Inferno DS: Inferno port to the Nintendo DS

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

October 12, 2008

Abstract

The Inferno DS port began in 2007 as a one-man Google Summer of Code project, to make Inferno available on a standard, cheap, networked device with graphics and audio. The GSoC project attracted a small group of developers that is completing the port, to make the device fully usable for application development. This paper describes the current status of the port. It reviews the background and the motivation for the work, provides a DS hardware overview, and discusses the kernel development process, focusing on the setup and development of Dis applications running on the DS. There is plenty of scope for futher work. We hope to encourage others to contribute to the project.

1. Background

The DS [1] native port of Inferno [2] was started by Noah Evans for GSoC 2007 [3]. At the the end of GSoC the port was starting to be usable under the no$gba [4] emulator, enough that it was possible to interact with Inferno's window manager **wm(1)** [1] using the emulated touch screen. Inferno also booted and ran on a real DS, but the touch screen did not work. In spite of its limitations the port provided enough basic functionality to encourage further development. The GSoC project sparked the interest of a small group of enthusiasts to finish the port and begin work on new applications suitable for the platform. It is an Open Source project, hosted on Google Code, and supported by discussions in Google Groups and on IRC.

1.1. Motivation

The current project shares the motivation stated by Noah Evans on his GSoC 2007 application [3]: by using cheap and easily accessible hardware, native Inferno on the DS would show a wide range of users the power and possibilities of the Inferno and Plan 9 approach to building distributed systems. On other platforms, instead of a native port, we might consider hosting Inferno under an existing system, but we found 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 thus increased curiosity about the advantages of a native port for DS software development. For instance, a proper operating system would overcome limitations of some homebrew programs for the DS, such as no multi-tasking, and it would give the benefits of having a coherent system with a standard set of tools. Furthermore, it would provide a "real" testbed for Limbo applications, including those developed in the inferno-lab [6]. The DS is particularly interesting as an Inferno target because it provides WiFi networking, allowing us to have fun with multi-user games and applications, including Voice-over-IP and jukebox programs using its audio input and output.

2. DS Overview

The native port had to address unusual aspects of the Nintendo DS hardware, so some knowledge of that is helpful. What follows is a small overview of the DS hardware organized in three subsections: the system processors, its inter-communication mechanisms, and last the built-in devices (and expansions).

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

2.1. Processors

The DS has two 32-bit ARM [7] processors: an ARM946E-S running at 66MHz that is in charge of the video and performs the main computations; and an ARM7TDMI at 33MHz that acts as a slave to deal with the remaining devices, including wireless, audio, touch screen, and power management.

The system is shipped with the following internal memory:

- 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 (usable 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

The two processors in the DS can communicate using combinations of the following methods:

- Shared memory: The 4Mb of ARM9 RAM starting at 0x02000000 can be shared by both processors. It can be configured so that one cpu can be given priority over the other when they access the memory concurrently.
- Hardware FIFOs: The DS FIFO controller allows the processors to exchange 32 bit values. It allows full-duplex communication, where each cpu has a destination queue that stores the values sent by the other cpu, and interrupts notify the appropriate cpu about queue activity.

  This mechanism is crucial as it allows sending messages to request actions. This is used for example to read and write the real-time clock, obtain 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 that allows one cpu ('local') to generate an IRQ to the other ('remote') cpu. We can use that to emulate WiFi receiver interrupts: when the ARM7 detects when a packet has been received it 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. It works well in combination with FIFOs, by passing FIFO messages with pointers to shared memory. This is analogous to passing parameters by value or by reference.

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

2.3. Devices

The Nintendo DS has the following built-in devices:

- Video: There are two 3-inch backlit LCD screens, each 256x192 pixels, with 18bit color depth. Each screen has a dedicated 2D video engine, and there is one 3D video engine that can be assigned to either screen.
- Sound: There are 16 sound channels (16x PCM8/PCM16/IMA-ADPCM, 6x PSG-Wave, 2x PSG-Noise). Output can be directed either to built-in stereo speakers, or to a headphone socket. Input can come either from a built-in microphone, or a microphone socket.

- Controls: A user interacts with the DS through a gamepad and a touch screen. The gamepad provides 4 direction keys plus 8 buttons, and the touch screen on the lower LCD screen can be used as a pointing device.
- Networking: WiFi IEEE802.11b wireless networking is provided by the RF2958 (aka RF9008) chip from RFMD. The main drawback is that there is no documentation from the manufacturer about its interfacing and programming. All that is known was reverse engineered by other projects. That information is gathered in [8][GBATEK, DS Wireless Communications] and also in the **dswifi** project and DSLinux [5]
- Specials: Additional devices include: a built-in real time clock, power managment device, hardware divide and square root functions and the ARM CP15 System Control Coprocessor (controlling cache, tcm, pu, bist, etc.)
- External Memory: There are two available slots: NDS slot (slot-1) and GBA slot (slot-2), which are the preferred way to plug in expansion cards and other devices. The slots are commonly used to provide storage on SD/TF cards. There are, however, other devices such as **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 to 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}`, which forms part of the *Inferno and Plan 9 compiler suite* [11]. It is used to build the ARM [12] binaries for both the ARM7 and ARM9 cpus, 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 generates a bootable image to be launched by the NDS loader running on the DS. The image contains everything required to describe how to boot the code, which includes the ARM7 and ARM9 binaries and their corresponding load addresses and entrypoints.

3.2. DS kernels

The Inferno DS port follows the usual pattern for a port of native Inferno to a new platform for an already-supported processor. Much of the code of the Inferno native kernel is platform-independent, including the IP stack. The Dis interpreter and built-in Limbo modules are also platform-independent. That platform-independent code only needs to be compiled, which is done automatically by a `mkfile`. A relatively small amount of platform-specific code must be written. The DS port shares much of the ARM-specific code with the other ARM ports of Inferno, including the 'on the fly' compiler (JIT) for Dis for the ARM processor. There are existing ports of Inferno to the ARM, which have been used both as a source of ideas and code. Inferno's earlier port to the iPAQ is the closest existing platform to the DS: both have touch screens, storage, audio and wireless networking, The underlying hardware is completely different, however, and the DS often looks like a small brother of the iPAQ: a slower 66 Mhz CPU clock, only 4 Mb of available RAM, small LCD displays and reduced wireless capabilities.

One of the first things to address in the port was how to use the two processors. The ARM9 cpu has 4 Mb of RAM, which permits it to run an Inferno kernel, but the slower ARM7 has only access to 64 Kb or EWRAM (exclusive RAM). Given this memory limitation the ARM7 cannot sensibly run an Inferno kernel. Instead it runs specialised code that manages the hardware devices assigned to the ARM7. The ARM7 kernel is interrupt driven. During its initialisation phase, it sets device interrupts, and configures the buttons, touch screen, FIFOs, and the devices on the SPI. It then switches to a low-power mode, where it endlessly waits for interrupts to wake it. The kernel currently has 2,630 lines of C code, over half of that in its WiFi interface, and 70 lines of assembly code.

The ARM9 runs the full Inferno kernel, and provides devices like **pointer(3)**, **ether(3)**, **rtc(3)**, **audio(3)**, etc. About 6,500 lines of C code and 310 lines of assembly code is specific to either the ARM processor or the DS platform. Most of that code is in device drivers. The implementation of the device drivers is unusual: because of the division of work between the processors, the drivers must access and control many of the physical devices via the ARM7, and we discuss that next.

3.3. Communication: FIFOs IPC

To avoid conflicts that would arise if sharing the hardware devices between cpus, each device is assigned exclusively to one cpu or the other. For example, the Serial Peripheral Interface (SPI) is owned by the ARM7. Many of the peripherals are accessed through SPI, including touch screen, WiFi, rtc, firmware, power management and audio. The LCD hardware by contrast is owned by the ARM9. Consequently, the ARM9 cannot directly drive the audio device, nor can the ARM7 directly display on the screen for debugging.

To overcome this, we use the interprocessor communication mechanisms listed above – FIFOs and shared memory – to implement a simple messaging protocol that allows one cpu to access devices owned by the other. It is a Remote Procedure Call protocol: each message is associated at the receiving cpu with a function that performs the work requested by the message. For simplicity the function and its arguments are encoded into a 32 bit message as follows:

```
msg[32] := type[2] | subtype[4] | data[26], where
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.
```

The encoding was chosen to have a notation that was easy to read in the calling code, yet accommodate all the data to be exchanged between the cpus:

`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 qualify the message type.

For example, given message `type[2] = Wifi` actions to be performed include initialising the WiFi controller, setting the WiFi authentication parameters, and preparing to send or receive a packet. Those and the other operations required can all be encoded using the 16 available message `subtypes`.

`data[26]` the data field is just big enough to allow passing of pointers into the 4Mbyte shared memory. (This will have to be revised when using memory expansions @ 0x08000000, 16 Mb)

The protocol has a simple implementation. For instance, here is the low-level non-blocking FIFO put function:

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

Here is an example of its use, extracted from devrtc.c, executed by the ARM9 side to read the ARM7 RTC:

```
ulong secs;
...
nbfifoput(F9TSystem|F9Sysrrtc, (ulong)&secs);
```

Because the hardware interface to the FIFO is the same for each processor, similar code can be used by the the ARM7 in the other direction, for instance to send a string to the ARM9 to *print* on the LCD. (The code is not identical because the ARM9 kernel environment includes scheduling.)

The interrupt-driven part of the FIFO driver is also straightforward. An extract is shown below to give the flavour:

```
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.

3.4. Graphics

The DS has two LCD screens, but the **draw(3)** device currently provides access only to the lower screen, because it is the only touch screen in the DS, and mapping touch screen coordinates to screen coordinates (pixels) makes obvious sense to a user: touching the screen refers to that point on the screen.

The DS port could also draw on the upper screen, but it will take some experimentation to determine how to best use both screens so the result still makes sense to both user and programmer. For example, although Limbo's **draw(2)** does not require that everything drawn be accessible through **/dev/pointer**, existing interactive applications effectively assume that.

One interesting alternative is to use the touch screen coordinates as relative instead of absolute: this would provide access to both screens, and visual feedback can be provided by a software cursor.

3.5. Memory

Having 4 Mb of RAM limits the programs that can be run. To overcome this memory limitation, it is possible to use slot-2 memory expansions; the expansions can add between 8 Mb and 32

Mb of RAM, Unfortunately, owing to the slot-2 bus width it can only perform 32-bit and 16-bit writes; when an 8-bit write is performed it results in garbage written to memory.

This problem is circumvented in DSLinux [5] by modifying the compiler to replace `strb` instructions with `swpb`, with appropriate changes to surrounding code. We might be able to do the same in the Inferno loader `5l` (since that generates the final ARM code), but failing that, make a similar change to the compiler `5c`.

3.6. DLDI

The Dynamically Linked Disc Interface (DLDI)[16], is a widespread way of accessing storage SD/TF cards. It provides the IO functions required to access storage independently of which boot card is being used. When a `file.nds` file is booted, the boot loader auto-patches the DLDI header contained inside the `file.file` with the specific IO functions for this card.

This has been partially implemented in the DS port, where the `devdldi.c` file provides a suitable DLDIhdr, which is properly recognised and patched by the boot loader.

The problem with this approach is that the DLDI patched code (*arm-elf*) contains instructions which modify a critical register without restoring it afterwards, which would panic the kernel.

For that reason, at this moment the DLDIhdr is only used to detect the card type and then select one of a set of compiled-in drivers, one for each type of card.

3.7. Application

As usual for Inferno ports, the existing Dis files for applications run unchanged (subject to available resources). At the application level the DS has some features that make it interesting.

User input comes from various buttons, and the touch screen. Graphical output is on two small LCD displays. As noted above, having two displays but only one with a touch screen presents a different graphical interface from the one that applications (and users) expect. This is currently the object of experimentation in the inferno-lab [6].

Whichever approach is chosen, being able to run Limbo applications in the full Inferno environment on the DS already opens the field for interesting applications, which combine graphics, touch, networking and audio. This can include games, VoIP, music, MIDI synths, and other more common uses, such as connecting to remote systems with **cpu(1)**, and managing them from the DS, or accessing remote resources using the **styx(5)** protocol.

3.8. Setting up the development environment

It is easy to set up Inferno to run on the Nintendo DS. An Inferno kernel that can be distributed as an `.nds` image is available for download from the Inferno DS project site [1]. A standard Inferno distribution is placed on an SD/TF card, and the .nds kernel image can be copied to an SD/TF card, to be booted by the NDS loader.

This kernel provides access to the underlying hardware through Inferno's normal device interface, namely through a file system interface that is used by applications to access most kernel services. The kernel includes the normal Inferno interfaces for **draw(3)**, **pointer(3)**, **ether(3)** and **audio(3)**, and a DS-specific **devdldi** that 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. Most of the kernel code is portable, including the whole of the Dis virtual machine, and just needs to be compiled. The platform-specific kernel code for any native port is fairly

small (on the order of a few thousand lines). There was already existing support for the ARM processor, and a few sample ports to ARM platforms to act as models. The device driver interface is simple and modular.

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. Emulators are still of great help to save test time.

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 and tested during the development of the port.

5. Future work

This project is *work in progress*, and significant things remain to do. There are undoubtedly places where a simple-minded implementation just to get things going needs to be redone. For example, the graphics implementation is being extended to allow Inferno to take advantage of both LCD screens, and the audio driver is being reworked to improve playing and recording quality.

One big task is to finish and test the wireless networking code. The DS will be much more interesting once it can communicate with other devices, because Inferno comes into its own in a networked environment. That will allow it to access file systems and devices provided by an **emu(1)** instance running hosted elsewhere. We can also speed development by booting remote kernels. The wireless provides only WEP and open modes at 2.0 Mbps. Once the WiFi code is fully working, it will be interesting to see how the relatively low data rate (in current terms) affects the use of the **styx(5)** protocol to access remote filesystems.

As low-level device support is completed, effort will shift from the kernel side to the applications side. Indeed, that is already happening with the inferno-lab [6] experiments with the Mux interface and with the QUONG/HexInput [15] keyboard to ease interaction with the system through the touch screen.

Please join in! [1]

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/.

[15] http://www.strout.net/info/ideas/hexinput.html.

[16] Michael "Chishm" Chisholm, Dynamically Linked Disc Interface. http://dldi.drunkencoders.com/index.php.