Inferno DS : Inferno port to the Nintendo DS

Salva Peiró
Valencia, Spain
saoret.one@gmail.com

September 14, 2008

Abstract

This document describes the work performed in the Inferno DS port.
It is organised as follows: starts with the background and the motivation for this work, continues a DS hardware overview, and then discusses the development process, focusing on the setup and development of dis applications running on the DS. At the end the conclusions and future work are presented.

1. Background

The DS [1] native port of Inferno [2] was started by Noah Evans for GSoC 2007 [3], at the the end of the GSoC the port was starting to be usable under the no$gba [4] emulator where it was possible to interact with Inferno's window manager: **wm(1)** [1] using the touch screen, but when running on the "real" DS touch wasn't working.

In spite of this all the basic functionality required to develop; like the ability to debug using the *print* statement had already been added, also emulators are of great help to save test time.

1.1. Motivation

Besides sharing the same motivation stated by Noah Evans on his GSoC 2007 application [3], there where a few more reasons to prefer a native port:

1. After checking that **emu(1)** on DSLinux [5] was not viable as when running with graphics the program crashed due to out of memory errors. There was the curiosity about the benefits of a native port, both in development and use.

2. Overcome limitations of some homebrew programs: like no multi-tasking and the benefits of having a coherent system and with a standard set of tools.

3. Have a "real" testbed for limbo applications, which could benefit applications like those developed in the inferno-lab [6].

2. DS Overview

What follows is a small overview of the Nintendo DS's hardware organized in three subsections: the system processors, it's inter-communication mechanisms, and last the built-in devices (and expansions).

2.1. Processors

The DS has two 32 bits ARM [7] processors, one more powerful ARM946E-S @ 66MHz which is in charge of the video and performs the main computations, and another ARM7TDMI @ 33MHz which acts as a slave to deal with the remaining devices: wireless, audio, touch, wireless, power management, etc.

The system is shipped with the following internal memory:

[1] the notation **page(section)**, refers to Inferno manual pages [14]

- 4096KB Main ARM9 RAM
- 96KB Main ARM7 WRAM (64Kb + 32K mappable to NDS7 or NDS9)
- 60KB TCM/Cache (TCM: 16K Data, 32K Code) (Cache: 4K Data, 8K Code)
- 656KB Video RAM (allocateable as BG/OBJ/2D/3D/Palette/Texture/WRAM memory)
- 256KB Firmware FLASH (512KB in iQue variant)
- 36KB BIOS ROM (4K NDS9, 16K NDS7, 16K GBA)

For more details see [8][GBATEK, NDS Overview].

2.2. Communication

Processor communication in the DS can be performed using this methods, together with their combinations:

- Shared memory: The 4Mb of ARM9 RAM starting at 0x02000000, can be shared by both processors, it's important to note that one of the cpus can be given priority (using the `EXMEMREG` register) over the other when concurrently accessing the memory.
- Hardware fifos: The DS fifo controller allows receiving/sending of 32 bit values from/to each cpu. This can be done in a full-duplex manner, where each cpu has a destination queue which stores the values sent by the other cpu. Notification of activity in the queues is performed through IRQ's to the respective cpu.

  This mechanism is crucial as it allows sending messages to request actions, this is used for example to read/write the RTC, obtaining the touch coordinates, perform wifi tasks and request audio samples to be played or recorded to the ARM7 cpu.
- Sync interrupt: The Sync IRQ is a simple mechanism which allows one cpu (local) to generate an IRQ to the other (remote) cpu, this can be use for example for emulating wifi rx activity as the ARM7 detects when a packet has been received and informs the ARM9 using Sync.

Given that accessing shared memory generates wait states to the cpu with less priority, it must be used with care, one approach which works well is using it in combination with fifos, by passing fifo messages with pointers to shared memory. This is analog to the function call by value or by reference.

See [8][GBATEK, DS Inter Process Communication (IPC)] for a more detailed description.

2.3. Devices

The system is has the following built-in devices:

- Video: There're two LCD screens (each 256x192 pixel, 3 inch, 18bit color depth, backlight), each of the screens has a dedicated 2D video engine, plus one 3D video engine which can be assigned to each of the screens.
- Sound: There're 16 sound channels (16x PCM8/PCM16/IMA-ADPCM, 6x PSG-Wave, 2x PSG-Noise), output can be directed either to: two built-in stereo speakers, or to a headphones socket, while input can come either from: a built-in microphone, or microphone socket.
- Controls: Interacting with the DS is achieved through a gamepad and a touchscreen: the gamepad provides 4 direction keys plus 8 buttons, while the touchscreen on the lower LCD screen can be used as a pointing device.

- Networking: Wifi IEEE802.11b, networking is provided by the RF2958 (aka RF9008) chip from RFMD. The main drawback of wifi, is that there is no documentation about it's interfacing and programming, instead all the code, information known has been reverse engineered. All the information is gathered in [8][GBATEK, DS Wireless Communications] and also in the **dswifi** project and DSLinux [5]
- Specials: Some aditional devices include: Built-in Real Time Clock, Power Managment Device Hardware divide and square root functions and CP15 System Control Coprocessor (cache, tcm, pu, bist, etc.)
- External Memory: There're two available slots: NDS slot (slot-1) and GBA slot (slot-2) which are the prefered way for plugging expansion cards and other devices. Their most common usage is to provide storage support to sd/tf cards. But there're also devices like **Dserial**, **CPLDStarter** or **Xport** [9], which provide uart, midi, usb and standard digital i/o interfaces together with CPLDs or FPGAs.

see [8][GBATEK, NDS Hardware Programming].

3. DS Port

This section describes the idiosyncrasies of the DS port, in particular those related with the setup, kernel and application development.

3.1. Environment

The development environment is the default shipped with Inferno. The compiler used is `5{a,c,l}` compiler which forms part of the *Inferno and Plan 9 compiler suite* [11]. It is used to build the ARM [12] binaries for both cpus: ARM7, ARM9, together with the companion tools: `mk, acid, ar, nm, size, etc.` which are used for building, debugging and examining the resulting binaries.

The only special tool required is `ndstool` [10] which is used to generate a bootable image to be launched by the NDS loader running on the DS, for this purpose the image contains everything required to describe how to boot the code, this includes: the ARM7 and ARM9 binaries and their correspoding load addresses, entrypoints, etc.

3.2. Communication: Fifos IPC

As it's been explained the DS system is composed by two cpus, this poses a problem when sharing the hardware devices between cpus, to eliminate this problem the devices are assigned to one cpu or the other, and example of this is the SPI (Serial Peripheral Interface) owned by the ARM7, there are some (a lot, really) of devices accessed through SPI: touch, wifi, rtc, firmware, power management, audio, ... The same happens with the lcd hardware which is owned by the ARM9, as a consequence of this it's impossible to use the print statement from ARM7.

To overcome this problems the adopted solution has been to use the available communication mechanism: fifos and shared memory, to provide an interface which allows communication of both cpus, and by extension to provide access to devices not owned by the cpu.

The interface is analog to function calls, that is each message is associated with a function which performs the work requested by the message. For the sake of simplicity the function and it's arguments are encoded into a 32 bit message, where the message encoding is as follows:

```
msg[32] := type[2] | subtype[4] | data[26],
field[n] refers to a field of n bits of length

type[2] := 00: System, 01: Wifi, 10: Audio, 11: reserved.
subtype[4] := 2^4 = 16 type specific sub-messages.
data[26] := data/parameters field of the message.
```

This structure has to accomodate all the required information shared by both cpus, thus what follows is some reasoning behind the message encoding, basically it's main purpose is to have a readable and easy to understand and manipulate notation.

`type[2]` is used to have messages organised in 4 bit types: System, Wifi, Audio and a Reserved type.

`subtype[4]` is used to further cualify the message type.

For example, given message `type[2] = Wifi` there're several actions to be performed like:

- initialising the wifi controller
- preparing for sending/receiving a packet
- setting the wifi authentication parameters
- ...

these can be encoded using the 16 available message `subtype[4]`'s.

`data[26]` the lenght of the data field is not choosen 'at random', instead it's choosed as the minimum size which can allow passing of ARM9 RAM addresses @ 0x02000000, 4Mb. This allows passing of "pointers", which can hold all the required arguments. Note: this will have to be revised when using memory expansions @ 0x08000000, 16 Mb

3.2.1. Fifos: send a msg

Here's an example extracted from devrtc.c executed by the ARM9 side to read the ARM7 RTC.

```
int
nbfifoput(ulong cmd, ulong data)
{
    if(FIFOREG->ctl & FifoTfull)
        return 0;
    FIFOREG->send = (data<<Fcmdlen|cmd);
    return 1;
}
...
ulong secs;
nbfifoput(F9TSystem|F9Sysrrtc, (ulong)(&secs));
```

Moreover as the fifos hardware and the implemented interface are simetric, the same code can be used by the the ARM7 to *sprint* strings and send them to the ARM9, which will be able to output them to the LCD using *print*:

```
int
print(char *fmt, ...)
{
    int n;
    va_list ap;
    char *sd  = SData;

    memset((void*)s, '\0', PRINTSIZE);

    va_start(ap, fmt);
    n = vsprint(s, fmt, ap);
    va_end(ap);

    while(!nbfifoput(F7print, (ulong)s));
    return n;
}
...
print("batt %d aux %d temp %d\n", IPC->batt, IPC->aux, IPC->temp);
```

3.2.2. Fifos: receive a msg

```
static void
fifotxintr(Ureg*, void*)
{
    if(FIFOREG->ctl & FifoTfull)
        return;
    wakeup(&putr);
    intrclear(FSENDbit, 0);
}

static void
fiforxintr(Ureg*, void*)
{
    ulong v;
    while(!(FIFOREG->ctl & FifoRempty)) {
        v = FIFOREG->recv;
        fiforecv(v);
    }
    intrclear(FRECVbit, 0);
}

static void
fifoinit(void)
{
    FIFOREG->ctl = (FifoTirq|FifoRirq|Fifoenable|FifoTflush);
    intrenable(0, FSENDbit, fifotxintr, nil, "txintr");
    intrenable(0, FRECVbit, fiforxintr, nil, "rxintr");
}
```

Here `fiforxintr` is executed when an message receive IRQ is triggered, then the fifo is examined to read the message, which is passed to `fiforecv` which knows the encoding of the messages, and invokes the corresponding function associated with each message.

3.3. DS kernels

The DS port shares a lot of similarities with the other Inferno's ARM ports, which have been used both as a source of inspiration and ideas. In particular with the **Ipaq** port as the platform is somehow similar to the DS: as both have touch screens, storage, audio and wireless networking, although the underlying hardware is completely different.

Still there're certain limitations inherent to the DS that make it look a small brother: like the 66 Mhz CPU clock, the 4 Mb of available RAM, small lcd displays and reduced wireless capabilties: only wep and open modes at 2.0 Mpbs, that should be examined once wifi code is fully working, to check it it affects the use of the styx(5) protocol to access remote filesystems.

Another interesting aspect is how the Inferno kernel running on the ARM9 provides devices like **pointer(3)**, **ether(3)**, **rtc(3)**, **audio(3)**, etc. which perform their work by requesting it to the ARM7 using the Fifo IPC seen above.

3.3.1. ARM7

While the ARM9 cpu has 4 Mb of RAM, which permits it to run an Inferno kernel, the smaller ARM7 has only access to 64 Kb or EWRAM (exclusive RAM).

Given this memory limitation the ARM7 can't run a Inferno kernel, but as it's been discussed above the ARM7 is required in order to access the devices owned by the ARM7. For this purpose runs specific code which basically manages the needed hardware devices and provides the fifo interface comented above in order the ARM9 can use it.

3.4. Application

At the application level the DS has some features that make it interesting:

The input interface: buttons and touchpad with the graphical output: two small lcd displays which present a challenge to developing applications for it.

This has implications in the graphical user interface to use, which is being object of experimentation in the inferno-lab [6], to find out how to best use the available lcd screens with the lower touch screen.

This opens field for interesting applications, which combine graphics, touch, networking and audio. This can include games, VoIP, music, midi synths toghether with other common uses, like: connecting/managing remote systems with **cpu(1)**, or accessing to remote resources using the **styx(5)** protocol.

3.4.1. Development

With a standard Inferno distribution placed on a sd/tf card, it's seamlessly to setup Inferno running on the Nintendo DS, as it only requires an Inferno kernel which can be distributed as .nds image available for download from the Inferno DS project site [1]. This kernel can be transfered to a sd/tf card, to be booted by the NDS loader.

This kernel provides access to the underlying hardware through devices **dev(10)**, this is: through a filesystem interface which is used by the applications to make use of the kernel services: **draw(3)**, **pointer(3)**, **ether(3)**, **audio(3)** and a specific **devdldi** which provides storage access to sd/tf cards.

With all this, the development of applications consists of the following steps:

1. setup Inferno emu on a development host: where the applications can be coded, compiled and tested, see [13] for more details.

2. test applications on a DS emulator *(optional)*: like no$gba [4] or desmume.

3. transfer applications (.dis files) to sd/tf card: to be launched after booting the Inferno DS kernel.

4. Conclusions

The main conclusion extracted during the development of the port has been how the careful design and implementation of the whole Inferno system have made the task of developing this port easier.

This has had also an effect on the tasks of locating and fixing errors, and introducing new functionality like input, storage, networking and audio which have become easier.

The benefits of the Inferno design [2] will be also noticed when developing limbo applications for the ds, as this area has been less used/tested during the development of the port.

5. Future work

As there're always things to do or rework this can be regarded as a *work in progress*, in particular the graphics side and audio are being reworked one to take advantage of both lcd screens under Inferno, and the other to improve audio playing and recording quality.

Another area that needs atention is the wireless networking, whose code needs to be tested and finished, as this will open field by communicating it with other devices, this will allow to boot remote kernels, access filesystems provided by a **emu(1)** instance running hosted, etc.

As things are being polished the work to be done will move from the kernel side to the applications side, as it's already happening with the inferno-lab [6] experiments with the Mux interface and with the quong/hexinput keyboard to ease the interaction with the system.

References

[1] Noah Evans, Salva Peiró, Mechiel Lukkien "Inferno DS: Native Inferno Kernel for the Nintendo DS". http://code.google.com/p/inferno-ds/.

[2] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard Trickey, Phil Winterbottom "The Inferno Operating System". Computing Science Research Center, Lucent Technologies, Bell Labs, Murray Hill, New Jersey USA http://www.vitanuova.com/inferno. http://code.google.com/p/inferno-os/.

[3] Noah Evans, mentored by Charles Forsyth, "Inferno Port to the Nintendo DS". Google Summer of Code 2007, http://code.google.com/soc/2007/p9/about.html.

[4] Martin Korth, "no$gba emulator debugger version". http://nocash.emubase.de/gba-dev.htm.

[5] Pepsiman, Amadeus and others, "DSLinux: port of uCLinux to the Nintendo DS". http://www.dslinux.org.

[6] Caerwyn Jones & co, "Inferno Programmers Notebook". http://caerwyn.com/ipn, http://code.google.com/p/inferno-lab

[7] ARM (Advanced Risc Machines), "ARM7TDMI (rev r4p3) Technical Reference Manual". ARM Limited, http://www.arm.com/documentation/ARMProcessorCores.

[8] Martin Korth, "GBATEK: Gameboy Advance / Nintendo DS Technical Info". http://nocash.emubase.de/gbatek.txt. http://nocash.emubase.de/gbatek.htm.

[9] Charmed Labs, "Xport". http://www.drunkencoders.org/reviews.php.

[10] DarkFader, natrium42, WinterMute, "ndstool Devkitpro: toolchains for homebrew game development". http://www.devkitpro.org/

[11] Ken Thompson, "Plan 9 C Compilers". Bell Laboratories, Murray Hill, New Jersey 07974, USA. http://plan9.bell-labs.com/sys/doc/compiler.html.

[12] David Seal, "The ARM Architecture Reference Manual", 2nd edition. Addison-Wesley Longman Publishing Co. http://www.arm.com/documentation/books.html.

[13] Phillip Stanley-Marbell, "Inferno Programming with Limbo". John Wiley & Sons 2003, http://www.gemusehaken.org/ipwl/.

[14] "The Inferno Manual". http://www.vitanuova.com/inferno/man/.