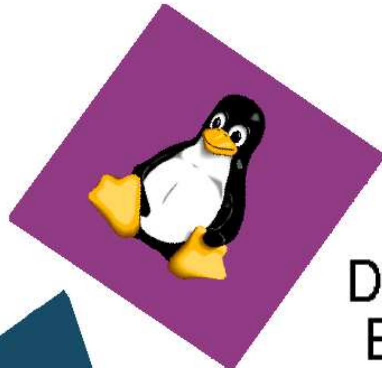

RokEPXA Development Environment From Scratch

Christophe Dubach
Alok Menghrajani

Nov 2003

Special thanks to Cédric Gaudin, Stéphane Magnenat, Julien Pilet and Damien Baumann for their continuous support.
To Miljan Vuletic and all the LAP professors without whom nothing would have happened.
To Nicolas Blanc for his help.

Contents

1 Introduction

Embedded systems are small computers that typically don't have a lot of memory. They usually also don't have keyboards or screens, so they communicate with the outside world using vt over the parallel port or telnet over the ethernet.

The goal of this document is to help you install Linux and some basic tools on such a system. This document is specifically written for the RokEPXA developped at EPFL, Switzerland; some part will probably also be useful for other architectures. This document is not a reference manual for the RokEPXA. When writting this document, we were migrating from linux 2.4.23 (which had bugs that severly affected us) to linux 2.6.0 (which was still in development phase). This means that recompling these tools in a couple of months will be very different from what we did.

This document will hopefully be extremly useful to understand how to use the development environment. The CD-ROM included with this report contains a complete, compiled and tested environment.

RokEPXA is designed for mobile robots, and has a really interesting architecture (Altera's EPXA); it combines two types of systems: an ARM922T microprocessor and a FPGA. This means while the arm microprocessor is running Linux and user programs like Java code, the FPGA can be specialized for tasks like controlling the robot's motors, doing image processing on camera input, serving as a network adapter or usb port, performing cryptographic calculations, etc...

RokEPXA is connected to the desktop machine (Linux development machine) in three ways:

- serial port - used for the vt terminal and also to transfer files from Redboot.
- parralel port - used for transfering Redboot via the JTag interface.
- ethernt connection - used for highspeed file transfer and development.

Since we had no hard disks, we created a mini Linux that we saved on the ROM (we have 8MB of ROM). While booting, this Linux will use a ramdisk, that is a virtual hard disk that lives in the RAM (we have 64MB of RAM). This mini Linux is what we transfer from Redboot. Once this system boots, it reprograms the FPGA in order to support the ethernet. We can then mount a remote file system and start working. (Besides transfer speed, the network file system also allows you to permanently save data across reboots).

After the development phase is over, you can put everything you need in the ROM and remove all the debugging features; this way you'll have a self-contained system.

1.1 Notes

1. You will often need root privileges on the desktop machine. You can either use the *sudo* command to gain these priviledges or login as root.

2. If you intend to use the same scripts as us, you should try to keep the same file names.

3. We noticed that newer versions of software sometimes create problems. If you want to avoid going through installation problems, you should stick to the same software versions as we had.

2 Hardware Configuration for the Rokepxa

The RokEPXA can be configured in different ways. The basic configuration is the power supply card, main card and ethernet adapter. Make sure you have the serial, parallel and rj45 (ethernet) cables connected.

Also check the jumpers are set correctly:

- W1, W2, W4 are set.

- W3 musn't be set. (So that we can access the flash ?)

2.1 Notes

1. You can either directly connect the rj45 cable to the desktop machine using a cross-cable, or you can use a normal cables and a hub.

3 Host System Configuration

Here is the list of tools you will need on the desktop machine:

- gcc - 2.95.4

- ld - 2.12.90.0.1

- ldd - 2.2.5

- autoconf - 2.58

- automake - 1.4-p4

- minicom - (1.83.1 ?)

- lrzsz (for file transfer in minicom).

4 Building the Toolchain

In order to generate code for the ARM processor, you will need a cross-compiler; a special version of gcc that will run on your desktop machine but generate code for the embedded architecture. The cross-compiler and some other tools that you will need are called a toolchain.

The compiler we will use is called uclibc. It is similar to gcc, except that the it's libary is significantly smaller in size.

We will also create a native toolchain (a compiler that will run and generate code for ARM).

We will use a script called buildroot (the script is actually a huge number of files kept in the directory *buildroot/*) to build our toolchains. This script can be downloaded from a CVS:
cvs -d:pserver:anonymous@uclibc.org:/var/cvs login
cvs -z3 -d:pserver:anonymous@uclibc.org:/var/cvs co -P buildroot

The buildroot script is great if it works (because it does a LOT of stuff). It can become really difficult to use if you run into problems. This is why we recommend that you use the same version as we did (the version included on our CD-ROM is patched/fixed).

The script will automatically download a couple of packages from the internet. Since our script was designed for kernel 2.4, we downloaded linux kernel 2.6.0 and told the script to use our kernel headers.

Make sure you have a clean PATH variable. The script might otherwise get confused.

4.1 Modifying the Makefile

The first thing you will have to modify is the *buildroot/Makefile*. We will tell the script that we are using an ARM processor. For some weird reason, floating point emulation (called soft float) doesn't work. So we will compile with hard float enabled, and we will later enable floating point emulation at the kernel level.

We tried to compile strace (a very useful tool), but ran into multiple problems. We believe the next versions of the buildroot scripts will fix these problems.

Here is the list of things we modified. If you use a newer version of the script, you will have to compare your configuration with our's.

- ARCH:=arm
- USE_UCLIBC_SNAPSHOT:=false
- USE_BUSYBOX_SNAPSHOT:=false

- BUILD_WITH_LARGEFILE:=false
  (you might want to consider setting true here due to bugs in various software like strace)
- TARGETS+=gcc3_3_target
- TARGETS+=ncurses ncurses-headers # for gdb support
- TARGETS+=gdb
- SOFT_FLOAT:=false
- remove tinylogin

You must select TARGETS+=system-linux instead of TARGETS+=kernel-headers if you downloaded linux yourself.

4.2 Setting the kernel source path

You must set the folloing in *buildroot/make/system-linux.mk*: LINUX_SOURCE=linux-2.6.0-test11-rmk1/

4.3 Running buildroot

You must run the buildroot script once in order to download uClibc. You will then need to fix some problems the the uClibc config and rerun the buildroot script.

cd buildroot/
make

4.4 Modifying the uClibc files

You must set the following in *buildroot/uClibc.config* and *buildroot/uClibc.config-locale*: UCLIBC_HAS_SYS_SIGLIST=y

There were some problems with missing system calls in linux 2.6.0 (eg: create_module and get_kernel_syms was removed). Some software (like busybox) were written to support both versions of linux. Other code (like uclibc) doesn't work with 2.6.0. These problems can be solved by using "clean" headers (which are unavailable right now, when we are writing these lines).

If you run into similar problems, you can take our fixed *create_modules.c* and *syscalls.c*.

You must copy *asm-generic* from the linux/include folder to *buildroot/build_arm/staging_dir/incl*
Same thing must be done for the folder *buildroot/build_arm/uClibc-0.9.24/include*.

There is a bug in *include/features.h* due to a missing #define __user. (Remember the include/ directory is copied in 3 places, so you must fix the bug everywhere).

In *ioperm.c* you must replace BUS_ISA by CTL_BUS_ISA.

Make sure that the file *buildroot/source/uClibc.config* has the line: UCLIBC_HAS_RPC=y

4.5 Running Buildroot

You can now run the Buildroot again.

Due to a bug in uClibc (which by default enables soft float when it is creating a cross compiler for the arm), the compilation of the buildroot script is done in two steps.

The script will ask you which specific arm you are using. We answered: 6. Arm 922T

When the script tries to create uClibc, you will get an error message like this:
buildroot/build_arm/staging_dir/bin/arm-linux-ld: ERROR: buildroot/build_arm/staging_dir/lib/gcc-lib/arm-linux/3.3.2/crtbeginS.o uses hardware FP, whereas libpthread-0.9.23.so uses software FP
File format not recognized: failed to merge target specific data of file

You must then delete: ARCH_HAS_NO_FPU everywhere in the file *buildroot/build_arm/uClibc-0.9.23/extra/Configs/Config.arm* and modify the file *buildroot/build_arm/uClibc-0.9.23/.config* :

- #ARCH_HAS_NO_FPU=y
- #UCLIBC_HAS_SOFT_FLOAT=y
- HAS_FPU=y

You must then do *make clean* followed by *make* in the uClibc directory and then *make* again in *buildroot/*.

Once the script finishes, you will have a folder containg files for the target system in *buildroot/build_arm/root/*, and the cross-compiler in *buildroot/build_arm/staging_dir/*.

Make sure you add the *staging_dir* to your path.

Don't forget to copy *buildroot/build_arm/root/* to the NFS.

4.6 Notes

1. You can try to understand and solve why soft floating point (having a compiler that generates the floating point code) doesn't work.

2. If you get an error message like: */bin/sh: line1: no: command not found* then you need to update the gettext utility (you are probabably missing msgfmt). You might also have to update autoconf to version 2.58 and automake.

3. In case something goes wrong and you need to rerun the buildroot script, don't use *make clean*. You should rather remove the directories: *build_arm* and *toolchain_build_arm*.

4. If you need to rebuilt Busybox (because you forgot to activate an option) then do a *make clean* in *buildroot/build_arm/busybox-0.60.5/* and delete the *buildroot/build_arm/root/* folder. Then rerun the script from *buildroot/*.

5. If you get an error like this:
buildroot/build_arm/staging_dir/lib/gcc-lib/arm-linux/3.3.2/../../../../arm-linux/bin/ld: cannot open crt1.o: No such file or directory
collect2: ld returned 1 exit status
It means the script used the wrong linker, you need to clear your PATH and start all over again.

4.7 Modifying the Busybox config

Busybox is a collection of Unix tools that allows you to have basic GNU tools in a single small executable.

The buildroot script should create Busybox for you. You can also create it manually. Download and decompress the sources. You must check in the Makefile that you are going to use the right cross-compiler.

You might also have to set DOSTATIC to true in the Makefile (if you don't have a loader).

The config file lets you choose exactly what programs you want, so you can minimize space usage. Here is what we used:

In *busybox.config*:

- remove CONFIG_FEATURE_2_x_MODULES
- remove CONFIG_FEATURE_QUERY_MODULE_INTERFACE
- add CONFIG_FEATURE_2_6=y
- add CONFIG_LSMOD, INSMOD, RMMOD
- remove FDISK, FDFLISH, FDFORMAT, ... because of a bug in *scsi.h*.

In *busybox.Config.h*:

- BB_ASH, BB_INIT, BB_STTY, BB_TTY
  ASH is the shell, if you wish to, you can select an alternate one. INIT is the first process, don't know what will happen if this is desactived.

- BB_AR, BB_CP, BB_ECHO, BB_GREP, BB_GUNZIP, BB_GZIP, BB_LN, BB_LS, BB_MKDIR, BB_MORE, BB_MV, BB_RM, BB_RMDIR, BB_TOUCH, BB_VI, BB_WC, BB_WHICH
  Basic shell tools.

- BB_INSMOD, BB_LSMOD, BB_MODPROBE, BB_RMMOD
  Tools required to load modules such as ethernet.

- BB_MKNOD
  Required to initialize the fpga.

- BB_MOUNT, BB_UMOUNT
  Tools required to mount the NFS.

- BB_IFCONFIG, BB_PING, BB_ROUTE, BB_TELNET
  Tools required for networking.

- BB_BASENAME, BB_DIRNAME, BB_PWD
  Tools that might be used by scripts.

You must also activate some features:

- #define BB_FEATURE_SH_IS_ASH
  Depending on the shell you have choosen.

- #define BB_FEATURE_USE_INITTAB
  Default and recommended.

- #define BB_FEATURE_NFSMOUNT, BB_FEATURE_MOUNT_FORCE
  Support for remote NFS volumes.

- #define BB_FEATURE_NEW_MODULE_INTERFACE
  Since we have a post 2.1 kernel.

- maybe enabling INSMOD_VERSION_CHECKING will allow checking versions for modules (the last time we tried it, it didn't work).

- #define BB_FEATURE_IFCONFIG_STATUS
  Useful for debugging.

- #define BB_FEATURE_IFCONFIG_HW
  This is important for us, since our fpga doesn't assign a MAC address.

After compiling once, you need to fix the following files: *include/linux/loop.h* add #include ¡asm/posix_types.h¿

5 Linux Kernel, version 2.6.0-test11-rmk1-rokepxa1

The Linux kernel that we used is 2.6.0-test11, with the rmk1 patch (a patch specifically for ARM), and Christophe's rokepxa patch. (Christophe's patch is actually based on Cédric's old patch).

5.1 Patching the Kernel

`man patch :-)`

The rmk1 patch (www.arm.linux.org.uk) adds support for ARM based linux.

The custom patch does the following:

- adds support for the ethernet (the cirrus module was removed in 2.6.0).
- corrects a bug in uart00 (the console freezes when you hit tab or sometimes other keys).
- fixes problems with pld_epxa.c
- defines the RokEPXA architecture.

5.2 Configuring

Before compiling the kernel, you need to configure it. You can edit the *.config* file, or use the *make menuconfig* command. We noticed that sometimes the kernel doesn't recompile correctly if only the *.config* file was modified, so we recommend always running *make menuconfig.*

Here is the list of options that you should enable.

Code maturity level option ⟶

  * Prompt for development and/or incomplete code/drivers

General setup ⟶

  * System V IPC

    Sysctl support (should be safe to disable, if you want to save 8KB !)

Loadable module support ⟶

  * Enable loadable module support
    * Module unloading
      * Forced module unloading (should be safe to disable).

- ⊘ DISABLE Set version information on all module symbols
  For some reason modules can't be loaded (there is some symbol mismatch) if this option is set. Probably due to the fact that we are cross-compiling.
- * Automatic Kernel module loading (should be safe to disable).

System type ⟶

ARM system type (RokeEPXA) ⟶

- X RokEpxa

RokeEpxa ⟶

- * Support for PLD device hotplugging (experimental)

General setup ⟶

- * NWFPE math emulation
  This option is needed because we don't have any floating point unit, and the JVM will require it.
- * Kernel support for ELF binaries
- * Kernel support for a.out binaries
- * Preemptible Kernerl (EXPERIMENTAL)
- STOP Default kernel command string:
  "console=ttyUA0,115200 mem=64M root=/dev/ram0 ramdisk_size=16384 initrd=0x2000000,987006"
  This option is very important ! console=... enables the embedded system to be controlled via a terminal, mem=... tells the kernel how much RAM we have, root=... enables us to boot from the RAM disk (since we don't have any hard disks), initrd=... tells the kernel where the initrd will be. The 987006 value is the file system's size. Each time you change the size of your file system, you must change this value and recompile the kernel.

Block devices ⟶

- * RAM disk support
  We need this since we don't have any hard disks.

  4096 Default RAM disk size
- * Initial RAM disk (initrd) support
  We need this since we don't have any hard disks.

Networking support ⟶

- * Networking support

  Networking options ⟶

- * Unix domain sockets
- * TCP/IP networking
  We need this since we will mount a NFS.

* Network device support

Ethernet (10 or 100Mbit) ⟶

- * Ethernet (10 or 100Mbit)
- M Cirrus support
  Depending on your hardware, you will need one of these modules.
  You must add this as a module, because the fpga needs to be initialized for the 'network card' to exist. The fpga will be initialized after the Linux has booted. The cirrus module will then be loaded.

Input device support ⟶

- * Serial i/o support
- * Serial port line descipline
- * Joysticks
  - * I-Force devices
    just in case we have some extra time :-)

Character devices ⟶

- * Unix98 PTY support
- 256 Maximum number of Unix98 PTYs in use (0-2048)

File systems ⟶

- * Second extended fs support
- * ROM file system support

Pseudo filesystems ⟶

- * /proc file system support
- * /dev file system support (OBSOLETE)
  - * Automatically mount at boot
  - * Debug devfs
- * /dev/pts file system for Unix98 PTYs

Pseudo filesystems ⟶

- * Compressed ROM file system support (Should be safe to dis-activate)

Network File Systems ⟶

* NFS file system support

  * Provide NFSv3 client support

Kernel hacking ⟶
You should enable everything as you will be working with a lot of experimental code, and the kernel might crash.

You can now do *make dep; make clean; make zImage; make modules; make modules_install.*

The kernel is created in arch/arm/boot/zImage. The kernel modules are created in /lib/modules/2.6.0-test11-rmk1.

5.3 Notes

1. Remember to recompile the kernel each time the file system's size changes. You can simply do a make zImage if you don't change anything else. I recommend changing the command string using *make menuconfig*, as *vi .config* might not always work.

2. Your final kernel should be slightly less than 1 MB.

6 File System, ext2 format

You must create an initrd file system, that will be stored in the ROM and loaded by the kernel when it boots. This initrd will be in ext2 format. It is stored as a normal file on your desktop machine, but you can mount it using the loopback device. Once mounted, you can transfer files into the file system (just like any other Linux device).

Your file system shouldn't be too small as you will need enough space to store the kernel, modules, basic Linux tools (shell and various other tools) and a JVM. On the other hand the bigger the file system, the longer it will take to transfer it to the ROM (an operation that takes about 30 minutes and that you normally should be doing only once, but you might have to do it several times if things go wrong). The file system must also of course be small enough to fit in the ROM (it will first be compressed). So in our case, a 6 MB file system is a good choice.

The kernel needs to know the size of file system. You therefore need to enter it in the kernel command (you can do this using *make menuconfig*). The size that you need to enter is the file system's compressed size. So you can now do a *ls -l initrd.ext2.gz* and copy the file size into the kernel.

6.1 Creating

Although the buildroot script created a folder *buildroot/build_arm/root/* which contain most of the files your file system will need, you must recreate a fresh file system.

There are two ways you can create the file system: grab an existing one and modify it, or create one from scratch.

The file system we will create will use the ext2 format. It will be stored as a normal file on your desktop machine, but you can mount it using the loopback device.

First create the image file: *dd if=/dev/zero of=initrd.ext2 bs=16M count=1.*

Now convert it to an ext2 file system: *sudo mke2fs -F -v -m0 -b 1024 initrd.ext2.*

Finally you can gzip the initrd: *gzip initrd.ext2*

6.2 Mounting

To mount the file system you need to first extract it: *gunzip initrd.ext2.gz*
Create a folder where it will be mounted: *mkdir fs*
Finally mount it: *sudo mount initrd.ext2 -f ext2 -o loop fs/*

6.3 Unmounting

To unmount, you must do the reverse operation. *sudo umount fs/*
rmdir fs
gzip initrd.ext2

6.4 Content

This part is slighty tricky. The buildroot script create *buildroot/build_arm/root/* which contains most of the files you'll need. Unfortunatly we need to strip this folder down so it fits on our tiny system. Here is what we did:

- Removed *share/*
- Moved *usr/lib/*, *usr/include/* and *usr/arm-linux/* to the NFS
- Copied *usr/bin/* to the NFS
- Removed the following files from *usr/bin/*:
  arm-linux-*, (everything that's not a symlink), leave the LOADER !!!
  (ld and ldd, enough). How big are the libaries?
- Copied *lib/* to the NFS

You can copy *buildroot/build_arm/root/* but you must first save some space: remove everything in *share* and copy the *lib* and *include* folders to the NFS.

COPY libgcc_s !!!

Here is the minimum content of the file system:

- Kernel modules
  sudo mkdir -p fs/lib/modules
  sudo cp -r /lib/modules/2.6.0-test11-rmk1 fs/lib/modules

- Busybox
  You will learn more about Busybox in the next section.

- FPGA ethernet design file
  The file is called rokepxa_extcam3d.sbi. FIXME

- /etc
  You must create the /etc directory and put in 3 files: fstab, inittab, startup.

- /dev
  You must create the /dev directory. HOW ? FIXME

6.5 Notes

1. It is a common mistake to tar.gz the file system. The file system should only be gzipped.

2. The mytools/ directory contain scripts to mount and unmount the file system.

3. Remember to change the kernel command option each time you modify the file system's content.

4. Sometimes simply mounting and unmounting a file system can change it's size.

5. It's a common mistake to change the file size in the kernel command line and forget to recompile the zImage.

6. If you remove a file, it won't reduce the filesystem's size unless you recreate a fresh one (we haven't found a way to zero the content of deleted data, so the compression isn't efficient).

7. Todo: modify buildroot so it creates a smaller root/.

8. Todo: modify buildroot so it puts the native compiler in a different folder than /usr/bin.

7 Muboot

You are now almost ready to boot your mini system. You must still create a boot loader. The boot loader is the very first piece of code that gets run when a system boots. We used Muboot, which does a basic check on the RAM and decompresses the kernel. The boot loader then hand over the control to the kernel, which creates a ramdisk and mounts the file system. It also displays and allows you to use the keyboard via vt (over the serial cable).

The Muboot tool takes as input a zImage kernel image and a file system. It puts them together and adds the boot loader code. The resulting file is called flash_boot_rokepxa.bin:
cp /linux-2.4.19-rmk7-rokepxa4/arch/arm/boot/zImage /muboot/src/ cp initrd.ext2.gz /muboot/src/initrd (Note: the initrd changes it's name)
cd /muboot/src/
make clean
make

7.1 Notes

1. You can improve the boot loader so that the file system's size is passed to the kernel at boot time (and not with the kernel command method). This way you won't need to recompile the kernel each time the file system's content changes. You can also improve the boot loader by supporting the bzImage format, which will decrease the image size.

2. There is a script in the mytools/ folder that creates the loader and flasher.

8 Flasher

The flasher is a tool that allows you to copy the flash_boot_rokepxa.bin into the ROM. This way whenever you reboot the rokepxa, you will always have your Linux.

The rokepxa can be booted in two different ways: restart (which is performed using the Jelie tool) which will reboot the machine using the RAM, and reset (which can be performed usign the reset button, Jelie or by switching the card off and on) which will reboot the machine using the ROM.

The process to flash the ROM happens in two steps. The flasher takes the flash_boot_rokepxa.bin and adds some code to copy write data to the ROM. The resulting file is: flasher. This file is then copied to the RAM (this process takes about 30 minutes) and the machine is restarted (boot from RAM). The flasher will then be run and will copy the content of the RAM (the flash_boot_rokepxa.bin) into the ROM. Once the process is terminated, you can reset the card and you should be able to boot into Linux.

Here is how you prepare the flasher:
cp flash_boot_rokepxa.bin /flasher/src/flash_image (Note: the file changes it's name)
cd /flasher/src/
make clean
make

9 Minicom

Minicom is a simple vt software that will allow you to communicate using the serial port. It hasn't got anything special, you can use any terminal software as long as it supports vt100 at 11500 bauds.

Minicom has a few things to be aware off:

- Activate line wraps.
- You can clear the screen by typing ctrl-A followed by C.
- ctrl-A Z provides help.

10 Jelie

Jelie is a tool that will allow you to transfer data to the RAM. It can also be used to reset or restart the rokepxa (this can be especially useful if you are working remotly). The Jelie tool uses JTAGS.

Before launching Jelie you should run your terminal (eg: Minicom) in another window.

There are prewritten scripts for Jelie (/jelie/scripts/), you should use them.

To run jelie:
cd /jelie/
sudo ./test

Here are some basic commands:

- list
  List all the commands. Make sure you have the commands starting with epxa.
- expa_configure
  Initialize the rokepxa. You should execute this command before using any other epxa commands. (The scripts usually do this for you).
- epxa_resetcpu
  Reset rokepxa (boots from RAM)

- expa_restartcpu
  Restart rokepxa (boots from ROM)
- source scripts/script_name.je
  Runs script stored in scripts/

To transfer the Redboot, just do: *source scripts/redboot.je*

11 Redboot

We use the Redboot bootloader (see www.redhat.com and sources.redhat.com/redboot/).

11.1 Installation

- Download jtag_redboot.bin using Jelie at address 0 (this will load the redboot_ram image from address 0x30000).
- Download rokepxa_redboot_ram.bin to the RAM (using Jelie) at address 0x30000.
- Reboot (don't restart !)
- Minicom should be set to 57600 bauds.
- Transfer rokepxa_redboot_rom.bin to the RAM (load -r -b 0x1000000 followed by ctrl-A S -¿ select xmodem and select the file).
- fis init
- Transfer rokepxa_redboot_rom from ram to rom: fis create -b 0x1000000 -l size -r 0x40000000 -e 0x00000000 RedBoot
- fis lock RedBoot
- cache on

11.2 Using

- Transfer initrd (to be done once only): load -r -b 0x2000000...
- fis create -b 0x02000000 -l size -f 0x40120000 -r 0x02000000 initrd
- Transfer linux (to be done once only): load -r -b 0x00200000...
- fis create -b 0x00200000 -l size -f 0x40020000 -r 0x00200000 Linux26
- fis load initrd
- fis load Linux26

- exec
- Quickly switch back to 115'000 bauds !

11.3 Notes

1. todo: recompile redboot so that we don't need to switch speeds.

12 Network

You will need network support for debugging purpose. Being able to mount a NFS file system will allow you to quickly access files (don't need to create a new initrd and wait 30 minutes to copy it to the ROM). It will also allow you to save files (since you are using a RAM disk, everything you write to the RAM disk is lost if the system crashes/reboots).

If you compiled everything properly (enabled network support in the kernel and added the tools in Busybox), then all you need to do is the following after the kernel has booted:

Create the PLD device: */bin/mknod /dev/pld c 254 0*.

Configure the PLD for ethernet support: */bin/cp /design/rokepxa_extcam3d.sbi /dev/pld*.

Load the ethernet driver: */sbin/insmod cirrus*.

Setup the network:
/sbin/ifconfig eth0 hw ether 90:00:00:10:10:10
/sbin/ifconfig eth0 192.168.0.2

Assuming that the nfs is configured in the fstab file, all there is left to do is: *mount /nfs*.

12.1 Notes

1. You might notice that *lsmod* reports that the cirrus module is begin used by 2. (This is ok ?).
2. If you want to unload and modify the fpga here is what you need to do:
   umount /nfs
   ifconfig eth0 down
   rmmod cirrus

13 Intermediate Testing

You can now check that everything is fine by mounting a remote file system and running a typical hello world application.

You should now be able to run any compiled code on the rokepxa. Try compiling a hello.c program on the desktop machine using uclibc. Make sure you compile your test program as static:
echo "int main() {printf("hello world
n"); return 0;" ¿ hello.c}
arm-linux-gcc -c hello.c
arm-linux-gcc –static -o hello hello.o

Copy the hello application to the NFS folder, mount the NFS on the rokepxa and run the application. If everything went fine, "hello world" should be displayed.

13.1 Notes

1. You can check that a file is compiled static: *arm-linux-ldd file_name*

Kernel panic: No init found. Try passing init= option to kernel.

- check that you have busybox on the file system - check that busybox is compiled static or that you have the right libraries in /lib

Out of blocks (decompressing initrd), means there is a mismatch in the kernel command and file system size. The kernel command is displayed right after the kernel is decompressed.

14 Kaffe, version 1.1.3 with custom patch

The last step is to compile the JVM. We recommend using the latest version of Kaffe, as the production version (1.0.7) is very old and doesn't compile easily.

We had to fix a bug with the JIT (just in time) engine. The patch will be included in 1.1.4.

Kaffe is compiled in two steps: first you will compile it for your desktop machine (this will generate a folder called kaffeh). Then you can cross-compile for the embedded system.

14.1 Compiling x86kaffe

Decompress the kaffe sources and run:
./configure –prefix=/home/.../x86kaffe/ –enable-pure-java-math Note: the prefix must be absolute
make
make install

You should test that the jvm is working by compiling *hello.java* and running it: *x86kaffe/bin/javac hello.java*
x86kaffe/bin/java hello

14.2 Compiling armkaffe

If you are using an older version of kaffe, you might have trouble with some files having asm instructions on multiple lines. You must add \'s manually.

(Copy the *x86kaffe/libraries/javalib/rt.jar* to */tmp/rt.jar*).

You must now clean the source directory:
make clean rm config.cache

Setup the cross-compiler and build armkaffe:
Make sure you have arm-linux-gcc in your path !
KAFFEH=/home/.../x86kaffe/bin/kaffeh ./configure –host=arm-linux –build=i386-linux –enable-pure-java-math –without-x –with-threads=pthreads –with-engine=jit –prefix=/usr/kaffe

Before continuing you need to edit the main *Makefile* and remove test from the subdirs.

make
make install

Once the build is sucessful, you should reset the environment (close the terminal and open a new one).

Now test armkaffe by running *hello.class* (generated by x86kaffe) on the arm, and also by recompiling *hello.java* on the arm.

14.3 Notes

1. We couldn't compile Kaffe with jthreads (that's why we use pthreads).
2. JIT3 isn't ported to ARM, that's why we specify JIT (JIT3 is the default).
3. You could try to compile Kaffe on the arm iteself, since we have a native toolchain.

A Overall view