

Gameboy Advance Trainermakerguide (aka HOWTO)

By Anonymous Man

(hell yes, I wanna stay anonymous)

Initial words and other stuff

Welcome to the Gameboy Advance Trainermakerguide (aka GBA Trainer HOWTO). The intention of this trainerguide is to show people that doing trainers neither has something to do with magic nor is very complex (if you are doing "basic" stuff). I for myself have never released a trainer to the public, I hack for fun and out of personal interest. But I feel that there are more people out there who want to do trainers but cannot find a start to do them, I hope this helps :-)

What do we need

For this trainerguide (and for your upcoming trainers) you'll need the following:

- a brain
- little bit ARM-Assembler knowledge
- a hexeditor
- an ARM-Assembler
- an ARM-Disassembler

I use the following tools, I suggest you get them before we start:

Hexeditor: Hex Workshop

(get it from **<http://www.bpsoft.com>**)

ARM-Assembler: Goldroad ARM-Assembler

(get it from **<http://www.goldroad.co.uk/grARM.html>**)

ARM-Disassembler: IDA Pro with ARM functionality

(Commercial stuff, get it from your local warez-dealer :-))

You could also try: dsmgba but I dunno if this is good.

(get it from **<http://www.postbox.javamaster.co.uk>**)

But for serious hacking try to get your hands on IDA Pro.

Before we start: some basics

This guide assumes that you already hacked some codes for the trainer. This is not always simple and infact the mainwork which has to be done. If you ask me now how to do this I can only tell you that you could try emulators (**like Visual Boy Advance**) which have built-in cheatcode finders. Also **no\$gba** is worth a look which features advanced breakpoint functionality.

Assumed you have hacked this codes the question is what does this code do?

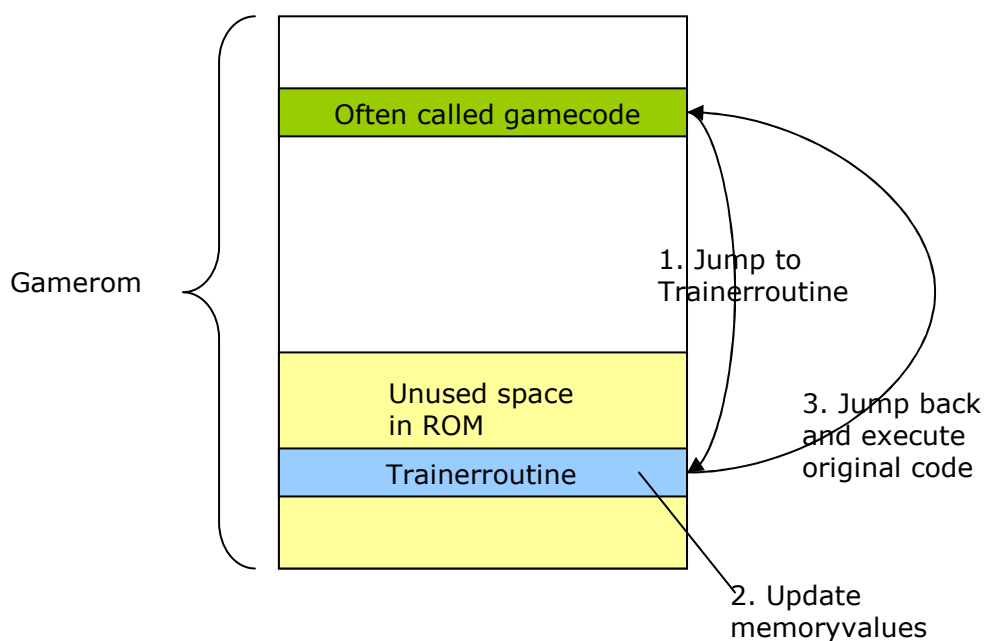
For example you have successfully hacked the following code for unlimited energy in Visual Boy Advance:

03004b20 06

Spoken this code means nothing else than: "Write bytevalue **0x06** to memorylocation **0x3004b20** in Gameboy Advance memory very often." So while you are playing the game with this code turned on when loosing an energy-points the value 0x06 (which represents your energy-points within the game) will be subtracted to 0x05 but then it will be almost immediately (you wont notice in the game) set to 0x06 again (so you have 6 energy-points again).

And this bytevalue 0x06 which represents your energy-points is stored at address 0x3004b20 in the Gameboy Advance memory.

Later, our trainer is actually the routine which writes this bytevalue 0x06 to address 0x3004b20 very often. And to make sure that the trainerroutine (which updates) is executed very often we have to hook it to a very often called gamecode which will call our trainerroutine (what a sentence :-)). I hope the following graph will help you to understand it:

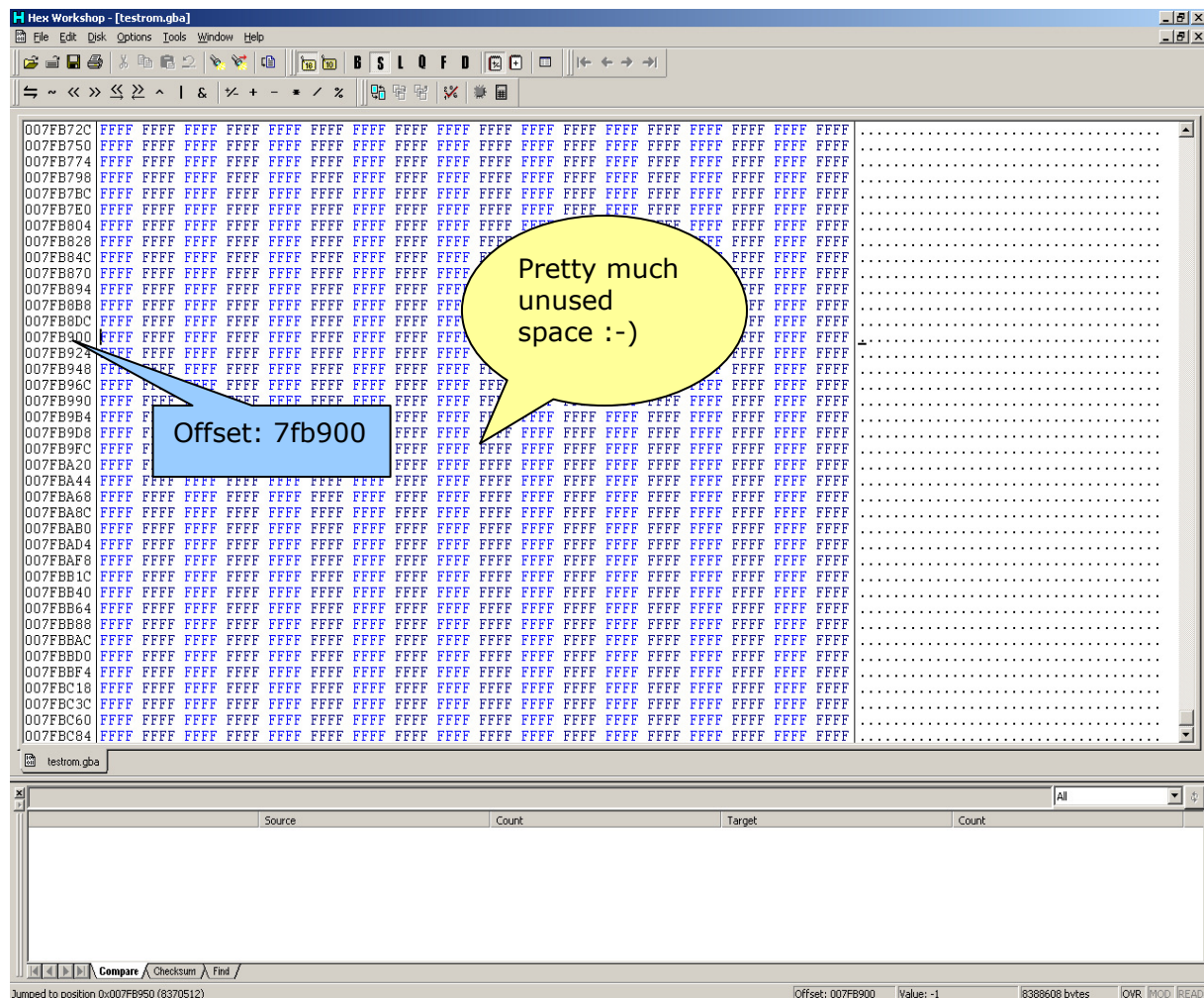


For other explanation on this issue I suggest you to check the Nintendo64 trainermaking guide made by Icarus of Paradox here:
<http://n64dev.50megs.com/trainer.html>

First step: Find relevant information about the gamerom

A place for the trainer:

Fire up HexWorkshop and search for unused space within the rom. Go to the end of the romfile and go upwards usually (**not** always) unused space is filled with 0xff or 0x00. For this trainerguide we assume that we have found a nice place in the gamerom at offset **0x7fb900**. Write this offset down on a piece of paper.



Find the "often called" gamecode for hooking our trainerroutine.

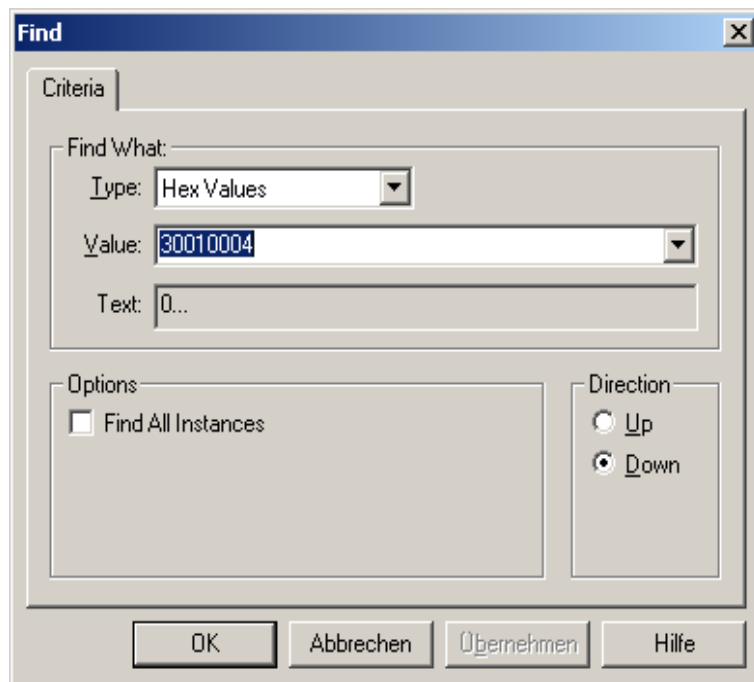
We could hook our routine to code in many places within the rom. But unfortunately we have to search for a decent routine which is called/executed very often so our hooked trainerroutine will be executed very often, too. On the Gameboy Advance a nice place for this is the routine which is called by the game to check which keys are pressed at

the moment. This routine is of course called very often by default because this is necessary to have a fluent control of the sprites in the game.

So fire up HexWorkshop again and search for following hex values:

30010004

In HexWorkshop this does look like the following:



In our testrom we get a hit for this at following offset: **0x3a2e4**. Write this offset down on a piece of paper.

0003A280	2E00	0004	00B5	0349	0348	0180	0348	04F0	BDFA	00BD	0000	0000	0802	0004	0D01	0008	00B5	0F49
0003A2A8	0A88	D043	0104	0814	0D49	0B88	021C	9A43	131C	0C4A	0880	1904	1380	0029	04D0	011C	0F20	0140	...C...
0003A2C0	0F29	00D0	00BD	0348	0649	074A	0880	04F0	9DFA	F7E7	0000	0000	3001	0004	480F	0003	4A0F	0003	.)....
0003A2F0	0802	0004	0D01	0008	0268	8118	4B68	0433	D118	0160	7047	0000	00B5	0423	0068	0331	5A42	1140
0003A314	0028	06D0	0268	8A42	03D2	4068	0028	F9D1	0020	0430	00BD	0000	F0B5	021C	0029	13D0	0424	C81C	..(..h
0003A338	6142	141C	1268	0840	002A	0BD0	1168	8142	06D2	5368	141D	002B	01D0	1A1C	F6E7	0022	002A	01D1	aB...h
0003A35C	0020	F0BD	2568	271C	1368	2968	0026	9842	02DC	8B42	00DC	0126	002E	04D0	2568	2968	271C	8142	...%h
0003A380	05D0	5368	141D	002B	01D0	1A1C	EAE7	2968	0A1A	082A	09D9	6B68	2A18	111D	4B60	2B68	1B1A	043B	..Sh..
0003A3A4	5360	2860	6960	6868	3860	281D	D5E7	0000	00B5	FFF7	B9FF	00BD	00B5	011C	0028	07D0	C022	031F	S'('i'
0003A3C8	9004	8342	03D3	0348	00F0	08F8	00BD	0248	FAE7	0000	4C0F	0003	500F	0003	30B5	031C	0029	1ED0	...B..
0003A3EC	1A68	081F	1D1C	002A	0F00	8242	0DD2	1468	1119	0B1D	8342	1CD0	5368	151D	002B	16D0	8342	01D2	.h....

Second step: Disassemble ROM to find morerelevant information

Okay.. I made this a new chapter. Now I use IDA Pro to disassemble the rom. I hope you have this one. If you use another disassembler do the following steps appropriate.

Okay, fire up IDA-Pro, load the rom and select ARM as processor type.

Go to location **0x3a2e4** where we have found the hit with HexWorkshop and disassemble the location above this offset (in **THUMB-MODE!**). For our testrom we will find a disassembly which looks like the following:

```

ROM:0003A2A4 ; -----
ROM:0003A2A4      PUSH    {LR}
ROM:0003A2A6      LDR     R1, =0x4000130
ROM:0003A2A8      LDRH    R2, [R1]
ROM:0003A2AA      MUN     R0, R2
ROM:0003A2AC      LSL     R1, R0, #0x10
ROM:0003A2AE      ASR     R0, R1, #0x10
ROM:0003A2B0      LDR     R1, =0x3000F48
ROM:0003A2B2      LDRH    R3, [R1]
ROM:0003A2B4      ADD     R2, R0, #0
ROM:0003A2B6      BIC     R2, R3
ROM:0003A2B8      ADD     R3, R2, #0
ROM:0003A2BA      LDR     R2, =0x3000F4A
ROM:0003A2BC      STRH    R0, [R1]
ROM:0003A2BE      LSL     R1, R3, #0x10
ROM:0003A2C0      STRH    R3, [R2]
ROM:0003A2C2      CMP     R1, #0
ROM:0003A2C4      BEQ     loc_3A2D0
ROM:0003A2C6      ADD     R1, R0, #0
ROM:0003A2C8      MOV     R0, #0xF
ROM:0003A2CA      AND     R1, R0
ROM:0003A2CC      CMP     R1, #0xF
ROM:0003A2CE      BEQ     loc_3A2D2
ROM:0003A2D0      loc_3A2D0                                ; CODE XREF: ROM:0003A2C
ROM:0003A2D0                                ; ROM:0003A2DE↓j
ROM:0003A2D0      POP     {PC}
ROM:0003A2D2 ; -----
ROM:0003A2D2      loc_3A2D2                                ; CODE XREF: ROM:0003A2C
ROM:0003A2D2      LDR     R0, =unk_0
ROM:0003A2D4      LDR     R1, =0x4000208
ROM:0003A2D6      LDR     R2, =0x8000100
ROM:0003A2D8      STRH    R0, [R1]
ROM:0003A2DA      BL      sub_3E818
ROM:0003A2DE      B       loc_3A2D0
ROM:0003A2DE      ; -----
ROM:0003A2E0      off_3A2E0      DCD unk_0                                ; DATA XREF: ROM:0003A2D
ROM:0003A2E4      dword_3A2E4    DCD 0x4000130                        ; DATA XREF: ROM:0003A2A
ROM:0003A2E8      dword_3A2E8    DCD 0x3000F48                        ; DATA XREF: ROM:0003A2B

```

Our offset

The yellow marker tells us that long-value 0x4000130 at offset 0x3a2e4 is used at offset 0x3a2a6 (see yellow marker above).

Write down the offset of the second command after 0x3a2a6 which is in this case: **0x3a2a8**:

```

ROM:0003A2A6      LDR     R1, =0x4000130
ROM:0003A2A8      LDRH    R2, [R1]
ROM:0003A2AA      MUN     R0, R2

```

Also **write down** the command which is executed there: **"ldrh r2,[r1]"** and **"ldr r1,=0x4000130"**. Also the offset of the command which we jump in back again after our trainer routine is called: **0x3a2aa**.

Then take a look around in this routine and find out which register we could use for our jump later on. Ofcourse we'll take an unused register which we can use (so that we don't trash anything important).

In this case the **R4** register would be nice to use. Note: In most of the cases R4 is a good idea but of course you have to take a look EVERYTIME you hack a rom to go sure.

We are done here, so close IDA-Pro.

Third step: Writing and adapting the trainerroutine

Okay we have to write a small assembler-program which suites for following assumed code: 03004b20 06. This code say that we have to write a bytevalue (0x06) to memoryadress 0x3004b20, remember?

So here is the code which does the trick:

```
ldr r4, =0x03004b20
mov r5, #6
strb r5, [r4]
```

Okay. But we are trashing registers r4 and r5 in here which could cause trouble when we return to the original gamecode later. So we have to save them in the stack, the new code would look like this:

```
push {r4-r5}

ldr r4, =0x03004b20
mov r5, #6
strb r5, [r4]

pop {r4-r5}
```

Well, we are still not ready. Our trainerroutine has to jump back after everything is done to the original gamecode again, you wrote down the offset of this which is **0x3a2aa**. This is a file-offset but we need the ROM offset so OR this with 0x08000000 and you get: 0x08003a2aa. Now add 1 to this just calulated rom-address and you get **0x08003a2ab**.


Phew.. now we can update the trainerroutine with this knowledge:

```
push {r4-r5}

ldr r4, =0x03004b20
mov r5, #6
strb r5, [r4]

pop {r4-r5}

ldr r4, =#0x08003a2ab
bx r4
```



Please note that we are using now the **R4** register we have found to be secure for operations!!

But well, something is still left to be done: before we are going to return we have to execute the command which we will trash for jumping into our trainerroutine, we wrote this down, remember? -> **ldrh r2,[r1] – Also we have to execute ldr r1,#0x4000130 before jumping back because we'll edit this also later!**

So here is our "final" trainerroutine:

```
push {r4-r5}

ldr r4, =0x03004b20
mov r5, #6
strb r5, [r4]

pop {r4-r5}

ldr r1,#0x4000130
ldrh r2,[r1]

ldr r4,#0x08003a2ab
bx r4
```

Well, really final? – Unfortunately no. This is our update-routine for the energy-value in the Gameboy Advance memory. But in the trainermenu the player can select if this option is turned on or turned off. Our current routine it doesn't care for what the user selects, the cheat would be always turned **on**. So we need something which tells us whether to do the update or not. For this reason the trainermenu has to put a TRUE (1) value somewhere if the options is flagged as turned on or FALSE (0) if not.

Do the following: Put a 0x01 to memorylocation **0x03007fb0** if you want to have the option and 0 if you don't want to have the option. You have to do this with your trainermenu, I won't go into this any further, I hope you know how to do some decent trainermenus for the GBA (if not, why the fucking hell are you reading this anyway?)

The location **0x03007fb0** is a relative safe place where you can store tiny stuff.

So our **FINAL** trainerroutine has to check this location also and does look like this:

@thumb

```
push {r4-r5}

ldr r4,#0x03007fb0
ldrb r5,[r4]
cmp r5,#1
bne end

ldr r4, =0x03004b20
mov r5, #6
```

Is there a „1“ at location 0x3007fb0? If no (bne) then end, if yes, go further (update)!

```

        strb r5, [r4]

end:
        pop  {r4-r5}

        ldr  r1,=#0x04000130
        ldrh r2,[r1]

        ldr  r4,=#0x08003a2ab
        bx   r4

```

Now compile this code with the Goldroad assembler. And you get a binary **.gba** file.

Fourth step: Last modifications to the gamerom.

Okay Now we have our **trainer.gba** file, the (still **clean, original**) rom and some (yet) unused values on our piece of paper :-)

Now it's about time to attach the trainer.gba to the rom and do other necessary stuff to get everything working.

The first thing you have to do is to put the binary trainer.gba file at the location we have found at the beginning (where the much space in rom is). In our case this was: **0x7fb900**. To achieve this you can use HexWorkshop in *insert*-mode or any other tool which does the trick.

Then, we have to patch the gamerroutine we have found so that it executes our trainerroutine which is now located at file-offset 0x7fb900 and at rom-offset (remember OR'ing 0x08000000) at location 0x087fb900. Now add 1 to this again to receive: **0x087fb901**

For this reason fire up HexWorkshop again and goto offset we got the hit by searching that strange value, remember? – I told you to write this down :-) – In our case it is: **0x3a2e4**.

At this location you'll find this strange value again 30010004:

```

BD 0000 0000 0802 0004 0001 0000
4A 0880 1904 1380 0029 04D0 011C
E7 0000 0000 3001 0004 480F 0003
60 7047 0000 00B5 0423 0068 0331
30 00BD 0000 F0B5 021C 0029 13DC

```

Now overwrite this with the rom-offset of our trainerroutine (with 1 added): **0x087fb901** (see above). But there is one thing you have to take care of when doing this: the endian-format! – Overwrite it by starting from the last byte, example: You want to overwrite with **0x087fb901** – then write **0x01b97f08**. So after the modification it has to look like this:


```

J00 0000 0802 0004 0001 0008 00
380 1904 1380 0029 04D0 011C 01
300 0000 01B9 7F08 480F 0003 42
347 0000 00B5 0423 0068 0331 52
3BD 0000 F0B5 021C 0029 13D0 00
41D 0000 01B9 1310 F0B5 0003 00

```

Now we are just one modification away from our first gba-trainer :-). Fire up HexWorkshop again and goto the location **0x3a2a8**. At this location we found that **"ldrh r2,[r1]"**-thing, I told you to write that down, remember?

Just that you know what I am talking about here is the disassembly of this location:

```

ROM:0003A2A4      PUSH    {LR}
ROM:0003A2A6      LDR     R1, =0x4000130
ROM:0003A2A8      LDRH    R2, [R1]
ROM:0003A2AA      MOV     R0, R2
ROM:0003A2AC      LDR     R1, R0, #0

```

Do you see that command **"ldr r1,=0x4000130"** ? Well, it is important to know which register is involved in this ldr because we now have to set our branch-command appropriately.

If it is R1 then overwrite the 16bit(2byte)-value at **0x3a2a8** with **0x0847**. Like this:

<pre> J03A284 2E00 0004 J03A2A8 0A88 0043 J03A2CC 0F29 00D0 J03A2FD 0802 0000 </pre>	before	<pre> J03A284 2E00 0004 J03A2A8 0847 0000 J03A2CC 0F29 00D0 J03A2FD 0802 0000 </pre>	after
--	--------	--	-------

This value is different if other registers are used. Below you will find a Table which tells you what value to write for different registers:

Register	Value (16Bit)
R0	0x0047
R1	0x0847
R2	0x1047
R3	0x1847
R4	0x2047
R5	0x2847
R6	0x3047

If we disassemble the rom after this modifications it will look like this:

```

ROM:0003A2A4      PUSH    {LR}
ROM:0003A2A6      LDR     R1, =0x87FB901
ROM:0003A2A8      BX      R1
ROM:0003A2AA      ; -----
ROM:0003A2AA      MOVN    R0, R2
ROM:0003A2AC      LSL     R1, R0, #0x10
ROM:0003A2AE      ASR     R0, R1, #0x10
ROM:0003A2B0      LDR     R1, =0x3000F48
ROM:0003A2B2      LDRH    R3, [R1]
ROM:0003A2B4      ADD     R2, R0, #0
ROM:0003A2B6      BIC     R2, R3
ROM:0003A2B8      ADD     R3, R2, #0
ROM:0003A2BA      LDR     R2, =0x3000F4A
ROM:0003A2BC      STRH    R0, [R1]
ROM:0003A2BE      LSL     R1, R3, #0x10
ROM:0003A2C0      STRH    R3, [R2]
ROM:0003A2C2      CMP     R1, #0
ROM:0003A2C4      BEQ     loc_3A2D0
ROM:0003A2C6      ADD     R1, R0, #0
ROM:0003A2C8      MOV     R0, #0xF
ROM:0003A2CA      AND     R1, R0
ROM:0003A2CC      CMP     R1, #0xF
ROM:0003A2CE      BEQ     loc_3A2D2
ROM:0003A2D0      loc_3A2D0                                ; CODE
ROM:0003A2D0      ; ROM:
ROM:0003A2D0      POP     {PC}
ROM:0003A2D2      ; -----
ROM:0003A2D2      loc_3A2D2                                ; CODE
ROM:0003A2D2      LDR     R0, =unk_0
ROM:0003A2D4      LDR     R1, =0x4000208
ROM:0003A2D6      LDR     R2, =0x800010D
ROM:0003A2D8      STRH    R0, [R1]
ROM:0003A2DA      BL      sub_3E818
ROM:0003A2DE      B       loc_3A2D0
ROM:0003A2DE      ; -----
ROM:0003A2E0      off_3A2E0      DCD    unk_0                ; DATA
ROM:0003A2E4      dword_3A2E4      DCD    0x87FB901            ; DATA
ROM:0003A2E8      dword_3A2E8      DCD    0x3000F48            ; DATA

```

You see that the ldr-command now loads in the location of our trainerroutine and jumps into that by using command "bx r1" (branch). The old command "ldrh r2,[r1]" has been overwritten with it (that's why we have to execute this in our trainerroutine before jumping back to 0x3a2aa. Congrats, your Trainer is done.. (okay almost, now do a decent trainermenu! :-))

Final words

I hope I could help you to improve your trainer-hacking skills on Gameboy Advance or to even show you how it is done. As you can see this is no magic; everyone with a brain can do trainers ;-)) – Of course the main-hacking is done when searching for trainercodes in ROMs. This is sometimes not easy but fun ;-)) – So to all Trainermakers: keep up the very good work!

Thank you for reading this. And thank you Illusion&x for all the information and permission to use this in here. And sorry for my miserable english.. ugh!

Signed,

The Anonymous Man

26-Feb-2004