Free operating systems for Nintendo DS: μClinux and RTEMS

J.-M. Friedt & G. Goavec-Merou

Association Projet Aurore

jmfriedt@femto-st.fr

slides available at http://jmfriedt.free.fr

July 10, 2009

Introduction

- Hardware is hardly understandable in modern computers: serial (SATA, I^2C) or fast (PCI) protocols difficult to understand
- Parallel protocols – easiest to use – are no longer available (ISA, processor bus 6502 or Z80, parallel port)
- Older, well documented hardware is still accessible on handheld gameconsoles.
- availability of an opensource emulator `desmume`

$\Rightarrow$ use the handheld game console Nintendo Dual Screen (NDS) for getting familiar with hardware/software interaction and instrument developement

This is **not** "yet another platform on which to run GNU/Linux". The principles described here are valid for any platform: the NDS is widely available and "low" cost, well documented.

Available hardware

- Two processors: ARM9 CPU and ARM7 coprocessor (bus sharing) $\Rightarrow$ ARM crosscompiler (`gcc`, `newlib` and `binutils`, appropriately patched depending on the target)
- 4 MB RAM (DS and DSLite: avoid DSi)
- wifi interface, but *no* asynchronous serial port (RS232)
- a legacy bus for Gameboy Advance compatibility: slot2
- a synchronous serial bus for reading game software: slot1 cartridge.

Requirement: get a cartridge for executing our own programs (games) on the NDS (M3DS Real).

Objective: display and transfer over the wifi network some "real world" data (sensor node, robotics ...)

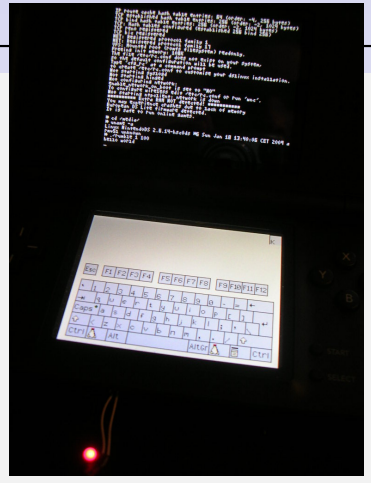
Let's start with a familiar environment

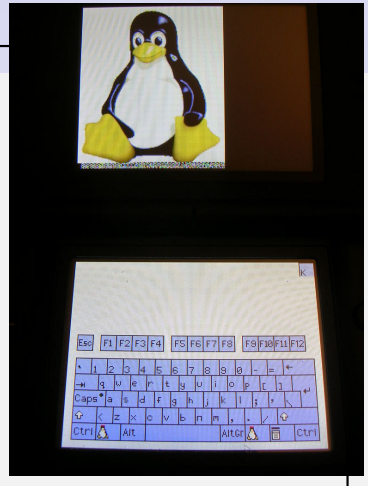
- Ready to use DSLinux image: Linux port to ARM9 processor thanks to the availability of `gcc` [1] (+newlib & binutils + patches)
- Familiar environment: posix, most hardware accessed through hardware modules `/dev`, no need to understand the underlying architecture, keyboard for typing commands on the bottom touchscreen
- Pre-compiled toolchain and linux image available at `http://kineox.free.fr/DS/dslinux-dldi.tgz`

$\Rightarrow$ shell and "simple" interfaces such as framebuffer (`/dev/fb0`) are readily available thanks to the work of the DSLinux team

[1] toolchain compiled for x86 platform: `http://stsp.spline.de/dslinux/toolchain`

Let's start with a familiar environment



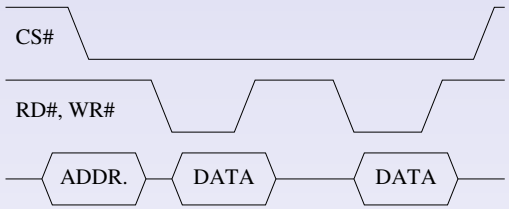


```
struct fb_var_screeninfo sinfo;
unsigned short * s_ptr;

inline void draw_pixel(int x, int y, int color)
{unsigned short *loc = s_ptr + \
    ((y+sinfo.yoffset)*sinfo.xres)+x+sinfo.xoffset;
 *loc = color;     // 5R, 5G, 5B
 *loc |= 1 << 15; // transparence ?
}

int main(int argc, char *argv[])
{char c;
 screen_fd = open("/dev/fb0", O_RDWR);
 ioctl(screen_fd, FBIOGET_VSCREENINFO, &sinfo);

 s_ptr = mmap(0,screen_height*screen_width/8,PROT_READ|PROT_WRITE,MAP_SHARED,screen_fd, 0);
 [...]
}
```

- Understand the frambuffer data organisation (16 bit depth=5R5G5B)
- common interface with classical framebuffer: port from one architecture to another is mostly transparent *as long as an hardware abstraction layer exists*
- /dev/ and /proc are supported

Slot2 cartridge bus

All processors are based on the same architecture: a data bus holds the information (what), the adress bus the location where the data are fetched/stored (where), the control signals indicate which operation to perform (read, write, interrupt, dma ...)
The older gameboy advance – including ARM7 bus – was well documented



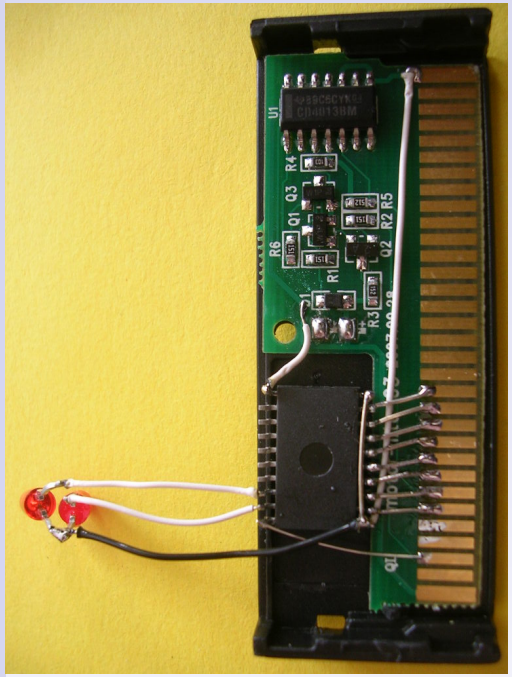
Let's try to use it to interface to the world ...

Two steps:

1. write informations to communicate with the world (no risk of damaging the hardware since output only)
2. read informations to gather informations on the environment (more challenging since the hardware must comply with the other peripherals of the CPU: but timing, high impedance bus)

Blinking LED

- No need for fancy hardware: use a Rumble Pack cartridge
- Program example for accessing a hardware address (memory mapped I/O in Freescale architecture):

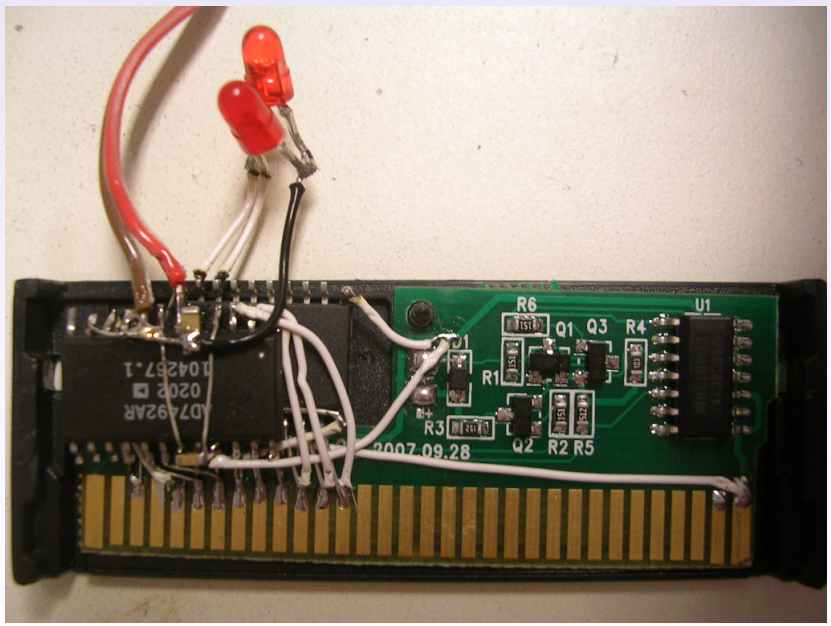


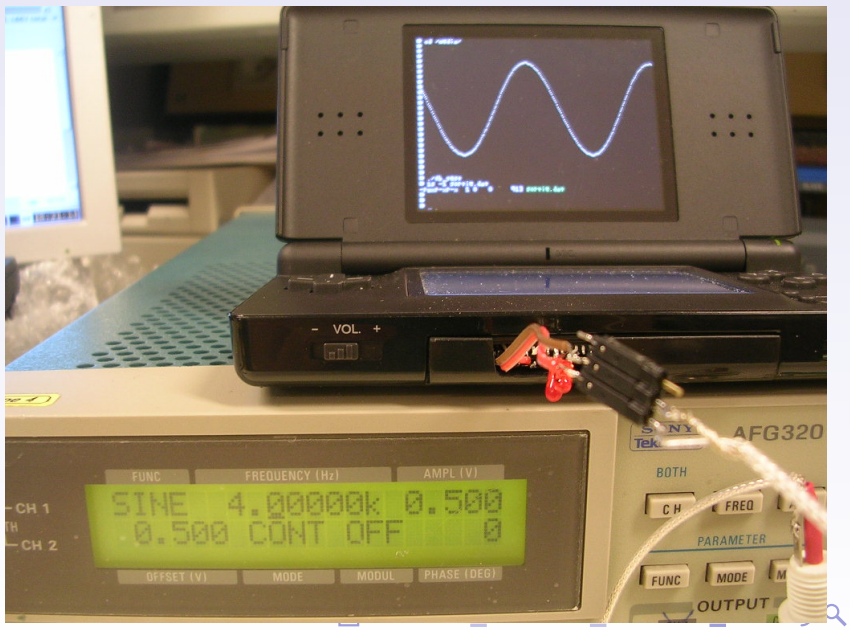
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>

int main(int argc,char **argv)
{printf("demo rumble : 1/3=%f\n",1./3.);
 if (argc>1) {
   *(unsigned short*)(0x8000000)=(unsigned short)atoi(argv[1]);
   sleep(1);                          // active le moteur sur argv→
         ↪[1]=2
   *(unsigned short*)(0x8000000)=0;
 }
 return(0);
}
```

1. define which address is associated with which hardware (address decoder)
2. define the data size (`*(unsigned shot*)`)
3. define which value to put on the data bus
4. the control signal are automagically generated by the processor

Data acquisition

- The only additional trick is to *keep the data bus lines high impedance* when you are not supposed to talk
- Use of an analog to digital converter *in parallel* (bus sharing thanks to RD#/WR# and CS#) to the latch
- This example: fast analog to digital conversion (ADC), theoretically 1 MS/s, practically half that speed
- ADC is more fancy than a latch: start conversion, wait, read conversion result (fixed delay or interrupt = kernel module)





ADC control example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#define TAILLE 255

int main(int argc,char **argv)
{int f,taille=TAILLE;
 volatile int k;
 char *c;

 c=(char*)malloc(TAILLE); // demontre malloc en l'absence de MMU
 for (f=0;f<TAILLE;f++)
   {*(unsigned short*)(0x8000000)=(unsigned short)0;
    for (k=0;k<10;k++) {} // NE PAS compiler en -O2
    // usleep(7);          // l'appel a usleep est trop long !
    c[f]=*(unsigned short*)(0x8000000)&0xff;
   }
 for (f=0;f<TAILLE;f++) printf("%x ",c[f]);printf("\n");
 return(0);
}
```

- user-space memory mapped device with fixed delay (uClinux $\Leftrightarrow$ no MMU)
- kernel space with fixed daly
- hardware interrupt generated by end-of-conversion (slow !)

Problem with uClinux

Conclusion of DSLinux: **memory footprint too large** for a 4 MB system $\Rightarrow$ follow the trend and add RAM, or **find a better use of the available resources**.

- most programs in `busybox` will run out of memory, even `less` ! only a few hundred kB remain after loading the kernel
- no luck in using portable graphic interfaces (SDL, Qtopia)
- one option is to use the memory expansion pack ... on slot 2, *i.e.* no access to the ARM9 bus left

Yet another solution: use another development environment, POSIX compatible yet with a small footprint.

RTEMS on NDS

- RTEMS is an *executive environment* for embedded devices, *i.e.* a framework for developing a monolithic application based on multiple modules.
- From the developper point of view, similar to an operating system, but no memory management, no scheduler, no dynamic loading of application ...
- Yet availability of OS functionalities such as TCP/IP stack, file format, threads ... even a shell & user input interface (Graffiti) !
- Dedicated functionalities for embedded system debugging: CPU use, stack use
- Real time + strong support (used by large national & international agencies)
- compiled with gcc (+ patches) $\Rightarrow$ familiar environment

An NDS BSP has been developed by M. Bucchianeri, B. Ratier, R. Voltz & C. Gestes [2]

[2] `http://www.rtems.com/ftp/pub/rtems/current_contrib/nds-bsp/manual.html`

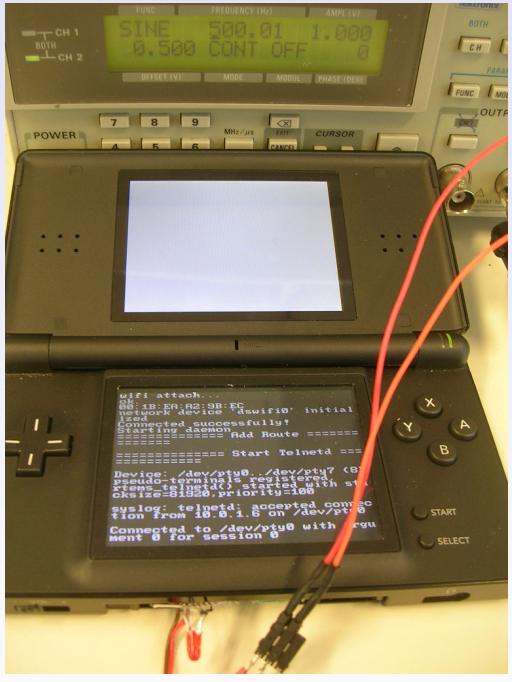
Basic program

- RTEMS provides `stdxx` and POSIX compatibility
- runs on `desmume` for all non-hardware related developments
- provides an hardware abstraction layer

but not a fully dynamic operating system so the resources needed by the application must be defined in the program

Hello World

framebuffer display



wireless data transfer

```
#include <bsp.h>
#include <stdlib.h>
#include <stdio.h>
#include <nds/memory.h>

rtems_id timer_id;
uint16_t l=0;

void callback()
{ printk("Callback %x\n",l);
  (*(volatile uint16_t*)0x08000000)=l;
  l=0xffff-l;
  rtems_timer_fire_after(timer_id, 100, callback, NULL);
}

rtems_task Init(rtems_task_argument ignored)
{ rtems_status_code status;
  rtems_name timer_name = rtems_build_name('C','P','U','T');

  printk( "\n\n*** HELLO WORLD TEST ***\n" );
  (*(vuint16*)0x04000204) = ((*(vuint16*)0x04000204) & ~ARM7_OWNS_ROM); // bus access to ARM9

  status = rtems_timer_create(timer_name,&timer_id);
  rtems_timer_fire_after(timer_id, 1, callback, NULL);
  rtems_stack_checker_report_usage(); // requires #define CONFIGURE_INIT

  printk( "*** END OF HELLO WORLD TEST ***\n" );
  while(1) ;
  exit( 0 );
}

/* configuration information */
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE

/* configuration information */
#define CONFIGURE_MAXIMUM_DEVICES                       40
#define CONFIGURE_MAXIMUM_TASKS                         100
#define CONFIGURE_MAXIMUM_TIMERS                        32
#define CONFIGURE_MAXIMUM_SEMAPHORES                    100
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES                20
#define CONFIGURE_MAXIMUM_PARTITIONS                    100
#define CONFIGURE_MAXIMUM_REGIONS                       100
```

Framebuffer access & math lib.

Framebuffer access similar to DSLinux: minor changes to adapt to the new device naming convention (exemple provided by M. Bucchianeri)

```
inline void draw_pixel(int x, int y, int color)
{ uint16_t* loc = fb_info.smem_start;
  loc += y * fb_info.xres + x;
  *loc = color;  *loc |= 1 << 15;   // 5R, 5G, 5B
}
...
struct fb_exec_function exec;
int fd = open("/dev/fb0", O_RDWR);
exec.func_no = FB_FUNC_ENTER_GRAPHICS;
ioctl(fd, FB_EXEC_FUNCTION, (void*)&exec);
ioctl(fd, FB_SCREENINFO, (void*)&fb_info);
...
#define CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE

rtems_driver_address_table Device_drivers[] =
{ CONSOLE_DRIVER_TABLE_ENTRY,
  CLOCK_DRIVER_TABLE_ENTRY,
  FB_DRIVER_TABLE_ENTRY,
  { NULL,NULL, NULL,NULL,NULL, NULL }
};
```

Drawing of a fractal displaying the resolution of a 3rd order polynom:

1. the NDS can be used for useful calculations including solving equations: floating point calculation emulation
2. access to framebuffer
3. adding a new command to the RTEMS shell

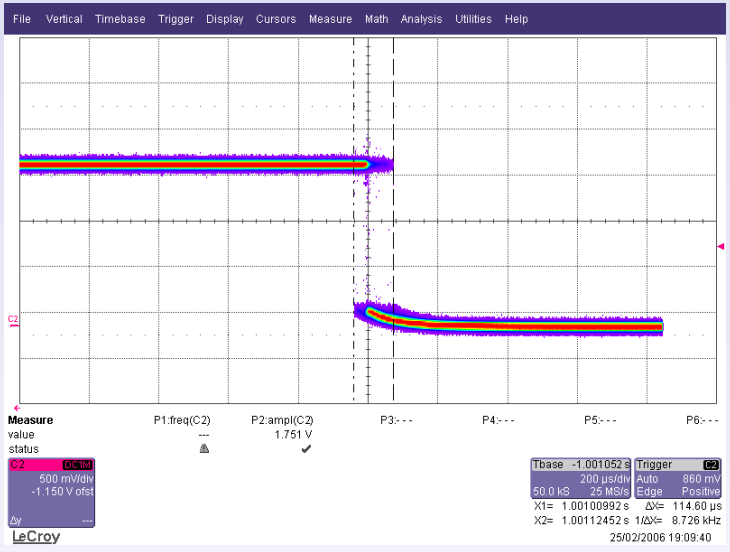
Hardware access (GPIO & ADC)

Bus control must be granted to the ARM9 CPU

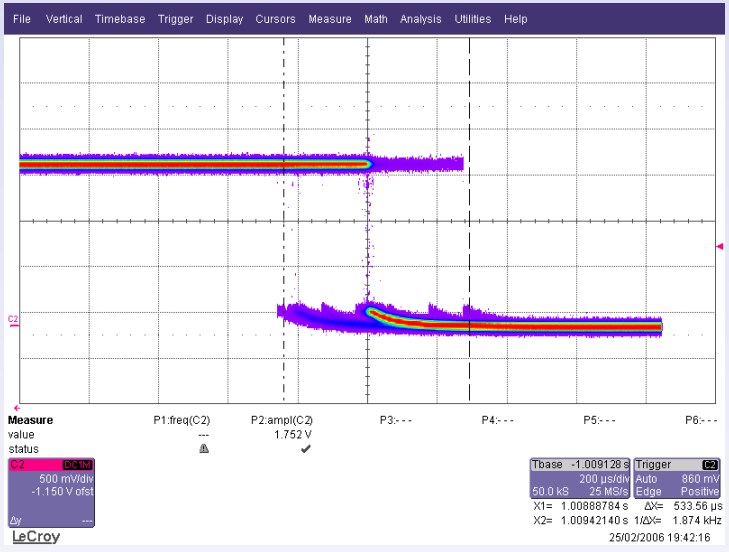
```
#include <nds/memory.h>

[...]
void callback()
{ printk("Callback %x\n",l);
  (*(volatile uint16_t*)0x08000000)=l;
  l=0xffff-l;
  rtems_timer_fire_after(timer_id, 100, callback, NULL);
}

rtems_task Init(rtems_task_argument ignored)
{rtems_status_code status;
 rtems_name timer_name = rtems_build_name('C','P','U','T');

// cf rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/source/arm9/rumble.c
// sysSetCartOwner(BUS_OWNER_ARM9);
// defini dans rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/include/nds/memo
  (*(vuint16*)0x04000204) = ((*(vuint16*)0x04000204) & ~ARM7_OWNS_ROM);

  status = rtems_timer_create(timer_name,&timer_id);
  rtems_timer_fire_after(timer_id, 1, callback, NULL);
[...]
}
```

Real time ?

- Real time is defined as a system with *maximum latency* between a signal and the processing of the information.
- The shorter the better, but an upper boundary must never be reached



RTEMS

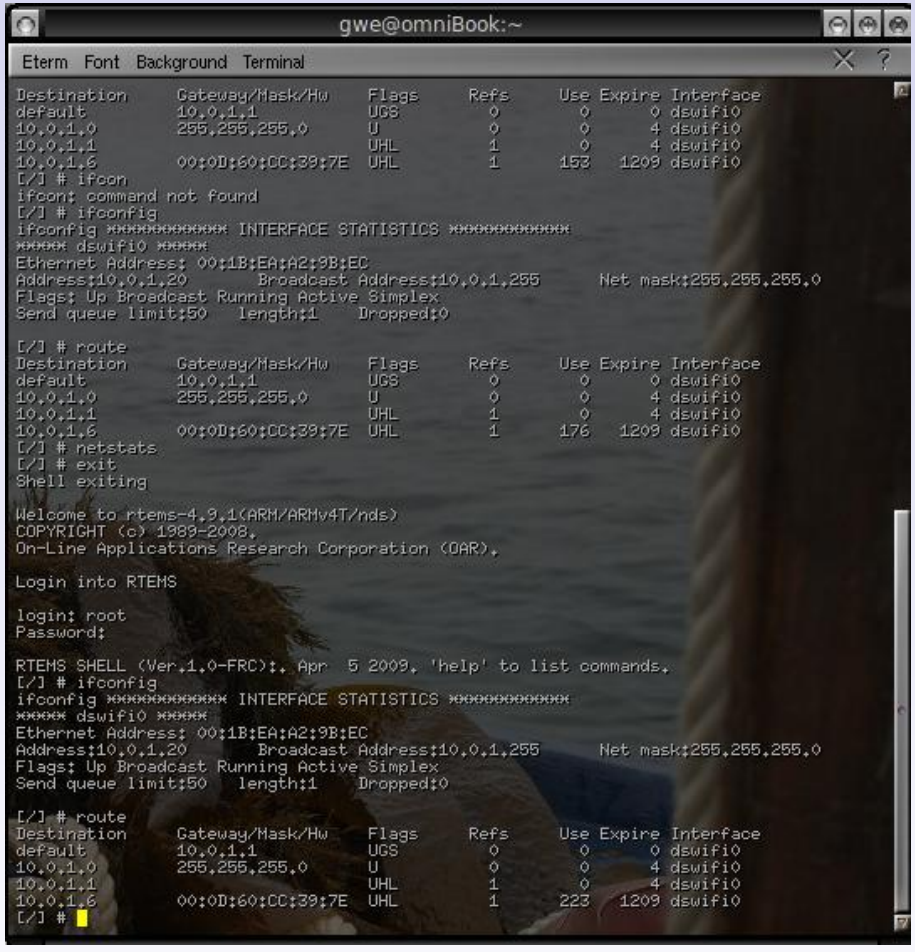


DSLinux

Three threads, two for blinking diodes and one running the Newton fractal drawing for $\simeq$10 seconds upon user request.
$\rightarrow$ DSLinux diplays latency shift during contect *changes*.

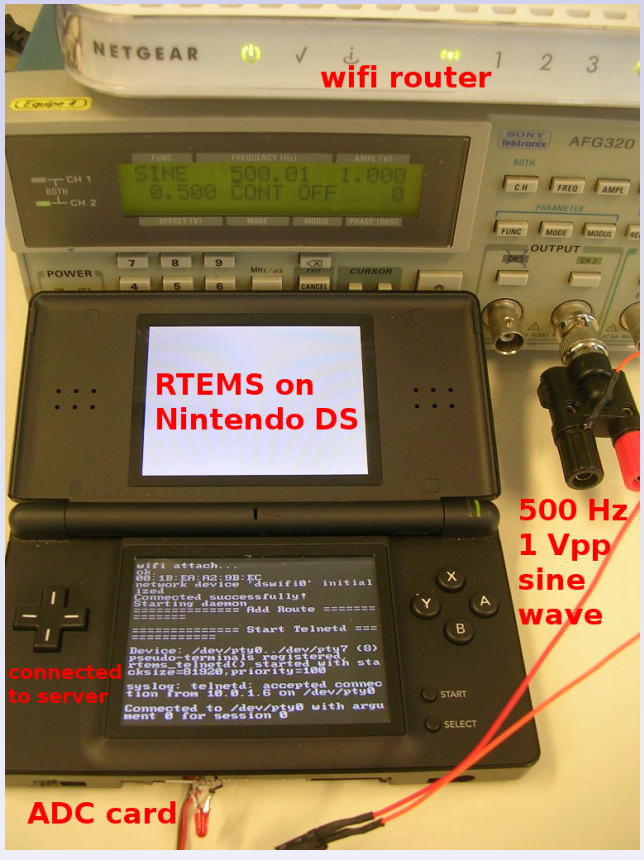
TCP/IP over wifi networking

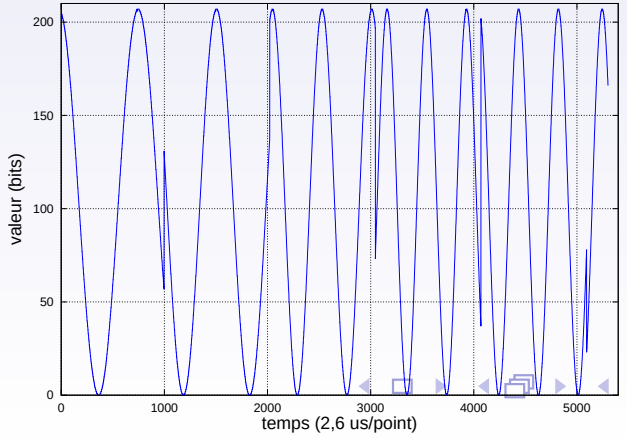
- correction of a bug in the data structure handling TCP/IP packets
- convenient connection to the shell (as opposed to the Graffiti interface)
- use of the wireless interface for data transfer (the `telnetd` server can handle other applications than a shell)



TCP/IP over wifi networking

- correction of a bug in the data structure handling TCP/IP packets
- convenient connection to the shell (as opposed to the Graffiti interface)
- use of the wireless interface for data transfer (the `telnetd` server can handle other applications than a shell)





TCP/IP over wifi networking

1. ifconfig:

```
static struct rtems_bsdnet_ifconfig netdriver_config = {
    RTEMS_BSP_NETWORK_DRIVER_NAME,
    RTEMS_BSP_NETWORK_DRIVER_ATTACH,
    NULL,             /* No more interfaces */
    "10.0.1.20",      /* IP address */
    "255.255.255.0", /* IP net mask */
    NULL,             /* Driver supplies hw addr */
};
```

2. route:

```
/* Network configuration */
struct rtems_bsdnet_config rtems_bsdnet_config = {
   &netdriver_config,
   NULL,             /* do not use bootp */
   0,                /* Default network task priority */
   0,                /* Default mbuf capacity */
   0,                /* Default mbuf cluster capacity */
   "rtems",          /* Host name */
   "trabucayre.com",/* Domain name */
   "10.0.1.1",       /* Gateway */
   "10.0.1.13",      /* Log host */
   {"10.0.1.13" },   /* Name server(s) */
   {"10.0.1.13" },   /* NTP server(s) */
};
```

3. telnetd: (server)

```
rtems_telnetd_initialize(
  rtemsShell,     /* "shell" function */
  NULL,           /* no context necessary for echoShell */
  false,          /* listen on sockets */
  RTEMS_MINIMUM_STACK_SIZE*20, /* shell needs a large stack */
  1,              /* priority */
  false           /* telnetd does NOT ask for password */
);
```

4. application:

```
void rtemsShell(char *pty_name, void *cmd_arg) {
        printk("========= Starting Shell =========\n");
        rtems_shell_main_loop( NULL );
        printk("========= Exiting Shell =========\n");
}
```

OR (another server than the shell)

```
void telnetADC( char *pty_name, void *cmd_arg) {
  char *c; int f;
  c=(char*)malloc(TAILLE);
  while (1) {
    for (f=0;f<TAILLE;f++) {
      *(unsigned short*)(0x8000000)=(unsigned short)0;
      c[f]=*(unsigned short*)(0x8000000)&0xff;
     }
  for (f=0;f<TAILLE;f++) printf("%x ",c[f]);
  printf("\n");
  }
}
```

Conclusion

- A game console provides the resources typically found in high grade embedded systems (*e.g.* routers, digital cameras) and hence a playground to get familiar the techniques associated with scarce resource
- hardware bus still understandable and usable for hardware interfacing
- once again, gcc is our friend, with a port for the ARM-architecture
- DSLinux requires too much memory to perform any useful function
- switch to a low memory footprint executive environment: RTEMS
- use and expand some of the available demonstration applications to suite most of our needs (framebuffer, text mode interface, character input ...)
- patched RTEMS to add full wifi communication functionality $\Rightarrow$ initial goal reached, *i.e.* wireless transmission of physical quantities obtained on an A/D converter

Acknowledgement

- Pierre Kestener (CEA/IRFU, Saclay, France) mentioned the NDS BSP of RTEMS
- M. Bucchianeri answered our questions concerning the use of the RTEMS BSP
- Santa Claus brought an NDS

Further reading:
J.-M Friedt & G. Goavec-Merou, *Interfaces matérielles et OS libres pour Nintendo DS : DSLinux et RTEMS*, GNU/Linux Magazine France **Hors Série 43** (August 2009) [in French]
(and inluded references)