Imran Nazar: The Smallest NDS File

Have you ever wondered what exactly is inside a Nintendo DS ROM file, and why the simple DS demos are so much larger than their GBA equivalents? Some people have, and this page documents their exploits.

Introduction

It started innocently enough. I was looking for a small ROM which would be used to test the framebuffer display mode of DSemu, a Nintendo DS emulator. LiraNuna agreed to put a small C demo together, to fill the 'main' screen with red, demonstrating the framebuffer's use. When compiled and spliced up, the .nds ended up at around 7.5KB.

That, LiraNuna thought, was a bit large for something that did so little as his demo evidently did. Stepping through with DSemu's debugger, I noticed a whole lot of code being run which wasn't strictly required: setting up cache parameters and the stack, clearing out regions of memory, and such like. Referred to as the crt0, this code is inserted into every project, to safeguard the execution environment.

Furthermore, there was the standard ARM7 code also inserted into the .nds file, which does such things as set up the touchscreen. All this, we thought, was a bit over-the-top for a demo that was literally doing almost nothing. So, the cut-down began.

Early stages: Chopping the ARM7

First off, LiraNuna thought the functionality of the ARM7 wasn't particularly required for this demo. So, the thought process went, why not simply tell that 'sub' CPU to enter an infinite loop and not do anything? The reasoning was sound, and so the ARM7 source file was replaced with a simple assembly file, looking something like this.

ARM7 cut-down: Infinite loop

main: b main

Once put together, that reduced the size of the overall .nds file by quite a way; down to approximately 5KB. However, I still thought that was a touch large. A quick peek into the .nds file showed why that was: the sub CPU, just like the main CPU, has a crt0 automatically inserted by the build process, and this made up the vast majority of the ARM7 portion of the .nds file.

Therefore, LiraNuna took the step of subverting a part of the build process, by deleting the result of the ARM7 compilation, and replacing it with a straight binary file, encoding the infinite-loop opcode.

ARM7 cut-down: Final binary

```
0000: FE FF FF EA
```

That left the overall binary at around 4KB. Still plenty of room for improvement, I thought.

The Next Step: Assembly

The main code was still in C, and compiled to Thumb binary. Stepping through that in DSemu's debugger, I noticed a few odd things introduced by the compiler, that seemed to do very little; values being left-shifted and then right-shifted again, to no overall effect, and similar oddities. So, the next logical step was to write that portion without the intervention of the compiler, in assembly.

LiraNuna put together a first attempt at an assembly version of the program, as follows.

ARM9 cut-down: First run

```
main:
@ sets POWER_CR
mov r0, #0x4000000
orr r0, r0, #0x300
orr r0, r0, #0x4
mov r1, #0x3
str r1, [r0]

@ sets mode
mov r0, #0x04000000
mov r1, #0x00020000
str r1, [r0]

@ sets VRAM bank a
mov r0, #0x04000000
add r0, r0, #0x240
mov r1, #0x80
strb r1, [r0]

@ loop
mov r0, #0x06800000
mov r1, #0x1F
orr r1, r1, #0x8000
mov r2, #0x18000

filloop: strh r1, [r0], #0x1
subs r2, r2, #0x1
bne filloop

lforever: b lforever
```

When compiled up, that definitely made a difference; the overall ROM size dropped to approximately 1.5KB. However, I started to have an inkling that we could do better. And that's when pepsiman piped up with a suggestion: place the code *inside* the .nds header.

Going deeper: Inside the .nds file

What did pepsiman mean by that? In order to understand that, it's important to know what a .nds ROM looks like, on the inside.

File offset	Component
0000	NDS ROM header (512 bytes)
0200	ARM9 binary
0200+ARM9	ARM7 binary
0200+both	Optional file table

The conventional layout dictates that the main CPU's binary be placed after the header, and the sub CPU's binary after that. However, that doesn't have to hold true all the time; the order can be swapped, blank space can be inserted between the binaries, or after them.

That's all well and good, but *inside* the header? In order to understand that, it's required to look inside that top chunk of the file: the ROM header.

Header structure: ndstool sample output

0x00 Game title
0x0C Game code ####
0x10 Maker code
0x12 Unit code 0x00
0x13 Device type 0x00
0x14 Device capacity 0x00 (1 Mbit)
0x15 (8 bytes blank space)
0x1E ROM version 0x00
0x1F reserved 0x04
0x20 ARM9 ROM offset 0x200
0x24 ARM9 entry address 0x2000000
0x28 ARM9 RAM address 0x2000000
0x2C ARM9 code size 0x3A0
0x30 ARM7 ROM offset 0x600
0x34 ARM7 entry address 0x3800000
0x38 ARM7 RAM address 0x3800000
0x3C ARM7 code size 0x8
0x40 File name table offset 0x608
0x44 File name table size 0x9
0x48 FAT offset 0x614
0x4C FAT size 0x0
0x50 ARM9 overlay offset 0x0
0x54 ARM9 overlay size 0x0
0x58 ARM7 overlay offset 0x0
0x5C ARM7 overlay size 0x0
0x60 ROM control info 1 0x00586000
0x64 ROM control info 2 0x001808F8
0x68 Icon/title offset 0x0
0x6C Secure area CRC 0x0000 (-, homebrew)
0x6E ROM control info 3 0x0000 0x70 (16 bytes blank space)
0x80 Application end offset 0x00000000
0x84 ROM header size 0x00000200
0x88 (36 bytes blank space)
0xAC PassMe autoboot detect 0x53534150 (“PASS”)
0xB0 (16 bytes blank space)
0xC0 Nintendo Logo (156 bytes)
0x15C Logo CRC 0x9E1A (OK)
0x15E Header CRC 0xC9D3 (OK)
0x160 (160 bytes blank space)

The entries highlighted red indicate regions of empty space in the header structure. These are normally left behind during the construction of the format, to allow for expansion. In this case, however, it’s possible to make use of the blank regions in the header for the purposes of holding code.

From looking at the above output, it's simple to see that the structure of the .nds file as a whole is dictated by the entries in this header. The fact that the ARM9 binary follows the header is simply due to the setting of "ARM9 ROM offset" to `0x200`, which is the first byte in the file after the header. Similarly, the ARM7 code following the ARM9 is a simple effect of the "ARM7 ROM offset" being set to `0x600`, which corresponds to an offset in the file of 1.5KB.

Simply by changing the "ROM offset" values in this header, it's possible to change the point from which the code for the CPUs is loaded, from the default location after the header to somewhere inside the header; overwrite the zeros in that position with ARM opcodes, and load from there. It seemed a good idea by pepsiman, and viable.

LiraNuna's ARM9 code seemed quite short, but I thought I could go one better, shrinking the code down further.

ARM9 cut-down: Second run

main: mov r0,#0x04000000 ; I/O space offset mov r1,#0x3 ; Both screens on mov r2,#0x00020000 ; Framebuffer mode mov r3,#0x80 ; VRAM bank A enabled, LCD str r1,[r0, #0x304] ; Set POWERCNT str r2,[r0] ; DISPCNT str r3,[r0, #0x240] ; VRAMCNT_A mov r0,#0x06800000 ; VRAM offset mov r1,#31 ; Writing red pixels mov r2,#0xC000 ; 96k of them lp: strh r1,[r0],#2 ; Write a pixel subs r2,r2,#1 ; Move along one bne lp ; And loop back if not done nf: b nf ; Sit in an infinite loop to finish

Once assembled, this code ended up looking like the following.

ARM9 cut-down: Assembled binary

0000 01 03 A0 E3 03 10 A0 E3 02 28 A0 E3 80 30 A0 E3 0010 04 13 80 E5 00 20 80 E5 40 32 80 E5 1A 05 A0 E3 0020 1F 10 A0 E3 03 29 A0 E3 B2 10 C0 E0 01 20 52 E2 0030 FC FF FF 1A FE FF FF EA

Definitely a little smaller; now the matter remained of where to put it, along with the ARM7 binary of one opcode (`EAFFFFFE`). The ARM7 was simple enough: the first region of blank space, 8 bytes, was ample space to place this opcode. The ARM7 offset was changed, the size changed to 4, and that part was done.

The ARM9 code was similarly simple to place in: the 160 bytes of free space at the end of the header seemed more than enough to stash the binary, and all that remained was to modify the ARM9 ROM offset and size.

And that, it seemed, was that. All the code fit comfortably into the header, and the final .nds was just 512 bytes in size. Surely that was all that could be done? Not quite.

To the core: Repositioning

As it turns out, not all 512 bytes of the header are used. The 160 bytes on the end are in the header simply by convention; one might as well say that the .nds file consists of a 352-byte header, 160 bytes of padding, and then the two CPU binaries. Was it possible to fit the 56-byte ARM9 binary somewhere else inside the header, and eliminate this padding?

I started by changing the "header size" field at `0x84` to reflect the new size of the header, which would be `0x160` bytes. Then, I started inserting the opcodes, until I had something like this.

ARM9 placement: Within the header

0070 01 03 A0 E3 03 10 A0 E3 02 28 A0 E3 80 30 A0 E3 0080 00 00 00 00 A0 01 00 00 04 13 80 E5 00 20 80 E5 0090 40 32 80 E5 1A 05 A0 E3 1F 10 A0 E3 03 29 A0 E3 00A0 B2 10 C0 E0 01 20 52 E2 FC FF FF 1A 50 41 53 53 00B0 FE FF FF EA 00 00 00 00 00 00 00 00 00 00 00 00

The fields in the header at 0x80, 0x84 and 0xAC can be seen, nestled within the ARM9 code. Now, this is quite a problem; if those values correspond to valid opcodes, they may be executed, and that might prove disastrous for the state of the program.

A disassembly was called for. I loaded up the new binary in DSemu, and the debugger gave the following output:

ARM9 cut-down: Code after insertion

mov r0,#0x04000000 mov r1,#0x3 mov r2,#0x00020000 mov r3,#0x80 andeq r0, r0, r0 andeq r0, r0, r0, lsr #3 str r1,[r0, #0x304] str r2,[r0] str r3,[r0, #0x240] mov r0,#0x06800000 mov r1,#31 mov r2,#0xC000 lp: strh r1,[r0],#2 subs r2,r2,#1 bne lp cmppls r3, #0x14 nf: b nf

It seems I was fortunate. The first two AND statements will never be executed, since they depend on the ZERO flag being set, and said flag is not set by the instructions above. As for the CMP, it slots into place after the VRAM-writing loop, which is indeed fortunate; if the CMP had fallen before the BNE, the loop may have executed forever, eventually running out of VRAM to write to.

Surprisingly fortunate, I thought; I hadn't planned for such a consequence, and it had simply come about due to the size and structure of the code. Either way, I wasn't about to complain.

Conclusion

So, there we have it. The smallest .nds file you're ever likely to see, which still does something. The ARM7 sticks itself into an infinite loop, and the ARM9 fills the main-core framebuffer with red before entering its own infinite loop. I eventually got my wish, of a small framebuffer-testing demo, but it was fun to get there.

Final binary: 352 bytes

0000 4E 44 53 2E 54 69 6E 79 46 42 00 00 23 23 23 23 NDS.TinyFB..#### 0010 00 00 00 00 00 00 FE FF FF EA 00 00 00 00 00 04 . . . . 0020 70 00 00 00 00 00 00 02 00 00 00 02 44 00 00 00 p. . . . .D. . . 0030 16 00 00 00 00 00 80 03 00 00 80 03 04 00 00 00 . . . . 0040 A0 01 00 00 00 00 00 00 A0 01 00 00 00 00 00 00 . . . . 0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . 0060 00 60 58 00 F8 08 18 00 00 00 00 00 00 00 00 00 .'X. . . . . 0070 01 03 A0 E3 03 10 A0 E3 02 28 A0 E3 80 30 A0 E3 . . . . (. . . 0.. 0080 00 00 00 00 A0 01 00 00 04 13 80 E5 00 20 80 E5 . . . . 0090 40 32 80 E5 1A 05 A0 E3 1F 10 A0 E3 03 29 A0 E3 @2. . . . .).. 00A0 B2 10 C0 E0 01 20 52 E2 FC FF FF 1A 50 41 53 53 . . . .. R. . . ..PASS 00B0 FE FF FF EA 00 00 00 00 00 00 00 00 00 00 00 00 . . . . 00C0 C8 60 4F E2 01 70 8F E2 17 FF 2F E1 12 4F 11 48 .'O..p. . . ./..O.H 00D0 12 4C 20 60 64 60 7C 62 30 1C 39 1C 10 4A 00 F0 .L 'd'|b0.9..J.. 00E0 14 F8 30 6A 80 19 B1 6A F2 6A 00 F0 0B F8 30 6B ..0j. . . j.j. . . .0k 00F0 80 19 B1 6B F2 6B 00 F0 08 F8 70 6A 77 6B 07 4C . . . k.k. . . .pjwk.L 0100 60 60 38 47 07 4B D2 18 9A 43 07 4B 92 08 D2 18 "8G.K. . . C.K. . . . 0110 0C DF F7 46 04 F0 1F E5 00 FE 7F 02 F0 FF 7F 02 . . . F. . . . . 0120 F0 01 00 00 FF 01 00 00 00 00 00 04 00 00 00 00 . . . . 0130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . 0140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . 0150 00 00 00 00 00 00 00 00 00 00 00 00 1A 9E 7B EB . . . .{.

http://imrannazar.com/content/files/TinyFB.nds

Two9A, with thanks to LiraNuna and pepsiman

Article dated: 22nd Sep 2006

- Articles
- Programming
- Fiction
- Get in touch

Get the RSS feed