Building a multimedia embedded Linux system from scratch

Michael Opdenacker
Bootlin
https://bootlin.com/

Tutorial goals

My goals

Show you how easy it can be to create a feature-rich embedded Linux system from scratch.

Advertise for cool stuff: Scratchbox, QEMU...

Your goals

Become a Linux system developer!

Know how to do all these things by yourself.

Be able to stick a “hand made” label on the back of your product.

Tutorial goals (2)

Here's what you can build by yourself in less than 2 hours:

Compile a Linux kernel from its sources
  Learn how to use the configuration interface

Learn how to use the Scratchbox cross-compiling environment.

Cross-compile various libraries and tools, in particular DirectFB and its media player.

Boot Linux through NFS

Shrink your filesystem: removing unused files, stripping libraries and executables.

Where to find these materials

Easier if you're not progressing at the same speed.

The slides are available on
https://bootlin.com/pub/conferences/2008/ols/

All the downloads can also be found there

Tutorial requirements

Required

An x86 notebook

A GNU/Linux distribution (see next page)

A connection to the Internet

Some free disk space (1 GB)

GNU/Linux distribution

Tutorial tested on Ubuntu 8.04
    Should still work fine at least on 7.10.

Debian:
    The same instructions should work (not tested)

Red Hat / SUSE and others:
    Should work too, but using the package management system provided by the distribution. You may have to look for names of packages corresponding to the ones we give.

Download Scratchbox - Debian / Ubuntu

Debian based distributions

Add the below line to `/etc/apt/sources.list`:

```
deb http://scratchbox.org/debian/ apophis main
```

Download the Scratchbox packages:

```
apt-get update
apt-get install scratchbox-core \
scratchbox-libs scratchbox-devkit-cputransp \
scratchbox-toolchain-cs2005q3.2-glibc-arm
```

Download Scratchbox - Other distros

Open the below URL in your browser:

http://scratchbox.org/download/files/sbox-releases/apophis/tarball/

Download the following files (you can click on the links):

scratchbox-core-1.0.10-i386.tar.gz
scratchbox-libs-1.0.10-i386.tar.gz
scratchbox-devkit-cputransp-1.0.7-i386.tar.gz
scratchbox-toolchain-cs2005q3.2-glibc2.5-arm-1.0.7.2-i386.tar.gz

Alternative (64 MB instead of 131 MB)

https://bootlin.com/pub/conferences/2008/ols/downloads/

Extract all these archives in / (will create a `/scratchbox` directory)

Now initialize Scratchbox:

```
/scratchbox/run_me_first.sh
```

Install other packages

Needed by make xconfig (Linux, BusyBox):

```
apt-get install libqt3-mt-dev g++
```

QEMU emulator:

```
apt-get install qemu
```

Download the Linux kernel

This will also save time:

http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.25.tar.bz2

Alternative: (39 MB instead of 47 MB)

https://bootlin.com/pub/conferences/2008/ols/downloads/linux-2.6.25.tar.lzma

Choosing a hardware platform

Looking for a platform with

An ARM CPU (choice made by many people)

A graphics controller

A sound card

Available through emulation in QEMU

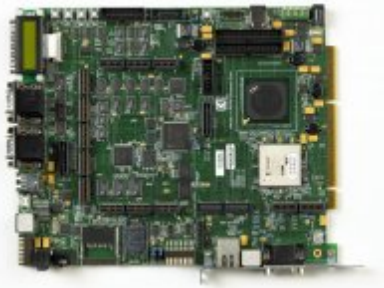
Easy to use in a public tutorial:
not too many custom things to do, having to devote time to external concerns.

Reviewed platform candidates

Neo 1973 emulated by QEMU
- Needs a custom or development QEMU
- Not supported by the mainstream Linux kernel.

Sharp PDAs emulated by QEMU
- Good overall support from mainstream Linux. Working LCD. But sound chip failing to initialize in mainstream Linux with I2C issues. Need to investigate why. But no onboard network device... Complicated development.

ARM Versatile PB board emulated by QEMU
- Missing sound (not supported by QEMU yet). Perfectly supported by mainstream Linux. LCD available.



Bottom up philosophy

Only building and adding the pieces that you need

It usually means compiling software from source, rather than using a build system with dependencies.

Wait for the tools to complain before adding extra libraries, software and even devices.

This way, you don't have so many things to learn and remember. It's easier to remember from failures than from successes ;-)

Caution: used it only with software. Your personal life may crash otherwise.

Cross compiling pain

Cross-compiling embedded tools by hand is rather easy:
Linux kernel, BusyBox...

Cross-compiling regular desktop / server libraries is painful:

- Need to instruct configure scripts to cross-compile.
- Need to avoid interference from host system libraries
- Makefiles try to test the availability of prerequisites by compiling and trying to execute test programs. Executing a target binary is not possible without tricks.
- Many tools and libraries were not tested in a cross-compiling environment by their developers.

Scratchbox

http://www.scratchbox.org

Cross-compiling toolkit designed to make embedded Linux application development faster and easier.

Provide a sandbox environment that emulates some characteristics of the target system

- Dependencies are not mixed with host system's libraries
- Transparent cross-compiling, making building tools believe they are doing a native compile job: no tweaking to support cross-compiling, and fast compiling on a cheap `x86` box.

Supported architectures : `arm` and `x86`.
Experimental support for `cris`, `mips` and `ppc`.

History

Research started in 2002

First public release in 2003

Scratchbox 1.0 in February 2005, « Apophis »

- Still maintained, with regular updates to various components from 2005 to 2008.

Scratchbox 2.0

- In development : version 1.99.0.23 released in February 2008.

Developed by Movial, sponsored by Nokia.

Used for the Maemo environment of the Nokia Internet tablets.

Features

Chrooted environment

Running on the host, but only target files are visible.

Transparent cross-compiling

Transparent execution, either through Qemu or remote execution using `sbrsh`.

Comes with ready-made cross-compiling toolchains and tools to build Debian packages.

Supports both uClibc and glibc.

Provides basic root filesystems.

Configuring Scratchbox

Scratchbox is installed in `/scratchbox`

Add your account to the Scratchbox system

```
sudo /scratchbox/sbin/sbox_adduser <user>
```

This will

- Add your user to the `sbox` group
- Create files and directories
  inside the Scratchbox system for your user

Need to log out and log in again for the group change to take effect

Login using `/scratchbox/login`

Scratchbox says « No current target »

Configuring a target

Run `sb-menu` to create a new target

Setup a new target

Target name: `mediaplayer`

Compiler: `cs2005q3.2-glibc-arm` (cross)

Devkits: `cputransp`

CPU transparency method: `qemu-arm-0.8.2-sb2` (emulation)

Do you wish to install a rootstrap on the target: `no`

Do you wish to install files to the target: `yes`
Just select the C library option.

Do you wish to select the target: `yes`

Exploring the target

The prompt is now
`[sbox-mediaplayer: ~] >`

Inside the `mediaplayer` target,
in your Scratchboxhome directory

Your target root filesystem is stored in `/targets/mediaplayer`

A set of symbolic links from `/` allows to think that you are actually running on the target

Some host tools, provided by Scratchbox, are still available
Example: `/scratchbox/tools/bin/vi`

Using Scratchbox

Test a simple program provided by Scratchbox

Extract

```
tar xfz /scratchbox/packages/hello-world.tar.gz
```

Configure and compile

```
cd hello-world
./autogen.sh
make
```

Check that the program is compiled for ARM

```
file hello
hello: ELF 32 bit LSB executable, ARM [...]
```

Run it: `./hello`

Using Scratchbox (2)

Possible to cross-compile and install libraries in a transparent way

```
./configure
make
make install
```

And then, to cross-compile programs using these libraries.

Cross-compiling is a lot easier.

How Scratchbox works (1)

- The user is chrooted into `/scratchbox/users/<user>/`
- This directory looks like a regular root filesystem
  - Most directories are symbolic links to `target/links/<dir>`
  - These are again symbolic links to `target/<target>/<dir>`
  - They are switched when changing the target
  - The home directory in `/scratchbox/users/<user>/home/` is not target-specific
- Various host directories are remounted inside the target using the `--bind` option of `mount`: `/scratchbox, /tmp, /proc, /dev/, /dev/pts, /dev/shm, /sys`

How Scratchbox works (2)

Target root filesystem is stored in
`/scratchbox/users/<user>/targets/<target>`

Contains the filesystem hierarchy that should be used on the embedded devices

Configuration file stored in
`/scratchbox/users/<user>/targets/<target>.config`

- Defines the architecture, CPU transparency method, cross-compiler, compiler and linker options, host compiler, host compiler options...
- Many other variables can be defined to configure the target

How Scratchbox works (3)

Toolchain binaries are executed through a wrapper

The `gcc` binary is a symlink to `sb_gcc_wrapper`, which runs the correct compiler depending on the target configuration.

Build systems think that they are building natively.

Outside of Scratchbox, the toolchain can be used in a normal way (`ARCH-linux-gcc`, etc.)

How Scratchbox works (4)

Host tools take precedence over target binaries

Host tools are hardwired to use libraries in `/scratchbox/host_shared/`

`PATH` is set so that host binaries are used in preference over target binaries, but it is not enough for absolute paths.

Scratchbox uses a technique called *binary redirection.*

Using `LD_PRELOAD`, some libc functions are overridden

- `exec()` and friends
- `uname()` so that the target architecture is correctly returned.
- etc.

How Scratchbox works (5)

CPU transparency

Execute target binaries transparently on the host

Uses the kernel `binfmt_misc` facility to run an interpreter when a target binary is run.

- See `/proc/sys/fs/binfmt_misc/` for its configuration.

The interpreter can then

- Use `QEMU` user emulation to run the binary
- Use `sbrsh` to execute the binary directly on the target device using a network connection.

Compiling BusyBox

Inside the chroot:

Download BusyBox:

```
wget http://busybox.net/downloads/busybox-1.11.1.tar.bz2
```

Extract its sources:

```
tar jxvf busybox-1.11.1.tar.bz2
```

Get a basic configuration for BusyBox:

```
cd busybox-1.11.1
wget https://bootlin.com/pub/conferences/2008/ols/busybox-1.11.1.config
cp busybox-1.11.1.config .config
```

Compile BusyBox:

```
make
make install
```

TIP: configuring BusyBox

BusyBox uses the same build system as the Linux kernel

You can run `make xconfig`
or `make menuconfig` to configure it.

However, this can't be done in Scratchbox, as the development environment neither contains the `Qt` headers nor the `curses` ones.

You can still do this outside the chroot!

```
cd /scratchbox/users/mike/home/mike
cd busybox-1.11.1
make xconfig
```

Kernel compiling environment (1)

Compile the kernel outside of Scratchbox!

Extract the Linux sources

```
tar jxf linux-2.6.24.tar.bz2
cd linux-2.6.24
```

Set the target architecture in Makefile:

```
ARCH ?= arm
```

Kernel compiling environment (2)

Find the location of the cross-compiler in Scratchbox:

```
dpkg -L scratchbox-toolchain-cs2005q3.2-glibc-arm
```

Result:

```
/scratchbox/compilers/cs2005q3.2-glibc-arm/bin/sbox-arm-linux-gcc
```

Specify the cross-compiler prefix in Makefile:

```
CROSS_COMPILE ?= sbox-arm-linux-
```

Specify the cross-compiler path in the environment:

```
export
PATH=/scratchbox/compilers/cs2005q3.2-glibc-arm/bin/:
$PATH
```

Kernel configuration (1)

Choose the standard configuration for ARM Versatile PB:

`ls arch/arm/configs` or `make help`

`make versatile_defconfig`

`make xconfig`

Remove unneeded features:

- Loadable module support (`MODULES`), Swapping (`SWAP`)
- Memory Technology Device support (`MTD`)
- I2C support (`I2C`), Sound (`SOUND`)
- USB support (`USB_SUPPORT`), MMC/SD card support (`MMC`)
- Filesystems: VFAT filesystem, Native Language support, Minix, cramfs, romfs, advanced partitions.

TIP: Use `[Ctrl] [F]` to look for kernel parameters

Kernel configuration (2)

Add NFS root filesystem support to your kernel (`ROOT_NFS`)

Add boot logo support (`LOGO`)

TIP: The configuration interface helps you to find missing dependencies if needed.
Requirement: `Option -> Show debug info`

If you have any doubt about your configuration, you could use ours: `linux-2.6.25.config`

Kernel compiling

```
make
```

TIP to compile faster:

```
make -j 4
```
(4 parallel compile jobs)

What else?

NFS environment

Make your target directory available through NFS

Install an NFS server:

```
apt-get install nfs-kernel-server
```

Add the below line to your `/etc/exports` file:

```
/scratchbox/users/mike/targets/mpl3x86
172.20.0.2(rw,no_root_squash,no_subtree_check)
```

Restart your NFS server:

```
/etc/init.d/nfs-kernel-server restart
```

Booting the emulated board

Download our run_qemu_nfs script from
https://bootlin.com/pub/conferences/2008/ols

Script contents:

```
sudo qemu-system-arm -M versatilepb
-net tap -net nic -m 64
-kernel linux-2.6.24/arch/arm/boot/zImage -
append "console=tty0 rw root=/dev/nfs
ip=172.20.0.2:172.20.0.1:172.20.0.1:255.255.
255.0:qemu:eth0
nfsroot=172.20.0.1:/scratchbox/users/mike/ta
rgets/mediaplayer"
```

First signs of life

```
Warning: unable to open an initial console
```

-> Create `/dev/console`:

```
cd /scratchbox/users/mike/targets/mediaplayer
sudo mkdir dev
sudo mknod dev/console c 5 1
```

TIP: run `ls -l /dev/console` on your workstation

```
Can't open /dev/tty2 (tty2 to tty5)
```

```
sudo mknod dev/tty2 c 4 2
sudo mknod dev/tty3 c 4 3
sudo mknod dev/tty4 c 4 4
sudo mknod dev/tty5 c 4 5
```

```
Please press Enter to activate this console
```

Bingo!

BusyBox init /etc/inittab

Documentation found in
http://busybox.net/downloads/BusyBox.html

By default, when no `/etc/inittab` is given,
it's equivalent to having the below settings:

```
::sysinit:/etc/init.d/rcS
::askfirst:/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/sbin/swapoff -a
::shutdown:/bin/umount -a -r
::restart:/sbin/init
```

Our /etc/inittab file:

We need to remove the `swapoff` command:

```
::sysinit:/etc/init.d/rcS
::askfirst:/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
::restart:/sbin/init
```

Download it from our website!

Our /etc/init.d/rcS

Useful to initialize the machine at each boot

`mkdir etc/init.d`

Create `etc/init.d/rcS`:

```
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
```

`chmod a+rx etc/init.d/rcS`

```
mkdir /proc
mkdir /sys
```

Bingo! `/proc` and `/sys` are not mounted at each boot, making basic commands work (`ps`, `mount`...)

Compiling zlib

Download zlib 1.2.3 from http://www.zlib.net/

```
./configure
  make
  make install
```

Compiling libpng

Download libpng 1.2.29 from http://www.libpng.org/

```
./configure
  make
  make install
```

Compile libjpeg 6b

Download libjpeg 6b from http://www.ijg.org

```
./configure
  make
  make install-lib
```

Compile freetype

Used for fonts

Download freetype 2.3.7 from http://freetype.org/

```
./configure
  make
  make install
```

Compile DirectFB

Download DirectFB 1.0.1 from http://directfb.org

Comment out line 1570 in `systems/fbdev/fbdev.c`:

```
//if (dfb_fbdev_compatible_format( var, 0, 5, 6, 5, 0, 11, 5, 0 ))
     return DSPF_RGB16;
```

Configure DirectFB and compile it

```
./configure --disable-x11 --with-gfxdrivers=none \
  --with-inputdrivers=keyboard,linuxinput,ps2mouse
make
make install
```

A few DirectFB tweaks

Let configuration and compiling tools know where the library is installed. In the scratchbox environment:

```
export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
```

Set the paths to newly installed stuff in `/usr/local`

Add the below line to `etc/init.d/rcS`:

```
export PATH=/usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin
export LD_LIBRARY_PATH=/usr/local/lib:/lib:/usr/lib
/bin/sh
```

Compile DirectFB examples

Download DirectFB-examples 1.0.0 (not 1.0.1!)
from http://directfb.org

```
./configure
  make
  make install
```

Start your target system again:
./run_qemu_nfs

Inside the virtual system: df_andi
Opening /dev/fb0 and /dev/fb/0 failed

Add /dev/fb0 and try again!
sudo mknod dev/fb0 c 29 0

More missing stuff

Inside the virtual system:

Opening /dev/fb0 and /dev/fb/0 failed

Add `/dev/fb0` and try again (find the numbers in the kernel sources: `Documentation/devices.txt`)

```
sudo mknod dev/fb0 c 29 0
```

Another complaint that `/dev/tty0` couldn't be opened

```
sudo mknod dev/tty0 c 4 0
```

Create the below directory, otherwise DirectFB programs will keep complaining that it is missing:

```
mkdir -p /usr/local/lib/directfb-1.0-0/gfxdrivers
```

Compiling xine-lib

Download xine-lib 1.1.14 from http://xinehq.de/

```
./configure
  make
  make install
```

Nice features in xine-lib

Contains most codecs it supports, including audio ones.
    No need to drag complex dependencies
    and compile lots of prerequisite libraries.

Supports DirectFB

Supports a huge number of formats!

Compile DirectFB-extra

Download DirectFB-extra 1.0.0 from http://directfb.org

```
./configure
  make
  make install
```

It contains a video player: `df_xine`

Create a sound device

In the target:

```
mknod dev/dsp c 14 3
```

More devices files would be required if we used the ALSA devices. With OSS compatibility offered by ALSA, `/dev/dsp` is sufficient for playback.

Play a video

Download our sample MPEG2 video from
https://bootlin.com/pub/conferences/2008/ols/videos/

Play this video:

```
df_xine building.avi
```

Press `[Esc]` to exit the player at the end.

Total filesystem size for now: 86 MB!
(video not counted)

Detect unused files (1)

Let's just keep the files need to play our MPEG2 video

We will get rid of

- Unused C and C++ libraries
- C headers, manual pages, documentation
- Unused image files (used by DirectFB examples)

Technique:

- Run a complete testsuite
- Find all the files that were accessed in the last 5 minutes.
- Copy only these files to a new root filesystem.
  - Copying selected files is easier than removing selected ones (loops to remove directories which are empty at the end)

Detect unused files (2)

Unfortunately, we couldn't get the NFS server (both kernel and userspace servers) to update the file access times.

We are going to boot on an ext2 filesystem

Copy the filesystem contents in an ext2 filesystem.

Create a 100 MB file and format it:

```
dd if=/dev/zero of=mediaplayer.ext2 bs=1M count=100
mkfs.ext2 -F mediaplayer.ext2
```

Now copy the root filesystem into it

```
mkdir /mnt/rootfs
sudo mount -o loop mediaplayer.ext2 /mnt/rootfs
rsync -a /scratchbox/users/mike/targets/mediaplayer/ /mnt/rootfs/
```

Detect unused files (3)

Download the `run_qemu_ext2` script
from https://bootlin.com/pub/conferences/2008/ols/

Boot the kernel and start the video.
Quit the video player (`[Esc]`).
Halt the virtual machine (`halt`), to make sure the root filesystem is unmonted.

Mount your filesystem image again

Find all the files accessed during the last 5 minutes:

```
find /mnt/rootfs -amin -5 > accessed_files
```

Detect unused files (4)

Issue: `qemu-system-arm` still living on Jan 1, 1970!

Consequence: the accessed files are dated Jan 1, 1970.

Not so much of a problem: let's keep only the files accessed more than 10000 days (approx. 27 years) ago!

```
find /mnt/rootfs/ -atime +10000 > accessed_files
```

Issues with atime information

First remove `/mnt/rootfs/` from all the lines in `accessed_files`

Files in `/dev/` went undetected.
- Is there a way to change this?
- Anyway, we manually add the `dev/` directory to `accessed_files`

Mount points also undetected: `/proc/` and `/sys/`
- Let's add them too!

Let's also keep BusyBox utilities, even if they were not used in the testsuite: `/bin/`, `/usr/bin/`, `/sbin/`. It's nice to still have commands like `ls`!

Copying only added files

Create a new (production) target directory:

```
mkdir /scratchbox/users/mike/targets/mediaplayerprod/
```

Copy `accessed_file` to the original directory:

```
cp access_file /scratchbox/users/mike/targets/mediaplayer
```

Download the `copy_accessed_files` script from our site:

```
#!bin/sh
source=/scratchbox/users/mike/targets/mediaplayer
target=/scratchbox/users/mike/targets/mediaplayerprod
for f in `cat access_file`;
do
  mkdir -p $target/`dirname $f`
  rsync -a $source/$f $target/$f
done
```

Run this script:

```
sudo ./copy_accessed_files
```

Test that your player still works

Copy `run_qemu_nfs` to `run_qemu_nfs_prod`

Modify it to use the `mediaplayerprod` directory

Also add `mediaplayerprod` to `/etc/exports`
and restart the nfs server:
`sudo /etc/init.d/nfs-kernel-server restart`

`./run_qemu_nfs_prod`

What's the new filesystem size?

Removing unneeded plugins

`usr/local/lib/xine/plugins/1.23` still contains plugins that we don't need. Apparently `xine-lib` reads them all, but we can do without some of them.

Do experiments removing some of these files (DVD playback, etc.) and keep only the files needed for MPEG2 audio playback, MPEG3 audio playback, FB video output, OSS sound output.

Result:

```
rm -rf xineplug_inp_dvd.so post/ xineplug_inp_cdda.so
xineplug_inp_cdda.so xineplug_inp_http.so
xineplug_inp_pvr.so xineplug_inp_v4l.so
xineplug_inp_vcdo.so xineplug_inp_vcd.so
xineplug_decode_sputext.so xineplug_decode_real.so
xineplug_decode_a52.so xineplug_inp_mms.so
xineplug_inp_dvb.so
```

Stripping executables and libraries

Removing unused debug info in these files

Stripping is architecture dependent: will have to use the `sbox-arm-linux-strip` command from the toolchain

To find whether a file needs stripping:

```
file usr/local/bin/df_xine
usr/local/bin/df_xine: ELF 32-bit LSB
executable, ARM, version 1 (SYSV), for
GNU/Linux 2.4.17, dynamically linked (uses
shared libs), not stripped
```

Find the files that need stripping

The easiest way is to use the `findstrip` utility:

```
sudo apt-get install perforate
```

```
cd /scratchbox/targets/users/mike/mediaplayerprod/
   for f in `findstrip`;
   do
     /scratchbox/compilers/cs2005q3.2-glibc-arm/bin/sbox-arm-linux-strip $f
   done
```

New size at the end?

TIP: can strip even further by using `sstrip` (super strip).

Unfortunately, `arm-linux-sstrip` is not available in the Scratchbox toolchain (can use Buildroot to make such a toolchain with `sstrip` support).

Boot on an initramfs root filesystem (1)

Usefulness: uses less storage size, because the filesystem is shipped in compressed form in the kernel image.

First add an `init` executable at the top of the target directory (required for initramfs booting):

```
cd /scratchbox/users/mike/targets/mediaplayerprod
ln -s sbin/init .
```

Move the video away in a `videos.ext2` filesystem image (the result can be downloaded from https://bootlin.com/pub/conferences/2008/ols/videos.ext2)

Create a video mount point for this filesystem:

```
sudo mknod dev/sda b 8 0
mkdir video
```

`etc/init.d/rcS`: mount this directory at startup time:

```
mount -t ext2 /dev/sda /video
```

Boot on an initramfs root filesystem (2)

Recompile your kernel with initramfs booting:

```
CONFIG_INITRAMFS_SOURCE="/scratchbox/users/mike/targets/mediaplayerprod"
```

Also disable networking to make the kernel lighter.

(can also download our `linux-2.6.25-initramfs.config` file)

Get our `run_qemu_initramfs` script

Copy `linux-2.6.25/arch/arm/boot/zImage` to `vmlinuz-arm-2.6.25`

```
./run_qemu_initramfs
```

Reduce the kernel size further by removing `printk` support as well as drivers that you may not need.

Size of kernel image (including the filesystem): 3.9 MB!!!!

Test the result by yourself!

Even if you didn't run the whole tutorial by yourself.

Go to https://bootlin.com/pub/conferences/2008/ols/

Download the following files:

`run_qemu`, `videos.ext2` and `vmlinuz-arm-2.6.25`

Run the demo:

```
chmod +x ./run_qemu
./run_qemu
```

Techniques to further reduce size:

Rebuild BusyBox with less features
(just the ones needed in the test suite)

`mklibs`: a tool to recompile shared libraries with only the symbols used in the executables in the filesystem.
However, not straightforward to use inside Scratchbox.

Encountered issues

NFS server not updating `atime` information.

`qemu-system-arm` still living on Jan 1, 1970!

DirectFB demo programs not working on Linux 2.6.26.

Scratchbox limitations

A big of black magic, a bit difficult to understand at first

CPU transparency with emulation may not support the exact code generated by your compiler, but just the instruction set supported by QEMU. Can use remote execution in this case.

Lack of reproducibility

- Everything is done by hand
- With tools like OpenEmbedded and Buildroot, you have a predictable way of generating a full system update.

References

Scratchbox
http://www.scratchbox.org

Scratchbox: Cross-compiling a Linux distribution
http://www.embedded-kernel-track.org/2005/scratchbox-fosdem2005.pdf
FOSDEM 2005, Brussels

Bringing Cross-Compiling to Debian
http://nchipin.kos.to/debconf-sbox2.pdf
Debconf 6, Mexico.

Book: Embedded Linux development with a hammer, a chainsaw and heavy explosives, by Rob Landley (Fabrice Bellard Editions).

What to remember

Multimedia system: need to use glibc
uClibc not fully ready for audio and video decoding.
Or perhaps multimedia applications are not tested enough with uClibc.

Building from scratch is the best way to get small and fast results

Good to have 2 filesystems: a development one and a production one generated from it (removing unused files, stripping binaries...)

What to remember (2)

Scratchbox toolchains can be used outside of SB, in particular to compile the kernel.

Booting on NFS is a very convenient way to develop your filesystem.

Thank you!

Bootlin services

Embedded Linux Training

Unix and GNU/Linux basics
Linux kernel and drivers development
Real-time Linux
uClinux
Development and profiling tools
Lightweight tools for embedded systems
Root filesystem creation
Audio and multimedia
System optimization

Custom Development

System integration
Embedded Linux demos and prototypes
System optimization
Linux kernel drivers
Application and interface development

Consulting

Help in decision making
System architecture
Identification of suitable technologies
Managing licensing requirements
System design and performance review

Technical Support

Development tool and application support
Issue investigation and solution follow-up with mainstream developers
Help getting started

https://bootlin.com