think of how you'd even begin to do a particular code (i.e. Gravity), etc. you can post on the messageboards here

at GSCentral and one of the more experienced ASM hackers should be able to offer some help. One of the reasons I say this is that not all games are the same, different programmers do things different ways; It's more a matter of learning the basic coding idea behind what is being done, so you'll be able to hack the same effects for different games.

Assumptions

- You have GSCC2k2 installed on your computer and you know how to use it with a Cheat Device (v3.2 or Higher), OR you're somewhat familar with Nemu 64 0.8 if you want to go that route.
- Obviously, you need to have a DB25 male to female parallel cable to hook the Cheat Device to your PC as well.
- You should have at least some knowledge of Hex, and a little Memory Editor experience wouldn't hurt.
- To really work with N64 ASM, you'll also need Niew, N64 ASM (R4300i) opcode documentation, and LemAsm.

Basic Info

How does this "ASM" hacking work?

You can't just search for ASM codes like normal codes, per se. For most applications, with the exception of enablers and possibly other special circumstances, you need a normal code to hack an ASM code. What we do is set a Breakpoint on a code (address) to find the ASM routine that reads/writes it.

What are Breakpoints?

Breakpoints are used to find the ASM that reads/writes to a specific address. When you set a BP, you're causing the debugger to watch the address that you're setting the BP on. When the address is read/written to, the debugger breaks (halts).

What good are they?

You can do almost anything with breakpoints.
Some of the more pratical uses are:

- Getting more of something/causing double damage
- Mega Jump
- Hit Anywhere
- Walk Through Walls
- Killing timers
- Infinite Health, ammo, etc
- Gain ammo when shooting

Are there different types of Breakpoints?

Yes, there are 3 types:

- Read (BPR) - This occurs when ASM reads the value of the address the debugger is watching. ASM will almost always read the value of your ammo and such in order to reduce it when you shoot.
- Write (BPW) - This is the most common. This occurs when ASM writes a value to the address the debugger is watching. We'll be using this primarily.
- Execute (BPX) - This one is a little different, and we don't use it that often. In this case, the debugger is watching a part of the ASM to see when it's executed. This is usually used to find more advanced codes.

Setting Breakpoints

Ok, before you can hack an ASM code, you need to know how to set a BP. If you're using GSCC2k2:

- Open GSCC2k2
- Go to the RAM Editor
- Click "Set BP"
- In the box that pops up, put in the address to BP on, e.g. 800ACE01. You only need to enter the address, not the value. Also note, the addres you enter must begin with '80' and not '81', '88', etc. Just change the address to '80' when you type it in.
- Choose the type of BP. In most cases, you'll only want "Write" checked, but there are cases when you'll use "Read" or both.
- Decide if you want the game to auto resume after the BP occurs. Usually, you can go ahead and auto resume, but sometimes you'll want to leave it so you can resume manually with Shift + F9. Also note that some games won't resume at all. You should be used to this. It happens when hacking normal codes too.
- Now when you click "Set BP" on this pop up, it goes away and you get a new pop up. It asks you to click Yes when the game halts.
- Next, cause the address you set your BP on to be read/written to (e.g. for ammo, you'd shoot a gun). The game should freeze and you can click Yes now.
- When you click Yes, a Notepad window will open. This gives you valuable info about the asm that's reading/writing that address.

If using Nemu64 0.8:

- Open the Memory editor (Plugins > Memory)
- Go to the address you want to BP on
- Decide if you want BPR or BPW ("Watch Type")
- Rightclick the value of the address you want a BP on. It will be highlighted a different color depending on the Watch Type. You'll notice that it's going to watch all 32bits.
- Next, cause the address to set the BP on to change (e.g. for ammo, you'd shoot). The game should freeze and the Commands window with come up (if it's not already open). The highlighted address is the ASM opcode that wrote to the address you're watching. open Registers (Plugins > Registers) to see all the values of the General Purpose regs (like the Notepad in GSCC2k2 shows you). If you've done this on GSCC2k2 before, you'll notice something cool about Nemu's Registers window... It has the COP1 regs! We'll get into those a little later.

Setting Breakpoints On Execute (BPX)

Nemu64 0.8: It's easy to set a BPX on Nemu. Just click beside the address of the opcode you want to BPX on in the Commands window, and a checkmark will appear beside it. BPX set :)

GSCC2k2: Now GSCC2k2, on the other hand, doesn't have a BPX option. But, I found my own way to do it. Originally, Parasyte/CodeMaster came of with a long routine that did it. My way is easier than that, but requires a little opcode knowledge. Take the address you want a BPX and check it and the opcode that follows it to:

```
LUI $K1, 8000
SW  $K1,0060($K1)
```

Now set a BPW on 80000060 and you have your BPX. What those 2 opcodes do is set $K1 to 80000000, then store $K1 to 80000060 just to cause your BPW to occur. $K1 isn't used by the game after boot up, so we don't have to worry about changing it.

What if the game halts when it shouldn't with GSCC2k2?

There are some cases where the BP will occur prematurely. If this happens, look at the Notepad window and get the value of "pc", it'll be the address of the ASM that read/wrote (e.g. 80021530). Back in GSCC2k2, you'll notice that your position in the ram editor has changed to the address of the ASM. Click in there on the address of the ASM (80021530 in this case) and change it to 2400 (Yes, that's 16-bit). Set your BP again (same address as the first time) and get the game to halt again. If it's premature, repeat the previous steps

till you get it to break at the right time. Note that this is where turning auto resume off comes in handy. You can leave the game halted, set the premature BP to 2400, then when you set your BP again to try for the right one, it will resume and wait for the next break.

What if the game halts when it shouldn't with Nemu64?

As I mentioned above, there are cases where there is a constant read/write on the address you're working with. With Nemu, you can try to just resume ("GO") from the commands window, or rightclick the opcode in the commands window and "--> Memory Debugger" to bring the Memory Editor back up and bring it to that address. Then you can NOP it by setting it to 0, and go back to the Commands window to resume ("GO").

TLB Mapping

TLB mapping is pretty evil in the eyes of ASM hackers. This is the reason you don't see 1337 ASM codes for Goldeneye and very few for Perfect Dark. It's like some kind of address aliasing. When you set a breakpoint on Goldeneye, for example, the PC address returned will be something other than the real PC address; in Goldeneye's case you'll get an address beginning with something like '7F' instead of '80'. There's no known way to get around it on the console, but you can with Nemu 0.8. If you set a Breakpoint on a TLB mapped game in Nemu, you'll see all the opcodes in the TLB addresses. The trick is finding the real ones. Rightclick the break address and choose the --> Memory Debugger. This will bring up the memory editor, but notice the addresses shown are the same address you saw in the debugger (TLB mapped). Look at the 32-Bit value of the first (break) address there. Bring up the Search window and do a 32-Bit search for that value. If you got more than 1 result, you'll have to figure out which one it is by comparing the opcodes before/after the one you searched for. There you have it, you got around one of the most evil programming tricks in the book. :)

An In-Depth Look at the Translation Look-aside Buffer (TLB) - by Parasyte

Formats

EntryLo0 & EntryLo1:
00 pppppppppppppppppppppppppp ccc d v g
(32-bit: 2.24.3.1.1.1)

p = Page frame number; the upper bits of the physical address.
c = Specifies the TLB page coherency attribute. (See below)
d = Dirty. If this bit is set, the page is marked as dirty and, therefore, writable This bit is actually a write-protect bit that software can use to prevent alteration of data.
v = Valid. If this bit is set, it indicated that the TLB entry is valid; otherwise, a TLBL or TLBS miss occurs.
g = Global. If this bit is set in both Lo0 and Lo1, then the process ignores the ASID during TLB lookup.

PageMask:
0000000 mmmmmmmmmmmm 0000000000000
(32-bit: 7.12.13)

m = Page comparison mask

EntryHi:
vvvvvvvvvvvvvvvvvv 00000 aaaaaaaa
(32-bit: 19.5.8)

v = Virtual page number divided by 2 (maps to two pages)
a = Address space ID field. (ASID) An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers.

Notes On TLB Page Coherency (c bits)

The TLB page coherency attribute (C) bits specify whether references to the

page should be cached; if cached, the algorithm selects between several coherency attributes. The table below shows the coherency attributes selected by the C bits.

TLB Page Coherency (C) Bit Values
0: Reserved
1: Reserved
2: Uncached
3: Cacheable noncoherent (noncoherent)
4: Cacheable coherent exclusive (exclusive)
5: Cacheable coherent exclusive on write (sharable)
6: Cacheable coherent update on write (update)
7: Reserved

Translating Virtual Addresses To Physical Addresses

In order to translate virtual to physical addresses, you must be able to read each TLB entry, looking for a virtual address hit.

Start by dumping the TLB entries one-by-one. First, load the CP0 Index register with the TLB entry index you wish to dump. You should start with 0x00, and work your way up to 0x2F. That gives 48 total TLB entries. After loading CP0 Index with the entry number, call a TLBR instruction. This will dump the TLB entry to the EntryLo0, EntryLo1, PageMask, and EntryHi registers in the formats described above.

After the TLB entry has been dumped, begin decoding the data in each register.

To decode a TLB entry, start with the p bits from the PageMask register. This value tells the size of a virtual page for the TLB entry. The page size can be determined by checking the values against the following table.

0x0000: 4KB (0x1000 bytes)
0x0003: 16KB (0x4000 bytes)
0x000F: 64KB (0x10000 bytes)
0x003F: 256KB (0x40000 bytes)
0x00FF: 1MB (0x100000 bytes)
0x03FF: 4MB (0x400000 bytes)
0x0FFF: 16MB (0x1000000 bytes)

If the value is anything other than listed, the page size is undefined\erroneous. Here is a bit of C which will allow you to grab the page size and a page mask:

```
u32 mask = (PageMask >> 1) | 0x0FFF;
u32 pagesize = mask+1;
```

Next, extract the virtual page number from EntryHi, based on pagesize.

```
u16 tmp = pagesize >> 12;
u32 vpn = (((EntryHi >> 13) / tmp) * tmp) << 13;
```

Then use the virtual page number as a mask to the virtual address. If the result is the same as the vpn, it's counted as a hit. If not, restart the process with the next TLB index.

```
if ((vaddr & vpn) != vpn) continue;
```

If it's a hit, find out if the virtual address is on an odd or even page, based on the TLB page size.

```
u32 odd = vaddr & pagesize;
```

Now perform a logical AND on EntryLo0 or EntryLo1, depending on the 'odd' variable. If the result is 0, the page frame is invalid. If successful, extract the page frame number. If not, restart the process with the next TLB index.

```
u32 pfn;
if (!odd) {
  if (!(EntryLo0 & 0x02)) continue;
  pfn = (EntryLo0 >> 6) & 0x00FFFFFF;
}
else {
  if (!(EntryLo1 & 0x02)) continue;
  pfn = (EntryLo1 >> 6) & 0x00FFFFFF;
}
```

Finally, if it gets this far, mask the virtual address with the 'mask' variable created earlier. Then attach the physical page number, and or with 0x80000000 to complete the process. Lastly, exit the loop. Checking the remaining entries for a hit is not required.

```
u32 paddr = (0x80000000 | (pfn * pagesize) | (vaddr & mask));
break;
```

And there you have it! Your paddr variable will be a pointer to the translated address.

Regs and Opcodes

What are Registers?

Ok, you set a BP, you have that Notepad window (or you're looking at the Registers Window in Nemu), and I bet you're wondering what that stuff means. The Notepad doc is called a reg dump. It lists the N64 registers and their current value. Registers, or regs as we call them, are variables. ASM uses these variables to carry out various operations. Here's a sample reg dump...

```
r0 - 00000000   at - 41A00000   v0 - 00000008   v1 - 80110728
a0 - 00000068   a1 - 00000004   a2 - 00000000   a3 - 00006D60
t0 - FFFFFFFF   t1 - 00000004   t2 - 00000006   t3 - FFFFFFFF
t4 - 00000008   t5 - FFFFFFE0   t6 - 00680000   t7 - 8019E0F4
s0 - 801A7190   s1 - 00000000   s2 - 80112148   s3 - 80199C0C
s4 - 800DC9E4   s5 - 00000000   s6 - 00000000   s7 - 00000000
t8 - 00040000   t9 - FFFFFFFE   k0 - 80794AEC   k1 - 00000AAA
gp - 00000000   sp - 80114808   fp - 00000000   ra - 80073930
pc - 80073948
```

Each of those regs holds a 32-Bit value. For example, v0 holds the value 00000008. These are referenced in ASM by a dollar sign (e.g. $v0).

Notes:

1. $pc is for our purposes ONLY (it's not an actual reg that you can use in your routines). It holds the address of the ASM that read/wrote to the address you set the Breakpoint on.

2. $r0 is also known as $zero. It's ALWAYS zero no matter what you try to change it to.

3. $k1 is always 00000AAA, but it doesn't have to be. Games doesn't use it after boot up, so it's a free reg for us to use.

4. $sp is the Stack Pointer. The games uses the address in this reg to store regs temporarily while doing things. You'll rarely, if ever, want to change this.

5. Stay away from $gp,$k0, and $fp. Those are used for special things and

changing them usually isn't good.

What are COP1 Registers?

The COP1 registers, also known as Floating Point (FP) registers, are used for floating point calculations. These regs are referenced as F0, F1, etc up to F31. No Dollar signs with them, and Niew doesn't show them, as it doesn't support COP1 opcodes. Some examples of these would be Size mods, Coordinates, some Timers, some Health & Ammo, etc. These FP regs can only be viewed when using Nemu. If you need to know what they are when hacking GSCC2k2, the only thing you could really do is write a long ASM routine to store the values of the FP regs to some emtpy RAM. FP numbers in RAM are stored as 32bit values. If you've ever played with a Bond game, you've seen them. Bond's health (3F80/42C8) was an FP. You'll notice in the Bond case, that your full health is 3F80 0000, but obviosly you only need to freeze the upper half (3F80) to get Infinite Health. What exactly is 3F80? 3F800000 is actually 1.0 and 42C80000 is 100.0. How can you calculate those? Well, I'd really like to know the actual math operations to convert from hex to float, but since I don't, I use FloatConvert. I didn't write that program. It's just something that Parasyte found floating around the net.

What are OpCodes?

Operation Codes, or OpCodes as we call them, are the actual lines of program code within the RAM. ALL OpCodes are 32 bit, but not all ASM codes are 32 bit. ASM codes can be anything from a single 8 bit code, to as many 16 bit codes as the Cheat Device/Emulator will let you enter. The Opcodes are 32 bit, as I said, so to change one OpCode completely you need two 16 bit codes. This is why some ASM codes are so long; it takes 2 lines to change a whole opcode.

What are COP1 OpCodes/Instructions?

COP1 (Co-Processor 1) handles all the Floating Point calculations. It has Opcodes/Instructions like SWC1, ADD.S, etc. You'll see these instructions in Nemu's Commands window, but Niew doesn't support them. That's what LemAsm is for. We'll get in to this in a little more depth later...

What is a NOP?

A NOP (No Operation) OpCode does absolutely nothing. Yes, I said NOTHING. Then why is it so damn useful? Simple. When you want to stop any OpCode from executing, like to stop the game from taking away health when you get hit, you change that OpCode to a NOP. There are 2 basic ways to NOP an OpCode:

- Long NOP - This is a 32 bit NOP, meaning it takes two 16 bit codes to use. A long NOP is (you guessed it) zero. For Example, to NOP the OpCode at 80023154, you would enter 81023154 0000 and 81023156 0000 into your Cheat Device.
- Short NOP - This is a 16 bit NOP, so it's a single code. This is what you will want to use most times to keep your codes shorter. The value of a Short NOP is 2400, so to NOP that OpCode at 80023154 this way use 81023154 2400. The reason Short NOPs work is because they're assembled as ADDIU $zero,$zero,xxxx. That's why only 1 line is needed. The other 16 bits don't matter since $zero is always 0 no matter what you try to do to it. hehe

What are Routines?

An ASM routine is just a specific piece of code that does something. It can be any length and do pretty much anything. An example of a routine would be a piece of code that is executed when you fire a weapon. It would read your current ammo, subtract 1 from it, and store the new value.

What are Jumps and Branches?

Jumps are what ASM uses to get from one piece of code (routine) to another routine. Branches have the same affect, but they are conditional jumps. This means they only jump when something is true/false (like an IF in C++/VB6). Both of these opcodes have what is called a Delay Slot. This means the opcode directly following it is to be executed before the jump.

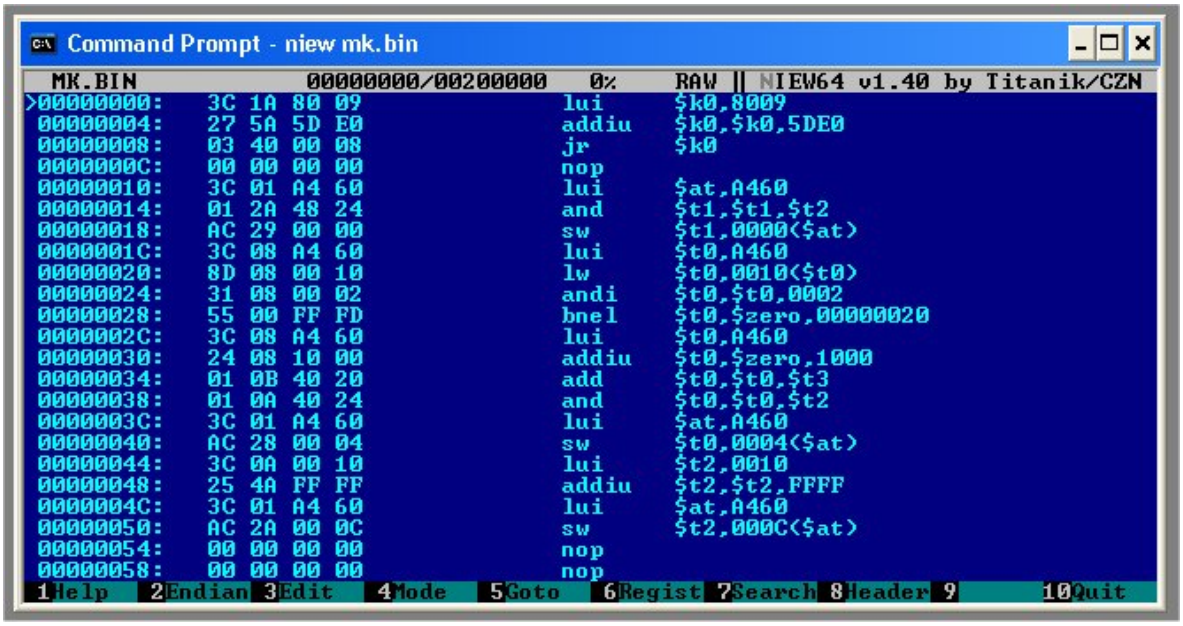
Bytes, Halfwords, Words, and Dwords

Opcodes read & write values to the ram in different sizes...

Byte: 8-Bit (00 - FF)
Halfword: 16-Bit (0000 - FFFF)
Word: 32-Bit (00000000 - FFFFFFFF)
Doubleword (Dword): 64-Bit (0000000000000000 - FFFFFFFFFFFFFFFF)

Using Niew

To use Niew, begin by using GSCC2k2/Nemu to get a ram dump of the game you're gonna edit the ASM of. Dump at least 800 - 802 (GSCC2k2 default), or dump all 4 megs (800 - 804) if need be. Now put that ram dump in the same folder as Niew and open a dos window. browse to the directory where niew is located and type "niew filename.bin" (filename is whatever you named the ram dump). The program will open and load that ram dump, but it'll look all garbled. Now press F4 to switch to ASM view. It'll look like this:



- The first column is the address (duh), the second is the value (32 bit), and the rest is the ASM.
- The ">" on the far left side indicates your current location.
- Press F5 and you'll be able to to type in the address column there. Put in the address of the ASM you want to edit, but make the first 2 digits of the address 00 instead of 80, then hit Enter.
- scroll up and down with the arrow keyes to look at all that ASM in that area.
- Press F3 once and the cursor will move to the value of the specified line. If you want to see what someone else's ASM code does, you'd type the value into the proper addresse(es).
- To edit the ASM, press F3 a second time. This will put the cursor in the ASM column. Edit the opcode or type in a new one, then hit Enter to go to the next line.
- You'll notice when you press enter, that the value of that address changes to the value repesenting the edited opcode. This is the value you'll need to enter into your cheat device.

Using LemAsm

First off, note that this could be considered a more advanced topic. You *should* familarize yourself with the basic ASM opcodes and Niew first, but it's up to you. LemAsm, thankfully, is a Windows program instead of a DOS app. So just open the RAM dump you're gonna work with use File - Open (duh). Now, you'll need to change to MIPs disassembly mode with Edit Mode and Show Reg Names. All 3 of these are in the View Menu, or you can hit F6,F3,F7 in any order. Now just type in the COP1 instruction you want to assemble and hit enter (like Niew). Now, one note with this program. I've noticed a bug that occurs

sometimes when I try to assemble ADD/SUB instructions. Hit enter and they may become DIV. You'll notice that the hex is something like 4406 1003. that 03 is typicly 3 for DIV, 2, for MUL, 1 for SUB, and 0 for ADD. Just plug the value into Nemu's Memory Editor and look at it in the commands window to see if you've got it right.

Getting Started

Think you're ready to hack your first ASM code? Well, let's do it. I've always found it easiest to learn by example, so I'll show you how to hack a code for Infite health, ammo, time, etc.

I'm going to use Infinite Ice for Mortal Kombat Mythologies, Sub-Zero as our example. Have you ever tried to hack a simple Infinite Health code or something of that type and found that it only works for that area of the game or is just completely random? This is one of those games. The RAM address of your health, ice, etc changes when you get to a checkpoint, get killed, etc. Despite moving around randomly, it always starts in the same place. When you boot up MK Myth and start a new game, your Ice is located at 800F0217. So we set a BPW on 800F0217, throw ice, and the game halts. Click YES for the reg dump to open. This is what we got:

```
r0 - 00000000   at - 80110000   v0 - 00000086   v1 - 800EFBC0
a0 - 800EFBC0   a1 - 0000FFFF   a2 - 800EFBC0   a3 - 0000009E
t0 - 00000000   t1 - 00000001   t2 - FFFFFFFF   t3 - 00000004
t4 - FFFF0000   t5 - 00000002   t6 - 00000009   t7 - 000000C5
s0 - 8010BF50   s1 - 8010BF50   s2 - 8010BF50   s3 - 0000001A
s4 - 0000001B   s5 - 00000000   s6 - 00000103   s7 - FFFF8000
t8 - 00000001   t9 - 00000001   k0 - 80794AEC   k1 - 00000AAA
gp - 00000000   sp - 800F0138   fp - 800F01E0   ra - 8004ABB8
pc - 8004AC78
```

The address of the ASM that wrote to 800F0217 is the address in $pc, 8004AC78. Now, NOP that address via GSCC2k2's RAM editor to see if it gives us Infinite Ice. If it doesn't, we need to set the BP on 800F0217 again, cause a break (throw ice, in this case), get the new regs, and NOP the new $pc. Repeat that til you find the one that works. The first one worked here, though.

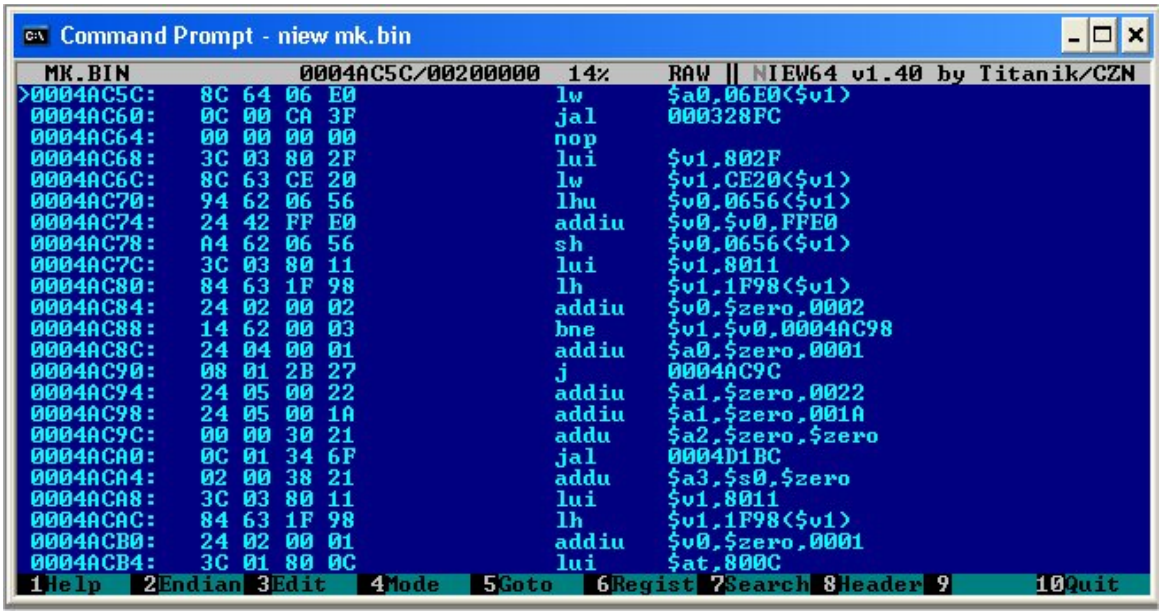
Working With ASM OPs

This is where it gets interesting. You know that ASM opcode we NOPed above to make Infinite Ice? Well, if you think that's all you can do with ASM, think again. To really get into ASM hacking, you'll need to download Niew. It's a small dos program that's used to view N64 ASM. See the Using Niew section for instruction on using it.

For this example, I'm going to use the Ice in MK Myth again. We've got our regs from the previous example:

```
r0 - 00000000   at - 80110000   v0 - 00000086   v1 - 800EFBC0
a0 - 800EFBC0   a1 - 0000FFFF   a2 - 800EFBC0   a3 - 0000009E
t0 - 00000000   t1 - 00000001   t2 - FFFFFFFF   t3 - 00000004
t4 - FFFF0000   t5 - 00000002   t6 - 00000009   t7 - 000000C5
s0 - 8010BF50   s1 - 8010BF50   s2 - 8010BF50   s3 - 0000001A
s4 - 0000001B   s5 - 00000000   s6 - 00000103   s7 - FFFF8000
t8 - 00000001   t9 - 00000001   k0 - 80794AEC   k1 - 00000AAA
gp - 00000000   sp - 800F0138   fp - 800F01E0   ra - 8004ABB8
pc - 8004AC78
```

Now that we have our reg dump, we need to use GSCC2k2/Nemu to get a RAM Dump. After you dump the RAM to a file, open it with Niew. Now press F4 to switch to ASM view and press F5 to enter the BP address (8004AC78), but start it with 0 instesd of 8. Scroll up a few lines so you can see what happens before and after your ice is written to. Here's what it looks like:



Now the part we need to look at in this case is 0004AC68 - 0004AC78. I'll explain what each opcode does...

lui $v1,802F

lw $v1,CE20($v1)

lhu $v0,0656($v1)

addiu $v0,$v0,FFE0

sh $v0,0656($v1)

lui $v1,802F	LUI - Load Upper Immediate: This puts the hex value (immediate) into the left (upper) half of the reg specified. Meaning $v1 now equals 802F0000.
lw $v1,CE20($v1)	LW - Load Word: This loads the word (32-bit) value of the reg in () + the hex offset into the reg on the left. The hex offset is signed though; the means if the offset is higher than 7FFF, the upper 16 bits are made FFFF. So, CE20 is actually FFFFCE20. The 32-bit value at 802ECE20 is "800EFBC0", so $v1 is 800EFBC0 now. Anyone familar with pointers? This is what they're for. hehe
lhu $v0,0656($v1)	LHU - Load Halfword Unsigned: This loads the halfword (16-bit) value from $v1 + 0656 hex (signed offset) into $v0. Now $v0 is the amount of Ice you currently have.
addiu $v0,$v0,FFE0	ADDIU - Add Immediate Unsigned Word: This adds the 2nd reg listed + the hex value and puts that value into the 1st reg, so $v0 = $v0 + FFE0. Now you're gonna say, "how does that decrease my ice?" Well, let's say your Ice, $v0, is 00000080. Now add FFE0 to it and you get 00010060.

sh $v0,0656($v1)	Store Halfword: Same as when you loaded this halfword value 2 opcodes above. Now you're just storing it. So how does it decrease your ice? It's only storing the halfword value in $v0 to the RAM. That means it's only storing the 16 bits (4 digits) on the right, so it's storing 0060. Now your ice was decreased by 20 (hex).

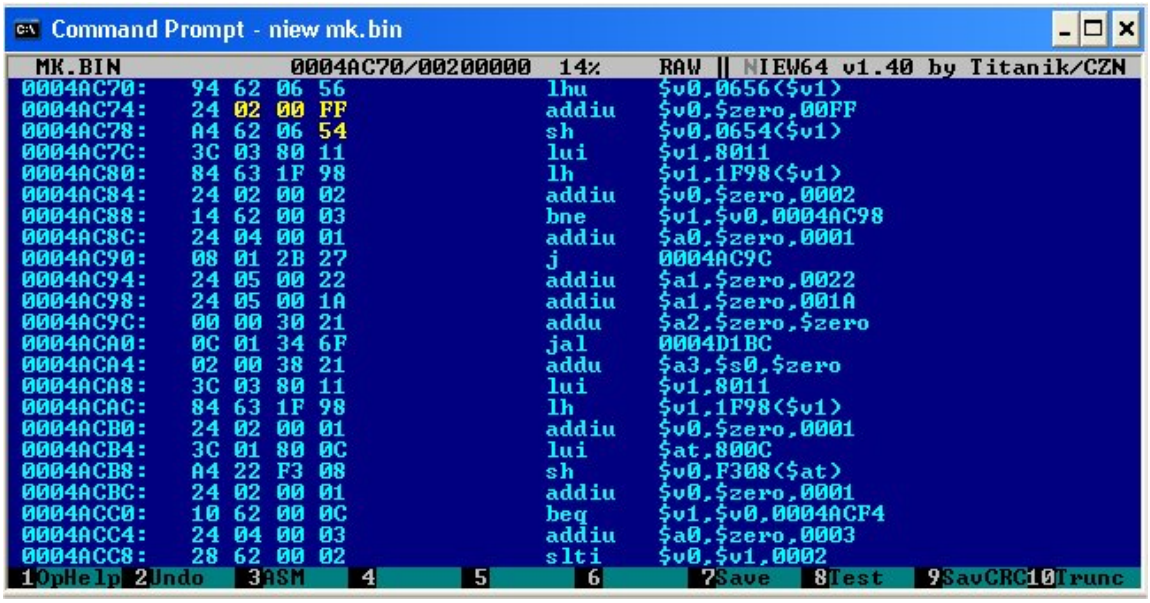
Pretty cool huh? Info about all the opcodes and what they do can be found in the R4300i Opcode Documentation (you did download it, right???). You can use this ASM to get other effects besides infinite ice. Obviously, you can change the FFE0 so you'll lose less ice, more ice, no ice at all, or even make it increase your ice. BUT, why stop at Ice? You can make this ASM routine do just about anything. I'll show you something relatively easy and useful you can do with this...

Ok, you remember that your ice is stored at 800F0217 initially. Well, your health is always 2 below your ice (it's stored at 800F0215 initially). So by making a small alteration to this ASM, you can give yourself Infinite Ice and cause your health to be refilled when you throw ice. We only need to change 2 opcodes for this.

We change:
addiu $v0,$v0,FFE0
sh $v0,0656($v1)

to:
addiu $v0,$zero,00FF
sh $v0,0654($v1)

Now $v0 = $zero + 00FF, so $v0 is 000000FF. $zero is just that. It's ALWAYS 0. You'll notice we barely changed the 2nd line. Health is stored 2 below ice, so we changed 0656 to 0654. Now what's the code to use for this effect? That's what the 2nd comlumn in Niew tells you. When you change each of those, the values in the 2nd column change accordingly. The values that are different are highlighted since they may or may not all change. Here's what it looks like:



Ok, we can see the a few values changed. The code would be:
8004AC75 0002
8104AC76 00FF
8004AC7B 0054

That's your code. You changed the ASM so it makes your health 00FF everytime

you use ice, and your ice never goes down because the asm that was decreasing it is changed to refill your health instead. We can shorten this code to 2 lines if we want to.
8104AC74 2400
8004AC7B 0054

The first line kills that ADDIU opcode and the 2nd makes it store the value of $v0 to your health instead of ice, so basicly, it copies the amount of ice you have into your health.

Hacking Some Various Types Of Codes Using ASM

I showed above you how to make Infinite Ammo, Health, etc and Modify you ammo consumption & such all in pretty much the same way there. Now I'll show you how to hack some other types of ASM codes. I'll try to start out with easier codes and progress to more advanced stuff. Always remember, "there's more than one way to skin a cat." The examples I'm showing you are never the only way of doing something, and they're by no means the only types of things you can do. The only limits in ASM hacking are the hacker's mind.

Infinite Ammo For All Guns

There are times when the method I stated above won't affect all your guns. Duke Nukem 64 is an example of this. Each gun has its own routine that decrements your ammo when you shoot. Think the only way to do it is a code for each gun? Nah. Let's make 1 code do it all.

Set a BPR on the ammo for whatever gun you have currently equipped (Pistol in my case - 802A5A00). The game should halt immediately.

Breakpoint/PC Address: 8006F8E0.

Opcode: LH $A2,59FE($A2)

What it does:
Loads Halfword value $A2 from $A2 + 59FE. $A2 is 802A0002.
(802A0002 + 59FE = 802A5A00) In this case, the reg that's holding the address we're going to need to write to is overwritten by the load opcode. This isn't a problem; it's actually useful in this case. That initial address in $A2 changes depending on what gun you have equipped so it can read the ammo you have for whatever gun is equipped. It's actually using this to display your ammo on the screen. Do we care? No, so let's change it and get Infinite Ammo for all weapons. :)

Since it's a Load Halfword (LH), we want it to Store Halfward (SH) instead. Sounds good, right? But what do we store? If you can find a reg that always has a useful value in it (like something between 0001 and FFFF) you can use it; otherwise you'd have to go through setting the value yourself. For now, we'll do this the easy way. Remember $K1 is always 0AAA? Well, we'll store it and always have AAA (2730) bullets for all guns.

Our new Opcode is: SH $K1,59FE($A2)
When you enter that into Niew, you'll notice only 1 16-Bit address changed, 8106F8E0. The value is now A4DB. There's your code :)

Modify Initial Stats When Starting A New Game

This type of code is reminiscent of the Game Genie® era. "Start With X Lives" was something that was seen for pretty much every game, but we've never seen that kinda thing on N64, have we? Well, we can now. I'll use Mario 64 as an example this time.

At the title screen/menu, set a BPW on Mario's lives (8033B21D). Now Start a new game. The break will occur at 8025500C, but this stores $zero to the address, so NOP it and try again. Now you'll get the right one.

Breakpoint/PC Address: 8025501C

Opcode: SB $T2,00AD($T3)

What It Does:
Stores Byte value $T2 to the address $t3 + 00AD.

Now we gotta make it store what we want to there. Start looking at the opcodes that come before it and find what sets $t2. Luckily, this one is actually the opcode right before it. 80255018: ADDIU $t2,$zero,0005. This is pretty simple. Just change the 0005 to whatever you want, so your code would be 8125501A ????.

Lets say, for the sake of argument, that you found a Load instead of an Add opcode there. You'd have 2 options.

1. Change the op to ADDIU $T2,$zero,????
2. Go to the address the Load op is loading $T2 from and change it there.

"But what if there's no load? I wanna modify something that you start with 0 of, and the opcode just stores $zero." This can be a bit trickier. You can use $K1 and have AAA of that item if you want -- the easy way out. Or you can find a place before the Store opcode to put an 'ADDIU $K1,$zero,????'. If you can replace an exiting op without it causing any problems, or find an empty space (NOP) somewhere there, you're in good shape. Otherwise, you'd need to Jump out of that routine to an empty area, set the reg, then Jump back. This may sound a bit complicated, but it's really quite easy once you get the hang of it. See Routines & Jumps for more details.

All Scores/Damage Counts For x Player/Team

This is easiest on sports type games (i.e. Gretzky Hocky '98); however it is possible to make player 2/CPU beat themself on a fighting game and is rather fun to watch. You simply set s BPW on the score, for example. Then it'll break when the game adds to your score. On Gretzky Hockey (Pal), you'll get the shots first, since it's watching that too (because the BPW covers more than 16-Bits). NOP that and get actually score to get it to break again. This will be the score asm. Gretzky's is at 8005ABCC.

opcode: SH $T8,3440($A2)
What you need to is look through the opcodes that come before it and find a suitable one to replace, preferably the opcode that sets $A2 in the first place (or a nice NOP to make use of). In this case we find a NOP at 8005ABA0. Now let's say we want i tto always add to team 1's score. That's 800EA870. Split that into 2 16-Bit halves, 800E and A870. Now because ASM uses signed values we need to take in account the value of the 2nd half of the address. If it's more than 7FFF, which it is, we need to add 1 to the 1st half. So 800E becomes 800F. Now we use 'LUI $A2,800F' in that NOP and change both the LH and the SH opcodes there to add A870 instead of 3440.

Before:
8005ABA0: NOP
8005ABAC: LH $T7,3440($A2)
8005ABCC SH $T8,3440($A2)

After:
8005ABA0: LUI $A2,800F
8005ABAC: LH $T7,A870($A2)
8005ABCC: SH $T8,A870($A2)

So our code is:
8105ABA0 3C06
8105ABA2 800F
8105ABAE A870
8105ABCE A870

You can do that same thing on fighting games, but P2/CPU won't actually beat themself up, so it's less entertaining. I've never actually accomplished this myself, but the way it's done starts out the same way as this, but then requires tracing through the ASM to find where it determines what player is actually being hit. This has been done for Mortal Kombat Trilogy if you'd like an example code to look at on your own.

Invcincibility

Using an Infinite Health code but tired of getting banged around? Here's your solution. I'll use Turok Dinosaur Hunter as an example on this one. If you know me, you had to know that'd be coming eventually, considering I hacked over 30,000 codes for that game. hehe

This starts out the same as most things. BPW on Health address and, get

hit by an enemy, game halts. The address of the breakpoint here is 80071494, but that's a Branch. The actual opcode that wrote it is the next one down, 80071498. NOPing that will stop your health from being decremented when hit, but doesn't make you untouchable. To do that, you need to scroll down through the OPs til you find a 'JR $xx'. Look at 80071654. Bingo! JR $RA

JR is a "Jump Register." This jumps to the address stored in $RA. Now, is there something loading $RA just before that? Yep, 8007164C. We need a BPX after that loads to find out what it is. On Nemu, this is easy, I mentioned above. On GSCC2k2, yo'ull have to set it up...

This is what I recommend for GSCC2k2 users:
80071498: LUI $K1,8000
80071658: SW $K1,0060($K1)

In GS code form:
81071498 3C1B
8107149A 8000
81071658 AF7B
8107165A 0060

What that does is set $K1 to 80000000 (the game won't using $K1 for anything, remember?) and stores it to 80000060, which you'll set a BPW on. Now why did I pick those addresses? Well, 80071498 is that opcode that stores helath, we don't what it does and we don't really care about it, so we can change it. Now 80071658 is the opcode directly after JR $RA. Lucky for us it's a NOP, so it works out well. Now when you get hit again, it'll break. This time, we know what the break address is going to be, no suprise there. What we wanna know is what $RA is at the time. In this case, it's 80060A24. Now when you look at the ASM here, you'll see that it immediately loads $RA from somewhere else and does another JR $RA. This is not always the case, but it is this time, so BPX again. Change 80071658 back to NOP, if you didn't didn't already, and put your SW $K1,0060($K1) at 80060A30 (the NOP following the new JR $RA). Now this time we get the breakpoint and $RA is 80061D4C; go there. Look up 2 opcodes and you should se a JAL (Jump And Link). Yep, 80061D40 is JAL 800609A4. That jumps to 800609A4, which appears to be the start of the routine that causes Turok to take damage. NOP it and see if you become Invincible. Yep, Turok is now a God. What you just did was trace some ASM. In some games, you'll have to repeat those steps and trace back even further.

Modify Jump Height (Mega Jump!)

This one's rather hard to explain. You basicly have to find your own way of doing this, but I'll give you the general concept here. You have to find the ASM the writes the Moon Jump address when you first press the jump button. That's harder than it sounds because there are usually multiple constant writes on the Moon Jump addy, and one of those could be the one you're looking for so you can't just NOP them. I've found it possible with Nemu64 to keep resuming and get my charater to jump between breaks, but this can be a bit tricky. I don't know if it's XP or what, but sometimes I can move and such on the game when the Commands window is highlighted instead of the main game window, and other times I can't. One way I've found to get this to happen is to load the Mario 64 rom, then immidiately click onto whatever window is behind Nemu; odds are, you'll be able to press whatever keys you've assigned for the controller and you'll hear Mario's face getting trashed (The little *doing* you hear when you click his face). Then you'll know this is working and you can go ahead and load the game you wanna hack. If you can, get that far, then open the Registers window (Commands window should already be open) and go to the "COP 1" tab. Now keep resuming and tryin gto jump, as I said, and everytime time it breaks look at the reg being stored (F4,F6, etc) and see what the value is. You're usually looking for the one that's 4xxxxxxxx when being stored to the jump address. Get that far? Good for you. The hard part is over. Now there are 2 ways you can change the jump height.

1. You can search for that value in the RAM. This is the easy way out, however you might need to do it more than once if the game has different types of jumps (i.e. Banjo has normal jump and backflip).

2. Change the ASM. To change COP1 regs though, you can't just use ADDIU. You have to set a normal reg (like $k1) to the value you want to use with LUI then use MTC1 to put it in the COP1 reg that the game is storing. In most cases, you could also stick with the regular ops and change the SWC1 to an SH/SW. For example sake, I'll show you the way to do it with COP1:

```
LUI $K1,????  - Load Upper Immediate will set $K1 to ????0000
MTC1 $K1,Fx   - Move To COP1 moves the value in $K1 to Fx (x can be 1-31, in
                our case it should be the reg that's being store to the Moon
                                Jump address.
```

I know this description is a little rough. Imagine trying to write it up. ;)

If you really want to do this with GSCC2k2, I'd suggest setting up the BPW as HyperHacker describes for his Anti-Gravity How-To below, but jumping up instead of falling. This is just a theory on my part, as I don't have a working shark anymore.

Anti-Gravity Codes (by HyperHacker)

You will need GSCC for this. Knowing some ASM may help but isn't required.

1) Find the character's Y Speed. If there's an L Button To Levitate or Moon Jump code for the game, chances are it works by setting the character's Y Speed to a constant value. (Also in some cases making them jump - if there's more than one line besides the activator, remove some until the code only works when you're in the air - that's the Y Speed.)
2) Open GSCC, go into the RAM editor and start the game.
3) Go into the breakpoint window, and create a Write BP on the character's Y Speed. Don't enable the BP yet, but use the Tab key to make sure the Set BP button will activate when you press Enter.
4) In the game, fall off something. (Don't jump, you have to be moving down.) While in the air, press Enter on the PC to enable the BP.
5) The game should halt immediately. Clear the message box that appears and GSCC should go to the BP address. (Check the N64regs.txt file it opens to be sure, since sometimes it goes to the wrong place. The number beside "PC" is the BP address.)
6) Write down the 16-bit value at this address. Change it to 2400 and the gravity should be disabled! (You may still slowly fall, some games do that.) If not, repeat the process (don't change this value back) until you find it.
7) Make a code out of it!

Note: No gravity may get annoying and make the game hard to play. I suggest you set it up like my Mario 64 codes, this way:
[Activator]
[Anti-gravity code]
[Activator]
[Enable Gravity code*]
[Activator]
[Y Speed = Positive code**]
[Activator]
[Y Speed = 0 code**]
[Activator]
[Y Speed = Negative code**]
Each activator should be different of course.
*The Enable Gravity code is simple. Take the Anti-gravity code and change the value back to what it originally was.
**These are codes to set the character's Y Speed to any positive value, 0, and any negative value, allowing the player to move up and down. Play around to find a nice speed. Also, some games crash when setting the speed to 0, in which case I recommend about 3000 for most games (which isn't stopped, but so super slow you wouldn't notice).

Walk Through Walls (General Theory Only)

I haven't done a great deal of looking for WTWs yet, but I thought I'd share the general concept of how you'd find them with ASM because they've always been such a hot topic.

Theory 1:

On games where you take damage from hitting walls (i.e. certain racing

games) you could BPW on healh/energy and hit a wall to cause the break. Then start backtracing the ASM a bit and NOPing the Jumps. I did this with F-Zero X and was able to pass through the walls, althrough you plummit to your death if you slow down. lol

Theory 2:

Now what about the typical games, where you just bounce your head off the walls but nothing happens? This may be a little tricky. The idea here would most likely be to BPW on your X/Z coordinate and find the ASM that causes you to move. Then look around for branches that skip it when you hit a wall, or an opcode that stores your X/Z again when you hit something. I don't imagine this will be easy, but I know someone will want to try it. :)

Breaking Limits

Ever wanna carry more than 100 bullets, but the game wouldn't let you? I'll show you why and how to get around it. Let's use Turok Dinosaur Hunter v1.0 for this one. Get near a clip with something other than the pistol equipped (to avoid extra breaks). Now set a BPR and BPW on the pistol ammo. You'll get an immediate break on this one. The ASM that breaks is actually what tells if you're allowed to equip the pistol or not (this is how I made the weapons shoot without ammo). NOP that and set the BPs again. Now you can walk over the clip to cause a break. You'll find yourself at 80057DB4, which is a Branch; the Load Word is in it's delay slot. Now if you look through the next few opcodes, you'll see these:

```
80057DC0: SLTI   $at,$t0,00C8
          -Set On Less Than Immediate.
          -if $t0 is less than 00C8, $at is 1, otherwise $at is 0
80057DC4: BNEL   $at,$zero,80057F68
          -If $at is Not Equal To $zero, then jump to 80057F68
80057DD8: SLTI   $at,$t0,0064
          -Set On Less Than Immediate.
          -if $t0 is less than 0064, $at is 1, otherwise $at is 0
80057DDC: BNEL   $at,$zero,80057F68
          -If $at is Not Equal To $zero, then jump to 80057F68
```

C8 (200) is the maximum pistol ammo when you have a backpack and 64 (100) is the max without the backpack.

If you look at 80057DB0 and B4 you'll see it loads the word (32-Bit) value from 80128D28 (the backpack's on/off address) and branches accordingly. You can NOP the Branch there to force the pistol to always have the packpack available, or force the Branch so the backpack is never avilable to the pistol. Now, we can do a couple different things with the opcodes I showed you above. You probably noticed by now that you can change the values on the SLTIs to whatever you want to mod the maximum pistol ammo with/without backpack. Now how bout killing the limit entirely? Easy. Just force the branch. We can do this by changing it to a Jump, but that'll make it a 2 line code to replace each Branch. Instead, we'll change the branch to BEQ $zero,$zero,80057F68. I use this just like I use Short NOPs. 1000XXXX is BEQ $zero,$zero,XXXX address. :)

Wait, we're not done yet. That will allow you to pickup ammo no matter how much you have, but if you try it, you'll see you're still capped at 64/C8. This is because there's another check where it actually adds the pickup to your total, so when you have 99 bullets and pickup 10 you still end up with 100. When you get the next breakpoint (you can NOP the LW here or just resume), you'll get a useless one at 8009269C. NOP it and go again. Now you'll be at 80057B14. See anything interesting?

```
80057B28: SLTI   $at,$t6,00C9
80057B2C: BNE    $at,$zero,80057D10
80057B30: ADDIU  $t9,$zero,00C8
80057B40: ADDIU  $t1,$zero,0064
80057B44: SLTI   $at,$t0,0065
80057B48: BNE    $at,$zero,80057D10
```

and again you'll see it check for the backpack and Branch accordingly.

With This group, notice the ADDIUs. You can change just these and make it so when a pickup would bump you over 100/200 bullets, it'll give you like 500 instead. You could change both the SLTI and ADDIUs to change the max ammo

with/without backpack. Lastly, you can break the limits entirely by forcing the Branches like I showed you.

So from 3 breakpoints, we can make all these codes, and probably more that I haven't even thought of... like adding bullet pickups to other weapons instead.

```
Max Bullets Modifier (Without Backpack)
81057B42 ????
81057B46 ????
81057DDA ????
Max Bullets Modifier (With Backpack)
81057B32 ????
81057B2A ????
81057DC2 ????
Pistol Can Shoot Without Ammo
810583F8 1000
Pistol Can Shoot Without Ammo (Alternate)
810583F4 240A
810583F6 0001
Break Max Bullets (Without Backpack)
81057B48 1000
81057DDC 1000
Break Max Bullets (Without Backpack)(Alternate)
81057B44 2501
81057DD8 2521
Break Max Bullets (With Backpack)
81057B2C 1000
81057DC4 1000
Break Max Bullets (With Backpack)(Alternate)
81057B28 25C1
81057DC0 2501
Always Have Backpack For Bullets
81057B20 2400
81057DB4 5400
Always Have Backpack For Bullets (Alternate)
81057B18 240F
81057DB0 2419
Always Have Backpack For Bullets (Alternate 2)
81057B22 0001
81057DB6 0001
```

Copy Bytes Codes

You might've noticed that PSX and all the newer systems have a code type called "Copy Bytes." What this does is copy values from one location in RAM to another. You can use ASM to do this on N64 if you want, since Datel didn't bother to include the code type in the N64 shark.

First, find an ASM routine to 'hook' into. You'll probably want something that's always executing, but there may be cases where you want something else. The easiest way I've found to do a Copy Bytes that's always copying is to find a routine that reads the button/stick activators for the game. I'll use Super Mario 64 as our example in this one....

We'll do a BPR on the button activator (00367054). We got 80323B58 as our break address. Now if we look a few opcodes down, we'll see that we can safely use $t2, $t3, $t4, and $t5. You could jump from 23B58, but I'm going from 23B54 instead -- just my preference. Now, let's jump to 80400000 so we have plenty of room to work, and I'll show you how to copy the value of one address to another.

```
80323B54: J 80400000 - the Jump (duh)
routine now begins at 8040000
LUI    $t2,8020
ADDIU  $t3,$t2,7702
ADDIU  $t2,$t2,7772
LH     $t4,0000($t2)
SH     $t4,0000($t3)
J      80323B5C
SW     $at,0000($t8)
```

That loads the 16-Bit value in 80207702 into $t4, then stores it to 80207772.

The Store Word (SW) at the end is just the opcode that the Jump replaced from 80323B54.

Now here is an example routine that copies a lot. This will copy 80207700 - 8020776C (Star & Coin Records for File 1) to 80207770 - 802077DC (Star & Coin Records on File 2).

```
80323B54: J 80400000 - the Jump (again)
routine begins at 80400000
LUI    $t2,8020
ADDIU  $t3,$t2,7770
ADDIU  $t2,$t2,7700
ADDU   $t5,$zero,$zero
LW     $t4,0000($t2)
SW     $t4,0000($t3)
ADDIU  $t2,$t2,0004
ADDIU  $t3,$t3,0004
SLTIU  $k1,$t5,006C
BNE    $k1,$zero,80400010
ADDIU  $t5,$t5,0004
J      80323B5C
SW     $at,0000($t8)
```

Ever used a For Loop in other types of programming? Same concept here. If $t5 is less than 6C $k1 is set to 1. If $k1 is Not Equal To $zero then it jumps back to the LW opcode and proceeds through again.

Writing Routines, Jumping In And Out, Etc

Ok, I bet you're wondering what you can do when you want to change something but there's no room to do it within the current routine (no NOPs or useless ops to change). This is where Jumping comes in. Remember in my Invincibility code example, I showed you how to make Turok Invincible? Well, what about if we want to do somethin ga little different? Like make him take Double or Half Damage? Multiply the damage? Well, first you need to find the opcode that subtracts from Turok's health when he's hit. We found the the opcode that actually writes the new helath value (80071498), so it should be a few opcodes or so before that. Look at 80071490. SUBU $V0,$T6,$V1. This means "$V0 = $T6 - $V1" so $V1 would be the amount of damage taken. We need to find a play to jump out of this routine now, so we can manipulate $V1 a bit. This has to happen after $V1 is loaded, but before it's subtracted from $T6. Also, You can't have 2 consecutive Jump or Branch opcodes because jumps and branches all have that Delay Slot and putting another Jump/Branch in there will crash the game. With that Delay slot you also need to take into account the opcode you're replacing with a Jump. I'll show you what I mean...

```
80071478: LW $V1,003C($SP)
8007147C: BEQ $V1,$ZERO,80071538
80071480: NOP
80071484: LH $T6,01DC($T0)
80071488: LUI $A0,8010
8007148C: ADDIU $A0,$A0,9E60
80071490: SUBU $V0,$T6,$V1
80071494: BGEZL $V0,800714A8
80071498: SH $V0,01DC(T0)
```

Ok, see 80071488 and 8007148C? Those need executed in that order, so if you put a jump at 80071488 that next opcode that uses $A0 would be in the Delay Slot. We don't want that. 80071484 looks good here. That loads Turok's current health. Write that opcode down before you change it because you want it to still get executed at some point in your routine, so you're not losing anything. Ok, we picked our address to Jump from. Now where do we jump to? You can jump to any part of the RAM that the game doesn't use. Most games don't use the following areas after boot up (so they're free for us to play with):
80000058 - 8000007C
80000090 - 800000C0
and typicly you'll find a good amount of space somwhere in the area between 800002B4 and 800003F0
Also, if you're hacking a game that doesn't require the expansion pack, but you have it anyway, then you've got plenty of RAM to play with (804-808!).

Ok, in this example, we'll Jump to 80000060. The Opcode we'll use is:

```
J 80000060
```

That amounts to
81071484 0800
81071486 0018
as our Jump code.

The routine itself is going to double, multiply, or divide $V1. I'll show you 3 ways of accomplishing this, but there are more.

Double:
ADDU $V1,$V1,$V1 - pretty self explanatory, I think $V1 = $V1 + $V1

Multiply: (This is where it gets a little more interesting)
ADDIU $K1,$ZERO,00?? - We set $K1 to the amount we want to multiply by
MULT $V1,$K1 - Multiply $V1 by $K1. The result is stored in "LO"
MFLO $V1 - Move the result from LO to $V1. LO and HI are used for some special calculations (like MULTU and DIVU). If used with a DIV opcode, LO is the division result and HI is the remainder.

Divide:
ADDIU $K1,$ZERO,00?? - We set $K1 to the amount we want to divide by
DIV $V1,$K1 - Divide $V1 by $K1. The result is stored in "LO"
MFLO $V1 - Move the result from LO to $V1.

Let's say you've decided on Dividing by 2 so Turok takes Half Damage. Next, we need to know where we're Jumping back to. This is normally the address of the Jump plus 8 (we need to skip the Delay Slot since it'a already done). So...

```
ADDIU $K1,$ZERO,0002
DIV $V1,$K1
MFLO $V1
J 8007148C
```

Wait a minute. What comes after the Jump? The Delay slot needs to have a valid ASM opcode (Like a NOP). Oh, what about that opcode that loads health? :)

```
ADDIU $K1,$ZERO,0002
DIV $V1,$K1
MFLO $V1
J 8007148C
LH $T6,01DC($T0)
```

That's more like it. Note that in some cases, you may want that opcode to execute first (like if you needed the health value for something there), but for this purpose we don't need to. When you can, I recommend putting the last opcode of your routine in the Delay Slot. If you wanted that health load first then you'd do it like this:

```
LH $T6,01DC($T0)
ADDIU $K1,$ZERO,0002
DIV $V1,$K1
J 8007148C
MFLO $V1
```

That would work just the same. Only difference here is that the LH is done first instead of last.

Once you get done typing that into Niew, you'll see your code is:
81071484 0800
81071486 0018
81000060 241B
81000062 0002
81000064 007B
81000066 001A
81000068 0000
8100006A 1812
8100006C 0801
8100006E C523
81000070 850E
81000072 01DC

There you have it. "Turok Takes Half Damage"

Tip: Jump And Link

Once you get used to jumping and jumping back, you may want to do it the easy way. Jump And Link (JAL) works like J but it stores the return address in $RA for you. Note that you can't always do this because the game may not load $RA again before it uses it. If you look down through the opcodes in the routine you're jumping out of, you should be able to fins a JR $xx at the end. If this is something other than $RA or it loads $RA from somewhere first, then you're in good shape. Otherwise you'd have to back up that reg somewhere if you're going to do it. A quick way to tell if you can do this other than lookin for the JR $xx is to see if you notice a JAL right in the area. If the game uses JAL there, then you can pretty well bet it's gonna load $RA before it uses a JR $RA. So to do the above code the way I like to do them....

81071484 becomes JAL 80000060
and
8100006C becomes JR $RA

The new code would be:
81071484 **0C00**
81071486 0018
81000060 241B
81000062 0002
81000064 007B
81000066 001A
81000068 0000
8100006A 1812
8100006C **03E0**
8100006E **0008**
81000070 850E
81000072 01DC

Tip: Using Regs Other Than $K1

If you need more than just 1 free reg to use in a routine, you'll have to look for ones that aren't in use. The safest thing to do, is use regs that you you will be set right after your routine. In the Tutok example, you could use $T6 since it's set at the end of your routine anyway. You could also safely use $V0, $T7, $AT, and $T8 in this example (look at routine to see why).

Expample Of A Longer ASM Routine

Well, you've seen some of the things you can do here already, but just for good measure I'm going to show you a couple longer, slightly more advanced routines that I wrote for WWF No Mercy.

Example 1: Universal P1 Ultra Code.

Ever play a game where the player & CPU codes can become switched because of how the game is setup? The breakpoint for this was a BPR on P1's Health/Spirit. The general idea here to to find which wrestler is human controlled and keep giving that wrestler Special, Full Spirit, Full Health, Max Health, and Full Health For All Body Parts (so you don't submit).

```
800FE770: JAL   80400000       -Jump to 80400000 And Link, $RA is now 800FE778
80400000: LB    $V0,00E9($A0)  -Load's the value of the CPU/Human address
80400004: ANDI  $V0,$V0,000F   -$V0 = $V0 AND 000F, if this is 0, it's human
80400008: BNEZ  $V0,80400038   -Branch On Not Equal, skip to addy if $V0 != 0
8040000C: ADDIU $V0,$ZERO,0004 -$V0 = 0 + 0004
80400010: SB    $V0,00EE($A0)  -Store Byte $V0 to $A0 + 00EE
80400014: ADDIU $V0,$ZERO,0064 -$V0 = 0 + 0064
80400018: SH    $V0,00AE($A0)  - Store Halfword $V0 to $A0 + 00AE
8040001C: SH    $V0,00AC($A0)  - the rest of this is the same way: set, store
80400020: SH    $V0,00AA($A0)
80400024: ADDIU $V0,$ZERO,4248
80400028: SH    $V0,02C4($A0)
8040002C: SH    $V0,02C8($A0)
80400030: SH    $V0,02CC($A0)
80400034: SH    $V0,02D0($A0)
80400038: SH    $V0,02D4($A0)
8040003C: JR    RA             -Jump Register: Jumps to address stored in $RA
80400040: NOP                  -We'll use a NOP as a Delay Slot since we don't
```

always want that SH to execute.

In this example, we used $A0, which the game sets to a different wresters' pointer each time it goes through. So we load the CPU/Control info for whichever wrestler the game is looking at and check if that wrestler is human. If it is, we go through our routine of writing that wrestler's stats. If not, we Branch (skip) over it and just Jump Back.

Using The COP1 Registers And Instructions

If you've gotten the hang of this stuff and maybe tried finding some more advanced types of codes, or even stopping timers on some games, you've probably noticed COP1 instructions. COP1 has its own set of opcodes and registers for working with floating point values.

How do values get into and out of the floating point registers (FPRs)?

They can either be loaded just like we typicly see, using LWC1, or by moving values from the main regs.

```
Loading from RAM:
LUI  $V0,8014      -Load Upper Immediate, we already know. $V0 = 80140000 now.
LWC1 F2,5420($V0)  -Load Word COP1 works just like LW but puts the value into
                    an FPR, F2 in this case.
LUI  $V1,8014      -Load Upper Immediate. $V1 = 80140000
LDC1 F4,5420($V1)  -Load Doubleword COP1. Same as above. Used to load a 64bit
                    value as a double precision floating point value, I guess.

Moving from the main regs:
MTC1 $V0,F2       -Move To COP1. F2 = $V0

Another example:
LUI  $V0,3FAB      -Load Upper Immediate. $V0 = 3FAB0000
ORI  $V0,22D1      -OR Immediate performs a bitwise logical OR on $V0. If you
                   haven't used OR before, it sets any bits that aren't set.
                   Since the lower bits aren't set (3FAB0000), this is
                   just like adding 22D1 to $V0. If $V0 is 3FAB2200 and you
                   OR it by 22D1, then it'll be 3FAB22D1, as this only sets
                   what wasn't already there.
MTC1 $V0,F2        -Move $V0 to FPR 2.

Getting values out of FPRs:
SWC1 F5,5420($V0)  -Store Word COP1. The usual method of writing the value to
                    a RAM location.
MFC1 F5,$V1        -Move From COP1. Moves the value from an FPR reg to a main
                    reg. Thus, $V1 would now be set to the value in F5.
```

How do I do math operations with FPRs?

Again, this is the same basic concept as you do normally, it's just more confusing. COP1 supports both single precision floating points and double precision floating points. You should have a fair idea of what the 2 mean, if you've programmed much.

```
Adding:
ADD.S F1,F4,F8     -Floating Point Add, Single Precision. F1 = F4 + F8.
ADD.D F1,F4,F8     -Floating Point Add, Double Precision. F1 = F4 + F8.

Subtracting:
SUB.S F1,F4,F8     -Floating Point Subtract, Single Precision. F1 = F4 + F8.
SUB.D F1,F4,F8     -Floating Point Subtract, Double Precision. F1 = F4 + F8.

Multiplying:
MUL.S F1,F6,F8     -Floating Point Multiply, Single Precision. F1 = F6 * F8
MUL.D F1,F6,F8     -Floating Point Multiply, Double Precision. F1 = F6 * F8

Dividing:
DIV.S F3,F9,F16    -Floating Point Divide, Single Precision. F3 = F9 / F16
DIV.D F3,F9,F16    -Floating Point Divide, Double Precision. F3 = F9 / F16
```

I like multiplication and division better in COP1 because it doesn't require

you to use the extra OP to extract the result. There are also some other neat COP1 OPs like NEG, ROUND, FLOOR, CEIL, etc. This was more of a basic intro to using COP1.

F) Hack Your Shark!? - by HyperHacker

It's actually possible to hack an N64 Gameshark/AR v3.3. You can use these codes with any game and hack the GS! Sweetness!

81791DF6 ????
81791E02 ???? Lowest Goto Address (normally 80000400)

81791E06 ????
81791E12 ???? Highest Goto Address (normally 803FFFFF)

81791E32 ????
81791E36 ???? Lowest Scroll To Address (normally 80000400)

81791E3A ????
81791E3E ???? Highest Scroll To Address (normally 803FFFFF)

81791E42 ????
81791E4A ???? Address Returned To When Attempting To Go To Invalid Address (normally 80000400)

807FFDE7 000? Is Controller Active

817FFDE8 ???? Button Pressed Data

817FFDEA ???? Joystick Data (if you can find any use for these last 3, tell me!)

817E9C98 ????
817E9C9A ???? Cursor X Coord

817E9C9C ????
817E9C9E ???? Cursor Y Coord

817E9CA0 ????
817E9CA2 ???? Cursor Mode

807E9D34 - 807E9D44 Text Searched For
You can use this to turn the Text Search into a string search for almost any number, but beware - If you enter a number between 41h and 5Ah, it will also search for numbers 20h higher. This is to allow strings to be found in caps or lowercase. Also, if you enter a number lower than 20h, it will crash everything, as these are special control bytes rather than text.

I've also hacked the active code list! Using this along with the codes to goto any RAM, you can alter active codes on the fly. Plus, if you know N64 ASM, you can create your own types of codes! The list starts at 807C0000. Basically, it's just a bunch of ASM that is executed several times per second. The formats are as follows:

80 Code Type (constant 8-bit write)
Format: 3C1A XXXX 375A YYYY 241B ZZZZ A35B 0000
ASM: lui $k0,XXXX
ori $k0,k0,YYYY
addiu $k1,$zero,ZZZZ
sb $k1,0000($k0)

81 Code Type (constant 16-bit write)
Format: 3C1A XXXX 375A YYYY 241B ZZZZ A75B 0000
ASM: lui $k0,XXXX
ori $k0,$k0,YYYY
addiu $k1,$zero,ZZZZ
sh $k1,0000($k0)

88 Code Type (8-bit write, G$ Button triggered)
Format: 3C1A BE40 375A 0000 875B 0000 0000 000F 337B 0400 1760 0004, then normal 80 code type format
ASM: lui $k0,BE40
ori $k0,$k0,0000

```
lh $k1,0000($k0)
sync
andi $k1,$k1,0400
bne $k1,$zero,00000028
Then normal 80 code type ASM

89 Code Type (16-bit write, G$ Button triggered)
Format: 3C1A BE40 375A 0000 875B 0000 0000 000F 337B 0400 1760 0004, then
normal 81 code type format
ASM: lui $k0,BE40
ori $k0,$k0,0000
lh $k1,0000($k0)
sync
andi $k1,$k1,0400
bne $k1,$zero,00000028
Then normal 81 code type ASM

D0 Code Type (8-bit activate if equal)
Format: 3C1A XXXX 375A YYYY 835A 0000 241B ZZZZ 175B 0004, then code to be
activated
ASM: lui $k0,XXXX
ori $k0,$k0,YYYY
lb $k0,0000($k0)
addiu $k1,$zero,ZZZZ
bne $k0,$k1,00000024
Then code to be activated

D1 Code Type (16-bit activate if equal)
Format: 3C1A XXXX 375A YYYY 875A 0000 241B ZZZZ 175B 0004, then code to be activated
ASM: lui $k0,XXXX
ori $k0,$k0,YYYY
lh $k0,0000($k0)
addiu $k1,$zero,ZZZZ
bne $k0,$k1,00000024

D2 Code Type (8-bit activate if not equal)
Format:
ASM:

D3 Code Type (16-bit activate if not equal)
Format:
ASM:

THE LIST MUST END WITH THE FOLLOWING CODE:
3C1A 8000 375A 0120 0340 0008 0000 0020
IF IT DOES NOT, THE GAME WILL FREEZE AS PROCESSING WILL NOT RETURN TO THE GAME
PROGRAM.
```

G) Hacking Playstation 2 Codes

There are no trainers, code gens, etc publicly available for PS2 as of right now, and according to CodeMaster doesn't look promissing. This is because it's so hard to get things like this working on the system. So how are some of us doing it?

WARNING: Either of these methods could potentially cause damage to your system if you enter bad codes! This isn't a common occurence, but you have been warned! The authors are not responsible for any damage you may cause to your PS2!

Guess & Check

Yeah, the old "Guess & Check" methods are usuable on PS2. This can be done with encrypted codes, but it's recommended that you play with RAW codes and encrypt them yourself. The Codebreaker PS2 actually has the ability to accept RAW codes, I think. This can make things easier. Still, you're left with a problem: There's too much RAM to play with, and constatly rebooting with a CD based cheat device is a real pain. The only shortcut I've found if you're really looking to find a specific effect is to use patch codes to conver a certain amount of the RAM at a time til you get the desired effect. Then narrow down the patch a little at a time til you lose the effect and you'll know where the code is. *NOTE: To pretty well eliminate the risk of*

messing up your PS2 with a faulty code, stay in the same area of the RAM as the existing codes for thew game you are hacking.

Using PS2DIS - Courtesy hellion (hellion00.thegfcc.com)

1. Introduction

   First things first. Using PS2DIS to home-hack GameShark codes is NOT easy (for the most part). It helps out a lot, but still requires a fair amount of programming knowledge to get the really good codes. I will try to show you in this guide how to make the best use of it you can. You will soon see why it is almost useless for a lot of (or most) games, but also why we were so successful with GTA3.

   The FAQ assumes that you have a basic knowledge of the hexidecimal number system and some key programming concepts (functions and variables).

2. Getting Started

   Here's what you'll need in order to use it:

   a. Download the program from HERE and extract it to your harddrive. (Hanimar's (the creator of PS2DIS) site is here: http://www.geocities.com/SiliconValley/Station/8269/ps2dis/)

   b. Using the DVD drive in your computer (or a friend that has one), copy the SLUS file off of the game. If the game is in CD-ROM format (instead of DVD), a normal CD drive will work fine. The SLUS files usually have a name like "SLUS_###.##". For example, the GTA3 SLUS file is named "SLUS_200.62". Some start with SCUS, and PAL format discs have ones that start with SLES. There are other variations out there, but that should cover the majority of them.

   c. Once that file is on your harddrive, open up PS2DIS (by running the ps2dis.exe exectutable) and use File->Open to open the file you copied off the disc.

   You are now ready to start your hacking. :)

3. Display

   This is what you should be looking at on your screen now:

   Top Part (Grey):

   This shows you the data as it appears in memory. I don't use this too often, but it gives you the addresses, hex values, and corresponding alphanumeric values of those hex bytes.

   Bottom Part (Blue):>

   1st column: This is the address of the current line of code. Since the PS2 instructions are 32-bits, it only shows

you every 4th address (this can be modified, as you'll see later).

2nd column: This is the 4 bytes of data that is stored at the address in column 1.

3rd column: This column is for labels (more about them later). This column is blank for most lines.

4th column: This shows you the disassembled instruction that corresponds with the data in column 2 (read more about instructions in the MIPS guide).

4. Navigation Controls

Getting around in the DIS is fairly easy once you get used to it. You can move between lines of code using the Up and Down arrows. The Page Up and Page Down keys do exactly what you'd expect them to do. Using Ctrl+Page Up or Ctrl+Page Down will jump up or down by a large number of addresses (+/- $00001000). Also, by holding Shift while using the Up and Down arrows, you can scroll up or down while keeping an address you want highlighted. This is handy just in case you tend to lose it when you scroll.

Occasionally, you will see a line of code where the disassembled code has an up or down arrow in it. This indicates a Jump or a Branch (more on these instructions in the MIPS guide). If you highlight that line of code and press the Right arrow, it will take you to whatever line of code that instruction was jumping or branching to. This is extremely useful in tracing sections of code to see what it does.

If you use the Right arrow to get to a Jump or Branch's destination, you can use the Left arrow to go back to the original instruction. Be careful though, because if you use it too many times, it will take you back to the address it started at when you opened the file and you'll lose the place that you were at.

If you know exactly what address you want to go to, you can press the G key. This will bring up a dialog where you can type in an 8-digit address and it will take you right there.

One of my favorite features of PS2DIS is the Label Listing. Press Ctrl+G to bring it up. It will allow you to jump to any of the labeled lines in the SLUS. It is very handy for jumping directly to certain functions or variables that are labelled.

5. Labels

Labels are the key to whether or not PS2DIS will be extremely helpful in hacking a game. There are basically three types of labels:

a. Labels for strings of text - These are most common. Every game I have seen has string labels in its SLUS. These are also the most useless types of labels (in most cases). Rarely will the strings give you a good idea of what a function does or where

variables are stored. String labels begin and end with the double-quote character (").

b. Labels for variables - These labels rarely appear in SLUS files. Making cheats with these labels is a relatively simple process of setting a value at that address. The hard part is knowing what value to set in order to make the cheat effective.

c. Labels for functions - These labels also very rarely appear. These labels appear at the beginning of a section of code that represents a function. The name will often (though not always) give you a fairly good idea of the purpose of that function. These labels are helpful in making cheats that either disable functionality (e.g. Disable Water For Cars), or change which functionality is used (e.g. Boat Guns On Cars)

As I said, most SLUS files do not contain variable or function labels. This makes it very difficult to hack games without the tools that the guys that work at GameShark have (the expensive stuff). However, GTA3 just happened to have tons of labels in it, which gave us lots of stuff to work with.

6. The Analyzer

This is probably the most useful tools that PS2DIS has to offer. When you invoke the Analyzer (Analyzer->Invoke Analyzer), it runs through then entire code segment and figures out all the places that each address is referenced from. This process could take a while on slower PCs, so be patient. Once it's finished, you can "mark" any line of code and cycle through all the addresses that reference that line.

To "mark" a line of code, simply highlight the line and press the Space Bar. The selected line will turn grey instead of blue.

To cycle through all of the "referers" for that line, press F3 to go forward and Shift+F3 to go backwards. Finding referers is usually only effective for a variable's address, the first line of a function, or the first byte in a string.

7. Miscellaneous

Finding Patterns - If you have a certain hex value that you want to search for, you can do that by clicking Edit->Find Patern. Click the "As Hex String" checkbox and put in what you are looking for. Remember though, that MIPS stores the data in Little Endian format (the bytes are reversed). For example, if you were trying to search for a value that would look like this in the 2nd column: "2403003d", you would want to search for the following hex string: "3d 00 03 24". It takes a while to get used to, but once you get used to it, it's easy.

Changing Address Display - You can make the DIS display all four bytes at an address individually instead of all at once by selecting the line and pressing the B key. You can change it back by selecting the first line in the group of four and pressing the C key. This is mostly useful in finding and editing string data.

8. Resources

   Use the resources below to learn more about the hexadecimal number system and the MIPS Assembly Language. MIPS is the assembly language that appears in PS2DIS when disassembling the SLUS files. Some of our best codes have come from modifying what the code actually does. In order to do this, you need to know how MIPS works.

   Hex Resources:

   - Binary/Hexadecimal Tutorial (not a very good one :\ )

   MIPS Resources:

   - CA225b MIPS Assembly Language Programming
   - Programmed Introduction to MIPS Assembly Language
   - CSc 116 Notes (another MIPS class)
   - MIPS Assembly Language Programmer's Guide

9. Examples

   The examples below required the GTA3 SLUS file (**SLUS_200.62**). We can't put it up here for download, so if you don't have the game or a DVD drive, you need to get it from somewhere else. :\

   - Labeled Variables

Disclaimer: Very few SLUS files have the proper labels for this kind of code. Don't be disappointed when this strategy doesn't work for most other games.

Open the GTA3 SLUS in the DIS. Today, we're going to be looking at a couple of simple variable-setting codes. The first thing we need to do is open the Label Listing (Ctrl+G) and see if we can find anything interesting.

Now, there are lots and lots of labels in the GTA3 SLUS, and most of them just aren't helpful at all. It takes a lot of patience to look through them to find the ones that are useful. To save you some time and to facilitate the learning process, I'll go ahead and point out a couple of them for you.

Example #1

With the Label Listing open, type **Lives** in the text box at the top. The selection will automatically move to the first label that starts with what you typed. If you press **Enter**, the DIS cursor will move to the line of code that has the selected label. You should now have selected the address for "LivesSavedWithAmbulance__6CStats". As you can probably guess from the label, this stores the number of people you have saved in the Paramedic Missions.

Now that we have an address (**00416EE8**), we need to figure out what value we want to use. Remember that this value needs to be in hexadecimal form. Let's say we want to have 1000 people saved. Converting 1000 to

hexadecimal gives us **3E8**. (Windows Calculator in Scientific Mode converts between decimal and hexadecimal easily)

With an address and value in hand, we need to figure out what GameShark command we want to use. Since 3E8 is larger than 8 bits (2 hex digits), we can't use the 8-bit Write. 3E8 is smaller than 16 bits (4 hex digits), so we can use the 16-bit Write. The format for that is **1aaaaaaa 0000dddd**, where **a** is the address and **d** is the value.

Putting it all together, we have: **10416EE8 000003E8** (dropping the first zero from the address to make it 7 digits, and adding a zero to the value to make it 4 digits).

Last, you need to encrypt the code. The encryption you use depends on the game you are hacking. GTA3 GameShark codes use the 1456E7A5 encryption, so we will use that as well.

Using the converter to encrypt the code gives us:

1000 People Saved
4CD8F110 1456E4FD

Example #2

Open the Label Listing again and type **Respray** in the text box. It should bring you to the label "RespraysAreFree__8CGarages". Press **Enter** to go to that address.

The name of this variable implies that this is a yes or no question. Are resprays free or not? In programming, this is known as a Boolean variable (stores True or False). True is normally represented by a 1, and False is represented by 0. Taking this into account, it would make sense that if we set this variable to 1, then resprays would be free.

So, we take the address (**00416F90**) and the value (**1**, 1 converted to hexadecimal is still 1) and figure out what command we need to use. Since 1 is less than 8 bits, we can use the 8-bit Write. The format for that is **0aaaaaaa 000000dd**.

Filling in the address and value, we have: **00416F90 00000001** (again, dropping the zero off the address and adding a zero to the value to make it two digits).

Encrypting with the 1456E7A5 encryption gives us:

Free Resprays
3CD8F0B8 1456E7A6

H) Hacking Sega Dreamcast Codes

The only known way to hack Dreamcast codes at this point is the old "Guess & Check" method. The Dreamcast Xploder (or cracked shark) is your best bet for this, as it doesn't require encrypting codes. If you really must use a shark, then you'll have to use DCCypt to decrypt/encrypt codes. Patch codes are a big help, as they'll save you from hours on end of entering 1 code at a time to try. Use a patch code to cover a certain amount of RAM your want to change to see the effects and see what you get. If you get an effect you like, just narrow down your patch a few addresses at a time til the effect stops showing up. Then

you'll know what addresses to try individually.

WARNING: Trying codes at random could potentially cause damage to your system if you enter bad codes, so stay in/near the area where other codes have already been found! This isn't a common occurence, but you have been warned! The authors are not responsible for any damage you may cause to your DC!

XI) How-to Guide - Gameboy Hacking

A) How to hack GameBoy Advance codes: (By Tolos)

You will need an emulator (I use VisualBoy Advance 9.1), and a GBA ROM to hack. DO NOT ASK ME FOR ROMS!

Before you begin, make sure the game and ROM are the same versions, and there is no "extra stuff" in the ROM -- usually an opening sequence by the rippers. Ripping the game yourself solves these problems.

Here's an example of finding infinite lives:

- Start VisualBoy Advance, and load your ROM.
- After it boots, start the game.
- Take note of how many lives you have.
- (Pause the game)Select the menu Cheats -> Search for Cheats.
- Click on Start, enter the number of lives (e.g. 3) into the text box near the bottom, and press Search.
- You should come up with quite a few results. Click OK, and continue the game.
- Die once, and press CTRL+C or open the Search for cheats menu again.
- Enter the number of lives you now have (e.g. 2) and press Search, without clicking on Start.
- If you still have a large amount of results, repeat the above three steps. Otherwise, click on one of the addresses, then press Add Cheat. Change the value. If your lives didn't change, try dying. Sometimes codes aren't immediately obvious. If that wasn't the code, try the others.

If none of the codes worked, you may need a different approach.
- Press start, and enter a number well greater than the number of lives, then then press Search.
- Lose a life, then do a Search for Cheats.
- In the Search Type box, select old value; in the Compare Type box, select Less or Equal, then press search.
- Repeat the above two steps.
- Select an address and click Add Cheat. Change the value, and die. If your lives didn't change, try the other addresses. You should find the code.

Basically, the same techniques that work for other games can be used for finding codes.

B) Gameboy Advance Size Modifiers How-To - by Icy Guy

What you need:

-A ROM of a GBA game (find one on your own)
-An emulator (VisualBoy Advance is highly recommended - any version will work)
-AR Crypt

For this guide, it is assumed that you are using VisualBoy Advance ("VBA"). It's also recommended that you assign a keyboard combination in VBA that will let you open the "Search for cheats..." menu quickly (Tools > Customize; click "CheatsSearch" and assign your keyboard combo - I use Ctrl+C).

Unknown Value Method:

This method is good for objects that normally change shape. In this example, we'll be hacking a size mod for your racer in Mario Kart: Super Circuit. (Hacking by example is a great way to learn.)

-Start VBA and load the MK:SC ROM.

-Once in the game, select "Time Trial" from the main menu. (If, like in this case, you're hacking a racing game, choosing a time trial is a good idea, because, depending on the game, you won't have to contend with other racers and the clock.)

-Choose your racer, and then select Bowser Castle 1 from the tracks menu.

-Now follow the normal track route until you reach the Thwomps.

-Go to the "Search for cheats..." menu (Cheats > Search for cheats...). Under "Search type," choose "Old value," under "Data size," choose "16 bits," under "Signed/Unsigned," choose "Hexadecimal," and, under "Compare type," choose "Equal." Put a check in the "Update Values" box.Now hit the "Start" button, and then "Search." It will tell you that there are too many results. Hit "OK," and then the "OK" in the search menu.

-Back in the game, go and get yourself flattened by a Thwomp. Now, while your character is still flat, go to the cheats menu, choose "Less than" as your compare type, and search. It will probably still say that you have too many results, but then again, it may not.

-Go back to the game and move out from under the Thwomp. When your character returns to normal size, go back to the cheats menu and do a Greater than search (the menu actually says "Greather than," but same thing). Now you'll get some results. But there are too many to handle, so go back to the game.

-Get flattened, and do the Less than search while flat. Go back to the game, and when your character pops back to normal size, do a Greater than search.

-Drive around the next couple of turns until you reach the next set of Thwomps. Do an Equal to search.

-If you were lucky (like me), the amount of search results will have been reduced drastically. Look at your results. If you checked the search menu while your racer was full size, one of the results will have a value of 0100. In this case, the address with that value is 03003D2E. So your raw, unencrypted code is 03003D2E 00000100, if you want to give people a code where your character is full size. That's pointless, so give the code a smaller value, like 000000C0 or something. Now we end up with 03003D2E 0000C000 (you must put most significant byte first, if I recall), which makes your racer 3/4 their normal size. Now punch that into AR Crypt and encrypt as you normally would.

-Sure, we have THAT code, but that's just the code for the X coordinate! Since GBA games are 2D (even if the game has 3D-style graphics, it's still 2D), there are two codes involved in a "complete" size modifier: the code for the X coordinate and the code for the Y coordinate. In this how-to, we have hacked the X coordinate, so let's hack the Y coordinate.

-One way to do this is by viewing the memory. So, in VBA, while your character is still on the screen, go to Tools > Memory viewer. Make sure the radio button next to "16-bit" is filled in. Type the address of the X coordinate (03003D2E) into the box next to "Go" and then hit "Go." It will take you to that part in the memory. If your character was full size when you did this, the view will look something like this...

03003D2E	0100 0100 0002 0000 0000 0000 0000 0000	.....
03003D3E	0000 0000 0000 0000 0000 0000 0000 0000	.....
03003D4E	0000 0000 0000 0000 0000 0000 0000 0000	.....
03003D5E	0000 0000 0000 0000 0000 0000 0000 0000	.....
03003D6E	0000 0000 0000 0000 0000 0000 0000 0000	.....
03003D7E	0000 0000 0000 0000 0000 0000 0000 0000	.....
03003D8E	0000 0000 0000 0000 0000 0000 0000 0000	.....
03003D9E	0000 0000 0000 0000 0000 0000 0000 0000	.....
03003DAE	0000 0000 0000 0000 0000 0000 0000 0000	.....
03003DBE	0000 0000 0000 0000 0000 0000 0000 0000	.....
03003DCE	0000 0000 0000 0000 0000 0000 0000 0000	.....

See those "0100"s in the upper-left? The first one is the X coordinate size (which you just hacked) and the second one is the Y coordinate size. In this case, the address for the Y coordinate is 03003D30, just two addresses higher

than the X coordinate size.

This is usually the case: the X coordinate and Y coordinate size modifiers are close like this. So if you find one, you've practically found the other.

Let's do a general recap:

-Start the search process and move somewhere else (or even change the area you are in, if you're hacking a code for your character). Do an Equal to search.

-Make the object get squished, smashed from the sides, or shrink. Do a Less than search. (If you can make the object get larger, do a Greater than search.)

-When it reverts/grows to normal size, do a Greater than search. (Less than if the object had grown.)

-Move around a bit and/or change other stuff and do an Equal to search. Do NOT make the object you want to modify get destroyed or go to an area where that object is not present.

-If you don't have a manageable number of results (less than 20), repeat.

-Look for codes with the value 0100. Check the area of memory around that/those code address(es) to find the size modifier for the other coordinate, if it didn't show up in your search.

-Encrypt the code(s) as needed.

Now what if the object you want to hack DOESN'T change shape? Then what?

Known Value Method:

This method is good for hacking size modifiers for objects that do not normally change shape. For this example, I'll be hacking a size mod for one of the computer racers, again in Mario Kart: Super Circuit.

-Load the ROM and select a new file.

-Select "Mario GP," any cup, any racer, and any track.

-Now, when the characters are waiting behind the finish line, open the cheat search menu, and do a 16-bit search for 0100. (I'm assuming you know how to configure it to your liking, since I'm assuming you followed along in the previous section of this guide.)

-You'll get a load of results. Check the results for a pair of addresses in which the second address is two addresses higher than the first (1st address + 2). There are several, so start testing them out, using the value C000.

-If a code doesn't work, go back to your results and start checking again. If any values are NOT 0100 or C000, search for 0100 again to get rid of them.

-Elimination will eventually lead you to 03003ECE, which is the X coordinate size modifier for the first CPU player. You can check out the memory to get the code for the Y coordinate, but here, you don't have to. The very next result is 03003ED0, which just so happens to be two addresses higher than the X coord code. Now we have the complete code set for CPU 1.

-Encrypt your code(s) as needed.

To recap, all you have to do is just search for 0100 (256 in decimal). As of now, this is the only value I have seen used for GBA size mods, but I have a feeling that programmers may use different values in the future.

So that's how to hack size modifiers for your Game Boy Advance games. Be warned that not all of them will be as easy to hack as they were in Super Circuit - I just used Super Circuit as an example because it's easy to hack, and I feel that the best way to learn to hack (aside from example) is to hack

an easy code/game first. Three closing tips:

-Some objects are seemingly "unhackable," so you feel that you may be forced to give up hacking a code. Don't let this discourage you.

-Some games, such as racing games, may have "patterns" that you can follow. In Super Circuit, for example, you can add 1A0 to each memory address to get the address of the next racer's size mod.

C) Finding GameBoy Advance Enabler Codes
- By Parasyte (Used with Permission)

Required Software:
GameBoy Advance ROM to hack (find on your own)
Mappy Virtual Machine - www.bottledlight.com/
GameShark Advance Code Encryption Program

GameBoy Advance Enablers are comprised of two parts, the first part is the hook routine. It overwrites some assembly in ROM/RAM with a jump to the GSA code handling routines. This jump must be executed many times per second. So a good place to hook would be the controller reading routine.
The Second part is an ID Code, it's not needed, but it's good to have if you like the GSA auto-detecting your game.

Hacking the hook code:
(this method only works on Thumb assembly in ROM)

1) Load up the rom in Mappy, then goto View -> Disassembler
2) Goto File -> Export -> Disassembly
3) Pick a destination txt file to save
4) Near the bottom you see three text boxes, leave the first one blank, type 08000000 into the second, and 200000 into the third
5) Select "Thumb" in the drop down list box
6) Click the Add button
7) Click the OK button, and wait a few seconds to a minute for the disassembly to complete.
8) Open the new txt file in a text editor, the file will be around 7MB
9) Text search for 04000130
10) You may find more than one place with this info, so be sure to write down ever address with that text string.
11) Going through all addresses, find two opcodes that might work. They can be anything BELOW the "04000130", and ABOVE any opcode beginning with a "B". The two opcodes you choose must be right next to each other, and they cannot have an "=" sign in any part of the line. Example of USABLE opcodes -

```
080002D2 add r1, r4, #0
080002D4 add r1, #24

080002DE ldrh r0, [r0]
080002E0 mvn r0, r0
```

Example of NON-USABLE opcodes -

```
080002E8 bne #$080002F2
080002EA ldr r0, =$03000540
```

12) Once you find an address that can be used, put the "F" code type on it, and use the value 00000101. Encrypt the code, and try it with any known codes, just to make sure it works. Example: F80002DE 00000101
13) If the game crashes, try using a value of 00000102. If the game still crashes, you will need to find another area to hook. The easiest method to do this envolves running the game in an emulator (VisualBoy Advance-SDL) and halting the game anywhere during execution. You should almost always end up in a piece of ASM that is executed many, many times. So just use the rules above and find a new address to use.

Hacking the ID Code:

1) Load the ROM into a hex editor
2) Jump to address 000000AC
3) Copy the 4 bytes at this address, then reverse all bytes. Example: "41424344"

becomes "44434241"
4) Encrypt this as an address with a value of 001DC0DE using the Game Shark Advance encrypter. Example: 44434241 001DC0DE
5) GameShark Advance should autodetect the game with this code in your(M) code

D) Creating AR V3 Codes Using AR Crypt - by Kenobi

{Special Acknowledgement: This program is based on the original AR Crypt algorithm by Parasyte.

You will need the AR Crypt program by Kenobi.
Get it here
GS Central is currently the only authorized site to host this program.
As such, neither the program creator or GS Central will not
be held responsible for un-authorized web sites carrying this program.
Authorization to host this ptogram is soley up to the program creator and
that authorization is final and binding.

Check "Expert Mode" in the Main menu, and check "AR V3" in the "Create..." Menu.

You can either choose the "Code Type" in the combobox or you can enter the "numbers separated by periods" found in this document in the "Data Size", "Code Type", "Code Subtype", "Special" and "Unused" spinedit boxes.

Then Type in the 1st line of the left memo your RAW codes on ONE LINE with SPACES between the 32bits numbers.

Press the "Create" Button.

Example

Example

**11) 2 Lines Special Codes (= starting with '00000000' and padded (if needed)
with "00000000").**

To add the 0s automatically, check the "pad with 0s" checkox of ARCrypt.

Type z09
1.2.0.x.x
XXXXXXXX : (02024EA4 -> 14224EA4)
0000ZZZZ : Writes Halfword ZZZZ to address XXXXXXXX."

You can either search the "Type z09" code type in the combobox, or you can do everything manually :
-set "Data Size" to 1, "Code Type" to 2, "Code Subtype" to 0, "Special" to x (=any value, for exemple 0) and "Unused" to x (=any value, for exemple 0), and finally check the
"pad with 0s" checkbox.

Type in the left memo your RAW address (for exemple "02024ea4") and your value
(8 digits, padded with 0es at the start if needed ; for exemple "00001234").
(That gives 02024ea400001234).

Now press the Create button. Ar Crypt will compute the ARV3 Raw (= unencrypted)
code, will display it on the 2nd line of the left memo, and will encrypt it in
the right memo.

Left memo :	Right memo:
02024ea0 00000001	
00000000 12224EA0	A81467EF C2D5BB2A
00000001 00000000	7FD49BC1 DBCF6C10

The good ARV3 Code is :

A81467EF C2D5BB2A
7FD49BC1 DBCF6C10

(One day, AR Crypt will pad the codes with 0es automatically. Until then,
you'll have to do it manually).

WARNING!!!!!!

All addresses obtained by decrypting V3 codes are in V3 RAW Format.
Let's say the decrypted V3 code is : XXXXXXXX YYYYYYYY

"Data Size" = (XXXXXXXX >> $19) AND 3
"Code Type" = (XXXXXXXX >> $1B) AND 7
"Code Subtype" = (XXXXXXXX >> $1E) AND 3
"Unknown 2" = (XXXXXXXX >> $18) AND 1
"Unknown 1" = (XXXXXXXX >> $12) AND 3

"Unknown 1" seems to never be used. It's not even calculated by the AR.
The data is just trashed. But it isn't useless. It might be used to create
codes that will have some kind of signature, so you might know if someone steal
your codes...

Morevover, you an also change the "x" to any allowed value to create even more
"signatures".

For exemple :
0.0.0.x.x will give you 4*2=8 different encryptions for the very same code.
x.1.0.x.x will give you 4*4*2=32 different encryptions for the very same code.

Then, to get the the real RAW address, you need to do :

XXXXXXXX = (XXXXXXXX AND $01FFFFFF)
RAW address = ((XXXXXXXX << 4) AND $0F000000) + (XXXXXXXX AND $0003FFFF)

E) Tutorial: Hacking Non-Standard Master Codes
this is several posts from the gscentral vb - courtesy parasyte

---

There are a few GBA games out there for which a master code cannot be found by
any of the 'standard' methods, such as using Kenobi's AR Crypt. The two
examples I will cover here are Baulder's Gate: Dark Alliance, and Phantasy
Star (a part of Phantasy Star Collection).

Baulder's Gate is an easy one. Just about anyone with the ROM and VBA-SDL can
hack the master code for it. Not a whole lot of assembly knowledge is required.
You just have to know what to look for.
If you load the game in AR Crypt and attempt to search for a master code,
you'll come up empty handed (be sure to use on the latest version of AR Crypt).
So to make our master code, we'll have to use a process which is a bit more
involved.

Load up the game in VBA-SDL, and get into the gameplay part of the game. (By the
looks of the damn intro, I thought it was an interactive novel, bah! Developers
need to stop forcing lame intros like that upon us gamers.) Once you are in the
game, and ready to go, open the ROM in a hex editor. Run a hex search for
"0047C0460847C046" -- this is the beginning of what I like to call the "Long
Branch Routines." Almost every game will have one, so take notes!
I found the long branch routines at file offset 006A9888. This, of course,
translates to GBA ROM address 086A9888. Knowing this, go back to VBA-SDL and
press F11 to bring up the debug console. In the debug console, type "dt
086A9888" This will disassemble the long branch routines into thumb assembly.
All games that have these long branch routines will look exactly the same, so
you probably won't ever have to disassemble it again. Just notice where all
the 'bx' instructions are:

086a9888 4700 bx r0
086a988a 46c0 mov r8, r8

```
086a988c 4708 bx r1
086a988e 46c0 mov r8, r8
086a9890 4710 bx r2
086a9892 46c0 mov r8, r8
```

Etc! You can easily spot the pattern. The bx instructions will rest on a 4-byte boundary, followed by 'mov r8,r8' which is the official Thumb instruction for NOP. (No Operation)
Anyway, these bx instructions are what you will use to find a long branch type master code. Let's start with the first one - 'bx r0.' Set a thumb breakpoint on that address using the command "bt 086A9888" then use the command "c" to continue running the game.
What we are looking for is one of these bx instructions which execute many times per second. After I set the breakpoint on 'bx r0,' I did not get any hits when I ran the game. So this one will not do! Press F11 again, and delete the breakpoint using command "bd 0". Now set a thumb breakpoint on the next bx instruction using command "bt 086A988C" and run the game with the "c" command.

Now here's where it starts to get interesting. Immediately after running the game, my breakpoint was hit, meaning the 'bx r1' instruction is used. Now what we want to do is copy down the value of the r1 register as reported by VBA. (We copy the value of r1, because we set a breakpoint on the 'bx r1' .. see how that works?) The register used by the bx instruction will always contain an address at this point. The bx instruction is used to branch\jump to the address contained in that register.
As of now, the r1 register contains value '03003ef8' which we know is a RAM address. I just temporarily wrote that into notepad, and now I am going to just run the game again, and let the breakpoint pause it once more. This time, I get the same address... I've already written that address down, so I will continue running the game (at least 20 times, if needed) to see if any other addresses are run.
About 5 runs later, r1 contains '0869fc05' -- a ROM address. This is a good start. I'll write this address down and simply continue running the game quite a few more time, recording every address, until I believe I have written all of the different addresses into notepad.

And here is my list, so far:
```
03003ef8
0869fc05
0869e145
02000634
02000514
0869e3d5
0869e0d9
```

Now that I believe I have all of the addresses which are used here, I will delete the breakpoint using "bd 0" and run the game. Then I'll move on to another area... Since I'm in the tavern, I have to go into the cellar.
Once there, I'm going to push F11 and set the breakpoint on the 'bx r1' instruction again, and repeat the process of writing down addresses. Make sure you keep your list of addresses clearly seperate.

Now I've got this for my second list:
```
0869e3d5
0869d5a1
0869d165
0869c881
03003ef8
0869fc05
0869d81d
0869cf61
```

What you want to do at this point is find addresses between each list that match. So based on my lists, here are the common addresses so far:

```
03003ef8
0869fc05
0869e3d5
```

Now, just to be safe, I am going to continue through the cellar and kill everything so I can exit the tavern.
Once I'm outside, I'll set the breakpoint again, write down more addresses,

and update my common addresses list... And funny that! My list of common addresses matches exactly the address list I made while outside of the tavern. Good enough, let's make us a master code!

We've got only three common addresses, as listed above. To make the master code, we need to search for these addresses in the ROM. When searching for 32-bit values in a GBA ROM, you have to 'byte swap' the value, and search for that. As an example, the first common address is '03003ef8.' If we split that into bytes, we get '03 00 3e f8.' Then we reverse the order of the bytes to byte swap it, and we get 'f8 3e 00 03.' Now search for 'f83e0003' in the rom. Well, our first minor snag. This pointer is not present in the ROM at all. So remove that address from the list, and move on.
Searching for the next pointer (05fc6908) turns up four possible locations. Our second minor snag -- any one of those four could be the correct one, or they could each be used for four different parts of the game. This is not good. For the sake of simplicity, set that address aside, and use it only in the emergency that our 3rd address is also no good.
Searching for the third pointer in the rom (d5e36908) we are left with a grand total of only one possible location! Success! We now have the location of our master code. The pointer rests at offset 007EF48C. So attach the normal 'C4' master code codetype to it. Then append a 'value' of 000084x3 to it, for a standard long branch type master code, where 'x' is the register number that is used by the bx instruction. In this case, it would be 1, for r1.

Baulder's Gate: Dark Alliance
(m)
C47EF48C 00008413

Pretty simple, eh?
If that third common address had been no good, you could have either gone back to test all 4 possibilites from the second pointer, or just move on to 'bx r2' and record the addresses from the r2 register. Usually after about r7, they stop getting used so often.

That about does it for now. I'll continue the tutorial with Phantasy Star later tonight, or tomorrow some time.

---

rrr. Getting back on track, here's how you hack a master code for a game like Phantasy Star within Phantasy Star Collection.

The first thing I'm going to say is that Phantasy Star Collection contains four executables. So this is one reason that hacking this particular game is so difficult. You got three executables -- one for each of the three games within the collection -- and the fourth executable is the initial loader\game selector.
The loader executable does not need any hacking, and two of the three game executables have easily reached master code locations. But, after doing a little research, it's plain to see that the Phantasy Star executable lies at ROM address 08738000. The problem here should be obvious; a standard bl-type master code absolutely CANNOT be used on any address above 0840001C. It is an impossibility.

If you begin to hack the Phantasy Star master code like we did with Baulder's Gate, you run into a HUGE problem right off the bat.... There are no long branch routines in the Phantasy Star executable. Bummer.
What we do to resolve this issue is take a completely different approach. We will start with the interrupt handler. Interrupts will occur several times per second, so they are perfect for our needs.
To find the interrupt handler, load up Phantasy Star. The title screen will work just as well as anywhere. If you're using VBA-SDL, (my emulator of choice for master code hacking) bring up the debug console, and use the 'mw' command to display the contents of the interrupt vector. The GBA interrupt vector is always at address 03007FFC, so use command "mw 03007FFC". The first value displayed will be [should be] a pointer -- this is the pointer to the interrupt handler. Mine is displaying '020207d0.'
This is an ARM address. You can tell, because the pointer is an even number. An odd number means it is a Thumb address. (For example, if it where

'020207d1' that would indicate Thumb assembly) Disassemble to ARM, starting at 020207d0. (Command "da 020207d0") And search for the first bx instruction you can find. Here is what I found:

```
02020884 e59f1010 ldr r1, [$0202089c] (=$02000070)
02020888 e0811002 add r1, r1, r2
0202088c e5910000 ldr r0, [r1]
02020890 e12fff10 bx r0
```

This assembly loads pointer 02000070 into register r1. Then it offsets the register by adding the value of r2 to it. Finally, it loads a pointer from the resulting address into register r0, and branches to the pointer in r0 using the bx instruction. This is a prime example of a "jump table." A jump table is a fairly large table of pointers stored in RAM. One of the pointers will be loaded from the table, and that pointer is jumped to, hence the name: jump table.
So let's start by dumping the jump table from address 02000070 using the command: "mw 02000070"

```
02000070 087384a1 0873a9fd 0873aa01 0873a9fd
02000080 0873a9fd 0873aa21 0876ca91 0873a9fd
02000090 0873a9fd 0873a9fd 0873a9fd 0873a9fd
020000a0 0873a9fd 0873a9fd 00000000 00000000
```

Looking good, eh! You can tell these are all ROM pointers, so maybe one of these will be something of importance that can be used as a master code. All of these pointers are odd numbers, and as we know, that means they are all pointers to Thumb assembly!

This jump table contains a total of five different pointers:

```
087384a1
0873a9fd
0873aa01
0873aa21
0876ca91
```

Let's disassemble a little bit from each of these addresses, and compare notes later. (Notice! Before you will be able to disassemble, you must 'convert' each pointer to an even number. Do this by simply subtracting 1 from the pointer. As an example with the first pointer, you will disassemble is using command: "dt 087384a0". Then the next using "dt 0873a9fc")

```
087384a0 b5f0 push {r4-r7,lr}
087384a2 4a2b ldr r2, [$08738550] (=$020004e8)
087384a4 482b ldr r0, [$08738554] (=$04000130)
087384a6 8801 ldrh r1, [r0, #0x0]
087384a8 4b2b ldr r3, [$08738558] (=$000003ff)
087384aa 1c18 add r0, r3, #0x0
087384ac 4041 eor r1, r0
...


0873a9fc 4770 bx lr


0873aa00 4904 ldr r1, [$0873aa14] (=$0400000c)
0873aa02 4a05 ldr r2, [$0873aa18] (=$00004409)
0873aa04 1c10 add r0, r2, #0x0
0873aa06 8008 strh r0, [r1, #0x0]
0873aa08 3908 sub r1, #0x8
0873aa0a 4a04 ldr r2, [$0873aa1c] (=$00007028)
0873aa0c 1c10 add r0, r2, #0x0
0873aa0e 8008 strh r0, [r1, #0x0]
0873aa10 4770 bx lr


0873aa20 4770 bx lr


0876ca90 4906 ldr r1, [$0876caac] (=$02000052)
```

```
0876ca92 8808 ldrh r0, [r1, #0x0]
0876ca94 2800 cmp r0, #0x0
0876ca96 d008 beq $0876caaa
0876ca98 8808 ldrh r0, [r1, #0x0]
0876ca9a 3801 sub r0, #0x1
0876ca9c 8008 strh r0, [r1, #0x0]
0876ca9e 0400 lsl r0, r0, #0x10
0876caa0 2800 cmp r0, #0x0
0876caa2 d102 bne $0876caaa
0876caa4 4902 ldr r1, [$0876cab0] (=$02000054)
0876caa6 2001 mov r0, #0x1
0876caa8 7008 strb r0, [r1, #0x0]
0876caaa 4770 bx lr
```

Now then... The second and fourth addresses only contain a "bx lr" instruction, meaning both routines simply return, doing absolutely NOTHING. I would not trust either of those. Of the remaining three, the first one looks very familiar... it's a joypad read routine! This is VEEERRRRRRY promising. So let's take the pointer to this joypad routine (087384a1) and do the byte swap trick. Hex search the Phantasy Star Collection rom for the value (a1847308), and wouldn't you know it! There's only one match, and it is definitely inside the Phantasy Star executable.
The pointer to the joypad routine is located at offset 007383FC... This means we get to make us a master code and test it out. Attach the C4 codetype, and append the 000084x3 value! If you remember the ARM assembly we disassembled in the interrupt handler, the first bx instruction we came across was 'bx r0'. So of course we are going to use the value of '0' to replace the 'x' in our new master code. (Again, this 'x' is used to select which register was originally used to branch to the original address.)

Phantasy Star Collection
Phantasy Star (m)
C47383FC 00008403

If you give her a test, you'll find that she sure works like a champ. Another difficult-to-hack master code solved. Case closed.

P.S. You may be wondering why you can't just use the interrupt handler pointer that you pulled from address 03007FFC. This is a great question. When the interrupt handler is run, the BIOS is what loads the pointer to jump to. But, the whole reason for telling the master code which register is used to run the original address is an important factor, here. After the AR is finished doing it's thing, it MUST run the original address. If it does not, you'll be left with nothing more than glitches and crashing. When AR runs that original address, it does so by executing a bx instruction.... By having that bx instruction use the same exact register which the original used, you eliminate any possible register corruption.
This leads us back to the original question; why can't I just use the pointer to the interrupt handler? Simply put, the best explanation is because a bx instruction is NOT used to run the interrupt handler. So you would only run the risk of corrupting registers by doing so.

I hope you enjoy these technical explanations. They are fun to write, and educational for everyone! It's a win-win situation.

---

dlong: How many rom patches does the long branch master code use up?

Long branch type master codes only use 2 of the 3 rom patches allocated for master codes. What the long branch type master code actually does is replace the pointer at the master code address with a pointer to 08000021 -- the place where AR keeps it's code engine entry point.

F) Gameboy Advance ASM Tutorial - by Kenobi

The following is a hands on tutorial in using VBA SDL in hacking rom patch codes.
The game used for this lesson was IronMan for GBA.
There are two examples given:Item never decreases code and invincilibity.
This tutorial is used with permission of Kenobi who generously gave of
his time to prepare and walk thru this tutorial with macrox.

Game info: In IronMan there is a blue orb that gives him advanced blast powers against his enemies. There are two levels to this power. The first is like a giant ray blast which destroys most of his enemies in a direct line of action, while the second level destroys all the enemies on screen.

ASM GBA Hacking - 1st Part Example #1 - Item Never Decreases

- Blue Orb Quantity never decreases code.

What you will need :

VBA (normal).
VBASDL.
The rom file. (Neither the authors nor GSC support the method of obtaining this file).
A save state at the very start of level 03, with the blue orb on Iron Man.
The address of the blue orb quantity. [0202CCF4]
Optional but essential info: GBA Tech document.
http://www.work.de/nocash/gbatek.htm
You will need to study this document to be able to understand the call functions and thus make the most out of VBA SDL hacking.

- Load the game in VBA (normal). Load the save state, and remove ALL the cheats (even if they are off).
  Take the blue orb if you didn't took it already, then save the save state.

- Now load the game in VBASDL, load the save state, then press F11 to enter the debugger.

- Type "BPW 0202CCF4 2" in the DOS window to set a break on write on address 0202CCF4 for 2 bytes.

- Type "c" in the DOS window go go back to the game.

- Use the blue orb power (press L). The game will freeze because the break on write did happen.
 (the quantity of blue orb decreased).

- Look in the DOS window. You'll see that :

```
Breakpoint (on write) address 0202ccf4 old:00000001 new 00000000
R00=00000000 R04=00000000 R08=0202cb00 R12=00000001
R01=0202ccf4 R05=00000001 R09=0202cbd0 R13=03007ba4
R02=0202cb84 R06=0202cbd0 R10=0202cbcc R14=0802c5fd
R03=080365bf R07=0202cbcc R11=00000000 R15=0802c63e
CPSR=6000003f (.ZC...T Mode: 1f)
0802c63c  9004 str r0, [sp,#0x10]
```

- The address of the very last line is what we need : 0802c63c.

- Go back to VBA (normal). Make "Tools" then "Dissassemble", click the "THUMB" button, and enter the address we just found.

- You'll see a lot of stuff. The 1st line will be :
  0802c63c 9004 str r0, [sp, #0x10]

- Go up one line. You'll see that :
  0802c63a 6008 str r0, [r1, #0x0]

- This should be the line that changed the value of the blue orb quantity, and that triggered the breakpoint.

- We'll make sure this line is the one we are looking for.
  Go back to VBASDL, load the save state, and press F11.

- Type in "bt 0802c63a" (this will set a THUMB breakpoint on address
  0802c63a. That means when the asm code stored at address 0802c63a
  is executed, the game will freeze).

- Type "c", then use the blue orb power. The game will freeze.

- Type "mw 0202CCF4". You'll see a lot of numbers, the very fist one
  (top left) will be the blue orb quantity. It should be "0001"

- Type "n", then type again "mw 0202CCF4" (you can use the up/down arrows
  to select a line you already typed in). You'll see that the number has
  changed.
  Now, it's "0000". That means the line 0802c63a is where the game saves the
  new value of the blue orb when you use one.

- Now we have 2 choices. We could just remove the asm code of line 0802c63a, so
  that the game never updates the quantity of the blue orb.
  The THUMB asm code that "deletes" a line (it's called "noop" which should mean
  no operations)is 46C0.

  How did I know it? Well, look at the GBA tech document http://www.work.de/nocash/gbatek.htm
  click on "Pseudo Instructions and Directives".
  You'll see :

```
  THUMB Pseudo Instructions
  nop              mov r8,r8
  ldr Rd,=Imm      ldr Rd,[r15,disp] ;use .pool as parameter field
  add Rd,=addr     add Rd,r15,disp
  adr Rd,addr      add Rd,r15,disp
  mov Rd,Rs        add Rd,Rs,0       ;with Rd,Rs in range r0-r7 each
```

  Look at the 1st like.
  The document tells me that, in THUMB Asm, nop = mov r8,r8. I just followed
  what it said :)

  And that's how I use 46C0, because it means "MOV R8, R8".

  To check that everything will be allright, select "Tools" then "Memory
  Viewer". Choose "16 bits", then enter the address "0802c63c". Type 46C0
  (that'll patch the game "on the fly"), then use the
  blue orb within the game : it's quantity won't decrease.

  So the raw AR V1/2 code could be :

```
  6401631D 000046C0
```

- We could also try to find the line where the games decrease the quantity of
  blue orbs, and remove it.
  The only way in THUMB asm to decrease a number is to subtract 2 values. So we
  need to find a instruction
  that looks like "sub rx, ry" (x and y being 2 numbers) or "sub rx, #0xYY" (YY
  being a number).

- Go back to VBA (normal), and go up one line until you find the "sub"
  instruction (shouldn't be hard to find :)).
  If you create a code that NOOP this line, the quantity of blue orb will never
  decrease.

- Even funnier. We could change this line with an instruction that does anything
  else to the value !
  For exemple, we could put any number we want (between 0 and 255).
  Or we could increase the number instead of decreasing it !
  Let's look at it to have some fun... :)

- Lets say you want to set the quantity of blue orb to 255 ($FF) when the player uses it.
  1st, look at the precise instruction of line 0802c638 (where the "sub" instruction lies).
  You see : 3801 sub r0, #0x1
  This instruction means "Remove 1 from the value stored in r0".

- In asm, the instruction that put a number in a register is called MOV (=move).
  To set the quantity of blue orb to 255, we need to change the instruction to :
  mov r0,#0x255
  You could try to find how to create the MOV instruction in hexadecimal in the gbatech documents...
  But MOV is one of the easier instructions to learn, and you can find it very easily by yourself.
  To create a MOV in hex, do this :
  - Put a "2" at the start.
  - Then put the # of the register you want to write to. We want to write to register 0. That makes "20"
  - Now put the value you want to write (in hexadecimal). 255 is $FF. The hex code will be "20FF".

  To check that everything will be alright, select "Tools" then "Memory Viewer". Choose "16 bits",
  then enter the address "0802c638". Type 20FF (that'll patch the game "on the fly"), then use the
  blue orb within the game. The 1st time you use it it'll be "normal", then it'll be "powerful" (because it's
  quantity/power will be 255). The code is working :)
  (btw : the gfx will be messed up because the max value of the blue orb should be 2).

- Now let's say you want to add 1 to the quantity of blue orb each time the player uses it.
  We need to change the "sub" instruction to an "add" instruction. Look at the GBA tech document.
  Click on "THUMB Instruction Set", then click on "THUMB.3: move/compare/add/subtract immediate".
  You'll see that :

```
Opcode Format
  Bit    Expl.
  15-13  Must be 001b for this type of instructions
  12-11  Opcode
           00b: MOV Rd,#nn      ;move     Rd   = #nn
           01b: CMP Rd,#nn      ;compare  Void = Rd - #nn
           10b: ADD Rd,#nn      ;add      Rd   = Rd + #nn
           11b: SUB Rd,#nn      ;subtract Rd   = Rd - #nn
  10-8   Rd - Destination Register  (R0..R7)
  7-0    nn - Unsigned Immediate    (0-255)


ARM equivalents for MOV/CMP/ADD/SUB are MOVS/CMP/ADDS/SUBS same format.
Execution Time: 1S
Return: Rd contains result (except CMP), N,Z,C,V affected (for MOV only N,Z).
```

- Start to look at the "Opcode" lines. You have :

```
 00b: MOV Rd,#nn      ;move     Rd   = #nn
 01b: CMP Rd,#nn      ;compare  Void = Rd - #nn
 10b: ADD Rd,#nn      ;add      Rd   = Rd + #nn
 11b: SUB Rd,#nn      ;subtract Rd   = Rd - #nn
```

- This is what we need : ADD Rd, #nn. Rd will be R0, and #nn will be 1.

- Now open 2 windows calculator in scientific mode. Choose "bin" for the first one, and "dec" for the second one.

- Lets start to create the instruction :

```
  Bit    Expl.
  15-13  Must be 001b for this type of instructions
```

+ That means you have to type "001" in the first windows calculator (the 1st 0es won't appear).

```
  12-11  Opcode
           00b: MOV Rd,#nn      ;move     Rd   = #nn
           01b: CMP Rd,#nn      ;compare  Void = Rd - #nn
           10b: ADD Rd,#nn      ;add      Rd   = Rd + #nn
           11b: SUB Rd,#nn      ;subtract Rd   = Rd - #nn
```

+ We want the "ADD" instruction, so you have to enter "10" in the first windows calculator (it'll show "110").

```
10-8   Rd - Destination Register  (R0..R7)
```

+ Destination register will be R0 (which means "0"). So type "000" (for bits 10, 9 and 8) in the first windows
  calculator (it'll show "110000").

```
7-0    nn - Unsigned Immediate    (0-255)
```

+ Now enter the value you want to add in the second windows calculator ("1"), then click on the "bin" button.
  It'll show "1". So type "00000001" (for bits 7-0) in the first windows calculator
  (it'll show "11000000000001").

+ Choose "hex" in the 1st windows calculator, and it'll show "3001". It's the hexadecimal form of "ADD r0, #0x1".

  To check that everything will be allright, select "Tools" then "Memory Viewer". Choose "16 bits",
  then enter the address "0802c638". Type 3001 (that'll patch the game "on the fly"), then use the
  blue orb within the game : the more you use it, the more powerful it'll becomes. The code it working !

ASM GBA Hacking - 2nd Part Example #1 - Item Never Decreases

You'll need :

either to install MAPPY VM :

http://www.bottledlight.com/tools/mappyvm_0.9d.zip

or

to get a good hexeditor (I use Winhex).

- Now let's try to find something way more difficult : a code that will make the game think you always have the blue orb. Use the dissassembler of VBA (normal), and go back at the that was subtracting the blue
  orb value (0802c638).

- Scroll up the lines until you find an instruction "bl..." or "bx...".
  You'll find a "bl $08040c44" at line 0802c5f8.
  Look at the line just under this one : 0802c5fa.

- Use VBA sdl (load the game, the save state, press F11) and type "bt 0802c5fa".
  Type "c", then use the blue orb : the game will freeze.

- Now the tricky part. You know where you are, but you don't know where you came

from (and we need to find that out).
There are a lot of way to find out where you came from, so I will only explain the way that works for Iron Man.

- Press "n" and look at all registers until you find something interessing. After about 27 "n", you'll see "0202ccf4" in R1...! It's the address of the blue orb quantity!

- Now look at the line just up. (0802c634) :
441 add r1, r8

- That means the game computed the value 0202ccf4 by doing r1 = r1 + r8. Now look at the values of r1 and r8 for
the line 0802c634 : R01=000001f4, R08=0202CB00

- That means that 0202CB00 is something like the base address for all the blue orb quantities addresses, and 1f4
is the "modifier" than, once added to 0202CB00, gives the blue orb quantity address for this particular level.

*If you use MAPPY !

- Now launch MAPPY VM, and load the Iron Man rom with it.

- Then make "File", "Export dissassembly". Change "ARM" to "THUMB", enter "08000000" in the middle input box,
and enter "200000" in the input box close to "THUMB". Click "Add", then choose a destination
(click the "..." button), and finally click "OK". Now open the file that has been created in wordpad.

- Search "0202CB00". You'll find nothing :(... That's not a big deal.
Look back in VBASDL dos window.
Scroll up, until you see that R01 value is different to 000001f4, and look the line just above.

- You'll see that :

0802c630 21fa mov r1, #0xfa

0802c632 0049 lsl r1, r1, #0x1

- Translate $fa to decimal (gives 250), then go back to worpad, and search "mov r1, #250".

- You'll find :

```
0802B2EE        mov     r1, #250
0802C630        mov     r1, #250
0802CDD4        mov     r1, #250
```

- As you can see, we already know for the second line (0802c630). So the place where the game checks
that you have enought blue orb must be either near 0802b2ee, or 0802cdd4.

*If you use Winhex!

- Open the rom file in winhex.

- Search this hex value : "00CB0202" (= 0202CB00 reverted). You'll find nothing :(... That's
not a big deal.

- Look back in VBASDL dos window. Scroll up, until you see that R01 value is different to 000001f4,
and look the line just above.

- You'll see that :

0802c630 21fa mov r1, #0xfa

```
        0802c632 0049 lsl r1, r1, #0x1
```

- Take the 21fa and the 0049, and write them like this : fa214900. Seach this number in winhex.

- You'll find one at 2B2EE, one at 2C630 and one at 2CDD4. Add 08000000 to these numbers to convert them into addresses. That gives :

```
        0802B2EE
        0802C630
        0802CDD4
```

- As you can see, we already know for the second line (0802c630). So the place where the game checks that you have enought blue orb must be either near 0802b2ee, or 0802cdd4.

- Go back to VBASDL. Quit the game (press esc if you are in the game, or type "q" then "y" in the dos window).

- Reload the game and the save state, use the blue orb power (so you have none left), press F11,
then type "bt 0802b2ee", "bt 0802C630", "bt 0802cdd4", then "c".

- Use the blue orb power : then game will freeze, and you'll be at address 0802b2ee. As you don't have any
blue orb power left, that means we found the place where the game will check if you have one left...

- Press "n" 5 times. You'll see that :

```
0802B2EE 21fa mov r1, #0xfa

0802B2F0 lsl r1, r1, #0x01

0802B2F2 add r0, r4, r1

0802B2F4 ldr r0, [r0,#0x0]

0802B2F6 cmp r0,#0x0

0802B2F8 ble $0802b2fc
```

- The three 1st lines will "create" the value 0202CFF4 in R0.

- The fourth line load the value that is in the address stored in R0 to R0 (that mean it'll load the quantity
of blue orb you have to r0).

- The fith line compare this value to 0. And the last line will jump to address 0802b2fc if the result of
the comparison is "less or egal".

- So we have different choices to make the game think you always have the blue orb :

+ We could change the "ldr r0, [r0, #0x0]" to something like "mov r0, #0x1" or "mov r0, #0x2".
Then the result of "cmp r0, #0x0" will always be false... The only problem is that this doesn't let you choose
which power you use...

+ So I'll use a trick. I'll change the instruction that read the blue orb quantity with an instruction that writes
to the blue orb quantity. And as the content of r0 won't be destroyed by my instructions, it'll stay "0202CCF4",
and when the game will compare it to 0 (thinking it's the quantity of blue orb), it'll always find that it is
superior to 0, hence it'll think I always have blue orbs...

```
  The hard part is that you can't do much with only one asm instruction. You can
  only use something like "str Rx, [Ry, #0xZZ]".
  That means you can't choose the value you will write, and the address where you
  want to write HAS to be in in the registers
  (and btw in THUMB we can only access registers 0-7 directly!)
  Hopefully, this is the case : R0 has the address of the blue orb values.
  Now I need to find what value I'm going to write. Let's look at the registers
  R0-R7 (the only one we can use) for line 0802D2F4 :

        R00=0202ccf4 R04=0202CB00
        R01=000001f4 R05=00000000
        R02=00000200 R06=00000000
        R03=0802e615 R07=0202CBD4

  R00 is the address for the bue orb quantity for this level. We'll write there,
  but we can't use this as the value we'll write,
  because it'll change for every level (we won't have a stable result).

  R01 and R04 are "linked" to R00 (R01+R04=R00). So here too, we can't use them
  as the value we'll write,
  because it'll change for every level (we won't have a stable result).

  R03 could be safe (not change with the level). Same thing for R05 and R06.

  R07 seems close to R04 (R07=R04+$D4), which means it might change with the
  levels, and won't give a stable result.


+ We can use, as data : R03=0802e615, R05=00000000, R06=00000000. R05 and R06
  are 0es. If we write 0es to the blue orb
  quantity, the game will launch the 1st power. And R03 is >1, so writing this
  value to the blue orb quantity will make
  then game launch the 2nd power (I discovered this by testing the values).

+ So we'll create 2 instructions :

  str r5, [r0, #0x0] (that will write the value of register r5 at address r0+#0x0)

  and

  str r3, [r0, #0x0] (that will write the value of register r3 at address r0+#0x0)


+ Lets look at the GBA tech documents. Click on Click on "THUMB Instruction
  Set", then click on
  "THUMB.9: load/store with immediate offset". You'll find that :

Opcode Format
  Bit    Expl.
  15-13  Must be 011b for this type of instructions
  12-11  Opcode (0-3)
          0: STR  Rd,[Rb,#nn]  ;store 32bit data   WORD[Rb+nn] = Rd
          1: LDR  Rd,[Rb,#nn]  ;load  32bit data   Rd = WORD[Rb+nn]
          2: STRB Rd,[Rb,#nn]  ;store  8bit data   BYTE[Rb+nn] = Rd
          3: LDRB Rd,[Rb,#nn]  ;load   8bit data   Rd = BYTE[Rb+nn]
  10-6   nn - Unsigned Offset              (0-31 for BYTE, 0-124 for WORD)
  5-3    Rb - Base Register                (R0..R7)
  2-0    Rd - Source/Destination Register  (R0..R7)


Return: No flags affected, data loaded either into Rd or into memory.
Execution Time: 1S+1N+1I for LDR, or 2N for STR


+ We need to use the 1st opcode, "0: STR  Rd,[Rb,#nn]  ;store 32bit data
  WORD[Rb+nn] = Rd"


+ Open the 1 windows calculator, and click on "bin".


15-13  Must be 011b for this type of instructions
```

-> Type "011" (calculator will show "11")

12-11 Opcode (0-3)
  0: STR Rd,[Rb,#nn] ;store 32bit data   WORD[Rb+nn] = Rd

-> Type "00" (calculator will show "100")

10-6   nn - Unsigned Offset   (0-31 for BYTE, 0-124 for WORD)

-> Type "00000" (calculator will show "11000000")

5-3   Rb - Base Register   (R0..R7)

-> R00 is the base register : type "000" (calculator will show "11000000000")

2-0   Rd - Source/Destination Register (R0..R7)

-> R05 will be the source register. 5 is 101 in binary, so type in "101"
(calculator will show "11000000000101")

+ Now click on the "HEX" button, and the calculator will show "6005". This is
the hex value of the instruction :

  str r5, [r0, #0x0]

+ Redo these steps to create the second instruction, but at the end enter "011"
for R03.
  The hex value will be "6003", which is the hex value for :

  str r3, [r0, #0x0]

Now you just have to create the V1/2 raw codes using the address 0802d2f4 and
these values 6005 and 6003.

1st code :

6401597a 00006005

2nd code :

6401597a 00006003

We could also use a joker code to select which power to use. In GBA, all the key
events are stored
at address $04000130.

Go back to VBA (normal), open the memory editor, select 16 bits, and go to the
address 04000130.
Check the "Automatic Update" box.

The first number will be 03FF.

Now press start. The number will change to 03F7.
Let go start, and press select. The number will change to 03FB.

That means we can create our codes that way :

D4000130 000003F7 (= "If the value at 04000130 if equal to the 03F7 (= start
button pressed), execute the next code")
6401597a 00006005

D4000130 000003FB (= "If the value at 04000130 if equal to the 03FB (= start

```
button pressed), execute the next code")
6401597a 00006003


Here is the V3 version of it (it's working, I tested it myself!) :

4A400130 000003F7
00000000 1801597A
00006005 00000000

4A400130 000003FB
00000000 1801597A
00006003 00000000
```

Explanations on Type 6 ('rom patching') Codes

```
Start the game, and press L : nothing will happen, because you don't have blue
orbs.

Now press Start (twice to exit the menu). The code 6401597a 00006005 will be
executed.
Press L : Iron Man will always use his 1st power.

Here is how it works :
The code will make the AR write the values 0401597a and 6005 at the special
place in the action replay.
I'll call this place the 'rom patching register'.
Then a special chip, called the 'FPGA' (thanks to Parasyte for the info), will
"patch"
the address 0802b2f4 (=2*0401597a) with the value 6005.
That will replace the instruction 'ldr r0, [r0,#0x0]' with 'str r5, [r0, #0x0]'.

Now the trick with the Type 6 codes (or rom patch codes) if that they are always
active,
until another code "erase" them (change the value of the 'rom patching
register').

That's why if you press Select, then L, Iron Man will always use his 2nd power.

Because the code 6401597a 00006003 will write the values 0401597a and 6003 to
the 'rom patching register'.
It'll replace the old '6005'. That means that now the 'FPGA" patches the address
0802b2f4 wirh the value '6003'
That will replace the instruction 'ldr r0, [r0,#0x0]' with 'str r3, [r0, #0x0]'.


One of the conclusion of this last explanation is that Type 6 codes are really
unlike the other one.
They need to be excuted only ONCE to be active until something changes the value
of the 'rom patch register'.



Finally, here is something even better (only for v3) :

1st set of codes:
48400131 00000003
(insert your invincibility
 code, raw v3,  here)

2nd set of codes:
4A400130 000001FF
00000000 1801597A
00006003 00000000

3rd set of codes:
4A400130 000001BF
00000000 1801597A
00006005 00000000
```

Here is what will happen during the game :

1st set of codes:
If the 8 bits value at address 04000131 if egal to 03 (= if you don't press L or R),
the invincibility code will be enable.

2nd set of codes:
If the 16 bits value at address 04000130 if egal to 1FF (= if you press L),
Iron Man will use its 2nd power.

3rd set of codes:
If the 16 bits value at address 04000130 if egal to 1BF (= if you press Up and L),
Iron Man will use its 1st power.

I know this is a bit useless on a 'real AR V3', that should handle multiple rom patching codes.
But on an AR2 upgraded to AR3, that can not handle multiple rom patching codes, it can be really
useful.

Finally, it's kinda hard to make this exact codes for AR2, because AR2 doesn't handle the
'If the 8bits value...' type of code. Only the 16bits value.
So to make a code that is activated if you don't press L or R, you'll have to do a lot of codes,
like that :

D4000130 00003FF (if no button are pressed)
(invincibility code)
D4000130 00003EF (if only Right is pressed)
(invincibility code)
D4000130 00003DF (if only Left is Pressed)
(invincibility code)
...

That means you have to write one set of code for every key combinaison you want the invicibility
to be active... Usually, that means : no button, Up, Down, Left, Right, A, B.
But you could also create code for Up+A, Down+A, Left+A, Right+A, Up+B, Down+B, Left+B, Right+B...

For this exemple, you could have to write 30 lines of codes to have the desire effect.

And you could also want to write codes for Up+Right, Down+Right... Or Up+Right+A... !
That could make up to 64*2 lines of codes, for something that is handled in 3 lines with the AR3 !

It's really stupid... So AR V3 rules ! :)

note:
BTW for assistance in making activator codes, the value of the GBA buttons are this:
at address 04000130, just remember to replace the first zero with D, that is D40000130.
So for an activator calling A D40000130 000003FE. Here are the other values.

Button A - 03FE

Button B - 03FD

Button L - 01FF

Button R - 02FF

Start - 03F7

Select - 03FB

Up - 03BF

Down - 03EF

Left - 03DF

Right - 037F

No buttons pressed - 03FF

All buttons pressed - 0000

For other combos just use the normal VBA. Load VBA, load any game, halt game, open mem viewer, goto address 04000130 at 16 bits, choose auto update, then start pressing buttons and hold and watch the value change.

ASM GBA Hacking - Example #2 - Invincibility

Using Break Point Read (bpr) to find Invincibility (from IronMan GBA level 1)

We will assume the reader has a basic knowledge of hacking health codes in the normal code generator of VBA.
We will also assume we have found the code for infinite health for level 1 which is 020200ed 00000064. We will also assume the reader has already saved a slot for level 1 for ease on hacking the code at hand. (The reader will be required to reset the game in the normal VBA several times in the exercise to follow.)

Now it is told to the reader to take on faith that using a break on read is easier to find than a break on write. It is also to be made clear that a study of ASM thumb calls is mandatory to understand to get anywhere in using SDL VBA as a tool in hacking codes. This study is not done overnight but will be an ongoing journey which if done with dedication will reward the reader with the ability to hack some very cool codes.

The goal here is to find an address that will "kill" the call of IronMan losing health and replace it with an address that will trick the game into thinking nothing is happening to IronMan and hence becoming invincible. The reader is to understand that he will need to have both the SDL and normal versions of VBA on hand and will use both programs to hack the code shown.

To begin we drag the IronMan rom over to the SDLk.exe file. This will start the SDL VBA graphics screen and the Dos Window that will be used to type commands and view call results. A special adaptation that must be used in this example is SDL VBA 1.4.1 with bpr support. The authors will not support the reader in obtaining the rom and leave the reader to his own devices for doing so.

Next we enter the 1st level of the game...

So load the game, enter the 1st level, and press F11 to enter the debugger.

Now we type bpr 020200ed 1 (this is the address where the health is stored, and the health takes one byte.

Now type c and resume the game. Now let IronMan take a hit.
The game will freeze and yield the following info in the Dos Window of SDL VBA. "080399da 9200 str r2, [sp, #0x0]" which is called when IM is hit. (IM = IronMan)

Next open the normal VBA program now. Do not close the SDK VBA.
Then open the dis-assembler on the tool bar and go to address "080399da", and check the thumb button.
This address is the same we found when called in SDL VBA when IM got hit.
Leave the debugger window open.
Now also open the memory editor of vba (normal), select 16 bits and go to "080399da" address too.

Now look at the dis-assembler of the normal vba, and scroll up one line

you see "Ldr r2, [r4, #0x0]".
That's where the break on read happened.
And that's where the game loads your life from memory.

Now look back at SDL VBA, after the break on read

you see r0=0, r1=03007c90, r2=00006400 (= your life), r3=0, r4=020200ec
(= address where the life is stored in memory),
r5=02020ff4...

What we'll try to do is to make the program not load your life value as it would normally and replace it with another value. The way we'll do this is to take the value in r4, and put it in r2.
Value in r4 is the address where your life is stored, and will always be a "big number", so it should make you invincible.

So go back to the memory editor of normal vba, go to address 080399d8, and change the value to 1c22. This is the same as "mov (r2, r4)", which means it'll put the value of r4 in register r2 (instead of loading IM HP to r2).
Once it's done, play a bit of the game in vba normal...at this point you will observe that both IM and the enemy are invincible. The code is working, but not the way we want. That's because at this address, the choice between "enemy HP" and "IM HP" has already be done, and the game is just executing a "script" that remove the damages to the HP. So we need to go "back" to find the place where the choice between "enemy is hit" and "IM is hit" is done (and we'll try to make the game never choose "IM is hit").

Go back to SDL VBA, and type "last", now type "c", and get hit again you'll see a lot of registers now.

Look at the first R15 = 08000838

R15 is VERY important. It'll ALWAYS is the address of the VERY NEXT INSTRUCTION that will be executed. When you know R15, you know the address you are at.

So before going to 080399da, the game was something near 08000838 (I say somewhere near because the "last" call is a bit buggy. it doesn't give the real r15, but a value close to it).

So look at address 08000838 in the dis-assembler of normal vba.

Scroll a bit up until you find a "bl", a "bx" or a "mov pc, ..." and you'll see...
"bl $080399c8".
Ok, that's where the program jumps to 080399c8, which is near the place where the break on read happened.

Now scroll up in vba dis-assembler, until you see a "bl ..", a "bx .."
or a "mov pc, ...".

that will be "0800820 bl $08049880"

now look at the instructions between the 2 bl's
(between 08000820 and 08000832), and find any "cmp".
You'll see "0800082a cmp r6, #0x0", and after that "0800082c beq $08000830".
That means that the game will compare the value of R6 and 0. And if they are equal, it'll jump to the address 0800082c.

Now open the memory editor of vba normal, go to the address 0800082c (the address of the beq), and change the value to d100, also reset the game and reload level 1 save. This will change the "beq" (= if equal) to "bne" (= if not equal)).

So once you've changed the value to d100, play a bit the game, and see what happens. You'll see that neither you nor the enemies are invincible... This is not a good choice to set the code we want. Go back to vba dis-assembler, and go back to address 08000820 (the 2nd bl we found).

So do the same thing as before from 08000820, scroll up until you find a "bl ..." (or a "bx ...", or a "mov pc, ..."), then scroll down until you find a "cmp".

You will see a "cmp" at 0800080c

and under if, at 0800080e, you'll see "beq ...".
So go back to the memory editor, go to address 0800080e, and change
"d0.." to "d1.."
Remember to reset the game and then reload level 1. Now play the game and
observe that IM is now invincible but the enemies die. This is what we had
set out to do!
Although the reader would expect the hack is done and the code determined we
must advise that we are "almost there"

Because changing d0 to d1 isn't good enough. It only "inverts" the condition
("if equal" becomes "if not equal"). It could seem good, but it may create some
bugs (for example, if an enemy was hit, IM could loose its HP). I know it's not
the case, but you never know what can happen. We need to create something that
is "always true", so chances of bug will be 0%. That's what we'll do in the
following lines.
To be 100% sure the code is working, we'll do it another way.

above the "beq", you see :

lsl r0, r0, 18
then
cmp r0, #0x0"

what we want, is either the cmp r0, #0x0 to be always true, or to be always
wrong (we don't know that yet), so that the "beq" (=if equal) always
(or never) executes.
"lsl r0, r0, #0x18" is not an important operation. It'll shit the value of
r0 #0x18 bits to the left (00000001 becomes 01000000). That mean we can edit
it to create our code.
We'll start to change the "lsl r0, r0, #0x18" with "mov r0, #0x0" (we do that
to make the "cmp r0, #0x0" always true).

So go to address 0800080a in the memory editor, and change 0600 with
2000 (= mov r0, #0x0).

Note about creating a simple MOV code :
2 0 00 : 2 = MOV ; 1st 0 is the destination register (0 to 7);
2nd and 3rd zeroes are the values (0 to FF).

So 25FF is "mov r5, #0xff"
2365 = "mov r3, #0x65"
and 2000 = "mov r0, #0x0"

-Reset the game and reload level 1.

It should make you invincible, and only the enemies will die.

So we have your code : put 2000 at address 0800080a
And the "raw rom patch code" is :
0800080a 00002000

Question : why do we have to go up some lines in the dissassembler?
Answer : because, as you saw, the very first code we found
(with the break on read) makes the player and the enemies invincible.
That means, the choice between enemies HP and player HP is made before
this place So we have to backtrack from the place the break on read happened.
That's why we kept going up in the addresses but the search down the chain
is the same.
So the rule is to repeat the search inspection as noted until we find a stable
address to do exactly what we wanted in the first place. The reader might be
thinking that this is a lot of work. The motivation for using. This method is
simple. You see, in IronMan the health code changes in address from level to level.
This means you would have to hack a unique code for each level.
Tedious at best and boring at worst in actually using the codes to play the game.
The ASM method gets to the heart of the problem by finding the address that holds
the true call on IronMan health and this one code is good for all levels.
The added plus is that IronMan is more than with inf. health...the code make him invincible.
That is, HP never decrease!

We hope these 2 examples serve as a beginning for the reader to explore further
the endless possibilities and power of ASM hacking for GBA and indeed in

any ASM hacking process for any platform.

G) Hacking with the ProAction Replay/GameShark PRO 3.0 for Game Boy -submitted by Curly8od

How To Hack For Action Replay Pro For Gameboy
Table of Contents

Updates Since version 1.2
Added How To Hack Character mod,Cut Scene Mod And Activate In Game Cheats
Added another question in the F.A.Q

1.
Inf Lives

Step 1:Start the game and as soon as you can move press the button on
top of the action replay
Step 2:Select code gen
Step 3:In code gen select start generator
Step 4:Start the game and lose a life
Step 5:Press the button on the action replay. Go to code gen and select
less and a number will appear at the bottom
Step 6:Do step 1 again then select code gen
Step 7:Select greater than a number with maybe letters will appear.
They mean: 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8 9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15
Step 8:Keep on doing steps 4-7 until you get 10 or less
Step 9:Go to view cheats and write down the number of codes you got
closest to ten
Step 10:Try the codes

2.
Inf Ammo

Step 1:Start the game and as soon as you can move press the button on
top of the action replay
Step 2:Select code gen
Step 3:In code gen select start generator
Step 4:Start the game and use some ammo
Step 5:Press the button on the action replay. Go to code gen and select
less and a number will
appear at the bottom
Step 6:Do step 1 again then select code gen
Step 7:Select greater than a number with maybe letters will appear.
They mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8 9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15
Step 9:Go to view cheats and write down the number of codes you got
closest to ten
Step 10:Try the codes

3.
Inf Money

Step 1:Start the game and as soon as you can move press the button on
top of the action replay
Step 2:Select code gen
Step 3:In code gen select start generator
Step 4:Start the game and spend some money
Step 5:Press the button on the action replay go to code gen and select

less and a number will
appear at the bottom
Step 6:Do step 1 again then select code gen
Step 7:Select greater than a number with maybe letters will appear.
They mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8 9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15
Step 9:Go to view cheats and write down the number of codes you got
Closest to ten
Step 10:Try the codes

4.
Have An Item

Step 1:Start the game and as soon as you can move press the button on
top of the action replay
Step 2:Select code gen
Step 3:In code gen select start generator
Step 4:Start the game and as soon as you can move press the button on
top of the action replay
and go to code gen and select less and some numbers will apear
Step 5:Then start the game again and get the item that you want then
press the button on the
action replay
Step 6:Go in to code gen and select greater than. go to view cheats and
write down 15 codes
Step 7:Try the codes you wrote down

5.
Walk Through Walls

Step 1:Start the game and as soon as you can move press the button on
top of the action replay
Step 2:Select code gen
Step 3:in code gen select start generator
Step 4:Start the game and move one step then press the button on the
top of the action replay
Step 5:Select less then a number with maybe letters will appear. They
Mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8 9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15
Step 6:Keep on doing steps 4-5 until 10 codes or less are found
Step 7:Go to view cheats and write down how many codes were found
Step 8:Try the codes

6.
Capture The Pokemon You Want

Step 1:Start the game and as soon as you can move press the button on
top of the action replay
Step 2:Select code gen
Step 3:In code gen select start generator
Step 4:Start the game and get in a wild battle with the pokemon you
want then press the button on top of the action replay
Step 5:Go to code gen and select less then a number with maybe letters
will appear thy mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8 9=9 0=0 a=10 b=11 c=12 d=13
e=14 f=15
Step 6:Keep on doing steps 4-5 till you get about 10 codes found
Step 7:Go to view cheats and write down how many codes were found
Step 8:Try the codes

7.
All Weapons

Step 1:Start the game and get a weapon then press the button on top of
the action replay
Step 2:Then go to code gen and select start gen then start the game
again and get a different
weapon
Step 3:Press the button on the action replay and go to code gen and
select different
Step 4:Start the game again and get the same weapon as last
Step 5:Then press the button on the action replay and select different
and numbers with maybe letters will appear they mean
1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8 9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15
Step 6:Keep on doing steps 4-5 until around ten codes are found
Step 7:Then view cheats and write down how many codes were found

Step 8:Try the codes you wrote down!

8.
Bullet Mod

Step 1:Start the game and as soon as you can move press the button on
top of the action replay.
Step 2:Select code gen.
Step 3:In code gen select start generator.
Step 4:Start the game again and get a gun and fire it then press the
button on the top of the action replay.
Step 5:Go to code gen and select different.
Step 6:Start the game again and get a different weapon then fire some
bullets the press the button
on the top of the action replay and go to code gen and select different
and some numbers maybe letters will appear they mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7
8=8 9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15.
Step 7:Keep on doing steps 4-6 till around ten codes are found.
Step 8:Go to view cheats and write down how many codes where found.
Step 9:Try the codes!.

9.
Character Mod
Step 1:Start the game and as soon as you can move press the button on
top of the action replay.
Step 2:Select code gen.
Step 3:In code gen select start generator.
Step 4:Start the game again and change charcter and press the button on
the top of the action replay
and go to code gen
Step 5:Select same and some numbers maybe
letters will appear they mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8
9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15.
Step 6:Start the game and chose a different character then press the
button on the action replay
and go to code gen
Step 7:Select different and some numbers maybe
letters will appear they mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8
9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15.
Step 8:Keep on doing steps 4-7 until around 10 or less codes are found
Step 9:Go to veiw cheats and write down how many codes where found
Step 10:Try the codes!

10.
Cut Scene Mod
Step 1:Start the game and as soon as you can move press the button on
top of the action replay.
Step 2:Select code gen.
Step 3:In code gen select start generator.
Step 4:get in to a cut scene and then press the button on top of the
action replay and go to code
gen and select different. Some numbers maybe
letters will appear they mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8
9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15.
Step 5:Play the same cut scene and press the button on the top of the
action replay. Go to code gen
and select Equal. Some numbers maybe
letters will appear thay mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8
9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15.
Step 6:Dont play a cut scene and press the button on the top of the
action replay. Go to code gen
and select different. Some numbers maybe
letters will appear they mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8
9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15.
Step 7:Keep on doing steps 4-6 until around 10 codes are found.
Step 8:go to view cheats and write down how many codes where found.
Step 9:Try the codes!

11.
Activate In Game Cheats
Step 1:Start the game and as soon as you can move press the button on
top of the action replay.

Step 2:Select code gen.
Step 3:In code gen select start generator.
Step 4:start the game again and put a code on and press the button on
the action replay. Go to
code gen and select greater. Some numbers maybe
letters will appear they mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8
9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15.
Step 5:start the game again and don't put a code on then press the
button on the top of the
action replay and go to code gen and select less and Some numbers maybe
letters will appear they mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8
9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15.
Step 6:Keep on doing steps 4-5 until around 10 codes are found.
Step 7:Go to view cheats and write down how many codes where found.
Step 8:Try the codes!

12.
Text Mod
Step 1:Start the game and as soon as you can move press the button on
top of the action replay.
Step 2:Select code gen.
Step 3:In code gen select start generator.
Step 4:Start the game again and talk to some one then talk to someone
different and press the button on the top of the action replay
Step 5:Go to code gen and select different and some numbers maybe
letters will appear they mean 1=1 2=2 3=3 4=4 5=5 6=6 7=7 8=8
9=9 0=0 a=10 b=11 c=12 d=13 e=14 f=15.
Step 6:Keep on doing steps 4-5 untill you have 10 or less codes!
Step 7:Try the codes!

13.
Some F.A.Q

Q. What Dose A Bullet Mod Do
A. Changes The Type Of Bullets Like If I Had A Shotgun And I Wanted It
To Shoot Pistol Bullets.

Q. Where Do I Find The Numbers And maybe Letters.
A. At The Bottom Of The Screen In Code Gen Your Action Replay Will
Freeze While You Press The Button And Then Numbers With Maybe Letters Will Appear
At The Bottom.

Q. What do the numbers and maybe letters tell you?
A. How many codes where found.

Credits: Omegakid,Pichu
Written by Aaron
Nickname curly8od
Email aaronjod@hotmail.com
Version 1.3
Official site www9.50megs.com\curly8od

F) Online code porter

You can convert a code from one version of the game by knowing the difference in offsets one version of a game has from another version. Please see the FAQ section for more info on how to do this. GSCentral has an automatic code porter that the reader is welcome to try. The authors would like to thank Crocc for permission to use the code porter found at GSCentral

www.gscentral.com/port.pl

Section 3 : Reference

XIII) Downloads

Name	Description	Author
GSCC2k2	Game Software Code Creator 2002 supports both N64 and PSX Gameshark Pro, older PSX GS versions, and Caetla. Has tons of options including Breakpoints (N64).	CodeMaster
Float Convert	Used for converting 32-Bit hex values to IEEE-754 Floating Point and vice versa.	???
Patch Code Unpatcher v1.1	Expands N64/PSX Patch ('50') codes.	Viper187
Gameboy/Gameboy Advance Utilities		
ARCrypt Final 2_2	Gameboy Advance Gameshark & AR code encryptor/decryptor	Kenobi & Parasyte
CBA Crypt	Gameboy Advance Codebreaker code encryptor/decryptor	Parasyte
Gamecube Utilities		
GCN Crypt v1.2	Encrypt/Decrypt Gamecube Action Replay Codes!	Parasyte
GCN AR Code Type Helper	Puts codes into proper decrypted AR format for you.	Parasyte
Playstation 2 Utilities		
PS2DIS 0.99	PS2 Disassembler, used for finding codes via SLUS files	Hanimar
PS2 Code Decoder - Beta 2	Code Encryptor/Decryptor	iN tHE mIND
MAXcrypt v1.0	Encrypts/decrypts Action Replay MAX codes	Parasyte
N64 Utilities		
Official N64 Hack Utilities 3.2	The official PC tools for GS Pro 3.2	Datel
Official N64 Hack Utilities	The official PC tools for GS Pro 3.3	Datel

3.3		
N64 Utils Patch	N64 Utils patch that allows dumping ROM.	???
RAM Compare v1.2	DOS app for comparing RAM dumps that you can get from certain N64 emulators.	The Phantom
Byteswap v1.0	Can convert Project64 save state files to regular RAM dumps. They must be .pj (uncompressed) save states!	The Phantom
VTR Compare 0.99 Beta	Viper's Text Ram Comparer uses text format RAM dumps from Nemu64 for searching comparisons. This baby has options even GSCC2k2 doesn't :)	Viper187
RAM 2 Text v1.0	Converts normal RAM dumps to text format for use with VTR Compare.	Viper187
Cheater64 v2.1	This is about the best comparer for hacking N64 with emulators thus far. It supports RAM dumps and PJ64 save states.	Viper187
Niew	Assembler/Disassembler (DOS) typicly used for ASM hacking - COP1 instructions aren't supported.	Titanik
LemAsm	Assembler/Disassembler - Windows - supports COP1 Instructions	Lemmy
R3400i Documentation	Wanna learn N64 Assembly language?	N/A
Xploder64	Official upgrade/codelist Utils for Xploder64	Fire International
GSProN64Crypt	A program that can decompress the .enc (GS ROM) file included with the official utils.	CodeMaster
Goldeneye 007 Target Time Calc v1.0	Use to calculate values for Wreck7's 'target time modifiers' on Goldeneye.	Viper187
Nemu	Nintendo 64 emulator	nemu
Nemu INI (Alternate)	Unofficial INI file for Nemu	???
PJ64	Nintendo 64 emulator	PJ64
PSX Utilities		

Official PSX GS Pro Hack Utilities v3.20	The official utils from Datel for use with GS Pro v3.20 or upgrading from 3.0	Datel
Official PSX CDX Hack Utilities v3.20	The official utils from Datel for use with GS CDX	Datel
Xplorer Button Value Calculator v1.3	Calculates button values for Xplorer Joker Codes	???
Caetla	An *unofficial* cheat rom for flashing to older cheat device hardware. PC Comms Link required.	???
Undatel 3	A program that can decompress the .enc (GS ROM) file included with the official utils.	???
Sega Dreamcast Utilities		
DCCrypt	Encrypt/Decrypt Dreamcast GS codes	Parasyte

XIV) GameShark / GameShark Pro FAQ

Many parts of this text originally appeared on GS Central.
Revised 11-11-00

Quick Links:

What is a GameShark?

A GameShark is a cheating and game enhancement device made by Datel. Interact was a North American distributor for Datel. Interact is now a memory and MadCatz owns the Gameshark(tm). It is similar to the device called a Game Genie made by Galoob Toys, that were made for the NES, SNES, Genesis, and Game Boy systems several years ago. It can be used to cheat in any game, e.g. allows players to do anything from being invincible to having infinite lives to having unlimited amounts of money. The GameShark can also be used to do some cool tricks, such as alter the color of objects such as your gun in GoldenEye 007, or even give you all of the weapons listed under the "All Guns" cheat which would normally be unavailable in Multi-Player! Codes of this nature although are not cheating the game can be worth while as they make the game more fun and allow the player to actually, temporarily alter the game and in effect make different variation of the game as originally created. The first thing that you learn about the GameShark is that it can open many interesting possibilities with games.
It is also called a Pro Action Replay in the United Kingdom and GameBuster in Germany.

What is a GameShark Pro?

The GameShark Pro is a new device made by InterAct that can do what its' predecessor the GameShark can do, and also has the capability to allow you to create your own GameShark codes by using the built in code generator. It comes with a bonus 'Hack Like A Pro' tutorial video.

What is a GameShark Lite?

The GameShark Lite is in essence a GameShark PRO without the PC connector port. A good example is the GS PRO 3.0. There are some other differences from the standard GS PRO but these details are not available at the time this text was being updated.

GameShark Pro for PSX

Some of the features are:
Built-in Code Generator: allows you to create your own GameShark codes.
Memory Editor: allows you to view the memory of your game and search for text.
V-Mem: Virtual Memory holds up to 120 blocks worth of game save data.
Explorer: It can play game soundtracks and view hidden Full Motion Video(FMV) sequences. You can store GameShark codes on a standard PlayStation Memory Card.
It comes with a bonus 'Hack Like A Pro' tutorial video.

GameShark Pro For N64

Some of the features:
Built-in Code Generator: allows you to create your own GameShark codes.
Memory Editor: allows you to view the memory of your game and search for text.
Memory Card Manager: can copy game saves from a memory card or a N64 game cartridge. You can store GameShark codes on a standard N64 Memory Card.
It comes with a bonus 'Hack Like A Pro' tutorial video.

GameShark for Game Boy

Some of the features:
Built in Code Generator: allows you to create your own GameShark codes.

The Code Generator can only search for unknown values, not exact. There is no memory editor nor text editor. New added codes can be stored only in version 3.0 and higher (GameShark Pro), older versions do not store new codes. You must reboot game after every code search with versions 3.0 and less. The new 4.0 version works in the same manner as its big brother, the N64 GameShark Pro where you can resume play after you perform searches and therefore no game reboot is needed. You can also take "snapshot", which is a quick save state, that can be loaded later, or saved on your computer. This device works with Game Boy Pocket and Color. This device does not work with the Original Game Boy.

GameShark PRO CDX for PSX and Dreamcast

These devices are similar to the above in that they are cheat devices. They work by first loading the device by using a boot GameShark CD from which codes can be loaded prior to loading and running the game. They are somewhat limited as compared to the GS PRO non CDX versions above as in the case of the GS for Dreamcast as there is as yet no code generator available.

What is a Code Breaker?

A Code Breaker is a cheating/hacking device that is made by Pelican Accessories for use with Game Boy Pocket and Color. This device does not work with the Original Game Boy. This device is more complex than the GameShark in that it has more functions. It has a Code Generator that searches for unknown and exact values with no rebooting needed, stores and sorts new codes and game listings, uses Code Breaker and GameShark format codes, Slow Mo speed select, rumble pack built in, real time save and restore, ram saves, message writer, word scout to find passwords in games, memory and text editor, copy and erase game saves, supports Game Boy Printer and has more robust secure game holder.

What is an Xploder/Xplorer?

An XploderXplorer (the name varies by country, is another cheat/hacking device that is made by Fire International (Blaze USA). The Gameboy version has very similar properties as the Game Shark such as the exact and unknown code search generator. The unique property of the Gameboy Xploder is its ability to alternate between exact and unknown searches. The other plus it has is its ability to offer several warning as changes are made to its code database. A blessing for any of us who are too fast with the fingers. It can store 1 real game save at a time. The Gameboy version has no memory or text editor. The N64 and PSX versions have memory viewers but no memory or text editors. The PSX version has an available PC utilities program for advanced hacking and is similar to the Game Shark utilities program. Little wonder both the Game Shark and the Xploder are the brain child of Wayne Beckett

What are the differences between the Game Genie, GameShark, GameShark Pro & Pro Action Replay?

The Game Genie was manufactured by Galoob, the GameShark is manufactured by MadCatz, and the Pro Action Replay is manufactured by Datel. You had to keep inputting codes into the Game Genie every time you wish to use them and the number of codes you may input was very limited. Both the Pro Action Replay, the GameShark and the GameShark Pro have a battery-backup that can save codes. One major difference between the three device is evident in that only the GameShark Pro has a code generator that can search for codes in the games RAM, whereas the others cannot.

Are these products authorized by Nintendo?

The GameShark, GameShark Pro and Pro Action Replay aren't authorized, endorsed, or supported by Nintendo. Special thanks goes to Avid Gamer for telling me that Nintendo filed many lawsuits against Galoob trying to prevent the Game Genie from being marketed in North America. Galoob did win the court case and did go on to selling the Nintendo version. However, a interesting fact is that Sega did not oppose the Game Genie, and instead even gave it a official license.

Where can I get a GameShark? GameShark PRO? GameShark PRO CDX? Code Breaker? Xploder?

In the U.S. try stores such as Software Etc, Babbages, and Electronics Boutique and FuncoLand (GameStop). In Canada I've only been able to find them at Electronics Boutique. Expect to pay about $39.99 to 49.95 U.S., and that roughly translates to $69.99 Cdn. U.K. gamers can get a Pro Action Replay from Datel. Check the MadCatz and Pelican web pages for further information: http://www.gameshark.com/ , http://www.pelicanacc.com/ , www.blaze-gear.com/ or http://www.xploder.net/, and http://www.codejunkies.com/ .

I have a GameShark or GameShark Pro, now what do I do?

Place your game cartridge onto the Game Shark's top. You can easily tell which is the top by noticing if the GameShark sticker on the GameShark is readable. Then insert your GameShark (that has the cartridge placed onto it) into your system system and turn the console power on. If you are using a CDX GameShark, insert the GameShark, select any codes wanted, and choose start game. It should ask you to insert the game CD, which you should do. After pressing a button, the game should load.

Help! My GameShark or GameShark Pro isn't working now!

There are a few reasons why your GameShark didn't work. Either you didn't follow the procedure listed, or you have a placed a game into your N64 that requires a Key Code to boot the game. Also you may not have pushed the GameShark (or cartridge) down far enough for it to fit in snugly. There is a very slight possibility that your GameShark is defective. Also beware not to switch the power off and on quickly as the GameShark might freeze on the title screen, with the LED Indicator displaying an 8 all of the time. If this should occur, please try the following as it may help reset the GameShark.

First make sure the console is off. Then remove the GameShark from the console and then remove the game from the GameShark. Next insert the game into the GameShark and then the GameShark into the console. Turn the console power on and the GameShark should again begin its' boot up and count down. If the GameShark still does not boot try the following:

First make sure the console is off. Then remove the GameShark from the console and then remove the game from the GameShark. Now gently wipe the edge connector of both the GameShark and the game cart with a soft cloth to dissipate any static charge there. Next insert the game into the GameShark and then the GameShark into the console. Turn the console power on and the GameShark should again begin its' boot up and count down. If the GameShark still does not boot it could mean that it is waiting for a game with a required matching key code to be inserted.

Another thing to try is to again start with the console power off and with both the shark and game already seated, to pull ever so slightly upward on the shark to lift it SLIGHTLY from the console pocket. (Sometimes the console socket is too deep for the shark connector board and this helps establish the proper seat to get the shark booted.) Try turning the power on again.

Review your situation and act accordingly. If this is the first time you are using a new GameShark I would seriously advise returning it to the store where it was bought for an exchange for a new GameShark.

Another possibility is that you have somehow corrupted the GameShark BIOS. In this case you can try to reflash (reinstall the BIOS) the GameShark. You will need access to a working GameShark, you defective GameShark, Shark Link (PC link cable), Hacking Utilities Software(available on the MadCatz and Datel websites, and a PC. Next, goto the section below on how to connect the PC and GameShark up together.

What is the LED Indicator?

The LED Indicator is merely the number inside the small glass piece on the front of your GameShark that counts down upon boot-up.

What are KeyCodes?

Certain games such as Diddy Kong Racing, Yoshi's Story, 1080 Snowboarding, F-Zero X and The Legend of Zelda: The Ocarina of Time cannot be used normally like Mario 64 can with your GameShark. They have chips to block out illegal copying of the game, and also, unfortunately, prevents the GameShark from accessing them. You require a version 1.08 of the GameShark or higher (depending on which game) to use your GameShark with these games. Games such as The Legend of Zelda: The Ocarina of Time must be used with a GameShark 2.2 or higher due to the 32 bit write change of it's key code. There is a new 32-bit input on the key codes in GameShark V2.2+, this new 'block' is the essential instruction code that tells the GameShark, "this instruction is '80 20 10 00'!" When using any of the "original key codes", the instruction will be "80 20 10 00". When using the Zelda key code, the instruction will be "80 19 00 00". Since you cannot change the instruction code on GS 2.1 and lower, there is no way to add the Zelda key code(on these GameSharks, the instruction is always '80 20 10 00'). InterAct is used to replace older GameSharks up to version 2.1 for a 2.21 for free. This late in the game, though, your only options is to find a newer version for sale somwhere. Gamestop, EBworld, and Ebay would all be good places to try.

KeyCodes:

Gameshark Version 1.08
Diddy Kong Racing
1080° Snowboarding
Kobe Bryant In NBA Courtside
Banjo-Kazooie
MLB Featuring Ken Griffey Jr.
Ken Griffey Jr.'s Slugfest: 4E F8 4D D6 0A B3 D6 0A B8

Yoshi's Story
Cruis'n World
F-Zero X: Unknown

Gameshark Version 1.09
Diddy Kong Racing
1080° Snowboarding
Kobe Bryant In NBA Courtside
Banjo-Kazooie
MLB Featuring Ken Griffey Jr.
Ken Griffey Jr.'s Slugfest: 59 A6 31 F5 13 B3 DA 50 FA

Yoshi's Story
Cruis'n World
F-Zero X: 05 63 14 98 D5 E4 CF CD 1A

Gameshark Version 2.00
Super Mario 64 & Others: 6334 F161 A72C 201C 2E

Diddy Kong Racing
1080° Snowboarding
Kobe Bryant In NBA Courtside
Banjo-Kazooie
MLB Featuring Ken Griffey Jr.
Ken Griffey Jr.'s Slugfest: 50 F2 49 08 7C 07 EE 6C 25

Yoshi's Story
Cruis'n World
F-Zero X: 8D 9A 8C DA F5 F2 B6 07 92

Gameshark Version 2.10
Super Mario 64 & Others: EB 03 0C 2C D2 3A AF C3 CE

Diddy Kong Racing
1080° Snowboarding
Kobe Bryant In NBA Courtside
Banjo-Kazooie
MLB Featuring Ken Griffey Jr.
Ken Griffey Jr.'s Slugfest: 78 69 4F BD AC EF E9 DD 79

Yoshi's Story

Cruis'n World
F-Zero X: 85 A2 B3 44 44 4C F1 C1 E4

Gameshark Version 2.21
Super Mario 64 & Others: 3E 75 22 68 00 99 BE BE F6 - 80 20 10 00

Diddy Kong Racing
1080° Snowboarding
Kobe Bryant In NBA Courtside
Banjo-Kazooie
MLB Featuring Ken Griffey Jr.
Ken Griffey Jr.'s Slugfest
Paper Mario: 2C 48 29 16 D4 3E 90 61 47 - 80 20 10 00

Yoshi's Story
Cruis'n World
F-Zero X: 94 CB D4 8E 52 9A 30 89 E7 - 80 20 10 00

The Legend of Zelda: Ocarina of Time
The Legend of Zelda:Majora's Mask
Perfect Dark
Conkers Bad Fur Day
Jet Force Gemini: 14 A8 3B CA CD F8 11 BE 50 - 80 19 00 00

Gameshark Pro Version 3.0 and 3.1
Super Mario 64 & Others: 70 14 FF AB 1A 91 14 49 B4 - 80 18 00 00

Diddy Kong Racing
1080° Snowboarding
Kobe Bryant In NBA Courtside
Banjo-Kazooie
MLB Featuring Ken Griffey Jr.
Ken Griffey Jr.'s Slugfest
Super Smash Bros.
Paper Mario: 5B E5 5F CE 93 89 D7 11 9F - 80 20 00 00

Yoshi's Story
Cruis'n World
F-Zero X: 33 31 66 BD 04 ED E3 62 DF - 80 20 04 00

The Legend of Zelda: Ocarina of Time
The Legend of Zelda:Majora's Mask
Perfect Dark
Conkers Bad Fur Day
Jet Force Gemini: 56 72 19 E1 9D 62 82 28 C9 - 80 19 00 00

Gameshark Pro Version 3.2
Super Mario 64 & Others: AF FA 90 67 C2 49 22 D0 12 - 80 18 00 00

Diddy Kong Racing
1080° Snowboarding
Kobe Bryant In NBA Courtside
Banjo-Kazooie
MLB Featuring Ken Griffey Jr.
Ken Griffey Jr.'s Slugfest
Super Smash Bros.
Paper Mario: BD B8 AF 1A E9 C2 8B 3B 30 - 80 20 10 00

Yoshi's Story
Cruis'n World
F-Zero X: B6 F4 6A E1 8B 0F C8 AB 67 - 80 20 04 00

The Legend of Zelda: Ocarina of Time
The Legend of Zelda:Majora's Mask
Perfect Dark
Conkers Bad Fur Day
Jet Force Gemini: 85 87 29 C5 3A 85 F7 50 F0 - 80 19 00 00

Gameshark Pro Version 3.3
Super Mario 64 & Others: 8F 89 AB A0 C3 4C 26 10 A4 - 80 18 00 00

Diddy Kong Racing
1080° Snowboarding

Kobe Bryant In NBA Courtside
Banjo-Kazooie
MLB Featuring Ken Griffey Jr.
Ken Griffey Jr.'s Slugfest
Super Smash Bros.
Paper Mario: 95 AC 21 BE 58 B0 4E F6 A8 80 20 10 00

Yoshi's Story
Cruis'n World
F-Zero X: C4 6F 1B C2 6C 6C 1F 67 1D 80 20 04 00

The Legend of Zelda: Ocarina of Time
The Legend of Zelda:Majora's Mask
Perfect Dark
Conkers Bad Fur Day
Jet Force Gemini: A9 24 53 52 5F 73 77 37 7D 80 19 00 00

What's the GS Button?

It's the button on the front of the GameShark. It is fairly small, and makes an amusing clicking sound. It is used for one-time only codes, and injects the code into the RAM only once, instead of having a constant effect. For Pro Action Replay owners it is called the "PAR button".

How do I add new codes?

Please refer to the owners manual.

Can I make codes?

Yes, you most certainly can! Refer to the "How-to's" above.

What do these 'XXX' and so on represent?

You either place your own digits there (for 'XX' or 'XXX') or you have to insert the coding that represents a certain level for that game. If it is 'XXXX' then it can probably be replaced with some special digits, that accompany that code. Sometimes it can represent the quantity of the code, and for the GameShark 0 is the lowest amount, while F is the highest.

What is a Memory Card Manager?

It enables you to view the contents of any memory card up to 1998 blocks in size. You do need a v2.0 or higher GameShark to have this ability. Follow the simple instructions to save, load, and copy between memory cards that are there on your GameShark. You can even copy save games from your game cartridge onto memory cards.

What is a SmartCard/Port?

The Smart Card Port is a reader/writer slot at the back of the GameShark that will be able to read SmartCards when they are available. InterAct says that in the future the slot will support a range of SmartCards, and will offer features including special game cheats and memory card support. This includes Ram, Rom, and Flashy Rom cards. SmartCard are reported to be only supported in the GameShark for PSX and the GameShark PRO for both the PSX and N64. This function was discontinued on the 3.2+ version of the N64 GS and replaced with the 25 pin parallel connector port.

How can I get a newer GameShark?

This late in the game, the only thing you can really do is find a newer version for sale either online at Gamestop, EBworld, Ebay, etc. or at a local store that deals in games.

What is the difference between the GameShark Pro and the Shark Link Trainer?

The devices work on the same principle. The major hardware differences between them are: The Shark Link Trainer consisted of a PC Comms link board that was inserted into a ISA slot of your PC for PSX owners and for N64 owners an additional special adapter that was for all intents and purposes a "stand in game shark". This adapter was essential to use for N64 owners as the Game Shark for N64 lacks any parallel connector port as in the PSX version. N64 owners also had to purchase a 4 meg ram expansion pack for their N64 console and replace the standard jumper that comes with it. This memory upgrade was also essential as the N64 lacks sufficient memory to operate the Shark Link alone for storage of found codes and activated codes during use. Custom written software and drivers were then loaded on the PC and then used to perform various searches of the game ram to find a varied assortment of different code addresses depending on what was displayed on the screen and what codes were active in the program at that moment. In essence, the user was taking game ram or memory "snapshots" in order to find what code addresses were increasing, decreasing or remaining the same as the game progress. This allowed the user to "zero in" on any particular code(s) of interest from infinite health to finding the debug menu screen. The user was then allowed to activate these codes immediately for testing and alter the codes as desired and make them active. In addition, the user of the PSX version of the shark link could immediately upload these new codes into their game shark while the N64 version user had to go through the laborious task of inputting these new codes by use of the game controller. Rocket Games Inc., (part of Datel) use to sell the Shark Link.
The GS PRO is different in that it no longer requires the PC Comms link card as it has the code generator built right in. It still currently requires the additional memory expansion pack in order to do the maximum code activation allowed per game and it still retains the property of altering codes and making them active. The GS PRO has known and unknown code search capability as well as text search mode to find hidden screens in certain games. Previous shark link users will welcome the familiar screen command environment they have come to know while those new to the code hacking world will come to know a powerful ally they can use to find those codes they always dreamed of. The Game Shark PRO 3.3 is the new Shark Link and Datel, Interact and independent organizations such as Game Software Code Creators Club http://www.cmgsccc.com/ now have software that allows the Game Shark to be connected to your PC. Other cheat device manufacturers such as Fire International (Blaze) also have their own version of PC hacking utilities for their devices as well. The hacking utilities as it is known, allows a code hacker to use more advanced hacking features and options of the Game Shark.

What is required to connect my PC and Game Shark PRO for N64?

Materials You Need:

-GameShark Pro 3.1 - 3.3
-25 Pin Printer Cable (available at Radio Shack, or any other computer/electronics store. We recommend using IEEE-1284 compliant, high speed parallel cables such as from Belkin.
-Computer, with a functioning LPT1 port
-N64 Utilities (available for free at www.gameshark.com/)
WARNING! USA GS owners should not download the file from Datel as this version is made for AR PRO owners. The AR PRO works on PAL TV systems only while the GS PRO works on NTSC.

Step By Step Procedure:

Step 1: Check to see if you have a printer connected to your PC's parallel port. Disconnect the printer from the cable if you do. You need to do this to continue. If you do not have a printer than this connect one end of your new cable to your PC (while off) parallel port, 25 pin female. Connect to www.gameshark.com and get the latest N64 hacking utilities from their web site. Install them on your PC but do not run them yet.

Step 2: Boot your PC and get into the PC BIOS screen. This is usually allowed during the first few seconds during boot up and is invoked by pressing some key on your keyboard such as F1. Consult your owners manual for more info. Once in the BIOS screen you should check that you parallel port is using either EPP or ECP, NOT Bi-directional (the Game Shark will not work with this setting). Warning! You many find you can longer

communicate with your printer once you reconnect you printer to your PC If your printer required bi-directional communication.
If you change the port to ECP or EPP then you should save the settings and reboot.

-Step 3: Place the GameShark into your N64 console while off. Now plug a game into the Game shark. Connect the other end of your cable into the back of the Game Shark (25 pin male connector). Turn on the N64, after the short loading time
you should be at the GameShark's Main Menu. If you are not, read the GameShark's Manual.

-Step 4: Start the N64 Utilities you installed earlier. Go to the Screen that has the 'System Information' info. Click on the 'Detect' button. You should see an affirmative reply it found the console. If you see "console didn't respond, you possibly did not do step 1 and 2 correctly.

-Step 5: Assuming you got through to this step ok you can now explore the utilities. The following info is used with permission from james007.

-Code Generator: The Code Generator acts a lot like the Code Generator built into the GameShark itself. Using the one on your computer is a lot slower, since the LPT1 port is only a 25kb/sec port. Extra features to look out for are the 'In Range' button, which allows you to search for a value in range of what you specify. The 'Resume Last' button allows you to resume a search incase the game crashed (good for games with Anti-GameShark Chips). The 'Search Range' area allows you to specify where the Code Generator will search in the games RAM. I usually use 80000000 to 80100000, since most codes are in between these two addresses. The 'Search History' area allows you to see the history of what type of searches you did.

-Results (Code Tester): This page allows you to test codes you have found using the Code Generator, and to enter in new codes. This area is pretty self-explanatory.

-Code List: This page allows you to modify your GameShark codes on your computer. This is far faster than entering them in using your N64 Controller. The first step is to Download your codes onto your computer using the 'Download Codes From Cartridge' area. Select the 'Browse' button to pick a location where you want the codes to go. Then push the 'Download Codes' button to copy them to your computer. Now you can modify them using a text editor like Notepad. When you're ready to upload your new code list, do the following. Go to the 'Code List Compiler' area and push the 'Browse' button. Search for the code list you want to upload. Once found push the 'Compile Codes' button, N64 Utilities will now check the syntax of the code list. If it is good it will say it had no errors. If it found errors it will tell you what line had errors and what the error was. Fix the error before uploading your codes. If you now have no errors, push the 'Upload Codes' button in the 'Upload Codes To Cartridge' area. It will send your code list to your GameShark. You now have to turn off your GameShark so it can load them up properly.

-Other Utilities: This page has two useful things. An upgrade place, where you can upgrade your GameShark, and a video capture area, where you can take screenshots of your game. The screenshots are saved as Bitmap files (*.bmp), which is a Windows standard graphic file. The files can be easily converted to other formats with third party programs. The 'Cartridge Upgrade' area allows you to upgrade your GameShark to the latest version.
Selecting the Overwrite code lists & settings' option will do just that, when you upgrade, it will overwrite your entire GameShark, including all your codes and settings.

REFLASHING the BIOS of a defective N64 GameShark:

(DISCLAIMER! Neither the authors, contributors, Datel, MadCatz, Interact or its subsidiaries, or GSCentral assume any responsibility in consequence of the results of the this procedure. The authors do not and will not be

held liable for anyone attempting this procedure. The reader is advised to understand that this is a last ditch effort to repair a defective GameShark. Neither Datel, InterAct, MadCatz, nor GSCentral endorse this procedure.)

Proceed like above from steps 1 - 3 but with this one major change: After you seat the working GameShark in the N64 console you will seat the defective GameShark into the working GameShark and then the game into the defective GameShark. Connect the Shark link cable into the bottom (working) GameShark. (Caution! You should use a working GameShark that is of the same version you are going to reflash the defective GameShark as this procedure will flash both GameShark BIOS to same version. You can choose to NOT overwrite your code list in the GameSharks if you don't want to loose your current codes in the working GameShark. The authors suggest you backup your codelist at any rate for safekeeping. Once you have everything ready proceed to flash the BIOS by going to the upgrade screen and choosing the upgrade option. Once the operation is complete, you will have repaired the defective GameShark and both GameSharks will be the same version. Turn off the N64 and remove the GameSharks from one another and if desired plug the Shark link cable into the former defective GameShark and reload the code list. One of the authors (macrox) has tried this procedure several times without negative results.

REFLASHING the BIOS of a defective Game Boy 3.0 GameShark:

(DISCLAIMER! Neither the authors, contributors, Datel, Interact, MadCatz, or its subsidiaries, or GSCentral assume any responsibility in consequence of the results of the this procedure. The authors do not and will not be held liable for anyone attempting this procedure. The reader is advised to understand that this is a last ditch effort to repair a defective GameShark. Neither Datel, InterAct, MadCatz, nor GSCentral endorse this procedure.)

The reader may find the need to reflash the Game Shark 3.0 for Gameboy should it become corrupted during use. Corrupted can mean many things such as the code database is lost. The code generator program is buggy or missing. The boot record of the shark can also be faulted. One procedure that has been found to revive the 3.0 is as follows:
Step 1 - Turn your Game Boy off.
Step 2 - Insert a fully working 3.0 shark in your Gameboy.
Step 3 - Insert the corrupted shark into the working shark.
Step 4 - Insert a game into the corrupted shark.
Step 5 - Turn the Gameboy on.
Step 6 - There should be a different icon on the right side of the Gameboy screen that indicates upgrade. Choose it and let the process complete without interruption.
Step 7 - Wait to see process is complete, turn your Gameboy off and remove the game and sharks from the Gameboy.
Step 8 - Insert shark that was corrupted in Gameboy with game and test shark.

Note: If no upgrade icon is seen in step 6: This may mean your corrupted shark has a hardware fault that will not and CANNOT be corrected by reflashing...ergo...your shark is dead.

What is GGCC2k2?

Game Software Code Creator 2002 (GSCC2k2) is a program devoloped by CodeMaster. It features many improvements and added options over Interact's old PC Utils. You can get it at http://www.cmgsccc.com/.

A code doesn't work for me for the other levels, what can I do?

Certain games have certain "beginning" codes, and certain letters that follow it. For example, all of the Jetpack codes for Shadows of The Empire begin with "801", and then for one of the levels it is soon followed by a "A55" for the Xizor's Palace level. If you wish to use that code for a different level, which doesn't always work for the other levels, then you merely switch those three (or if two) digits with the digits that represent the level that you wish the code in. Sometimes you may need to find a different three-digit code that that specific code needs to work.

What games are easy or difficult to hack with a GameShark?

Please refer to above document.

Have there been any theories made for hacking with a GameShark?

Please refer to above document.

What does the term "Porting Codes" mean?

It means that you are transferring a code from one version of the game to another. This example is from Mortal Combat Trilogy v1.0 & v1.1. The v1.0 Codes-

P1 No Energy: 8016984D 0000
P2 No Energy: 80169B21 0000

v1.1 Codes-
P2 Automatically Dies: 80169C61 0000
P1 Automatically Dies: 8016998D 0000

Notice how the two versions of the game are the same except for the sixth, seventh, and eighth digits. I find usually that transferring codes from one game to another the sixth digit goes up one 'step' in the code. (E.G. 'C' would become 'D'). Further info is available in this document. See how to hack codes section for online code porter site.

How can I tell what version I have of a game?

One way at the moment to see what version of a game that you own is by testing all of the known versions of the game that you own with your copy of the game. Depending on which version of codes works, that is the version that you own! With most games you can tell just by looking at the label, but not in some cases.
There is a part of the label that reads: NUS-006 (USA) NUS-NWGE-USA All games have something like that on the label the first line seems to stay the same from game to game but the "NWGE" is different sometimes. The example above shows v1.0 of the game. Here is what v1.1 looks like: NUS-006 (USA)NUS-NWGE-USA-1

Does the GameShark support Japanese games?

Yes, the GameShark can hack foreign games.

Can I play Japanese games on the GameShark?

Yes, you can. Simply plug the foreign game in and start the game without any codes! PSX Users must do that "swap trick".

This edited version of the GameShark FAQ was made possible by contributions over the last couple of years by these fine people:
Jim Reinhart, Avid Gamer, Kong K. Rool(aka Parasyte), ShadowKnight(aka Dawn of Time/Rune), Gaming Freak, and special thanks to Gavin Thornton, Ali Yates and Marios(from Datel) for helping get the original shark link off the ground. A Special Thanks to Bill Kaufman (CodeBoy) for supplying Game Sharks to the GSCentral beta test team and to Kris Schineller at Blaze USA for providing sample N64 and GB Xploders for testing and review. Finally, a special thanks to Wayne Beckett - the Game Shark creator, without whom, the concept of Game Shark may never may have become a reality.

What are the Z64 and V64?

The Z64 is a utility that is inserted into the Nintendo 64. A game cartridge is inserted into the Z64, and it is then possible to store the games contents (ROM) onto a 100 Mb zip floppy. The game can later be played without the

original cartridge by using the 100 Mb disk. The game ROM can also be copied to a personal computer, and played on an emulator. The Doctor V64 is a similar utility, that is inserted in the bottom of the Nintendo 64.

Note:

Copying games is illegal. GSCentral does not endorse the illegal copying of games, and can not be held responsible for any illegal activites engaged in.

Macrox - April-21-2001
Additions by Tolos - June-14-2003

XV) Playstation Xplorer/Xploder Code Types: In-Depth FAQ - Courtesy of Hackman

---

XPLORER CODE STRUCTURE

1.	3-Code	7.	9-Code (Do-if-False)
2.	8-Code	8.	B-Code (Slide Code)
3.	4-Code (Slow Motion)	9.	C-Code
4.	5-Code (Supercode)	10.	D-Code
5.	6-Code (Modcode)	11.	F-Code (Auto Activation)
6.	7-Code (Do-if-True)	12.	Joker Command

Xplorer Code Structure

It seems rather amazing that an Xplorer code, a mere sequence of 12 numbers, accomplishes such stunning results as invulnerability. On the following pages we will describe these mysterious codes in detail -- specifically their structures, types, and functions. This knowledge is not necessary if you only want to occasionally hack a simple cheat. However, in order to rise to the class of the professional hacker in all around abilities this section can not be avoided. It should also be noted here that all codes are described using HEX values and all mathematical computations are done using the base HEX.

The Xplorer code usually consists of the following parts:

CSAAAAAA DDDD

C = Code identification or Control Code which tells the Xplorer, what follows next (left half of the first byte or 1 number).

S = Status of the code (right half of the first byte or 1 number).

A = Storage Address in the PlayStation RAM with a range from 000000 to 01FFFF, (3 bytes or 6 numbers).

D = Data, these values are written to the given address A, (2 bytes -word- or 4 numbers).

Note: For the sake of clarity, a blank character often is inserted after each fourth number in the code. If you enter a code in the Xplorer, omit this blank character. Thus - way of writing: 8009 1234 5600; way of entering: 80091234 5600.

We will explain the types of codes (C) in detail on the following pages.

(Comment: This section assumes that you are activating ALL codes by selecting the name of the game.)

We mark the status thereby by an italic S. (example: 8Sxx xxxx) it can contain only 8 or 0. If it is 8, then you must switch this code on manually, since it is not activated automatically when selecting the codes by Game Name in the Xplorer menu. That makes sense, because with so many variations of game play there are usually codes of the following type:

Infinite Energy
8001 1234 03E7

On Hit Death

```
8001 1234 0001
```

To have both codes switched on is unreasonable since they affect the same address. The PSX CPU must decide whether the player has infinitely energy or can be defeated immediately. If you only want infinite energy to be activated when you select the entire game then the code becomes:

```
Infinitely Energy
8001 1234 03E7

On Hit Death
8801 1234 0001
```

Now, when you select the Game Name to turn on all the codes, all will be activated EXCEPT One Hit Death. Additionally, if you have multiple code lines the status value must be entered in the FIRST code line:

```
Have All Items

B8A6 0002 0001
1007 836C 6401
B021 0002 0001
1007 84B8 84A7
```

The above code now must be manually activated even though you may have chosen to activate ALL game codes by selecting the name of the game.

Status Note: The developers of the Xplorers have a method of encrypting codes in such a way that they function with the Xplorer but not with other Cheat Carts. Such codes you can recognize by the status "6" (more recently "4" and "5" also). Encrypted codes are published only by FCD (aka Blaze International in USA.)

Xplorer Code Types

3-Code

The 3-Code is the simplest. It describes a memory cell with a fixed amount. The code 3000 0004 0001 writes the byte 01 into the address 000004.

Example:
The code 300D 8A42 0003 keeps the number of the lives constant at 3. If you would rather have 9, then change the value (data) part of the code to 9: 300D 8A42 0009.

Composition:
3Sxx xxxx 00yy

where: xx xxxx address yy value (byte)

Note: If you should see a 3 or 8 code which possesses the status (S) of 6 (more recently "4" and "5" also), it concerns an encrypted code.

8-Code

Together with the 3-Code is an 8-Code, the basis code, which describes the memory location directly. With the 8-Code two bytes are registered in two addresses. The code 8000 0004 FF01 writes directly into the address 0000 0004 the word value FF01 which has the following result:

```
Memory Cell New Contents
0000 0004 01
0000 0005 FF
```

Note: The 8-Code of the PlayStation permits only the access to even storage addresses (technical term: "even alignment"). An attempt to go around this adjustment is not possible and can provoke a crash. Make sure that at the

eight place of the memory address you have an even hexadecimal number (0, 2, 4, 6, 8, A, C or E).

With the help of the 8-Codes you can register (decimal) a value between 0 and 65535 into a specific memory cell. Convert the value, 9999, into hexadecimal which results in the HEX value of 270F and corresponds to the right part of the code -- the data (D) part. However, the value would be stored in reversed order as 0F27.

Note: The PlayStation processor stores bytes of larger numbers in reverse order.

An 8-Code has the same effect as two 3-Codes with the two storage locations being directly next to one another. It is sometimes meaningful to split an 8-Code up into two 3-Codes if we want only one part of it to function.

Suppose you have a code 8008 8AEC FF01 with the description "Rifle + 255 cartridges" or "Rifle with Unlimited Ammunition". Experienced Coders can see immediately where the values lie that describe each cheat (I.e., FF = Unlimited Ammo and 01 = Rifle). Usually programmers will note in an individual memory cell whether an article is available or not. The absence is usually marked by 00, the presence by 01. Each processor understands an instruction of the kind..... if contents of the memory cell x contains 0 , then do this...
Such fragments are used if you change from a pistol to a rifle -- then the program must examine whether you have the rifle at all. The possession variable does not necessarily have to be 01 - you could use 05, 27 or any other number. Usually, however, one takes 01 because it can be associated so nicely with the variables "yes" or "on." This would also correspond with the original language of each microchip's binary values of 00 and 01.

For manipulating the ammunition or the number of the lives, where not just possession matters but the actual quantity is important, one usually selects a maximum value like 99 or 255 (FF in hexadecimal, the highest value for a byte). Naturally one could use most any number here because the value always remains constant. But, for psychological reasons a large number is recommend.

The code 8008 8AEC FF01 writes into the memory cell 0008 8AEC the value 01 in order to specify the rifle and into the cell 8008 8AED the value FF in order to make 255 cartridges available. Split up into two 3-Codes we have the following:

```
3008 8AEC 0001
3008 8AED 00FF
```

Note: In the Memory Editor the code would look like this (note the reversed values):

```
80088AE0    00 00 00 00 00 00 00 00 00 00 00 00 01 FF 00 00
```

The two 3-Codes have the same effect as the one 8-Code. You can, however, omit the second 3-Code in order to limit the overall effect -- you have received the weapon but must look for the ammunition independently. If you omit the first 3-Code, you must find the rifle only and would not have to concern yourself with getting ammunition. An 8-Code is unsuitable for this choice as the code

```
 8008 8AEC 0001
```

gives you the rifle without any ammunition.

Examples
1. In the address 0000 0004 an 01 means that you possess the appropriate weapon. In the following address, 0000 0005, the quantity of ammunition for that weapon is stored. You would like to possess the maximum number of bullets, i.e., 255 (FF in hexadecimal). The correct code is then 8000 0004 FF01.

2. In the address 0000 000A the number of possible magic points are stored. Since we want to make this value as large as possible, and only have 4 numbers within which to do so, we take the maximum value of 9999 (0F27 HEX). The correct code is then 8000 000A 270F.

3. From the following two codes:
3001 26AA 0063
3001 26AB 00C7
you would like to make an 8-Code because it is shorter. The result is:
8001 26AA C763

4. The code 8008 8AEC FF01 describes two memory cells. If you want to leave one of the two cells unaffected, split the code in two as follows:
3008 8AEC 0001
3008 8AED 00FF
and then omit one of the two lines.
Composition
8Sxx xxxx yyyy
where: xx xxxx even storage address
yyyy value which can be registered (2 bytes - word)

4-Code (Slow Motion Code)

With the help of the 4-Code you slow down the execution of a game. Logically it is called a Slow Motion Code. A meaningful application of this type code is when your opponent is very fast and you cannot react quickly enough. With the Slow Motion Code you can create for yourself a greater chance to win.

The 4-Code is not connected with a certain storage address because the processing of the game was not intended to be slowed down from the start. It is recommended that the activation of the slow motion code be accomplished by depressing a particular JoyPad key. For such codes, which are to apply only during certain time periods, there is the so-called Joker Code or Command. A Code which is activated with such a Joker Command is only effective as long as the selected key or combination of keys is held. We will discuss the Joker Command at a later place in more detail.

Because of the missing storage address a 4-Code is quite naked -- consisting of all zeros. It exclusively contains the delay value which should not exceed 12 (0C). Delays are accomplished internally as the processor runs through a series of millisecond rotations or loops (technical term: to over loop).

Example

You would like for the expiration of play to be slowed down as long as you hold the Select key. You have previously determined that the Joker Command address is D009 1234 0100 (that is usually quite simple). You feel that a suitable delay value is 6. The complete code is then:

D009 1234 0100 (Push the Select Button)
4000 0000 0006

Composition
4S00 0000 000x
Where: x Number of repetitions or 'loops' (delay value)

5-Code (Supercode)

The 5-Code is that so-called Supercode. It is meaningful if you want to describe several memory cells one behind the other. Some examples are:

- changing a text, a player's name for example.
- manipulating a whole inventory list.
- changing a diagram, like a texture.

For the Supercode you must make arrangements for three data types:

- the memory cell, starting from which the changes become effective.
- the number of places which can be changed.
- the new entries.

In your preferred combat game your favorite character is named Eddy. However, you would find it more to your liking if he was called Rico. With the text search function of X-link you determine the storage address starting from which the character string "Eddy" is found -- for example 0002 67AA. Determine now the hexadecimal ASCII-Code of the character string "Rico." That can be done most simply through "manipulation" of the memory editor of X-link. Write into the right column (the ASCII Column) the character string "Rico" and then check to the left which HEX values have changed as a result of your changes to the name "Eddy.".

Note: X-Link for DOS generates a 5-Code automatically if it concerns text.

Select in the menu option / further options / advanced code generator / string code where you must indicate only the storage address and the text in order from it to provide you with a completed code.

Example

You would like to insert the text string "GBP!" starting from the address 0002 67AA. The correct code is:

G = 71 ASCII DEC = 47 HEX
B = 66 ASCII DEC = 42 HEX
P = 80 ASCII DEC = 50 HEX
! = 33 ASCII DEC = 21 HEX

5002 67AA 0004
4742 5021 0000

Composition

5Sxx xxxx 0yyy
zzzz zzzz zzzz
zzzz zzzz zzzz
.....

Where: xx xxxx Address (in our example 0002 67AA)
yyy Number of repetitions (for example 004 or 4 letters)
zzzz zzzz zzzz The bytes which can be used (for example GBP!)

Fill up the remaining spaces in the last line with zeros in order to have a ﬠfullﬠ or complete line of code.

6-Code (Modcode)

The 6-Code lets a piece of program code implement on a certain event - for instance when depressing a key or describing a certain memory cell. The code consists of several parts: the kind of event which leads to the temporary abort (BREAK) of normal program processing, the memory cells concerned, and the program code which is to be implemented with the arrival of the event.

The 6-Code is the code most complicated for the inexperienced. A basic requirement is a comprehensive knowledge of the assembly language programming of the processor. Consequently, we present the code only briefly.

A summary of the code is as follows.

Composition
6S?? ???0 YYYY
AAAA AAAA CCCC
FFFF FFFF XXXX
XXXX XXXX XXXX
....

Where: ? All the same value.
YYYY Number of bytes which are used.
AAAAAAAA BREAK address (here 001F AB92 and 3).
FFFF FFFF BREAK POINT mask (on 0FFF FFFC set).
XXXX XXXX The bytes which can be used.
CCCC Type of BREAK POINT at the BREAK address:
E180 The instruction gotten by the processor but is not yet implemented.
EE80 The data to be read or written
E680 The data to be read.
EA80 The data to be written.
EF80 The data to be either read, written or implemented (instruction).

Note: X-Link for DOS contains two templates which determines your 6-Codes via the advanced code generator. They must indicate only the necessary addresses and values. The Mod code has the form: "If the address X has the value Y, then add / subtract the value A to / from the address B." After the input of the four values the generator makes a pertinent 6-Code. The slip code changes contents of a memory cell as a function of depressing a specific key.

7-Code (Do-if-True)

This Joker code works with an If-Then relationship: If a specific memory cell contains a certain value, then the Xplorer becomes active and writes into a fixed (same or other) memory cell a fixed value. The 7-Code is identical to the D0-code of the GameBuster.

Example

You would like to receive after collecting a certain weapon the pertinent ammunition. Or in hexadecimal language: If the address 0001 2346 (weapon) contains the value 0001, then the address 0001 8450 (ammunition) is to receive the byte with the value of 63 HEX. As a code that results in the following:

```
7001 2345 0001
3001 8450 0063 (63 HEX = 99 DEC)
```

Composition

```
7Sxx xxxx yyyy         Where  xx xxxx  Address (in our example 0001 345)
                                yyyy    Value (in our example 0001)
```

Observe that always two bytes (word) are queried at one time (yyyy).

Note: The Joker code generator of X-Link for DOS generates a finished 7-Code based on your defaults. Do not confuse the Joker code with the Joker Command which we will present later.

9-Code (Do-if-False)

The 9-Code is the opposite of the 7-Codes. It activates only when the code which follows directly after it does NOT contain the expected value. Or differently, the following code is ignored if the condition is fulfilled.

Example

In address 0001 8430 the number of your cartridges is stored. However, the game always crashes in Level 2 if the code is activated. In all other Levels the code functions as designed. With the following example you query the Level which your on and if you are NOT on the second level then the code for the ammunition is activated:

```
9011 8AFE 0002
3001 8430 0063
```

Composition

```
9Sxx xxxx yyyy

Where:     xx xxxx     Address (in our example 0011 8AFE)
              yyyy       Value (in our example 0002)
```

Observe that always two bytes (word) are queried at one time (yyyy).

Note: The Joker code generator of X-Link for DOS generates a finished 9-Code based on your defaults.

B-Code (Slide Code)

(A more in-depth discussion can be found in the Advanced Xplorer Code Types FAQ)

The B-code or so-called Slide Code shortens long code lists to a compact instruction. For games in which there is an inventory in existence such as a chest with collected treasures, a whole list of full spells, or a selection of different weapons, the storage location of each item in the inventory list is usually incremented by the same value. The solution to using fewer code lines is the B-code. A code list for All Items might look like this:

```
8001 1234 0101
8001 1236 0101
8001 1238 0101
8001 123A 0101
8001 123C 0101
8001 123E 0101
```

```
8001 1240 0101
8001 1242 0101
8001 1244 0101
```

This can be shortened with the help of a B-Code by using only two lines. The address in this example is always increased by 2 and the value remains the same in each case.

From our example list above:

```
B009 0002 0000
1001 1234 0101
```

Note: The Slide code generator of X-Link for DOS generates a finished B-Code based on your defaults.

The exact function results from the composition:

```
Composition
B0ww AAAA DDDD
10xx xxxx yyyy

Where:            ww     Number of repetitions plus 1 (in the example 8+1)
                AAAA     Size of the address step (in the example 2 = always
                         even, 2 or greater)
                DDDD     Size of the data step (in the example 0 = always the
                         same value)
             xx xxxx     Start address (in the example 0001 1234)
                yyyy     Initial data (in the example 0101)
```

Slide codes can be combined with a D-, C-, 7-, or 9-Codes.

C-Code

This If-Then code was only taken up in order to remain compatible with the GameBuster. The appropriate Xplorer code is a 7-Code. The C-Code corresponds to the D-Code. The only difference was that in quite early version of the GameBuster / GameShark the C-Code was responsible for the upper megabyte of the PlayStation. The D-Code functioned only within the address range from 0000 0000 to 000F FFFF while the C-Code only from 0010 0000 to 001F FFFF. Today the C-Code is no longer needed since the D-Code can now be used for this entire memory block. For the sake of completeness, nevertheless, a composition of the code:

```
Composition
C0xx xxxx yyyy

Where:     xx xxxx     Address in the upper megabyte which is to be examined
              yyyy     Value for which one examines
```

Observe here always two bytes (word) are queried at one time (yyyy).

D-code

An If-Then code identical to the 7-Code: With this code the following line is implemented only if a certain condition is fulfilled -- that condition being a certain value in a certain memory cell. The D-Code is only supported in order to remain compatible to the GameBuster codes. For newly created codes one recommends using a 7-Code in lieu of a D-Code. (Note: for the newer versions of the GameBuster there are other types of D-Codes such as D1, D2, etc. that are not compatible with the Xplorers and will not be presented here.)

Example

If the storage location 0000 0100 contains the value 0000 then write the value FF in 0011 1234 and 0011 1235. The storage location 8011 1236 FF00 is not affected as the D-Code only writes to the storage location of the first line of code directly following it:

```
D000 0100 0000      (If 8000 0100 contains 0000 then activate the first
                       line of code.)
8011 1234 FFFF
```

```
8011 1236 FF00      (This line of code is not written to.)

Composition
D0xx xxxx yyyy

Where:     xx xxxx   Address which is to be examined (in the example 0001 1234)
              yyyy   Value for which one examines

Observe that always two bytes (word) are queried at one time (yyyy).
```

F-Code (Auto Activation)

```
The F-Code is the equivalent of the Master or Must-Be-On Code of the
GameBuster / GameShark. It is called Auto Activation and resembles the D- or
7-Code except that it is automatically selected when any other code is selected
in the Select Cheats Menu. Additionally, unlike the D0- and 7-Codes where only
the first line of code is activated when the Do-if-True conditions are met, the
F-Code will activate ALL lines of Code for the game that have been initially
selected in the Select Cheats Menu. One should realize here that NO codes are
'activated' during the game until the conditions of the F-Code are met. As with
the 8-Code, two bytes are always queried.

Example

The game crashes because of some specific code conflicts in the Main Menu or
during the loading of Introduction Films. Consequently, you only want the codes
to be active AFTER the game has started. You find out that during the
Introduction the address 0011 8AFE does NOT contain the value 612A -- it only
contains that value AFTER the game has loaded. Thus we want the codes that we
have initially selected to be activated only if the address 0011 8AFE contains
the value 612A. If the 612A value is missing then the codes are not activated.

"GAME NAME"
-------codes-------
8007 84EC 00FF    <<<<< I.e., Gets activated if originally selected in the
                        Select Cheats Menu.
Auto Activation
F011 8AFE 612A    <<<<< Do-if-True code. ALL of the selected code lines, both
                        above and below this code line, are activated if this
                        condition is met. The codes remain inactive till then.

Infinite Gel and All Items
8011 A72C 270F    <<<<< I.e., Gets activated if originally selected in the
                        Select Cheats Menu.
8011 AEEC FFFF
B0A6 0002 0001
1007 836C 6401

Next Cheat Description
8007 84B8 84A7
-------codes-------

Note: This also works with Joker Commands.

Composition
FSxx xxxx yyyy
Where:     xx xxxx     Inquiry address (in our example 0011 8AFE and F)
              yyyy     Inquiry value (in our example 612A)
```

Joker Command

```
(A more in-depth discussion can be found in the GameShark Code Types FAQ)

The Xplorer writes a constant value to a memory cell many times per second. This
is called constant write or always on. Sometimes, however, it would be
desirable if a code were active only at certain times because:

 - A Cheat is to be switched on only in an "emergency" -- you would like to
   play predominantly in an honest fashion.
 - A Cheat works properly only during the actual game and causes the PlayStation
   to crash during the Introduction or at the Main Menu.
 - A Cheat functions only on certain levels.
 - A Cheat does not function in certain situations.
```

The solution is the Joker Command. It permits activation of a code temporarily while a certain key or combination of keys of the Controller is held.

Note: Cheats can be also be deactivated temporarily if the Xplorer's on / off button is switched off or, with the Xplorer FX, by using the In-Game menu to switch the codes off.

In each game there is at least one place in memory (or memory address) where the pressed keys of the JoyPad are shown by a specific value. This address is usually accessed by the use of a D- or 7-Code.

A Joker Command is thus a D- or 7-Code whose address is a storage location for a specific JoyPad button or combination of buttons. Joker Commands can be used with any game and combined with any code type.

You can select from the Table below the pertinent key code. Some games use this value in a reverse format called a Reversed Joker Command. Here only the key value changes. Otherwise the Reversed Joker Command functions exactly like a Normal Joker Command.

Normal Joker Command Digits		Reversed Joker Command Digits	
VALUE	KEY	VALUE	KEY
0001	L2	0100	L2
0002	R2	0200	R2
0004	L1	0400	L1
0008	R1	0800	R1
0010	Triangle	1000	Triangle
0020	Circle	2000	Circle
0040	X	4000	X
0080	Square	8000	Square
0100	Select	0001	Select
0800	Start	0008	Start
1000	Up Arrow	0010	Up Arrow
2000	Right Arrow	0020	Right Arrow
4000	Down Arrow	0040	Down Arrow
8000	Left Arrow	0080	Left Arrow

In cheat data bases or code sites you will usually find the JoyPad keys that are used in place of ????. I.e., Replace the question marks by a key of your choice. You can combine several keys such that a code is only activated if you press Select + Left Arrow + Triangle by adding the numerical values, in HEX, of the selected keys:

Normal Joker Command	Reversed Joker Command
0100 = SELECT	0001 = SELECT
0004 = L1	0400 = L1
0010 = Triangle	0010 = Triangle
-	-
0114 = Select+L1+Triangle	0411 = Select+L1+Triangle

Note: Note that these calculations are accomplished in hexadecimal. That can be done quite easily with a scientific calculator -- if necessary use the one that comes with Windows.

There are also complementary and reverse-complementary versions to these two types of values. In order to make the complementary of a value, you subtract it from FFFF:

- subtract 0100 (Reversed for #L2) from FFFF to get FEFF
- subtract 0200 (Reversed for #R2) from FFFF to get FDFF
- subtract 0300 (Reversed for #L2 + #R2) from FFFF to getFCFF

These are known as Reversed Complementary Joker Commands or R.C.J.C.

Example

You found the following code:

Infinite Energy
8012 3456 00C8

In addition we have found the Normal JoyPad address (This will be discussed later.) and made it a D-Code:

D001 2876 ???? (This is now our Joker Command.)

Further we know that the key Select is not used in the game so nothing "critical" would happened if it was pressed. Now you look in the table and you will see that the value for Select is 0100. Enter the following:

Infinite Energy
D001 2876 0100
8012 3456 00C8

Start the game with this code selected. First nothing happens with respect to the "Heros" energy -- but as soon as you press Select the energy is refurbished. But only as long as you keep the Select button pressed -- release it, or press any other key, and it no longer functions.

Last Modified: June 3, 2000 - Hackman

Section 4: Legal
