Developing embedded devices using opensource tools: application to handheld game consoles

G. Goavec-Merou, S. Guinot, J.-M Friedt

Association Projet Aurore, Besançon, France

manuscript, slides and associated documents at `http://jmfriedt.free.fr`

August 7, 2009

The purpose of this presentation is not so much to provide more examples of "yet another platform on which to run uClinux" but rather provide general methods for identifying the right operating system for the right hardware. Understanding methods for porting an operating system to a new platform is the arguably the best solution not to be limited by the available tools but only by knowledge and hardware limitations. With these purposes in mind, we will illustrate the use of two operating systems – the version of Linux for MMU-less architectures, uClinux, and RTEMS – to game consoles. The former example focuses on the ARM-based Nintendo Dual Screen (NDS) running DSLinux and, because of hardware limitations on the available memory – RTEMS. The latter example will focus on the development of a coherent buildroot environment for compiling uClinux and the associated tools (mainly busybox) to the MIPS-based Sony Playstation Portable (PSP).

1 Nintendo Dual Screen (NDS)

The NDS provides the hardware representative of typical embedded applications: an ARM based architecture with multiple CPUs – an ARM9 main processor and an ARM7 coprocessor for managing peripherals – 4 MB RAM and most important, access to the bus of the processors. Indeed, in order to provide compatibility with previous handheld consoles (Gameboy Advance), the so-called slot2 cartridge uses the data and address busses common to both processors. Hence, we obtain the basic requirements on a widely available platform to develop embedded applications including dedicated acquisition and control hardware controlled by software running on operating systems.

The objective we set for the work on the NDS is as follows:

- we wish to develop dedicated hardware to monitor the state of the environment (data acquisition) and possibly control this environment (command)
- we wish to use efficiently all resources provided by the hardware, and especially the wifi interface for *wireless* monitoring and control of the environment
- using wifi means using a TCP/IP stack, so we will focus on operating systems running on the NDS and providing such features.

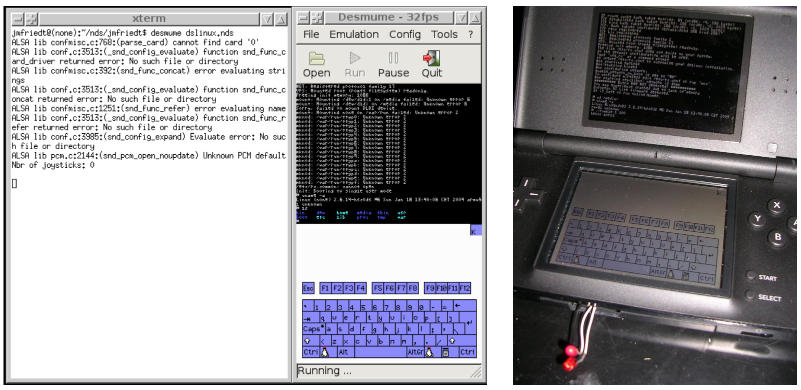
Developing software on the NDS requires the acquisition of a cartridge for transfering our programs – stored for example on a microSD memory card – to the console. Since part of the data transfer is performed through an encrypted channel, dedicated cartridges were developed for that purpose: all the examples described here were tested on a DS Lite using a M3DS Real [1] bought for about 25 euros including a rumbling pack which will be used as hardware prototyping board.

1.1 DSLinux

Our first objective is to get familiar with an NDS environment before developing dedicated hardware. GNU/Linux is arguably the most popular opensource development environment at the

[1] `www.m3adapter.com`

moment so we start using first a binary distribution of DSLinux (`http://dslinux.org`), the NDS port of uClinux [2], before installing during a second step the development environment needed when we wish to write our own application and kernel modules for accessing the hardware.



```
jmfriedt@(none):~/nds/jmfriedt$ desmume dslinux.nds
ALSA lib confmisc.c:768:(parse_card) cannot find card '0'
ALSA lib conf.c:3513:(_snd_config_evaluate) function snd_func_c
ard_driver returned error: No such file or directory
ALSA lib confmisc.c:392:(snd_func_concat) error evaluating stri
ngs
ALSA lib conf.c:3513:(_snd_config_evaluate) function snd_func_c
oncat returned error: No such file or directory
ALSA lib confmisc.c:1251:(snd_func_refer) error evaluating name
ALSA lib conf.c:3513:(_snd_config_evaluate) function snd_func_r
efer returned error: No such file or directory
ALSA lib conf.c:3985:(snd_config_expand) Evaluate error: No suc
h file or directory
ALSA lib pcm.c:2144:(snd_pcm_open_noupdate) Unknown PCM default
Nbr of joysticks: 0
```

Figure 1: Left : DSLinux running on the `desmume` NDS emulator. Right: the same result is obtained on the console, here shown with the modified rumble pack cartridge for controling LEDs (section 1.2).

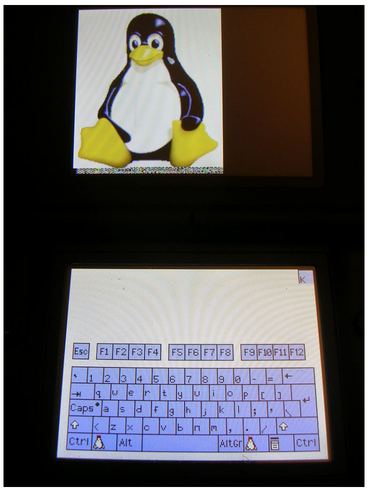
Beyond the kernel and some tools provided by busybox, DSLinux is provided with a virtual keyboard on the touchscreen. The easiest way of testing DSLinux is at first to download the binary image at `http://kineox.free.fr/DS/dslinux-dldi.tgz`: storing the file `dslinux.nds` in the `nds` directory will add a new game to the list which links to the DSLinux bootloader. Any attempt to run more than basic commands will quickly lead to errors associated with memory overflow: although 4 MB is definitely insufficient to run efficiently uClinux, this environment will provide a convenient tool to become familiar with hardware control on the NDS. As long as no hardware access is needed, the NDS emulator `desmume` (`http://www.desmume.com/`) provides a convenient environment to test programs without the need to copy the binary file (a.k.a game) to the microSD cartridge and boot the NDS, a process quickly boring during trial and error processes.

The point of using an operating system (Fig. 1), providing features such as support for filesystems, a scheduler and an abstraction layer between the application and the hardware, becomes obvious on the example displayed Fig. 2. The framebuffer accessed through `/dev/fb0` allows the use of programs classically executed on GNU/Linux with no other modification than to consider that the 16-bit display is memory-mapped and each pixel is made of 5 bits red, 5 bits green and 5 bits blue. In order to compile this example, the crosscompilation toolchain for generating ARM binaries on an Intel x86 platform must be downloaded at `http://stsp.spline.de/dslinux/toolchain`, an easier process than manually compiling `gcc` as a crosscompiler after applying the necessary NDS patches (this procedure will be demonstrated within the `buildroot` environment for the PSP in the next section 2). Once the toolchain is installed, the latest archive of the source codes for compiling DSLinux are fetched at `http://stsp.spline.de/dslinux/dslinux-snapshot.tar.gz`: getting the crosscompilation environment started is achieved by typing `make xsh` *after* having at least once configured the environment with `make menuconfig`. Within this environment, compiling a C program named `hello.c` is achieved by `$CC $CFLAGS $LDFLAGS hello.c -o hello`.

Most of the DSLinux directories are symbolic links towards the microSD card:

```
jmfriedt@ns39351:~/dslinux/romfs$ ls -l | cut -c 53-100
 bin
 boot
 dev
 etc -> media/linux/etc
 home -> media/linux/home
```

[2]uClinux is the port of the Linux kernel to environments which lack memory management units – MMU – associated with tools dedicated to low power and small memory footprint such as busybox and uClibc.



```
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);
 [...]
}
```

Figure 2: Exemple of accessing the framebuffer device `/dev/fb0` to display a bitmap image on the top screen.

```
lib -> media/linux/lib
media
opt
proc
sbin
tmp
usr -> media/linux/usr
var -> media/linux/var
```

so that adding our programs is simply a matter of adding the binary files in the right executable format (Binary Flat) in a dedicated directory of the memory card, accessible at `/media`. This familiar posix-compatible environment can already provide most functionalities commonly available on a PC running GNU/Linux, so that porting new text-mode or graphic applications using the framebuffer is painless within the restrictions of the remaining memory available. However, porting GNU/Linux applications to the NDS is hardly an exciting activity, and our purpose is to control dedicated embedded hardware thanks to the game console.

1.2 Hardware for data acquisition and control

The NDS and NDS Lite (but *not* DSi) provide a so-called Slot2 port for inserting catridges compatible with Nintendo's previous handheld game console, Gameboy Advance. This port, well documented [3], provides an access to all necessary signals of the bus common to the two ARM processors. We will focus on the 16-bit data bus (*what* information is transmitted), the 24-bit address bus (*where* in the address space is the peripheral located) and the 4-bit contol bus including read (RD#), write (WR#), Chip Select (CS#) and one hardware interrupt line. All signals are 3.3 V high. A preliminary chip select is applied since the signals appear on this bus only within the right address space starting at 0x8000000: peripherals are memory mapped [4]. In order to optimize the few available signals, the data bus is also used as the upper part of the address bus during the first cycle of all transactions: the timing defining whether the data bus holds part of the address or the data is defined by the level of the CS# signal. The data written to the peripheral are available on the data bus when *both* CS# and WR# are low (as opposed to CS# low and WR# high which indicates the data bus holds the least significant part of the address): the data must be latched on the rising edge of WR#. From a software point of view, we access the

[3] `www.ziegler.desaign.de/GBA`

[4] Alternatively, is seems the RAM extension is memory mapped to addresses starting at 0xE000000, with separate address and data busses. We have not tested this way of accessing Slot2.

16-bit memory address located at 0x8000000 and put the value `v` on the data bus with `*(unsigned short*)0x8000000=v`. This syntax can be used either in kernel space or in user space since the ARM9 of the NDS lacks an MMU (running this command on a processor with MMU will generate a Segmentation Fault since the user space program is not allowed to access address regions it had not been granted access to).

This example hence provides the necessary interface between the software and the signals generated on the busses: writing to the address 0x8000000 puts the value `v` on the data bus, and triggers the control signal sequence displayed in Fig. 3. By connecting a latch [5] to the data bus, the value of `v` is mirrored and memorized upon the rising edge of `WR#`: if LEDs are connected to the latch output, the value of `v` appears on the LEDs. This strategy is typically used to control digital outputs (stepper motor, switches, ...).

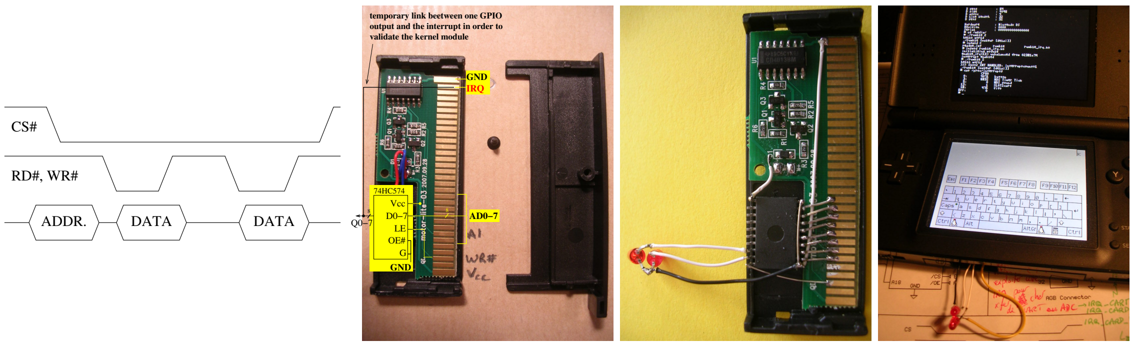

CS#
RD#, WR#
ADDR.
DATA
DATA
temporary link beetween one GPIO output and the interrupt in order to validate the kernel module
GND
IRQ
74HC574
Vcc
D0–7
LE
OE#
G
GND
Q0–7
AD0–7
A1
WR#
Vcc

Figure 3: Evolution of the signals available on the Slot2 during a transaction: the Chip Select signal activates the peripheral, the `RD#` and `WR#` control signals define the direction of the data transfer to the address defined by the address bus whose lowest significant byte is multiplexed with the data bus. A circuit using the whole address space must latch the lowest significant byte of the address bus on the decaying edge of `CS#` before using the data provided on these same pins on the rising edge of `RD#` or `WR#`.

1.3 Data acquistition: adding an ADC

Acting on its environment is only half the fun of embedded designs: more important is the capability to learn the state of the environment we are acting on, *i.e.* *acquire* informations. Since most physical values of interest are continuous rather than discrete, we shall get immediately to the most interesting aspect of analog to digital conversion, since the acquisition of a digital signal is only a specific case of the strategy described here.

In the previous example, the value was put on the databus to control a peripheral, so the processor was always the master of the bus in control of the timing and the voltages on the data bus. Reading a value is a more complex matter since for a short time the peripheral is the one to define the voltage on the data bus: allowing the peripheral to talk to the processor at the wrong time (*i.e.* when the processor is not listening) yields a conflict (two masters are talking on the data bus at the same time) and hence short circuits, potentially damaging the hardware. Hence, reading values from the peripherals requires that the peripheral strictly meets timing constraints stating that the signals connected to the data bus are always in a high impedance state (the peripheral does not define the voltage on the data bus) unless the processor is in the listen state as defined by a low value on the read RD# control signal. In the case of an analog to digital converter (ADC), the control sequence is sightly more complex (Code 1) since

[5] a latch is an integrated digital circuit used to memorize the input value either on the level or the edge of a clock signal, and keep this output whatever happens on its input as long as the clock signal does not trigger a new intput to output transfer.

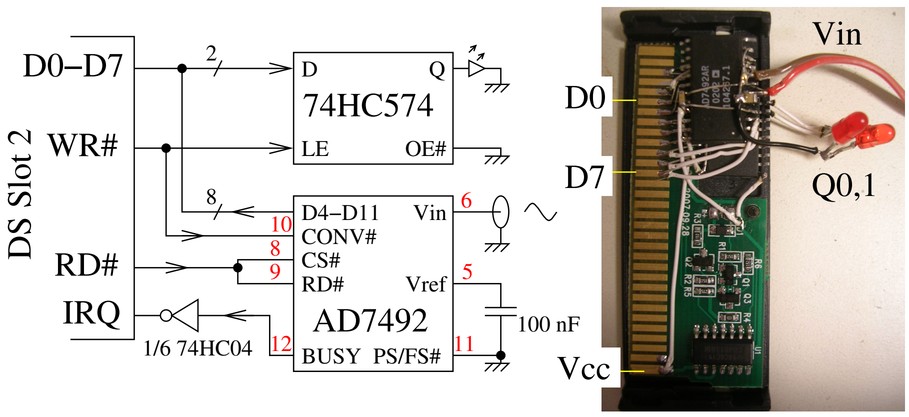


Figure 4: Data acquisition using an analog-to-digital converter (AD7492) connected to the bus. Since here the peripheral communicates to the processor, care must be taken to keep the data bus in a high impendance state most of the time and only bring it to a low impedance state – providing the result of each measurement on the data bus – when requested by the control bus. The circuit is connected through the modified Rumble Pack cartridge inserted in the Slot2. Soldering SMD components is not difficult but requires an appropriate microscope.

- the conversion must be triggered by the processor by *writing* a value on the data bus to trigger the conversion (high to low pulse on the CONVST# pin of the ADC)
- wait for the conversion to complete: either using a fixed delay (empty loop) or reading the end of conversion signal from the ADC
- request the result of the conversion by reading a value in the address space of the peripheral. At this time, the RD# control signal goes low and the ADC is allowed during this time to define the voltage on the data bus with the pattern associated with the binary value of the measurement
- once the read cycle is completed, the RD# goes high and hence the ADC releases the data bus by putting its output pins back to a high impedance state.

The result of such a procedure is displayed in Fig. 5, with the sequential analog to digital conversion of signals at a rate up to 300 kHz and a display of the results of the measurement on the top display. In the case of DSLinux with little load on the system, running such a conversion either in user space (allowed thanks to the lack of MMU which allows the user to write in memory locations that would otherwise be forbidden and yield to a Segmentation Fault) or kernel space (from a kernel module) provides the same performances. The case of detecting the end of conversion from an interrupt will not be developed here, but let us mention that the interrupt service routine (ISR) management from kernel space strongly *reduces* the sampling rate due to the high software complexity between the start of conversion and interrupt management when compared to the empty loop shown in Code 1.

Being able to acquire data from the modified cartridge, we wish to go on with wireless data transfer using the wifi interface ... but the available 4 MB are simply insufficient to run even basic network configuration commands. Two strategies can thus be adopted:

1. add more resources to compensate for the lack of efficiency of the software. This is the usual personnal computing approach, allowing most users to require multi-gigahertz CPU cores to move windows on a graphical interface and type text in a word processing application. On the NDS, this strategy means using a memory extension pack, which unfortunately uses slot2 and means all the hardware developments used so far are no longer usable with this approach,

```
#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);
}
```

Table 1: Sample program for reading a value from a peripheral connected to the bus of Slot 2. In this case of the analog to digital conversion, the conversion is triggered by a write and after a pre-defined delay tuned to reach the maximum sampling rate, the result is read.

2. change operating system or developing environment to adapt to the available resources, attempting to reach the goal we set of wireless data transfer and control without reducing functionalities but by selecting a more efficient environment.

The second strategy is the one adopted here and leads us to leave DSLinux for a more compact and memory efficient environment: the Real Time Executive for Multiprocessor Systems (RTEMS).

1.4 RTEMS

As an alternative to the resource hungry DSLinux, we will explore the use of a real time *executive environment* – RTEMS (`http://www.rtems.com/`). An executive environment provides the developer with the feeling of working on an operating system with abstraction layers between software and hardware, a scheduler for executing multiple tasks simultaneously, and libraries such as filesystem access or network stacks (most significantly TCP/IP as we shall see later). However in order to run with minimal memory footprint, RTEMS generates a single monolithic application and will not dynamically link libraries or programs. An interactive shell might be run on RTEMS but most embedded applications will not need such a tool. Nevertheless, tools dedicated to embedded application developments are provided such as CPU consumption, stack size and availability and, upon request, network or filesystem associated commands.

A POSIX compatibility layer means that the developer under GNU/Linux will quickly feel in a familiar environment when working under RTEMS. However, beyond the usual header followed by functions and a main program, RTEMS requires a precise definition of the needed resources as a set of `#define` macros at the end of the program, with constraints developed later in this document.

1.4.1 Programming examples: basic structure, framebuffer and shell

A Board Support Package (BSP) for the NDS has been developed by M. Bucchianeri, B. Ratier, R. Voltz and C. Gestes, as described at `http://www.rtems.com/ftp/pub/rtems/current_contrib/`

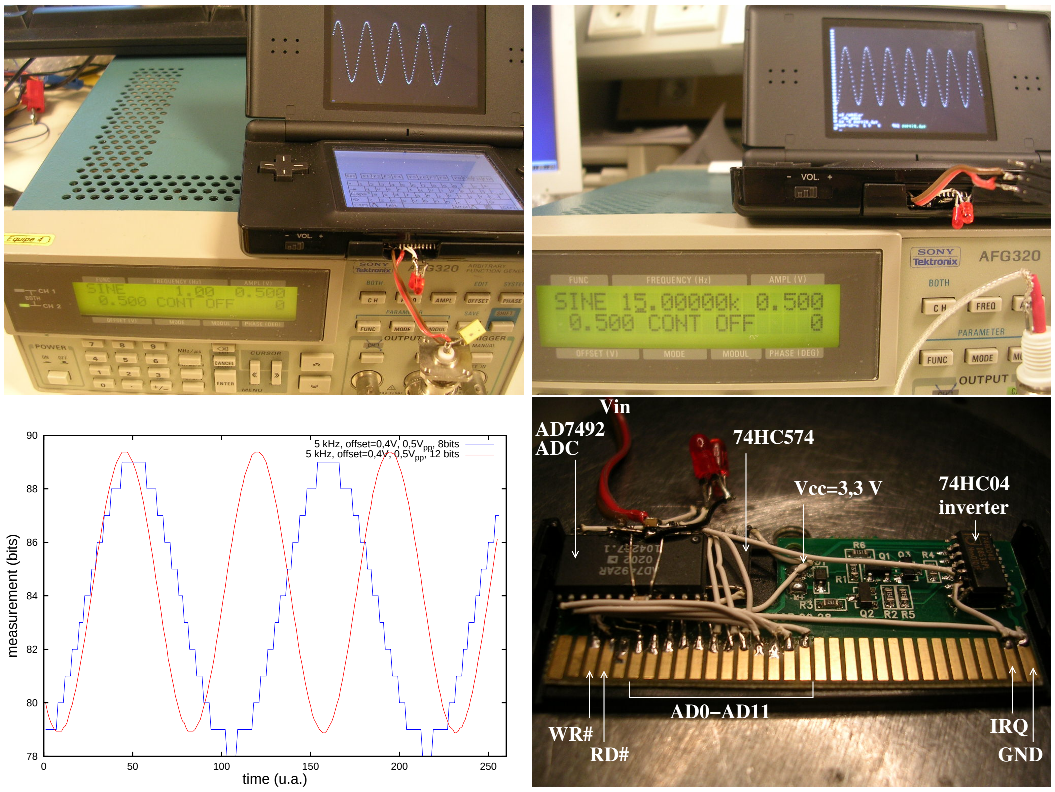


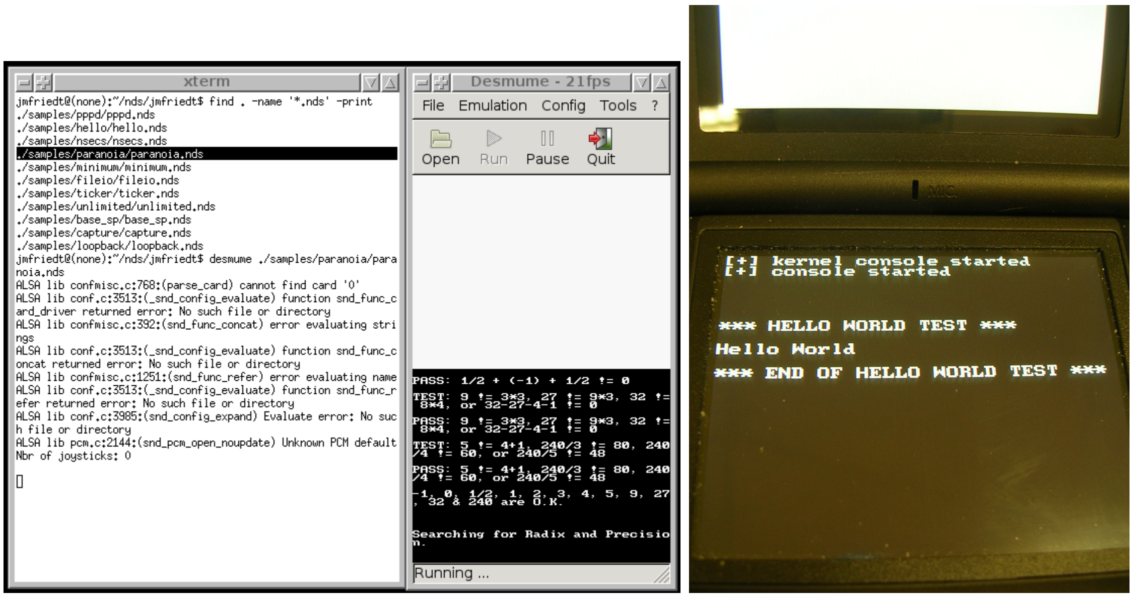
Figure 5: Top: a frequency synthesizer generates a sine wave signal with 1 V amplitude, offset 0.5 V and frequency 1 Hz (left, representative of a slowly varying signal such as a temperature) and 15 kHz (right, in order to estimate the maximum sampling rate). These data are obtained using the circuit shown in Fig. 4. The software controling this peripheral is presented in Tab. 1 while the display is performed using the framebuffer functionality of DSLinux as illustrated previously. Bottom: comparison of 8 and 12-bit wide conversions (reading a `char` or a `short` on the data bus) and the actual implementation of the data acquisition circuit in a volume compatible with the insertion of the cartridge in Slot2.

`nds-bsp/manual.html`. In order to compile RTEMS, we must install a new crosscompilation toolchain including the appropriate patches, as described at `http://www.rtems.com/onlinedocs//doc-current/share/rtems/pdf/started.pdf` [6]. As a quick summry of this documentation, the latest versions of the usual toolset `gcc`, `binutils` and `newlib` must be downloaded at `ftp://ftp.rtems.com/pub/rtems/SOURCES/4.9/` and configured for the ARM target with the option `--target=arm-rtems4.9`. Be aware that compiling the toolchain within the source archive induces many problems: a solution is provided at `http://www.mail-archive.com/gcc-bugs@gcc.gnu.org/msg191702.html`, but it is safer to avoid problems by compiling in a different target directory than the one including the source codes. Once a functional toolchain is compiled and installed in a directory included in the PATH (about 100 MB of disk space are required), RTEMS itself must be compiled.

Once again we take care to compile in another directory than the one containing the source code: starting from a clean directory `rtems-4.9.1`, the BSP is configured using `../rtems-4.9.1/configure -target=arm-rtems4.9 -enable-rtemsbsp=nds` followed by the compilation with `make` (Fig. 6).

In order to compile our own programs or some of the usual examples provided with RTEMS

[6]more precisely 4.1.4.2 page 22



```
jmfriedt@(none):~/nds/jmfriedt$ find . -name '*.nds' -print
./samples/pppd/pppd.nds
./samples/hello/hello.nds
./samples/nsecs/nsecs.nds
./samples/paranoia/paranoia.nds
./samples/minimum/minimum.nds
./samples/fileio/fileio.nds
./samples/ticker/ticker.nds
./samples/unlimited/unlimited.nds
./samples/base_sp/base_sp.nds
./samples/capture/capture.nds
./samples/loopback/loopback.nds
jmfriedt@(none):~/nds/jmfriedt$ desmume ./samples/paranoia/para
noia.nds
ALSA lib confmisc.c:768:(parse_card) cannot find card '0'
ALSA lib conf.c:3513:(_snd_config_evaluate) function snd_func_c
ard_driver returned error: No such file or directory
ALSA lib confmisc.c:392:(snd_func_concat) error evaluating stri
ngs
ALSA lib conf.c:3513:(_snd_config_evaluate) function snd_func_c
oncat returned error: No such file or directory
ALSA lib confmisc.c:1251:(snd_func_refer) error evaluating name
ALSA lib conf.c:3513:(_snd_config_evaluate) function snd_func_r
efer returned error: No such file or directory
ALSA lib conf.c:3985:(snd_config_expand) Evaluate error: No suc
h file or directory
ALSA lib pcm.c:2144:(snd_pcm_open_noupdate) Unknown PCM default
Nbr of joysticks: 0
```

```
PASS: 1/2 + (-1) + 1/2 != 0

TEST: 9 != 3*3, 27 != 9*3, 32 !=
 8*4, or 32-27-4-1 != 0

PASS: 9 != 3*3, 27 != 9*3, 32 !=
 8*4, or 32-27-4-1 != 0

TEST: 5 != 4+1, 240/3 != 80, 240
/4 != 60, or 240/5 != 48

PASS: 5 != 4+1, 240/3 != 80, 240
/4 != 60, or 240/5 != 48

-1, 0, 1/2, 1, 2, 3, 4, 5, 9, 27
, 32 & 240 are O.K.


Searching for Radix and Precisio
n.
```

```
[+] kernel console started
[+] console started

*** HELLO WORLD TEST ***
Hello World
*** END OF HELLO WORLD TEST ***
```

Figure 6: Left: RTEMS is executed in the `desmume` emulator. Right: one of the simplest examples, executed from the console.

[7], we defined some variables

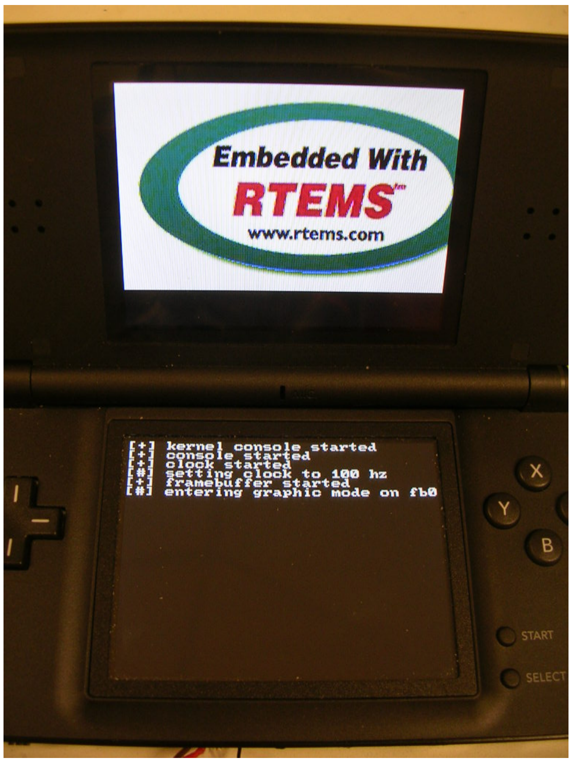
```
export PATH=/home/jmfriedt/rtems/bin/:$PATH
export RTEMS_ROOT=/opt/rtems-4.9
export RTEMS_MAKEFILE_PATH=/opt/rtems-4.9/arm-rtems4.9/nds
```

(obviously adapted to the reader's environment) and compile with `make` (Fig. 7).

Although all basic RTEMS examples are functional and provide plenty of inspiring source codes, a description of the general program structure might ease the first contact with this new environment:

- RTEMS provides an environment that makes the developer feel like working on an operating system, even though the result of the compilation is a single monolithic application which lacks such aspects as dynamic loading a program or a library. The execution of the program is scheduled with the call to as many tasks as defined by the developer depending on events.
- within the context of embedded device development with low memory footprint, a resource hungry interactive shell is a luxury useless to most applications and not activated in the default configuration.
- although written in C, an RTEMS application requires a different program structure than the usual C program describing an application for GNU/Linux. A POSIX compatibility layer reduces the adaptation time for the new RTEMS developer used to developing under unix. As usual, some `#include` in the header declare which configuration files to load, but most important are the multiple `#define` which define which options of the BSP to activate. Since each additional functionality brings its own need for additional resources, reducing to the minimum the requirements is a good practice for embeded designs. However, missing one necessary option will make the application crash in ways that are not always obvious to debug: as an example described at `http://www.rtems.com/wiki/index.php/DebuggingHints`, calling the `sleep` without activating the timer function will make the program wait indefinitely since the timer never tells the requested time has elapsed. The

[7]available at `http://rtems.org/wiki/index.php/RTEMS_CVS_Repository`



```
[+] kernel console started
[+] console started
[+] clock started
[#] setting clock to 100 hz
[+] framebuffer started
[#] entering graphic mode on fb0
```

Figure 7: Example of using the framebuffer interface of the NDS following principles used in uClinux, but this time with RTEMS. This example is based on a sample program provided by M. Bucchianeri. Since we are unable to access the microSD storage support, the image is stored in the program.

order in which these `#define` are declared is important since variables declared in the first configurations might induce different behaviours for the next definitions.

We wish to go beyond the simple software programming to focus, as was done with DSLinux, on controling dedicated hardware peripherals. In order to make some LEDs connected to a latch blink (Code 3), one must give control of the Slot2 bus (to which the Rumble Pack is connected) to the ARM9 CPU rather than to the ARM7 coprocessor. This result is achieved using the command `sysSetCartOwner(BUS_OWNER_ARM9);`: without this function, the CPU running RTEMS does not have access to the cartridge bus and the LEDs will not change state. This command is defined in `rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/source/arm9/rumble.c` and the macro was interpreted manually to ease the compilation step: we find this command in the line controling the content of memory location `0x04000204` in our sample program. Notice that this step was not necessary with DSLinux since the ARM9 CPU had already been granted access to the bus.

1.5 Interactive shell and "typing" commands

As opposed to an operating system designed to interact with a user, RTEMS only provides a shell as one of many options for developing embedded systems.

The RTEMS shell is started using the command
`rtems_shell_init("SHLL",RTEMS_MINIMUM_STACK_SIZE * 4,100,"/dev/console",0,1);` either as the main task, or as one of the threads running in parallel with the other tasks performed by the program. The most common means of comunicating with a shell is to type commands: the lack of keyboard has been compensated for by the authors of the NDS BSP by porting the PALib `graffiti` application. In order to activate this character recognition software so that

```
#include <bsp.h>
#include <rtems/fb.h>
#include <rtems/console.h>
#include <rtems/clockdrv.h>
#include "fb.h"

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <rtems/mw_fb.h>

static struct fb_screeninfo fb_info;

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;    // 5R, 5G, 5B
  *loc |= 1 << 15; // transparence ?
}

void draw_ppm()
{int x,y,bpp;char r,g,b;
 int l=0;
 for (y=0;y<161;y++)
   for (x=0;x<296;x++) {
     r=image[l++];g=image[l++];b=image[l++];
     bpp=(((int)(r&0xf8))>>3)+(((int)(g&0xf8))<<2)+(((int)(b&0xf8))<<8);
     if (x<256) draw_pixel(x, y, bpp);
   }
}

rtems_task Init(
  rtems_task_argument ignored
)
{
```

```
  struct fb_exec_function exec;
  int fd = open("/dev/fb0", O_RDWR);

  if (fd < 0)
    { printk("failed\n");
      exit(0);
    }

  exec.func_no = FB_FUNC_ENTER_GRAPHICS;
  ioctl(fd, FB_EXEC_FUNCTION, (void*)&exec);
  ioctl(fd, FB_SCREENINFO, (void*)&fb_info);

  draw_ppm();

  while (1) ;

  exec.func_no = FB_FUNC_EXIT_GRAPHICS;
  ioctl(fd, FB_EXEC_FUNCTION, (void*)&exec);
  close(fd);printk("done.\n");exit(0);
}

/* configuration information */

#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 }
};

#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS      10
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 1
#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

Table 2: Program for displaying a PPM image on the framebuffer, based on a sample program provided by M. Bucchianeri.

the user can enter characters by writing on the touchscreen, the left key (L) must be held down while writing. The set of characters and the order in which to draw the lines is described at `http://www.palib.info/wiki/doku.php?id=day3#keyboard`. An alternative to typing commands this way will be described later (section 1.5.1) once the TCP/IP stack is running and a server associated with a shell is linked to a socket connexion.

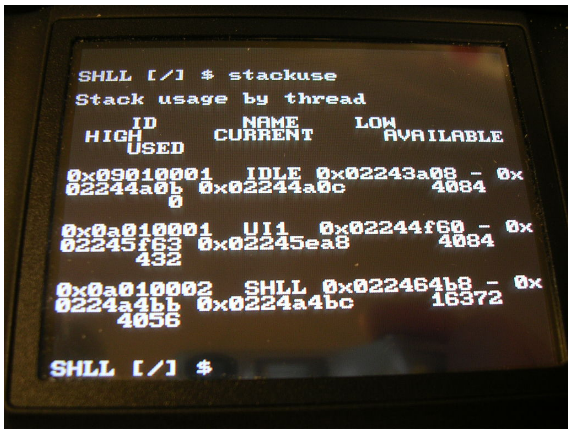
Beyond the classical basic unix commands available from the shell, some functions dedicated to embedded device development have been added, most significantly informations concerning the time spent by the CPU on each tasks (`cpuuse`) and the address space associated with the stack of each process (`stackuse`) for diagnostics purposes (Fig. 8). The content of the memory is dumped using `mdump`. Additional functions associated with filesystem access and network access are available when the appropriate `#define` flags have been activated.

In addition to the default shell commands, the user can define new custom commands. In this example, we wish to convert an autonomous program accessing the framebuffer to a shell command called `newt` which draws Newton's fractal (the attraction basins of the three roots of the complex polynom $z^3 - 1$, $z \in \mathbb{C}$). This application also demonstrates the software emulation of floating point calculations, access to the framebuffer and parameter handling by the application.

The prototype of the function thus added to the shell is similar to the usual `main(int,`

```
#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

/* This seetings overwrite the ones defined in confdefs.h */
#define CONFIGURE_MAXIMUM_POSIX_MUTEXES                    32
#define CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES        32
#define CONFIGURE_MAXIMUM_POSIX_KEYS                       32
#define CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS             10
#define CONFIGURE_MAXIMUM_POSIX_THREADS                    128
#define CONFIGURE_MAXIMUM_POSIX_TIMERS                     10
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS           200

#define STACK_CHECKER_ON
#define CONFIGURE_INIT

#include <rtems/confdefs.h>

/* end of file */
```

Table 3: Sample program for making LEDs connected at the output of a 74HC574 latch located instead of the motor of the Rumble Pack blink. Notice the 21st line granting access of the slot2 bus to the ARM9 CPU.



```
SHLL [/] $ stackuse
Stack usage by thread
     ID          NAME      LOW
 HIGH          CURRENT        AVAILABLE
     USED

0x09010001   IDLE 0x02243a08 - 0x
02244a0b 0x02244a0c          4084
       0

0x0a010001   UI1  0x02244f60 - 0x
02245f63 0x02245ea8          4084
     432

0x0a010002   SHLL 0x022464b8 - 0x
0224a4bb 0x0224a4bc         16372
    4056

SHLL [/] $
```

Figure 8: One of the original and most useful RTEMS commands dedicated to the development of embedded systems: the size of the stack associated with each thread.

`char**)`: the `newtn` commands executes the `my_main` function and argument passing is performed in the same way than one would with the classical C `main`:

```
rtems_shell_cmd_t Shell_USERCMD_Command = {
  "newtn",                    /* name */
  "newt [mag.]",              /* usage */
  "user",                     /* topic */
  my_main,                    /* command */
  NULL,                       /* alias */
  NULL                        /* next */
};
```

Since `graffiti` is rather difficult to use and typing a 5-letter command is virtually impossible, we define a shorter alias (`n`) to this command:

```
rtems_shell_alias_t Shell_USERECHO_Alias={
"newtn", /* command*/
"n"     /* alias */
};
```

This command and its alias are added to the shell [8] with

```
#define CONFIGURE_SHELL_USER_COMMANDS &Shell_USERCMD_Command
#define CONFIGURE_SHELL_USER_ALIASES  &Shell_USERECHO_Alias
#define CONFIGURE_SHELL_COMMANDS_INIT
#define CONFIGURE_SHELL_COMMANDS_ALL
#include <rtems/shellconfig.h>  // must be AFTER #define
```

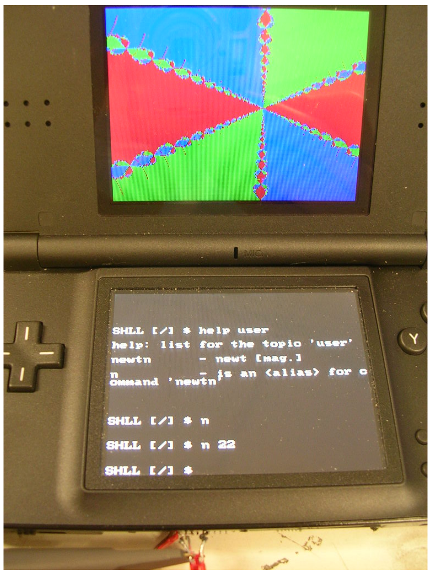
In this example, the command `newtn` (or its alias `n`) draws Newton's fractal with a magnification factor given as argument in the command line (Fig. 9).

1.5.1 Application example: wireless data acquisition system

As a final note to the development on the NDS, we shall conclude with the realization of the original goal of this work, the wireless transfer of data acquired from the NDS. The TCP/IP stack originally provided with RTEMS proved to be incompatible with the NDS wifi library and some data ordering error had to be corrected: the patch we provide is available at `http://jmfriedt.free.fr/rtems_nds_wireless.patch`.

Once a functional TCP/IP stack running on top of the wifi interface is available, running a shell or a dedicated application on a socket requires

[8] `http://www.rtems.com/onlinedocs//doc-current/share/rtems/html/shell/shell00008.html`



```
SHLL [/] $ help user
help: list for the topic 'user'
newtn         - newt [mag.]
n             - is an <alias> for c
ommand 'newtn'


SHLL [/] $ n

SHLL [/] $ n 22

SHLL [/] $
```

Figure 9: A new RTEMS command (`newtn`): calculation and drawing of Newton's fractal, used as an opportunity to illustrate the software emulation of floating point calculation and accessing the framebuffer display.

1. the configuration of the wireless interface of the NDS using a wifi-compatible commercial game. Indeed, the NDS uses a uniform storage format of up to three access points including their MAC address for automatic network configuration. RTEMS and the NDS wifi library follow this scheme and hence require a preliminary configuration. One possible solution for point to point communication avoiding the need for an accesspoint is to configure an Asus eeePC 701 laptop as an access point as described at `http://jmfriedt.free.fr/fred/fred.html`

2. once a commercial game has been used to configure the MAC address of the access point we wish to connect to (to be stored in the first configuration slot), a RTEMS program follows the usual network configuration procedure, namely the interface configuration for defining the IP address, the routing table, the creation of a socket to listen to incoming connections and finally a link between the incoming data and a server. This server might be either a shell for an interactive session with the user (much more convenient to use than the character recognition application `grafiti`) or a dedicated server as will be demonstrated here,

3. all `printf()` calls within this server are redirected to this socket rather than to the terminal associated with the console displayed on the screen, so that by printing in ASCII format the measurement results, a `telnet` session from a PC to the NDS is enough to gather measurements without the need for the development of a dedicated client.

The structure of a RTEMS program using the TCP/IP stack follows the same scheme as familiar to the network configuration on a unix system: first the interface is configured and given an IP address, here by defining the following structure:

```
/* Default network interface */
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 hardware address */
};
```

which is equivalent to the `ifconfig` command under unix. *RTEMS_BSP_NETWORK_DRIVER_NAME* provides the name of the interface and *RTEMS_BSP_NETWORK_DRIVER_ATTACH* the initialization method.

The second structure

```
/* 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) */
};
```

is equivalent to the familiar *route* command associated with the DNS configuration in *resolv.conf*, providing network configuration informations.

```
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 */
);
```

The first parameter of the previous command is a handler to the function called upon network connexion (*i.e.* the equivalent of the name of the server called during a TCP/IP connexion): in this example, we wish to call a RTEMS shell:

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

Using the TCP/IP stack requires the inclusion of the header file `rtems/telnetd.h` in the source code as well as the link with the library `LD_LIBS += -ltelnetd` to be included in the Makefile of the appliaction.

These steps are adapted in the program 10 in which the call to a RTEMS shell is replaced by a server providing results of analog to digital conversions, hence converting the NDS into a wireless controled data acquisition system (Fig. 11).

2 Sony PlayStation Portable (PSP)

The newer Sony handheld console provides less opportunities for hardware developments: although an asynchronous serial port (RS232) is available, most peripherals such as the USB port are not (yet) supported, and no parallel bus is available. Hence, dedicated hardware needs to be developed around an external microcontroller in charge of low level signal generation or acquisition with higher level communication through the serial port. Nevertheless, providing a coherent development environment including an opensource operating system is a prerequisite if we hope to ever see these peripherals supported without accessing the proprietary Sony operating system whose functions are for example used by the homebrew community. The MIPS-based architecture includes 32 MB RAM suitable to run uClinux and embedded applications, hence our objective of providing a coherent BSP for the PSP.

As opposed to development on the NDS which required acquiring a dedicated cartridge for runnin games stored on a microSD card, the PSP is designed with a MemoryStick (Sony's proprietary non-volatile mass storage format): after updating the PSP's firmware, custom software including an uClinux bootloader can be run from this medium.

```
/*  ADC to network */

#include <stdlib.h>
#include <stdio.h>
#include <bsp.h>
#include <rtems/telnetd.h>
#include <nds/memory.h>
#include <rtems/rtems_bsdnet.h>

// [...] NETWORK CONFIGURATION

#define TAILLE 1024
/* callback for telnet */
void telnetADC( char *pty_name, void *cmd_arg) {
        char *c;
        int f;

        printk( "Connected to %s with argument %p \n",
                pty_name, cmd_arg );

        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");
        }
}

/* Init task */
rtems_task Init(rtems_task_argument argument){
        fprintf(stderr, "\n\n*** Telnetd Server Test ***\n\r" );
        fprintf(stderr, "======== Initializing Network ========\n");
        rtems_bsdnet_initialize_network ();
        fprintf(stderr, "======== Start Telnetd ========\n");
        (*(volatile uint16_t*)0x04000204) = ((*(volatile uint16_t*)0x04000204) & →
            ↪~ARM7_OWNS_ROM);
        rtems_telnetd_initialize(
          telnetADC,        /* callback function */
          NULL,             /* no context necessary for echoShell */
          false,            /* false == listen on sockets */
          RTEMS_MINIMUM_STACK_SIZE * 20,  /* shell needs a large stack */
          1,                /* priority .. we feel important today */
          false             /* telnetd does NOT ask for password */
        );
        while(1);
}

#include "../rtems_common.h"
```

Figure 10: Sample code for transfering through a wireless network the results of analog to digital conversions, using the NDS as a remote controled acquisition system.

2.1 Buildroot for MIPS architectures

The MIPS R4000 compatible processor is supported by `gcc` with the `-mips3` option. Beyond the compiler, a consistent toolchain for compiling the kernel and the applications is needed to provide a development environment. Amongst the Board Support Package (BSP) tools, `buildroot` is arguably the most widely used, and some additional software needs to be adapted to provide an environment for developing for the PSP.

As opposed to the commonly used executable binary format ELF most familiar to developers on personnal computer whose CPU include a MMU, uClinux uses a simpler format called `binary`

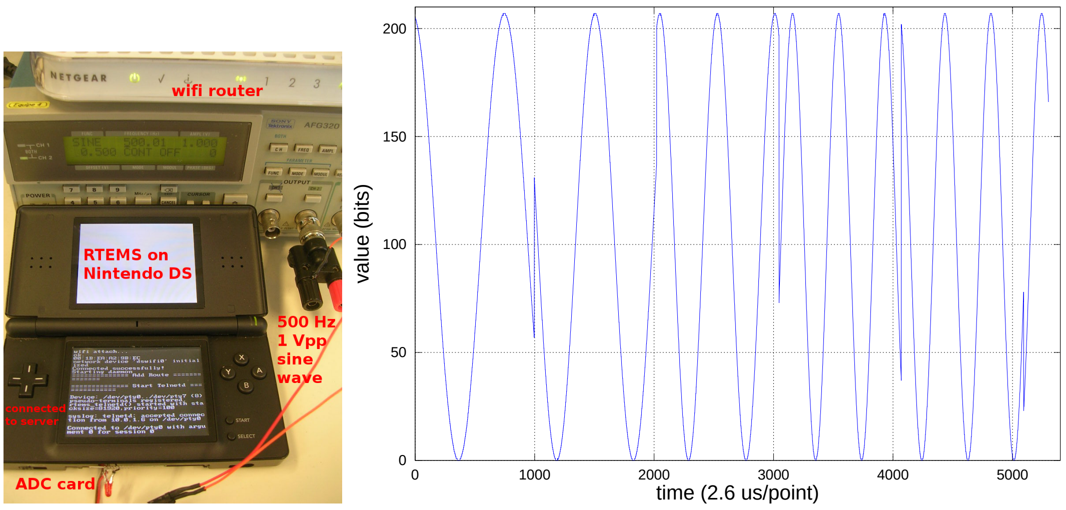


Figure 11: Left: setup for acquiring analog signals using RTEMS, and data transfer as 1024 word packets using a Wifi link through a router. Right: signal acquired and transfered through a wireless link, at first a sine wave at 500 Hz whose frequency has been increased to 1 kHz. These signals provide an estimate of the sampling rate of about 300 kHz. This simple demonstration does not claim continuity of the acquired data set: the discontinuities every 1024 points is due to the time associated with data transfer since a double-buffer scheme has not been used here.

`flat` (BFLT). This format, based on the older `a.out` format, provides a subset of the functionalities provided by ELF: smaller and easier to load, executable files in this format are better suited to embedded applications with fewer resources.

The MIPS cross-compilation toolchain is known to work well since many routers are developed around processors based on this architecture. It must however be modified for our purpose in order to generate BFLT outputs instead of the usual ELF. In order to reach this purpose, a dedicated linker must be provided: `elf2flt`, originally unavailable for this arhitecture [1].

The `elf2flt` package is composed of the following programs:

- the shell script `ld-elf2flt`
- the program `elf2flt` able to convert ELF binaries to th Binary Flat (BFLT) format
- the program `flthdr` for editing the header of BFLT files.
- `ld-eld2flt` replaces the usual `ld` linker when the `-elf2flt` compiler option is used

The `ld-elf2flt` script replaces the original `ld` linker. `ld` is now called `ld.real`. If the `ld` script is called with the `-elf2flt` option, then `ld.real` followed by `ld-elf2flt` are successively called to generate an ELF formatted and then BFLT binaries. Otherwise, `ld.real` alone is called. Hence, the toolchain behaviour is only modified when the `-elf2flt` option is provided to the linker. As an example of the compilation commands:

```
# mipsel-psp-gcc -Wl,-elf2flt -o test test.c
# ls
test.gdb test
```

Notice that `elf2flt` generates two files. The binary file with the `.gdb` extension includes debugging informations and can be used with `gdbserver`. The FLAT format only includes the few mandatory sections for executing the content of the binary file: `.text`, `.data` and `.bss`. It does not include the debugging information sections classically found in ELF formatted files.

As a summary of this contribution, the main difficulty in building the MIPS-PSP toolchain is to include the `elf2flt` after `binutils`. Good documentations describing the BFLT formats are the uClinux kernel source code of the executable file loader available in `fs/binfmt_flat.c`, and a less exaustive set of informations is available at `http://www.beyondlogic.org/uClinux/bflt.htm`.

2.2 uClinux as a PSP game

uClinux must be loaded in the PSP's volatile memory in order to run. As with all operating systems, a bootloader must be used to load the operating system in memory and initialize the hardware accordingly: here the bootloader will appear as a game, and as such has access to all of Sony's operating system functionalities to access the hardware, as would a real mode bootloader running on a PC and accessing the BIOS and associated software interrupts. Since most of the PSP hardware is proprietary and most of Sony's functions have be understood but not reverse engineered, the bootloader is our last chance to initialize the needed hardware before filling the RAM with uClinux. We will see later that another project (the IPL SDK) aims at reverse engineering some of these functions for understanding hardware access and providing native C systems calls for operating systems such as uClinux.

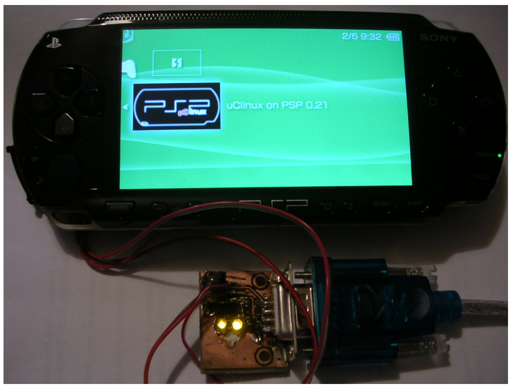


Figure 12: The most convenient communication interface with the PSP: the asynchronous serial port next to the audio connector is compatible with RS232. uClinux will be loaded as a game over Sony's operating system, so the native functions for accessing hardware peripherals are only available to the bootloader but no longer once uCinux is loaded.

The PSP's firmware must be updated to allow the execution of programs from the MemoryStick slot: the procedure varies upon the original firmware version. While this procedure seemed originally risky since the flash memory of the game console could be definitely corrupted, it seems that the latest upgrade procedures are more reliable and less risky when upgrade softwares are taken from reliable web sites. The author is using a firware v.1.50 for a PSP Light bought in December 2006.

Once the PSP software is updated, our own native PSP applications are compiled using the right toolchain [9] and the toolset called the PSP SDK [10]. These tools are needed if the reader wishes to compile, develop or modify the uClinux bootloader: reading the source code of J. Mo's working version [2] – which differs from the original work of C. Mulhearn [3] in its ability to read compressed (`gzip`ed, hence the need for the `zlib` library [11]) images – is a good starting point to learn how to read a file from the MemoryStick, uncompress it if needed (case of a compressed kernel image), initialize the needed peripherals of the PSP and load the new operating system in memory before calling it.

[9] `http://ps2dev.org/psp/Tools/Toolchain/psptoolchain-20070626.tar.bz2`
[10] `http://ps2dev.org/psp/Projects/PSPSDK`
[11] described at `http://www.psp-programming.com/tutorials/c/lesson04.htm`.

The program in charge of loading uClinux image file, uncompressing it and copying the resulting data in RAM in the address space starting at 0x88000000 [4], activate peripherals (serial port, CPU cache) and finally jump to the starting address of the the kernel is presented here:

```
#define KERNEL_ENTRY        0x88000000
#define KERNEL_PARAM_OFFSET 0x00000008
#define KERNEL_MAX_SIZE     (size_t)( 4 * 1024 * 1024 )   /* 4M */

#define printf              pspDebugScreenPrintf

BOOL loadKernel(void ** buf_, int * size_)
{
  gzFile zf;
  void * buf;
  int size;

  zf = gzopen( s_paramKernel, "r" );
  buf = (void *)malloc( KERNEL_MAX_SIZE );
  size = gzread( zf, buf, KERNEL_MAX_SIZE );
  gzclose( zf );
  *buf_ = buf;
  *size_ = size;
}
/*------------------------------------------------------------------------*/
void transferControl(void * buf_, int size_)
{
    KernelEntryFunc kernelEntry = (KernelEntryFunc)( KERNEL_ENTRY );

    /* prepare kernel image */
    memCopy( (void *)( KERNEL_ENTRY ), buf_, size_ );

    uart3_setbaud( s_paramBaud );
    uart3_puts( "Booting Linux kernel...\n" );

    kernelEntry( 0, 0, kernelParam );
}
```

A copy of this bootloader, compiled as a PSP game using `make kxploit` for generating the usual two directories `pspboot` et `pspboot%` ready for use on the MemoryStick, is available on the author's website at `http://jmfriedt.free.fr/`.

```
$ make kxploit
psp-gcc -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall -D_PSP_FW_VERSION=150   -c -o main.o main.c
psp-gcc -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall   -
psp-gcc -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall -D_PSP_FW_VERSION=150  -L. -L/usr/local/pspdev/psp/sdk/lil
...
```

We can now load an operating system in the volatile memory of the PSP: we wish to be able to compile this software in order to adapt it to our needs. However, an operating system, and the associated tools for exploiting the facilities provided by this environment, is a rather large piece of software, for which some dedicated compilation tools have been developed: `buildroot` is the one we have chosen to use here.

2.3 Description and usage of the `buildroot` environment

`buildroot` is a set of Makefiles for generating a toolchain (set of tools for compiling programs for a given target architecture), a Linux kernel and a *rootfs* (image of the files needed to configure and run the operating system). `buildroot` automates the download of source codes and the needed patches, configures and installs the programs in the *rootfs*. Various *rootfs* image formats are supported: squashfs, cramfs, ext3, cpio archive, initramfs, etc... In the case of the PSP, the file loaded in memory is obtained by compressing with `gzip` the raw binary image `buildroot-psp/project_build_mipsel/linuxonpsp/linux-2.6.22/arch/mips/boot/vmlinux.bin`).

This image includes the kernels and the embedded *rootfs* as an initramfs. The option `BR2_TARGET_ROOTFS_INITRAMFS=y` must hence be activated during the `buildroot` configuration.

Considering the `buildroot` directory structure might seem at first sight complex, we will present here some of the most common directories and describe their content:

- the `toolchain` directory includes all necessary Makefiles to compile the toolchain. For example, `toolchain/gcc` holds the files needed for generating a functional `gcc` for the requested target, and so for all the supported versions:

  ```
  $ tree -L 1 toolchain/gcc/
    toolchain/gcc/
    |-- 3.3.5
    |-- 3.3.6
  ...
    |-- 4.2.1
    |-- Config.in
  ...
    |-- gcc-uclibc-3.x.mk
    |-- gcc-uclibc-4.x.mk
  ```

  The files `gcc-uclibc-3.x.mk` and `gcc-uclibc-4.x.mk` are elements of the Makefile needed to compile gcc depending on the selected configuration.

- during the toolchain compilation step, the various components are located in the directory `toolchain_build_xxxx`: for the PSP, `xxxx` is `mipsel`.

- `package` includes the Makefiles for the applications. As an example, `wget` (non busybox) is located in:

  ```
  $~/buildroot$ tree package/wget/
    package/wget/
    |-- Config.in
    `-- wget.mk
  ```

  `wget.mk` is the fragment of the Makefile for compiling `wget`. A quick glance at this file shows some interesting targets:

  ```
  $~buildroot/package/wget$ cat wget.mk
  ...
  wget: uclibc $(TARGET_DIR)/$(WGET_TARGET_BINARY)

  wget-clean:
          rm -f $(TARGET_DIR)/$(WGET_TARGET_BINARY)
          -$(MAKE) -C $(WGET_DIR) clean

  wget-dirclean:
          rm -rf $(WGET_DIR)
  ...
  ```

  From the buildroot top directory, the command `make wget` will add the command `wget` to the *rootfs*. `make wget-clean` cleans the directory in which the compilation of `wget` takes place (*i.e.* `build_mipsel/wget`). `make wget-dirclean` is the method to remove even this directory. In general, the methods `$(app)`, `$(app)-clean` and `$(app)-dirclean` are available for all applications.

  The same analysis of the `busybox` directory shows the same directory structure including all the versions and associated patches available.

- most applications (also called *packages*) are compiled in `build_mipsel` (variable `BUILD_DIR` in the various Makefiles of the packages). This directory is dynamically created during the compilation step. Each package is uncompressed there, configured and compiled:

  ```
  $~/buildroot$ ls build_mipsel/
  fakeroot-1.8.10  makedevs  psposk2  staging_dir
  ```

  Notice the presence of the `build_mipsel/staging_dir` directory. It is used in the various Makefiles under the `STAGING_DIR` variable and is basically the root directory of the cross-compilation environment. The `CFLAGS` and `LDFLAGS` variabes are defined in `buildroot` to direct towards `$(STAGING_DIR)/usr/include` and `$(STAGING_DIR)/usr/lib`. When a new library is compiled, it is not immediately moved to the *rootfs* but is first added in the `STAGING_DIR` directory. This is an important step, especially if another application compiled later needs this library. The `configure` script of this application will need to detect the availablity of the library and the associated header files in order to generate the right Makefile with the appropriate functionalities. In the case a critical dependency is not met, the compilation will simply not occur. A good habit when packaging an application is to always use the `STAGING_DIR` directory when installing this application (`make install`). The needed files will later be copied from the `STAGING_DIR` to the *rootfs*. This step acts as a filter and removes all the documentation files or the programs not needed on the target system. Access permission on some files are also updated at this stage.

- the main applications such as `busybox` or the Linux kernel are not compiled within the `BUILD_DIR` directory but in

  `project_build_mipsel/linuxonpsp` (variable `PROJECT_BUILD_DIR`):

  ```
  $~/buildroot$ ls project_build_mipsel/linuxonpsp/
  buildroot-config  busybox-1.9.0  linux-2.6.22  linux-2.6.22-bk  root
  ```

  Notice the `root` directory, defined in the Makefiles under the `TARGET_DIR` variable. As hinted by the name, this is the target directory for the *rootfs* used to generate the final image in the desired format (ext3, cpio, squashfs, cramfs, etc...). Exploring this directory provides some insight of the organization of the directory structure and the space used on the target system:

  ```
  $~/buildroot$ tree project_build_mipsel/linuxonpsp/root/
  |-- bin
  |   |-- busybox
  |   |-- cat -> busybox
  |   |-- cp -> busybox
  [...]
  |-- etc
  |   |-- TZ
  |   |-- fstab
  |   |-- group
  |   |-- hostname
  |   |-- hosts
  |   |-- init.d
  |   |   |-- S20urandom
  [...]
  |   |-- services
  |   `-- shadow
  |-- home
  |   `-- default
  ```

```
|-- init -> sbin/init
|-- lib
|   |-- libgcc_s.so -> libgcc_s.so.1
|   `-- libgcc_s.so.1
|-- linuxrc -> bin/busybox
|-- proc
|-- root
|-- sbin
|   |-- getty -> ../bin/busybox
|   |-- init -> ../bin/busybox
|   `-- mdev -> ../bin/busybox
[...]
```

- the `target` diretory includes all the Makefiles needed for the compilation of the final *rootfs* final. For example, when generating a `squashfs` formatted image, the part of the Makefile located in `target/squashfs/squashfsroot.mk` will be used. We also find there the skeleton of the `TARGET_DIR` directory (*target skeleton*) :

  ```
  $~/buildroot$ ls target/generic/target_skeleton/
  bin  dev  etc  home  lib  mnt  opt  proc  root  sbin  tmp  usr  var
  ```

  The user can include in these directories a script that is not specifically associated with one specific application.

As a summary, buildroot is divided in two main classes of directories: those containing the Makefiles, packages and the toolchain on one hand, and on the other hand the directories in which work is performed during the compilation, created when needed.

As a conclusion, let us emphasize one of the downsides of `buildroot`: its strong dependency on the coherence on the cross-compilation host environment. When configuring applications, tools such as `pkg-config` for example have the poor habit in case of failure to look in the host directory `/usr/lib/pkgconfig/`. One can imagine, when generating a *rootfs* on a host on which `dbus` is installed, that some problems will be met during the execution step on the targt system when such misconfigurations occur. In order to avoid such annoyances, a good habit is to perform all cross-compilation steps in a minimalistic and well understood *chroot* environment. A `debootstrap` installation of a stable debian distribution provides a good starting point for fine cross-compilation environment, as described at `http://wiki.easyneuf.org/index.php/Buildroot_HOWTO` [in French].

2.4 The serial port interface

The PSP provides several asynchronous serial ports: the wired connection next to the headset, and the wireless infrared link. Although an interrupt is associated to the events occuring on these ports (the UART is defined with interrupt number 0 [5]), the uClinux use of the asynchronous ports is based on a periodic polling of the port status, probably for historic reasons. This polling is performed at the same time than the management of other periodic events controlled by the timer interrupt (CPUTIMER interruption number 66 in [5]).

The communication through the serial port UART3 has been implemented in `linux/drivers/serial/serial_psp`

```
void psp_uart3_txrx_tick()
{
  if ( !s_psp_uart3_port_data.shutdown &&
       ( s_psp_uart3_port_data.txStarted ||
         ( !s_psp_uart3_port_data.rxStopped &&
           !( PSP_UART3_STATUS & PSP_UART3_MASK_RXEMPTY ) ) ) )
  {
```

```
    up( &s_psp_uart3_port_data.sem );  // wakes up character reception
  }
}
```

This interrupt service routine wakes up a kernel thread in charge of testing whether a character is available in the queue of the UART (Fig. 13): `psp_port_txrx_thread()`. Waking up this function is hence controled by a timer interrupt, initiliazed in `psp.c` under `arch/mips/psp` (function `psp_cputimer_handler()`). While reading data, access to the serial port is locked by the mutex `init_MUTEX_LOCKED( &portData->sem );` which was initialized in the kernel thread `psp_port_txrx_thread()`.

We find here an initialization sequence similar to that used for the joypad:

```
static int __init psp_serial_modinit()
{
[...]
portData->txThreadId = kernel_thread( psp_port_txrx_thread,
                         port,
                         CLONE_FS | CLONE_SIGHAND );
[...]}
```

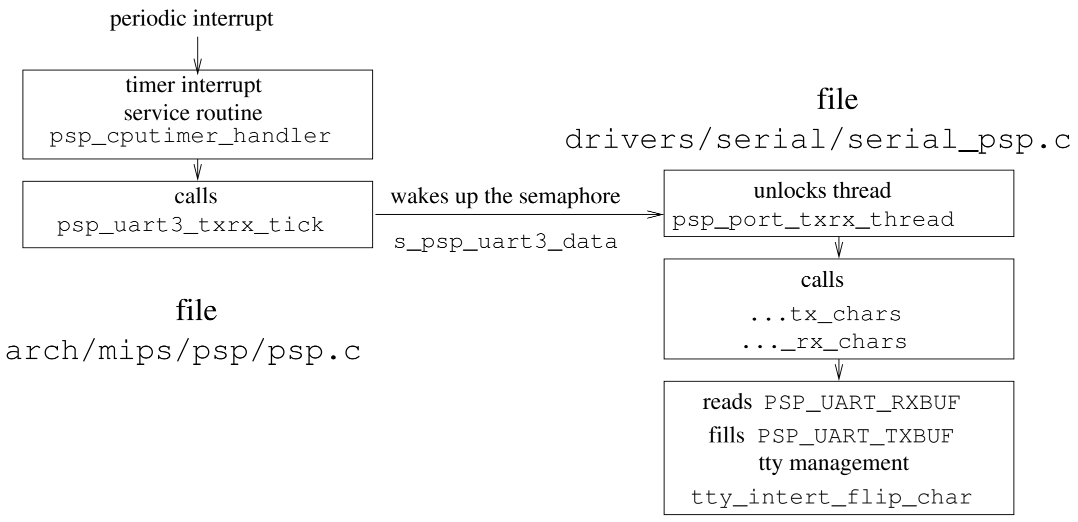

periodic interrupt
timer interrupt service routine psp_cputimer_handler
calls psp_uart3_txrx_tick
wakes up the semaphore
s_psp_uart3_data
file arch/mips/psp/psp.c
file drivers/serial/serial_psp.c
unlocks thread psp_port_txrx_thread
calls ...tx_chars ..._rx_chars
reads PSP_UART_RXBUF
fills PSP_UART_TXBUF
tty management
tty_intert_flip_char

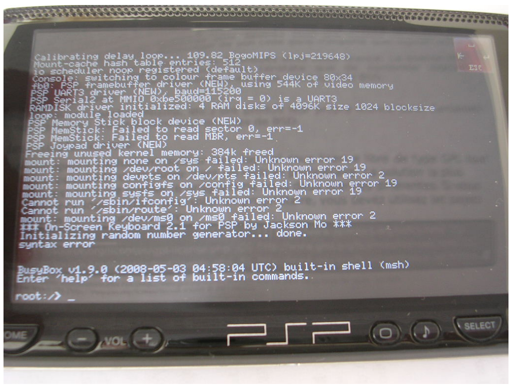
Figure 13: The periodic timer interrupt is also used for periodically probing the status of the serial port and interacting with the console.

The management of the serial port of the PSP, named `/dev/ttySRCi`, is described in the file `serial_psp.c` in the director `drivers/serial` of the kernel. We find there on the one hand a management of the communication following the description found in the IPL SDK and in homebrew PSP software, and on the other hand the interface between the serial port and a terminal – the console being one particular case.

From a user perspective, creating a console on a serial port is obtained in the usual Linux way (Fig. 14) by adding in `/etc/fstab` the line

```
# Put a getty on the serial port
ttyS2::respawn:/sbin/getty -L ttyS2 115200 vt100
```

So far, we have been able to get a working uClinux environment on PSP and communicate through a serial terminal. Our interest however is to add dedicated hardware to the game console:



```
Calibrating delay loop... 109.82 BogoMIPS (lpj=219648)
Mount-cache hash table entries: 512
io scheduler noop registered (default)
Console: switching to colour frame buffer device 80x34
fb0: PSP framebuffer driver (NEW), using 544K of video memory
PSP UART3 driver (NEW), baud=115200
PSP Serial2 at MMIO 0xbe500000 (irq = 0) is a UART3
RAMDISK driver initialized: 4 RAM disks of 4096K size 1024 blocksize
loop: module loaded
PSP Memory Stick block device (NEW)
PSP MemStick: Failed to read sector 0, err=-1
PSP MemStick: Failed to read MBR, err=-1
PSP Joypad driver (NEW)
Freeing unused kernel memory: 384k freed
mount: mounting none on /sys failed: Unknown error 19
mount: mounting /dev/root on / failed: Unknown error 19
mount: mounting devpts on /dev/pts failed: Unknown error 2
mount: mounting configfs on /config failed: Unknown error 19
mount: mounting sysfs on /sys failed: Unknown error 19
Cannot run '/sbin/ifconfig': Unknown error 2
Cannot run '/sbin/route': Unknown error 2
mount: mounting /dev/ms0 on /ms0 failed: Unknown error 2
*** On-Screen Keyboard 2.1 for PSP by Jackson Mo ***
Initializing random number generator... done.
syntax error

BusyBox v1.9.0 (2008-05-03 04:58:04 UTC) built-in shell (msh)
Enter 'help' for a list of built-in commands.

root:/> _
```

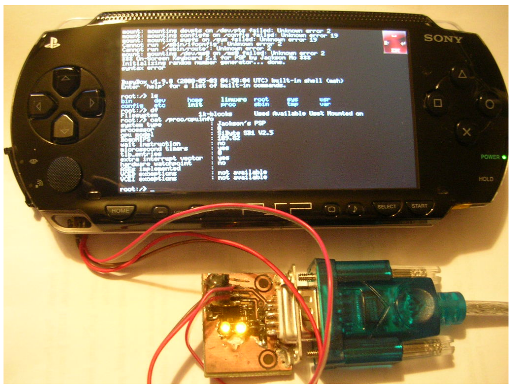
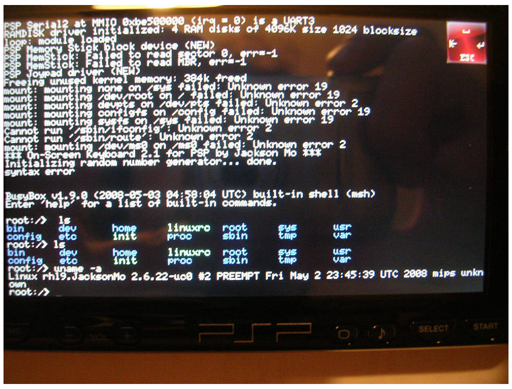


Figure 14: A working uClinux environment: left, the boot sequence and right, some commands typed from a PC running `minicom` as serial terminal software.

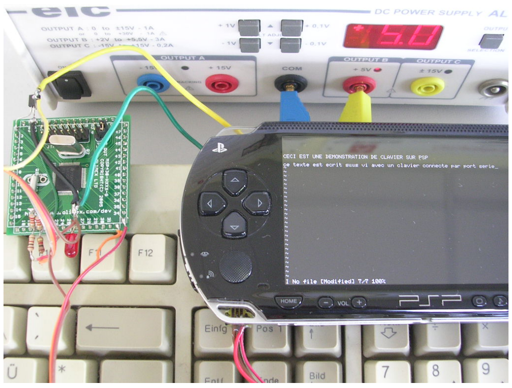
since there is no parallel bus and communication port other than the asynchronous serial port, the only possibility to add functionalities is through an external microcontroller. We will illustrate this aspect by adding an external keyboard and transfer data to the PSP through the RS232 port.

2.5 Application example: adding a PS2 compatible keyboard

Injecting characters received from the serial port in the tty layer [6, chap. 18 "TTY Drivers"] (`tty_flip_char()` function of `psp_uart3_rx_chars()` is exploited in two ways: either by running a terminal program on a PC transmitting characters through the serial port (for example `screen` or `minicom`), or by connecting a keyboard after converting its protocol to a RS232 stream. The first solution is the easiest but the least convenient since it requires a PC to communicate, removing the point of a handheld game console running GNU/Linux. The second one is the one discussed here.



```
PSP Serial2 at MMIO 0xbe500000 (irq = 0) is a UART3
RAMDISK driver initialized: 4 RAM disks of 4096K size 1024 blocksize
loop: module loaded
PSP Memory Stick block device (NEW)
PSP MemStick: Failed to read sector 0, err=-1
PSP MemStick: Failed to read MBR, err=-1
PSP Joypad driver (NEW)
Freeing unused kernel memory: 384k freed
mount: mounting none on /sys failed: Unknown error 19
mount: mounting /dev/root on / failed: Unknown error 19
mount: mounting devpts on /dev/pts failed: Unknown error 2
mount: mounting configfs on /config failed: Unknown error 19
mount: mounting sysfs on /sys failed: Unknown error 19
Cannot run '/sbin/ifconfig': Unknown error 2
Cannot run '/sbin/route': Unknown error 2
mount: mounting /dev/ms0 on /ms0 failed: Unknown error 2
*** On-Screen Keyboard 2.1 for PSP by Jackson Mo ***
Initializing random number generator... done.
syntax error

BusyBox v1.9.0 (2008-05-03 04:58:04 UTC) built-in shell (msh)
Enter 'help' for a list of built-in commands.

root:/> ls
bin     dev     home    linuxro  root    sys     usr
config  etc     init    proc     sbin    tmp     var
root:/> ls
bin     dev     home    linuxro  root    sys     usr
config  etc     init    proc     sbin    tmp     var
root:/> uname -a
Linux rh19.JacksonMo 2.6.22-uc0 #2 PREEMPT Fri May 2 23:45:39 UTC 2008 mips unkn
own
root:/>
```



```
CECI EST UNE DEMONSTRATION DE CLAVIER SUR PSP
ce texte est ecrit suus vi avec un clavier connecte par port serie_
~
~
~
I No file [Modified] 7/7 100%
```

Figure 15: Left: OnScreen Keyboard (visible as the icon on the top-right of the PSP screen), probably convenient to SMS writers. Right: a dedicated microcontroller (here an MSP430) takes care of converting the PS2 signals to RS232. uClinux must be adapted so that this new source of input signal is connected with the default tty.

We have added a PS2 keyboard using the simplest solution: the PS2 protocol (synchronous two-way protocol) is converted to an RS232-compatible stream (asynchronous protocol with separated signals for the emission and reception) using an MSP430 microcontroler. Only minor changes were brought to one of the examples provided with `msp-gcc` (the `gcc` cross-compiler targetting the MSP430) in `examples/mspgcc/pc_keyboard`. Without getting into the details of the PS2 protocol which is well managed by the microcontoler, we only modify this example program to communicate the result of a keystroke through the serial port rather than the default display on an LCD screen. We also must remove once every two character code since a keycode is generated each time a key is hit *and released*.

3 Conclusion

We have demonstrated the use of operating systems on two handheld game consoles, the Nintendo Dual Screen (NDS) and the Sony Playstation Portable (PSP). Understanding general principles of cross compilation toolchains and using opensource tools free to adapt to our purposes allowed us to work either on ARM or MIPS based architectures with uClinux or RTEMS ports.

We have seen that the Slot2 parallel bus of the NDS provides the required signal to interface dedicated hardware for control and data acquisition. Although providing a more powerfull processor and more memory, the PSP provides fewer opportunities for instrumentation and embedded hardware development since neither wifi nor USB are yet reverse engineered and the only communication port with external devices is an asynchronous seria port.

As is often the case in embedded hardware design, the right tools rather than the most powerful ones are ofter the ones providing the best performances: using opensource tools is one way of only being limited by one's knowledge and imagination rather than by hardware compliant with the few commercial tools the developer might have secured funding for. Futhermore, the uniformity of the opensource tools whatever the target architecture (`gcc`, `binutils` and `newlib` within the `buildroot` framework) means that adapting to a new architecture is possible within minimum time and cost once these environments are understood.

4 Acknowledgement

We are greateful to the GNU/Linux Magazine France team – and more specifically Denis Bodor – for allowing us to translate samples of the text used in articles [7, 8] published in this journal as well as using some of the pictures illustrating these documents. P.Kestener (CEA/IRFU, Saclay, France) mentioned the availability of the NDS BSP of RTEMSC. M. Bucchianeri patiently answered our questions as we were discovering RTEMS and the NDS BSP.

References

[1] Xiptech is a company providing some tools and a cross compilation toolchain targeted towards the MIPS architecture: *Porting uClinux to MIPS* available at `http://www.xiptech.com/download/porting.zip`

[2] the web site of Jackson Mo "Linux On PSP" includes the tools we used for generating a usable buildroot environment: `jacksonm88.googlepages.com/linuxonpsp.htm`

[3] the original web site `df38.dot5hosting.com/~remember/chris/` is unfortunately no longer active. A similar web site providing similar informations is available at `http://www.bitvis.se/articles/psplinux.php`

[4] TyRaNiD, *The Naked PSP*, presentation available at `http://ps2dev.org/psp/Tutorials/PSP_Seminar_from_Assembly.download`

[5] groepaz/hitmen, *Yet another PlayStationPortable Documentation*, available at `http://hitmen.c02.at/files/yapspd/psp_doc/`, and more specifically a list of the interrupts of the PSP detailed at `http://hitmen.c02.at/files/yapspd/psp_doc/chap9.html`

[6] J. Corbet, A. Rubini & G. Kroah-Hartman, *Linux device drivers*, O'Reilly (2005) available at `http://lwn.net/Kernel/LDD3`

[7] S. Guinot & J.-M. Friedt, *GNU/Linux sur Playstation Portable*, GNU/Linux Magazine France 114, March 2009, pp.30-40

[8] 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